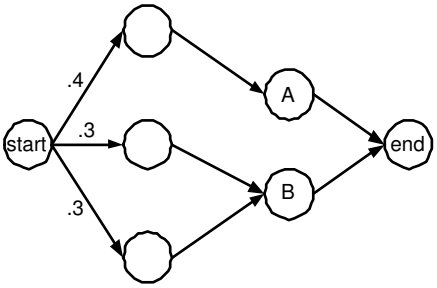
CSE 527 Lecture Notes
Lecture 18, Nov. 26, 2003
Martha Mercaldi - mercaldi@cs

Outline

1. Other analyses of HMMs
2. We've assumed we know the model, but how do you generate one?
   (a) Structure
   (b) Training (learn parameters given a structure)

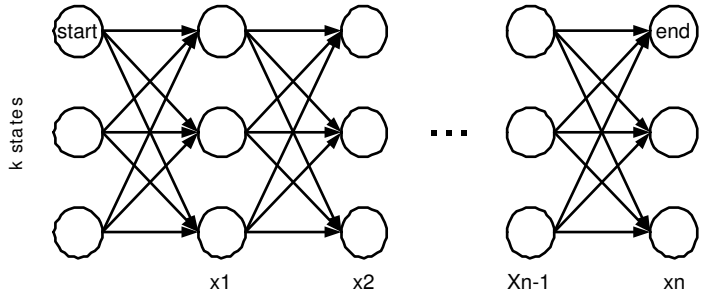
Hidden Markov Models

Last class we looked at the Viterbi Algorithm which selects the most probable path through the lattice ($Max_\pi(P(x,\pi))$), but it is not necessary to have a single path that dominates, as shown in the following example:



The most probable path goes through A, but the most probable state (after 2 characters) is state B. While in theory this problem can be arbitrarily bad, in practice the algorithm tends to work.

Viterbi Review



The probability of the most probable path ending in state l at time $i+1$:
$V_l(i+1) = e_l(x_{i+1}) \cdot max_k(V_k(i) \cdot a_{kl})$ (where $e_l(b)$ is character b's emission probability in state l and a_{kl} is the transition probability from state k to state l)
The Viterbi path simply traces the most likely path from the end node to the start node.

NB: It is typical to use logs for this calculation, because the fraction gets very small, and you may underflow.

However, rather than calculate the most probable path, what if we just consider the probability that you're in a given state? The calculation would be similar except that you look at all preceding states and add them up instead of taking tha maximum (to calculate the total probablity summed over all the paths leading to a state):
$f_k(i) = P(x_1...x_i|\pi_i = k)$
$f_l(i+1) = e_l(x_{i+1}) \cdot \Sigma_k f_k(i) \cdot a_{kl}$
$P(x) = \Sigma_k f_k(n) \cdot a_{k0}$
This is commonly called "the forward algorithm." Because it's a sum of products, we cannot use logarithms as

simply as we did in the other algorithm, so to avoid underflow we can either keep track of a scaling factor as you calculate or use a table to evaluate (and interpolate) a tailored function that is basically the logarithm of a sum of exponentials of logarithms.

What if we wanted to calculate the probability of a path from state k to the end? $b_k(i) = P(x_{i+1}...x_n|\pi_i = k)$ This is calculated by the "backward" algorithm. $b_k(i) = \Sigma_l a_{kl} \cdot e_l(x_{i+1}) \cdot b_l(i+1)$ and $b_k(n) = a_{k0}$, the transition probability to the final state. We calculate the probability of a sequence given state k at time i:

$$P(x, \pi_i = k) = P(x_1...x_i, \pi_i = k) \cdot P(x_{i+1}...x_n|x_1...x_i, \pi_i = k) = P(x_1...x_i, \pi_i = k) \cdot P(x_{i+1}...x_n|\pi_i = k)$$

Note that the last step follows from the Markov property: behavior past step i is independent of previous behavior given the state at step i ($\pi_i = k$). From this, it's easy to calculate the posterior probability of being in state k at time i:

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

Posterior Decoding

Terminology— *posterior:* based on probability after seeing data; *prior:* based on probability before seeing data

If the most likely path accounts for less than half of the probability, posterior decoding may make sense.

$\hat{\pi}_i = argmax_k(P(\pi_i = k|x))$
Notice that because transitions are not taken into account, it is possible that this sequence of states isn't even legal in the model.

