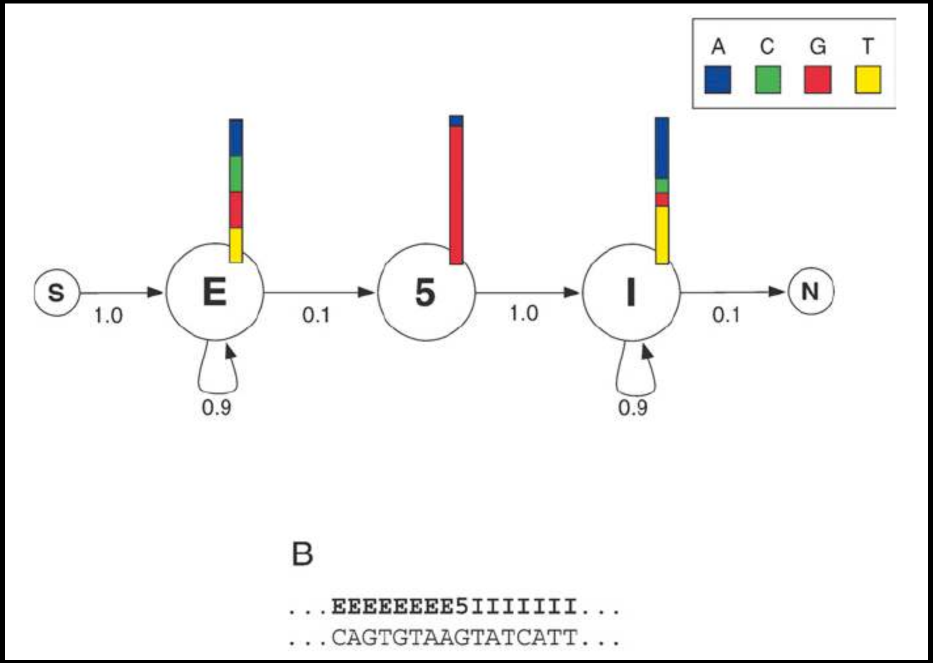
Markov Chains and …Hidden Markov Models



Source: Benjamin Schuster-Bockler & Alex Bateman (2007)

George Vernikos – gvernikos@gmail.com

Hidden Markov Models
…a little bit of history

Hidden Markov Models: a very general form of probabilistic model for sequences of symbols. Types of question we can answer with HMMs, include: “Does this sequence belong to a particular family?” “Assuming the sequence does come from some family what can we say about its internal structure? (e.g. identify an alpha helix in a protein sequence)”.

Hidden Markov Models
…a little bit of history

Hidden Markov Models: a very general form of probabilistic model for sequences of symbols. Types of question we can answer with HMMs, include: “Does this sequence belong to a particular family?” “Assuming the sequence does come from some family what can we say about its internal structure? (e.g. identify an alpha helix in a protein sequence)”.

The overwhelming majority of literature on HMMs sits on speech recognition, where HMMs were first applied in the 1970s (Rabiner 1989). After recording, a speech signal is divided into pieces, called frames, of 10-20 milliseconds. After some preprocessing each frame is assigned to one out of a large number (typically 256) of predefined categories.

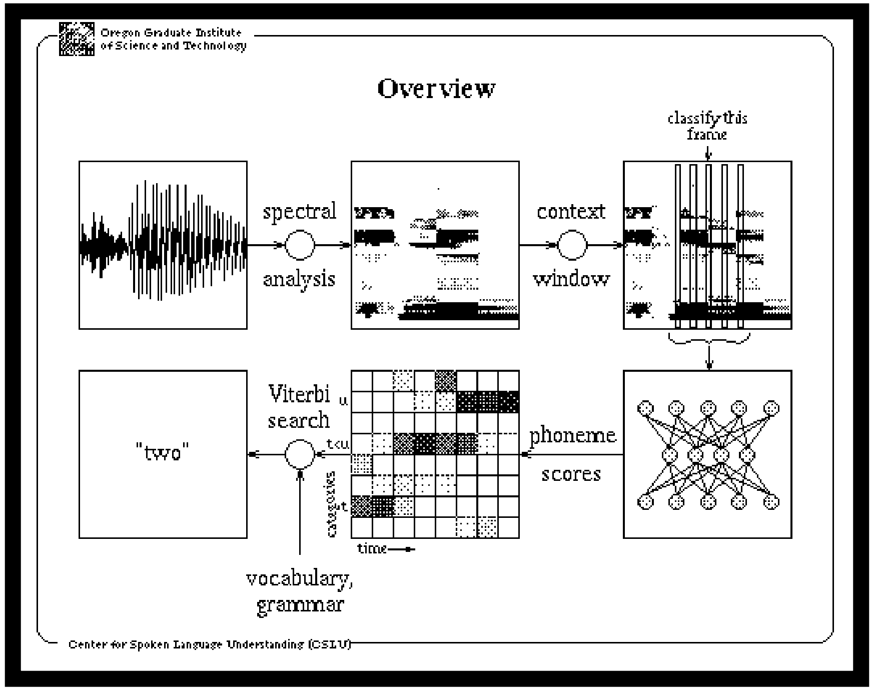
Hidden Markov Models
…a little bit of history

Hidden Markov Models: a very general form of probabilistic model for sequences of symbols. Types of question we can answer with HMMs, include: "Does this sequence belong to a particular family?" "Assuming the sequence does come from some family what can we say about its internal structure? (e.g. identify an alpha helix in a protein sequence)".

The overwhelming majority of literature on HMMs sits on speech recognition, where HMMs were first applied in the 1970s (Rabiner 1989). After recording, a speech signal is divided into pieces, called frames, of 10-20 milliseconds. After some preprocessing each frame is assigned to one out of a large number (typically 256) of predefined categories.

The speech signal is then represented as a long sequence of category labels and from that the speech recognizer has to find out what sequence of phonemes (or words) was spoken. The problems are that there are variations in the actual sound uttered, and there are also variations in the time taken to say various parts of the word.

Hidden Markov Models
…in speech recognition



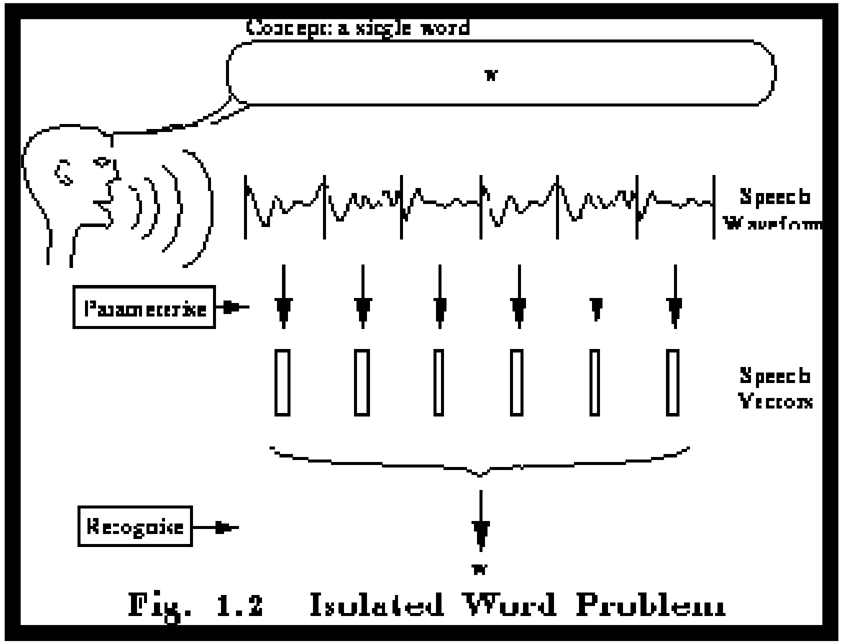


Fig. 1.2 Isolated Word Problem

Hidden Markov Models
…a little bit of history

In biology we have similar problems to deal with, e.g. we typically want to know what protein family a given sequence belongs to. Here the primary sequence of amino acids is analogous to the speech signal and the protein family to the spoken word it represents. The time-variation of the speech signal corresponds to having insertions and deletions in the protein sequences.

Hidden Markov Models
…a little bit of history

In biology we have similar problems to deal with, e.g. we typically want to know what protein family a given sequence belongs to. Here the primary sequence of amino acids is analogous to the speech signal and the protein family to the spoken word it represents. The time-variation of the speech signal corresponds to having insertions and deletions in the protein sequences.

Example: CpG islands

In the human genome wherever the dinucleotide CG occurs, the C nucleotide is typically chemically modified by methylation. There is a relative high chance of this methyl-C mutating into a T, with the consequence that in general CpG dinucleotides are rarer in the genome that would be expected from the independent probabilities of C and G.

Hidden Markov Models
…a little bit of history

In biology we have similar problems to deal with, e.g. we typically want to know what protein family a given sequence belongs to. Here the primary sequence of amino acids is analogous to the speech signal and the protein family to the spoken word it represents. The time-variation of the speech signal corresponds to having insertions and deletions in the protein sequences.

Example: CpG islands

In the human genome wherever the dinucleotide CG occurs, the C nucleotide is typically chemically modified by methylation. There is a relative high chance of this methyl-C mutating into a T, with the consequence that in general CpG dinucleotides are rarer in the genome that would be expected from the independent probabilities of C and G.

For biological important reasons the methylation process is suppressed in short stretches of the genome, such as around promoters or start regions of many genes. In these regions we see many more CpG dinucleotides than elsewhere, and in fact more C and G in general. Such regions are called CpG islands (Bird 1987). They are typically a few hundred to a few thousand bases long.

CpG islands & Markov chains

A. Given a short stretch of genomic sequence, how would we decide if it comes from a CpG island or not?

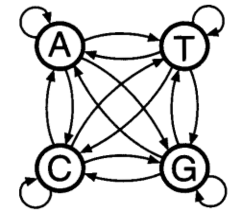
B. Given a long piece of sequence how would we find the CpG island in it, if there are any?

CpG islands & Markov chains

A. Given a short stretch of genomic sequence, how would we decide if it comes from a CpG island or not?

B. Given a long piece of sequence how would we find the CpG island in it, if there are any?

What short of probabilistic model should we use for CpG islands? We know that dinucleotides are important. We therefore want a model that generates sequences in which the probability of a symbol depends on the previous symbol. The simplest such model is a classical Markov chain:

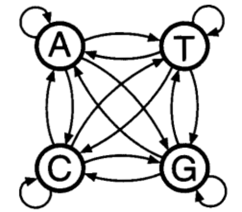


CpG islands & Markov chains

A. Given a short stretch of genomic sequence, how would we decide if it comes from a CpG island or not?

B. Given a long piece of sequence how would we find the CpG island in it, if there are any?

What short of probabilistic model should we use for CpG islands? We know that dinucleotides are important. We therefore want a model that generates sequences in which the probability of a symbol depends on the previous symbol. The simplest such model is a classical Markov chain:



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

$$a_{st} = P(x_i = t \mid x_{i-1} = s)$$

Markov chains

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

$$P(x) = P(x_L, x_{L-1}, \ldots, x_1)$$
$$= P(x_L \mid x_{L-1}, \ldots, x_1) P(x_{L-1} \mid x_{L-2}, \ldots, x_1) \ldots P(x_1)$$

by applying *(PX,Y)* = *P(Y)P(X|Y)* many times.

Markov chains

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

$$P(x) = P(x_L, x_{L-1}, \ldots, x_1)$$
$$= P(x_L \mid x_{L-1}, \ldots, x_1) P(x_{L-1} \mid x_{L-2}, \ldots, x_1) \ldots P(x_1)$$

by applying *(PX,Y)* = *P(Y)P(X|Y)* many times.

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

The previous equation therefore becomes:

$$P(x) = P(x_L \mid x_{L-1}) P(x_{L-1} \mid x_{L-2}) P(x_2 \mid x_1) \ldots P(x_1)$$
$$= P(x_1) \prod_{i=2}^{L} a_{xi-1xi} \qquad (3.2)$$

Markov chains
… modelling the beginning and end of sequences

Notice that as well as specifying the transition probabilities we must also give the probability $P(x_1)$ of starting in a particular state. It is possible to introduce an extra *begin* (*B*) and *end* (*E*) state to the model:

$$P(x_1 = s) = a_{Bs}$$

$$P(E \mid x_L = t) = a_{tE}$$

which is the probability of ending with residue *t*. We can treat those two new states are "*silent*" states.

