Lecture Notes: Introduction to Hidden Markov Models

Introduction

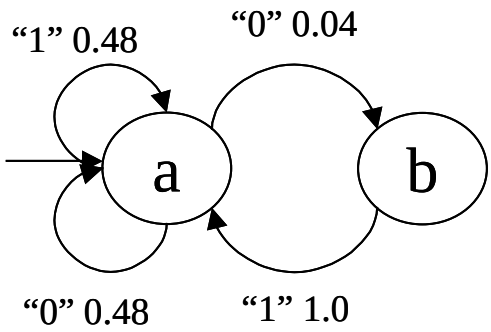
A Hidden Markov Model (HMM), as the name suggests, is a Markov model in which the states cannot be observed but symbols that are consumed or produced by transition are observable. A speech generation system might, for example, be implemented as a HMM and speak a word as it transitions from one state to another. Similarly a speech understanding system might "recognize" a word on a transition. In this sense HMM's can be thought of as generative or as interpretative. The HMM is the same but in one case the transitions emit symbols (such as words) and in the other case consumes symbols. We will therefore treat observations and actions interchangeably in the foregoing.

Hidden Markov Models and simple extensions of them are very popular in a variety of fields including computer vision, natural language understanding, and speech recognition and synthesis (to name a few). Often HMM's are a natural way of modeling a system and in other cases they are "force-fit" to a problem to which they are not quite ideal. The immense popularity of HMM's is that very fast, linear time, algorithms exist for some of the most important HMM problems. This allows, for example, speech recognition systems to operate in real-time.

A HMM is defined as the four tuple $<s_1,S,W,E>$ where s_1 is the start state, S is the set of states, W is the set of observation symbols, and E is the set of transitions. A transition is also a four tuple such as $<s_2$,"had", s_3, 0.3>. This example described a transition from state s_2 to s_3 in which the word "had" is either emitted or consumed and the probability of taking the transition is 0.3. We will usually write a transition as T(s_2,"had", s_3, 0.3) or as:

$$P(s_2 \xrightarrow{"had"} s_3) = 0.3$$

Sometimes we do not know the starting state but we have a pdf for the starting state. We can therefore, with improved generality replace s_1 with a starting state pdf. We will continue to assume that we know the starting state for the remainder of this discussion in order to simplify examples but generalizing the starting condition to a pdf adds no additional complexity to the algorithms presented.



Consider the, very simple, example below:
The transitions are depicted as arcs that indicate the symbol that is consumed or emitted, in quotes, as the transition is taken and a number that indicates the probability that a transition is taken.

A couple of points are worth noting at this point. First, the probabilities of all transitions from a state must sum to 1.0 and second, multiple transitions out of a state can occur with the same symbol. Looking at state “a” in the figure above shows that the symbol “0” can be consumed by two different transitions. One of them changes the state to “b” while the other leaves the system in state “a”. It is because of this that the states cannot be known deterministically. If transitions could be uniquely identified by the symbol they produce/consume we would track, with certainty, the current state of the system.

