An introduction to Hidden Markov Models

Richard A. O'Keefe

2004–2009

1 A simplistic introduction to probability

A probability is a real number between 0 and 1 inclusive which says how likely we think it is that something will happen.

Suppose there are N things that can happen, and we are interested in how likely one of them is. Let's say that the N possible events are represented by the values of a variable x, and the event we are interested in is represented by the outcome $x = 42$. Then $\Pr(x = 42)$ is the probability in question.

But how do we know what number to use? One rule which is sometimes used is "if there are N possible outcomes and you can't think of any reason why one should be any more or less likely than another, take the probability of each to be $1/N$." This is called the *rule of indifference*. It's a convenient rule, but we have no right to expect it to give the right answer.

There has been a lot of philosophical debate over the centuries about what probabilities mean. Some people think that they are or should mean people's strength of belief. The simplest interpretation, and the one I was taught, is the *frequency* interpretation. Consider a sequence of observations of x, $x_1, x_2, ...x_M$. Then $(\sum_{i=1}^{M}(x_i = 1))/M$, the number of cases where the event of interest *did* happen divided by the total number of cases where it *might* have happened, is the observed probability that $x = 1$. We can define the probability to be the limit as $M \to \infty$.

The observed probability is a good estimate of the limit probability, but it is only an estimate. As Dilbert once observed, shift happens.

The observed proportion is exactly what we want for Hidden Markov Models.

Probabilities satisfy some useful laws.

Range All probabilities are between 0 and 1 inclusive. That is, for any event e, $\Pr(e) \in [0, 1]$

Certainty All logical truths have probability 1. If event e must happen, $\Pr(e) = 1$. Conversely, all logical falsehoods have probability 0. If event e cannot happen (perhaps because we don't allow it to), $\Pr(e) = 0$.

Finite Additivity If events e and f are mutually exclusive (that is, they cannot both happen together, but neither need occur) then $\Pr(e \vee f) = \Pr(e) + \Pr(f)$.

For example, if in a certain die the face numbered 1 turns up $\frac{1}{7}$ of the time and the face numbered 6 turns up $\frac{2}{7}$ of the time, then (face 1 *or* face 2) turns up $\frac{3}{7}$ of the time.

Complement An event e cannot both happen (e) and not happen ($\neg e$). An event either happens or it doesn't ($e \vee \neg e$ is certain). So $\Pr(e)+\Pr(\neg e) = 1$ or $\Pr(\neg e) = 1 - \Pr(e)$.

For example, the probability that neither face 1 nor face 2 turns up is $1 - \frac{3}{7} = \frac{4}{7}$ for our imaginary die.

Disjunction Let e and f be any two events. If e occurs, then $e \wedge f$ might have occurred; if f occurs, then $e \wedge f$ might have occurred. So just adding up $\Pr(e) + \Pr(f)$ is too big, it counts $e \wedge f$ twice. So $\Pr(e \vee f) = \Pr(e) + \Pr(f) - \Pr(e \wedge f)$. For more events, see *The Principle of Inclusion and Exclusion* in any book on combinatorics.

For example, consider a fair die. The probability of an even number is $\frac{1}{2}$. The probability of a number greater than 3 is $\frac{1}{2}$. But the probability of (an even number or a number greater than 3) is not $\frac{1}{2} + \frac{1}{2} = 1$, because the numbers which are even and greater than 3 (4,6) have been counted twice. The right answer is $\frac{1}{2} + \frac{1}{2} - \frac{1}{3} = \frac{2}{3}$.

Implication If x happens whenever y happens, then $\Pr(x) \geq \Pr(y)$. Greater than or equal because there might be other ways for x to happen that do not involve y. For example, the probability that I will get wet is greater than the probability that it rains and I go out without an umbrella because I might have a shower.

Conjunction If e and f both occur, then e occurs. By the implication rule, $\Pr(e) \geq \Pr(e \wedge f)$. To put it another way, $\Pr(e \wedge f) \leq \min(\Pr(e), \Pr(f))$.

Independence If e_1 and f are independent events then $\Pr(e \wedge f) = \Pr(e)\Pr(f)$. Unfortunately, this is the definition of independence. But a common-sense "e does not causally affect the outcome of f, f does not causally affect the outcome of e, they have no common cause which affects them both in a related way" will get you a long way.

Given If we learn something, call it E, we move from a world in which we don't know whether E happened or not to one where we know that it did. We can expect this to alter probabilities. The chance that someone has Parkinson's disease is fairly low; the chance that someone who has a resting tremor has Parkinson's disease is rather higher. The probability that H happens *given that* we know that E has happened is written $\Pr(H|E)$.

$\Pr(H|E) = \Pr(H \wedge E)/\Pr(E)$.

Convex combination The convex combination of a set of probability distributions is itself a probability distribution. That is, suppose $p_1, \ldots, p_k : S \to [0, 1]$ are probability distributions over some space S and $w_1, \ldots, w_k \in$

$[0, 1]$ are real numbers such that $w_1 + \ldots + w_k = 1$. Then $w_1 p_1 + \ldots + w_k p_k$ is a probability distribution over S. We can see this intuitively: the sum is the distribution you get when first you make a random choice i from $\{1, \ldots, k\}$ according to the probability distribution w and then make a random choice from S according to the probability distribution p_i.

That's all the probability you need to grasp what's going on in Hidden Markov Models. If you want to explore basic probability on the Web, try `http://www.probabilitytheory.info/`. For applications in computing, *Concrete Mathematics* by Graham, Knuth, and Patashnik is a really lovely book.

1.1 Likelihood

That's all the *probability* you need, but not quite all the *statistics*. There is a concept that is bewilderingly similar to probability, but importantly different. It's called *likelihood*.

Suppose we have one probabilistic model M and several outcomes E_1, E_2, $\ldots$, E_n. We can ask "what is the probability of E_i under M", and the number we get tells us how often we should expect to see E_i happen if the experiment were repeated many time, or how strongly we should expect E_i to happen if it were just done once. In either case, we are concerned with *predicting* future events.

Suppose we have many models M_1, M_2, $\ldots$, M_k and one outcome P which has already happened. We can ask "what was the probability of E under M_j". These numbers are not the probability that E will happen. We already know it happened. Instead of telling us about the event, these numbers, called likelihoods, tell us about the models. We usually pick the model that would have made the likelihood biggest. This is called Maximal Likelihood estimation. With likelihoods, we are concerned with *explaining* past events.

2 Matching

Suppose we have a collection of sequences which we have aligned, and want to find other (sub-)sequences that look like them. How can we summarise our sequences?

- *Raw data.* We could simply keep the raw data and match a (sub-)sequence against them. We could look for exact matches. There are plenty of algorithms for searching for "any of a set of strings". A simple one is to build a TRIE holding all the strings and then walk along the new sequence matching each prefix against the trie. This would cost $O(mn)$ where m is the longest sequence in our initial collection and n is the length of the new sequence.

  We could try alignment against each sequence.