Using Markov chains for discrimination

We can use equation 3.2 to calculate the values for a likelihood ratio test. From a set of human DNA sequences we extract a total of 48 putative CpG islands and derive two Markov chain models, one for the regions labeled as CpG islands (+ model) and the other from the remainder of the sequence (- model). The transition probabilities for each model are set using the equation

$$a_{st}^{+} = \frac{c_{st}^{+}}{\sum_{t'} c_{st'}^{+}} \qquad (3.3)$$

and its analogue for the - model, where C_{st}^{+} is the number of times letter *t* is followed by letter *s* in the labeled regions (i.e. the ML estimators for the transition probabilities):

+	A	C	G	T	-	A	C	G	T
A	0.180	0.274	0.426	0.120	**A**	0.300	0.205	0.285	0.210
C	0.171	0.368	0.274	0.188	**C**	0.322	0.298	0.078	0.302
G	0.161	0.339	0.375	0.125	**G**	0.248	0.246	0.298	0.208
T	0.079	0.355	0.384	0.182	**T**	0.177	0.239	0.292	0.292

Using Markov chains for discrimination

We can use equation 3.2 to calculate the values for a likelihood ratio test. From a set of human DNA sequences we extract a total of 48 putative CpG islands and derive two Markov chain models, one for the regions labeled as CpG islands (+ model) and the other from the remainder of the sequence (- model). The transition probabilities for each model are set using the equation

$$a^{+}_{st} = \frac{c^{+}_{st}}{\sum_{t'} c^{+}_{st'}} \qquad (3.3)$$

and its analogue for the - model, where C_{st}^{+} is the number of times letter *t* is followed by letter *s* in the labeled regions (i.e. the ML estimators for the transition probabilities):

+	A	C	G	T	-	A	C	G	T
A	0.180	0.274	0.426	0.120	**A**	0.300	0.205	0.285	0.210
C	0.171	0.368	0.274	0.188	**C**	0.322	0.298	0.078	0.302
G	0.161	0.339	0.375	0.125	**G**	0.248	0.246	0.298	0.208
T	0.079	0.355	0.384	0.182	**T**	0.177	0.239	0.292	0.292

Using Markov chains for discrimination

where the first row in these case contains the frequencies with which an *A* is followed by each of the four bases, and so on for the other rows, so each row sums to one. These numbers are not the same; *G* following *A* is much more common than *T* following *A*.

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

$$S(x) = \log \frac{P(x \mid \text{model}^{+})}{P(x \mid \text{model}^{-})} = \sum_{i=1}^{L} \log \frac{a^{+}_{xi-1xi}}{a^{-}_{xi-1xi}} = \sum_{i=1}^{L} \beta_{xi-1xi}$$

where *x* is the sequence and $\beta_{xi\text{-}1xi}$ are the log likelihood ratios of the corresponding transition probabilities. A table *β* is given below in bits:

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

Using Markov chains for discrimination

where the first row in these case contains the frequencies with which an *A* is followed by each of the four bases, and so on for the other rows, so each row sums to one. These numbers are not the same; *G* following *A* is much more common than *T* following *A*.

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

$$S(x) = \log \frac{P(x \mid \text{model}^{+})}{P(x \mid \text{model}^{-})} = \sum_{i=1}^{L} \log \frac{a^{+}_{xi-1xi}}{a^{-}_{xi-1xi}} = \sum_{i=1}^{L} \beta_{xi-1xi}$$

where *x* is the sequence and β_{xi-1xi} are the log likelihood ratios of the corresponding transition probabilities. A table *β* is given below in bits:

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

Using Markov chains for discrimination … CpG islands

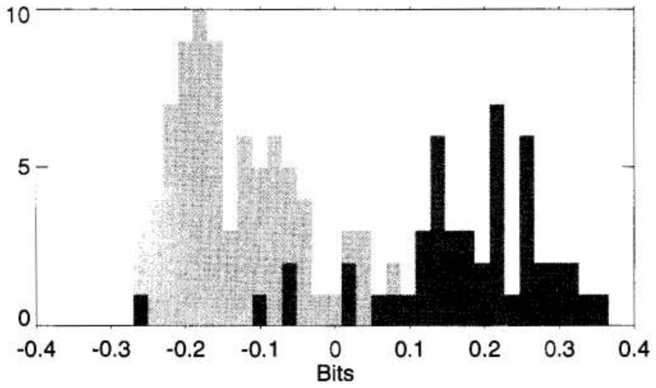


Figure 3.2 *The histogram of the length-normalised scores for all the sequences. CpG islands are shown with dark grey and non-CpG with light grey.*

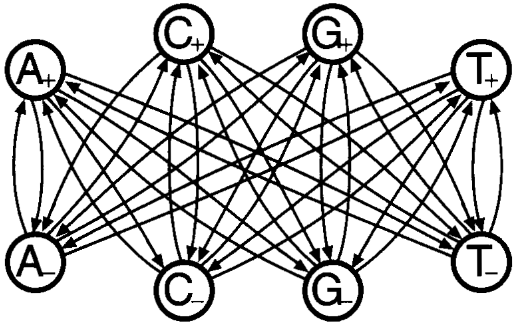


Figure 3.3 *An HMM for CpG islands. In addition to the transitions shown, there is also a complete set of transitions within each set, as in the earlier simple Markov chains.*

Hidden Markov Models

"How do we find CpG islands in a long unannotated sequence?"

A simple approach would be to use the Markov chain models that we built earlier by calculating the log-odds score for a window size, say, 100 nucleotides around every nucleotide in the sequence and plotting it. We would expect CpG islands to stand out with positive values.

Hidden Markov Models

“How do we find CpG islands in a long unannotated sequence?”

A simple approach would be to use the Markov chain models that we built earlier by calculating the log-odds score for a window size, say, 100 nucleotides around every nucleotide in the sequence and plotting it. We would expect CpG islands to stand out with positive values.

However this is somehow unsatisfactory if we believe that CpG islands have sharp boundaries and are of variable lengths. Why use a window size of 100? A more satisfactory approach is to build a single model for the entire sequence and incorporate both Markov chains.

Hidden Markov Models

To simulate in one model the “islands” in a “sea” of non-island genomic sequence, we want to have both the Markov chains in the same model, with a small probability of switching from one chain to the other at each transition point.

Hidden Markov Models

To simulate in one model the “islands” in a “sea” of non-island genomic sequence, we want to have both the Markov chains in the same model, with a small probability of switching from one chain to the other at each transition point.

We re-label the states as follows: A_+, C_+, G_+, T_+ which emit A, C, G, T in CpG island regions and A_-, C_-, G_-, T_- which emit A, C, G, T in non-island regions. The transition probabilities in this model are set so that within each group they are close to the transition probabilities of the original component model, but with a small chance of switching into the other component. Overall there is more chance of switching from + to – than vice versa, so if left to run free, the model will spend more of its time in the – non-island states than in the island states.

Hidden Markov Models

To simulate in one model the “islands” in a “sea” of non-island genomic sequence, we want to have both the Markov chains in the same model, with a small probability of switching from one chain to the other at each transition point.

We re-label the states as follows: A_+, C_+, G_+, T_+ which emit A, C, G, T in CpG island regions and A_-, C_-, G_-, T_- which emit A, C, G, T in non-island regions. The transition probabilities in this model are set so that within each group they are close to the transition probabilities of the original component model, but with a small chance of switching into the other component. Overall there is more chance of switching from + to – than vice versa, so if left to run free, the model will spend more of its time in the – non-island states than in the island states.

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

Hidden Markov Models

We need to distinguish the *sequence of states* from the *sequence of symbols*. Let us call the *state* sequence the *path*, *π*. The *i*th state in the path is called π_i:

$$a_{kl} = P(\pi_i = l \mid \pi_{i-1} = k) \qquad (3.4)$$

Hidden Markov Models

We need to distinguish the *sequence of states* from the *sequence of symbols*. Let us call the *state* sequence the *path*, *π*. The *i*th state in the path is called π_i:

$$a_{kl} = P(\pi_i = l \mid \pi_{i-1} = k) \qquad (3.4)$$

Because we have decoupled the symbol *b* from the states *k*, we must introduce a new set of parameters for the model, $e_k(b)$:

$$e_k(b) = P(x_i = b \mid \pi_i = k) \qquad (3.5)$$

the probability that symbol *b* is seen in state *k* (i.e. the *emission* probabilities).

Hidden Markov Models

We need to distinguish the *sequence of states* from the *sequence of symbols*. Let us call the *state* sequence the *path*, π. The ith state in the path is called π_i:

$$a_{kl} = P(\pi_i = l \mid \pi_{i-1} = k) \qquad (3.4)$$

Because we have decoupled the symbol b from the states k, we must introduce a new set of parameters for the model, $e_k(b)$:

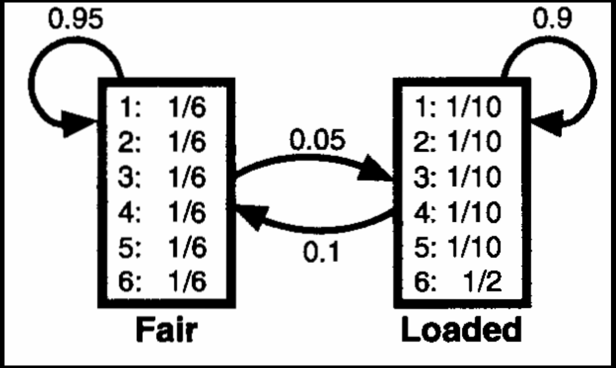
$$e_k(b) = P(x_i = b \mid \pi_i = k) \qquad (3.5)$$

the probability that symbol b is seen in state k (i.e. the *emission* probabilities).

To illustrate emission probabilities we switch back to the casino example. In a casino they use a fair die most of the time, but occasionally they switch to a loaded die. The loaded die has probability 0.5 of a six and probability 0.1 for the numbers one to five. Assume that the casino switches from a fair to a loaded die with probability 0.05 and 0.1 for switching back.

Hidden Markov Models

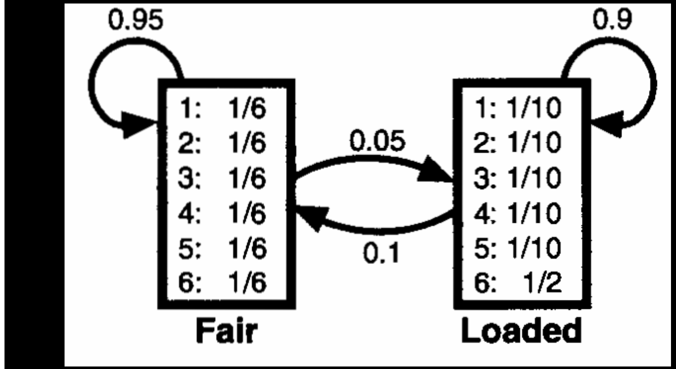
Then the switch between dice is a Markov process:



What is hidden in the above model?

Hidden Markov Models

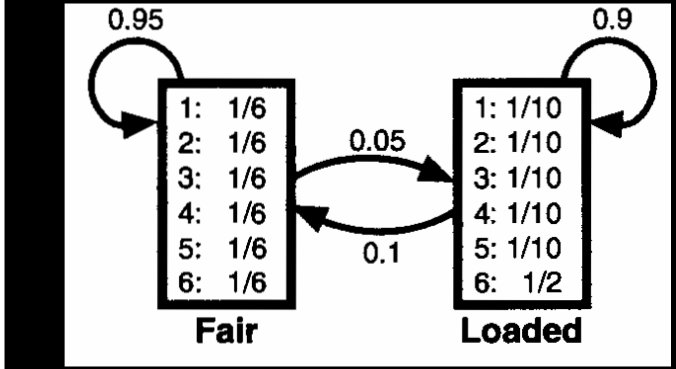
Then the switch between dice is a Markov process:



What is hidden in the above model? If you can just see a sequence of rolls you do not know which rolls used a loaded die and which used a fair one, because that is kept secret by the casino; that is the *state sequence is hidden*.

Hidden Markov Models

Then the switch between dice is a Markov process:



What is hidden in the above model? If you can just see a sequence of rolls you do not know which rolls used a loaded die and which used a fair one, because that is kept secret by the casino; that is the *state sequence is hidden*.

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

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

Most probable state path
… the Viterbi algorithm

Although it is no longer possible to tell what state the system is in by looking at the corresponding symbol, it is often the sequence of underlying states that we are interested in.

Most probable state path
… the Viterbi algorithm

Although it is no longer possible to tell what state the system is in by looking at the corresponding symbol, it is often the sequence of underlying states that we are interested in.

To find out what the observation sequence “means” by considering the underlying states is called *decoding* in the jargon of speech recognition. There are several approaches to decoding; here we will discuss the most common one, called the Viterbi algorithm.

Most probable state path
… the Viterbi algorithm

Although it is no longer possible to tell what state the system is in by looking at the corresponding symbol, it is often the sequence of underlying states that we are interested in.

To find out what the observation sequence “means” by considering the underlying states is called *decoding* in the jargon of speech recognition. There are several approaches to decoding; here we will discuss the most common one, called the Viterbi algorithm.

In general there may be many state sequences that could give rise to any particular sequence of symbols, for example:

$[\ C_+\ G_+\ C_+\ G_+\ ]$

$[\ C_-\ G_-\ C_-\ G_-\ ]$

$[\ C_+\ G_-\ C_+\ G_-\ ]$

would all generate the symbol sequence CGCG. However they do so with very different probabilities.

Most probable state path
… the Viterbi algorithm

Although it is no longer possible to tell what state the system is in by looking at the corresponding symbol, it is often the sequence of underlying states that we are interested in.

To find out what the observation sequence “means” by considering the underlying states is called *decoding* in the jargon of speech recognition. There are several approaches to decoding; here we will discuss the most common one, called the Viterbi algorithm.

In general there may be many state sequences that could give rise to any particular sequence of symbols, for example:

$[\ C_+\ G_+\ C_+\ G_+\ ]$

$[\ C_-\ G_-\ C_-\ G_-\ ]$

$[\ C_+\ G_-\ C_+\ G_-\ ]$

would all generate the symbol sequence CGCG. However they do so with very different probabilities.

