

Chapter 2

Single Species Gene Finding

A gene finding model usually consists of a main algorithm that serves as a kind of “umbrella” algorithm for a large number of rather complex submodels. The submodels represent various features of a gene, such as exons, introns, and splice site models. Each submodel scores the probability, or likelihood, that each given sequence region constitutes the corresponding gene feature, and then these scores are passed on up to the main algorithm. The main algorithm uses these scores as foundation for parsing the entire input sequence into complete gene structures that adhere to the biological rules implemented in the model. This chapter covers a range of mathematical models commonly used as main algorithms in single species gene finding. Similar models used for comparative gene finding are presented in Chap. 4, while the various kinds of submodels used for specific gene features are presented in Chap. 5.

2.1 Hidden Markov Models (HMMs)

One reason for the popularity of Markov models is that, due to their flexibility, most processes can be approximated by a Markov chain. Markov theory is a well-studied technique and includes a machinery of powerful algorithms to be used in data analysis. The word “chain” may indicate that the random process generates a discrete chain of events, but a Markov chain can evolve both in discrete and continuous time, and have either a discrete or continuous state space. The Markov chains we will consider here, however, will all have a discrete, finite state space. Moreover, since most Markov models presented in this book will be of discrete-time type, this will be our main focus in this section. But since continuous-time Markov models will be mentioned in connection with substitution models in pairwise alignments (Sect. 3.1), we give a brief account of that theory as well. For more details, see [16].

A powerful extension of the Markov theory are the hidden Markov models (HMMs). HMMs were originally developed for speech recognition, with one of the best references being the introduction by Rabiner [30]. Nowadays, HMMs have become an integral part of bioinformatics with applications including modeling the

periodic patterns occurring in a gene, the sequence alignment pairing of nucleotides and amino acids, and the point mutation process of sequence evolution. For a deeper and more general description of HMMs applied to bioinformatics, see [18].

2.1.1 Markov Chains

A *random process*, also called a *stochastic process*, is basically the evolution in time of some random variable. For instance, the mutation process in evolution can be seen as a random process. What makes the process random is that it jumps randomly between different *states* in a *state space*. A *Markov chain* is a random process which is “memoryless” in the sense that the next jump only depend on the current state, and not the past of the process. This property, called the *Markov property*, is described in more detail below.

We typically write a random process as a sequence of indexed random variables (X_1, X_2, \dots) , where X_t is the *state* of the random process at time index $t \in T$. If the index set T is finite or countable, such as the integers, we call the process a *discrete-time* random process. If the indices come from a continuous set, such as an interval on the real line, the process is a *continuous-time* random process. The process evolves by jumping between the states in a state space S . Just as with the time index, the state space can be finite, countable, or continuous. Note that there is no initial assumption about independence between the random variables in the process. Different settings on the index set T , the state space S , and various interdependencies between the indices in the process make up a wide variety of random processes. Markov chains are thus a special case of this.

Discrete-Time Markov Chains

Consider a physical process that at any instant in time will reside in one of N possible states, call them $S = \{s_1, \dots, s_N\}$. Assume that the process jumps between states at discrete time points $t = 1, 2, 3, \dots$, and let X_t denote the state at time t . Using the definition of conditional probabilities, the probability of any sequence of random variables (X_1, \dots, X_T) can be for states $i_1, \dots, i_T \in S$ be decomposed as

$$\begin{aligned}
 \mathbf{P}(X_1 = i_1, \dots, X_T = i_T) &= \\
 &= \mathbf{P}(X_T = i_T | X_{T-1} = i_{T-1}, X_{T-2} = i_{T-2}, \dots, X_1 = i_1) \\
 &\quad \cdot \mathbf{P}(X_{T-1} = i_{T-1} | X_{T-2} = i_{T-2}, \dots, X_1 = i_1) \\
 &\quad \dots \\
 &\quad \cdot \mathbf{P}(X_2 = i_2 | X_1 = i_1) \\
 &\quad \cdot \mathbf{P}(X_1 = i_1).
 \end{aligned} \tag{2.1}$$

The conditional probabilities in (2.1) represent the probabilities to jump from state X_t to X_{t+1} , possibly conditioning on all the past states. What characterizes a Markov

chain, however, is that it is “memoryless”. That is, given the current state, the future and the past of the process are independent. This feature, called the *Markov property*, can be formalized as follows:

Definition 2.1 The process (X_1, X_2, \dots) is a *Markov chain* if it for $i, j, i_1, \dots, i_{t-2} \in S$ satisfies the *Markov property*

$$\mathbf{P}(X_t = j | X_{t-1} = i, X_{t-2} = i_{t-2}, \dots, X_1 = i_1) = \mathbf{P}(X_t = j | X_{t-1} = i). \quad (2.2)$$

The probability of a sequence (X_1, \dots, X_T) , generated by a Markov chain, thus becomes

$$\mathbf{P}(X_1 = i_1, \dots, X_T = i_T) = \mathbf{P}(X_1 = i_1) \prod_{t=2}^T \mathbf{P}(X_t = i_t | X_{t-1} = i_{t-1}). \quad (2.3)$$

Definition 2.2 The probability of the first state X_1 is determined by the *initial distribution* $\pi = \{\pi_1, \dots, \pi_N\}$, where

$$\pi_i = \mathbf{P}(X_1 = i), \quad i \in S \quad \sum_{i=1}^N \pi_i = 1. \quad (2.4)$$

The chain proceeds according to the *transition matrix* $\mathbf{A} = (a_{ij})_{i,j \in S}$, which is an $(N \times N)$ -matrix consisting of *transition probabilities*

$$a_{ij} = \mathbf{P}(X_t = j | X_{t-1} = i), \quad i, j \in S. \quad (2.5)$$

The transition matrix is *stochastic*, meaning that all entries are nonnegative $a_{ij} \geq 0$, and each row adds up to one

$$\sum_{j=1}^N a_{ij} = 1. \quad (2.6)$$

A Markov chain with transition probabilities as in (2.5) is said to be of *first order*, due to its dependency on only the previous state. This can be generalized, however, such that each state depends on several of the previous states. For instance, a *second-order* Markov chain depends on the previous two states, and has transition probabilities on the form

$$a_{ijk}^{(2)} = \mathbf{P}(X_t = k | X_{t-1} = j, X_{t-2} = i), \quad i, j, k \in S. \quad (2.7)$$

Just as in the first-order case, the transition probabilities are nonnegative and the rows sum up to one

$$\sum_{k=1}^n a_{ijk} = 1. \quad (2.8)$$

To generalize further, a k th-order Markov chain depends on the k previous states and is defined as

$$a_{\mathbf{i}j}^{(k)} = \mathbf{P}(X_t = j | X_{t-1} = i_1, X_{t-2} = i_2, \dots, X_{t-k} = i_k). \quad (2.9)$$

where $\mathbf{i} = (i_1, \dots, i_k)$ and $i_1, \dots, i_k, j \in S$. The sequence X_{t-1}, \dots, X_{t-k} is sometimes referred to as the *context* of X_t . Note that while a first-order Markov chain of N states has an $(N \times N)$ transition matrix, a k th-order Markov chain has an $(N^k \times N)$ transition matrix, with one row for each of the N^k possible contexts. A Markov chain of zeroth-order has no context and only consists of independent state frequencies π_i .

Example 2.1 Building a Markov chain from data

As a toy-example, consider a machine that generates a DNA sequence according to a first-order Markov chain. The state space $S = \{A, C, G, T\}$ is illustrated in Fig. 2.1 where the states are represented as circles, and the arrows between them represent the transitions. The machine starts in some state according to the initial distribution $\pi = \{\pi_A, \pi_C, \pi_G, \pi_T\}$, generates the corresponding DNA base, and then jumps to a new state according to the transition probabilities a_{ij} , $i, j \in S$.

Assume that the machine generated a DNA sequence of length $T = 24$,

CCTCCCGGACCCTGGGCTCGGGAC

By noting that

$$a_{ij} = \mathbf{P}(X_t = j | X_{t-1} = i) = \frac{\mathbf{P}(X_{t-1} = i, X_t = j)}{\mathbf{P}(X_{t-1} = i)}, \quad (2.10)$$

we can deduce that a first-order Markov chain on $S = \{A, C, G, T\}$ models dinucleotide frequencies $\{AA, AC, AG, AT, \dots, TG, TT\}$. Thus, by counting the number of times nucleotide i is followed by nucleotide j for all $i, j \in S$ in our sequence, we can estimate the model parameters by

$$\hat{\pi}_i = \frac{c_i}{\sum_k c_k} \quad \hat{a}_{ij} = \frac{c_{ij}}{c_i}, \quad (2.11)$$

Fig. 2.1 The state space of a DNA sequence generating machine

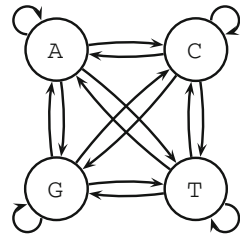


Table 2.1 The frequency counts and estimated model parameters

	c_{ij}	To (j)				c_i
		A	C	G	T	
From (i)	A	0	2	0	0	2
	C	0	5	2	3	11
	G	2	1	5	0	8
	T	0	2	1	0	3

where c_i is the frequency count of the single residue i , and c_{ij} is the frequency count of the dinucleotide $\{ij\}$ for $i, j \in S$. The dinucleotide frequency counts of the observed sequence above are shown in Table 2.1. Note that since the sequence ends with a C, the C-row will not add up.

The estimated model parameters thus become

$$\hat{\pi} = (0.08, 0.46, 0.33, 0.13), \quad \hat{A} = \begin{pmatrix} 0.00 & 1.00 & 0.00 & 0.00 \\ 0.00 & 0.56 & 0.22 & 0.22 \\ 0.25 & 0.12 & 0.62 & 0.00 \\ 0.00 & 0.67 & 0.33 & 0.00 \end{pmatrix}. \quad (2.12)$$

Using the estimated model we can predict the next nucleotide in the sequence. For instance, given that $X_T = C$, there is an estimated 56 % chance that the next symbol is ‘C’. Similarly, an entire new sequence can be scored based on this model. For instance

$$\begin{aligned} p(\text{CCTG}) &= \\ &= \mathbf{P}(X_1 = C) \mathbf{P}(X_2 = C | X_1 = C) \mathbf{P}(X_3 = T | X_2 = C) \mathbf{P}(X_4 = G | X_3 = T) \\ &= \pi_C \cdot a_{CC} \cdot a_{CT} \cdot a_{TG} \\ &= 0.0167. \end{aligned} \quad (2.13)$$

Such scoring can be used to examine how characteristic a new sequence is to the given model and, for instance, to distinguish a coding sequence from a noncoding sequence. This is discussed further in Example 2.2. Probabilities of indices at longer distances in the process can be determined similarly by using

$$\begin{aligned} \mathbf{P}(X_{T+2} = C | X_T = C) &= \\ &= \sum_{k \in S} \mathbf{P}(X_{T+2} = C | X_{T+1} = k) \mathbf{P}(X_{T+1} = k | X_T = C) \\ &= \sum_{k \in S} a_{Ck} a_{kC} \\ &= 0.00 \cdot 0.00 + 0.56 \cdot 0.56 + 0.22 \cdot 0.12 + 0.22 \cdot 0.67 \\ &= 0.49. \end{aligned} \quad (2.14)$$

□

In general, the n -step transition matrix $\mathbf{A}^{(n)} = (a_{ij}(n))_{i,j \in S}$, corresponding to the n th power of \mathbf{A} represents the transitions from i to j in n steps, where

$$a_{ij}(n) = \sum_{k \in S} a_{kj} a_{ik}(n-1). \quad (2.15)$$

The previous example is an example of a Markov chain that does not vary over time. The transition probabilities are the same regardless of where we are in the sequence, that is, X_t is independent of how long the process has run.

Definition 2.3 A Markov chain is said to be *time-homogeneous* (or just *homogeneous*) if the following condition holds

$$\mathbf{P}(X_t = j | X_{t-1} = i) = \mathbf{P}(X_h = j | X_{h-1} = i) \text{ for } t \neq h. \quad (2.16)$$

and *inhomogeneous* otherwise.

Example 2.2 Markov chain classification of *E. coli*

The single most powerful method of discriminating between coding and noncoding sequences is to use the statistical differences in sequence patterns. We use the same model as in Example 2.1 with the state space shown in Fig. 2.1.

Assume that we want to use this model to discriminate between coding and noncoding sequences in the bacteria *Escherichia coli*. First, we use a training set of known coding and noncoding sequences to estimate the model parameters. Table 2.2 shows the dinucleotide frequencies and base counts for coding and noncoding sequences in the *E. coli* strain O157:H7 [26].

The probability of a new sequence (X_1, \dots, X_T) is given by

$$\mathbf{P}(X_1 = i_1, \dots, X_T = i_T) = \pi_{i_1} \prod_{t=1}^{T-1} a_{i_t, i_{t+1}}. \quad (2.17)$$

The probabilities π and a_{ij} can be estimated as in (2.11) using the frequency counts in Table 2.2. Now, in order to test if the given sequence is coding or not, we can

Table 2.2 The dinucleotide frequency counts in *E. coli* O157:H7 coding and noncoding sequences

		Coding to (j)				c_i	c_{ij}	Noncoding to (j)				c_i
		A	C	G	T			A	C	G	T	
From (i)	A	0.310	0.224	0.199	0.268	0.245	A	0.321	0.204	0.200	0.275	0.262
	C	0.251	0.215	0.313	0.221	0.243	C	0.282	0.233	0.269	0.215	0.239
	G	0.236	0.308	0.249	0.207	0.273	G	0.236	0.305	0.235	0.225	0.240
	T	0.178	0.217	0.338	0.267	0.239	T	0.207	0.219	0.259	0.314	0.259

calculate the probability in (2.17) for two different models, coding and noncoding, using the corresponding frequency counts in Table 2.2.

The two probabilities are then compared using a *likelihood-ratio test*, or a *log-odds ratio* decision rule

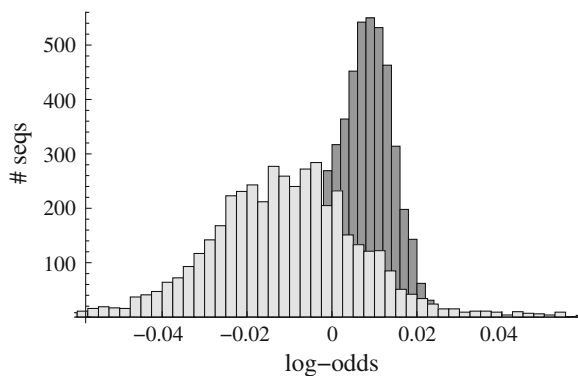
$$S(X) = \log \frac{\mathbf{P}_C(X_1 = i_1, \dots, X_T = i_T)}{\mathbf{P}_N(X_1 = i_1, \dots, X_T = i_T)} \begin{cases} > \eta \Rightarrow \text{coding,} \\ < \eta \Rightarrow \text{noncoding,} \end{cases} \quad (2.18)$$

where \mathbf{P}_C is the probability when the parameters have been estimated using coding frequencies, and \mathbf{P}_N the corresponding probability using noncoding frequencies. The threshold value η is chosen to satisfy a desired significance level (e.g., $\alpha = 0.05$). It is customary in sequence analysis to use logarithms of the probabilities to prevent the probabilities of long sequences from falling below computer precision and become numerically unstable. As a positive side effect products are transformed into sums, which results in a more efficient computation.

The decision rule in (2.18) is of course very crude. The (length-normalized) log-odds scores of coding versus noncoding sequences in *E. coli* are illustrated in Fig. 2.2. We see that while the peaks of the two distributions are separated, which is necessary in order to discriminate between the models, the overlap is significant, making it hard to separate coding sequence from noncoding sequence based on this score alone. Several improvements to the decision rule would be possible already at this early stage. For one thing, a more sensitive approach would utilize the fact that coding sequences are organized in codons. Thus, a quick fix would be to upgrade the above model to a second-order Markov chain, using transition probabilities trained on triplets rather than on dinucleotides.

Moreover, it is a known fact that the probability of a triplet in a coding region depends on its position with respect to the reading frame of the sequence. Thus, an even better model would be an *inhomogeneous* second-order Markov chain. We would then train four different Markov chains, one for each coding frame and one for noncoding sequences. An example of this is given in Sect. 5.3.5. \square

Fig. 2.2 Distribution of log-odds ratio scores of length-normalized coding (dark gray) and noncoding (light gray) sequences in *E. coli*



Example 2.2 illustrates a strategy for classifying an unknown DNA sequence into coding or noncoding. What we really want, however, is to extract one or several coding regions from a longer sequence consisting of intermediate stretches of non-coding regions. Furthermore, in organisms where splicing may occur, we would like to combine the coding regions into complete gene structures if possible. The Hidden Markov Model (HMM) theory, presented in the next section, provides a suitable framework for this.

Stationarity and Reversibility

An important question of Markov theory is the limit behavior of the chain. What are the characteristics of a process that has run for a long time? Although the chain itself will never converge toward a specific state (unless $a_{ii} = 1$ for some $i \in S$), the state distribution may still stabilize. More specifically, what is the probability of state i occurring when time goes to infinity? Will the behavior of the chain converge? We call a distribution over the state space $\tau = \{\tau_1, \dots, \tau_N\}$ a *stationary distribution* if

- (a) $\tau_i \geq 0$ for all i , and $\sum_i \tau_i = 1$.
- (b) $\tau = \tau A$, which is to say that $\tau_j = \sum_{i=1}^N \tau_i a_{ij}$ for all $j \in S$.

The stationary distribution is sometimes called the *invariant*, *equilibrium*, or *steady state* distribution. The concept of stationarity is central in Markov theory, since convergence toward a stationary distribution somehow guarantees that the process is well-behaved in some respect. The stationary distribution may or may not exist, and even if it exists, the process may or may not ever reach it. We need a couple of more concepts before we can state the requirement for a stationary distribution to exist.

Definition 2.4 We say that state $i \in S$ *communicates* with state $j \in S$, writing $i \rightarrow j$, if, starting from i , the probability of ever reaching state j is positive. That is, if $a_{ij}(m) > 0$ for some $m \geq 0$. We say that i and j *intercommunicate* if $i \rightarrow j$ and $j \rightarrow i$. Furthermore, we say that the state space is *irreducible* if all its states intercommunicate.

Definition 2.5 We call a state *recurrent* if the probability of eventually returning is 1. That is, if

$$\mathbf{P}(X_t = i \text{ for some } t > 1 | X_1 = i) = 1. \quad (2.19)$$

If this probability is strictly less than 1, we say that the state is *transient*.

Note that although we will return to a recurrent state with probability one, the expected time of return may very well be infinite. To make sure the expected return time is finite, we need an additional restriction on the recurrence. Starting in state $X_1 = i$, let T_i be the time until the first return to state i

$$T_i = \min\{t > 1 : X_t = i | X_1 = i\}. \quad (2.20)$$

Definition 2.6 We say that a recurrent state is *positive* if the expected time of return is finite $E[T_i] < \infty$.

Now we can state the following important result:

Theorem 2.1 *An irreducible chain has a stationary distribution τ if and only if all states are positive recurrent. In that case, τ is unique and is given by $\tau_i = 1/E[T_i]$.*

However, just because the stationary distribution exists, it is not guaranteed that the chain ever reaches it. For this we need an extra condition.

Definition 2.7 A state i is said to have *period* $d(i)$ if any return to the state must occur in multiples of $d(i)$ time steps. Formally, the period of state i is defined as

$$d(i) = \gcd\{n : a_{ii}(n) > 0\}, \quad (2.21)$$

where ‘gcd’ stands for the ‘greatest common divisor’. We say that state i is *periodic* if $d(i) > 1$ and *aperiodic* otherwise. That is to say that $a_{ii}(n) = 0$ unless n is a multiple of $d(i)$. Furthermore, a Markov chain is said to be aperiodic if at least one of its states is aperiodic.

Theorem 2.2 *If the chain is irreducible and aperiodic, then for all $i, j \in S$*

$$a_{ij}(n) \rightarrow \frac{1}{E[T_i]} \text{ as } n \rightarrow \infty. \quad (2.22)$$

Note that the limit in Eq. 2.22 is what gives the stationary distribution in Theorem 2.1. Thus, if the chain is irreducible and aperiodic with positive recurrent states, the transition probabilities converge to the stationary distribution.

Another useful property is *time reversibility*. Let X_1, \dots, X_T be an irreducible, positive recurrent Markov chain with initial probabilities π and transition matrix \mathbf{A} . Let Y_1, \dots, Y_T be the chain running in reverse, that is

$$Y_t = X_{T-t}. \quad (2.23)$$

Then Y is a Markov chain as well, with transition probabilities b_{ij} say.

Definition 2.8 We say that X is *time everisible* if $a_{ij} = b_{ij}$ for all $i, j \in S$.

Thus, since

$$\begin{aligned} b_{ij} &= \mathbf{P}(Y_t = j | Y_{t-1} = i) \\ &= \mathbf{P}(X_{T-t} = j | X_{T-(t-1)} = i) \\ &= \frac{\mathbf{P}(X_{T-(t-1)} = i | X_{T-t} = j) \mathbf{P}(X_{T-t} = j)}{\mathbf{P}(X_{T-(t-1)} = i)} \\ &= a_{ji} \frac{\pi_j}{\pi_i}, \end{aligned} \quad (2.24)$$

it holds that X is time reversible if and only if $\pi_i a_{ij} = \pi_j a_{ji}$.

Theorem 2.3 *For an irreducible chain, if there exists a distribution π such that*

$$0 \leq \pi_i \leq 1, \quad \sum_i \pi_i = 1, \quad \pi_i a_{ij} = \pi_j a_{ji} \text{ for all } i, j,$$

then the chain is time reversible, positive recurrent, and stationary with stationary distribution π .

The interpretation of time reversibility is that it is not possible to determine the direction of the process, or the order of states, just by observing the state sequence. This is a very useful property for substitution models in sequence alignment (see Sect. 3.1) as it allows us to model the distance between two evolutionary-related sequences by analyzing the process of evolving one into the other, rather than making inferences about the distance to some unknown common ancestor in between.

Continuous-Time Markov Chains

A continuous-time Markov chain is very similar to its discrete counterpart. It jumps between states in a state space $S = \{s_1, \dots, s_N\}$ and is parametrized by its initial distribution and transition probabilities. The main difference is that instead of making the jumps at discrete time points, the chain makes a transition after having spent a continuous amount of time in the state. The time spent in a state is called the *holding time*, or *waiting time*. The holding time in discrete-time chains is thus always equal to 1, while for continuous-time processes the holding time is a continuous random variable.

Let $\{X(t) : t \geq 0\}$ be a continuous random process, indexed by the positive real numbers, and with a discrete state space $S = \{s_1, \dots, s_N\}$. The process is Markov if it satisfies the *Markov property*, which for continuous-time processes translates to

$$\mathbf{P}(X(t_n) = j | X(t_0) = i_0, \dots, X(t_{n-1}) = i_{n-1}) = \mathbf{P}(X(t_n) = j | X(t_{n-1}) = i_{n-1}), \quad (2.25)$$

for a sequence of times $t_1 < t_2 < \dots < t_n$ and for all $j, i_0, i_1, \dots, i_{n-1} \in S$. Just as in the discrete case, the first state $X(t_0)$, ($t_0 = 0$), is given by an initial distribution $\pi = \{\pi_1, \dots, \pi_N\}$, but the transition probabilities now need to be parametrized by time as well. We denote the probability of making a transition from state i to state j between time points s and t , where $s < t$, as follows

$$a_{ij}(s, t) = \mathbf{P}(X(t) = j | X(s) = i), \quad s < t. \quad (2.26)$$

When the transition probabilities are independent of how long the process has run, we call the chain *time-homogeneous* (or just *homogeneous*).