- *Consensus sequences.* To get a consensus sequence from an alignment, pick the commonest entry in each column. If there is a tie, choose arbitrarily from the commonest entries, or use a wild card. For example,

P	H	Y	L	O	G	R	A	M	S
P	O	L	Y	H	E	D	R	O	N
Q	U	O	T	A	T	I	O	N	S
P	U	G	N	A	C	I	O	U	S
P	U	?	?	A	?	I	O	?	S

  Consensus sequences can be matched fairly quickly. However, they over-generalise because they don't recognise any dependence between positions and they under-generalise because they don't mention entries that do occur but are less frequent. To limit under-generalisation we have to use approximate matching.

- *Sequence logos.* A sequence logo is a graphical presentation of the probability of each entry at each position. For matching purposes, we can think of these things as independent probabilities, so

```
match logo seq = loop logo seq 1.0
 where loop [] [] p = p
       loop (prob:logo) (x:seq) p = loop logo seq (p*prob x)
```

  Sequence logos don't under-generalise as much as consensus sequences. They still over-generalise by considering the choice of amino acid or nucleotide at one position to be independent of the choice at every other position.

- *Regular expressions.* Regular expressions are a very practical pattern-matching facility. They are nice to work with because they form a well behaved algebra, you can even solve a system of equations where the left hand sides are variables and the right hand sides are regular expressions.

  One of the nice things about regular expressions is that their power is great enough to be useful while at the same time being small enough that they can be implemented efficiently.

  It is particularly useful that they can handle gaps.

  I'll use a subset of UNIX syntax for regular expressions. The full definition (or rather definitions, there are several variants) can be found in

  – "`man 7 regex`" on Linux,

  – "`man 7 re_format`" on MacOS X, or

  – "`man -s 5 regex`" on Solaris.

  PERL extends regular expression syntax further, so far that the pleasant speed properties of regular expressions are completely lost. We are only concerned with the basics here.

- The empty expression is a regular expression, which always matches an empty sequence.
- If e is a regular expression, (e) is a regular expression that means the same thing. Parentheses can be used to resolve operator precedence.
- If c is a character, other than a special character, then c is a regular expression that matches c.
- If c is a special character, then $\backslash c$ is a regular expression that matches c.
- If $c_1 \ldots c_n$ are characters, then $[c_1 \ldots c_n]$ is a regular expression that matches any one of those characters.
- `.` (dot) is a special character that matches any character.
- If $c_1 \ldots c_n$ are characters, then $[\hat{}c_1 \ldots c_n]$ is a regular expression that matches any character other than those.
- If e_1, e_2, $\ldots$, e_n are regular expressions, then $e_1|e_2|\ldots|e_n$ is a regular expression that matches whatever e_1 matches *or* whatever e_2 matches *or* ... whatever e_n matches.
- If e_1, e_2, $\ldots$, e_n are regular expressions, then $e_1e_2 \ldots e_n$ with no visible operator between the e's is a regular expression matching any match of e_1 *followed by* any match of e_2 *followed by* ... any match of e_n.
- If e is a regular expression, $e*$ is a regular expression matching 0 or more adjacent matches of e.
- If e is a regular expression, $e+$ is a regular expression matching 1 or more adjacent matches of e.
- If e is a regular expression, e? means $(e|)$ (match e or match the empty sequence), 0 or 1 matches of e.

Going back to the consensus sequence example,

```
(PHYLOGRAMS|POLYHEDRON|QUOTATIONS|PUGNACIOUS)
```

is a regular expression, and so is

```
PU..A.IO.S
```

and so is

```
[PQ][HOU][YLOG][OHA][GETC][RDI][ARO][MONU][SN]
```

So a regular expression can do what the raw data can, and what a consensus sequence can, and can get close to what a sequence logo can (but not exactly). It can do more. It can do

```
(PHYLO|POLY)(GRAMS|HEDRON)|(QUOTATION|PUGNACIOU)S
```

thus capturing some medium-distance dependencies between letters.

We can convert a regular expression with m symbols into a non-deterministic finite state automaton with at most $6m$ states. By keeping a set of possible current states as a bit string, we can match a non-deterministic FSA against a string of n characters in $O(mn)$ time, which is pretty much like naïve string matching.

We can convert a non-deterministic FSA with m states into a deterministic FSA, which in the worst case might contain $O(2^m)$ states. So this preprocessing step could be exponential in the size of the pattern. But once that is done, a sequence of length n can be matched against a deterministic FSA in time $O(n)$. The worst case preprocessing time is bad, but people routinely use regular expressions without worrying about it.

- *Hidden Markov Models.* Regular expressions can handle more structure than the others, but they aren't "fuzzy"; they don't give us probabilities.

  Basically, Hidden Markov Models are finite state automata with transition probabilities and emission probabilities. They are about the simplest generalisation of regular expressions that give us probabilities, and about the most complex ones that we can afford to use.

3 Three steps to Hidden Markov Models

The climax of this section is a formal definition of the structure of Hidden Markov Models. Before I get there, I want to present Finite State Automata and Markov Chains in a similar style, so that you can more easily see what the similarities and differences are.

3.1 Finite State Automata

A Deterministic Finite State Automaton is a quintuple (S, V, s_0, f, T) where

- $S = \{s_1, \ldots, s_N\}$ is the set of **states**. We may as well take the states to be the integers 1–N, and this is done below.

  The only thing a finite state automaton can remember is what state it is in now. It keeps no history.

- $V = \{v_1, \ldots, v_M\}$ is the **vocabulary**, the set of symbols that may be recognised. We may as well take the symbols to be the integers 1–M and this is done below.

- $s_0 \in S$ is the **initial state**.

- $f \subseteq S$ is the set of **final states**.

- $T : S \times V \to S$ is the **transition function**. At each step, if the current state is q and the next symbol is x the new state will be $T(q, x)$. if we represent states and symbols by integers, the transition function can be represented by a **transition matrix** $T = (t_{ij})_{i \in S, j \in V}$.

It is fairly easy to take a regular expression and generate a *non*-deterministic finite state automaton from it. There are known algorithms for turning non-deterministic FSAs into deterministic ones and minimising them. Books on "automata and language theory" and books about compiler construction usually describe these algorithms. There are programs to do this for several languages:

lex the traditional lexical analyser generator for UNIX. Emits tables and code in C. Originally supported Ratfor as well. Proprietary.

flex a free fast extension of lex. Supports C and C++.

jflex a rewrite of flex in Java for Java. Free.

aflex a rewrite of flex in Ada for Ada. Free.

ml-lex a lex look-alike for the Standard ML language. Free.

alex a lex look-alike for Haskell 98. Free.

leex a lex look-alike for Erlang. Free.

Note that regular expression libraries often compile to fairly straightforward backtracking code, because while compiling regular expressions to deterministic FSAs is well understood and produces very fast matches, the compilation process itself is slow. Libraries trade faster compilation for possibly slower execution.

The basic execution method for a regular expression matcher goes like this:

$s := s_0$;
do {
    $x :=$ next symbol();
    $s := T[s, x]$;
} **while** ($x \notin f$);

3.2 Markov Chains

Markov chains are the simplest probabilistic device for generating sequences. A Markov chain is a triple (S, π, A) where

- $S = \{s_1, \ldots, s_N\}$ is the set of **states**. N is the number of states. We may as well take the states to be the integers 1–N and this is done below.