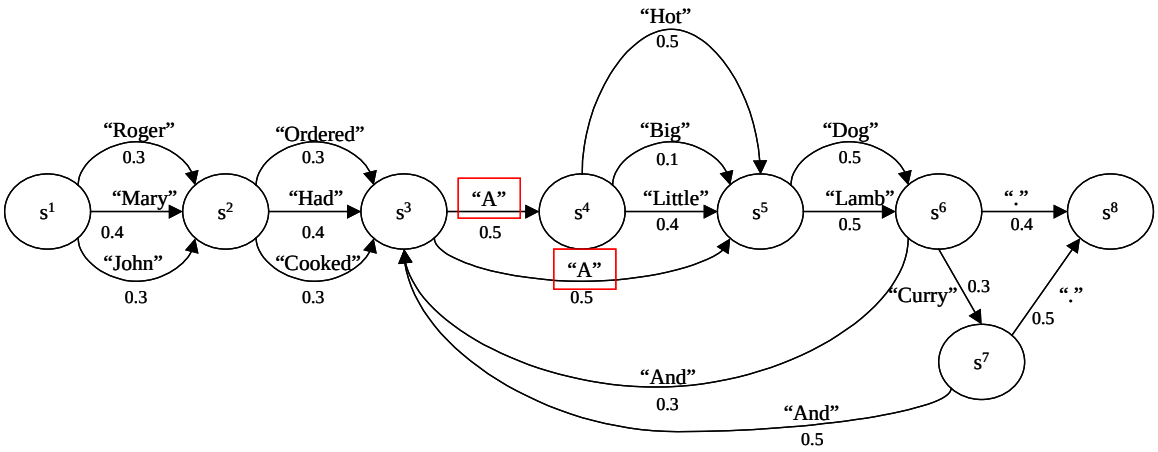
The above example can be described as **$<s_1, S, W, E>$**

Where:

S_1 = a
S = { a, b }
W = { 0, 1 }
E = { <a, “1”, a, 0.48> <a, “0”, a, 0.48> <a, “0”, b, 0.04> <b, “1”, a, 1.0> }

A sentence parsing example

Let's look at a more interesting example.



The HMM depicted above defines a very simple sentence generator. Notice that from state s_3 there are two transitions that emit the word "A". Imagine that, starting from state s_1 the words "Roger Ordered A" had been emitted. We know that the first state was s_1, the second state was s_2, and the third state was s_3 because given the emitted words there is no ambiguity. After the word "A" has been emitted however we do not know whether the system is in state s_4 or s_5 and, given the transition probabilities, would rationally assign equal likelihood to each of those states. As it happens the ambiguity is resolved as soon as the next word is emitted. If the next word is "Hot" we know that the current state is S_5 and the preceding state that had formerly been ambiguous must have been S_4.

Examples of some of the sentences that can be produced by this model are:

S_1: Mary had a little Lamb and a big dog.
S_2: Roger ordered a lamb curry and a hot dog.
S_3: John cooked a hot dog curry.

We can calculate the probability that a word sequence, such as S_3, be emitted by this HMM by multiplying the probabilities of the transitions taken in order to produce the sentence.

$P(S_3)=0.3*0.3*0.5*0.5*0.3*0.5=0.003375$
We can do this because we know that the transitions are conditionally independent. This is the Markovian assumption. S_3 is a six word sequence and we will often refer to an **n** symbol sequence as $w^{1,n}$. [1]

[1] W is used to denote 'observation' because the Greek lowercase Omega (for Observation) looks like a "w".

Finding out the probability of a sequence of transitions is sometimes a useful thing to do, as we will soon see, but it is by no means the only thing that we wish to do. Finding the probability of a sequence of observations or actions is referred to as **Evaluation** and can be computed in both the forward and the reverse direction (i.e. working forward from the first observation, or working back from the last observation). We will shortly introduce algorithms for doing this but first let consider some other things that we wish to compute.

We have already alluded to the problem of inferring the state sequence from the observation. In general we cannot know for sure what sequence of states will be taken for a given sequence of observations although sometimes, such as in the above example, we can. In general the best that we can do is to estimate the most likely state trajectory for a given sequence of observations. This problem is often referred to as **Decoding**.

Another common requirement is to learn the probabilities associated with transitions in the system by being given a representative training sequence rather than being given the transition probabilities directly. The idea is to find the set of transition probabilities that maximizes the likelihood of the training sequence. Not surprisingly this is the **Learning** problem.