That is, for a homogeneous Markov process it holds that

$$a_{ij}(s, t) = a_{ij}(0, t - s) \text{ for all } i, j, s < t. \quad (2.27)$$

Henceforth, we write $a_{ij}(t) = a_{ij}(0, t)$ for the transition probability of a homogeneous chain over time period t , and let $\mathbf{A}(t) = (a_{ij}(t))_{i,j \in S}$ denote the transition matrix for this time period. As in the discrete-time case the rows of the transition matrix sum to one

$$\sum_{j=1}^N a_{ij}(t) = 1, \quad (2.28)$$

and a time interval can be split up in smaller segments by

$$a_{ij}(s+t) = \sum_{k=1}^N a_{ik}(s) a_{kj}(t) = \sum_{k=1}^N a_{ik}(t) a_{kj}(s) \text{ if } s, t \geq 0. \quad (2.29)$$

If we assume that the transition probabilities $a_{ij}(t)$ are continuous functions of t , we can assume that for an infinitely small time interval “nothing happens.” That is, as $h \downarrow 0$

$$a_{ij}(h) \rightarrow \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j, \end{cases} \quad (2.30)$$

and the transition matrix reduces to the identity matrix

$$\mathbf{A}(t) \rightarrow \mathbb{I} \text{ as } t \downarrow 0. \quad (2.31)$$

A difficulty that arises with continuous-time Markov chains is that we no longer have a clear notion of the rates of change. In the discrete-time theory the transition probabilities both represent the changes over unit times, as well as the rates of change between states. In a continuous setting, however, a time interval can be divided into infinitely many subintervals, such that while the transition probability $a_{ij}(t)$ gives us the probability of changing from state i to state j in time t , it does not tell us how *many* changes that have occurred in between.

We need some notion of the “instantaneous” rates of change. That is, assuming that $X(t) = i$, we would like to know the behavior of the process in a small time interval $(t, t+h)$, where $h > 0$ is very close to 0. Various things may happen during that time, but for a small enough h the events reduce to one of the two possibilities:

- Nothing happened with probability $a_{ii}(h) + o(h)$, implying that the state is the same at time t as at $t+h$.
- The chain made a single move to a new state with probability $a_{ij}(h) + o(h)$.

The $o(h)$ (little-o) is an error term that accounts for any extra, unobserved, transitions during the time interval. The term $o(h)$ basically states that for small enough h the probability of any extra events becomes negligible, and the probability of a particular transition is approximately proportional to h .

That is, there exist constants $\{\mu_{ij} : i, j \in S\}$ such that

$$a_{ij}(h) \approx \begin{cases} \mu_{ij}h & \text{if } i \neq j, \\ 1 + \mu_{ii}h & \text{if } i = j. \end{cases} \quad (2.32)$$

The matrix $\mathbf{Q} = (\mu_{ij})_{i,j}$ is called the *transition rate matrix*, also known as the *generator* of the transition matrix $\mathbf{A}(t)$. Note that $\mu_{ij} \geq 0$ if $i \neq j$, and $\mu_{ii} \leq 0$. The elements μ_{ij} for $i \neq j$ models the rate at which the chain enters state j from i , while $-\mu_{ii}$ models the rate at which the chain leaves state i . Moreover, when the chain leaves state i (with rate $-\mu_{ii}$), it must enter one of the other states, giving

$$\mu_{ii} = -\sum_{j \neq i} \mu_{ij}, \quad (2.33)$$

with the result that the rows of \mathbf{Q} sum to 0. The relation between the rate matrix \mathbf{Q} and the transition matrix $\mathbf{A}(t)$ can be deduced using the *forward equations*

$$\frac{da_{ij}(t)}{dt} = \sum_{k \in S} a_{ik}(t)q_{kj}, \quad (2.34)$$

or, similarly, using the *backward equations*

$$\frac{da_{ij}(t)}{dt} = \sum_{k \in S} q_{ik}a_{kj}(t). \quad (2.35)$$

Subject to the boundary condition $\mathbf{A}(0) = \mathbb{I}$, where \mathbb{I} is the identity matrix, the forward and backward equations are given by

$$\mathbf{A}(t) = e^{\mathbf{Q}t} = \sum_{n=0}^{\infty} \frac{t^n}{n!} \mathbf{Q}^n, \quad (2.36)$$

where \mathbf{Q}^n is the n th power of \mathbf{Q} . The properties of the transition rate matrix can be summarized as follows:

- The non-diagonal elements q_{ij} correspond to the probability per unit time of jumping from state i to state j .
- The row sums of the non-diagonal elements $q_i = -q_{ii}$ correspond to the total transition rate out of state i .
- The total transition rate q_i is also the rate at which the time to the next jump decreases. That is, the holding time of state i is exponentially distributed with parameter q_i .
- The number of jumps in a time interval is Poisson distributed with parameter q_i .

The transitions of a continuous-time Markov process can be viewed as an embedded discrete-time Markov chain, also known as the *jump process*.

The transition probability a_{ij} of the jump process, is the conditional probability of jumping from state i to state j , given that a transition occurs, and is given by

$$a_{ij} = \begin{cases} \frac{q_{ij}}{q_i} & \text{if } i \neq j, \\ 0 & \text{if } i = j. \end{cases} \quad (2.37)$$

Analogously to the discrete case, a distribution $\boldsymbol{\tau} = \{\tau_1, \dots, \tau_N\}$ on the state space is a *stationary distribution* if $\tau_i \geq 0$, $\sum_i \tau_i = 1$, and $\boldsymbol{\tau} = \boldsymbol{\tau}\mathbf{A}(t)$ for all $t \geq 0$. In terms of the rate matrix, $\boldsymbol{\tau}$ is a stationary distribution if and only if

$$\boldsymbol{\tau}\mathbf{Q} = \mathbf{0}. \quad (2.38)$$

Theorem 2.4 *Let X be an irreducible Markov chain with transition matrix $\mathbf{A}(t)$.*

- (a) *If there exists a stationary distribution $\boldsymbol{\tau}$, it is unique and $a_{ij}(t) \rightarrow \tau_j$ as $t \rightarrow \infty$.*
- (b) *If there is no stationary distribution then $a_{ij}(t) \rightarrow 0$ as $t \rightarrow \infty$.*

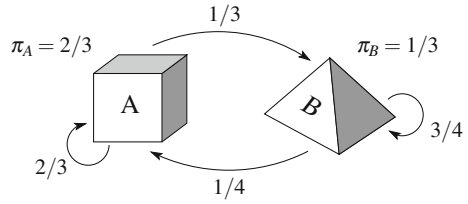
2.1.2 Hidden Markov Models

While the observed output in a standard Markov model is simply the sequence of states, a *hidden* Markov model (HMM) is comprised of two interrelated random processes, a hidden process and an observed process. The hidden process is a Markov chain jumping between states as before, but it can only be observed via the observed process. The observed process generates output through random functions associated with the underlying hidden states, and is generally not Markov. In other words, given the current state the hidden process is independent of the observed process. The observed process, however, typically depends both on its previous outputs and on the hidden process. For our purposes we only need to treat HMMs with a finite state space and a discrete, finite-valued observed process producing a finite output sequence, but it may be noted that the theory is applicable to more general situations.

Example 2.3 A simple HMM

Assume we have two dice, A and B . Die A has six sides and generates numbers between 1 and 6, while die B only has four sides and generates numbers between 1 and 4 (Fig. 2.3). Assume that we roll the dice, one at a time, and switch between the dice according to a Markov chain. The state space of the Markov chain is thus $S = \{A, B\}$, and the observed outputs are the numbers we produce by rolling the die. Die A emits each number with probability $1/6$, and die B emits each number with probability $1/4$. We generate numbers from this model as follows:

Fig. 2.3 A two-state HMM, where the hidden states are the dice, and the observed outputs are the roll outcomes



1. Choose initial die according to distribution $\{\pi_A, \pi_B\} = \{2/3, 1/3\}$.
2. Roll the die and observe the outcome.
3. Choose next die according to the transition probabilities a_{ij} , where i is the row and j the column index in the table below.

a_{ij}	A	B
A	2/3	1/3
B	1/4	3/4

4. Continue from 2.

Assume now that we only know the observed sequence of numbers, and know nothing about in which sequence the dice were rolled. Thus, the die sequence (state sequence) is *hidden* from us, and the die numbers are our observed sequence generated through random functions depending on the hidden state. The HMM algorithms can help us determine the most likely state sequence for the observed sequence, given our model. \square

We let $\{X_t\}_{t=1}^T$ denote the Markov process as before with state space $S = \{s_1, \dots, s_N\}$, initial probabilities $\pi = \{\pi_1, \dots, \pi_N\}$, and transition probabilities a_{ij} , $i, j \in S$. At each step t , the process emits an observation Y_t , where $\{Y_t\}_{t=1}^T$ denotes the observed process, taking values in some symbols set $V = \{v_1, \dots, v_M\}$. Each variable Y_t depends on the current (hidden) state X_t , and possibly on the previous outputs Y_1, \dots, Y_{t-1} . For simplicity we will use the shorthand $Y_a^b = Y_a, \dots, Y_b$ for a subsequence between time indices a and b . We denote the *emission distribution* of Y as

$$b_j(Y_t|Y_1^{t-1}) = \mathbf{P}(Y_t|Y_1^{t-1}, X_t = j). \quad (2.39)$$

To summarize, our HMM is characterized by the state space S , the emission alphabet V and the initial, transition and emission probabilities $\{\pi_i, a_{ij}, b_j : i, j \in S\}$.

One way to more easily understand the procedure of a Markov model is to view it as a “sequence generating machine”, by which the observed sequence could be generated as in Algorithm 1.

Algorithm 1 Generating output from a standard HMM

```

 $t = 1$ 
Choose  $X_t$  according to  $\pi$ 
while  $t < T$  do
    Emit  $Y_t$  according to  $b_{X_t}(Y_t|Y_1^{t-1})$ 
    Jump to state  $X_{t+1}$  according to  $a_{X_t, X_{t+1}}$ 
     $t = t + 1$ 
end while

```

The joint probability of the hidden and the observed process is determined by noting the following:

$$\begin{aligned}
 \mathbf{P}(X_t = j, Y_t | X_{t-1} = i, X_1^{t-2}, Y_1^{t-1}) &= \\
 &= \mathbf{P}(X_t = j | X_{t-1} = i) \mathbf{P}(Y_t | X_t, Y_1^{t-1}) \\
 &= a_{ij} b_j(Y_t | Y_1^{t-1}).
 \end{aligned} \tag{2.40}$$

Using the same notation as Rabiner in [30], we let $\theta = \{\pi, A, B\}$ denote the model, where π is the initial distribution, and A and B represent the transition and emission probabilities, respectively. Then the joint probability of the entire hidden and observed sequence, under the model can be written as

$$\mathbf{P}(X_1^T, Y_1^T | \theta) = \pi_{X_1} b_{X_1}(Y_1) \prod_{t=2}^T a_{X_{t-1}, X_t} b_{X_t}(Y_t | Y_1^{t-1}). \tag{2.41}$$

In what follows, while always conditioning on the model, we omit θ in the notation.

Thus far we have described the model, generating the hidden state sequence, that is underlying our observations. In gene finding, or in any other situation of classification, we are sitting at the other end. Typically, we are faced with an observed sequence that we would like to assign state labels to. In other words, we would like to classify, or *parse*, the observed sequence. In order to do so in the framework of HMMs, we need means to:

1. Estimate the parameters of the model.
2. Validate the model.
3. Use the model as a predictive tool.

In the first step, called the *training* step, we build our model by estimating its parameters from a set of training data. That is, we use a set of sequences that are representative for the patterns we are looking for, and where we know the state labels for each observed symbol. The second step, called the *evaluation* step, is a check that our model is a reasonable representation of reality. In this step we calculate the probability that the observed data was produced by our model.

The main difficulty, when going from a standard Markov chain to HMMs, is that there is no unique correspondence between the state sequence and the observed sequence. An observed sequence could be achieved by many different state

sequences, or many different *paths* through the state space. The goal of the third step, called the *parsing* step, is to determine the most likely state sequence that could have generated the observed sequence. We say that we *parse*, *classify*, *annotate*, or *decode* the observed sequence by attaching state labels to each observed symbol. The resulting state sequence then corresponds to the most probable path through the model.

By means of *dynamic programming*, described below, we can efficiently solve these problems. The corresponding HMM algorithms, utilizing the dynamic programming method, are called the *forward algorithm*, the *backward algorithm*, and the *Viterbi algorithm*. The evaluation and the decoding steps are solved directly using these algorithms. The solution to the training problem is more complicated and involves using a variant of the *expectation–maximization* (EM) algorithm called the *Baum–Welch algorithm*, described in Sect. 6.5.

2.1.3 Dynamic Programming

Many optimization problems have a recursive structure, or an *optimal substructure*, where the optimal solution can be divided into subproblems, which themselves have optimal solutions. Example 2.4 illustrates the concept of breaking a recursive structure into substructures.

Example 2.4 Fibonacci numbers

As an example of a recursive structure, consider the Fibonacci numbers, where each subsequent number in the series is the sum of the previous two numbers:

$$f(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ f(n-1) + f(n-2) & \text{if } n > 1. \end{cases} \quad (2.42)$$

Although not an optimization problem, it illustrates how redundant a naive implementation of $f(n)$ would be. If we, for instance, were to compute $f(5)$, using a direct (top-down) approach, we would call $f(2)$ three times, and $f(3)$ two times (see Fig. 2.4), and the number of computations needed to calculate $f(n)$ would grow exponentially with n . Such a problem, where the recursive solution contains a relatively small number of distinct subproblems repeated many times, is said to have *overlapping subproblems*. By representing each subproblem by one node only, we get a *directed acyclic graph* (DAG) (see Fig. 2.5), instead of the much redundant tree representation. Instead of having an exponentially growing number of computations, the problem grows linearly in n , since we only have to calculate each $f(n)$ once. \square

An efficient solution to problems such as that in Example 2.4 is the *dynamic programming* algorithm, which has attained a central role in computational sequence analysis [13]. Dynamic programming is a general recursive decomposition technique

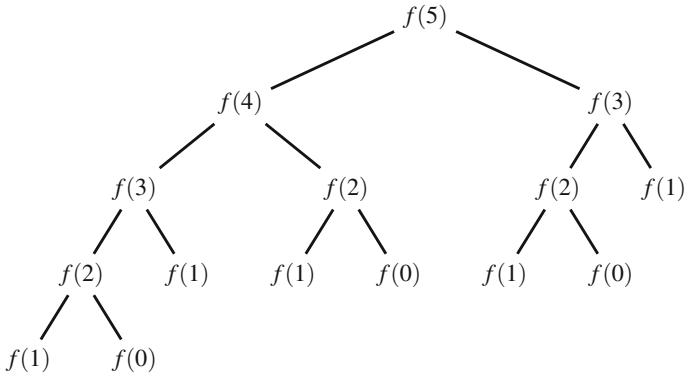
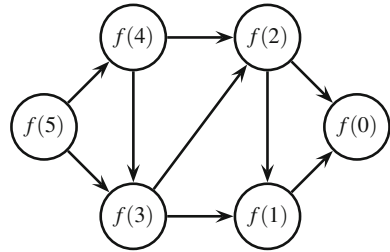


Fig. 2.4 A recursive tree illustrating the sub-calculations needed to determine $f(5)$

Fig. 2.5 A directed acyclic graph of the computation of the Fibonacci number $f(5)$



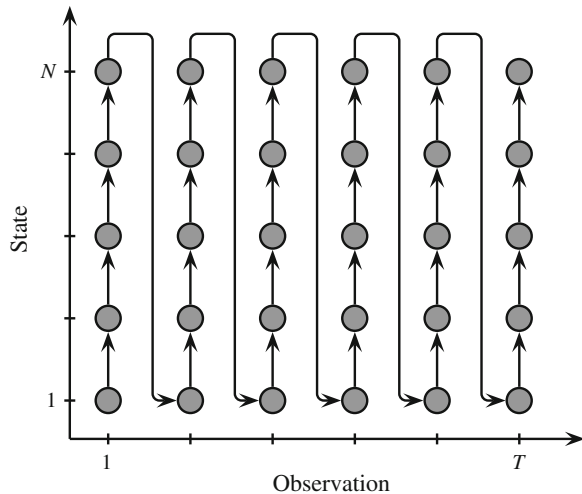
for global optimization problems, exhibiting the properties of optimal substructures and overlapping subproblems. The word ‘programming’ does not refer to the process of coding up a computer program for the purpose, but to the tabular mode of the computation. The trick used in dynamic programming is to store (or cache) and reuse the solutions to the subproblems, an approach called *memoization* (not memorization) in computing. The standard dynamic programming approach has three components:

1. The *recurrence relation*.
2. The *tabular computation*.
3. The *traceback*.

In the recurrence relation we establish the recursive relationships between the variables, such as in (2.42). In the tabular computation the calculations are organized in a table that is filled in one column at a time (see Fig. 2.6). There are in general two approaches to do this:

- In a *top-down approach* the problem is broken down into subproblems, which are calculated the first time they are called and then stored for further calls. This approach combines recursion and memoization. Figure 2.4 illustrates a top-down calculation of the Fibonacci algorithm. The contribution of dynamic programming is that the calculation of each subproblem is stored and a map function is used to keep track of which subproblems already have been calculated.

Fig. 2.6 The tabular computation goes through the table column by column



- In a *bottom-up approach* all subproblems are calculated and stored in advance. This is more efficient than the previous, but is less intuitive as it may be difficult in certain applications to figure out all subproblems needed for the calculation in advance. A bottom-down calculation of $f(5)$ would simply calculate all Fibonacci numbers subsequently, $f(0)$, $f(1)$, $f(2)$, $f(3)$, $f(4)$, $f(5)$.

The bottom-up approach is what is commonly used in HMM algorithms and sequence alignment, and is what we will consider from now on. Once the table of subproblems have been filled (bottom-up), we traceback through the table to obtain the optimal solution. In the Fibonacci example, the calculation is finished already in the tabular calculation, but in other situations such as in sequence alignment we still need to figure out the optimal solution to the global problem using the table of subproblem solutions. The easiest way to facilitate the traceback is to store pointers during the tabular computation from each cell in the table to the optimal previous position. These pointers are then followed in the traceback to determine the optimal *path* through the table. In the following sections we will show how dynamic programming is employed in HMMs.

2.1.3.1 Silent Begin and End States

Before we proceed to describe the HMM algorithms, we need to explain the notion of *silent states*. A silent state is a state with no output. Since the first state of a Markov chain follows a special initial distribution, adding a silent begin state X_0 to the model will simplify the formula in (2.41)

$$\mathbf{P}(X_1^T, Y_1^T) = \prod_{t=1}^T a_{X_{t-1}, X_t} b_{X_t}(Y_t | Y_1^{t-1}), \quad (2.43)$$

where now

$$a_{X_0, X_1} = \pi_{X_1}. \quad (2.44)$$

Similarly, we can model the end of the sequence by adding a silent end state X_{T+1} , such that

$$\mathbf{P}(X_{T+1}|X_T) = a_{X_T, X_{T+1}}. \quad (2.45)$$

The end state is usually not included in a general Markov chain, where the length of the chain may be undetermined and the end can occur anywhere in the sequence [11]. But since we will deal exclusively with finite sequences, adding an end state will enable the modeling of the sequence length distribution. Moreover, as we will see in Sect. 2.2.4, the inclusion of a silent begin and end state can become a valuable means to reduce computational complexity.

2.1.4 The Forward Algorithm

The *forward algorithm* is used to calculate the probability (or likelihood) of the observed data under the given model. The recurrence relation in dynamic programming is represented by the *forward variables*, defined as the joint probability of the hidden state at time $t = 1, \dots, T$ and the observed sequence up to that time,

$$\begin{aligned} \alpha_i(t) &= \mathbf{P}(Y_1^t, X_t = i) \\ &= \sum_{j \in S} \mathbf{P}(Y_1^t, X_t = i, X_{t-1} = j) \\ &= \sum_{j \in S} \mathbf{P}(Y_t, X_t = i | Y_1^{t-1}, X_{t-1} = j) \mathbf{P}(Y_1^{t-1}, X_{t-1} = j) \\ &= \sum_{j \in S} \mathbf{P}(X_t = i | X_{t-1} = j) \mathbf{P}(Y_t | Y_1^{t-1}, X_t = i) \mathbf{P}(Y_1^{t-1}, X_{t-1} = j) \\ &= \sum_{j \in S} a_{ji} b_i(Y_t | Y_1^{t-1}) \alpha_j(t-1). \end{aligned} \quad (2.46)$$

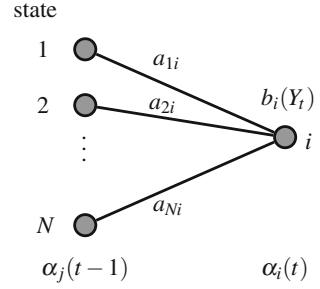
For initialization we add a silent state X_0 , where

$$\alpha_i(0) = \pi_i, \quad i \in S, \quad (2.47)$$

and for termination we add a silent end state X_{T+1} , where

$$\alpha_i(T+1) = \mathbf{P}(Y_1^T, X_{T+1} = i) = \sum_{j \in S} \alpha_j(T) a_{ji}. \quad (2.48)$$

Fig. 2.7 Each node is a sum of the forward variables at the previous position



The desired probability of the observed data, given the model, is then given by

$$\mathbf{P}(Y_1^T) = \sum_{i \in S} \mathbf{P}(Y_1^T, X_{T+1} = i) = \sum_{i \in S} \alpha_i(T + 1). \quad (2.49)$$

The forward variables are efficiently calculated using the tabular computation in dynamic programming. The calculations are organized in a table as in Fig. 2.6, that is filled in one column at a time for increasing state and time indices. The name *forward* in the forward algorithm comes from the fact that we move forward through the data. That is, each variable $\alpha_i(t)$ at time t is a (weighted) sum over all variables at time $t - 1$ (see Fig. 2.7).

The implementation of the forward algorithm is illustrated in Algorithm 2.