  The only thing a Markov chain can remember is what state it is in now. It keeps no history.

  The states of a Markov chain can be observed directly; the sequence of states is the output.

- $\pi : S \to [0,1] = \{\pi_1, \ldots, \pi_N\}$ is the **initial probability distribution** on the states. It gives the probability of starting in each state. We expect that $\sum_{s \in S} \pi(s) = \sum_{i=1}^{N} \pi_i = 1$. You should think of π as a column vector.
- $A = (a_{ij})_{i \in S, j \in S}$ is the **transition probability matrix**. If the "machine" is in state j, it may be in state i on the next clock tick with probability a_{ij}. We expect that $a_{ij} \in [0,1]$ for each i and j, and that $\sum_{i \in S} a_{ij} = 1$ for each j.

Running a Markov chain goes like this:

```
s := select a state from distribution π;
for (;;) {
        output s;
        s := select a state from distribution A[−, s];
}
```

We can determine the probability distribution after each step inductively:

$$\begin{aligned} \pi^{(0)} &= \pi \\ \pi^{(n+1)} &= A.\pi^{(n)} = A^n.\pi \end{aligned}$$

For example, let us consider a Markov chain with two states s_1=Fine, s_2=Rainy. We'll start with the rule of indifference: $\pi = \begin{pmatrix} 1/2 \\ 1/2 \end{pmatrix}$. Our transition matrix will respect the rule of thumb that tomorrow will probably have similar weather to today: $A = \begin{pmatrix} 4/5 & 2/5 \\ 1/5 & 3/5 \end{pmatrix}$. For example, this says that if today is rainy, the probability that tomorrow will be fine is 2/5.

Day	Pr(fine)	Pr(rain)
0	0.5	0.5
1	0.6	0.4
2	0.64	0.36
3	0.656	0.344
4	0.6624	0.3376
⋮	⋮	⋮
∞	2/3	1/3

Note that these are predictions. When it comes to day 3, we might *know* that it is raining. It wouldn't just be silly to say "the probability of rain is 0.344" if we *know* it is raining, it would be incoherent. We would restart our predictions with $\pi' = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

"Travesty" is a method for generating scrambled text. It's an application of Markov chains. Here's an example:

```
f% travc 3 160 <hamlet.txt
o her of say cons let almy rapitate. exeunterdo
shout of one fit, fromine in thirtuest? would i
```

```
have tal from him fear is make as delay nay,
thoughtly so will c
```

How does this work? A kth-order travesty generator keeps a "left context" of k symbols. Here $k = 3$, one context is "fro". At each step, we find all the places in the text that have the same left context, pick one of them at random, emit the character we find there, and shift the context one place to the left. For example, the text contains "(fro)m", so we emit "m" and shift the context to "rom". The text contains "p(rom)ise", so we emit "i" and shift the context to "omi". The text contains "n(omi)nation", so we emit "n" and shift the context to "min". The text contains "(min)e", so we emit "e" and shift the context to "ine". And so we end up with "fromine".

How is this a Markov chain? The states are $(k + 1)$-tuples of characters, only those substrings that actually occur in our training text. By looking at the output we can *see* what each state was. There is a transition from state s to state t if and only if the last k symbols of s are the same as the first k symbols of t, and the probability is proportional to the number of times t occurs in the training text.

A Travesty generator can never generate any (local) combination it has not seen; it cannot generalise. There are no percent signs in `hamlet.txt`, for example, so `travc` will never generate one. There are no occurrences of the word "computer" in `hamlet.txt`, so `travw` will never generate it.

3.3 Hidden Markov Models

A Hidden Markov Model H is a quintuple (S, V, π, A, B) where

- $S = \{s_1, \ldots, s_N\}$ is the set of **states**. N is the number of states. We may as well take the states to be the integers 1–N and this is done below.

  Note that the (S, π, A) components form a Markov chain; since a Markov chain keeps no history, neither does a Hidden Markov Model, so the only thing a Hidden Markov Model can remember is what state it is in now.

  The states of a Hidden Markov Model are *hidden*; we never observe them directly.

- $V = \{v_1, \ldots, v_M\}$ is the **vocabulary**, the set of symbols that may be emitted. We may as well take the symbols to be the integers 1–M and this is done below.

- $\pi : S \to [0, 1] = \{\pi_1, \ldots, \pi_N\}$ is the **initial probability distribution** on the states. It gives the probability of starting in each state. We expect that $\sum_{s \in S} \pi(s) = \sum_{i=1}^{N} \pi_i = 1$. You should think of π as a column vector.

- $A = (a_{ij})_{i \in S, j \in S}$ is the **transition probability matrix**. If the "machine" is in state j, it may be in state i on the next clock tick with probability a_{ij}. We expect that $a_{ij} \in [0, 1]$ for each i and j, and that $\sum_{i \in S} a_{ij} = 1$ for each j.

- $B = (b_{ij})_{i \in V, j \in S}$ is the **emission probability matrix**. if the "machine" is in state j, it may emit symbol i on with probability b_{ij}.

You should think of emitting a symbol and starting to jump to a new state as happening at the same time.

Time is discrete and starts with 1.

If we just had (S, π, A) we would have a structure called a Markov chain. The probability that a Markov chain is in state i at time t is $(A^t.\pi)_i$. (Raise the transition matrix to the power t, multiply it on the right by the column vector π, take the i component of the result.)

We can simulate the behaviour of a Hidden Markov model thus:

$t := 1$
$i :=$ a random element of S according to π
for (;;) {
    $j := i$
    $i :=$ a random element of S according to $A[, j]$
    emit a random element of V according to $b[, j]$
    $t := t + 1$
}

The (S, π, A) parts of a Hidden Markov Model make a Markov chain, so we can predict state and symbol probability distributions after n steps:

$$\begin{aligned} \pi^{(n)} &= A^n.\pi \\ \psi^{(n)} &= B.A^n.\pi \end{aligned}$$

But remember the states of the underlying Markov chain are *hidden*; all we can observe is the *symbols* that are emitted.

Hidden Markov Models are widely used in computational linguistics for part-of-speech tagging. If you read the sentence "Thursday Next works for Jurisfiction and saw her first grammasite in *Great Expectations*", unless you are a Jasper Fforde fan you won't have seen "Jurisfiction" or "grammasite" before, but from what you know about English you can figure out that "Jurisfiction" is either the name of a person or the name of an organisation. In either case "Jurisfiction" is a Proper Noun. You can also work out that "grammasite" is a Common Noun. A part of speech tagger can figure this out too, which is essential if we are to deal with real text, which practically always (Heaps' law) contains unfamiliar words.

When we apply HMMs to biological sequences, we are in effect treating biological sequences as being like language. Oddly enough, this seems to make sense. Several people have pointed out that biological sequences have language-like statistics.

4 The three problems

There are three problems we want to solve for Hidden Markov Models. Hidden Markov Models are of practical interest mainly because all these problems can

be solved in practical time for long sequences.