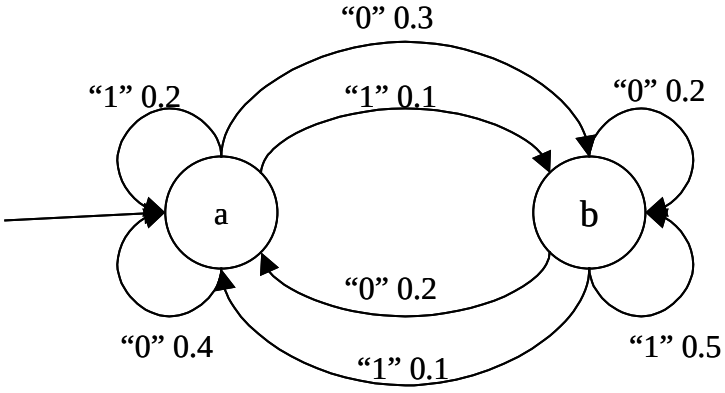
HMM Decoding - Finding the most likely path

We begin with the decoding problem by introducing the Viterbi algorithm for finding the most likely sequence of states given a set of observations.

$$\sigma(t) = \arg\max_{s^{1,t}} P(s^{1,t} \mid w^{1,t-1})$$

A sequence of **t-1** observations will result in a sequence of **t** states. In the foregoing we represent the most likely sequence of **t** states given a sequence of observations $w^{1,t-1}$ as $\sigma(t)$. We will use the operator **o** to concatenate new states onto an existing state sequence. So for example, if $\sigma(3)$ is the state sequence $s_1\ s_2\ s_3$, $\sigma(3)$ o s_4 is the sequence $s_1\ s_2\ s_3\ s_4$.

To illustrate the Viterbi algorithm we will use the HMM depicted in the figure below.



Given the known starting state **a** we know that before any observations are made that P(**a**)=1.0 and P(**b**)=0.0. After the first observation the state sequence must therefore be

either **aa** or **ab**. At each state a state might transition to one of many different states one might therefore naively assume that the algorithm would be exponential because there is a branching factor involved at each observation. Surprisingly and delightfully this is not the case this is not the case and the algorithm is, as we advertised earlier, linear! The reasoning is as follows:

We are not interested in finding all possible state sequences—just the most likely state sequence. At an arbitrary point before all of the observations have been seen we cannot know what is the most likely sequence. What is currently the most likely sequence might be eliminated completely when the next observations comes along if there is no transition that extends the previously most likely sequence with the observation. Consequently the currently most promising sequence might seem very unlikely if the latest transition has a very low probability or impossible if there is no such transition. Imaging the case where the only transition that could possibly account for the latest observation is from one state. We will need to have maintained, in our algorithm, a sequence that ends in that state so that we can extend it with the latest observation. We must therefore maintain for every state the most likely sequence that ends in that state. There may of course be many other ways of ending in that state but since we are looking for the most likely sequence there is nothing to be gained by maintaining anything but the one that has the highest probability. The Viterbi algorithm therefore works by maintaining for each state:

1. The most likely sequence of states that ends in that state, and
2. The probability of that sequence. Sometimes the probability will be zero.

The table below shows the steps of the Viterbi algorithm as it processes the observations (starting from ε the empty sequence of observations). At each step, for each state, we calculate the probability of extending the previous state sequences so as to end in the state in question. So we can get from state b to state b with an observation of "1" using the transition T(b,"1",b,0.5) but since the probability of the state sequence **b** was 0.0 the probability of the sequence **bb** is also 0.0 whereas the probability of the sequence **ab** is 1.0*0.1=0.1. As you can see the table grows linearly with the number of observations and only the previous time step needs to be stored between iterations so the space utilized

States		ε	1	11	111	1110
a	Sequence	a	aa	aaa	aaaa	abbba
	Probability	1.0	0.2	0.04	0.008	0.005
b	Sequence	b	ab	abb	abbb	abbbb
	Probability	0.0	0.1	0.05	0.025	0.005

is constant and the time grows linearly with observations.