Algorithm 2 The forward algorithm

```

t = 1
Choose  $X_1$  according to  $\pi$ 
while  $t < T$  do
    Emit  $Y_t$  according to  $b_{X_t}(Y_t | Y_1^{t-1})$ 
    Jump to state  $X_{t+1}$  according to  $a_{X_t, X_{t+1}}$ 
    t = t + 1
end while

```

2.1.5 The Backward Algorithm

There is a useful HMM algorithm closely related to the forward, called the *backward algorithm*, which is used in particular when solving the training problem in Sect. 6.5. As the recursion in the forward algorithm proceeds in a forward direction with respect to time, the recursion for the backward variables goes in the opposite direction. The backward variable $\beta_i(t)$ is the probability of all observed data after time t , given the observed data up to this time and given that the state at time t is $X_t = i$. As with the

forward, we initialize by using a silent state, but since we are going backwards the initial state of the algorithm is the end state X_{T+1} of the chain

$$\beta_i(T+1) = 1, \quad i = 1, \dots, N. \quad (2.50)$$

For $t = T, T-1, \dots, 1$ we define the backward variables as

$$\begin{aligned} \beta_i(t) &= \mathbf{P}(Y_{t+1}^T | Y_1^t, X_t = i) \\ &= \sum_{j \in S} \mathbf{P}(Y_{t+1}^T, X_{t+1} = j | Y_1^t, X_t = i) \\ &= \sum_{j \in S} \mathbf{P}(X_{t+1} = j | X_t = i) \mathbf{P}(Y_{t+1} | Y_1^t, X_{t+1} = j) \mathbf{P}(Y_{t+2}^T | Y_1^{t+1}, X_{t+1} = j) \\ &= \sum_{j \in S} a_{ij} b_j(Y_{t+1} | Y_1^t) \beta_j(t+1). \end{aligned} \quad (2.51)$$

We finish in the silent begin state X_0 , but this does not need special treatment for the backward algorithm. The algorithm simply terminates upon calculation of $\beta_i(0)$, for $1 \leq i \leq N$. Note that, similarly to the forward algorithm, we can calculate the probability of the observed sequence using the backward algorithm as well.

$$\begin{aligned} \mathbf{P}(Y_1^T) &= \sum_{i \in S} \mathbf{P}(Y_1^T, X_1 = i) \\ &= \sum_{i \in S} \mathbf{P}(Y_2^T | Y_1, X_1 = i) \mathbf{P}(Y_1 | X_1 = i) \mathbf{P}(X_1 = i) \\ &= \sum_{i \in S} \pi_i b_i(Y_1) \beta_i(1). \end{aligned} \quad (2.52)$$

2.1.6 The Viterbi Algorithm

The purpose of using HMMs in biological sequence analysis is to utilize their strengths as a predictive tool. That is, given that we have a model and have trained its parameters, we would like to use it to classify, or *decode*, an unlabeled sequence of observations. In other words, we would like to find the *optimal* state sequence for the given observations and the given model. However, the solution to this problem depends on our definition of “optimal”. As discussed in [30], depending on the optimality criterion chosen, the solution might not even be valid. For instance, one natural criterion would be to choose the sequence of states that are individually most likely, a method commonly referred to as *posterior decoding* and discussed further in Sect. 2.1.7.1. This approach maximizes the number of correct individual states, but as soon as some state transitions in the state space have probability zero, we stand the risk of ending up with a state sequence that is indeed optimal in the sense that it

reaches the highest likelihood, but that is impossible to achieve. In the end, what we would like to find is the single best state sequence among all *valid* ones. The HMM procedure that achieves this is called the *Viterbi algorithm*. The Viterbi algorithm formulation is essentially the same as the forward algorithm, except sums are replaced by maxima, and we need a little extra bookkeeping to track the maximizing terms.

We would like to optimize the probability of the hidden state sequence, given the observed data $\mathbf{P}(X_1^T | Y_1^T)$. Note, however, that this probability is maximized at the same point as the joint probability $\mathbf{P}(X_1^T, Y_1^T)$. Therefore, we define the Viterbi variables as the joint probability of hidden and observed data up to time t , maximized over all valid state sequences. The initial conditions for the recurrence relation are the same as for the forward algorithm. The Viterbi variables for the initial silent state X_0 are given by

$$\delta_i(0) = \pi_i, \quad i = 1, \dots, N. \quad (2.53)$$

The tabular computation proceeds for $t = 1, \dots, T$ using the recurrence relation

$$\begin{aligned} \delta_i(t) &= \max_{X_1^{t-1}} \mathbf{P}(Y_1^t, X_1^{t-1}, X_t = i) \\ &= \max_{X_1^{t-2}, j} \mathbf{P}(Y_1^t, X_1^{t-2}, X_{t-1} = j, X_t = i) \\ &= \max_{X_1^{t-2}, j} \mathbf{P}(Y_1^{t-1}, X_1^{t-2}, X_{t-1} = j) \mathbf{P}(X_t = i | X_{t-1} = j) \mathbf{P}(Y_t | Y_1^{t-1}, X_t = i) \\ &= \max_{1 \leq j \leq N} \delta_j(t-1) a_{ji} b_i(Y_t | Y_1^{t-1}). \end{aligned} \quad (2.54)$$

As a result, each $\delta_i(t)$ represents the highest probability of all paths up to time t , ending in state i . To facilitate the traceback we store pointers from the current position to the optimal previous position,

$$\psi_i(t) = \operatorname{argmax}_{1 \leq j \leq N} a_{ji} b_i(Y_t | Y_1^{t-1}) \delta_j(t-1). \quad (2.55)$$

These pointers will be used to retrieve the optimal path through the state space. As for the forward algorithm, the computation is terminated by calculating the Viterbi variables for the silent end state X_{T+1}

$$\begin{aligned} \delta_i(T+1) &= \max_{X_1^T} \mathbf{P}(Y_1^T, X_1^T, X_{T+1} = i) \\ &= \max_{1 \leq j \leq N} a_{ji} \delta_j(T). \end{aligned} \quad (2.56)$$

The probability of the most likely state sequence is then given by

$$\mathbf{P}(\text{most likely state sequence}) = \max_{1 \leq i \leq N} \delta_i(T+1). \quad (2.57)$$

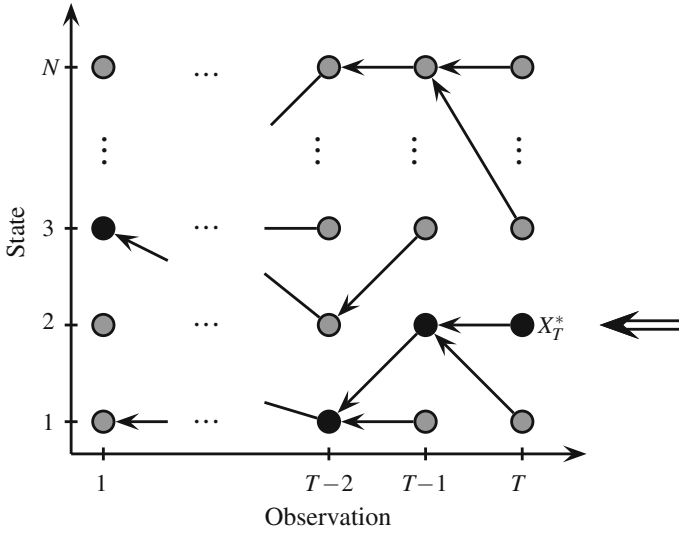


Fig. 2.8 The traceback starts in X_T^* and moves back through the state space, following the stored pointers $\psi_i(t)$

To extract the actual state sequence giving rise to this probability, we start the traceback in the silent end state giving rise to the highest value on its Viterbi variable,

$$X_{T+1}^* = \underset{1 \leq i \leq N}{\operatorname{argmax}} \delta_i(T+1), \quad (2.58)$$

and backtrack recursively through the dynamic programming table (see Fig. 2.8) using

$$X_t^* = \psi_{X_{t+1}^*}(t+1), \quad t = T, T-1, \dots, 0. \quad (2.59)$$

The resulting state sequence, $X_0^*, X_1^*, \dots, X_{T+1}^*$ is then the optimal, or most probable, state sequence for the given observations and given model, and represents a *parse* or an *annotation* of the observed sequence. The implementation of the Viterbi algorithm is illustrated in Algorithm 3.

While the Viterbi algorithm is very efficient at finding the single best path, and is suitable to use when one path clearly dominates, it is less effective when several paths have similar near-optimal probabilities. In such cases posterior decoding might work better, even though it is not guaranteed to produce a valid solution. Posterior decoding is discussed further in Sect. 2.1.7.1.

Algorithm 3 The Viterbi algorithm

```

/* Initialize */

for  $i = 1$  to  $N$  do
  Initialize:  $\delta_i(0) = \pi_i$ 
end for

/* The tabular computation */

for  $t = 1$  to  $T$  do
  for  $i = 1$  to  $N$  do
     $\delta_i(t) = 0$ 
    for  $j = 1$  to  $N$  do
      if  $a_{ji} \delta_j(t-1) > \delta_i(t)$  then
         $\delta_i(t) = \delta_j(t-1) a_{ji}$ 
         $\psi_t(i) = j$ 
      end if
    end for
     $\delta_i(t) = b_i(Y_t | Y_1^{t-1}) \delta_i(t)$ 
  end for
end for

```

2.1.7 EasyGene: A Prokaryotic Gene Finder

The gene finding problem in prokaryotes is quite different from that in eukaryotes. In particular, the prokaryotic genomes are much more dense, with much less intergenic regions and with rarely any splicing. As a result, while eukaryote genomes may contain less than 10% of coding sequence, prokaryotes tend to be very gene rich with as much as 90% of the sequence being coding. Moreover, the prokaryotic binding sites are usually located in direct vicinity of the protein-coding regions, and can be included in the model and thereby strengthen the gene signal. In contrast, eukaryotes binding sites can be located long distances from the actual gene and are often difficult to associate with the corresponding genes. However, although much simpler, the gene finding task is far from trivial even in prokaryotes, and is complicated by several issues.

Gene finding in prokaryotes is usually conducted by looking for *open reading frames* (ORFs). That is, long stretches of potentially coding sequences surrounded by a pair of candidate in-frame start and stop codons, but void of in-frame stop codons in between. The issue that arises in such an approach is that of separating real genes from ‘spurious’, or random, ORFs. The shorter the sequences considered, the more difficult this task becomes. Therefore, it is common to apply a minimum length threshold on the ORFs considered in these searches. Sharp and Cowe [37] suggested a threshold of 100 amino acids as a good trade-off between the number of missed short genes and the number of predicted spurious ORFs. It turns out, however, that such a threshold is very crude. Prokaryotic genomes contain plenty of spurious ORFs above that size, and a significant amount of true genes below it [40]. Another issue, much more prevalent in prokaryotes than in eukaryotes, is the

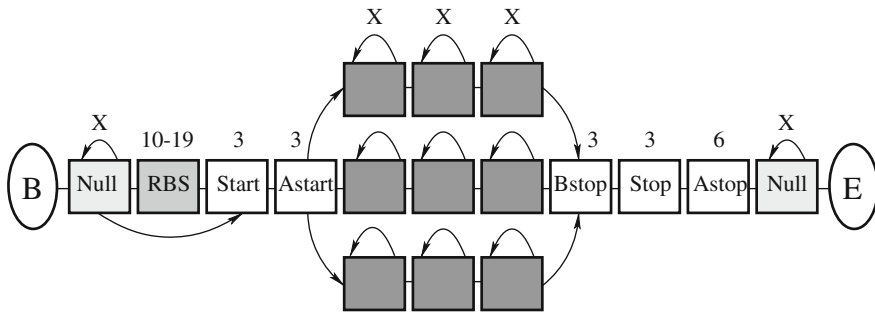


Fig. 2.9 The EasyGene gene model. The numbers above the boxes represent the number of bases modeled by that submodel, where 'X' indicates a variable number. B and E are begin and end states, the NULL-states cover intergenic region before and after the gene, the RBS state include the RBS and the spacer bases to the next state, Start and Stop model the start and stop codons, Astart, Bstop and Astop explicitly model the codons directly after the start codon and surrounding the stop codon

problem of overlapping genes, which makes the accurate detection of translation start sites notoriously difficult.

EasyGene [20] is an HMM-based prokaryotic gene finder, that attempts to address these issues. EasyGene is fully automated in that it extracts training data from a raw genomic sequence, and estimates the states for coding regions and ribosomal binding sites (RBS) used to score potential ORFs. The EasyGene state space is illustrated in Fig. 2.9. The *B* and *E* states are silent begin and end states of the HMM, and the NULL-states model everything that is not part of the gene, and not in direct vicinity of the gene. The RBS state includes the RBS as well as the bases between the RBS and the next state, and the START and STOP states correspond to the start and stop codons of the gene, respectively. While eukaryotic genes almost always start with ATG, prokaryotes use a number of alternative start codons. *E. coli* (K-12 strain), for instance, uses ATG in about 83 % of its genes, GTG in 14 % and TTG in 3 % of the cases, and an additional one or two very rare variants [4]. The stop codons are typically TAA, TAG, and TGA in both eukaryotes and prokaryotes, even if alternatives are known to exist [15]. The codons directly after the start codon and the codons surrounding the stop codon tend to follow a distribution different from the rest of the gene [38], a feature that can be used to strengthen the start and stop signals. This feature is explicitly modeled in the ASTART, BSTOP, and ASTOP states.

The model for the internal codons consists of 3 parallel submodels, allowing the HMM to keep separate statistics for atypical genes. Each submodel, consists of a series of 3 codon models, where each codon model is a 4th-order Markov model consisting of three states, one for each DNA base of the codon, capturing the codon position dependency of coding sequences. As a result the length distribution becomes negative binomial with parameters (n, p) , where n is the number of serial codon models, and p the probability of transitioning out of the specific codon model. This model allows for more general length distributions than the geometric, which

would be the result of using one codon model alone (this issue is discussed further in Sect. 2.2).

2.1.7.1 Posterior Decoding

As a consequence of using duplicated codon states, the length of an ORF is only realized as the sum over many HMM paths. While the Viterbi algorithm is a very efficient decoding algorithm when one path dominates, it is not appropriate when several paths have similar probabilities. Therefore, EasyGene uses posterior coding instead, also known as the *forward-backward algorithm* [30], where the *individually* most likely sequence of states is computed. The details on the forward-backward algorithm are given in Sect. 6.5, but in short we use the probabilities of being in a given state $s_i \in S$ at time t , given the observed sequence to determine the individually most likely state sequence. The forward-backward variables are defined as

$$\gamma_t(i) = \mathbf{P}(X_t = s_i | Y_1^T) = \frac{\alpha_i(t) \beta_i(t)}{\mathbf{P}(Y_1^T)}, \quad (2.60)$$

and the resulting optimal state sequence is given by

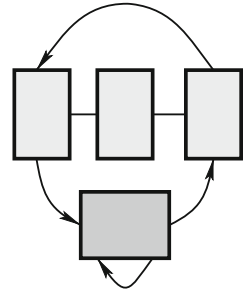
$$X_t^* = \operatorname{argmax}_{s_i \in S} \gamma_t(i), \quad 1 \leq t \leq T. \quad (2.61)$$

Assuming that there are no frameshifts or sequencing errors in the sequence, there is exactly one stop codon for each start codon, and, thus, the probability of a gene is equivalent to the posterior probability of its gene start. As a consequence, we can easily extract all possible start codons for a gene in the case of several similar scores. Moreover, in the case of overlapping genes the Viterbi algorithm would only report the highest scoring, while using posterior decoding each gene is scored and reported separately. However, posterior decoding merely bunches together independent underlying states, without checking that the parse is valid. Although this is not a big problem in prokaryotic gene finding, we still need to be careful when interpreting the output of posterior decoding.

2.1.7.2 Statistical Significance of Predictions

Along with its gene predictions, EasyGene reports a measure of statistical significance for each ORF. The measure is based on a comparison of the predicted ORFs to the expected number of ORFs in a random sequence. The random sequence model, called the NULL-model (different from the NULL-states in the main model), is a third-order Markov chain using the same base frequencies as the overall genome in question. It consists of a third-order state for intergenic regions, and a reverse codon model to capture genes on the reverse strand (see Fig. 2.10). The significance

Fig. 2.10 The NULL-model in EasyGene is used to model random sequence with the same background statistics as the overall genome. The state space consists of a third-order intergene state and a reverse codon model to capture genes on the reverse strand



measure is based on a log-odds score between the model and the NULL-model,

$$\beta = \log \frac{\mathbf{P}(Y|M)}{\mathbf{P}(Y|N)}, \quad (2.62)$$

where $\mathbf{P}(Y|M)$ is the probability that sequence Y contains an ORF under the model, and $\mathbf{P}(Y|N)$ the same probability for the NULL-model. This score is different from that reported by Genscan, which reports posterior exon probabilities based on the HMM (see Sect. 2.2.4.3).

2.2 Generalized Hidden Markov Models (GHMMs)

A major problem with standard HMM is the intrinsic modeling of state duration. Outputting exactly one observation per jump leads to a length distribution that is exponentially decaying, something that often is unsuitable for many applications. The solution to this problem is called a hidden *semi*-Markov model, or a *generalized* HMM (GHMM). The word ‘generalized’ comes from this fact, that instead of having geometric length distributions we can use a length distribution of our choice.

2.2.1 Preliminaries

A standard HMM makes a transition at each time unit t , such that the *transition time* is always equal to one, and the observed output is exactly one symbol per unit. However, by making a number of *self-transitions* into the same state, we can observe a coherent subsequence emitted from the same state. We call the length of such a sequence the *duration* of the state. Due to the Markov property, rendering the process memoryless, the duration follows a geometric distribution (see Sect. 5.2.1). Hence, using a standard HMM for the purpose of gene finding, for instance, would impose a geometric distribution on each state. It has been noted, however, that exons

in particular tend to follow a length distribution that is statistically very different from the geometric distribution. Thus forcing such a model on the sequence data may lead to bad predictions.

A *semi*-Markov model, has the same structure as a standard HMM, except that the process stays in each state a random amount of time before its next transition, as opposed to standard HMMs, where the time in each state always equals one (note that we are only considering discrete-time processes here). More formally, a semi-Markov process is a process W whose state sequence is generated by a Markov process X as before, but whose transition times are given by another process ζ that may depend on both the current state and the next. Thus, since properties of ζ may depend on the next state in X , the process W is in general not Markovian. The associated process (X, ζ) , however, *is* a Markov process, hence the name semi-Markov.

In a *hidden* semi-Markov model, or a *generalized* hidden Markov model as we will call it henceforth, there is a duration distribution associated with each state. When a state emits output, first a duration is chosen from the duration distribution, and then the corresponding length of output is generated. This generalization can improve performance by allowing for more accurate modeling of the typical duration of each particular state. As a result, the indices of the hidden and the observed process will start to differ as soon as a state has a duration that is longer than 1. For example, assume that the model generated the following output:

Hidden:	$X_1:$	$X_2:$	$X_3:$
Observed:	$Y_1 Y_2 Y_3$	$Y_4 Y_5$	$Y_6 Y_7 Y_8 Y_9$

In order to handle this, we separate the time index notation in the state sequence and in the observed sequence as follows. Given that the hidden process is in state X_l , let d_l denote the duration of X_l chosen from a length distribution $f_{X_l}(d_l)$. To keep track of the indices in the hidden versus the observed sequences, we introduce partial sums for the number of emitted symbols up to (and including) state X_l

$$p_l = \sum_{k=1}^l d_k, \text{ and } p_0 = 0. \quad (2.63)$$

We let L denote the length of the Markov process, X_1^L , and T the length of the observed process, Y_1^T . For simplicity we assume that $p_L = T$, meaning that all of the observed output generated in the final state X_L is included in the observed sequence. Now the state sequence X_1^L , the duration sequence d_1^L , and the length of the state sequence L , are hidden from the observer, and the observed data remains to be the observation sequence Y_1^T . The joint probability of the hidden and observed data becomes

$$\mathbf{P}(Y_1^T, X_1^L, d_1^L) = \prod_{l=1}^L a_{X_{l-1}, X_l} f_{X_l}(d_l) b_{X_l}(Y_{p_{l-1}+1}^{p_l} | Y_1^{p_{l-1}}), \quad (2.64)$$

where X_0 is the silent begin state as before, with

$$a_{X_0, X_1} = \pi_{X_1}. \quad (2.65)$$

One drawback with GHMMs is that statistical inference is harder than for standard HMMs. In particular, the Baum–Welch algorithm for parameter training is not applicable. The Baum–Welch algorithm is a generalized EM-algorithm (expectation–maximization), that uses counts of transition–emission pairs to update the expectation part of the algorithm. Details on the Baum–Welch algorithm, and on how to train GHMMs, can be found in Chap. 6.

2.2.2 The Forward and Backward Algorithms

One of the attractive features of using a generalized HMM for gene finding is that it provides a natural way of computing the posterior probability of a predicted generalized state, given the observed data. How this is done is described in Sect. 2.2.4.3, in the framework of the gene finding software Genscan [7]. First, we need to adjust the forward and backward algorithms in (2.46) and (2.51), respectively, to fit the GHMM framework.

The Forward Variables

Recall that the forward variables are defined as the joint probability of observed sequence up to time t , and the hidden state at time t . In a GHMM, however, we need to adjust the definition slightly, since each state can have variable durations. We define the forward variables $\alpha_i(t)$ as the probability of the observed data, and that the hidden state i at time t actually *ended* at time t . This is to say that $X_l = i$ and $p_l = t$ for some $1 \leq l \leq L$. In what follows we let D be the maximum duration of a state.

$$\begin{aligned} \alpha_i(t) &= \mathbf{P}\left(Y_1^t, \{\text{some hidden state } i \text{ ends at } t\}\right) \\ &= \mathbf{P}\left(Y_1^t, \bigcup_{l=1}^L (X_l = i, p_l = t)\right) \\ &= \sum_{j \in S} \sum_{d=1}^D \mathbf{P}\left(Y_1^t, \bigcup_{l=1}^L (X_l = i, p_l = t, d_l = d), \bigcup_{l=1}^L (X_l = j, p_l = t - d)\right) \\ &= \sum_{j \in S} \sum_{d=1}^D \left[\mathbf{P}\left(Y_1^{t-d}, \bigcup_{l=1}^L (X_l = j, p_l = t - d)\right) \right. \end{aligned} \quad (2.66a)$$

$$\cdot \mathbf{P}\left(\bigcup_{l=1}^L (X_l = i, p_l = t, d_l = d) \middle| \bigcup_{l=1}^L (X_l = j, p_l = t - d)\right) \quad (2.66b)$$

$$\cdot \mathbf{P}\left(Y_{t-d+1}^t \middle| Y_1^{t-d}, \bigcup_{l=1}^L (X_l = i, p_l = t, d_l = d), \bigcup_{l=1}^L (X_l = j, p_l = t - d)\right) \Bigg] \quad (2.66c)$$

The first term (2.66a) is simply $\alpha_j(t - d)$. To handle the conditioning on unions in the second (2.66b) and third (2.66c) terms, we make use of the following two lemmas.

Lemma 2.1 *If sets A , B , and C satisfy $B \cap C = \emptyset$ and $\mathbf{P}(A|B) = \mathbf{P}(A|C)$, then $\mathbf{P}(A|B \cup C) = \mathbf{P}(A|B)$.*

Lemma 2.2 *If for set A and disjoint sets B_1, \dots, B_n we have that $\mathbf{P}(A|B_i) = \mathbf{P}(A|B_1)$ for all $1 \leq i \leq n$, then $\mathbf{P}(A|\bigcup_{i=1}^n B_i) = \mathbf{P}(A|B_1)$.*

As a result, for the second term (2.66b) we get

$$\begin{aligned} & \mathbf{P}\left(\bigcup_{l=1}^L (X_l = i, p_l = t, d_l = d) \middle| \bigcup_{l=1}^L (X_l = j, p_l = t - d)\right) \\ &= \mathbf{P}\left(\bigcup_{l=1}^L (X_l = i, p_l = t, d_l = d) \middle| X_1 = j, p_1 = t - d\right) \\ &= \mathbf{P}(X_2 = i, p_2 = t, d_2 = d \middle| X_1 = j, p_1 = t - d) \\ &= a_{ji} f_i(d). \end{aligned} \quad (2.67)$$

Similarly, the third term (2.66c) becomes

$$\begin{aligned} & \mathbf{P}\left(Y_{t-d+1}^t \middle| Y_1^{t-d}, \bigcup_{l=1}^L (X_l = i, p_l = t, d_l = d), \bigcup_{l=1}^L (X_l = j, p_l = t - d)\right) \\ &= \mathbf{P}(Y_{t-d+1}^t \middle| Y_1^{t-d}, p_1 = t - d, d_2 = d, X_2 = i). \\ &= b_i(Y_{t-d+1}^t \middle| Y_1^{t-d}). \end{aligned} \quad (2.68)$$

Thus, the forward algorithm results in

$$\alpha_i(t) = \sum_{j \in S} \sum_{d=1}^D a_{ji} f_i(d) b_i(Y_{t-d+1}^t \middle| Y_1^{t-d}) \alpha_j(t - d). \quad (2.69)$$

We initialize and terminate as before with

$$\alpha_i(0) = \pi_i, \quad (2.70)$$

$$\alpha_i(T + 1) = \mathbf{P}(Y_1^T, X_1^L, X_{L+1} = i) = \sum_{j \in S} \alpha_j(T) a_{ji}. \quad (2.71)$$

Just as in the non-generalized case, the probability of the observed sequence, given the model, is given by summing over the terminal forward variables

$$\mathbf{P}(Y_1^T) = \sum_{i \in S} \alpha_i(T+1). \quad (2.72)$$

The Backward Variables

The backward variable $\beta_i(t)$ denotes the probability of all the observed data after time t , given the observed data up to time t and given that the hidden state i ended at time t . Skipping the details, the backward variables for GHMMs are given by

$$\begin{aligned} \beta_i(t) &= \mathbf{P}\left(Y_{t+1}^T \mid \bigcup_{l=1}^L (X_l = i, p_l = t)\right) \\ &= \sum_{j \in S} \sum_{d=1}^D a_{ij} f_j(d) b_j(Y_{t+1}^{t+d} | Y_1^t) \beta_j(t+d). \end{aligned} \quad (2.73)$$

The backward algorithm is initiated as before, using $\beta_i(T+1) = 1$ for all $i \in S$, and is terminated upon calculation of $\beta_i(0)$.