4.1 Problem 1 — Classifying

Given a Hidden Markov Model H, what is the probability $\Pr(\langle X_1 X_2 \ldots X_T \rangle | H)$ of observing some sequence of symbols?

If we can solve that problem, then if we have several models $H_1, \ldots, H_K$, we can find the probability for each, and classify the observed sequence as having most likely been generated by the one with the highest probability. That is, we can use the Maximal Likelihood idea. In symbols,

$$class(\overrightarrow{X}) = \arg\max_k \Pr(\overrightarrow{X}|H_k)$$

4.2 Problem 2 — Decoding

Given a Hidden Markov Model H, what is the best sequence of states $\hat{s}(1) \ldots \hat{s}(T)$ that would explain the observed sequence of symbols $\langle X_1 X_2 \ldots X_T \rangle$?

If the states correspond to things like "is in an active site", "is in a transmembrane domain", "is in an alpha helix", "is in a sheet", then this might let us "parse" a protein and discover something about its structure. In fact "Profile Hidden Markov Models" are a special case of HMMs and this is one of their uses.

4.3 Problem 3 — Training

How do we learn the parameters of an HMM from observations?

5 Problem 1 — Classifying

Key idea: to process a sequence, consider a forward or backward loop that processes it one element at a time.

Here we break the sequence $X(1 \ldots T)$ into two parts, a "past" sequence $X(1 \ldots t)$ and a "future" sequence $X(t+1 \ldots T)$. In the Hidden Markov Model, each symbol emission and each state transition depend only on the current state; there is no memory of what happened before, no lingering effects of the past. This means that we can work on each half separately.

The motive for splitting the sequence into two parts is a hope that we might be able to use induction on t. This turns out to work. The inductive calculation where t advances from 1 towards T is called the *forward* calculation, while the calculation where t is decremented down from T towards 1 is called the *backward* calculation.

5.1 The Forward Calculation

Key idea: sometimes a straightforward proof by induction doesn't work out, and proving a *stronger* claim turns out to be easier, because you have more to

work with at each step. In the same way, when designing a loop, introducing some more variables and establishing a stronger result may be easier. All is well provided we can recover the answer to our original question from an answer to the stronger one. Here it turns out to be useful to ask about the state as well as the sequence.

Define $s(t)$ to be the state the HMM is in at time t.

Define $\alpha(t, i)$ to be the probability of (observing the prefix $X(1 \ldots t)$ **and** being in state i at time t), given our current Hidden Markov Model H. In symbols,

$$\alpha(t, i) = \Pr(X(t \ldots 1) \wedge s(t) = i | H)$$

How can we determine values for α?

5.1.1 Base case

The base case is $t = 1$. The probability that $s(1) = i$ is the thing that we already gave the name π_i to; it is one of the parameters we know if we know what H is. The probability of $X(1)$ given that $s(1) = i$ is also something we know; we gave it the name $b[X(1), i]$. Therefore

$$\alpha(1, i) = b[X(1), i].\pi_i$$

Coding this in C is easy, after shifting the indexing origin from 1 to 0.

```
for (i = 0; i < N; i++)
    alpha[0][i] = b[X[0]][i] * pi[i];
```

This calculation is clearly $O(N)$.

5.1.2 Step case

Suppose we know $\alpha(t, j)$ for $j = 1 \ldots N$. How can we determine $\alpha(t + 1, i)$?

The probability of $X(1 \ldots t + 1)$ **and** $s(t + 1) = i$ can be factored as (the probability of $X(1 \ldots t)$ **and** $s(t + 1) = i$) **and** $X(t + 1)$. With respect to the emission of the last symbol, this is just like the base case. So $\alpha(t + 1, i)$ will be something times $b[X(t + 1), i]$.

The probability of $X(1 \ldots t)$ **and** $s(t + 1) = i$ is obviously related to $\alpha(t, i)$, but that has $s(t) = i$ instead of $s(t + 1) = i$.

What we have to do is sum over all the possible intermediate states j.

$$\alpha(t + 1, i) = b[X(t + 1), i].(\sum_{j=1}^{N} a_{ij}.\alpha(t, j))$$

Coding this in C is easy too. I'm going to express it as determining $\alpha(t, -)$ from $\alpha(t - 1, -)$, but the idea's the same.

```
for (t = 1; t < T; t++) {
    for (i = 0; i < N; i++) {
        s = 0;
        for (j = 0; j < N; j++) s += a[i][j] * alpha[t-1][j];
        alpha[t][i] = b[X[t]][i] * s;
    }
}
```

This calculation is clearly $O(N^2.T)$.

5.1.3 The final probability

In order to determine which of several models ascribes the highest probability to a sequence, we want to determine the probability of that sequence, not the probability of that sequence and some state.

However, the HMM must be in some state at the end, so we just sum over all possible final states.

$$\Pr(X(1\ldots T)|H) = \sum_{i=1}^{N} \alpha(T,i)$$

We can code that in C like this:

```
s = 0;
for (i = 0; i < N; i++) s += alpha[T-1][i];
```

This is clearly $O(N)$, so the whole calculation is $O(N^2.T)$.

5.1.4 Improving the cost

If we have a model, such as a profile HMM, where some transitions are forbidden, we only have to sum over the allowed transitions.

Define $\text{pred}(i) = \{j \in 1\ldots N | a_{ij} \neq 0\}$.

Define $\text{succ}(j) = \{i \in 1\ldots N | a_{ij} \neq 0\}$.

Then $\alpha(t+1,i) = b[X(t+1),i].(\sum_{j\in\text{pred}(i)} a_{ij}.\alpha(t,j))$.

This turns the calculation from $O(N^2.T)$ to $O(E.T)$ where E is the number of non-zero elements in the A matrix. This is somewhere between N and N^2. For profile HMMs, E is $O(N)$, making the whole calculation $O(N.T)$.

5.2 The Backward Calculations

Define $\beta(t,i)$ to be the probability of observing the suffix $X(t+1\ldots T)$ given the HMM parameters H and the fact that $s(t) = i$. In symbols, $\Pr(X(t+1\ldots T)|H \wedge s(t) = i)$. (Note how this dovetails with the definition of α.)

Here the induction works from T down, which is why it's called the "backward" algorithm.

5.2.1 Base Case

When $t = T$, the output sequence we have to explain is empty. The probability of getting the empty sequence when the empty sequence is the only thing you *can* get is clearly 1.

$$\beta(T, i) = 1$$

```
for (i = 0; i < N; i++) beta[T-1][i] = 1;
```

5.2.2 Step Case

As with the forward calculation, we have to multiply an emission probability, a transition probability, and a rest-of-sequence probability.

$$\beta(t-1, j) = \sum_{i=1}^{N} b[X(t), i].a_{ij}.\beta(t, i)$$

```
for (t = T-1; t > 0; t--) {
    for (j = 0; j < N; j++) {
        s = 0;
        for (i = 0; i < N; i++)
            s += b[X[t]][i] * a[j][i] * beta[t][i];
        beta[t-1][j] = s;
    }
}
```

This is clearly $O(N^2.T)$, and can be reduced to $O(E.T)$ by summing over succ(j) instead of all i.