What if we want to know which of two (or some number of) subsets of the states was the machine in at some specific time? (i.e. in the loaded and fair dice example, there were multiple "fair" states and multiple "loaded" states)

$g(k)$ is some function on the states
$G(i|x) = \Sigma_k P(\pi_i = k|x) \cdot g(x)$

C_pG Example

Data: 41 human sequences, totaling 60 kbp, with 48 C_PG islands with an average length of 1 kpb each.

Viterbi: Found 46 of the 48 islands, plus 121 false positives. With the addition of a post processing step applied to the data which merged any two islands that were within 500bp of eachother and deleted short islands (<500 bp), Viterbi still found 46/48, but with only 67 false positives.

Posterior Decoding: Found 46 of the 48 islands and 276 false positives. With the postprocessed data, posterior decoding still found 46/48 islands, but only 93 false positives.

There was little difference between the two methods in this example (with Viterbi being slightly better), but we'll see cases presently where it makes a much bigger difference.

Model Training

(For now we defer the question of how to decide on the structure of a model, and ask how, given a structure, to identify appropriate parameters.)

Given a model topology and independant training sequences, we want to find emission probabilities for each state and transition probabilities for each transition.

If the path π through the HMM is known, we can use Maximum Liklihood Estimation (MLE).

$a_{kl} = \frac{count(k \to l)}{count(k \to anywhere)}$
$e_k(b) =$ a similar ratio, the fraction of the time that you're in state k that you emit a b.

If π is hidden use Expectation Maximization (EM) to estimate the state sequences and paramaters in an iterative loop (Given π estimate Θ (the parameters) , then with that Θ estimate π, and repeat until results converge).

If you've got a model with 8 states, for each state you'd need 8 transition probabilities and 4 emission probabilities. These $8 * 12$ parameters are quite a few, so even for this small a state diagram we require a fair amount of training data.

Viterbi Training

Make your initial parameter estimates (either randomly or preseeded if you have strong prior knowledge of what they might look like). Calculate the Viterbi path for each training sequence and from that count the transitions and emissions, thereby creating a new Θ. Use this new Θ and recalculate the Viterbi path for each training sequence. Iterate until Θ stops changing.

Advantages:

- Fast and simple
- Viterbi path is discrete and therefore this algorithm will converge

Disadvantages:

- May converge only on a local optimum, not the global one.
- Not actually maximizing the liklihood we want.

Regarding this last disadvantage, we are looking only at the most probable paths, not at all paths. Also, we are not getting any paramater estimates along paths that do not occur in the training data.

Baum-Welch Training

The general consensus is that this works better than Viterbi Training, but because of its slower speed, people don't always use it.

$P(\pi_i = k, \pi_{i+1} = l|x, \Theta) = \frac{f_k(i) \cdot e_l(x_{i+1}) \cdot b_l(i+1)}{P(x|\Theta)}$

Number of $k \to l$ transitions $= \Sigma_{trainingdata} \frac{1}{P(x_j)} \cdot \Sigma_i P(\pi = k, \pi_{i+1} = l|x, \Theta)$

The number of emissions is a similar formula.

On the training sequences we run the forwards/backwards algorithm and use the above formulas to arrive at some Θ. We then use that Θ to rerun the above algorithm and generate a new set of values for Θ. Repeat this loop until the values for Θ converge. Convergence in this algorithm (as opposed to Viterbi Training) is a little trickier because you are optimizing a continuous sequence. Other disadvantages include the risk of overfitting the data or identifying local maxima. It also has high training data requirements.

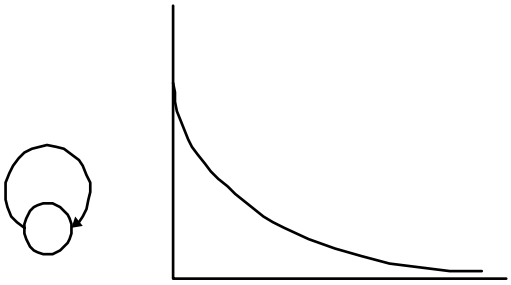
Summary

Viterbi: best single path
Forward: estimate probability by summing over all paths
Backwards: similar to forward, but on the back end of the sequence
Baum-Welch: training based on EM and forward/backward algorithms