2.2.3 The Viterbi Algorithm

Now that we know what the GHMM forward algorithm looks like, adjusting the Viterbi algorithm of the standard HMM as straightforward. Recall from Sect. 2.1.6 that the conditional probability $\mathbf{P}(X_1^T | Y_1^T)$ of the state sequence given the observed data is maximized by the same state sequence as the joint probability $\mathbf{P}(Y_1^T, X_1^T)$. The same holds in the GHMM situation; the state sequence and the associated durations that maximizes $\mathbf{P}(X_1^L, d_1^L | Y_1^T)$ also maximizes the joint probability $\mathbf{P}(Y_1^T, X_1^L, d_1^L)$ given in (2.64). As for the standard HMMs, the Viterbi algorithm only differs from the forward algorithm in that the sums are replaced by maxima. Therefore, skipping the technical details, the tabular computation of the GHMM Viterbi algorithm for $t = 1, \dots, T$ becomes

$$\begin{aligned} \delta_i(t) &= \max_{l, X_1^{l-1}, d_1^l} \mathbf{P}(Y_1^t, X_1^{l-1}, X_l = i, p_l = t) \\ &= \max_{j,d} \delta_j(t-d) a_{ji} f_i(d) b_i(Y_{t-d+1}^t | Y_1^{t-d}). \end{aligned} \quad (2.74)$$

The Viterbi algorithm is initiated and terminated by

$$\delta_i(0) = \pi_i, \quad (2.75)$$

$$\delta_i(T+1) = \max_j \delta_j(T-1) a_{ji}. \quad (2.76)$$

The probability of the most likely sequence of states and durations is given by

$$\mathbf{P}(\text{most likely sequence of states and durations}) = \max_{1 \leq i \leq N} \delta_i(T+1). \quad (2.77)$$

As we evaluate $\delta_i(t)$ we record the values of the optimal previous position in the dynamic programming table, which now includes *two* values. In addition to knowing the most likely previous state $\psi_i(t)$ we need to know the most likely duration of that state, in order to jump back to the right previous cell in the table. That is, we record the previous state and its duration in the pair of variables

$$(\psi_i(t), \phi_i(t)) = \underset{j,d}{\operatorname{argmax}} \delta_j(t-d) a_{ji} f_i(d) b_i(Y_{t-d+1}^t | Y_1^{t-d}), \quad (2.78)$$

where now $\psi_i(t)$ corresponds to the maximizing previous state j , and $\phi_i(t)$ to the maximizing duration d of that state. The most probable state sequence thus ends in a state i^* which has duration $\phi_{i^*}(T+1)$ and is preceded by state $\psi_{i^*}(T+1)$, and the whole sequence is unraveled by backtracking using the ϕ 's and the ψ 's.

2.2.4 Genscan: A GHMM-Based Gene Finder

Genscan [7] is probably one of the most popular single species gene finders of all times, and included several novel improvements when it first was published. The novel features include:

- The ability to predict multiple genes in a sequence.
- The ability to predict partial genes at the end of the sequence.
- The ability to predict genes on both strands simultaneously.
- Binning the parameter set into several submodels, depending on the G+C content of the input sequence.
- Modeling long-range internal dependencies in splice sites using Maximal Dependence Decomposition (MDD).

While most gene predictors up to that point assumed the existence of exactly one complete gene in the sequence to be analyzed, Genscan allows the recognition of both multiple and partial genes. Moreover, the gene prediction is efficiently performed on both strands simultaneously, by simply adding a mirror image of the state space, connected via the intergenic state (see Fig. 2.11). Other improvements include G+C dependent model parameters. Typically, the gene density is higher and the gene

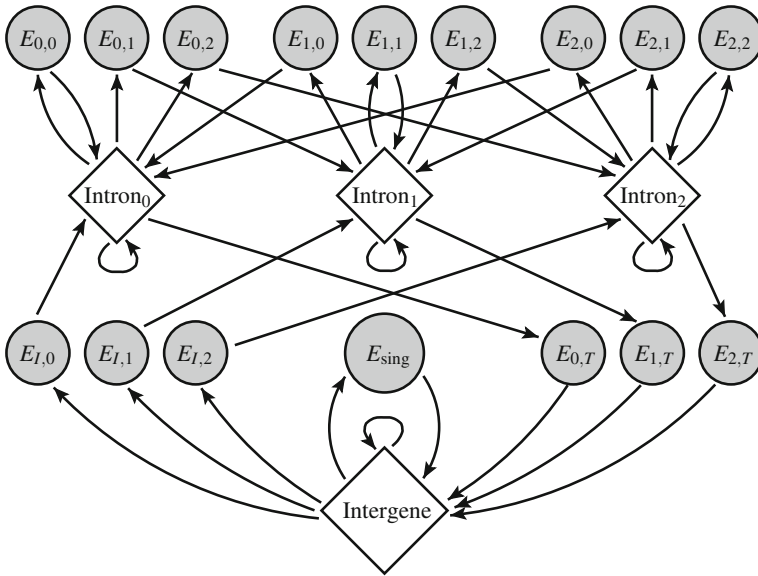


Fig. 2.11 A simplified version of the Genscan state space, modeling only the forward strand and consisting of E -states and I -states alone. The *diamond-shaped* I -states are non-generalized states emitting one symbol at a time, while the *circular* E -states are complex submodels with generalized length distributions and including the splice site models

structure is more compact in regions of higher G+C content (see Sect. 5.3.1). Since this directly affects the model parameters, Genscan separates the training sequences into four sets; sequences of less than 43 % GC-content, 43–51 %, 51–57 %, and more than 57 % G+C content. When a sequence is to be analyzed, the G+C content is first calculated, and the corresponding set of parameters is applied in the prediction process. Another novel feature in Genscan is the splice site predictor. Signal sequences in general, and splice sites in particular exhibit significant internal dependencies between nonadjacent positions, something that is not easily captured by common weight matrices, or even with higher order Markov models. Genscan uses the Maximal Dependence Decomposition (MDD) model for modeling splice sites. The MDD breaks down the splice site sequence into its specific positions and creates a decision tree arranged by the position dependencies in the order of importance. The MDD is described in more detail in Sect. 5.4.3. The improvements introduced in Genscan were quickly adapted by the gene finding community, and now constitute an integral part of most modern gene finders.

A simplified version of the Genscan state space is illustrated in Fig. 2.11. The figure only shows the forward strand and only includes exons (E -states), introns, and intergene (I -states). The full state space consists of 27 separate states, 13 for each strand, and a joint intergene state. The additional states include promoters, UTR-states, and polyA-signals as well. However, the identification of such regions

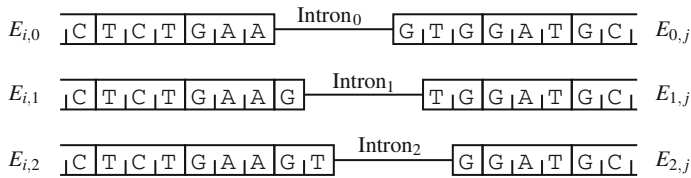


Fig. 2.12 Illustrating the notion of exon and intron phase. Intron_{*j*} comes between exon $E_{i,j}$ and exon $E_{j,k}$, $j = 0, 1, 2$

is significantly more difficult than the E - and I -states, leading to much less reliable predictions in comparison to the protein-coding portion of the genes. The reverse strand can be included in the state space by adding a mirror image of the forward states, joined by a common intergenic state. The diamond-shaped I -states are simple, non-generalized states with geometrically distributed durations. The circular E -states are more complex: they include codon compositions, a generalized length distribution, and exon boundary models (splice sites, start or stop). Four types of exons are defined, depending on which boundaries that contain them: single exons (start to stop), initial exons (start to donor), internal exons (acceptor to donor), and terminal exons (acceptor to stop). The introns and internal exons are separated into three groups, corresponding to the phase of the surrounding exons. Exons can be spliced anywhere in the reading frame; exactly between two complete codons, after the first base of the codon, or after the second base (see Fig. 2.12). If an internal intron appears exactly between two complete codons, it has phase 0, while if it occurs after the first or second nucleotide, it has phase 1 or 2, respectively. The internal exons are indexed correspondingly, with $E_{i,j}$ signifying that the previous exon ended with i extra bases (and the preceding intron has phase i), and the current exon ends with j extra bases. The notion of phase is discussed further in Sect. 5.1.1. At a first glance it may seem unnecessary to include such a large number of exon and intron states, but it turns out to be computationally efficient, as it allows us to keep track of the phase without requiring anything more than a first-order Markov chain.

It is typically the exon states that need to be generalized in the gene finding state space, since their length distributions differ significantly from the geometric distribution. Also, the length distribution tend to differ between different exon types as well (see Sect. 5.2 for details). Therefore, Genscan uses separate empirical length distributions for single, initial, internal, and terminal exons. The introns and intergenic regions, however, seem to follow the geometric length distribution fairly well provided that a certain minimum threshold is exceeded. Also, the 5'UTR and 3'UTR state lengths are modeled using geometric distribution.

2.2.4.1 Sequence Generation Algorithm

Consider generating a genomic sequence from the Genscan model. Say that we start off in the intergenic state and do a number of self-transitions. The state duration of

the intergene and the introns in each step is always $d_l \equiv 1$. Thus, each time we visit the intergenic state, a single-base Y_t is generated according to some distribution $b_{IG}(Y_t|Y_1^{t-1})$. The output may depend on all of the preceding sequence, but most models only include contexts of a few bases back. Eventually, after some geometrically distributed time t we make a transition into a different state, for instance the $E_{I,2}$ state. The $E_{I,2}$ state represents an initial exon including the two first bases of its last codon. An exon duration d is generated according to some generalized length distribution $f_{E_{I,2}}(d)$, where $f_{E_{I,2}}(d) = 0$ if $(d \bmod 3) \neq 2$. We generate d bases Y_{t+1}^{t+d} according to emission distribution $b_{E_{I,2}}(Y_{t+1}^{t+d}|Y_1^t)$, and jump with probability 1 to intron I_2 . Note that while the exon sequence may depend on the entire previous sequence as well, it is typically modeled as independent of the preceding I -state. In intron I_2 we continue as in the intergene state, executing a geometric number of self-transitions and emitting a state-specific base in each step, before jumping to one of the internal exons $E_{2,i}$ or the terminal exon $E_{2,T}$. The exon sequence will finish off the last codon of the previous exon, before continuing to generate complete codons. It should be clear by now that the length of each specific exon state is fixed mod 3.

The algorithm for generating a sequence of predetermined length from the Genscan model is summarized in Algorithm 4. The last state of the algorithm is truncated appropriately to ensure that $p_L = T$, meaning that the observed sequence ends exactly at the last base of the last state.

Note that we left out any mentioning of the splice sites surrounding the introns. While the I -states are simple, non-generalized states, the exons are themselves complex submodels. Each exon submodel consists of its length distribution, the model for the coding sequence as well as the exon boundaries (start, stop, donor, and acceptor signals). Details on the exon submodels are given in Chap. 5.

Naturally, using Markov models as sequence generators are only a approximation of the reality. The process at which real genes are generated is bound to be far more complex than any kind of mathematical model. However, using such models as approximations of the real processes provide a powerful tool for the identification of genes and enables us to reconstruct highly complex gene structures. Some limitations of the Genscan model, however, include: (1) Only protein-coding genes are considered, as the pattern for RNA genes are quite different and would need separate models to be detected [32]. (2) Only introns in between protein-coding exons are modeled, and not UTR introns for example. (3) Overlapping and alternatively spliced genes are not handled.

2.2.4.2 Reducing Computational Complexity

Gene prediction, as well as sequence analysis in general, involves dealing with large quantities of data, and an important question is how feasible the calculations of the HMM algorithms are in practice. We can address this by estimating the number of multiplications (or additions, if we use logarithms) required for each forward and backward variable. The emission distribution $b_i(Y_{t+1}^{t+d}|Y_1^t)$ is commonly calculated

Algorithm 4 The Genscan model

```

/* Initialize */

 $l = 1$ 
 $p_0 = 0$ 
Choose  $X_1$  according to  $\pi$ 
Choose state duration  $d_1$  according to  $f_{X_1}(\cdot)$ 
 $p_1 = d_1$ 

/* Generate sequence */

while  $p_l \leq T$  do
  Emit  $Y_{p_{l-1}+1}^{p_l}$  according to  $b_{X_l}(\cdot|\cdot)$ 
  Jump to state  $X_{l+1}$  according to  $a_{X_l, X_{l+1}}$ 
   $l = l + 1$ 
  Choose state duration  $d_l$  according to  $f_{X_l}(\cdot)$ 
   $p_l = p_{l-1} + d_l$ 
end while

/* Truncate the last state to get  $p_L = T$  */

if  $p_l > T$  then
  Emit  $Y_{p_{l-1}+1}^T$  according to  $b_{X_l}(\cdot|\cdot)$ 
end if

```

as the product of d single-base probabilities $\mathbf{P}(Y_t|Y_{t-1}, \dots, Y_{t-k})$, each of which is typically conditioned on a number k of previous bases. Therefore, if D is the maximum duration of a state and N the number of states, the order of computing a forward variable $\alpha_i(t)$ is $O(ND^2)$. If T is the length of the DNA sequence, there are NT such variables, leading to a total of $O(TN^2D^2)$ operations to compute the forward algorithm, and $O(TN)$ bytes to store the variable values. This means that the complexity of the problem is linear in the length of the sequence being analyzed, which is a desired property. However, there is a lot of structure in the topology of the model in Fig. 2.11 that we can take advantage of to get a significant reduction in computational complexity. First, we partition the state space into exon states E and intron and intergenic states I , where $S = E \cup I$, and let N_E and N_I denote the number of separate states in each class, such that $N = N_E + N_I$. Using the structure of the state space, we can actually reduce the memory requirement from $O(TN)$ to $O(TN_I)$ by storing $\alpha_i(t)$ only for the I -states.

Looking at the model, it is clear that an E -state *must* be followed by an I -state. Also, because of the alternation between E - and I -states, at any given time and state, either the previous state, or the state before that, was an I -state. Returning to the forward recursion in (2.69), we can separate the summation over the previous state j between the two-state classes E and I .

$$\begin{aligned}\alpha_i(t) &= \sum_{j \in S} \sum_{d=1}^D \alpha_j(t-d) a_{ji} f_i(d) b_i(Y_{t-d+1}^t | Y_1^{t-d}) \\ &= \sum_{j \in I} \sum_{d=1}^D \alpha_j(t-d) a_{ji} f_i(d) b_i(Y_{t-d+1}^t | Y_1^{t-d})\end{aligned}\quad (2.79a)$$

$$+ \sum_{j \in E} \sum_{d=1}^D \alpha_j(t-d) a_{ji} f_i(d) b_i(Y_{t-d+1}^t | Y_1^{t-d}). \quad (2.79b)$$

The first term (2.79a) depends on forward variables for previous I -states only, and needs no further modification. To make the same come true for the second term (2.79b) we need to step back one state further, which then must be an I -state. The memory-reduced forward recursion can then be written as

$$\alpha_i(t) = \sum_{j \in I} \sum_{d=1}^D \alpha_j(t-d) a_{ji} f_i(d) b_i(Y_{t-d+1}^t | Y_1^{t-d}) \quad (2.80a)$$

$$+ \sum_{j \in E} \sum_{k \in I} \sum_{d=1}^D \sum_{e=1}^D \left[a_{kj} \alpha_k(t-e-1) f_j(e) b_j(Y_{t-d-e+1}^{t-d} | Y_1^{t-d-e}) \right. \quad (2.80b)$$

$$\left. \cdot a_{ji} f_i(d) b_i(Y_{t-d+1}^t | Y_1^{t-d}) \right]. \quad (2.80c)$$

In (2.80b), instead of referring to the forward variables of the E -states, we can move one step further back and use the forward variables of the preceding I -states, and include the contribution of the E -states explicitly. As a result we do not have to store the forward variables of the E -states.

Further simplifications can be made. First, recall that the I -states always have duration $d \equiv 1$. As a result, the summation over d can be dropped and $f_i(d)$ can be set to 1. Second, when leaving an exon $E_{i,j}$ we jump to I_j with probability 1. Thus, the transition probability $a_{ji} = 1$ for $j \in E$. Note also that the only way we can jump directly between I -states is via self-transitions, which means that the first sum over I -states only has one positive term. Third, we note that for any given pair of I -states (i, k) with an intervening E -state, the connecting E -state is unique, call it $E_{k,i}$. Thus, the summation over $j \in E$ in (2.80b) has only one positive term as well, the one involving $E_{k,i}$. Finally, the forward recursion becomes

$$\begin{aligned}\alpha_i(t) &= b_i(Y_t | Y_1^{t-1}) \left[a_{ii} \alpha_i(t-1) \right. \\ &\quad \left. + \sum_{k \in I} \sum_{e=1}^D \alpha_k(t-e-1) a_{k, E_{k,i}} f_{E_{k,i}}(e) b_{E_{k,i}}(Y_{t-e}^{t-1} | Y_1^{t-e-1}) \right].\end{aligned}\quad (2.81)$$

As a result, the required memory needed to store the forward variables becomes $O(T N_I)$ and the number of operations $O(T N_I^2 D^2)$. Since $N_I = 4$ and $N = 20$ we have achieved an 80% reduction in memory usage and a 96% reduction of the number of operations needed. Depending on the choice of exon emission distribution $b_{E_{i,j}}$ it may be possible to reduce the number of operations further, and significantly boost the performance of the algorithm. For instance, if it is possible to make a lookup table to compute exon probabilities in a small number of operations, then the complexity of the forward calculation may be reduced by a factor D to $O(T N_I^2 D)$.

If we use the Genscan reduction of the state space, an extra need for a silent begin and end state arises. Without an end state, the likelihood of the observed data would be calculated using

$$\mathbf{P}(Y_1^T) = \sum_{i=1}^N \alpha_i(T). \quad (2.82)$$

In the reduced model, however, we only store $\alpha_i(T)$ for the I -states, $i \in I$. Although we do not allow the sequence (or predictions of the sequence) to begin or end in the middle of an exon, we still want it to be possible to predict an exon with a boundary at Y_1 or Y_T , respectively, and the sum in (2.82) would have to run over all states, not just the I -states. We get around this by adding silent begin and end states to the model. For the begin state X_0 we set the initial distribution to be positive only for I -states. The initialization conditions are therefore

$$\alpha_i(0) = \pi_i, \quad i \in I. \quad (2.83)$$

Similarly, for the end state X_{L+1} we set transition probabilities from X_L to be positive only for transitions to I -states. The termination conditions become

$$\alpha_i(T+1) = \mathbf{P}(Y_1^T, X_{L+1} = i), \quad i \in I. \quad (2.84)$$

As before, the expression for $\alpha_i(T+1)$ is the same as for the other forward variables in (2.81), except it does not have the output term b_i . The likelihood of the observed data can then be calculated as usual

$$\mathbf{P}(Y_1^T) = \sum_{i \in I} \alpha_i(T+1). \quad (2.85)$$

In the same manner, the backward and the Viterbi algorithms can be optimized to reduce memory and computational complexity. The backward algorithm simplifies to

$$\begin{aligned} \beta_i(t) &= \beta_i(t+1) a_{ii} b_i(Y_{t+1} | Y_1^t) \\ &+ \sum_{k \in I} \sum_{e=1}^D \beta_k(t+e+1) a_{i,E_{i,k}} f_{E_{i,k}}(e) b_{E_{i,k}}(Y_{t+1}^{t+d} | Y_1^t) b_j(Y_{t+e+1} | Y_1^{t+e}), \end{aligned} \quad (2.86)$$

with initialization $\beta_i(T+1) = 1, i \in S$.

The optimized Viterbi algorithm becomes

$$\delta_i(t) = b_i(Y_t|Y_1^{t-1}) \cdot \max \left\{ \delta_i(t-1)a_{ii}, \max_{\substack{k \in I \\ 1 \leq e \leq D}} \left\{ \delta_k(t-e-1)a_{k,E_{k,i}} f_{E_{k,i}}(e) b_{E_{k,i}}(Y_t^{t-1}|Y_1^{t-e-1}) \right\} \right\} \quad (2.87)$$

with initialization $\delta_i(0) = \pi_i$, $i \in I$ and termination as in (2.87) but without the b_i term. The backtracking procedure has to be changed slightly for the optimized algorithms. We record the previous I -state that achieved the maximum, and if this max involved passing through an exon state we need to record the maximizing duration of that exon as well. Otherwise, we record that the max was achieved via a self-transition with a duration $d = 1$.

It is possible to speed things up further by using the fact that certain features are (almost) always present in some of the states. For example, every initial exon must start with a start codon ATG, and every terminal exon must end with one of three possible stop codons; TAA, TAG, or TGA. Similarly, almost all donor sites have consensus GT as the first two bases of the intron, and almost all acceptor sites have an AG consensus as the final two bases of the intron. If we are willing to limit ourselves to genes matching these rules only, we can restrict the summation over the state length in the forward algorithm to sum only over lengths that are consistent with these rules. The extent to which this reduces the computations will depend on the composition of the sequence being analyzed.

Repeat masking can also help in speeding up the algorithms. The key observation is that certain repeats (in particular long interspersed repeats such as the *Alu* repeat) never occur in coding exons. Therefore, it is possible to substantially reduce the number of potential exons to be considered (and summed over in the algorithm). The effect on the computational complexity will depend on the frequency of repeats and their structure in the sequence being analyzed.