5.3 Scaling

I have shown you simple C code for this problem. What I now have to admit is that it doesn't work. To see that it doesn't work, let's put all the pieces for calculating α together. The following C program uses a simple HMM in which every state is equally likely to be the starting state, all transitions are equally likely, and all emissions are equally likely. The sequence to explain is $00\ldots00$, which is as likely as any other sequence in this model.

```
#define M 20 /* number of symbols */
#define N 50 /* number of states */
#ifdef BIG
#define T_MAX 280
#define T_INC  10
#define real double
#else
#define T_MAX 40
```

```
#define T_INC  1
#define real float
#endif

static int const X[T_MAX];
static real pi[N];
static real a[N][N];
static real b[M][N];

static void init(void) {
    real f;
    int i, j;

    f = 1.0 / N;
    for (i = 0; i < N; i++) pi[i] = f;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) a[i][j] = f;
    f = 1.0 / M;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++) b[i][j] = f;
}

static real prob(int T) {
    real alpha[T_MAX][N];
    real s;
    int i, j, t;

    for (i = 0; i < N; i++)
        alpha[0][i] = b[X[0]][i] * pi[i];
    for (t = 1; t < T; t++) {
        for (i = 0; i < N; i++) {
            s = 0;
            for (j = 0; j < N; j++) s += a[i][j] * alpha[t-1][j];
            alpha[t][i] = b[X[t]][i] * s;
        }
    }
    s = 0;
    for (i = 0; i < N; i++) s += alpha[T-1][i];
    return s;
}

#include <stdio.h>

int main(void) {
    int T;
```

```
    init();
    for (T = T_INC; T < T_MAX; T += T_INC)
        printf("prob[%d] = %g\n", T, prob(T));
    return 0;
}
```

I would encourage you to try this. In the default case, we run out of `float` exponent range at $T = 34$. In the BIG case, we run out of `double` exponent range somewhere about $T = 250$.

The problem is that while *some* sequence must be produced, *each* sequence is extremely unlikely, and that the longer a sequence is, the less likely it is.

However, we can work around this problem. In order to compare probabilities, any monotonic transformation of the probabilities will do. In particular, it is enough to compare the logarithms of the probabilities.

The basic idea is to *scale* the αs to keep them in a reasonable range. One way to do this is to ensure that $\sum_{i=1}^{N} \alpha(t,i) = 1$ for all t.

5.3.1 Base Case

$$\begin{aligned} \alpha'(1,i) &= b[X(1),i].\pi_i \\ c_1 &= \sum_{i=1}^{N} \alpha'(1,i) \\ \hat{\alpha}(1,i) &= \alpha'(1,i)/c_1 \end{aligned}$$

5.3.2 Step Case

$$\begin{aligned} \alpha'(t+1,i) &= b[X(t+1),i].(\sum_{j=1}^{N} a_{ij}.\hat{\alpha}(t,j)) \\ c_{t+1} &= \sum_{i=1}^{N} \alpha'(t+1,i) \\ \hat{\alpha}(t+1,i) &= \alpha'(t+1,i)/c_{t+1} \end{aligned}$$

5.3.3 The final probability

$$\begin{aligned} \Pr(X(1\ldots T)|H) &= (\sum_{i=1}^{N} \hat{\alpha}(T,i)).(\prod_{t=1}^{T} c_t) \\ \log \Pr(X(1\ldots T)|H) &= \log(\sum_{i=1}^{N} \hat{\alpha}(T,i)) + \sum_{t=1}^{T} \log c_t \end{aligned}$$

All of this may be easier to see in C.

```
#define M 20 /* number of symbols */
#define N 50 /* number of states */
#ifdef BIG
#define T_MAX 280
#define T_INC  10
#define real double
#else
#define T_MAX 40
#define T_INC  1
#define real float
#endif

static int const X[T_MAX];
static real pi[N];
static real a[N][N];
static real b[M][N];
static real c[T_MAX];

static void init(void) {
    real f;
    int i, j;

    f = 1.0 / N;
    for (i = 0; i < N; i++) pi[i] = f;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++) a[i][j] = f;
    f = 1.0 / M;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++) b[i][j] = f;
}

#include <math.h>

static real prob(int T) {
    real alpha[T_MAX][N];
    real s;
    int i, j, t;

    s = 0;
    for (i = 0; i < N; i++) {
        alpha[0][i] = b[X[0]][i] * pi[i];
        s += alpha[0][i];
    }
    c[0] = s;
    for (i = 0; i < N; i++) alpha[0][i] /= s;
```

```
    for (t = 1; t < T; t++) {
        for (i = 0; i < N; i++) {
            s = 0;
            for (j = 0; j < N; j++) s += a[i][j] * alpha[t-1][j];
            alpha[t][i] = b[X[t]][i] * s;
        }
        s = 0;
        for (i = 0; i < N; i++) s += alpha[t][i];
        c[t] = s;
        for (i = 0; i < N; i++) alpha[t][i] /= s;
    }

    s = 0;
    for (i = 0; i < N; i++) s += alpha[T-1][i];
    s = log(s);
    for (t = 0; t < T; t++) s += log(c[t]);
    return s;
}

#include <stdio.h>

int main(void) {
    int T;
    real const log10 = log(10);

    init();
    for (T = T_INC; T < T_MAX; T += T_INC)
        printf("log10 prob[%d] = %g\n", T, prob(T)/log10);
    return 0;
}
```

From the output of this compiled with -DBIG, we see that the common logarithm of the probability of a 250-element sequence of zeros is -325.257, so the probability is about 5.53×10^{-326}. While small, that isn't zero.

5.4 Using vector functions

A common structure keeps on popping up in the code we've seen so far. It's the dot product of two vectors. It pays to write a function to compute the dot product, and to call that. We'll also use functions for summing a vector and scaling it. Elementwise product is not the same as dot product. We have a use for that too. We'll also find a use for clearing the elements of a vector. Here they are in C:

```
double dot(double const x[], double const y[], int n) {
    double s;
```

```
    s = 0.0;
    for (int i = 0; i < n; i++) s += x[i]*y[i];
    return s;
}

double sum(double const x[], int n) {
    double s;

    s = 0.0;
    for (int i = 0; i < n; i++) s += x[i];
    return s;
}

void scale(double x[], int n, double a) {
    for (int i = 0; i < n; i++) x[i] /= a;
}

void product(
    double d[], double const x[], double const y[], int n
) {
    for (int i = 0; i < n; i++) d[i] = x[i]*y[i];
}

void clear(double d[], int n) {
    for (int i = 0; i < n; i++) d[i] = 0.0;
}
```

Modern compilers are very good at optimising code like this. There is a standard interface for vector and matrix operations called the Basic Linear Algebra Subprograms, which includes functions like these. There are heavily optimised implementations of the BLAS for all major machines. Fortran 90 even includes DOT() and SUM() in the language.