Find the most probable *path* π

Most probable state path
… the Viterbi algorithm

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

$$\pi^* = \arg\max_{\pi} \ P(x,\pi) \qquad (3.7)$$

Most probable state path
… the Viterbi algorithm

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

$$\pi^* = \arg\max_{\pi} \; P(x, \pi) \qquad (3.7)$$

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

$$u_l(i+1) = e_l(x_{i+1}) \max_k (u_k(i) a_{kl}) \qquad (3.8)$$

Most probable state path … the Viterbi algorithm

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

$$\pi^* = \underset{\pi}{\arg\max}\ P(x,\pi) \qquad (3.7)$$

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

$$u_l(i+1) = e_l(x_{i+1})\max_k(u_k(i)a_{kl}) \qquad (3.8)$$

All sequences have to start in state 0 (the begin state), so the initial condition is that $u_0(0) = 1$. By keeping pointers backwards, the actual state sequence can be found by backtracing. The full algorithm is:

Most probable state path
… the Viterbi algorithm

Initialisation (*i* = 0):

$$u_0(0) \;=\; 1,\; u_k(0) \;=\; 0 \; for\; k \;> 0$$

Recursions *(i* = 1 … *L):*

$$u_l(i) \;=\; e_l(xi)\max{}_k(u_k(i-1)a_{kl})$$

$$ptr_i(l) = \arg\max{}_k(u_k(i-1)a_{kl})$$

Termination:

$$P(x,\pi^*) = \max{}_k(u_k(L)a_{k0})$$

$$\pi^*_L = \arg\max{}_k(u_k(L)a_{k0})$$

Tracecback *(i* = *L … 1)*:

$$\pi^*_{i-1} = ptr_i(\pi^*_i)$$

Most probable state path … the Viterbi algorithm

Initialisation (*i* = 0):

$$u_0(0) \;=\; 1,\; u_k(0) \;=\; 0 \; for \; k \; > 0$$

Recursions *(i = 1 … L):*

$$u_l(i) \;=\; e_l(xi)\max_k(u_k(i-1)a_{kl})$$

$$ptr_i(l) = \arg\max_k(u_k(i-1)a_{kl})$$

Termination:

$$P(x,\pi^*) = \max_k(u_k(L)a_{k0})$$

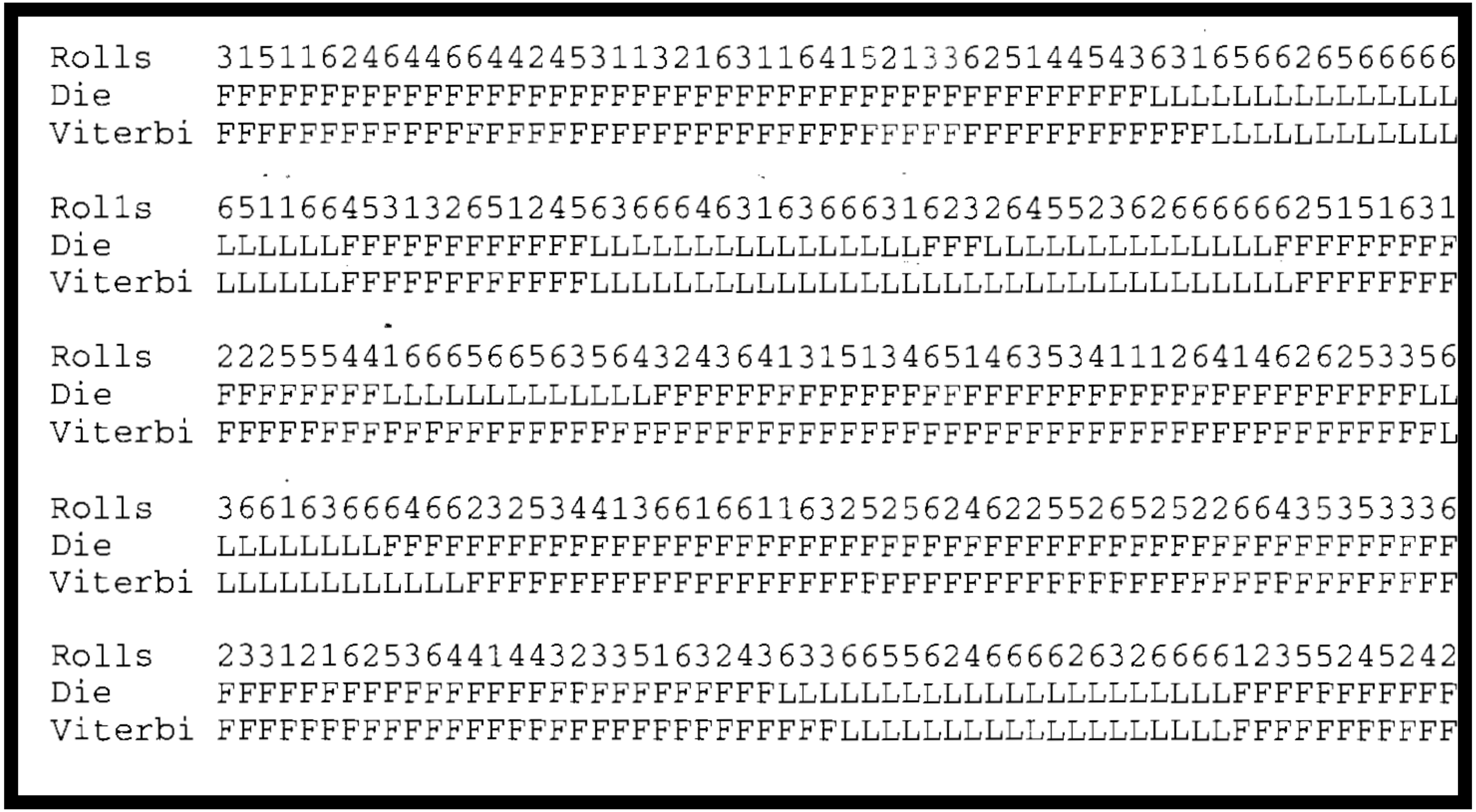
$$\pi_L^* = \arg\max_k(u_k(L)a_{k0})$$

Tracecback *(i = L … 1)*:

$$\pi_{i-1}^* = ptr_i(\pi_i^*)$$

Multiplying many probabilities always yields very small numbers that will give overflow errors on any computer. For this reason the *Viterbi* algorithm should always be done in log space.

The occasionally dishonest casino



```
Rolls   315116246446644245311321631164152133625144543631656626566666
Die     FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLL
Viterbi FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLL

Rolls   651166453132651245636664631636663162326455236266666625151631
Die     LLLLLLFFFFFFFFFFFLLLLLLLLLLLLLLLLLFFFLLLLLLLLLLLLLLFFFFFFFFF
Viterbi LLLLLLFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLFFFFFFFF

Rolls   222555441666566563564324364131513465146353411126414626253356
Die     FFFFFFFFLLLLLLLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLL
Viterbi FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFL

Rolls   366163666466232534413661661163252562462255265252266435353336
Die     LLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Viterbi LLLLLLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Rolls   233121625364414432335163243633665562466662632666612355245242
Die     FFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLLLLFFFFFFFFFFF
Viterbi FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLFFFFFFFFFFF
```

The forward algorithm

To calculate the probability of an entire sequence *P(x)* for an HMM we must add the probabilities for all possible paths to obtain the full probability of *x*, because many different state paths can give rise to the same sequence *x*:

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

The forward algorithm

To calculate the probability of an entire sequence *P(x)* for an HMM we must add the probabilities for all possible paths to obtain the full probability of *x*, because many different state paths can give rise to the same sequence *x*:

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

This probability can be calculated by a similar dynamic programming procedure to the *Viterbi* algorithm, replacing maximization steps with sums. This is called the *forward* algorithm.

The forward algorithm

To calculate the probability of an entire sequence *P(x)* for an HMM we must add the probabilities for all possible paths to obtain the full probability of *x*, because many different state paths can give rise to the same sequence *x*:

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

This probability can be calculated by a similar dynamic programming procedure to the *Viterbi* algorithm, replacing maximization steps with sums. This is called the *forward* algorithm.

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

$$f_k(i) = P(x_1 \dots x_i, \pi_i = k) \quad (3.10)$$

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

The forward algorithm

Initialisation (*i* = 0):

$$f_0(0) \ = \ 1, f_k(0) \ = \ 0 \, for \ k \ > 0$$

Recursions *(i = 1 … L):*

$$f_l(i) \ = \ e_l(xi) \sum_k f_k(i-1) a_{kl}$$

Termination:

$$P(x) = \sum_k f_k(L) a_{k0}$$

The backward algorithm

The *Viterbi* algorithm finds the most probable path through the model. But what if we want to know what the most probable state is for an observation x_i?

The backward algorithm

The *Viterbi* algorithm finds the most probable path through the model. But what if we want to know what the most probable state is for an observation x_i?

More generally we may want the probability that observation x_i came from state k given the observed sequence, i.e. $P(\pi_i = k|x)$. This is the posterior probability of state k at time i when the emitted sequence is known.

The backward algorithm

The *Viterbi* algorithm finds the most probable path through the model. But what if we want to know what the most probable state is for an observation x_i?

More generally we may want the probability that observation x_i came from state k given the observed sequence, i.e. $P(\pi_i = k|x)$. This is the posterior probability of state k at time i when the emitted sequence is known.

We first calculate the probability of producing the entire observed sequence with the ith symbol being produced by state k:

$$P(x,\pi_i = k) = \underline{P(x_1 \ldots x_i, \pi_i = k)}\,\underline{P(x_{i+1} \ldots x_L \mid \pi_i = k)} \qquad (3.12)$$

The backward algorithm

The *Viterbi* algorithm finds the most probable path through the model. But what if we want to know what the most probable state is for an observation x_i?

More generally we may want the probability that observation x_i came from state k given the observed sequence, i.e. $P(\pi_i = k|x)$. This is the posterior probability of state k at time i when the emitted sequence is known.

We first calculate the probability of producing the entire observed sequence with the ith symbol being produced by state k:

$$P(x,\pi_i = k) = \underline{P(x_1 \ldots x_i, \pi_i = k)}\underline{P(x_{i+1} \ldots x_L \mid \pi_i = k)} \quad (3.12)$$

The first term is recognized as $f_k(i)$ that was calculated by the forward algorithm. The second term is called $b_k(i)$:

$$b_k(i) = P(x_{i+1} \ldots x_L \mid \pi_i = k) \quad (3.13)$$

The backward algorithm

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

Initialisation ($i = L$):

$$b_k(L) = a_{k0} \text{ for all } k$$

Recursions ($i = L$-1, …, 1):

$$b_k(i) = \sum_l a_{kl} e_l(x_{i+1}) b_l(i+1)$$

Termination:

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

The backward algorithm

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

$$P(\pi_i = k \mid x) = \frac{f_k(i)b_k(i)}{P(x)} \qquad (3.14)$$

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

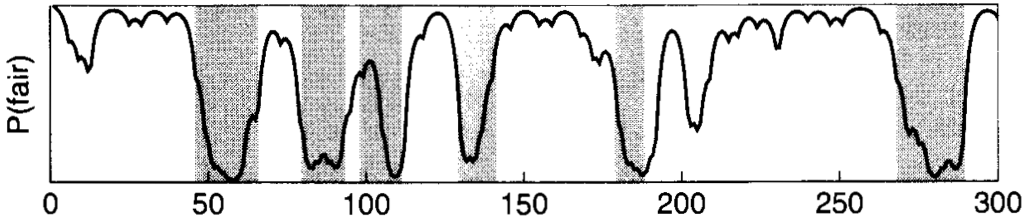


Figure 3.6 *The posterior probability of being in the state corresponding to the fair die in the casino example. The x axis shows the number of the roll. The shaded areas show when the roll was generated by the loaded die.*

[F] [B] [V] algorithms

✓What is the probability of observing *X*?

Forward algorithm

✓What is the probability that the internal state at time *i* was a specific state *k*?

Backward algorithm

✓What is the most probable path of hidden states?

Viterbi algorithm

HMMs … parameter estimation

We assume that we have a set of example sequences (*training sequences*) of the type that we want to model. Let these be $x^1, \ldots, x^n$. Working in log space the log probability of the sequences is:

$$l(x^1,\ldots,x^n \mid \theta) = \log P(x^1,\ldots,x^n \mid \theta) = \sum_{j=1}^{n} \log P(x^j \mid \theta) \quad (3.17)$$

where θ represents the entire current set of values of the parameters in the model (all the *as* and *es*).

HMMs … parameter estimation when the state sequence is known

When the paths are known the estimation of the probability parameters is easy. We can simply count the number of times each particular *transition* or *emission* is used in the training dataset. Let these be A_{kl} and $E_k(b)$. Then the ML estimators for be a_{kl} and $e_k(b)$ are:

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

HMMs … parameter estimation when the state sequence is known

When the paths are known the estimation of the probability parameters is easy. We can simply count the number of times each particular *transition* or *emission* is used in the training dataset. Let these be A_{kl} and $E_k(b)$. Then the ML estimators for be a_{kl} and $e_k(b)$ are:

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

To avoid overfitting if there are insufficient data, we should add pseudocounts to the A_{kl} and $E_k(b)$ before suing 3.18.

A_{kl} = number of transitions k to l in the training data + r_{kl}.

$E_k(b)$ = number of emissions of b from k in the training data + $r_k(b)$.

HMMs … parameter estimation when the paths are unknown

When the paths are unknown for the training sequences, there is no longer a direct closed form equation for the estimated parameter values, and some form of iterative procedure must be used.

HMMs … parameter estimation when the paths are unknown

When the paths are unknown for the training sequences, there is no longer a direct closed form equation for the estimated parameter values, and some form of iterative procedure must be used.

A standardly used algorithm is the *Baum-Welch* algorithm (Baum 1972). It first estimates the A_{kl} and $E_k(b)$ by considering probable paths for the training sequences using the current values of a_{kl} and $e_k(b)$. The BW algorithm is a special case of a very powerful general approach to probabilistic parameter estimation called EM algorithm.