2.2.4.3 Exon Probabilities

One of the attractive features of using a GHMM for gene prediction is that it provides a natural way of computing the *posterior* probability of a predicted exon, given the observed data. Say that we have predicted an exon E^* of type s_e between bases a to b , such that the length of the exon is $d = (b - a + 1)$. We would like to compute the probability that the prediction is correct, i.e., the probability that the exon is part of a real gene and is predicted in the correct frame.

$$\begin{aligned}
\mathbf{P}(E^* \in s_e \text{ is correct} | Y_1^T) &= \mathbf{P}\left(\bigcup_{l=1}^L (X_l = s_e, d_l = d, p_{l-1} = a - 1) \middle| Y_1^T\right) \\
&= \frac{\mathbf{P}\left(Y_1^T, \bigcup_{l=1}^L (X_l = s_e, d_l = d, p_{l-1} = a - 1)\right)}{\mathbf{P}(Y_1^T)}. \tag{2.88}
\end{aligned}$$

The denominator is simply the probability calculated in (2.85). To simplify the numerator in (2.88), we recall that for any given pair of I -states surrounding an E -state, the connecting E -state is uniquely defined. The opposite holds true as well; every exon type is surrounded by a unique pair of I -states. Thus, if $X_l = s_e$ we can determine the previous and next I -states, call them i^- and i^+ . The union in the numerator in (2.88) can then be split into two unions, one for the preceding I -state i^- and one for the subsequent exon and I -state i^+ , and the desired probability can be calculated using intermediate values of the forward and the backward algorithms.

$$\begin{aligned}
&\mathbf{P}\left(Y_1^T, \bigcup_{l=1}^L (X_l = s_e, d_l = d, p_{l-1} = a - 1)\right) \\
&= \mathbf{P}\left(Y_1^T, \bigcup_{l=1}^L (X_l = i^-, p_l = a - 1), \bigcup_{l=1}^L (X_l = s_e, X_{l+1} = i^+, d_l = d, p_l = b)\right) \\
&= \mathbf{P}\left(Y_1^{a-1}, \bigcup_{l=1}^L (X_l = i^-, p_l = a - 1)\right) \\
&\quad \cdot \mathbf{P}\left(Y_a^b, \bigcup_{l=1}^L (X_l = s_e, X_{l+1} = i^+, d_l = d, p_l = b) \middle| Y_1^{a-1}, \bigcup_{l=1}^L (X_l = i^-, p_l = a - 1)\right) \\
&\quad \cdot \mathbf{P}\left(Y_{b+1}^T \middle| Y_1^b, \bigcup_{l=1}^L (X_l = i^+, p_l = b + 1)\right) \\
&= \alpha_{i^-(a-1)} a_{i^-,e} f_{s_e}(d) b_{s_e}(Y_a^b | Y_1^{a-1}) \beta_{i^+(b+1)}. \tag{2.89}
\end{aligned}$$

This probability can be interpreted as the probability that there is an exon of particular type s_e running exactly from positions a to b in the sequence. The forward variable $\alpha_{i^-(a-1)}$ represents the probability of all possible parses to the left of the exon that ends in the appropriate I -state i^- , while the backward variable $\beta_{i^+(b+1)}$ captures the probability of all parses to the right of the exon, beginning in I -state i^+ (see Fig. 2.13). Thus, the probability is constituted by the sum over *all* possible pairings of parses to the left and the right of the exon in question, and not only by intrinsic properties of the exon model itself. While the exon scores in many other gene finders depend on local properties such as splice signals and codon composition, the exon probability in Genscan is affected by the entire sequence. For instance, the probability of an initial exon will be boosted if it is preceded by a strong promoter signal at an appropriate distance upstream of a . This procedure, generally referred to

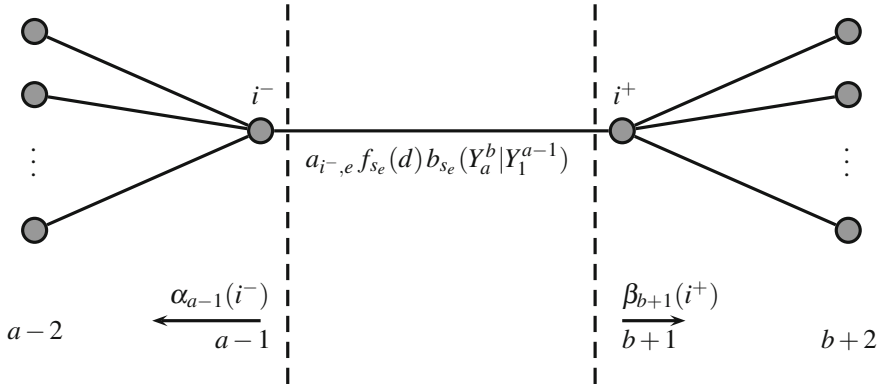


Fig. 2.13 Illustration of the forward-backward procedure for calculating the probability of a given predicted exon

as the *forward-backward algorithm*, is presented in [30] as a method for reestimating parameters in the training process, and is discussed in more general terms in Sect. 6.5.

We might prefer an exon probability that is less specific than the one in (2.89). For instance, we might want to know the probability that there is any kind of exon at all in a certain region, rather than having to specify the exon type. There is a second kind of probability that can help address this issue.

Note that the probability of being in (and at the end of) state i at time t is given by

$$\alpha_i(t) \beta_i(t) = \mathbf{P}(Y_1^T, \bigcup_{l=1}^L (X_l = i, p_l = t)). \quad (2.90)$$

This yields

$$\sum_{i \in I} \alpha_i(t) \beta_i(t) = \mathbf{P}(Y_1^T, \text{state } i \text{ at time } t \text{ is an } I\text{-state}). \quad (2.91)$$

Therefore, if we normalize (2.91) by the probability of the entire sequence $\mathbf{P}(Y_1^T)$, and subtract the result from 1, we get

$$\begin{aligned} \mathbf{P}(\text{the hidden state at time } t \text{ is some kind of exon} | Y_1^T) &= \\ &= 1 - \frac{\sum_{i \in I} \alpha_i(t) \beta_i(t)}{\mathbf{P}(Y_1^T)}. \end{aligned} \quad (2.92)$$

This offers an alternative kind of probability to the exon probability in (2.89). It does not help much in determining what kind of exon it is or where its boundaries are, but may help indicating alternative candidate exon regions. Such regions could be missed by the Viterbi algorithm since the Viterbi only determines the single most

likely sequence of exons, and there could be highly probable alternative splicings of a gene that goes undetected.

2.3 Interpolated Markov Models (IMMs)

Markov models have successfully been used as content sensors in DNA sequence analysis, both as discriminators between coding and noncoding sequences (see Chap. 5) as well as the detection of regular motifs such as eukaryotic promoters [25]. Usually higher order Markov models are required, due to long-range dependencies within a sequence. For instance, a model for coding regions should at least be of 2nd-order, because of the organization of nucleotides into codons. A 5th-order or higher would be even more preferred, since neighboring codons tend to be dependent as well. Basically, the higher the order the more sensitive the model is. The drawback, however, is that as the order increases, the required size of the training data grows exponentially. For instance, in a training model of order k there are 4^{k+1} probabilities to estimate. Thus, for a 5th-order model the training set has to be large enough to contain all 4096 possible hexamers, and sufficiently many times to provide reliable estimates, which is rarely the case in gene finding. The training data gets even more sparse, when used for the recognition of regulatory motifs such as promoters [25], or for automatic correction of sequence errors in low-quality data such as ESTs [39]. A common solution to the problem of sparse training data is to “smooth” the parameter estimates in some way in order to avoid zero probabilities. Smoothing strategies include using *pseudocounts*, *backing-off procedures*, or *interpolation techniques*. Using pseudocounts simply involves various ways of adding extra counts to the observed frequencies (see Sect. 6.2). Backing-off procedures involve setting the model to operate on a lower order when training data is insufficient.

In *interpolated Markov models* (IMMs) the order of the model is not fixed. Instead an interpolation of several Markov models of different orders is used. In pattern recognition these models are called *variable-order Markov models* [2], in data compression they go under the name of *variable-length Markov models* or *context trees* [31], and in speech recognition they are commonly referred to as *stochastic language models* [36]. The idea of IMMs is that although some oligomers occur too rarely to give reliable estimates, some may be very frequent, and would provide useful information to the prediction if included. Thus, instead of falling back to a lower order Markov model altogether, an IMM attempts to use the extra strength of the higher order, whenever there is data to support a longer context. As a result IMMs can capture both large and small dependencies in the sequence corresponding to the available statistics in the training set. Although an IMM is usually less powerful than a hidden Markov model, it has proved successful for many applications, where the problem of sparse training sets is frequent.

2.3.1 Preliminaries

The likelihood of a sequence Y_1^T can be decomposed as

$$\mathbf{P}(Y_1^T) = \prod_{t=1}^T \mathbf{P}(Y_t | Y_1^{t-1}). \quad (2.93)$$

However, using the entire previous sequence as context requires a huge training set, and is very expensive computationally. Thus a common approximation is to use an upper limit k of the context length.

The resulting model becomes a k th-order Markov model

$$\mathbf{P}(Y_1^T) \approx \prod_{t=1}^T \mathbf{P}(Y_t | Y_{t-k}^{t-1}). \quad (2.94)$$

Now, assume that we want to classify a sequence into one of N possible states, or classes $S = \{s_1, \dots, s_N\}$. The conditional probabilities in (2.94) need to be estimated for each class, from a training set of known classification. The sequence is then classified into the state with the highest likelihood

$$s^* = \operatorname{argmax}_{s_i \in S} \mathbf{P}(Y_1^T | s_i). \quad (2.95)$$

The maximum likelihood (ML) estimates of the conditional probabilities in (2.94) are given by

$$\hat{P}(Y_t | Y_{t-k}^{t-1}) = \frac{f(Y_{t-k}^t)}{f(Y_{t-k}^{t-1})}, \quad (2.96)$$

where $f(Y_a^b)$ denotes the frequency count of the sequence Y_a^b . One problem, with the ML-estimates, however, is that when some k -mers are very infrequent, they may yield very unreliable estimates, or probability zero even. Even though some such k -mers may actually not belong to the specific class, and should yield a zero count, others may be missing due to sparse training data. The trick used in IMMs to overcome this problem, is to combine the Markov models of different orders. The next section describes two different interpolation schemes used to combine k -mers of different lengths in order to smooth the estimates of low-frequent oligomers.

2.3.2 Linear and Rational Interpolation

Interpolated Markov models can be seen as a generalization of fixed-order Markov models, where a combination of models of different orders is used. Instead of one

fixed order, the conditional probabilities in (2.94) are estimated using a combination of the ML-estimates in (2.96). *Linear interpolation* [36] can be written as

$$\tilde{P}(Y_t|Y_{t-k}^{t-1}) = \rho_0 \frac{1}{L} + \rho_1 \hat{P}(Y_t) + \rho_2 \hat{P}(Y_t|Y_{t-1}) + \cdots + \rho_k \hat{P}(Y_t|Y_{t-k}^{t-1}), \quad (2.97)$$

where $\hat{P}(Y_t|Y_{t-k}^{t-1})$ is the ML estimate in (2.96). The coefficients ρ_i are positive constants that sum to one, such that \tilde{P} is still a probability, and the factor $1/L$ is a kind of “pseudocount” that ensures that none of the conditional probabilities are set to zero. The ML-estimates \hat{P} of the different order models are based on counts of oligomers in the training data. The coefficients ρ_i can be optimized with respect to the likelihood using the EM-algorithm described in Sect. 6.4, where they are treated as hidden variables in an HMM.

One problem with linear interpolation is that all the different orders in (2.97) are treated equally, although some estimates may be less reliable than the others. *Rational interpolation* [36] includes a weight function that scores the reliability of a context Y_{t-i}^{t-1} , such that

$$\tilde{P}(Y_t|Y_{t-k}^{t-1}) = \frac{\sum_{i=0}^k \rho_i \cdot g(Y_{t-i}^{t-1}) \cdot \hat{P}(Y_t|Y_{t-i}^{t-1})}{\sum_{i=0}^k \rho_i \cdot g(Y_{t-i}^{t-1})} \quad (2.98)$$

where the denominator is needed for normalization. The weight function g can be chosen different. For instance in [36] a sigmoid function is chosen, where the shape depends on a constant bias C

$$g(Y_{t-i}^{t-1}) = \frac{f(Y_{t-i}^{t-1})}{f(Y_{t-i}^{t-1}) + C}. \quad (2.99)$$

For $C = 0$ the model reduces to linear interpolation, but for $C > 0$ we obtain the rational interpolation model

$$\tilde{P}(Y_t|Y_{t-k}^{t-1}) = \frac{\sum_{i=0}^k \rho_i \frac{f(Y_{t-i}^{t-1})}{f(Y_{t-i}^{t-1}) + C}}{\sum_{i=0}^k \rho_i \frac{f(Y_{t-i}^{t-1})}{f(Y_{t-i}^{t-1}) + C}}. \quad (2.100)$$

The bias has the most impact on small datasets, and the larger the training set, the smaller the influence of C . One problem with rational interpolation is that the

EM-algorithm cannot be used to estimate the coefficients ρ_i . Instead a gradient descent method can be used to reach a local optimum [36].

2.3.3 GLIMMER: A Microbial Gene Finder

Microscopic organism that are too small to be observed by the naked eye are often referred to as *microbes*, or *microorganisms*. They do not constitute a specific classification, but can be found in almost all different taxa of life, including bacteria, animals, fungi, and plants, and include both prokaryotes and eukaryotes. Microbes are typically unicellular, and exist anywhere in the biosphere where there is liquid water, and they can survive extreme conditions such as heat, cold, acidity, salt, and darkness. Besides their importance in a wide variety of areas such as food production, water treatment (removing contaminants), and energy production (fermenting ethanol), microbes pose as important models and tools for biotechnology, biochemistry, genetics, and molecular biology. Examples of popular microbes are the budding yeast *Saccharomyces cerevisiae* and the bacterium *Escherichia coli*.

Splicing is rare in microbial genomes (as in prokaryotes in general, see Sect. 2.1.7). Thus, the issue of microbial gene finding is not to find actual coding sequences and determine the gene structures, but to identify the correct reading frame, and to separate overlapping genes. The main component in a microbial gene finder is, thus, the content sensor that scores coding potential and captures dependencies between nucleotides in open reading frames (ORFs). In addition, only some preprocessing to select potential ORFs, and some post-processing to handle overlapping gene candidates are necessary.

GLIMMER (Gene Locator and Interpolated Markov ModelER) is a microbial gene finder, particularly suited for bacteria, archaea, and viruses [9, 10, 35]. It uses an IMM to discriminate between coding and noncoding regions. The program consists of two sub-modules; one that builds the IMM from training data, and one that uses the resulting model to score new sequences.

2.3.3.1 Gene Prediction

GLIMMER scores a new sequence Y_1^T using a k th-order IMM

$$\mathbf{P}(Y_1^T) = \sum_{t=1}^T \text{IMM}_k(Y_{t-k}^t), \quad (2.101)$$

where IMM_k is calculated as

$$\text{IMM}_k(Y_{t-k}^t) = \rho_k(Y_{t-k}^{t-1}) \cdot \mathbf{P}(Y_t | Y_{t-k}^{t-1}) + (1 - \rho_k(Y_{t-k}^{t-1})) \cdot \text{IMM}_{k-1}(Y_{t-k}^t). \quad (2.102)$$

Table 2.3 Example in [45] of the gene prediction output of GLIMMER

orfID	start	end	frame	score
-----	-----	-----	--	-----
> <i>Escherichia coli</i> O157:H7				
orf00001	11952	98	-3	2.84
orf00003	351	133	-1	5.25
orf00004	312	2816	+3	11.33
orf00005	2854	3750	+1	10.02
orf00007	3751	5037	+1	13.63

The training of the probabilities $\mathbf{P}(Y_t|Y_{t-k}^{t-1})$ and the coefficients ρ_k is described in the next section. The gene prediction procedure of GLIMMER goes as follows:

1. Identify all ORFs longer than a given threshold in the input sequence.
2. Score the ORFs in each of the six reading frames, using (2.101).
3. Select ORFs scoring higher than a given threshold for further analysis.
4. Examine selected ORFs for overlaps.
5. Report ORFs that passed the overlap analysis.

Instead of scoring the entire input sequence, open reading frames (ORFs) exceeding a given minimum length are selected and scored using (2.101). Gene dense genomes usually contain overlapping genes, and such occurrences are investigated further. If two ORFs of different reading frames overlap by more than some preset minimum, the overlapping region is scored separately in all six reading frames, and if the longer ORF scores higher than the shorter in this region, the shorter ORF is dropped from the analysis. GLIMMER outputs a set of predicted genes, along with notes on overlaps that may need further examination. An example of the output format of the final gene predictions is given in Table 2.3 (taken from [45]). The columns represent the ORF identifier, the start and stop coordinates of the genes, the reading frame, and the gene score.

A further extension of the IMM used in GLIMMER are the *Interpolated context models* (ICMs). While an IMM can use as many bases in the context as the training data allows, an ICM can use *any* bases in the given context. That is, bases not necessarily adjacent to one another. The idea is to capture the high dependence on codon position when scoring a base. Often the ICM will choose a context identical to that the corresponding IMM would have chosen, but sometimes, as in the case with the third codon position, a slightly different model will be used.

2.3.3.2 Training the IMM

Seven submodels are trained by GLIMMER on a set of known sequences; one for each reading frame (three for each strand), and one for noncoding regions. Each submodel is trained by counting the occurrences of all oligomers of lengths $1, \dots, k + 1$ where k is a predefined maximum context depth (default is $k = 8$ in GLIMMER). The last

base of the oligomer defines the frame, and the preceding bases represent the context of that base. The likelihoods $\mathbf{P}(Y_t|Y_{t-k}^{t-1})$ in (2.102) are estimated directly from these frequency counts using

$$\mathbf{P}(Y_t|Y_{t-k}^{t-1}) = \frac{f(Y_{t-k}^t)}{\sum_{y \in V} f(Y_{t-k}^{t-1}, y)}, \quad (2.103)$$

where (Y_{t-k}^{t-1}, y) denotes the concatenation of the context sequence and symbol $y \in \{A, C, G, T\}$.

The coefficients $\rho_k(Y_{t-k}^{t-1})$ are determined using two criteria: if the context Y_{t-k}^{t-1} occurs frequently enough, the actual frequency is used and the weight ρ_k is set to 1. The frequency threshold used is 400, and gives about 95 % confidence that the estimated probabilities are within ± 0.05 of their true value [35]. Otherwise, if a context occurs less than 400 times in the training set, the frequency of the context is compared to the IMM-value of the one base shorter context, to see if the longer context adds information to the prediction.

The comparison is performed using a χ^2 -test

$$\rho_k(Y_{t-k}^{t-1}) = \begin{cases} 0 & \text{if } c < 0.5, \\ \frac{c}{400} \sum_{y \in V} f(Y_{t-k}^{t-1}, y) & \text{if } c \geq 0.5 \end{cases} \quad (2.104)$$

where c is the probability, taken from the χ^2 -distribution, that the frequencies of the longer context differ from the IMM-values of the shorter context. That is, if we let

$$X^2 = \sum_{y \in \{A, C, G, T\}} \frac{\left[f(Y_{t-k}^{t-1}, y) - \text{IMM}_{k-1}(Y_{t-k+1}^{t-1}, y) \right]^2}{\text{IMM}_{k-1}(Y_{t-k+1}^{t-1}, y)}, \quad (2.105)$$

where X^2 is χ^2 -distributed with 3 degrees of freedom, the probability c is given by

$$c = \mathbf{P}(\chi_3^2 \geq X^2). \quad (2.106)$$

In effect, the frequencies of the longer context are compared to the IMM-values of the one base shorter context, and if there is a significant difference between contexts the longer context serve as a better predictor and gets a higher value on the coefficient ρ . If there is little difference, meaning that the longer context adds no significant information, the longer context model gets a lower value on ρ .

2.3.3.3 GlimmerM

Eukaryotes such as the yeast *S. cerevisiae*, or the malaria parasite *Plasmodium falciparum*, are commonly analyzed using gene finders optimized for human. While these genomes have a gene density that is significantly lower than for microbes, they are still very gene rich, and a prokaryote gene finder may perform better than a human gene finder. GlimmerM is based on the GLIMMER method, but is optimized for gene densities around 20%, and has incorporated the *GeneSplicer* splice site detector [27]. Moreover, GlimmerM uses a combination of decision trees and IMM-based exon scoring. The decision trees, built by the OC1 system [24], estimate the probability that a given sequence is coding, and the resulting gene models are accepted if the IMM-score for the coding sequence is above a certain threshold.

2.4 Neural Networks

Artificial neural networks were first developed in an attempt to mimic the information processing and learning of the biological nervous system, such as the brain, in order to acquire some of their immense computational power. While the artificial neurons used in neural networks today remain quite far from their biological counterpart, their computational power has proved useful in a number of fields. Neural network models have traditionally been used in speech and image recognition, but has become more and more popular as components in DNA sequence analysis. In general, neural networks are suitable for classification problems with computationally complex patterns and many hypotheses to be evaluated in parallel.

Neural networks are essentially nonlinear mappings between a set of input variables and a set of output variables. An advantage of neural networks over other such mappings is that while many other techniques grow exponentially with the dimension of the input space, neural networks typically only grow linearly, or quadratically, with input dimension. We give a brief overview of the neural networks most commonly used in computational biology, the backpropagated feed-forward neural networks. For a more thorough treatment, see for instance [1].

2.4.1 Biological Neurons

Biological nervous systems, such as the brain, consist of myriads of *neurons*, which are specialized cells designed to process and transmit information. Learning, for instance, takes place when neurons communicate with each other. Each neuron can connect to several thousands of others, and multiple neurons can fire in parallel. Hence, the human brain, consisting of something like 10^{11} neurons, constitute a parallel processor with a capability that is vastly superior to the most advanced computer clusters that exist today.

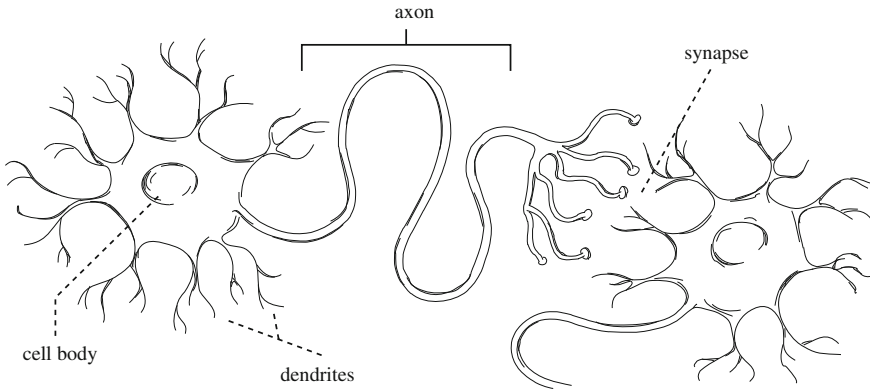


Fig. 2.14 A biological neuron consists of a cell body, a dendritic tree, and an axon. The space between the axon and the dendrites of the next neuron is called the synapse. The neuron receives signals on its dendrites, and transport them through the axon and into the synapse over to the next neuron

A neuron is typically composed of a cell body, a *dendritic tree*, and an *axon* (see Fig. 2.14). The neuron receives signals on the dendrites and releases (fires) signals through the axon. Connected neurons are separated by a small physical gap called a *synapse*. The information is carried through the system in the form of electrochemical pulses, or *action potentials*, that are passed on from neuron to neuron. A neuron can receive thousands of such pulses from different neurons, and each pulse may change the potential of the dendritic membrane, either by inhibiting or exciting the generation of further pulses. If the sum of these pulses exceeds a certain threshold the neuron “fires” by generating a new pulse that travels into the synapse and over to the next neuron.

2.4.2 Artificial Neurons and the Perceptron

An *artificial neuron* is, similarly to a biological neuron, composed by a cell body, dendrites, and an axon (see Fig. 2.15). The inputs, that are received through the dendrites, get integrated in some manner, and if the result exceeds a given threshold, the neuron transmits an output. Thus, an artificial neuron is simply a computational unit, that maps input values to one or more outputs. The computation is done in two steps: first the input values $\mathbf{x} = (x_1, \dots, x_N)$ are integrated into a single value a , through some *integration function* $a = h(\mathbf{x})$, and then this value is transformed by some nonlinear function g , called the *activation function*, to produce an output value $y = g(a) = g(h(\mathbf{x}))$.