Formally, we can describe the Viterbi algorithm as follows:

For a sequence of T visible actions W^T and a HMM with c hidden states.
Let $\sigma^i(t)$ be the maximum likelihood path that accounts for the observation/action sequence W^T and which ends in state s_i.Below is the pseudo code for the Viterbi

$$\sigma^i(1) = s_i$$

$$\sigma^i(t+1) = \sigma^j(t) \circ s_i,\ j = \arg\max_{k=1}^{c} P(\sigma^k(t)) P(s_k \xrightarrow{w_t} s_i)$$

algorithm.

```
1.  Begin
2.    Initialize σ^i(1) ← {s^i}, α^i(1) ← {1 if s^i is start state, 0 otherwise}
3.    for (t ← 1; t ← t+1; t ≤ T) {            // for each obs
4.     initialize j ← 0, p ← -1
5.     for (m ← 0; m < c; m ← m+1)  {   // each state
6.       for (k ←0; k < c; k ← k+1)         // compute arg max for j
7.         if (α^k(t) tprob(s^k, s^m, w^t)>p)
8.           p ←α^k(t) tprob(s^k, s^m, w^t) and  j ← k
9.       α^m(t+1) ← p
10.      σ^m(t+1) ← σ^j(t) o {s^m}    // concatenate on the next state
11.   return σ^m(T+1) with m that maximizes α^m(T+1)
12. end
```

HMM Evaluation - HMM forward probabilities

Using the same HMM as before, repeated below for convenience, consider the observation sequence "1110". We wish to calculate the probability of that sequence being generated by the HMM.
We begin by considering the observations in the order that they are observed—the so-called forward probabilities. In this algorithm we are not concerned with a sequence of states. Instead, we want to know the sum of the probabilities of all paths that account for the sequence of observations. In principle the final state can be any of the states. Let $\alpha^i(t)$ be the probability $P(w^{1,t-1}, s^t = s_i$. Clearly if we know $\alpha^i(t)$ for all states **i** we can calculate the probabilities of $\alpha^i(t+1)$ for all states **i** by simply extending computing and summing all extensions of the previous states. This is precisely what we do for the forward-probabilities algorithm. To get $P(w^{1,t})$ therefore we simply sum the probabilities of all sequences ending in each of the states.
The table below shows the steps of the forward-probabilities algorithm for the sequence of observations "1110".

Consider the entry for $\alpha^i(3)=0.07$. This is calculated by extending the path that ends in state **a** with the transition "1" with probability 0.1 hence 0.2*0.1=0.02 and extending the

path that ends in state **b** with the transition "1" with probability 0.5 hence 0.1*0.5=0.05. Summing these two routes yields 0.02+0.05=0.07.
More formally we can describe this algorithm as follows:

Let $\alpha^i(t)$ be the probability $P(w^{1,t-1}, s^t=s_i)$.

$$P(w^{1,T}) = \sum_{i=1}^{c} P(w^{1,T}, S^{T+1} = s_i)$$

The base case for our recursive definition is the starting state before any observations are

$$P(w^{1,T}) = \sum_{i=1}^{c} \alpha^i(T+1)$$

seen. We must start in the start state so:

$$\alpha^i(1) = P(w^{1,0}, s^1 = s_i) = \begin{cases} i = 1 \rightarrow 1.0 \\ otherwise \rightarrow 0 \end{cases}$$

and the recursive step extends it

$$\alpha^j(t+1) = P(w^{1,t}, s^{t+1} = s_j) = \sum_{i=1}^{c} P(w^{1,t-1}, s^t = s_i) P(s_i \xrightarrow{w_t} s_j)$$

The pseudo code for the forward probabilities is very given below:For a sequence of T visible actions V^T and a HMM with c hidden states where tprob(a,b,c) is the transition probability for transitioning from state a to state b with observation/action c

```
1.  Begin
2.     Initialize α^i(1) ← {1 if s^i is start state, 0 otherwise}
3.     for (t ← 1; t ← t+1; t ≤ T) {
4.       initialize for all values of i
5.       for (j ← 0; j ← j+1; j < c)
6.         initialize α^j(t+1) ← 0
7.         for (i ← 0; i ← i+1; i < c)
8.           α^j(t+1) ← α^j(t+1) + α^i(t)tprob(s^i, s^j, w^t)
9.     }
10.    return sum α^i(T+1) summed over all values of i (states)
11. end
```

Backward Probabilities

Backward probabilities work in exactly the same way as forward probabilities but we start from the final observation and work backwards. You may be wondering why we

would ever want to do such a thing. It turns out that it is useful for solving the training problem, which we will describe next.