HMMs … parameter estimation when the paths are unknown

When the paths are unknown for the training sequences, there is no longer a direct closed form equation for the estimated parameter values, and some form of iterative procedure must be used.

A standardly used algorithm is the *Baum-Welch* algorithm (Baum 1972). It first estimates the A_{kl} and $E_k(b)$ by considering probable paths for the training sequences using the current values of a_{kl} and $e_k(b)$. The BW algorithm is a special case of a very powerful general approach to probabilistic parameter estimation called EM algorithm.

Then 3.18 is used to derive new values of the *as* and *es*. This process is iterated until some stopping criterion is reached. The overall log likelihood of the model is increased by the iteration, and hence the process will converge to a local maximum.

HMMs … parameter estimation when the paths are unknown

When the paths are unknown for the training sequences, there is no longer a direct closed form equation for the estimated parameter values, and some form of iterative procedure must be used.

A standardly used algorithm is the *Baum-Welch* algorithm (Baum 1972). It first estimates the A_{kl} and $E_k(b)$ by considering probable paths for the training sequences using the current values of a_{kl} and $e_k(b)$. The BW algorithm is a special case of a very powerful general approach to probabilistic parameter estimation called EM algorithm.

Then 3.18 is used to derive new values of the *as* and *es*. This process is iterated until some stopping criterion is reached. The overall log likelihood of the model is increased by the iteration, and hence the process will converge to a local maximum.

Unfortunately there are usually many local maxima, and which one you end up with depends strongly on the starting values of the parameters.

HMMs … parameter estimation when the paths are unknown

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

HMMs … parameter estimation when the paths are unknown

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

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

HMMs … parameter estimation when the paths are unknown

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

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

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

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

where $f_k^j(i)$ is the forward variable calculated for sequence j and $b_l^j(i)$ is the corresponding backward variable.

HMMs … parameter estimation when the paths are unknown

Similarly we can find the expected number of times that letter *b* appears in state *k*:

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

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

HMMs … parameter estimation when the paths are unknown

Similarly we can find the expected number of times that letter *b* appears in state *k*:

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

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

Having calculated these expectations the new model parameters are calculated again via 3.18. We can iterate using the new values of the parameters to obtain new values of the *A*s and *E*s but since we are converging in a continuous-values space we will never in fact reach the maximum, so we need to set a convergence criterion.

HMMs … parameter estimation when the paths are unknown

Summary of Baum-welch:

Initialisation: Pick arbitrary model parameters.

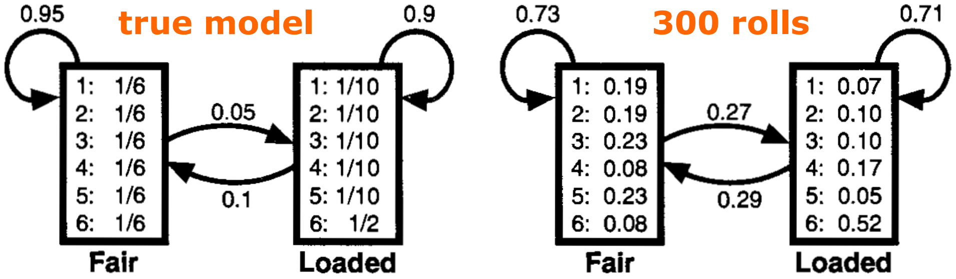
Recurrence:

```
        Set all the A and E variables to their pseudocount values r or to zero.
        For each sequence j = 1…n:
                Calculate $f_k(i)$ for sequence j using the forward algorithm.
                Calculate $b_k(i)$ for sequence j using the backward algorithm.
                Add the contribution of sequence j to A (3.20) and E (3.21).
        Calculate the new model parameters using 3.18
        Calculate the new log likelihood of the model.
```

Termination: Stop if the change in log likelihood is less than some predefined threshold.

The occasional dishonest casino

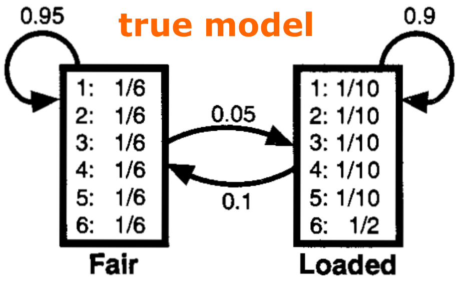
We are suspicious that a casino is using a loaded die, but we do not know for certain. Night after night we collect data observing rolls. When we have enough we want to estimate the model. From this sequence of observations a model was estimated using BW. Initially all the probabilities were set to random numbers.

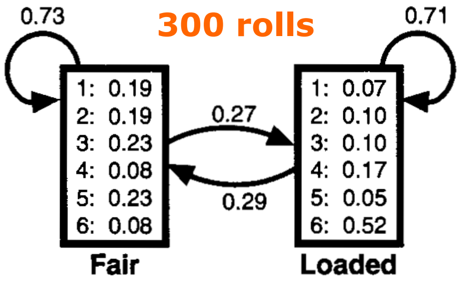


You can see they are fairly similar although the estimated transition probabilities are quite different. This is problem of local maxim due to low number of observations.

The occasional dishonest casino

We are suspicious that a casino is using a loaded die, but we do not know for certain. Night after night we collect data observing rolls. When we have enough we want to estimate the model. From this sequence of observations a model was estimated using BW. Initially all the probabilities were set to random numbers.



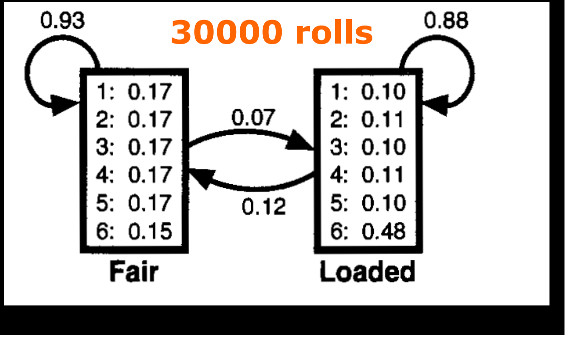


You can see they are fairly similar although the estimated transition probabilities are quite different. This is problem of local maxim due to low number of observations.

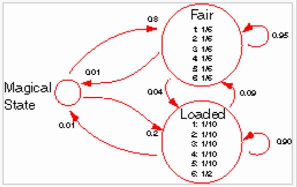
We repeat with 30000 random rolls:

And this time we came closer to the true model:

The correct model	0.101 bits
Model estimated from 300 rolls	0.097 bits
Model estimated from 30000 rolls	0.100 bits



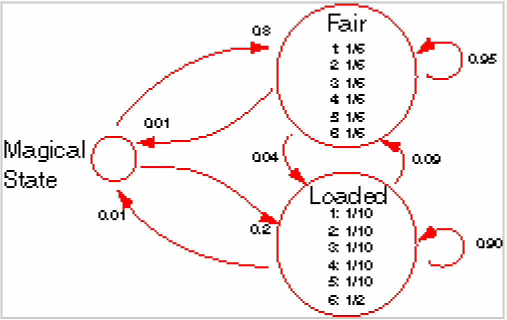
```
//The core of the program is the createCasino() method.
//This creates an instance of the MarkovModel class that implements the model.
public static MarkovModel createCasino() {
    Symbol[] rolls=new Symbol[6];

    //set up the dice alphabet
    SimpleAlphabet diceAlphabet=new SimpleAlphabet();
    diceAlphabet.setName("DiceAlphabet");

    for(int i=1;i<7;i++) {
      try {
        rolls[i-1]= AlphabetManager.createSymbol((char)('0'+i),""+i,Annotation.EMPTY_ANNOTATION);
        diceAlphabet.addSymbol(rolls[i-1]);
      } catch (Exception e) {
        throw new NestedError(
          e, "Can't create symbols to represent dice rolls"
        );
      }
    }

//Next, distributions representing the emission probabilities of the fair die and
//loaded die states are created (named fairD and loadedD respectively)
int [] advance = { 1 };
Distribution fairD;
Distribution loadedD;
try {
  fairD =DistributionFactory.DEFAULT.createDistribution(diceAlphabet);
  loadedD =DistributionFactory.DEFAULT.createDistribution(diceAlphabet);
} catch (Exception e) {
  throw new NestedError(e, "Can't create distributions");
}
EmissionState fairS = new SimpleEmissionState("fair",Annotation.EMPTY_ANNOTATION, advance, fairD);
EmissionState loadedS = newSimpleEmissionState("loaded",Annotation.EMPTY_ANNOTATION, advance, loadedD);
```



The resultant HMM looks like this.

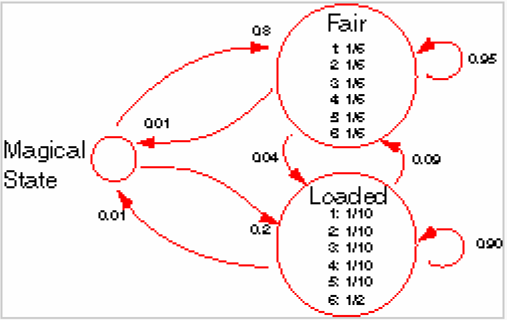
Source: http://www.biojava.org/wiki/BioJava:Tutorial:Simple_HMMs_with_BioJava

```
//The HMM is then created with these states
SimpleMarkovModel casino = new SimpleMarkovModel(1,diceAlphabet, "Casino");
try {
  casino.addState(fairS);
  casino.addState(loadedS);
  }
catch (Exception e) {
   throw new NestedError(e, "Can't add states to model");
  }

//Next, we need to model the transitions between the states.
 try {
      casino.createTransition(casino.magicalState(),fairS);
      casino.createTransition(casino.magicalState(),loadedS);
      casino.createTransition(fairS,casino.magicalState());
      casino.createTransition(loadedS,casino.magicalState());
      casino.createTransition(fairS,loadedS);
      casino.createTransition(loadedS,fairS);
      casino.createTransition(fairS,fairS);
      casino.createTransition(loadedS,loadedS);
    } catch (Exception e) {
      throw new NestedError(e, "Can't create transitions");
    }

//The emission distributions fairD and loadedD we set up earlier need to be initialised
try {
      for(int i=0;i<rolls.length;i++)   {
        fairD.setWeight(rolls[i],1.0/6.0);
        loadedD.setWeight(rolls[i], 0.1);
      }
      loadedD.setWeight(rolls[5],0.5);
    } catch (Exception e) {
      throw new NestedError(e, "Can't set emission probabilities");
    }
```

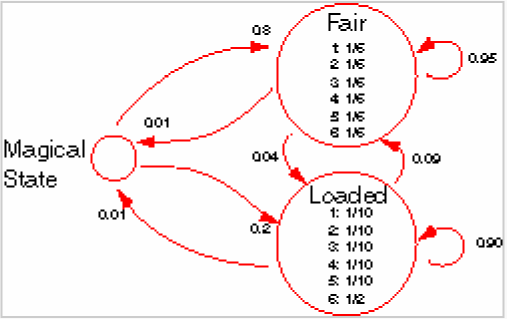


The resultant HMM looks like this.

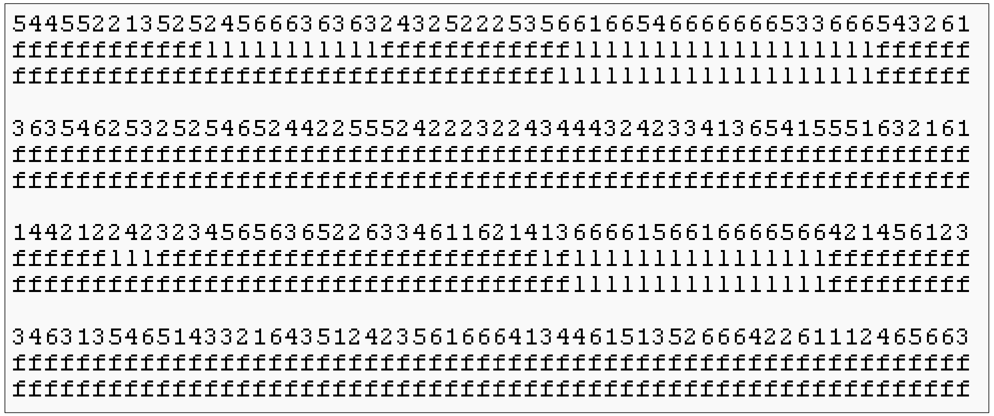
```
//set up transition scores.
    try {
      Distribution dist;

      dist = casino.getWeights(casino.magicalState());
      dist.setWeight(fairS, 0.8);
      dist.setWeight(loadedS, 0.2);

      dist = casino.getWeights(fairS);
      dist.setWeight(loadedS,              0.04);
      dist.setWeight(fairS,               0.95);
      dist.setWeight(casino.magicalState(), 0.01);

      dist = casino.getWeights(loadedS);
      dist.setWeight(fairS,               0.09);
      dist.setWeight(loadedS,              0.90);
      dist.setWeight(casino.magicalState(), 0.01);
    } catch (Exception e) {
      throw new NestedError(e, "Can't set transition probabilities");
    }

//Having completed constructing the MarkovModel, all that remains is to return it to the caller.
return casino;

//Having created the MarkovModel, we create the corresponding dynamic programming object
DP dp=DPFactory.DEFAULT.createDP(casino);

//Now, at last, we have something we can use! To generate a sequence of dice throws with this model, we do
StatePath obs_rolls = dp.generate(300);

//Next, we want to test one of the DP algorithms in the DP object
SymbolList roll_sequence =obs_rolls.symbolListForLabel(StatePath.SEQUENCE);
SymbolList[] res_array = {roll_sequence};
StatePath v = dp.viterbi(res_array,ScoreType.PROBABILITY);

//print out obs_sequence, output, state symbols.
for(int i = 1; i <= obs_rolls.length()/60; i++) {
        for(int j=i*60; j
```



The resultant HMM looks like this.