The simplest kind of artificial neurons, first proposed by McCulloch and Pitts [22], uses binary values (0 or 1) both for inputs and the output. The integration function is

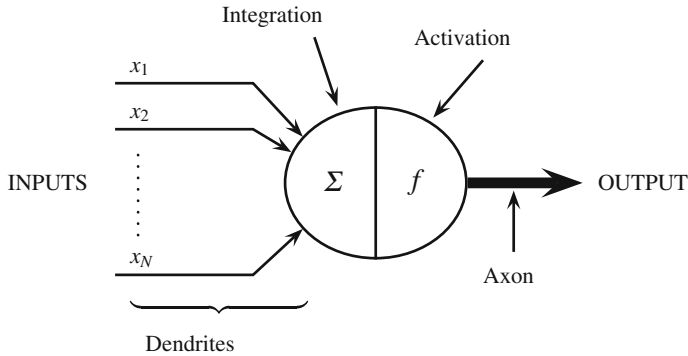


Fig. 2.15 An artificial neuron attempts to mimic a biological neuron, and consists of a cell body, dendrites, and an axon. The inputs are weighted and summed, before the activation function decides whether the neuron should fire or not

an unweighted sum of *excitatory* and *inhibitory* edges. If any of the inhibitory edges is 1, the neuron is inhibited and the output is 0.

Otherwise, if all inhibitory edges are 0, the integrated value is the sum of the excitatory edges

$$a = \sum_{i=1}^N x_i. \quad (2.107)$$

The activation function is the *Heaviside step function* (or *threshold function*)

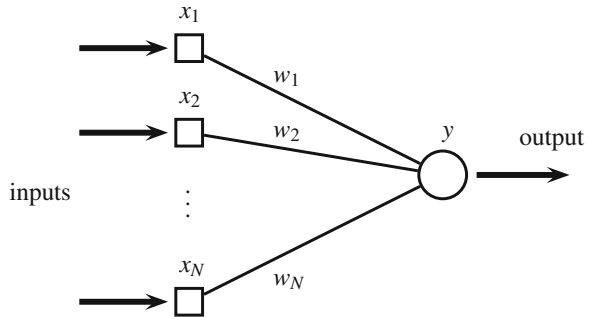
$$\phi(a) = \begin{cases} 1 & \text{if } a > \theta, \\ 0 & \text{otherwise,} \end{cases} \quad (2.108)$$

where θ is called the *threshold*, or the *bias*. If the integrated value a exceeds the threshold, and ϕ takes value 1, the neuron fires. Otherwise it takes the deactivated value 0.

However, although very useful for the computation of logical functions in finite automata, the McCulloch–Pitts neurons are rather limited. A generalization of the McCulloch–Pitts neuron, called the *perceptron*, was developed by Rosenblatt [33]. In its simplest form, it is basically the McCulloch–Pitts neuron with real-valued inputs and associated weights. The input values x_1, \dots, x_N , $x_i \in \mathbb{R}$, are fed into the node through edges with associated weights w_1, \dots, w_N . The integration function is the weighted sum

$$a = \sum_{i=1}^N w_i x_i, \quad (2.109)$$

Fig. 2.16 A single-layer network diagram



and the activation function is the same threshold function (2.108) as in the McCulloch–Pitts neuron. A network only consisting of one neuron like this, is sometimes called a *single-layer network*, because it consists of a single layer of weights (see Fig. 2.16). In analogy with the biological neuron, the inputs x_i represent the level of activity of the neurons connected to the current neuron, and the weights w_i signify the strengths of these connections.

A further generalization of the perceptron is to allow for more general activation functions. The activation function somehow determines how powerful the output of the neuron should be. While biological neurons choose between “fire” or “not fire”, mathematically it is more convenient with a smoother (differentiable) activation function.

A popular choice is the *logistic sigmoid* function

$$\phi(a) = \frac{1}{1 + e^{-\sigma(a+\theta)}}, \quad (2.110)$$

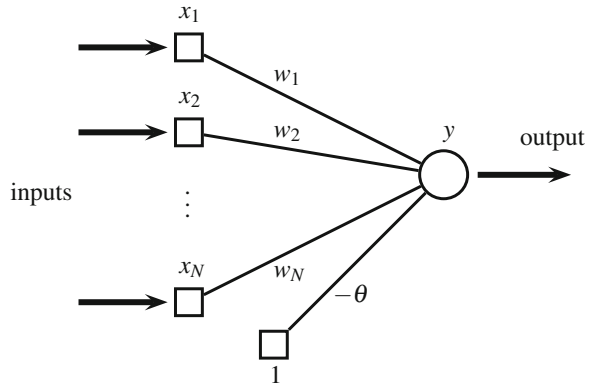
where θ is the bias that moves the curve away from zero, and σ a parameter that affects the steepness of the curve. The bias θ can be viewed as the number of pulses needed for the neuron to fire. Training a neural network involves estimating the values of the edge weights and of the bias parameter. For convenience it is common to invoke the bias into the network by adding an extra input variable $x_0 \equiv 1$, and an associated edge $w_0 = -\theta$ (see Fig. 2.17). Using this, and assuming $\sigma = 1$, gives the simpler form of the activation function

$$\phi(a) = \frac{1}{1 + e^{-a}}. \quad (2.111)$$

A small modification to (2.110) gives the ‘tanh’ activation function

$$\phi(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}, \quad (2.112)$$

Fig. 2.17 A single-layer network diagram with an added bias node



which is symmetric and therefore may achieve faster convergence of the training algorithms in some cases.

2.4.3 Multilayer Neural Networks

The architecture of a neural network is usually either *feed-forward* or *recurrent*. A feed-forward network is devoid of loops; it is a directed acyclic graph where the information moves in only one direction, from the input nodes, possibly through one or more hidden layers, and to the output nodes. The counterpart, recurrent networks, contain cycles. We will only consider feed-forward networks here, since almost all applications in computational biology use layered feed-forward network models.

A multilayer neural network is a further generalization of the single-layer network, where the network function is composed by several successive functions. In a network architecture, this can be seen as successive layers of nodes, or processing units, with connections running from the nodes in one layer to the next. A node can be either *hidden* or *visible*, where visible nodes are typically those connected to the outside world, such as the input and the output nodes, and the hidden nodes occupy layers in between. A layer that consists of only hidden nodes is called a *hidden layer*, and the total number of layers define the *depth* of the network. A layered network does not contain cycles, and usually each node in one layer is connected to each of the nodes in the next. Figure 2.18 illustrates a two-layer feed-forward network.

Note that we choose not to include the input layer when counting the depth of the network. This is because the input nodes are not really processing units, but only holders of the input values. With this convention, the depth corresponds to the layers of weights to be estimated from training data. Also, a multilayer network does not have to be fully connected as in Fig. 2.18; a more economical model would be preferred whenever possible.

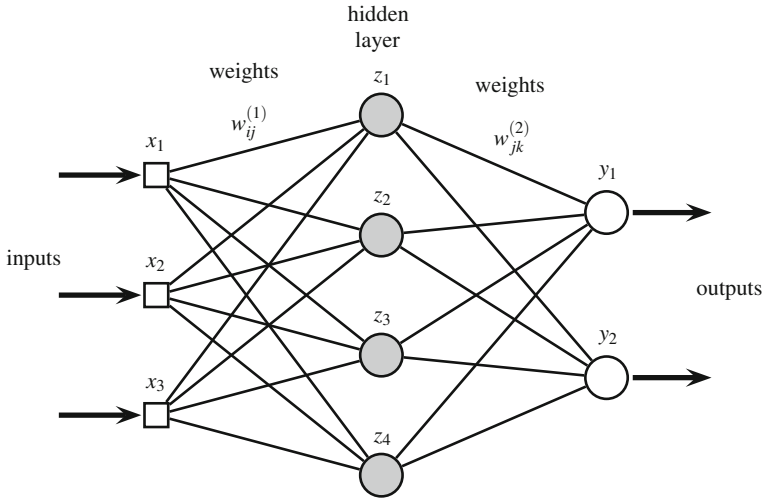


Fig. 2.18 A two-layer feed-forward network diagram

Consider a two-layer network with N input units (x_1, \dots, x_N) , a hidden layer of M hidden units (z_1, \dots, z_M) , and K output units (y_1, \dots, y_K) .

If ψ is the activation function of the hidden units and ϕ the activation of the output units, the network can be represented mathematically as

$$\begin{aligned}
 y_k &= \psi \left(\sum_{j=1}^M w_{jk}^{(2)} \cdot z_j \right), \quad k = 1, \dots, K \\
 &= \psi \left(\sum_{j=1}^M w_{jk}^{(2)} \cdot \phi \left(\sum_{i=1}^N w_{ij}^{(1)} x_i \right) \right). \quad (2.113)
 \end{aligned}$$

While it is possible to use different activation functions for different layers, it is common to use the same for all, such that $\psi = \phi$. A *multilayer perceptron* is a multilayer network, with either the threshold function (2.108) or the logistic sigmoid function (2.110) as activation function. The advantage with the sigmoid function is that it is differentiable, which enables the use of a very powerful training procedure called the *backpropagation algorithm* described in Sect. 6.7.

2.4.4 GRAIL: A Neural Network-Based Gene Finder

GRAIL [43, 44] is a neural network-based gene finder that scores potential exons by combining the scores of a number of content and signal sensors. Four types of

exons are recognized: initial, internal, terminal, and single exons. These exon types represent open reading frames in combination with their specific boundaries: start codon to donor site (initial), acceptor to donor site (internal), acceptor site to stop codon (terminal), or start to stop codon (single). The gene prediction is performed in four separate steps:

1. Extract all possible exon candidates.
2. Remove improbable exons.
3. Score remaining exons.
4. Construct gene models.

The first step is a preprocessing step, where all possible exons in the sequence are extracted. A candidate exon consists of an open reading frame surrounded by the corresponding exon boundaries. This first step produces a huge number of candidates, typically several thousands just in a sequence of 10,000 bp [43]. Thus, in the second step a number of heuristic rules are applied to remove improbable exons. In the third step, all remaining exon candidates are scored by a feed-forward neural network, which has been trained by the backpropagation algorithm described in Sect. 6.7. The input to the network is a feature vector of various coding measures and splice site scores for each exon candidate. In the fourth and final step, the scored exon candidates are combined into frame-consistent gene models.

The GRAIL neural network consists of 13 input nodes, two hidden layers with seven, and three nodes, respectively, and one output node. A network diagram is shown in Fig. 2.19. The hidden layer of seven nodes, not shown in the figure, is part of the splice site scoring. A mathematical representation of the network can be written as

$$y = \phi \left(\sum_{k=1}^3 w_k^3 \phi \left(\sum_{j=1}^7 w_{kj}^2 \phi \left(\sum_{i=1}^{13} w_{ji}^1 x_i \right) \right) \right), \quad (2.114)$$

where ϕ is the logistic activation function

$$\phi(x) = \frac{1}{1 + e^{-x}}. \quad (2.115)$$

The weights w are trained using the backpropagation algorithm.

During training, the output is evaluated using a matching function M , that measures the overlap of the candidate exon with the true exon(s),

$$M(\text{candidate}) = \frac{\sum_i m_i}{\text{length}(\text{candidate})} \frac{\sum_i m_i}{\sum_j \text{length}(\text{exon}_j)}, \quad (2.116)$$

where $\sum_i m_i$ is the number of bases of the candidate exon that overlap true exons, and $\sum_j \text{length}(\text{exon}_j)$ is the total length of all exons that overlap the candidate. Thus, $0 \leq M \leq 1$ with

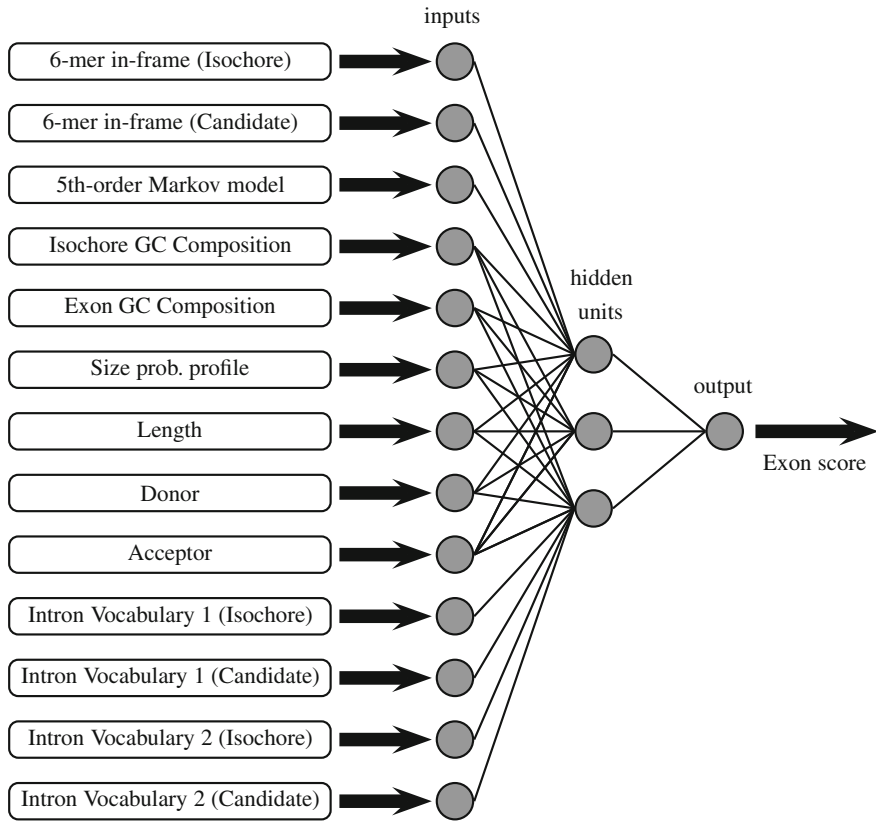


Fig. 2.19 The GRAIL neural network for scoring candidate exons. The network consists of 13 input nodes, two hidden layers of seven (not shown), and three nodes, respectively, and one output node delivering the final exon score. The figure is reproduced from [43], ©1996 IEEE

$$M = \begin{cases} 1 & \text{if prediction is correct} \\ 0 & \text{if no overlap with true exons.} \end{cases} \quad (2.117)$$

The feature vector fed into the GRAIL neural network consists of 13 measures for each candidate exon, including various coding measures and splice site measures [43]. Coding potential is scored using both a frame-dependent 6-tuple preference model and a fifth-order inhomogeneous Markov model. These measures are not independent, but by applying supervised learning (labeled training examples), the weights are adapted for all features together. The splice site detector in GRAIL is in itself a neural network, that combines the scores from several measures. Neural networks applied to splice site detection are described in more detail in Sect. 5.4.4.

2.5 Decision Trees

A *decision tree* is a kind of tree diagram that can be used to choose between different decisions for an object, by connecting series of tests on different features of the object. Decision trees are a common ingredient in clinical research, in which various features of the patient lie as ground for diagnosis into one of two or more clinical categories. Traditional statistical methods struggle in such situations, where the set of possible features may be large, or the interactions between the features are complex, or the feature values do not follow a known distribution. Moreover, the outcome of the analysis may be difficult to interpret, for instance if diagnosis is presented in terms of probabilities. An advantage of decision trees is that it enables the reduction of rather complex datasets into simple and comprehensible data structures. In addition, being a nonparametric technique, decision trees avoid the problems of making assumptions about the distribution.

Decision trees can be applied to classification problems, in which objects need to be classified into different classes based on a set of features, or attributes, that characterize the object. In this context decision trees are also called *classification trees*. Here we give a brief overview of the decision tree theory applied to single species gene finding. For a more thorough treatment, confer for instance the books by Breiman et al. [6] or Quinlan [29].

2.5.1 Classification

Decision trees can be used to classify an object based on a set of features that characterize the object. A decision tree consists of internal nodes and leaf nodes. The leaf nodes contain the class labels, and each internal node performs a test on one specific features. A new object is classified by passing it down from the root of the tree, through a series of tests on its features, finally ending up in one of the leaf nodes. In each node the corresponding feature is tested, and depending on the answer the object is passed down into one of its child nodes. The process is recursed until the object reaches a leaf node and receives its classification. In other words, given a set of features, a decision tree represents a series of rules that are used for classification of the corresponding object. The features can be of any type, binary, categorical, or numerical, while the class labels must be qualitative.

Using an existing decision tree for classification is easy. The trick to decision tree analysis is the actual construction of the tree, called *decision tree learning*, using a training set of objects with corresponding feature values and known class labels. Given a large set of possible features, decision tree learning techniques have been developed to choose both which features that are relevant, and in which order they are to be tested. Example 2.5 illustrates a simple dataset, borrowed from [28], containing only categorical feature values.

Table 2.4 A simple decision tree training set

Object	Features				Class
	Outlook	Temperature	Humidity	Windy	
1	Sunny	Hot	High	False	N
2	Sunny	Hot	High	True	N
3	Overcast	Hot	High	False	P
4	Rain	Mild	High	False	P
5	Rain	Cool	Normal	False	P
6	Rain	Cool	Normal	True	N
7	Overcast	Cool	Normal	True	P
8	Sunny	Mild	High	False	N
9	Sunny	Cool	Normal	False	P
10	Rain	Mild	Normal	False	P
11	Sunny	Mild	Normal	True	P
12	Overcast	Mild	High	True	P
13	Overcast	Hot	Normal	False	P
14	Rain	Mild	High	True	N

With kind permission from Springer Science + Business media: [28, p. 87, Table 1]

Example 2.5 A simple decision tree training set

The following example is borrowed from [28]. In this training set the observed objects are *Saturday mornings*. Suppose we use a number of different weather features to determine whether we will undertake a certain activity or not. The classification of the objects is thus either *P* or *N* for positive or negative instances, respectively, where a positive instance means that the activity will take place. The weather features and the corresponding values used are

Feature	Values
Outlook	Sunny, overcast, rain
Temperature	Cool, mild, hot
Humidity	Normal, high
Windy	True, false

The dataset is presented in Table 2.4. Given this training set we would like to build a decision tree that, based on the feature values of a new Saturday morning, can be used to determine whether the activity in question will happen or not. In the next section, we describe how to build such a decision tree from data. \square

2.5.2 Decision Tree Learning

Depending on which order the features are tested, there are many ways to build a complete decision tree from the same training set. By the principle of *Occam's Razor*, the shortest hypothesis should always be preferable. Or, in terms of decision trees, the tree that is optimal for a given dataset is the smallest one. However, creating an algorithm that, for a general set of features, always finds the smallest tree is an NP-complete problem, basically meaning that it cannot be solved in reasonable time. Therefore, numerous algorithms have been created that search for close to optimal trees, among the most noted ones being ID3 [28], C4.5 [29], and CART [6]. These algorithms typically use a greedy recursive procedure which, while creating reasonable trees, cannot guarantee to find the optimal solution. Such algorithms typically consist of the following basic steps:

1. Determine the feature that best splits the data.
2. For each *pure* subset (all of the same class), create a leaf node with that class. For each impure subset, return to 1.
3. Stop when no more splits are possible and all paths end with a leaf node.

We call a set of objects *pure* if all objects belong to the same class, and *impure* otherwise. For instance, for a given feature, we can group the objects according to their feature values. If that grouping corresponds completely with the grouping according to class label, it represents a *pure split* of the dataset.

Which feature that best splits the data is determined using some kind of *measure of impurity*. A popular measure, for instance used by the ID3 algorithm, is the *Shannon entropy*, or simply *entropy*. Suppose that we have a training set D of n objects each characterized by a set of features A_1, \dots, A_p , and each with a known class label $c_i \in C, i = 1, \dots, n$, where C is the set of all classes. The entropy of such a set can be written as

$$H(D) = - \sum_{c \in C} p_c \log_2 p_c, \quad (2.118)$$

where the sum runs over all possible classes, and where p_c is the probability of belonging to class $c \in C$. The entropy basically measures the uncertainty, level of randomness, or *information content* of the dataset. The more uniform the distribution is, the higher the entropy. The base 2 of the logarithm transforms the value into “bits” commonly used in information theory. The entropy assigns measure zero to pure sets and reaches its maximum when all classes have equal probabilities. Alternative impurity measures include the *Gini index* and the *twoing rule*.

$$\text{Gini} = 1 - \sum_c p_c^2, \quad (2.119)$$

$$\text{Twoing} = \frac{|T_L||T_R|}{n^2} \left(\sum_{c \in C} \left| \frac{L_c}{|T_L|} - \frac{R_c}{|T_R|} \right| \right)^2, \quad (2.120)$$

where, for a split at node T containing n objects, $|T_L|$ and $|T_R|$ are the numbers of objects to the left and to the right of the split, respectively, and L_c and R_c are the numbers of objects having class label c to the left and the right of the split, respectively. The Gini index chooses the split attempting to separate as large a class from the rest as possible, while the twoing rule attempts to split the data as central as possible. Which splitting rule that works best depends on the application (cf. [5, 6]).

The feature that best splits the training data is the one that causes the largest decrease in impurity. The goal is to create descendant subsets that are purer than its parents. This decrease in impurity is calculated using a measure called the *information gain*: for a set D of n objects the information gain of splitting over a specific feature A is given by

$$IG(D, A) = H(D) - \sum_{v \in A} \frac{|D_v|}{|D|} H(D_v), \quad (2.121)$$

where the sum runs over all possible feature values of A , D_v is the set of objects in D that take value v for feature A , and $|D_v|$ and $|D|$ denote the numbers of objects in each set (i.e., $|D| = n$). The second part of (2.121) in fact corresponds to an entity known as the *conditional entropy* $H(D|A)$ of D , given the attribute values of A .

Now we can calculate the information gain of splitting the dataset in each of the features. Then the feature with the highest information gain is chosen to be tested first and the test is placed in the root of the tree. Branches are created for each possible value of the feature, the dataset is split into subsets according to their values on the chosen feature, and the procedure is repeated in the child nodes.

Example 2.6 A simple decision tree training set (cont.)

We illustrate how the decision tree for the data in Table 2.4 is built using entropy and information gain. First, in order to calculate the entropy $H(D)$ in (2.118) of the entire dataset, we estimate the class probabilities by the relative frequencies for class labels P and N :

$$p_P = 9/14, \quad p_N = 5/14.$$

Thus, the entropy becomes

$$H(D) = -(9/14) \log_2(9/14) - (5/14) \log_2(5/14) \approx 0.940.$$

Next, if we were to split the data according to attribute ‘Outlook’, we would split the dataset into groups according to the feature values ‘sunny’, ‘overcast’, or ‘rain’.