Let's see what prob() looks like using our new functions.

```
static real prob(int T) {
    real alpha[T_MAX][N];
    real s;

    product(alpha[0], b[X[0]], pi, N);
    c[0] = sum(alpha[0], N);
    scale(alpha[0], N, c[0]);

    for (int t = 1; t < T; t++) {
        for (int i = 0; i < N; i++) {
            alpha[t][i] = b[X[t]][i] * dot(a[i], alpha[t-1], N);
```

```
        }
        c[t] = sum(alpha[t], N);
        scale(alpha[t], N, c[t]);
    }

    s = log(sum(alpha[T-1], N));
    for (int t = 0; t < T; t++) s += log(c[t]);
    return s;
}
```

We have eliminated a lot of indexing, replacing it with function calls that express the meaning of the loops. There is less and simpler code, so it is easier to write and easier to read.

6 Problem 2 — Decoding

Given the parameters H of an HMM and an observed sequence $\langle X_1 X_2 \ldots X_T \rangle$, what sequence of states $\hat{s}(1) \ldots \hat{s}(T)$ would best explain that sequence?

6.1 Sequence of best states

One approach is to say "at each time t, which state i is the most probable?" That is, which state is *locally* the most likely, regardless of what precedes or follows?

Define $\gamma(t,i) = \Pr(s(t) = i | H, \overrightarrow{X})$.

By the definition of conditional probability, this is (the probability of being in state i at time t **and** observing $\overrightarrow{X}$) divided by (the probability of observing $\overrightarrow{X}$).

By the rule for "and" and our chosen decomposition of time into $1 \ldots t$ and $t+1 \ldots T$, the numerator is (the probability of being in state i at time t **and** observing $X(1) \ldots X(t)$) multiplied by (the probability of observing $X(t+1) \ldots X(T)$ given that the state was i at time t).

So

$$\begin{aligned} \gamma(t,i) &= \frac{\alpha(t,i).\beta(t,i)}{\Pr(\overrightarrow{X}|H)} \\ &= \frac{\alpha(t,i).\beta(t,i)}{\sum_{i=1}^{N} \alpha(t,i).\beta(t,i)} \end{aligned}$$

For each T, $\gamma(t,-)$ is a probability distribution. The most likely state is simply the one with the highest probability. So we have

$$\hat{s}(t) = \arg \max_{1 \le i \le N} \gamma(t,i)$$

6.1.1 Scaling

We've already seen that the alphas get very small. So do the betas. Just as we have to scale the $\alpha(t,i)$ by a scale factor depending on t but not i, so we have to scale the $\beta(t,i)$ by a scale factor depending on t but not i.

Since the *same* scale factors are used in the numerator and the denominator of $\gamma(t,i)$ they cancel out, and we can compute

$$\gamma(t,i) \quad = \quad \frac{\hat{\alpha}(t,i).\hat{\beta}(t,i)}{\sum_{i=1}^{N} \hat{\alpha}(t,i).\hat{\beta}(t,i)}$$

6.1.2 C Code

Let's suppose we have $\hat{\alpha}$ in `alpha[]` and $\hat{\beta}$ in `beta[]`. It turns out that we shall have a use for γ in Problem 3, so we'll store the computed values of γ in `gamma[]`.

```
static real alpha[T_MAX][N];
static real beta [T_MAX][N];
static real gamma[T_MAX][N];

static int s_hat[T_MAX];

static void compute_gamma(int T) {
    real s;

    for (int t = 0; t < T; t++) {
        s = 0;
        for (int i = 0; i < N; i++)
            s += gamma[t][i] = alpha[t][i] * beta[t][i];
        for (int i = 0; i < N; i++) gamma[t][i] /= s;
    }
}

static void compute_s_hat(int T) {
    int best_i;
    real best_p;

    for (int t = 0; t < T; t++) {
        best_i = -1, best_p = -1.0;
        for (int i = 0; i < N; i++)
            if (gamma[t][i] > best_p)
                best_i = i, best_p = gamma[t][i];
        s_hat[t] = best_i;
    }
}
```

The cost of computing the γ and the cost of computing the $\hat{s}$ are both clearly $O(N.T)$.

6.1.3 Using vector functions

Just as we introduced a 'dot' function to hide a loop and make it reusable (and give the compiler a helping hand) so we can make a reusable 'arg_max' function.

```
int arg_max(real const x[], int n) {
    int  best_i;   /*  0 .. n-1 */
    real best_x;   /* x[best_i] */

    assert(n >= 1);
    best_i = 0;
    best_x = x[0];
    for (int i = 1; i < n; i++) {
        if (x[i] > best_x) {
            best_i = i;
            best_x = x[i];
        }
    }
    return best_i;
}

int arg_min(real const x[], int n) {
    int  best_i;   /*  0 .. n-1 */
    real best_x;   /* x[best_i] */

    assert(n >= 1);
    best_i = 0;
    best_x = x[0];
    for (int i = 1; i < n; i++) {
        if (x[i] < best_x) {
            best_i = i;
            best_x = x[i];
        }
    }
    return best_i;
}
```

With these in our library, we can now write

```
static void compute_gamma(int T) {
    for (int t = 0; t < T; t++) {
        product(gamma[t], alpha[t], beta[t], N);
        scale(gamma[t], N, sum(gamma[t], N));
    }
```

```
}

static void compute_s_hat(int T) {
    for (int t = 0; t < T; t++) {
        s_hat[t] = arg_max(gamma[t]);
    }
}
```

in which the relationship between the mathematics and the code may be easier to see.

6.1.4 A difficulty

Suppose we have a model where some of the transition probabilities are zero. Then the state sequence we get this way need not be a *possible* sequence: $a[\hat{s}(t)][\hat{s}(t+1)]$ could be 0.

That may not matter. If states correspond to local properties of a molecule, such as being a transmembrane domain, we may be interested in local estimates. We're *expecting* some of these estimates to be wrong. In fact, if $\gamma(t, \hat{s}(t)) < \frac{1}{2}$ the estimated states will be wrong more than half the time. So we may not mind if a transition is impossible; the implied transition could have been wrong because one of the estimated states was wrong anyway.

When we care is when we want to use the fitted state sequence *as* a sequence.

6.2 Best sequence of states

We're now trying to find the state sequence $\hat{s}(1 \ldots T)$ which is *globally* the most likely. That is, we want to maximise $\Pr(\hat{s}|H, \overrightarrow{X})$. Because we are only varying $\hat{s}$, this works out to be equivalent to maximising $\Pr(\hat{s} \wedge \overrightarrow{X}|H)$.

In general, if we are interested in "local" properties of the state sequence, the sequence of best states may do, but if we are interested in "global" properties of the state sequence, this is what we need. In particular, we want to compute the frequency of transitions from globally consistent inferred sequences in the training code.

There is a dynamic programming method for solving this problem. It is called the **Viterbi algorithm**.

The Viterbi algorithm should remind you of the method for computing the αs. The step where we walk backwards over an array of numbers to reconstruct the optimal sequence should remind you of the same step in sequence alignment.

This time we need two numbers for each time and state: how good is this state at this time, and how did we reach this state at this time.

$$\delta(t, i) = \max_{\hat{s}(1..t-1)} \Pr(s(1 \ldots t-1) = \hat{s}(1 \ldots t-1) \wedge s(t) = i \wedge X(1 \ldots t)|H)$$

That is, $\delta(t, i)$ is the highest probability of being in state i at time t, over all state sequences that account for the first t observed symbols.