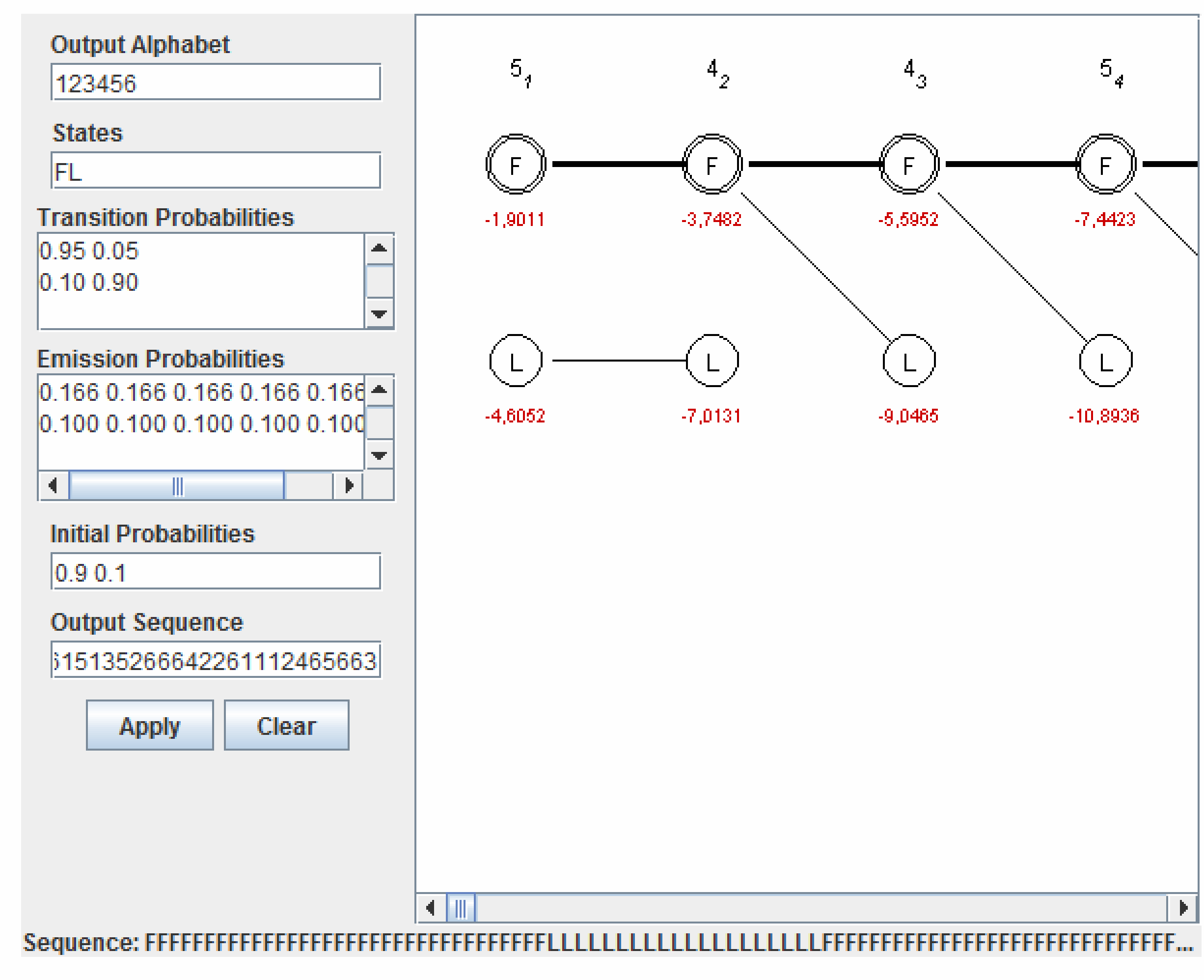


The resultant HMM looks like this.



```
544552213525245666363632432522253566166546666666653366654326 1
ffffffffffffllllllllllllffffffffffffllllllllllllllllllllffffff
ffffffffffffffffffffffffffffffffffllllllllllllllllllllffffff

363546253252546524422555242223224344432423341365415551632161
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

144212242323456563652263346116214136666156616666566421456123
fffffflllfffffffffffffffffffffffflflllllllllllllllllfffffffff
fffffffffffffffffffffffffffffffffffllllllllllllllllfffffffff

346313546514332164351242356166641344615135266642261112465663
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

Viterbi Demo



Output Alphabet
123456

States
FL

Transition Probabilities
0.95 0.05
0.10 0.90

Emission Probabilities
0.166 0.166 0.166 0.166 0.166
0.100 0.100 0.100 0.100 0.100

Initial Probabilities
0.9 0.1

Output Sequence
5151352666422611124656663

Sequence: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF...

Source: http://www.cs.umb.edu/~srevilak/viterbi/

HMM … for Genomic Islands

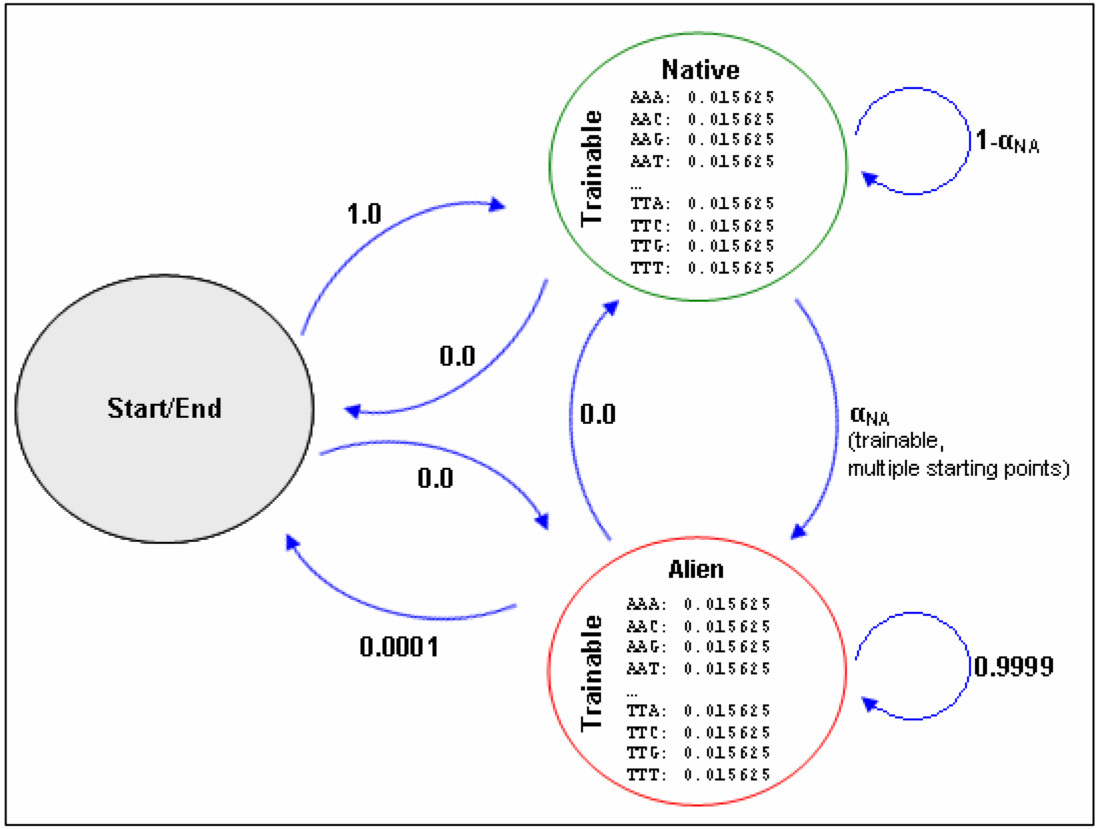


Figure .: The architecture of the two-state (Native, Alien), second order HMM, used in a change-point detection framework.

HMM … for Genomic Islands

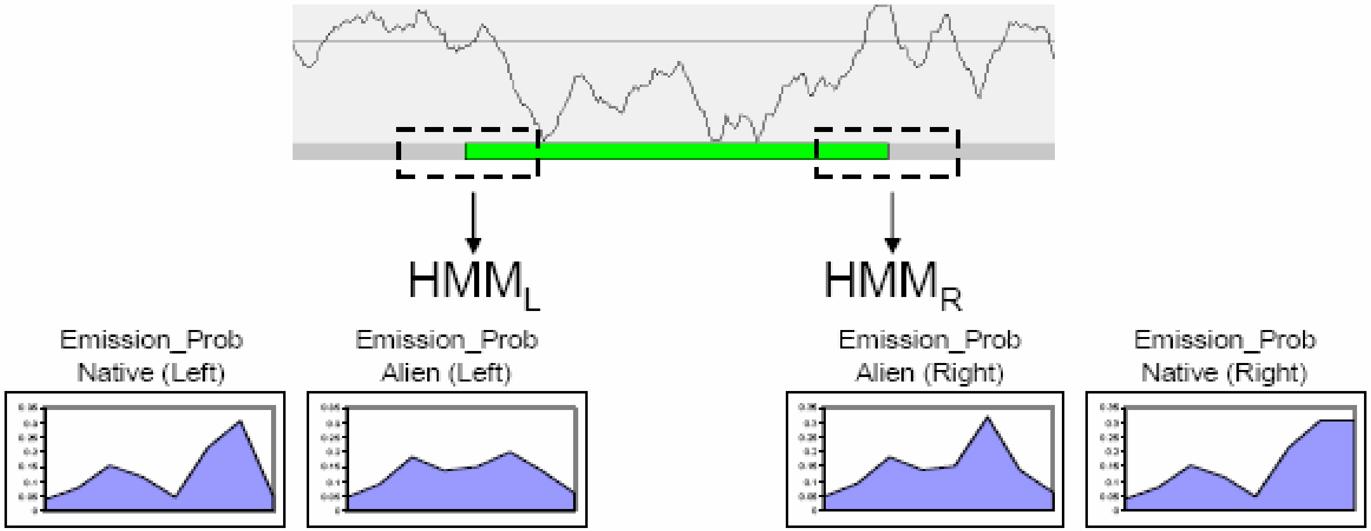
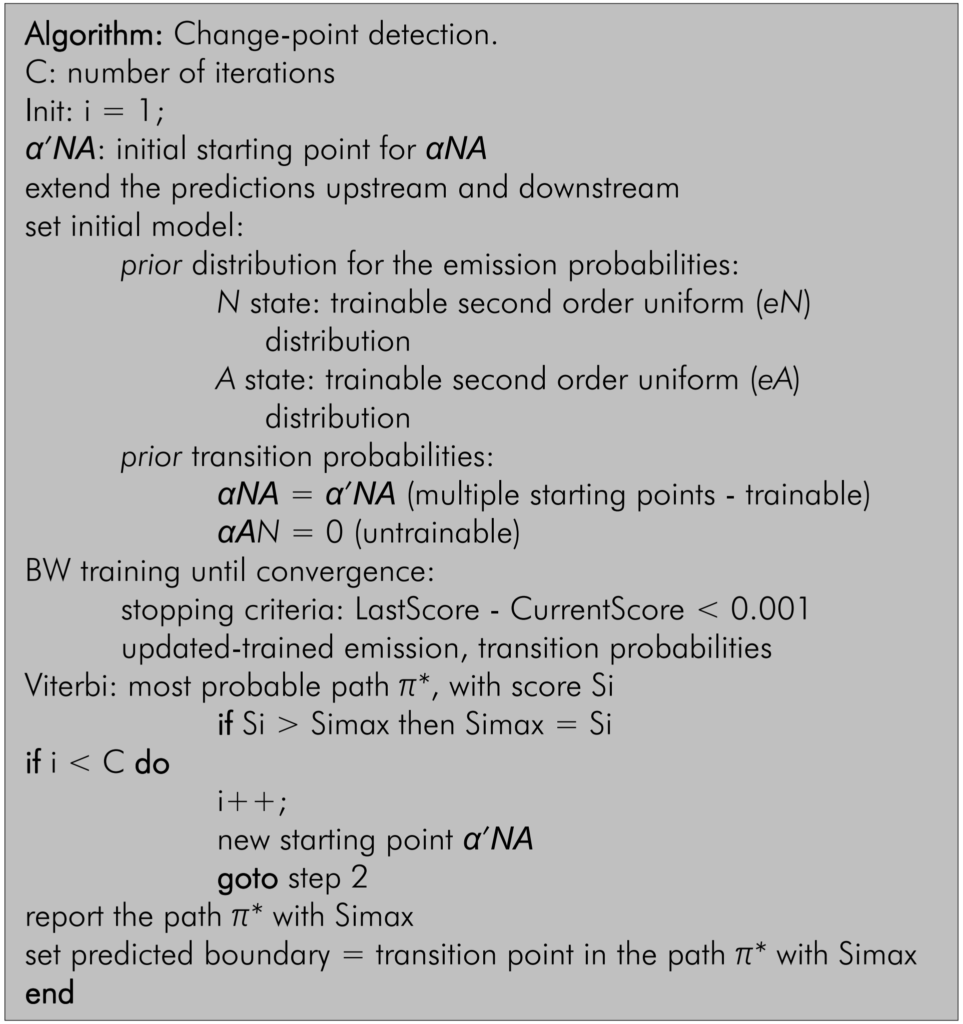
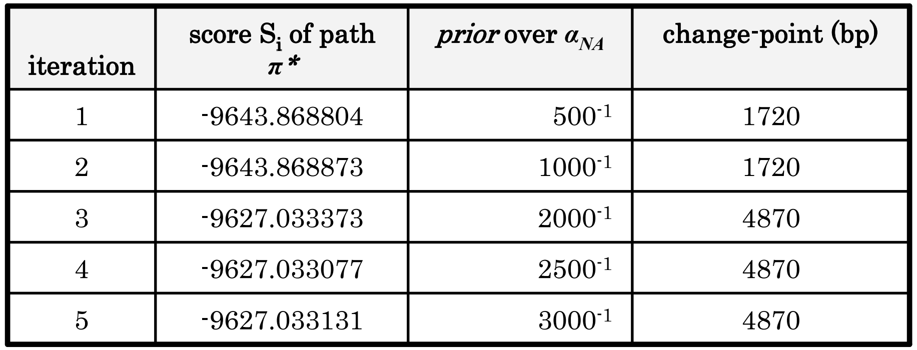


Figure 2.4: Two side-specific HMMs trained for the left (HMM_L) and for the right (HMM_R) boundary of each predicted GI.