Outlook	Class	Outlook	Class	Outlook	Class
Sunny	N	Overcast	P	Rain	P
Sunny	N	Overcast	P	Rain	P
Sunny	N	Overcast	P	Rain	P
Sunny	P	Overcast	P	Rain	N
Sunny	P			Rain	N

The entropies of the subsets become

$$\text{Sunny: } H(D_v) = -(3/5) \log_2(3/5) - (2/5) \log_2(2/5) \approx 0.971$$

$$\text{Overcast: } H(D_v) = -(4/4) \log_2(4/4) = 0$$

$$\text{Rain: } H(D_v) = -(3/5) \log_2(3/5) - (2/5) \log_2(2/5) \approx 0.971$$

The resulting information gain for ‘Outlook’ thus becomes

$$IG(D, \text{Outlook}) = 0.940 - \left(\frac{5}{14} \cdot 0.971 + \frac{4}{14} \cdot 0 + \frac{5}{14} \cdot 0.971 \right) \approx 0.247.$$

Similarly, we get for the other features

$$IG(D, \text{Temperature}) \approx 0.029$$

$$IG(D, \text{Humidity}) \approx 0.152$$

$$IG(D, \text{Windy}) \approx 0.048$$

We see that ‘Outlook’ achieves the highest information gain and we therefore place it in the root node. We draw three branches from this node, one for each of the values of ‘Outlook’, and continue. Next we note that the ‘overcast’ group is pure (all objects have label P), and we therefore insert a leaf node with class label P . The other two subsets are impure and need to be split further. The information gain is now calculated over the corresponding subsets of objects. For instance, the subset ‘sunny’ now contains $n = 5$ objects, and the information gain is calculated for the features ‘Temperature’, ‘Humidity’, and ‘Windy’ for this subset,

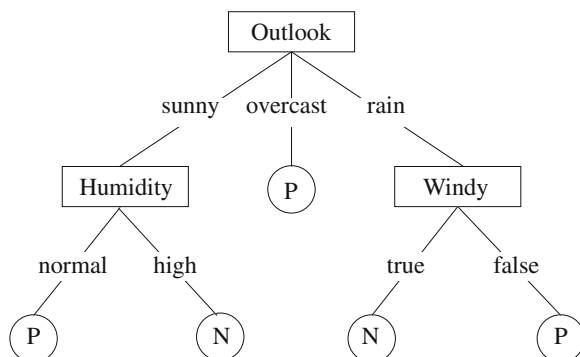
$$IG(D_{\text{sunny}}, \text{Temperature}) \approx 0.571$$

$$IG(D_{\text{sunny}}, \text{Humidity}) \approx 0.971$$

$$IG(D_{\text{sunny}}, \text{Windy}) \approx 0.020$$

Humidity achieves the highest information gain for this subset, and is placed in the corresponding node. The procedure continues until all subsets are pure and can be finished off with leaf nodes. The resulting tree is shown in Fig. 2.20. Note that the feature ‘Temperature’ is never used. The ‘Temperature’ feature is very impure, meaning that it has very weak (if any) association with the classification, and the tree reaches its leaf nodes without having to take that feature into consideration. \square

Fig. 2.20 The resulting tree of the data in Table 2.4. With kind permission from Springer Science + Business media: [28, p. 87, Fig. 2]



The resulting decision tree classifies the objects in the training set perfectly. The risk is, however, that the tree is too specific to the training set, and will not be able to correctly classify new objects presented to it. This problem is known as *overfitting*, and is commonly solved by some kind of *pruning* procedure. Pruning basically means that parts of the tree will be cut off by turning internal nodes into leaf nodes. This makes the tree less specific to the training set, but more flexible to new data. Confer for instance [6, 29] for more details.

We have treated only categorical or binary feature values so far, but the feature values are allowed to be numerical as well. The node tests in the decision tree would then typically involve inequalities such as $x_i \geq 4.2$ versus $x_i < 4.2$, or possibly separation into several subintervals. There are many different methods for dealing with numerical values, but most of them involve discretizing the data in some manner in order to treat them as categorical values. A rather different treatment is introduced by the OC1 algorithm [24], used by the MORGAN gene finder presented next. OC1 does not split the data for one specific feature, but uses linear combinations of the feature values to determine the best decision tree.

2.5.3 MORGAN: A Decision Tree-Based Gene Finder

MORGAN (Multi-frame Optimal Rule-based Gene ANalyzer) [34] is a gene finder that combines decision trees with dynamic programming and signal sensor algorithms. The dynamic programming algorithm is used to search through all possible parses of the sequence, while the decision tree algorithm and the signal sensors provide scores of the different parts of the potential gene. The decision trees are built using the OC1 system [24], which uses something called *oblique tests* in the decision tree nodes. In order to estimate probabilities of a potential exon or intron, the OC1 also includes a random component which means that it can produce different trees for the same data each time it is run.

Before the MORGAN system can be trained, the training set, consisting of raw genomic DNA sequences with known exons and introns, is transformed into the form of objects, class labels, and features. This is done by first identifying all potential start, stop, donor, and acceptor sites, scoring above a certain threshold, in the training sequences. Next, candidate exons are identified by combining the corresponding boundary sites (start-donor for initial exons, acceptor-donor for internal exons, and acceptor-stop for terminal exons), and requiring an open reading frame (ORF) between the sites. Similarly, potential introns are identified by pairing up donor and acceptor sites, with an additional length constraint (between 20 and 16,000 bp), but without the ORF requirement. For each of the three types of exons and the intron, a decision tree is constructed. Since the true exons and introns are known in the training set, the identified candidate exons and introns receive a label that is either ‘true’ or ‘false’. Thus, the objects are the potential exons and introns, and the class labels are ‘true’ or ‘false’ revealing which objects are real or not.

The features used by MORGAN to characterize the objects include boundary site scores, an in-frame hexamer statistic, and a position asymmetry statistic. The signal sensors used to score the boundary sites are a first-order Markov model for the start sites based on the *Kozak sequence* (see Sect. 5.3.2), and second-order Markov models for the splice sites. Since no consensus sequence is known for the sequence surrounding the stop sites, the stop codons are simply identified directly. These type of submodels are discussed further in Chap. 5.

The in-frame hexamer statistic for a subsequence between positions i and j in the sequence is given by

$$IF_6(i, j) = \begin{cases} \sum_{k=0,3,6,\dots,j-6} \log(f_k / F_k) \\ \sum_{k=1,4,7,\dots,j-6} \log(f_k / F_k) \\ \sum_{k=2,5,8,\dots,j-6} \log(f_k / F_k) \end{cases} \quad (2.122)$$

where f_k is the frequency of the hexamer starting in position k in coding sequences, and F_k is the frequency of the hexamer among all hexamers in the training set, in all reading frames [41]. The position asymmetry statistic, presented in [12], counts the frequency of each nucleotide in each of the three codon positions.

The OC1 system [24], used to build the decision trees, is specifically designed to handle numerical feature values. OC1 does not split the data according to their feature values, but uses *linear discriminant* kind of tests, where, instead of using interval tests such as $x_i \geq 4.2$, each internal node contains a linear combination of one or more features,

$$a_1x_1 + a_2x_2 + \dots + a_px_p \geq a_{p+1}. \quad (2.123)$$

Since this linear combination represents a hyperplane that is nonparallel to the axes in feature space, this is called an *oblique split*.

After the decision tree is built, OC1 prunes the tree using a method called *complexity pruning* [6]. Basically, a complexity measure is calculated for each internal

node based on the number of misclassifications that would result on the training set if that node were turned into a leaf, combined with the size of the subtree rooted at that node. The node with the largest complexity measure is then turned into a leaf. The series of increasingly smaller trees are then tested on a separate part of the training set, and the tree with the highest accuracy on this set is kept as the output of the system.

2.6 Conditional Random Fields

In gene prediction we want to connect the observed sequence data to a sequence of labels corresponding to the underlying gene model. A successful approach to this has been to employ hidden Markov models (HMMs), described earlier in this chapter. One disadvantage with HMMs, however, is that in order to make computations feasible, two rather strong independence assumptions have to be made: (i) given the current state (i.e., current sequence label), the next state is conditionally independent of everything else, and (ii) the observed output from each state only depends on the underlying state. With these assumptions, the HMM machinery comes together very nicely, but often the observed sequence include complex interdependencies that when ignored may significantly hurt classification performance. Conditional random fields (CRFs) [19] were developed mainly to fill this gap. CRFs offer an alternative to HMMs, where, instead of making simplifying assumptions, the model is extended to include interdependence features. The cost of this added flexibility, however, is increased computational complexity and a less straightforward interpretation of the parameters. This section gives a brief encounter of CRFs, in the context of computational gene prediction. More general and detailed descriptions can be found for instance in [19, 42].

2.6.1 Preliminaries

We recall from Sect. 2.1.1 that a *random process* is a collection of random variables that is indexed by some ordered set T . Such a collection can typically be used to model the evolution of a system or the development of a physical process over time, where the system or process switches randomly between *states*, or *phases*. If the index set T is ordered it is often referred to as “time”, and the indexed collection of random variables can be lined up as in a chain of events. Hidden Markov models (HMMs), described in Sect. 2.1, are a special kind of random processes, that consist of two interrelated process: a Markov process that is hidden from the observer, corresponding to the state labels we want to predict (e.g., exons, introns, intergene, etc.), and an observed process corresponding to the observed output we wish to annotate (e.g., the DNA sequence).

A *random field* is a generalization of random processes where the process evolves in a multidimensional space, and the time index is replaced by a corresponding multidimensional coordinate vector. Random fields are useful for instance to model spatial data such as the pixels in image analysis, where both the position and the value (*attribute*) of the process are of interest. As for random processes, there are many kinds of random fields, but a family of models relevant to this section are the *Markov random fields*, also known as *Markov networks*. A Markov random field is a collection of random variables having a similar Markov property as for Markov chains, that can readily be described by an *undirected graph*. Basically, the Markov property for random fields state that given the neighbors in the graph, a random variable is conditionally independent of everything else. Markov random fields are similar to *Bayesian networks*, described in Sect. 5.4.7, in how the dependency structure is represented. The difference is that Bayesian networks are directed and acyclic, while Markov random fields are undirected and possibly cyclic. A *conditional random field* (CRF) is an extension of Markov random fields in the same manner as an HMM is an extension of a Markov chain. That is, a CRF is a Markov random field in which each random variable can be conditioned upon a set of global observations.

2.6.2 Generative Versus Discriminative Models

Before we move on we need to introduce some new notation. In the HMM framework described earlier in this chapter, the hidden label sequence is denoted \mathbf{X} and the observed sequence \mathbf{Y} . In the CRF community, however, this notation is usually switched. Therefore, to avoid confusion, throughout this section the observed sequence, also called the *input* sequence, is denoted \mathbf{O} , and the hidden *output* sequence is denoted \mathbf{H} .

Generative models is a family of models where the joint probability of the hidden and the observed sequence can be factorized as

$$\mathbf{P}(\mathbf{O}, \mathbf{H}) = \mathbf{P}(\mathbf{O})\mathbf{P}(\mathbf{H}|\mathbf{O}). \quad (2.124)$$

A generative model thus allows us to draw samples from it, in order to “generate” synthetic examples of the observed sequence given the hidden. However, due to high dimensionality and complex dependencies the distribution of the observed sequence may be difficult to render, which is why numerous independence assumptions often need to be made to make the computations tractable. *Discriminative models*, on the other hand, is a family of conditional distributions $\mathbf{P}(\mathbf{H}|\mathbf{O})$ where the hidden sequence to be classified is modeled directly. The distribution of the observed sequence is ignored and thereby the need for independency assumptions on the observed sequence is avoided. By supplying a model for the marginal distribution of the observed sequence, the conditional distribution of the discriminative model could be used to compute the joint distribution as in (2.124), but since the conditional distribution is all we need for classification, this is usually not done.

In this manner, there are *generative-discriminative* model pairs, where one model can be converted into the other using *Bayes' rule* (see Sect. 5.4.7). One such pair is the *naive Bayes classifiers* and the *logistic regression*. Assume that we want to determine a single classification label H , based on a vector of observations or *features* $\mathbf{O} = (O_1, O_2, \dots, O_n)$. The naive Bayes classifier is based on the joint probability of the classification label and the observations, which can be factorized as

$$\mathbf{P}(H, \mathbf{O}) = \mathbf{P}(H) \prod_{i=1}^n \mathbf{P}(O_i | H). \quad (2.125)$$

The logistic regression classifier is instead based on the conditional probability and assumes that the logarithm of the conditional distribution, $\log \mathbf{P}(H | \mathbf{O})$, is a linear function of \mathbf{O} , such that

$$\mathbf{P}(H | \mathbf{O}) = \frac{1}{Z(\mathbf{O})} \exp \left(\theta_H + \sum_{i=1}^n \theta_{H,i} O_i \right) \quad (2.126)$$

where $Z(\mathbf{O})$ is a normalization factor and θ_H is a *bias weight* corresponding to the initial $\log \mathbf{P}(H)$ component in the naive Bayes formula in (2.125). To write this in more compact form we can define *feature functions* that are indicator functions for a single class only. That is, we let $f_{H',j}(H, \mathbf{O}) = \mathbf{1}_{\{H'=H\}} O_j$ represent the feature weights and $f_{H'}(H, \mathbf{O}) = \mathbf{1}_{\{H'=H\}}$ the bias weights. By instead using a common index k for all different feature functions f_k and their corresponding weights θ_k , the logistic regression model can be written as

$$\mathbf{P}(H | \mathbf{O}) = \frac{1}{Z(\mathbf{O})} \exp \left(\sum_{k=1}^K \theta_k f_k(H, \mathbf{O}) \right). \quad (2.127)$$

By training the naive Bayes classifier in (2.125) to maximize the conditional likelihood, we achieve the logistic regression classifier, and if the logistic regression classifier is trained to maximize the joint distribution we achieve the naive Bayes. In a similar manner, HMMs and CRFs are a generative-discriminative pair, and for suitable choices of feature functions in the CRFs we can convert one model into the other.

An important note is that while the two models in a generative-discriminative pair exactly mirror one another in theory, this is rarely true in practice. In order for this to hold we need access to the true distributions, but in practice we are usually left to work with estimations and approximations resulting from only having samples of the true distributions. Therefore, it matters which model we choose, generative or discriminative, and the choice for a given application may not be obvious as both approaches have their pros and cons. If we focus merely on the classification task, discriminative models can be highly superior, both in terms of computational complexity and in terms of the level of dependencies they can include. They impose

conditional independence assumptions on the hidden sequence pretty much in the same manner as in generative models, and they describe how the hidden sequence may depend on the observed, while interdependencies within the observed sequence need not be explicitly stated. This way discriminative models can include very complex dependencies and overlapping features which may improve the classification accuracy. However, generative models are usually more flexible, in particular when it comes to training, and are more easily interpreted. Also, generative models are better at handling missing, latent, or partially labeled data, and can sometimes perform better than a discriminative model as a result. Therefore, which approach to use has to be guided by the application in question [3, 21].

2.6.3 Graphical Models and Markov Random Fields

In many statistical applications we have prior knowledge about the ordering of a set of variables, either of the temporal ordering of events or in terms of dependency structures. Such knowledge can often be illustrated in a graphical model $G = (V, E)$, where V are the vertices and E the connecting edges. The vertices correspond to the random variables and the edges represent the dependency structure between these variables. Graphical models can be divided into two main classes: *directed acyclic graphs* (DAGs) and *undirected graphs*. Two important models for our purposes are *Bayesian networks* which are a kind of DAGs, described in Sect. 5.4.7, and *Markov random fields*, which are undirected graphs that will be discussed a little further in this section. For a more comprehensive treatment on graphical models and random fields, see for instance [23].

We say that random variables A and B are *conditionally independent* given a third random variable C if and only if

$$\mathbf{P}(A, B|C) = \mathbf{P}(A|C)\mathbf{P}(B|C), \quad A, B, C \in V. \quad (2.128)$$

Conditional independence is a powerful concept as it can be used to factorize complex multivariate distributions into products of factors acting on smaller subsets of the random variables. Any joint distribution of a set of random variables can be represented by a DAG, where the edges correspond to conditional dependencies between the variables, and the absence of an edge implies conditional independence between the variables of the corresponding vertices.

Now, let $\mathbf{X} = (X_v)_{v \in V}$ be a collection of random variables. Recall from (2.1) that the probability of any such set and for any ordering can be decomposed into a product of conditional probabilities

$$\mathbf{P}(\mathbf{X}) = \mathbf{P}(X_1) \prod_{v=2}^V \mathbf{P}(X_v | X_1, \dots, X_{v-1}). \quad (2.129)$$

For a graph $G = (V, E)$, if there exists an ordering v_1, \dots, v_d of the vertices (i.e., of the random variables) that is consistent with the graph, meaning that a directed edge $v_i \rightarrow v_j \in E$ implies the ordering $i < j$, then G is called *directed acyclic graph* (DAG). We define the *parents* $\pi(v)$ of a vertex $v \in V$ as the set of vertices having a directed edge to v . A *directed model* is then a family of distributions that factorize as

$$\mathbf{P}(\mathbf{X}) = \prod_{v \in V} \mathbf{P}(X_v | \mathbf{X}_{\pi(v)}) \quad (2.130)$$

where X_v is the random variable at vertex v and $\mathbf{X}_{\pi(v)}$ is the set of random variables of the parent vertices of v . Because of the recursiveness in this decomposition, the resulting graph is *acyclic*, meaning that it does not contain any loops, resulting in a DAG (see Fig. 2.21a for an illustration). A common family of directed acyclic models are Bayesian networks, described in Sect. 5.4.7, and hidden Markov models and neural networks described earlier in this chapter can both be considered special cases of Bayesian networks.

In Markov random fields, on the other hand, the underlying graph is *undirected* and may be cyclic, representing a correlation between the random variables rather than a causality. An *undirected graph* is a graph where the edges have no direction. That is, for two vertices $i, j \in V$ the edges $\langle i, j \rangle$ and $\langle j, i \rangle$ are equivalent. Since there is no direction of the edges, there is no ordering of the random variables, meaning that the distribution can no longer be factorized according to a set of parents as in (2.130). Instead, an undirected graph can represent a family of distributions that each factorize according to a set of *factors*. A factor can be any strictly positive, real-valued function, and do not necessarily correspond to a conditional probability, which is why we also need a normalization factor to achieve a proper probability distribution. Formally, given a set of random variables \mathbf{X} and a collection of A subsets $\{\mathbf{X}_a\}_{a=1}^A$, an undirected graphical model is the set of distributions that can be written as

$$\mathbf{P}(\mathbf{X}) = \frac{1}{Z} \prod_{a=1}^A \psi_a(\mathbf{X}_a) \quad (2.131)$$

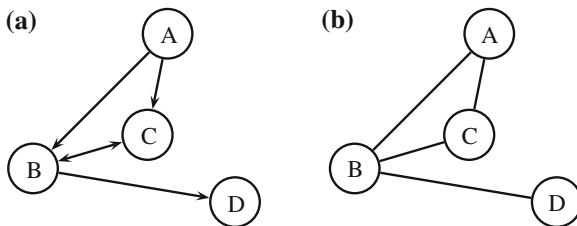


Fig. 2.21 An illustration of a graphical representation of four random variables, A, B, C, D . **a** A directed acyclic graph where the joint distribution factorizes as $\mathbf{P}(A, B, C, D) = \mathbf{P}(A)\mathbf{P}(B|A, C)\mathbf{P}(C|A, B)\mathbf{P}(D|B)$. **b** The corresponding undirected cyclic graph. Each node with its former parents form a complete subgraph of the graph

for any choice of positive factors $\Psi_a(\mathbf{X}_a) > 0$ for all \mathbf{X}_a . The constant Z , also known as the *partition function*, is a normalization factor

$$Z = \sum_{\mathbf{X}} \prod_{a=1}^A \Psi_a(\mathbf{X}_a), \quad (2.132)$$

where the sum runs over all possible assignments to the set \mathbf{X} . The factors Ψ_a are also called *local functions*, because they act on local subsets of the graph vertices, or *compatibility functions*, because they represent how compatible the values in a subset \mathbf{X}_a are with each other.

There is a clear connection between directed and undirected graphs. To see this, assume that the distribution of \mathbf{X} factorizes with respect to an undirected acyclic graph. Instead of talking about the parents of a vertex as in directed graphs, we now talk about the *neighbors* $n(v)$ of $v \in V$, meaning all vertices connected to v by an edge. In a directed graph a random variable X_v is conditionally independent of all predecessors in the graph, given its parents $\pi(v)$. In undirected graphs the corresponding conditional independence structure is represented by simple graph separation. It may be tempting to think that a DAG can be converted into an undirected graph simply by dropping the direction of the edges as in Fig. 2.21, but this does not hold in general. A v-shape in a DAG with edges $A \rightarrow B \leftarrow C$ would in the undirected graph result in a structure $A - B - C$ where A and C are conditionally independent given B , which clearly is not true. Instead we need to add an edge between A and C in the undirected graph to indicate their connection. This way of linking “unmarried” nodes is called *moralization*. Unfortunately, we lose some information of the DAG in the process, and we therefore cannot move in the other direction, creating a DAG from an undirected graph.

A special type of conditional independence structure is given by a Markov property formulation, similar to that of Markov chains, and that can be defined at three different levels: the *global Markov property*, the *local Markov property*, and the *pairwise Markov property*. The *global Markov property* of an undirected graph states that any two subset of random variables are conditionally independent given a separating subset. That is, for three subsets of vertices $A, B, C \subset V$ we say that \mathbf{X}_A is conditionally independent of \mathbf{X}_B given \mathbf{X}_C if and only if the vertices in C separates those in A from those in B . In essence, this means that if we remove all the vertices in C from the graph, the sets A and B are no longer connected. The *local Markov property* states that a random variable X_v is conditionally independent of all other variables in the graph, given its neighbors, and the *pairwise Markov property* states that two random variables not connected by an edge, are conditionally independent given everything else. The global property implies the local, which in turn implies the pairwise property. However, if we add the assumption that the distribution of the random variables is *positive*, meaning that $\mathbf{P}(X_v) > 0$ for all $v \in V$, we achieve equivalence between the three Markov properties. A *random field* is a generalization of random processes in which a collection of random variables are indexed by a multidimensional space. In a *Markov random field* the index space is an

undirected graph $G = (V, E)$ that fulfills the local Markov property. That is, for each vertex $v \in V$, given its neighbors $n(v)$ the corresponding random variable X_v is conditionally independent of everything else. That is,

$$\mathbf{P}(X_v | \mathbf{X} \setminus X_v) = \mathbf{P}(X_v | \mathbf{X}_{n(v)}), \quad v \in V \quad (2.133)$$

where $\mathbf{X} \setminus X_v$ denotes all variables in \mathbf{X} except X_v .

The conditional independences of an arbitrary distribution can be difficult to sort out, and a convenient subclass of Markov random fields are those that use the *maximal cliques* of the graph as the factorization subsets. A *clique* is a subgraph of G that is fully connected, meaning that there is an edge between every pair of vertices in the subgraph. Furthermore, a *maximal clique* is a clique that cannot be extended further without breaking the full connectedness property. The set of factors operating on the maximal cliques of G are called *potential functions*. A joint distribution, factorized by its maximal cliques, is then proportional to the product of the potential functions. The *Hammersley–Clifford theorem* [14, 23] gives that any positive distribution that satisfies the local Markov property can be factorized according to its maximal cliques. Such a Markov random field is sometimes called a *Gibbs random field*, which is popular in statistical physics because it can be represented by a *Gibbs distribution* for appropriate potential functions. A *Gibbs distribution* is a measure that factorizes over the maximal cliques \mathcal{C} of the undirected graph G , and where the distribution takes the log-linear form

$$\mathbf{P}(\mathbf{X}) = \frac{1}{Z} \exp(-H(\mathbf{X})). \quad (2.134)$$

where $H(\mathbf{X}) > 0$ is called the *energy function* of configuration \mathbf{X} . The meaning of an energy function can be somewhat abstract, but it relates to the energy used to describe the organization of atoms in thermodynamical systems. For instance, the more ordered the atoms in a metal are, the lower the energy (see Sect. 3.2.8 for more on this). In the Gibbs distribution, the factors in (2.131) thus take the form

$$\Psi_c(\mathbf{X}_c) = \exp(-H(\mathbf{X}_c)) \quad (2.135)$$

where H is the energy of the subset of random variables in clique $c \in \mathcal{C}$, and the energy function $H(\mathbf{X})$ sums over the maximal cliques \mathcal{C} . We can now give a more formal statement of the Hammersley–Clifford theorem:

Theorem 2.5 (The Hammersley–Clifford Theorem) *A positive distribution is a Markov random field if and only if it is a Gibbs random field.*

An important note is that although the maximal clique factorization corresponds to the conditional independence structure of the graphical model, the potential functions in themselves do not necessarily have a probabilistic interpretation. They merely represent constraints on the underlying random variables, which in turn effect the global probability distribution, but that do not directly translate into probabilistic terms.