We can compute $\delta(t, i)$ by induction.

6.2.1 Base case for δ

At $t = 1$ there are no preceding states, so

$$\delta(1, i) = b[X(1), i].\pi_i$$

This is the same as $\alpha(1, i)$.

6.2.2 Step case for δ

$$\delta(t + 1, i) = b[X(t + 1), i].(\max_{j=1}^{N} a_{ij}.\delta(t, j))$$

This is the same as the computation of $\alpha(t + 1, i)$ except for using "max" instead of "sum".

6.2.3 Final probability for δ

The score for the state sequence as a whole is the best score for any final state. That is,

$$\hat{P} = \max_{i=1}^{N} \delta(T, i)$$

6.2.4 ψ, or, how did we get here?

For each time step other than the first, we need to record *which* transition was the most likely one. This is the job of ψ. There is no base case as such because the initial state has no predecessor. So

$$\psi(t - 1, i) = \arg\max_{j=1}^{N} a_{ij}.\delta(t, j)$$

We use this to read off the estimated states. The final state is the one with the highest δ. Preceding states are obtained from ψ.

$$\begin{aligned} \hat{s}(T) &= \arg\max_{i=1}^{N} \delta(T, i) \\ \hat{s}(t - 1) &= \psi(t - 1, \hat{s}(t)) \end{aligned}$$

6.2.5 C code

We don't particularly want $\hat{P}$ and we have no use for δ except to determine $\hat{s}$. So we shall make δ a local variable. We don't need to store both δ and ψ. But we shall, in order to keep the code simple.

```
static void Viterbi(int T) {
    real delta[T_MAX][N];
    int  psi  [T_MAX][N];
```

```
    int  t;
    int  i, j;
    real best_x;
    int  best_j;

    for (i = 0; i < N; i++) {
        delta[0][i] = b[X[0]][i] * pi[i];
    }
    for (t = 1; t < T; t++) {
        for (i = 0; i < N; i++) {
            best_x = 0,  best_j = -1;
            for (j = 0; j < N; j++) {
                real x = a[i][j] * delta[t-1][j];
                if (x > best_x) best_x = x, best_j = j;
            }
            psi[t][i] = best_j;
            delta[t][i] = b[X[t]][i] * best_x;
        }
    }
    best_x = 0, best_j = -1;
    for (j = 0; j < N; j++)
        if (delta[T-1][j] > best_x)
            best_x = delta[T-1][j], best_j = j;
    s_hat[T-1] = best_j;
    for (t = T-1; t > 0; t--)
        s_hat[t-1] = psi[t][s_hat[t]];
}
```

Since this has three nested loops (t: T, i: N, j: N) it is easy to see that the cost is $O(N^2.T)$. Since the innermost j loop only needs to consider values of J for which $a_{ij} \neq 0$, it's clear that the cost is, in general, $O(E.T)$.

Note that we never assign any value to `psi`[0][...]. That's not a mistake. We never use those elements, so why initialise them?

6.2.6 Scaling

If you actually try the code above, be careful to check for δ underflowing to 0. With the same M, N, π, A, and B as used in the previous scaling experiment, this happens at $t = 15$ in single precision and $t = 108$ in double precision.

So yes, we do have to scale the δs. We may as well scale them so that $\sum_{i=1}^{N} \delta(t, i) = 1$. Scaling is allowed because we never compare $\delta(t, i)$ with $\delta(t', i')$ for $t' \neq t$; all that matters is the ratio of δs at the *same* time t.

```
static void Viterbi(int T) {
    real delta[T_MAX][N];
    int  psi  [T_MAX][N];
    int  t;
```

```
    int  i, j;
    real scale;
    real best_x;
    int  best_j;

    for (i = 0; i < N; i++) {
        delta[0][i] = b[X[0]][i] * pi[i];
    }
    for (t = 1; t < T; t++) {
        scale = 0;
        for (i = 0; i < N; i++) {
            best_x = 0,  best_j = -1;
            for (j = 0; j < N; j++) {
                real x = a[i][j] * delta[t-1][j];
                if (x > best_x) best_x = x, best_j = j;
            }
            psi[t][i] = best_j;
            delta[t][i] = b[X[t]][i] * best_x;
            scale += delta[t][i];
        }
        for (i = 0; i < N; i++) delta[t][i] /= scale;
    }
    best_x = 0, best_j = -1;
    for (j = 0; j < N; j++)
        if (delta[T-1][j] > best_x)
            best_x = delta[T-1][j], best_j = j;
    s_hat[T-1] = best_j;
    for (t = T-1; t > 0; t--)
        s_hat[t-1] = psi[t][s_hat[t]];
}
```

There are two things about scaling:

- whenever you multiply lots of small numbers, or lots of big numbers, you are likely to need scaling, *even though* you are using floating point arithmetic.
- once you know you need it, it's not that hard.

6.3 Other optimality criteria

We've looked at two extremes: "get the states right even if the sequence is impossible" and "get the sequence right even if the states are less likely". We could choose other optimality criteria. One criterion that might be interesting would be "get the transitions right", that is, try to maximise the number of correct *adjacent pairs* of states. That might be interesting, but we don't have the time for it.

7 Problem 3 — Learning

Here's a taste of how it works.

A Hidden Markov Model H is a quintuple (S, V, π, A, B). The kind of data we are dealing with tells us what V is. We choose the state set S, usually by picking the number of states.

If we knew π, A, and B, we could decode sequences.

If we had decoded sequences, we could *measure* π, A, and B. The AWK script hmminf.awk does this, given decoded sequences.

> The Paradox of Life
> Philosophical grook.
>
> A bit beyond perception's reach
> I sometimes believe I see
> that Life is two locked boxes, each
> containing the other's key.
>
> — Piet Hein.

If only we had decoded sequences! If only we knew the parameters!

But what if we guessed values for the parameters, perhaps at random, decoded some sequences, and then re-estimated the parameters from that, then re-decoded the sequences, then re-re-estimated the parameters, then re-re-decoded the sequences, then ...

This is precisely what the EM (Expectation/Maximisation) algorithm does. You can prove that the algorithm converges to a result, but since it is a form of hill-climbing, it may well converge to a *local* optimum. We can try repeating the process with different initial random guesses, but we still don't have an optimality guarantee.

However, it would be a bad idea to work simply from decoded sequences. As we work through the decoding process, we compute *probabilities* that the state at each place is such and such. The decoding process described above them picks the best alternative at each place. This throws away information. This is not a good idea. Instead of computing (number of transitions $i \to j$) divided by (number of times in state i) we compute (sum of probabilities of transitions $i \to j$) divided by (sum of probabilities that the state is i).

Instead of reading off transitions from a decoded sequence, then, we determine transition probabilities from the data.

Define $\xi(t, i, j)$ to be the probability that state t is i and state $t + 1$ is j, given the observed sequence $\overrightarrow{X}$ and the current HMM parameters A, B, π.