```
Algorithm: Change-point detection.
C: number of iterations
Init: i = 1;
α′NA: initial starting point for αNA
extend the predictions upstream and downstream
set initial model:
        prior distribution for the emission probabilities:
                N state: trainable second order uniform (eN)
                    distribution
                A state: trainable second order uniform (eA)
                    distribution
        prior transition probabilities:
                αNA = α′NA (multiple starting points - trainable)
                αAN = 0 (untrainable)
BW training until convergence:
        stopping criteria: LastScore - CurrentScore < 0.001
        updated-trained emission, transition probabilities
Viterbi: most probable path π*, with score Si
                if Si > Simax then Simax = Si
if i < C do
                i++;
                new starting point α′NA
                goto step 2
report the path π* with Simax
set predicted boundary = transition point in the path π* with Simax
end
```



iteration	score S_i of path $\pi *$	*prior* over α_{NA}	change-point (bp)
1	-9643.868804	500^{-1}	1720
2	-9643.868873	1000^{-1}	1720
3	-9627.033373	2000^{-1}	4870
4	-9627.033077	2500^{-1}	4870
5	-9627.033131	3000^{-1}	4870

```
import java.io.*;
import org.biojava.bio.symbol.*;
import org.biojava.bio.seq.*;
import org.biojava.bio.seq.io.*;
import org.biojava.bio.dp.*;
import org.biojava.bio.*;
import org.biojava.bio.seq.db.*;
import org.biojava.bio.seq.impl.*;
import org.biojava.bio.dist.*;
import org.biojava.utils.*;
import java.util.*;

class ChangepointLeft{

public static SymbolList seqL;
public static int order;
public static int flatOrRandom;
public static int trainOrUntrain;
public static Distribution dist;
public static int duration;
public static ModelTrainer mt;
public static int transition_point=0;
public static int count=0;

	//make alphabets
	static FiniteAlphabet DnaAlphabet = DNATools.getDNA();

	public static void main (String args[]) throws Exception{

	if(args.length != 5) {
	throw new Exception("Use: sequence.fa order.int flatD.bin trainableTrans.bin duration.int");
	}

	try{
```

```
File seqFile = new File(args[0]);
               order = Integer.parseInt(args[1]);
               flatOrRandom = Integer.parseInt(args[2]);
               trainOrUntrain = Integer.parseInt(args[3]);
               duration = Integer.parseInt(args[4]);

               if((flatOrRandom != 0) & (flatOrRandom != 1)) {
               throw new Exception("Use flatD.bin: only binary i.e. 0 or 1: . . 1/0 . .");
               }
               if((trainOrUntrain != 0) & (trainOrUntrain != 1)) {
               throw new Exception("Use trainableTrans.bin: only binary i.e. 0 or 1: . . . 1/0 .");
               }

               SymbolTokenization rParser = DnaAlphabet.getTokenization("token");

               SequenceBuilderFactory sbFact = new FastaDescriptionLineParser.Factory(SimpleSequenceBuilder.FACTORY);
               FastaFormat fFormat = new FastaFormat();

               SequenceIterator seqI = new StreamReader(new FileInputStream(seqFile),
                                                fFormat,
                                                rParser,
                                                sbFact);
               seqI.hasNext();

               Sequence seq2 = seqI.nextSequence();
               SequenceDB seqs = new HashSequenceDB();
               seqL = seq2;

               MarkovModel island = createModel();
               DP dp=DPFactory.DEFAULT.createDP(island);

              Sequence seq = new SimpleSequence(
                   SymbolListViews.orderNSymbolList(seq2, order),
                   null,
                   seq2.getName() + "-o" + order,
                   Annotation.EMPTY_ANNOTATION
               );

               seqs.addSequence(seq);
```

```
TrainingAlgorithm ta = new BaumWelchTrainer(dp);

            ta.train(
                  seqs,
                  0.01,
            new StoppingCriteria() {
                  public boolean isTrainingComplete(TrainingAlgorithm ta) {

                  try {
                  // XmlMarkovModel.writeModel(ta.getDP().getModel(), System.out);
                  //out2.write(ta.getCycle() + "\t" + ta.getCurrentScore() + "\n");
                  }catch (Exception ex) {ex.printStackTrace();}
                  //System.out.println(ta.getCycle() + "\t" + ta.getCurrentScore());
                  //return (ta.getCycle() >=2);
                  return Math.abs(ta.getLastScore() - ta.getCurrentScore()) < 0.001;
                  }
            }
            );
```

```
//Viterbi

            SymbolList [] rl = {SymbolListViews.orderNSymbolList(seq2, order)};

            StatePath statePath = dp.viterbi(rl, ScoreType.PROBABILITY);

            for(int i = 0; i <= statePath.length() / 60; i++) {

                 for(int j = i*60; j < Math.min((i+1)*60, statePath.length()); j++) {
                       //System.out.print(statePath.symbolAt(StatePath.STATES, j+1).getName().charAt(0));
                       char state=statePath.symbolAt(StatePath.STATES, j+1).getName().charAt(0);
                      count++;
                       //it prints the states in binary mode for art user_graph
                       if(state == 'a'){
                       //out.write("0 1");
                       }
                       else{
                       transition_point=count;
                       //out.write("1 0");
                       }

                 }

            }

            System.out.print(transition_point + " " + statePath.getScore());

            }catch (Exception e) {
            e.printStackTrace();
      }
}
```

```
//creates the model
   public static MarkovModel createModel() {

      List l = Collections.nCopies(order, DNATools.getDNA());
      Alphabet alpha = AlphabetManager.getCrossProductAlphabet(l);

      int [] advance = { 1 };
      Distribution typicalD;
      Distribution atypicalD;

      try{

            //check if higher order; else normal dist
            if(order >1){
            typicalD = OrderNDistributionFactory.DEFAULT.createDistribution(alpha);
            atypicalD = OrderNDistributionFactory.DEFAULT.createDistribution(alpha);
            }
            else{
            typicalD = DistributionFactory.DEFAULT.createDistribution(alpha);
            atypicalD = DistributionFactory.DEFAULT.createDistribution(alpha);
            }


      }catch (Exception e){
      throw new AssertionFailure("Can't create distributions", e);
      }
```

```
EmissionState typicalS = new SimpleEmissionState("typical", Annotation.EMPTY_ANNOTATION, advance, typicalD);
EmissionState atypicalS = new SimpleEmissionState("atypical", Annotation.EMPTY_ANNOTATION, advance, atypicalD);

    SimpleMarkovModel island = new SimpleMarkovModel(1, alpha, "Island");

    try{
        island.addState(typicalS);
        island.addState(atypicalS);
    }catch (Exception e){
    throw new AssertionFailure("Can't add states to model", e);
    }

    //set up transitions between states
    try {
        island.createTransition(island.magicalState(),typicalS);
        island.createTransition(island.magicalState(),atypicalS);
        island.createTransition(typicalS,island.magicalState());
        island.createTransition(atypicalS,island.magicalState());
        island.createTransition(typicalS,atypicalS);
        island.createTransition(atypicalS,typicalS);
        island.createTransition(typicalS,typicalS);
        island.createTransition(atypicalS,atypicalS);
    }catch (Exception e){
    throw new AssertionFailure("Can't create transitions", e);
    }
```

```java
//set up emission probabilities
      try {
            SymbolList highOrderSeq = SymbolListViews.orderNSymbolList (seqL, order);
            Hashtable symbol= new Hashtable();

           for (Iterator i = highOrderSeq.iterator(); i.hasNext(); ) {
                  Symbol sym = (Symbol) i.next();

                  if(!symbol.containsKey(sym)){
                  //uniform weights for atypical emmision probs
                  atypicalD.setWeight(sym,0.25);
                  typicalD.setWeight(sym, 0.25);
                  symbol.put(sym, new Integer(1));
                  }
            }

            if(flatOrRandom == 0){
            //it randomizes the atypical emission probs
            DistributionTools.randomizeDistribution(atypicalD);
            DistributionTools.randomizeDistribution(typicalD);
            }

      }catch (Exception e) {
      throw new AssertionFailure("Can't set emission probabilities", e);
      }
```

```
//set up transition scores.
        try {
                {
                //if user option =1 then it trains ; if 0 then untrained
                if(trainOrUntrain ==0){
                //it keeps the transition probs untrainable
                dist = new UntrainableDistribution (island.transitionsFrom(island.magicalState()));
                }
                else{
                dist = island.getWeights(island.magicalState());
                }
                dist.setWeight(typicalS,          1.0);
                //since it will always start at start at state typicalS
                dist.setWeight(atypicalS,         0.0);
                island.setWeights(island.magicalState(), dist);
                }

                {
                // always trainable
                dist = island.getWeights(typicalS);
                float T_A = (float)1/duration;
                float T_T = (float)1-T_A;
                //1/region = 1/7500
                dist.setWeight(atypicalS,            T_A);
                //1-1/7500
                dist.setWeight(typicalS,             T_T);
                //zero since it will always end at atypical
                dist.setWeight(island.magicalState(), 0.0);
                island.setWeights(typicalS, dist);
                }
        {
        // always trainable
        dist = island.getWeights(typicalS);
        float T_A = (float)1/duration;
        float T_T = (float)1-T_A;
        //1/region = 1/7500
        dist.setWeight(atypicalS,         T_A);
        //1-1/7500
        dist.setWeight(typicalS,          T_T);
        //zero since it will always end at atypical
        dist.setWeight(island.magicalState(), 0.0);
        island.setWeights(typicalS, dist);
        }

        {
        //always untrainable
        dist = new UntrainableDistribution (island.transitionsFrom(atypicalS));
        //when it changes it persists for ever.
        dist.setWeight(typicalS,          0.0000000000000000000000000000001);
        dist.setWeight(atypicalS,         0.9999);
        //it was 0.0001  but it throwed NaNs
        dist.setWeight(island.magicalState(), 0.0000999999999999999999999999999);
        island.setWeights(atypicalS, dist);
        }
    }catch (Exception e) {
    throw new AssertionFailure("Can't set transition probabilities", e);
    }