We can define $\beta^i(t)$ just as we did for $\alpha^i(t)$ but starting from the end.

$$\beta^i(t) \equiv P(w^{t,T} \mid s^t = s_i)$$

Notice that it is the starting state that is constrained to be s_i and not, as it was for $\alpha^i(t)$, the ending state.
It is interesting to note that

$$\beta^1(1) = P(w^{1,T} \mid s^1 = s_1)$$
$$= P(w^{1,T})$$

which follows because a sequence of one state must be the start state.

The base case of our recursive definition is similar to before but starts, as we would expect, from the end.

$$\beta^i(T+1) = P(\varepsilon \mid s^{T+1} = s_i) = 1$$

The starting state of the entire sequence is, by definition, the start state. W can work back with our recursive definition just as we did before. Notice that we are using state s_j, the target state of the transitions, rather than s_i in this definition.

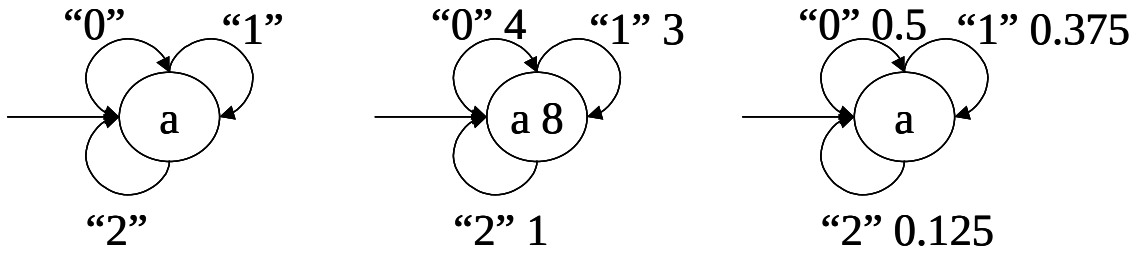
$$\beta^j(t-1) = \sum_{j=1}^{c} P(s_i \xrightarrow{w_{t-1}} s_j)\beta^j(t) = \sum_{j=1}^{c} P(s_i \xrightarrow{w_{t-1}} s_j)P(w^{t,T} \mid s^t = s_j)$$

The pseudo code for backward-probabilities is very similar to that of forward-probabilities so we will leave it as an exercise for the reader.

HMM Training - (Baum-Welch Algorithm)

We now have enough apparatus to attack the learning problem. In the learning problem we start out with the HMM definition without the transition probabilities. Our goal is to find the set of transition probabilities that maximize the likelihood of the training sequence provided. It should be noted from the outset that the algorithm described, the Baum-Welch algorithm also known as the forward-backward algorithm does not guarantee to find the global maximum. It finds the local maximum and as such its usefulness depends upon the HMM being trained.

Consider a very simple case of an HMM with a single state and three transitions. Such an HMM is depicted below left without transition probabilities.



Training Sequence: 01010210

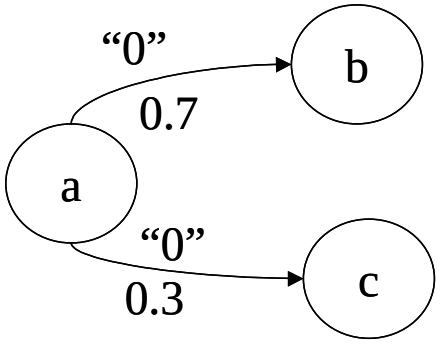
Given a training sequence such as “01010210” we can play the observations through the model counting the number of times each transition is taken and the number of times each state takes a transition. The result of this that with our trivial example is that state a takes 8 transitions, the transition “0” is taken four times, the transition “1” is taken three times, and the transition “2” is taken just once (left). We can obtain from these counts a consistent set of transition probabilities by dividing the number of times a transition is taken by the number of transitions taken by the state from which the transition transitions (right).