This is just the probability of

- observing the prefix $X(1 \ldots t)$ and ending in state i and
- a transition from state i to state j and
- the emission of symbol X_{t+1} while in state j and

- observing the suffix $X(t+2 \ldots T)$ given that $s(t+1) = j$

divided by the total probability of all such sequences.

But we already have names for these parts:

- $\alpha(t, i)$
- a_{ij}
- $b_{jX(t+1)}$
- $\beta(t+1, j)$

So

$$\xi(t, i, j) = \frac{\alpha(t, i) a_{ij} b_{jX(t+1)} \beta(t+1, j)}{\sum_{u=1}^{N} \sum_{v=1}^{N} \alpha(t, u) a_{uv} b_{vX(t+1)} \beta(t+1, v)}$$

Note that the numerator just multiplies four constants. At time t we can compute the numerator in $O(N^2)$ time, we can sum the values to get the denominator in $O(N^2)$ time, and we can divide the numerator values by the denominator in $O(N^2)$ time, so the total cost of computing ξ is $O(N^2 T)$.

One point to watch here is that $\xi(t, i, j)$ is only defined for $1 \leq t \leq T-1$. In programming we call this a "fencepost" issue: if you have a fence with T posts, it has $T-1$ panels.

Scaling is of course an issue, but we can handle it by using the scaled $\hat{\alpha}$ and $\hat{\beta}$, $\sum_{u=1}^{N} \sum_{v=1}^{N} \xi(t, i, j) = 1$, so we don't have to worry about scaling the ξ values themselves.

We've already named and computed the probability of $s(t)$ being i given the parameters and observed sequence. That's $\gamma(t, i)$. So to compute

- the sum of probabilities of transitions $i \to j$ divided by
- the sum of probabilities that the state is i

we just compute

$$a'_{ij} = \frac{\sum_{t=1}^{T-1} \xi(t, i, j)}{\sum_{t=1}^{T-1} \gamma(t, i)}$$

as the new estimate for A, and this clearly takes $O(N^2 T)$ time.

What about a new estimate for B?

For clarity, I want to use a convention you may not have met before. You may have thought that treating false as 0 and true as 1 was a peculiarity of C. (Or PL/I, if you'd ever heard of PL/I.) Not so. It's a useful convention in mathematics, called the "Iverson convention". See `Iverson_bracket` in the Wikipedia. The idea is that $[e]$ is 1 if e is true, or 0 if e is false. Indeed, it's a very "strong" zero in that $[e]d$ is defined to be zero when e is false, even if d is not defined. It's useful when we want to consider just some of the numbers in a range. The book "Concrete Mathematics" by Graham, Knuth, and Patashnik is well-nigh essential reading for computer scientists (as contrasted with programmers), and provides motivation and many examples.

To estimate the probability of emitting symbol k, we have to confine our attention to those positions where the observed symbol is k, and that's exactly what the Iverson bracket is for.

We divide the estimated number of times the state was i and the symbol was k by the estimated number of times the state was i.

$$b'_{ik} = \frac{\sum_{t=1}^{T} \gamma(t,i)[X_t = k]}{\sum_{t=1}^{T} \gamma(t,i)}$$

C code for computing B' is

```
for (int i = 0; i < N; i++) {
    clear(B_prime[i], M);
}
for (int t = 0; t < T; t++) {
    for (int i = 0; i < N; i++) {
B_prime[i][X[k]] += gamma[t][i];
    }
}
for (i = 0; i < N; i++) {
    scale(B_prime[i], M, sum(B_prime[i], M));
}
```

What happened here?

We've turned the sum inside out.

$$\sigma_{ij} = \sum_{t=1}^{T} e(t,i)[f(t,i) = j]$$

where the range of f is included in the range of j can be programmed as

```
for (int i = ...) clear(sigma[i], ...);
for (int t = ...) {
    for (int i = ...) sigma[i][f(t,i)] += e(t,i);
}
```

One gain here is that we make a single pass over `gamma[][]`, which your computer's cache will like. Indeed, if this were the only use of `gamma[][]`, we could fuse this calculation with `compute_gamma()` and never store more than a single row of `gamma[][]` at all.

There's one more parameter we have to update, and that's π.

$$\pi'_i = \gamma(1,i)$$

There's a nasty snag here. When it comes to A' and B', every position in the observed sequence gives us some information. But only the beginning tells us anything about pi'. In order to estimate pi' we need information from many sequences, several times more sequences than we have states.

One way to arrange this is

```
for (int s = 0; s <= #sequences; s++) {
    train(A_prime[s], B_prime[s], pi_prime[s],
          A, B, pi, sequence[s]);
}
A' = average(A_prime);
B' = average(B_prime);
pi' = average(pi_prime);
```

Of course we'd want to turn this inside out too. (How?)

In the lecture I discussed the idea of a "learning rate" or "momentum" parameter. If you are trying to keep a running estimate x_t of something, and y_t is the new information, you can use

$$x_{t+1} = \mu x_t + (1 - \mu) y_{t+1}$$

where $\mu = 0$ means ignoring the past and $\mu = 1$ means ignoring the present, and a value somewhere in between lets you learn without jumping to conclusions from noise.

One final point I want to mention is the difference between

- learning to *convergence* and
- learning to *criterion*

In learning to convergence, we repeat the training process over and over again until the estimates of A, B, and π stop changing.

In learning to criterion, we set some quality criterion, such as the proportion of a test collection that must be correctly classified, and repeat the training process until this criterion is reached. If the process converges before criterion is reached, we can try again with a different set of random initial parameters, or we might decide to try having more hidden states. If the process reaches criterion, we might not care if it converged or not. (Look up "over-fitting").

We've now presented what it is we *do* to learn a Hidden Markov Model, but not why it *works*.

These notes are based on the description of the algorithm in

> *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*, by Lawrence R. Rabiner, Proceedings of the IEEE, Volume 77, number 2, February 1989.

which you can find on the Web. That tells us that "there is no known way to analytically solve for the model which maximizes the probability of the observation sequence. In fact, given any finite observation sequence as training data, there is no optimal way of estimating the model parameters. We can, however, choose $\ldots(A, B, \pi)$ such that $[\Pr(\vec{X}|(A, B, \pi))]$ is locally maximized using an iterative procedure such as the Baum-Welch method $\ldots$ or using gradient techniques"

What he's talking about is the *hill-climbing* method described above, that works by starting with some estimate of the parameters, and then repeats an

improvement step. This is a kind of algorithm that we talked about when looking at inferring phylogenies. We saw then that you get a *local* optimum which is *not* guaranteed to be the best result possible. As Rabiner puts it, "It should be pointed out that the forward-backward algorithm leads to local maxima only, and that in most problems of interest, the optimization surface is very complex and has many local maxima".

That paper also tells you where to find some of the mathematical background. A web search for "Hidden Markov Model" or "Baum-Welch algorithm" or "EM algorithm" will tell you more than you ever wanted to know.

8 Profile Hidden Markov Models

We don't have time to cover these in the lectures either, but do look them up on the web. They are a kind of HMMs with a special structure. There are states that represent insertion, states that represent deletion, and states that represent no change to length.