  return island;
 }

}
```

Viterbi …
online DEMO (exercise)

Source: http://www.cs.umb.edu/~srevilak/viterbi/

Target sequence: “ATGCATGCATGGGGCC”

Alphabet: [A, T, G ,C]

# of states: 2

Transition: There is 0.2 probability of switching from state1to state2. There is 0.9 probability of switching from state2 to state1.

Emission: In state1 the frequency of observing A, T, G, C is their expected frequencies assuming a zero-th order alphabet. In state2 P_A = P_T =0.1 and P_G = P_C .

Initial probabilities: The probability of the model starting in state1 is 0.6.

Deliverables:

A. Build the model.

B. Run the prediction.

C. Record the most probable state path.

D. Design the HMM architecture.

HMMER

hmmalign - align sequences to a profile HMM

hmmbuild - construct profile HMM(s) from multiple sequence alignment(s)

hmmconvert - convert profile file to a HMMER format

hmmemit - sample sequences from a profile HMM

hmmfetch - retrieve profile HMM(s) from a file

hmmpress - prepare an HMM database for hmmscan

hmmscan - search sequence(s) against a profile database

hmmsearch - search profile(s) against a sequence database

hmmsim - collect score distributions on random sequences

hmmstat - display summary statistics for a profile file

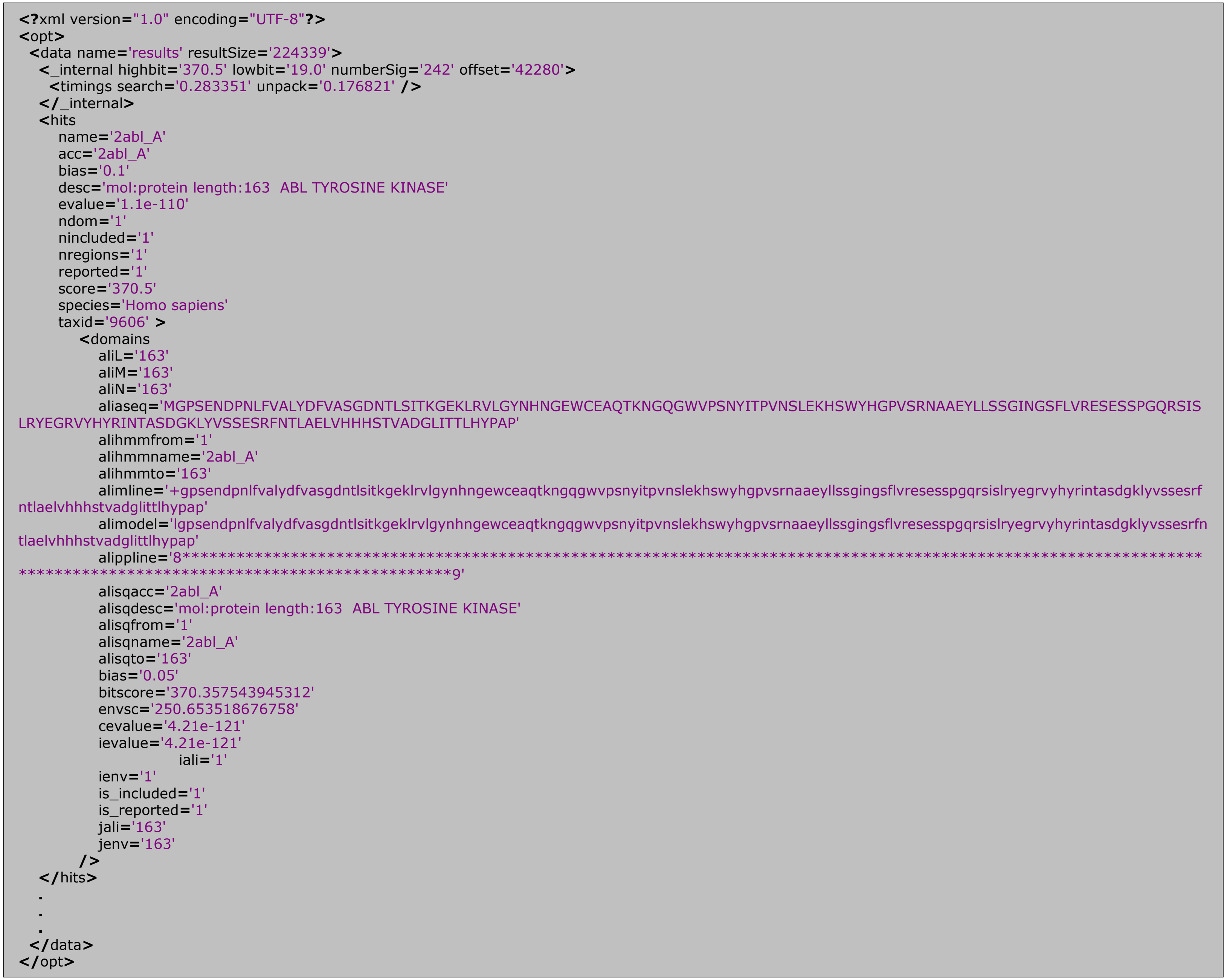
jackhmmer - iteratively search sequence(s) against a protein database

phmmer - search protein sequence(s) against a protein sequence database

SOURCE: http://hmmer.janelia.org/

HMMER (cURL)

```
shell% curl -L -H 'Expect:' -H 'Accept:text/xml' -F seqdb=pdb -F algo=phmmer
-F seq='<test.seq' http://hmmer.janelia.org/search/phmmer
```



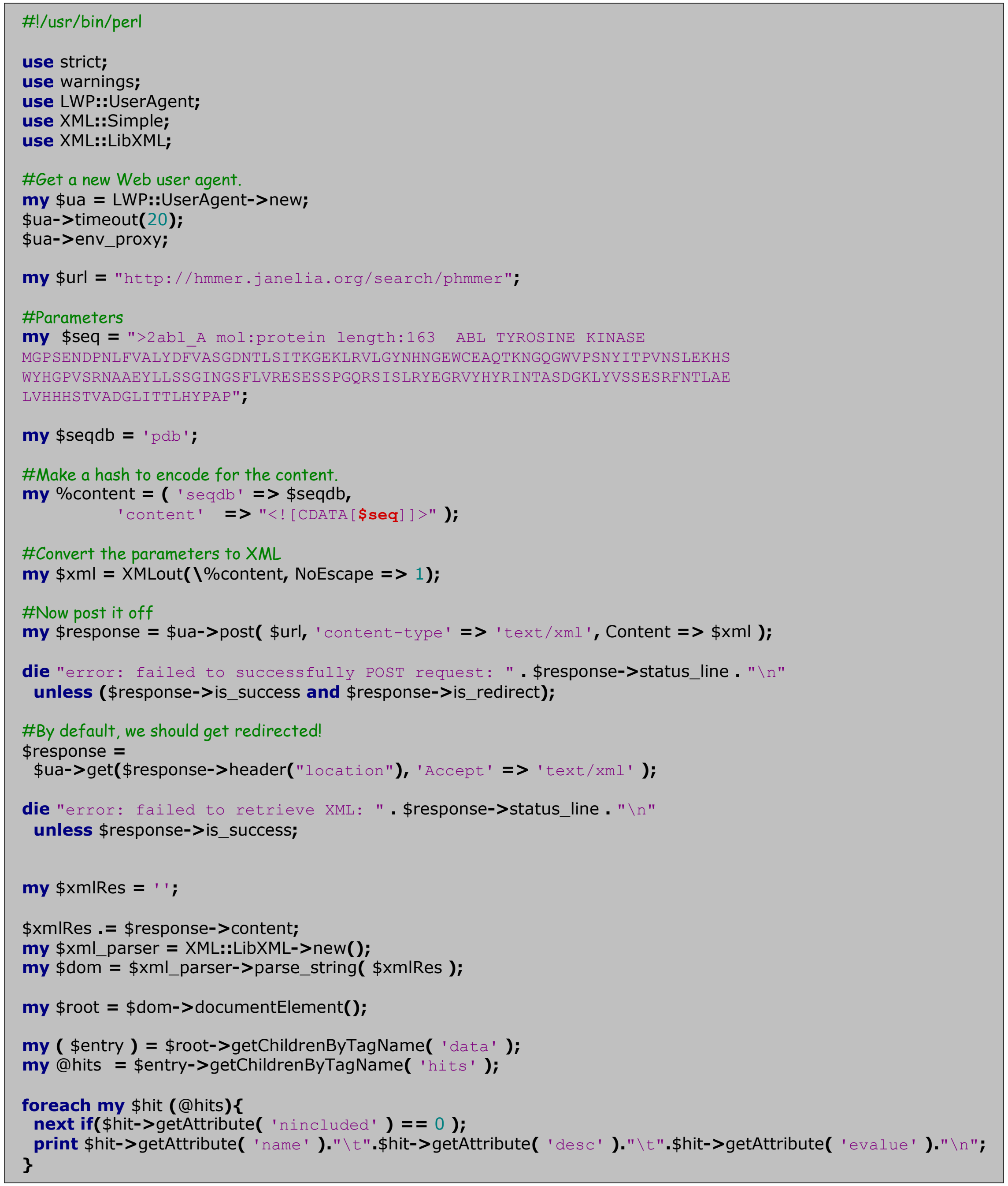
```
<?xml version="1.0" encoding="UTF-8"?>
<opt>
  <data name='results' resultSize='224339'>
    <_internal highbit='370.5' lowbit='19.0' numberSig='242' offset='42280'>
      <timings search='0.283351' unpack='0.176821' />
    </_internal>
    <hits
        name='2abl_A'
        acc='2abl_A'
        bias='0.1'
        desc='mol:protein length:163  ABL TYROSINE KINASE'
        evalue='1.1e-110'
        ndom='1'
        nincluded='1'
        nregions='1'
        reported='1'
        score='370.5'
        species='Homo sapiens'
        taxid='9606' >
            <domains
                aliL='163'
                aliM='163'
                aliN='163'
                aliaseq='MGPSENDPNLFVALYDFVASGDNTLSITKGEKLRVLGYNHNGEWCEAQTKNGQGWVPSNYITPVNSLEKHSWYHGPVSRNAAEYLLSSGINGSFLVRESESSPGQRSIS
LRYEGRVYHYRINTASDGKLYVSSESRFNTLAELVHHHSTVADGLITTLHYPAP'
                alihmmfrom='1'
                alihmmname='2abl_A'
                alihmmto='163'
                alimline='+gpsendpnlfvalydfvasgdntlsitkgeklrvlgynhngewceaqtkngqgwvpsnyitpvnslekhswyhgpvsrnaaeyllssgingsflvresesspgqrsislryegrvyhyrintasdgklyvssesrf
ntlaelvhhhstvadglittlhypap'
                alimodel='lgpsendpnlfvalydfvasgdntlsitkgeklrvlgynhngewceaqtkngqgwvpsnyitpvnslekhswyhgpvsrnaaeyllssgingsflvresesspgqrsislryegrvyhyrintasdgklyvssesrfn
tlaelvhhhstvadglittlhypap'
                alippline='8*****************************************************************************************************************
***********************************************9'
                alisqacc='2abl_A'
                alisqdesc='mol:protein length:163  ABL TYROSINE KINASE'
                alisqfrom='1'
                alisqname='2abl_A'
                alisqto='163'
                bias='0.05'
                bitscore='370.357543945312'
                envsc='250.653518676758'
                cevalue='4.21e-121'
                ievalue='4.21e-121'
                                iali='1'
                ienv='1'
                is_included='1'
                is_reported='1'
                jali='163'
                jenv='163'
            />
    </hits>
    .
    .
    .
  </data>
</opt>
```

HMMER (perl)

send

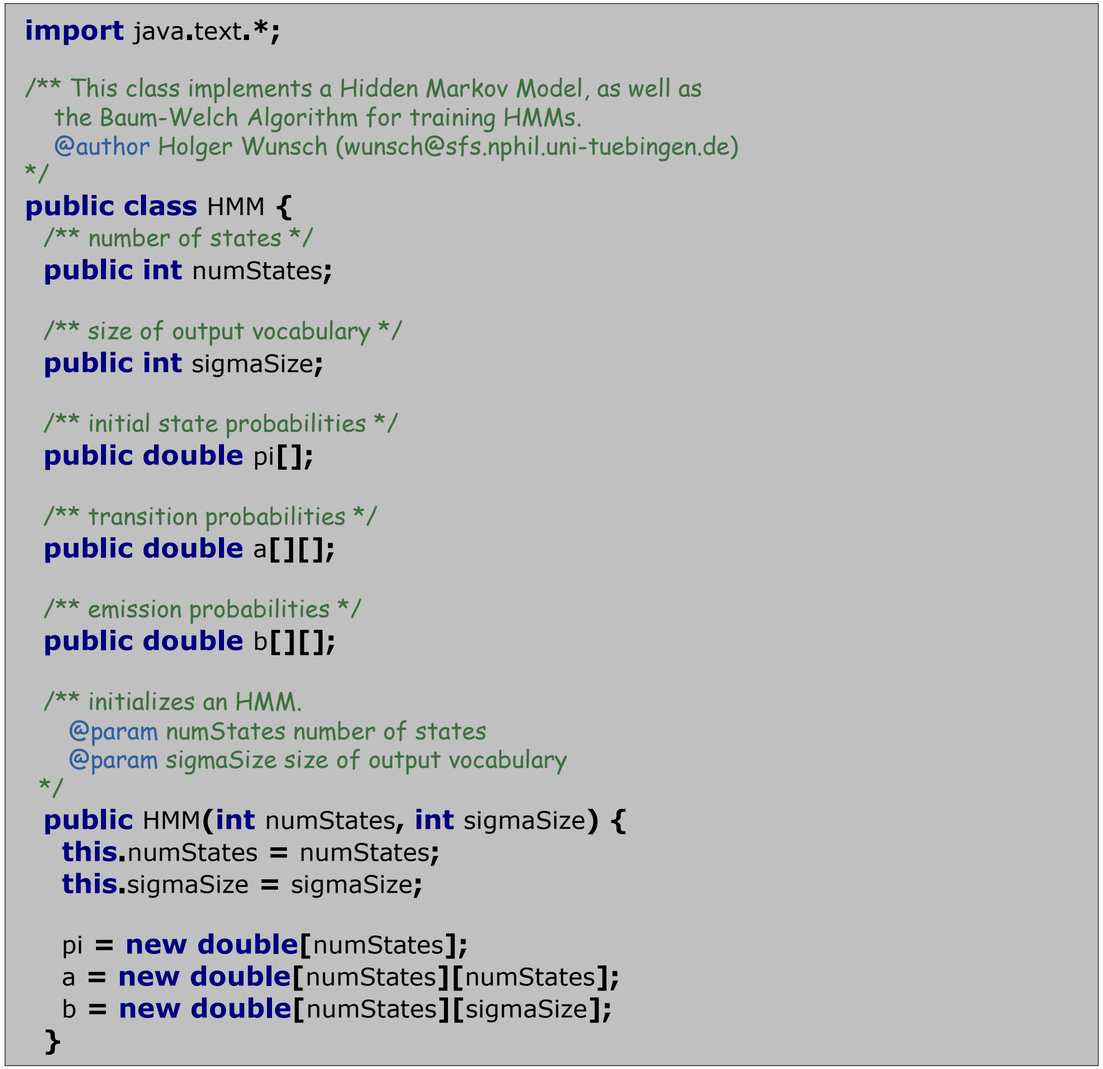
```
#!/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;
use XML::Simple;