2.6.4 Conditional Random Fields (CRFs)

A *conditional random field* (CRF) is a Markov random field where each random variable in the field may also be conditioned upon a set of global observations [19]. A CRF can be seen as an extension of logistic regression where the hidden variables are conditioned on the observed sequence. CRFs are also closely related to the hidden Markov models (HMMs) described earlier in this chapter. In fact, for a suitable choice of clique potentials, HMMs and CRFs form a generative-discriminative model pair in the same way as naive Bayes and logistic regression discussed above. The main difference between HMMs and CRFs is that while HMMs model the joint distribution $\mathbf{P}(\mathbf{H}, \mathbf{O})$ of an observed input sequence \mathbf{O} and a hidden output \mathbf{H} and (recall that we changed the HMM notation from (\mathbf{X}, \mathbf{Y}) to (\mathbf{H}, \mathbf{O})), CRFs focus on the conditional distribution $\mathbf{P}(\mathbf{H}|\mathbf{O})$ of the hidden sequence, given the observed. Thus, in the conditional setting, the observed distribution $\mathbf{P}(\mathbf{O})$ does not need to be modeled explicitly, leading to a simpler model which can allow the inclusion of complex dependencies within the observed sequence. Another advantage is that the potential functions can depend on the data, for instance by incorporating global features into the local potential functions, something that is very hard to do in generative models. The main disadvantage of CRFs is that they need to be trained on labeled data and the training process is typically very computer intense. General graphs typically become intractable fast, while graphs with a chain or tree structure may still be manageable.

The choice of potential functions for CRFs is closely related to the maximum entropy method described in Sect. 5.4.6. In order to include interdependencies within the observed sequence as well as other local and global knowledge of the data, we define a set of input *features*. A CRF is then a Markov random field where the clique potentials are conditioned on this feature set, denote it \mathbf{K} ,

$$\mathbf{P}(\mathbf{H}|\mathbf{O}, \mathbf{K}) = \frac{1}{Z(\mathbf{O}, \mathbf{K})} \prod_{c \in \mathcal{C}} \psi_c(\mathbf{H}_c|\mathbf{O}, \mathbf{K}). \quad (2.136)$$

Finding the maximum entropy distribution that satisfies K features f_k is an optimization problem under constraints, and if we choose log-linear clique potentials as in logistic regression we get for clique $c \in \mathcal{C}$

$$\psi_c(\mathbf{H}_c|\mathbf{O}) = \exp\left(\sum_{k=1}^K \lambda_{ck} f_{ck}(\mathbf{H}_c, \mathbf{O})\right) \quad (2.137)$$

where λ_{ck} is the Lagrangian multiplier associated with feature f_{ck} . Note also that we can make the features clique specific. The resulting CRF distribution becomes

$$\mathbf{P}(\mathbf{H}|\mathbf{O}) = \frac{1}{Z(\mathbf{O})} \exp\left(\sum_{c \in \mathcal{C}} \sum_{k=1}^K \lambda_{ck} f_{ck}(\mathbf{H}_c, \mathbf{O})\right). \quad (2.138)$$

Linear-Chain CRFs

For sequence models the common choice is a *linear-chain CRF*, which models the correlation between adjacent hidden variables in a linear sequence similarly to HMMs. In a linear graph each vertex only has two neighbors, and the maximal cliques simply constitute each pair of adjacent vertices connected by an edge. Therefore, we can define the clique potentials on the edges $e = \langle i - 1, i \rangle$ instead on vertex subsets

$$\Psi_e(H_e) = \exp \left(\sum_{k=1}^K \lambda_k f_k(H_{i-1}, H_i, \mathbf{O}) + \sum_{k=1}^K \mu_k g_k(H_i, \mathbf{O}) \right) \quad (2.139)$$

where f_k are feature functions of the local transitions and the global observed sequence, and g_k feature functions of the sequence label at position i and the observed sequence. The features f_k and g_k are, thus, closely connected to the transitions and emissions in an HMM. In fact, by choosing features exactly corresponding to the logarithm of the transition and emission probabilities in an HMM, the conditional distribution $\mathbf{P}(\mathbf{H}|\mathbf{O})$ rendered from the joint distribution $\mathbf{P}(\mathbf{H}, \mathbf{O})$ in an HMM, is a CRF. That is, by rewriting the joint distribution as

$$\mathbf{P}(\mathbf{O}, \mathbf{H}) = \frac{1}{Z} \prod_{t=1}^T \exp \left(\sum_{i,j \in S} \theta_{ij} \mathbf{1}_{\{H_t=j\}} \mathbf{1}_{\{H_{t-1}=i\}} + \sum_{i \in S} \sum_{v \in V} \mu_{iv} \mathbf{1}_{\{H_t=i\}} \mathbf{1}_{\{O_t=v\}} \right) \quad (2.140)$$

and defining the parameters as

$$\begin{aligned} \theta_{ij} &= \log \mathbf{P}(H_t = j | H_{t-1} = i) \\ \mu_{iv} &= \log \mathbf{P}(O_t = v | H_t = i) \\ Z &= 1 \end{aligned} \quad (2.141)$$

we achieve a direct correspondence between the HMM and the related CRF. To see this, by using a generic notation f_k for features and θ_k for the corresponding parameters, we transfer the the formula in (2.140) into

$$\mathbf{P}(\mathbf{O}, \mathbf{H}) = \frac{1}{Z} \prod_{t=1}^T \exp \left(\sum_{k=1}^K \theta_k f_k(H_{t-1}, H_t, O_t) \right). \quad (2.142)$$

The conditional distribution achieved by using (2.142) in

$$\mathbf{P}(\mathbf{H}|\mathbf{O}) = \frac{\mathbf{P}(\mathbf{O}, \mathbf{H})}{\sum_{\mathbf{O}'} \mathbf{P}(\mathbf{O}', \mathbf{H})} \quad (2.143)$$

is then a special type of linear-chain CRFs that only include features for the transitions and emissions modeled by a standard HMM. However, general linear-chain CRFs are not limited to the use of indicator functions, but can use any real-valued set of functions in place of the feature functions f_k in (2.142). For instance, since CRFs do not model the observed input sequence, we can let the feature functions depend on the entire observation sequence without having to alter the dependency structure in the graphical model.

2.6.5 Conrad: CRF-Based Gene Prediction

Generalized (GHMMs), described in Sect. 2.2, have proved very powerful in gene prediction, as they are flexible, easy to train, and easily interpreted probabilistically. The disadvantages include the difficulties to include external information, such as various homology sources, long-ranging sequence features, and unknown dependencies both within and between the external sources. Since CRFs avoid the problems of modeling the observed input data, they can easily incorporate various sources of information, regardless of unknown dependencies and long-range effects.

Conrad [8] is a gene prediction software based on semi-Markov CRFs (SMCRFs), which has inherited the generalized (semi-Markov) features of GHMMs and combined them with discriminative features from the CRF framework. As before, we have a hidden state sequence \mathbf{H} of labels to be predicted, and an observed sequence \mathbf{O} corresponding to the given DNA sequence to be labeled. Again, a CRF expresses the conditional probability $\mathbf{P}(\mathbf{H}|\mathbf{O})$ as opposed to GHMMs which model the joint probability $\mathbf{P}(\mathbf{H}, \mathbf{O})$ of the hidden and the observed data. The conditional probability is as before expressed in log-linear form

$$\mathbf{P}(\mathbf{H}|\mathbf{O}) = \frac{1}{Z_{\lambda}(\mathbf{O})} \exp \left(\sum_j \lambda_j F_j(\mathbf{H}, \mathbf{O}) \right) \quad (2.144)$$

where λ_j is the feature weight, F_j a feature function, which in itself is a sum of features (see below), and $Z_{\lambda}(\mathbf{O})$ the normalizing factor.

The hidden sequence is assumed to be a linearly structured vector of labels such as “exons”, “introns”, and “intergenes”, with one label per nucleotide in the observed sequence. Or conversely, the observed sequence can be segmented into p intervals $\{I_i\}_{i=1}^p = \{(t_i, u_i, v_i)\}_{i=1}^p$ of equally labeled segments (e.g., corresponding to an entire exon), with start at nucleotide t_i , end at u_i , and the same label v_i all through the segment. The segmentation p naturally varies and is determined as part of the prediction. As in GHMMs, Conrad assumes that each interval (t_i, u_i, v_i) only depends on the adjacent neighboring intervals I_{i-1} and I_{i+1} . The feature function F_j is therefore written as a sum of *localized* feature functions

$$F_j(\mathbf{H}, \mathbf{O}) = \sum_{i=1}^p f_j(t_i, u_i, v_i, v_{i-1}, \mathbf{O}). \quad (2.145)$$

The partitioning of the observed sequence is similar to the generalized (semi-Markov) feature of GHMMs and is what makes the CRF semi-Markov. The prediction of hidden labels produced by the SMCRF for a given observed sequence is the segmentation \mathbf{H} that maximizes then the conditional probability $\mathbf{P}(\mathbf{H}|\mathbf{O})$.

Feature Selection

The major issues when applying SMCRFs to gene prediction is the construction of suitable feature functions f_j , and the training of their corresponding weights λ_j . The advantage over GHMMs, as mentioned earlier, is that these features are not required to be independent or to have a probabilistic interpretation. Conrad is constructed to use both *generative features*, inherited from GHMMs, and *discriminative features*, with the result that Conrad can behave either as a pure GHMM or as a SMCRF or anywhere in between. The generative features in Conrad are:

- *Reference features*: modeling the internal sequence composition of the different model states, using a third-order Markov model. These features do not include the segmentation boundaries such as start and stop codons, or splice sites.
- *Length features*: modeling the state length distributions of exons, introns, and intergenic regions. The intergene lengths are modeled using an exponential distribution (the continuous counterpart of the geometric distribution), and exon and intron lengths are modeled by a mixture of two gamma distributions.
- *Transition feature*: modeling the transition probabilities between states.
- *Boundary features*: modeling state boundary signals such as start and stop codons and splice sites.
- *Phylogenetic features*: modeling species homology through state-specific multiple alignments.

By using only reference, length, transition, and boundary features with all weights set to $\lambda_j = 1$, Conrad is equivalent to the conditional probability computed by the corresponding GHMM by taking

$$\mathbf{P}_{GHMM}(\mathbf{H}|\mathbf{O}) = \frac{\mathbf{P}_{GHMM}(\mathbf{H}, \mathbf{O})}{\mathbf{P}_{GHMM}(\mathbf{O})}. \quad (2.146)$$

In the GHMM, we let a_{ij} denote the transition probability between states $i, j \in S$, π_i the initial probability of $i \in S$, and $q_j(O_{t_i}, O_{u_i})$ the emission probability for the segment O_{t_i}, \dots, O_{u_i} , now including the duration probability as well (emission and duration were separated in Sect. 2.2). The joint probability then takes the form

$$\mathbf{P}_{GHMM}(\mathbf{H}|\mathbf{O}) = \pi_{v_1} \prod_{i=2}^p a_{v_{i-1}, v_i} q_{v_i}(O_{t_i}, O_{u_i}) \quad (2.147)$$

and the features in Conrad translates to

$$f_{GHMM}(v_{i-1}, v_i, t_i, u_i, \mathbf{O}) = \begin{cases} \log(q_{v_i}(O_{t_i}, O_{u_i})) + \log(\pi_{v_i}) & \text{if } t_i = 1 \\ \log(q_{v_i}(O_{t_i}, O_{u_i})) + \log(a_{v_{i-1}, v_i}) & \text{if } t_i > 1. \end{cases} \quad (2.148)$$

This version of Conrad (called ConradG-1) is similar to Genscan [7] described in Sect. 2.2.4. ConradG-2, which includes phylogenetic features for two-species comparisons is similar to Twinscan [17] described in Sect. 4.1.2.

Discriminative features are features lacking a probabilistic interpretation. The use of discriminative features enables the ability to incorporate long-range effects and unknown dependencies, or any other type of information that may be difficult to model probabilistically. Conrad incorporates a few discriminative features that represent information commonly used when annotations are curated manually, but that is difficult to include in a probabilistic setting. The discriminative features are:

- *Gap features*: modeling gaps in the multiple alignments that are not captured by the phylogenetic features.
- *Footprint features*: modeling the positions at which the different species in the multiple alignment are aligned.
- *EST features*: modeling the connection between the EST alignments and the state fragmentation of the hidden label sequence.

For instance, the gap feature for a specific exon E takes the form

$$f_{GAP,E}(v_{i-1}, v_i, t_i, u_i, \mathbf{O}) = \sum_{k=t_i}^{u_i} \begin{cases} 1 & \text{if } v_i = E \text{ and gap of length 1 or 2 (mod 3) at } k \\ 0 & \text{otherwise,} \end{cases} \quad (2.149)$$

thus counting the number of gaps in the alignment that would cause a frameshift in the coding sequence. The features are similar for introns and intergenes. Also, the footprint and EST features work the same way, by summing similar indicator functions while scanning through the state segment.

Parameter Training

The feature weights λ_j are trained from labeled example sequences. The common approach to train the weights in CRFs is to use *conditional maximum likelihood* (CML) described in Sect. 6.8. That is, for a single pair of training sequences $(\mathbf{H}^0, \mathbf{O}^0)$, the CML estimator is given by

$$\hat{\lambda}_{CML} = \underset{\lambda}{\operatorname{argmax}} (\log \mathbf{P}(\mathbf{H}^0 | \mathbf{O}^0)). \quad (2.150)$$

The maximum is typically found using a gradient-based technique (see Sect. 6.6), where the specific choice of algorithm depends on the formulation of the CRF. For SMCRFs the common approach is to use dynamic programming algorithms similar to the forward and the backward algorithms in HMMs.

Another approach, introduced by the Conrad group, is to use something called *maximum expected accuracy* (MEA). CML optimizes the accuracy of the prediction indirectly by maximizing the likelihood of the hidden sequences given in the training set. Instead, one would like to optimize the accuracy directly, but this becomes intractable since changing the weights causes changes in the segmentation, which in turn changes the accuracy in a discontinuous way. Instead the objective function is defined as the expected accuracy over the entire distribution of segmentations defined by the SMCRF. However, in order to compute this, we first need to need a similarity metric. We define a similarity function S between the training set $(\mathbf{H}^0, \mathbf{O}^0)$ and a certain label sequence \mathbf{H} as

$$S(\mathbf{H}, \mathbf{H}^0, \mathbf{O}^0) = \sum_{t=1}^T s(H_{t-1}, H_t, H_{t-1}^0, H_t^0, \mathbf{O}^0, t) \quad (2.151)$$

where s are some kind of similarity functions over dinucleotides that can be set as suited. For gene prediction, the function S is divided into two parts, corresponding to splice sites and internal nucleotides. The nucleotide similarity score is simply counting the number of correctly labeled nucleotides in each state, while the splice site similarity scores consider both the labeling and the placement of the splice boundary.

The objective function used to optimize the weights is then defined as the expectation of the similarity function

$$A_{MEA}(\lambda) = E_{\lambda}[S(\mathbf{H}, \mathbf{H}^0, \mathbf{O}^0)] = \sum_{\mathbf{y}} \mathbf{P}_{\lambda}(\mathbf{H}|\mathbf{O}^0) S(\mathbf{H}, \mathbf{H}^0, \mathbf{O}^0) \quad (2.152)$$

and MEA estimator is given by

$$\hat{\lambda}_{MEA} = \underset{\lambda}{\operatorname{argmax}} A_{MEA}(\lambda). \quad (2.153)$$

This maximum is again achieved by using gradient-based methods. However, since the objective function is not concave in λ , there is no guarantee that the global maximum is reached. To achieve the best results, the initial weights are set by using the CML estimates.

References

1. Baldi, P., Brunak, S.: Bioinformatics: The Machine Learning Approach. MIT Press, Cambridge (2001)
2. Begleiter, R., El-Yaniv, R., Yona, G.: On prediction using variable order Markov models. *J. Artif. Intell.* **22**, 385–421 (2004)
3. Bishop, C.M., Lasserre, J.: Generative or discriminative? Getting the best of both worlds. *Bayesian Stat.* **8**, 3–24 (2007)

4. Blattner, F.R., Plunkett, G., Bloch, C.A., Perna, N.T., Burland, V., Riley, M., Collado-vides, J., Glasner, J.D., Rode, C.K., Mayhew, G.F., Gregor, J., Davis, N.W., Kirkpatrick, H.A., Goeden, M.A., Rose, D.J., Mau, B., Shao, Y.: The complete genome sequence of *Escherichia coli* K-12. *Science* **277**, 1453–1469 (1997)
5. Breiman, L.: Some properties of splitting criteria. *Mach. Learn.* **24**, 41–47 (1996)
6. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: *Classification and Regression Trees*. Chapman & Hall, New York (1984)
7. Burge, C., Karlin, S.: Prediction of complete gene structures in human genomic DNA. *J. Mol. Biol.* **268**, 78–94 (1997)
8. DeCaprio, D., Vinson, J.P., Pearson, M.D., Montgomery, P., Doherty, M., Galagan, J.E.: Conrad: gene prediction using conditional random fields. *Genome Res.* **17**, 1389–1398 (2007)
9. Delcher, A.L., Harmon, D., Kasif, S., White, O., Salzberg, S.L.: Improved microbial gene identification with GLIMMER. *Nucleic Acids Res.* **27**, 4636–4641 (1999)
10. Delcher, A.L., Bratke, K.A., Powers, E.C., Salzberg, S.L.: Identifying bacterial genes and endosymbiont DNA with Glimmer. *Bioinformatics* **23**, 673–679 (2007)
11. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: *Biological sequence analysis. Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge (1998)
12. Fickett, J.W., Tung, C.-S.: Assessment of protein coding measures. *Nucleic Acids Res.* **20**, 6441–6450 (1992)
13. Gusfield, D.: *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge (1997)
14. Hammersley, J., Clifford, P.: Markov fields on finite graphs and lattices. <http://www.statslab.cam.ac.uk/~grg/books/hammfest/hamm-cliff.pdf>
15. Jukes, T.H., Osawa, S.: The genetic code in mitochondria and chloroplasts. *Experientia* **46**, 1117–1126 (1990)
16. Karlin, S., Taylor, H.M.: *A First Course in Stochastic Processes*, 2nd edn. Academic Press, New York (1975)
17. Korf, I., Flicek, P., Duan, D., Brent, M.R.: Integrating genomic homology into gene structure prediction. *Bioinformatics* **17**, S140–S148 (2001)
18. Koski, T.: *Hidden Markov Models for Bioinformatics*. Springer, Berlin (2001)
19. Lafferty, J., McCallum, A., Pereira, F.: Conditional random fields: probabilistic models for segmenting and labeling sequence data. In: *Proceedings of International Conference Machine Learning*, pp. 282–289 (2001)
20. Larsen, T., Krogh, A.: EasyGene—a prokaryotic gene finder that ranks ORFs by statistical significance. *BMC Bioinform.* **4**, 21–35 (2003)
21. Ng, A.Y., Jordan, M.I.: On discriminative versus generative classifiers: a comparison of logistic regression and naive Bayes. In: *NIPS* (2001)
22. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biol.* **52**, 99–115 (1943)
23. Murphy, K.P.: *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge (2012)
24. Murthy, S.K., Kasif, S., Salzberg, S.L.: A system for induction of oblique decision trees. *J. Artif. Intell. Res.* **2**, 1–32 (1994)
25. Ohler, U., Harbeck, S., Niemann, H., Nöth, E., Reese, M.G.: Interpolated Markov chains for eukaryotic promoter recognition. *Bioinformatics* **15**, 362–369 (1999)
26. Perna, N.T., Plunkett, G., Burland, V., Mau, B., Glasner, J.D., Rose, D.J., Mayhew, G.F., Evans, P.S., Gregor, J., Kirkpatrick, H.A., Pósfai, G., Hackett, J., Klink, S., Boutin, A., Shao, Y., Miller, L., Grotbeck, E.J., Davis, N.W., Lim, A., Dimalanta, E.T., Potamouisis, K.D., Apodaca, J., Anantharaman, T.S., Lin, J., Yen, G., Schwartz, D.C., Welch, R.A., Blattner, F.R.: Genome sequence of enterohaemorrhagic *Escherichia coli* O157:H7. *Nature* **409**, 529–533 (2001)
27. Perte, M., Lin, X., Salzberg, S.L.: GeneSplicer: a new computational method for splice site prediction. *Nucleic Acids Res.* **29**, 1185–1190 (2001)
28. Quinlan, J.R.: Induction of decision trees. *Mach. Learn.* **1**, 81–106 (1986)
29. Quinlan, J.R.: *C4.5: Programs for machine learning*. Morgan Kaufmann Publishers, San Mateo (1993)

30. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. *Proc. IEEE* **77**, 257–286 (1989)
31. Rissanen, J.: A universal data compression system. *IEEE Trans. Inf. Theory* **29**, 656–664 (1983)
32. Rivas, E., Eddy, S.R.: Noncoding RNA gene detection using comparative sequence analysis. *BMC Bioinform.* **2**, 8 (2001)
33. Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. *Psychol. Rev.* **65**, 386–408 (1958)
34. Salzberg, S.L., Delcher, A.L., Fasman, K.H., Henderson, J.: A decision tree system for finding genes in DNA. *J. Comput. Biol.* **5**, 667–680 (1998)
35. Salzberg, S.L., Delcher, A.L., Kasif, S., White, O.: Microbial gene identification using interpolated Markov models. *Nucleic Acids Res.* **26**, 544–548 (1998)
36. Schukat-Talamazzini, E.G., Gallwitz, F., Harbeck, S., Warnke, V.: Rational interpolation of maximum likelihood predictors in stochastic language modeling. In: *Proceedings of Eurospeech'97*, pp. 2731–2734. Rhodes, Greece (1997)
37. Sharp, P.M., Cowe, E.: Synonymous codon usage in *Sacharomyces cerevisiae*. *Yeast* **7**, 657–678 (1991)
38. Shmatkov, A.M., Melikyan, A.A., Chernousko, F.L., Borodovsky, M.: Finding prokaryotic genes by the 'frame-by-frame' algorithm: targeting gene starts and overlapping genes. *Bioinformatics* **15**, 874–886 (1999)
39. Shmilovici, A., Ben-Gal, I.: Using a VOM model for reconstructing potential coding regions in EST sequences. *Comput. Stat.* **22**, 49–69 (2007)
40. Skovgaard, M., Jensen, L.J., Brunak, S., Ussery, D., Krogh, A.: On the total number of genes and their length distribution in complete microbial genomes. *Trends Genet.* **17**, 425–428 (2001)
41. Snyder, E.E., Stormo, G.D.: Identification of protein coding regions in genomic DNA. *J. Mol. Biol.* **248**, 1–18 (1995)
42. Sutton, C., McCallum, A.: An introduction to conditional random fields. *Found. Trends Mach. Learn.* **4**, 267–373 (2011)
43. Xu, Y., Mural, R.J., Einstein, J.R., Shah, M.B., Uberbacher, E.C.: GRAIL: a multi-agent neural network system for gene identification. *Proc. IEEE* **84**, 1544–1552 (1996)
44. Xu, Y., Uberbacher, E.C.: Computational gene prediction using neural networks and similarity search. In: Salzberg, S.L., Searls, D.B., Kasif, S. (eds.) *Computational Methods in Molecular Biology*, pp. 109–128. Elsevier Science B.V., Amsterdam (1998)
45. <http://www.cbcb.umd.edu/glimmer/>

Comparative Gene Finding
Models, Algorithms and Implementation
Axelson-Fisk, M.
2015, XX, 382 p. 81 illus., Hardcover
ISBN: 978-1-4471-6692-4