More formally, If C is a function that counts the number of times that a transition is taken when the training sequence is run through the model, we can estimate the transition probabilities as follows:

$$P_e(s_i \xrightarrow{w_k} s_j) = \frac{C(s_i \xrightarrow{w_k} s_j)}{\sum_{l=1}^{c} \sum_{m=1}^{\omega} C(s_i \xrightarrow{w_m} s_l)}$$

It is fun to show prove that the transition probabilities resulting from the above algorithm do in fact maximize the likelihood of the training sequence. That proof is left as an exercise for the reader.

If we could deterministically follow transitions in this way we would be done but we cannot because there may be, as we discussed earlier, multiple transitions out of a state that have the same observation/action.



Consider the example above. There are two transitions out of state **a** for the observation "0". One has probability 0.7 and the other 0.3. This suggests a solution. Instead of counting the number of times a transition is taken count instead the prorated amount of transitions taken. So in this case we would count 0.7 for the transition to state b and 0.3 for the transition to state c. We would count 0.7+0.3=1.0 for the number of transitions taken out of **a** (instead of 2).

This solves our problem of ambiguous transitions but one small problem remains: We don't know the transition probabilities because they are what we are trying to learn!

It turns out that if we guess a set of transition values and then count the transitions using the prorated scheme described above the resulting transition probabilities will have improved somewhat towards a local maximum and so by iterating as long as an improvement occurs we can find the local maximum.

1.Guess a set of transition probabilities.
2.while (improving) {
3. propagate-training-sequences
4.}

There are two obvious ways of calculating "improving" in the pseudo code above. The conceptually easiest way is to scan through the old and new transition probabilities to see which has changed the most. When the maximum change of any transition drops below a predefined accuracy θ the training loop can be exited. The problem with this approach is that it involves scanning through all transition probabilities—of which there may be a very large number. An alternative, and computationally less expensive approach, is to calculate the cross-entropy after each iteration. When the cross-entropy decreases by less than θ we are done.

Cross entropy is:

$$-\frac{1}{n}\sum_{w^{1,T}} P^{M-1}(w^{1,T})\log_2 P^{M}(w^{1,T})$$

Where P^{M-1} is the probability as estimated by the previous iteration's model and P^{M} is the probability as estimated by the current iteration's model. Cross-entropy is a common measure used to compare models and avoids the cost of scanning through all of the transitions.

All that remains then is to formalize the calculation of the transition counts for the training sequence.

Consider a training sequence consisting of T observations. There will be T+1 states chained together by T transitions. The prorated count of a transition is the number of times that the transition was taken during the training sequence. We can calculate the probability of the entire sequence $P(w^{1,T})$ as the backward-probability $\beta^{i}(1)$. In order to consider the transitions imagine that we cut the chain of transitions at some point t at which a transitions takes the system from state s_i to s_j. The probability of the sequence can now be written as the product of the probability of the part preceding the transition $\alpha^{i}(t)$, the probability of the transition itself, and the probability of the rest of the sequence $\beta^{j}(t+1)$. This probability is not the same as $\beta^{i}(1)$ because it picks a particular transition at time t. If we summed over all possible transitions at time t we would get $P(w^{1,T})= \beta^{i}(1)$. We can divide by $P(w^{1,T})$ to get the probability of that particular transition happening at time step t and we can get the prorated transition count for that transition by summing over all possible places in the sequence where that transition had an opportunity to happen.

$$C(s_i \overset{w_k}{\rightarrow} s_j) = \frac{1}{\beta^{i}(1)} \sum_{t=1}^{T} \alpha^{i}(t) P(s_i \overset{w_k}{\rightarrow} s_j) \beta^{j}(t+1)$$