#Get a new Web user agent.
my $ua = LWP::UserAgent->new;
$ua->timeout(20);
$ua->env_proxy;

my $url = "http://hmmer.janelia.org/search/phmmer";

#Parameters
my  $seq = ">2abl_A mol:protein length:163  ABL TYROSINE KINASE
MGPSENDPNLFVALYDFVASGDNTLSITKGEKLRVLGYNHNGEWCEAQTKNGQGWVPSNYITPVNSLEKHS
WYHGPVSRNAAEYLLSSGINGSFLVRESESSPGQRSISLRYEGRVYHYRINTASDGKLYVSSESRFNTLAE
LVHHHSTVADGLITTLHYPAP";

my $seqdb = 'pdb';

#Make a hash to encode for the content.
my %content = ( 'seqdb' => $seqdb,
          'content'  => "<![CDATA[$seq]]>" );

#Convert the parameters to XML
my $xml = XMLout(\%content, NoEscape => 1);

#Now post it off
my $response = $ua->post( $url, 'content-type' => 'text/xml', Content => $xml );

#By default, we should get redirected!
if($response->is_redirect){

  #Now make a second requests, a get this time, to get the results.
  $response =
  $ua->get($response->header("location"), 'Accept' => 'text/xml' );

  if($response->is_success){
    print $response->content;
  }else{
    print "Error with redirect GET:".$response->content;
    die $response->status_line;
  }
}else{
  die $response->status_line;
}
```

retrieve



```
#!/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;
use XML::Simple;
use XML::LibXML;

#Get a new Web user agent.
my $ua = LWP::UserAgent->new;
$ua->timeout(20);
$ua->env_proxy;

my $url = "http://hmmer.janelia.org/search/phmmer";

#Parameters
my  $seq = ">2abl_A mol:protein length:163  ABL TYROSINE KINASE
MGPSENDPNLFVALYDFVASGDNTLSITKGEKLRVLGYNHNGEWCEAQTKNGQGWVPSNYITPVNSLEKHS
WYHGPVSRNAAEYLLSSGINGSFLVRESESSPGQRSISLRYEGRVYHYRINTASDGKLYVSSESRFNTLAE
LVHHHSTVADGLITTLHYPAP";

my $seqdb = 'pdb';

#Make a hash to encode for the content.
my %content = ( 'seqdb' => $seqdb,
          'content'  => "<![CDATA[$seq]]>" );

#Convert the parameters to XML
my $xml = XMLout(\%content, NoEscape => 1);

#Now post it off
my $response = $ua->post( $url, 'content-type' => 'text/xml', Content => $xml );

die "error: failed to successfully POST request: " . $response->status_line . "\n"
  unless ($response->is_success and $response->is_redirect);

#By default, we should get redirected!
$response =
  $ua->get($response->header("location"), 'Accept' => 'text/xml' );

die "error: failed to retrieve XML: " . $response->status_line . "\n"
  unless $response->is_success;


my $xmlRes = '';

$xmlRes .= $response->content;
my $xml_parser = XML::LibXML->new();
my $dom = $xml_parser->parse_string( $xmlRes );

my $root = $dom->documentElement();

my ( $entry ) = $root->getChildrenByTagName( 'data' );
my @hits  = $entry->getChildrenByTagName( 'hits' );

foreach my $hit (@hits){
  next if($hit->getAttribute( 'nincluded' ) == 0 );
  print $hit->getAttribute( 'name' )."\t".$hit->getAttribute( 'desc' )."\t".$hit->getAttribute( 'evalue' )."\n";
}
```

Hmm.java



```java
import java.text.*;

/** This class implements a Hidden Markov Model, as well as
   the Baum-Welch Algorithm for training HMMs.
   @author Holger Wunsch (wunsch@sfs.nphil.uni-tuebingen.de)
*/
public class HMM {
  /** number of states */
  public int numStates;

  /** size of output vocabulary */
  public int sigmaSize;

  /** initial state probabilities */
  public double pi[];

  /** transition probabilities */
  public double a[][];

  /** emission probabilities */
  public double b[][];

  /** initializes an HMM.
     @param numStates number of states
     @param sigmaSize size of output vocabulary
   */
  public HMM(int numStates, int sigmaSize) {
    this.numStates = numStates;
    this.sigmaSize = sigmaSize;

    pi = new double[numStates];
    a = new double[numStates][numStates];
    b = new double[numStates][sigmaSize];
  }
```

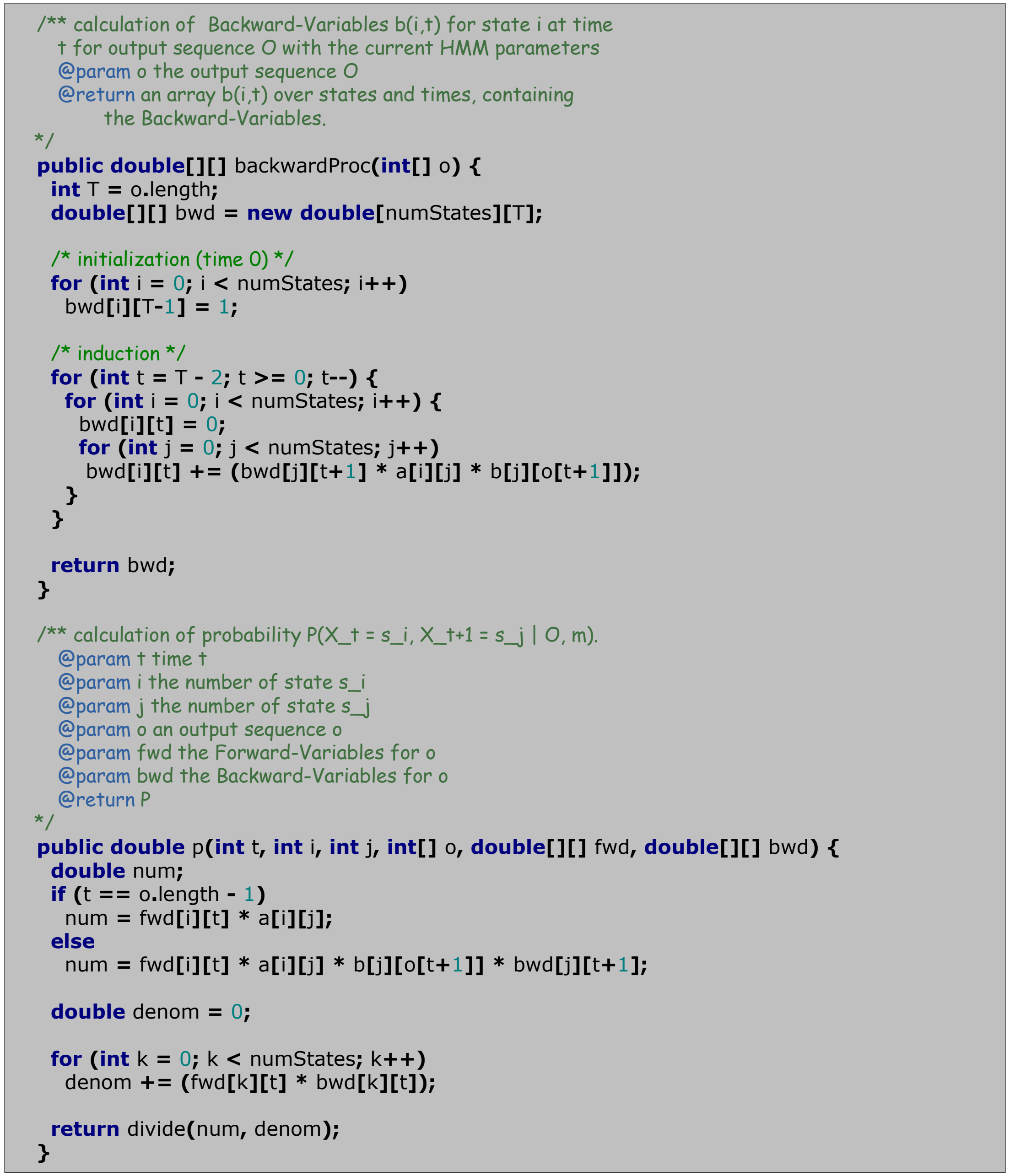
Hmm.java

```java
/** implementation of the Baum-Welch Algorithm for HMMs.
   @param o the training set
   @param steps the number of steps
*/
public void train(int[] o, int steps) {
  int T = o.length;
  double[][] fwd;
  double[][] bwd;

  double pi1[] = new double[numStates];
  double a1[][] = new double[numStates][numStates];
  double b1[][] = new double[numStates][sigmaSize];

  for (int s = 0; s < steps; s++) {
    /* calculation of Forward- und Backward Variables from the
    current model */
    fwd = forwardProc(o);
    bwd = backwardProc(o);

    /* re-estimation of initial state probabilities */
    for (int i = 0; i < numStates; i++)
      pi1[i] = gamma(i, 0, o, fwd, bwd);

    /* re-estimation of transition probabilities */
    for (int i = 0; i < numStates; i++) {
      for (int j = 0; j < numStates; j++) {
        double num = 0;
        double denom = 0;
        for (int t = 0; t <= T - 1; t++) {
          num += p(t, i, j, o, fwd, bwd);
          denom += gamma(i, t, o, fwd, bwd);
        }
        a1[i][j] = divide(num, denom);
      }
    }
```

Hmm.java

```
  /* re-estimation of emission probabilities */
  for (int i = 0; i < numStates; i++) {
   for (int k = 0; k < sigmaSize; k++) {
    double num = 0;
    double denom = 0;

    for (int t = 0; t <= T - 1; t++) {
     double g = gamma(i, t, o, fwd, bwd);
     num += g * (k == o[t] ? 1 : 0);
     denom += g;
    }
    b1[i][k] = divide(num, denom);
   }
  }
  pi = pi1;
  a = a1;
  b = b1;
 }
}


/** calculation of Forward-Variables f(i,t) for state i at time
  t for output sequence O with the current HMM parameters
  @param o the output sequence O
  @return an array f(i,t) over states and times, containing
      the Forward-variables.
*/
public double[][] forwardProc(int[] o) {
 int T = o.length;
 double[][] fwd = new double[numStates][T];

 /* initialization (time 0) */
 for (int i = 0; i < numStates; i++)
  fwd[i][0] = pi[i] * b[i][o[0]];

 /* induction */
 for (int t = 0; t <= T-2; t++) {
  for (int j = 0; j < numStates; j++) {
   fwd[j][t+1] = 0;
   for (int i = 0; i < numStates; i++)
    fwd[j][t+1] += (fwd[i][t] * a[i][j]);
   fwd[j][t+1] *= b[j][o[t+1]];
  }
 }

 return fwd;
}
```

Hmm.java



```java
/** calculation of  Backward-Variables b(i,t) for state i at time
  t for output sequence O with the current HMM parameters
  @param o the output sequence O
  @return an array b(i,t) over states and times, containing
       the Backward-Variables.
*/
public double[][] backwardProc(int[] o) {
  int T = o.length;
  double[][] bwd = new double[numStates][T];

  /* initialization (time 0) */
  for (int i = 0; i < numStates; i++)
    bwd[i][T-1] = 1;

  /* induction */
  for (int t = T - 2; t >= 0; t--) {
    for (int i = 0; i < numStates; i++) {
      bwd[i][t] = 0;
      for (int j = 0; j < numStates; j++)
       bwd[i][t] += (bwd[j][t+1] * a[i][j] * b[j][o[t+1]]);
    }
  }

  return bwd;
}

/** calculation of probability P(X_t = s_i, X_t+1 = s_j | O, m).
  @param t time t
  @param i the number of state s_i
  @param j the number of state s_j
  @param o an output sequence o
  @param fwd the Forward-Variables for o
  @param bwd the Backward-Variables for o
  @return P
*/
public double p(int t, int i, int j, int[] o, double[][] fwd, double[][] bwd) {
  double num;
  if (t == o.length - 1)
    num = fwd[i][t] * a[i][j];
  else
    num = fwd[i][t] * a[i][j] * b[j][o[t+1]] * bwd[j][t+1];

  double denom = 0;

  for (int k = 0; k < numStates; k++)
    denom += (fwd[k][t] * bwd[k][t]);

  return divide(num, denom);
}
```

Hmm.java

```java
/** computes gamma(i, t) */
public double gamma(int i, int t, int[] o, double[][] fwd, double[][] bwd) {
  double num = fwd[i][t] * bwd[i][t];
  double denom = 0;

  for (int j = 0; j < numStates; j++)
    denom += fwd[j][t] * bwd[j][t];

  return divide(num, denom);
}

/** prints all the parameters of an HMM */
public void print() {
  DecimalFormat fmt = new DecimalFormat();
  fmt.setMinimumFractionDigits(5);
  fmt.setMaximumFractionDigits(5);

  for (int i = 0; i < numStates; i++)
    System.out.println("pi(" + i + ") = " + fmt.format(pi[i]));
  System.out.println();

  for (int i = 0; i < numStates; i++) {
    for (int j = 0; j < numStates; j++)
      System.out.print("a(" + i + "," + j + ") = " +
                       fmt.format(a[i][j]) + "  ");
    System.out.println();
  }

  System.out.println();
  for (int i = 0; i < numStates; i++) {
    for (int k = 0; k < sigmaSize; k++)
      System.out.print("b(" + i + "," + k + ") = " +
                       fmt.format(b[i][k]) + "  ");
    System.out.println();
  }
}

/** divides two doubles. 0 / 0 = 0! */
public double divide(double n, double d) {
  if (n == 0)
    return 0;
  else
    return n / d;
}
}
```