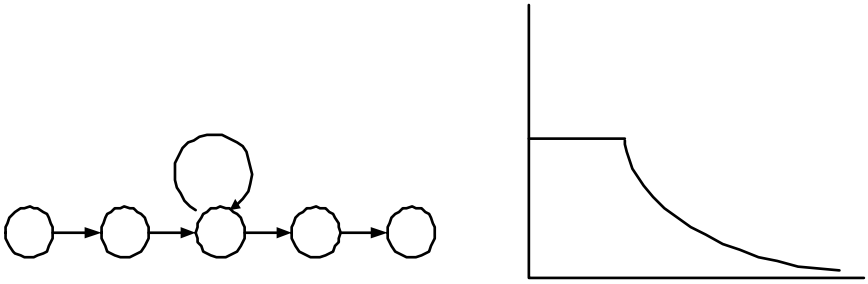
Model Structure

We want to find the correct number of states and topology to link them together. This tricky because there are many degrees of freedom in this problem and models that appear different can actually be equivalent. It is important in practice to constrain the allowable structures as much as possible. In particular, allowing all n^2 transitions between states in an n state model and "letting the data pick the model" doesn't work well in practice.

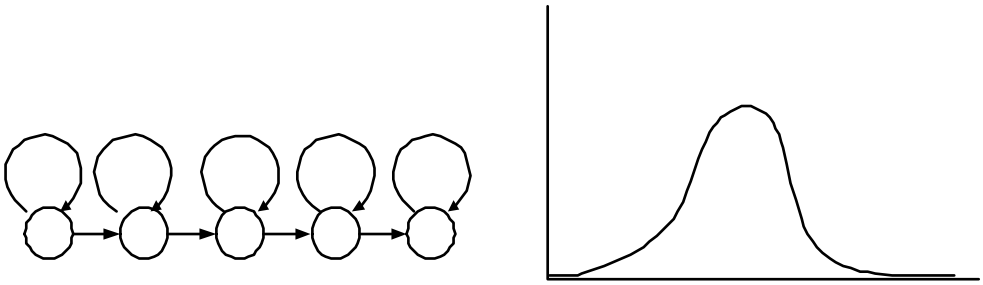
As another example of a model topology issue, consider the length distributions of features predicted by HMM's. For example, the lengths of C_pG islands predicted by our C_pG HMM is basically geometrically distributed $(p^n(1-p)$, where p is the probability of returning to a + state and $1-p$ is the probability of transitioning from a + to a − state), but in reality the C_PG islands don't follow this distribution (which is perhaps why the "postprocessing" step was needed to reduce false positives).



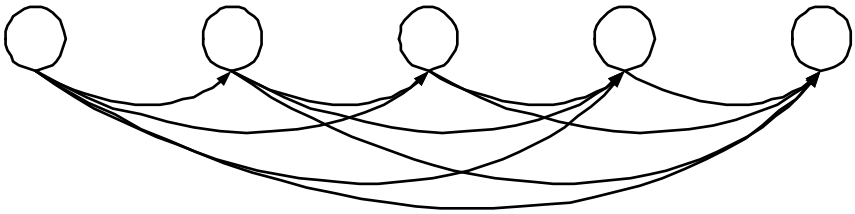
You could play with the model a little and require a minimum number of states thereby changing the distribution:



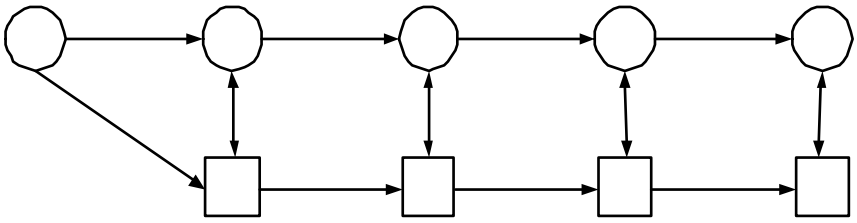
Alternatively, if you chain together several looping states, the "law of large numbers" will kick in and push you towards a bell shape



The downside of all of this is that the number of states quickly grows, making the model slow and requiring many parameters. For example, look at "profile HMMs" (next lecture) for recognizing similar proteins. Because it is possible to delete arbitrary runs of amino acids the model has $O(n^2)$ transitions.



However, we can get around this by implementing "silent" states corresponding to the normal states but, which don't emit any amino acid. We allow the model to switch arbitrarily between emitting and silent states. This gets us back to $O(n)$ transition probablities to learn from training data, at the expense of less flexibility is specifying length distributions of consecutive deletions—a good compromise for modeling protein families, but certainly not universally appropriate.



("silent" states)

If length distributions really differ from the above geometric distribution variants, it is also possible to explicity model residence time in a certain state, although this complicates the training and inference algorithms. We'll see an example of this in a week or so when we talk about gene finding.