We can use this simple equation to calculate the counts used in the equation

$$P_e(s_i \overset{w_k}{\rightarrow} s_j) = \frac{C(s_i \overset{w_k}{\rightarrow} s_j)}{\sum_{l=1}^{c} \sum_{m=1}^{\omega} C(s_i \overset{w_m}{\rightarrow} s_l)}$$

(repeated from above for convenience). Notice that the $1/\beta^{i}(1)$ occurs in both the numerator and the denominator and can thus be dropped from the count calculation.

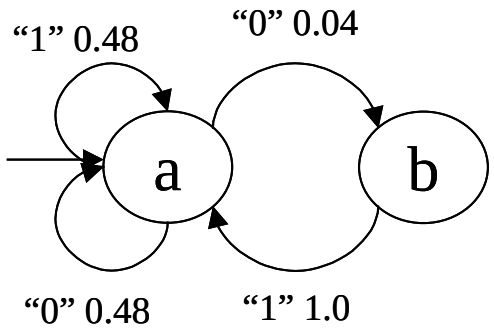
Baum-Welch Pseudo Code

The pseudo code for this algorithm is shown below followed by a hand worked example.

```
1.  do
2.    initialize training sequence w^T,  convergence criteria θ;
3.    oldtprob=tprob                  // remember old values
4.    compute α^i(t), β^i(t) for all values of i and t
5.    for all values of i, j, k, and t
6.      C(s^i, s^j, w^k)(t) ← α^i(t) tprob(s^i, s^j, w^k) β^j(t)
7.    for (i ← 0; i<c; i ← i+1) {
8.      initialize t^i ← 0
9.      for (j ← 0; j<c; j ← j+1)
10.       for (s in Symbols) t^i ← t^i + C(s^i, s^j, w^s)
11.   for (j ← 0; j<c; j ← j+1)
12.     for (s in Symbols)  tprob(s^i,s^j,w^s)=C(s^i,s^j,w^s)/t^i
13.  }
14.  while (maxChanged(tprob, oldtprob)> θ)
```

Baum-Welch example

Consider the following, very simple, example consisting of two states and four transitions with the training sequence "01011". The probabilities on the transitions are our "starting guess".



First we calculate α for all of our states (**a** and **b**) for all of the steps of our training sequence as follows (all hand calculated numbers have been rounded for readability):

	1	2	3	4	5	6
	ε	0	1	0	1	1
$\alpha_a(t)$	1	0.48	0.27	0.13	0.072	0.035
$\alpha_b(t)$	0	0.04	0	0.01	0	0

Next we calculate β for all of our states (**a** and **b**) for all of the steps of our training sequence as follows:

	1	2	3	4	5	6
	ε	0	1	0	1	1
$\beta_a(t)$	0.035	0.062	0.13	0.23	0.48	1
$\beta_b(t)$	0	0.13	0	0.28	1	1

Now, for all of our (four) transitions, we estimate the transition probabilities at each time step and then sum them to produce the total (see the total column in the table below). We can avoid the step of normalizing the transition counts (total column) because the normalization factor cancels out. The new probability P(T(a,0,b)) is calculates as:

$$\frac{C(a,0,b)}{C(a,0,b)+C(a,0,a)+C(a,1,a)} = \frac{0.01}{0.01+0.06+0.095} = \frac{0.01}{0.165} = 0.06$$

The numerator is the count for the specific transition whose probability is being estimated and the denominator is the sum of all transitions out of the starting state for that transition.

	0	1	0	1	1	Total	New P
T(a,0,b)	0.0052	0	0.0052	0	0	0.01	0.06
T(b,1,a)	0	0.0052	0	0.0048	0	0.01	1
T(a,0,a)	0.030	0	0.03	0	0	0.06	0.36
T(a,1,a)	0	0.03	0	0.03	0.035	0.095	0.58