6.047/6.878 Lecture 7: HMMs II, September 25, 2008

October 1, 2008

The previous lecture introduced hidden Markov models (HMMs), a technique used to infer "hidden" information such as whether a particular nucleotide is part of the coding sequence of a gene, from observable information, such as the sequence of nucleotides. Recall that a Markov chain consists of states Q, initial state probabilities p, and state transition probabilities A. The key assumption that makes the chain a Markov chain is that the probability of going to a particular state depends only on the previous state, not on all the ones before that. A hidden Markov model has the additional property of emitting a series of observable outputs, one from each state, with various emission probabilities E. Because the observations do not allow one to uniquely infer the states, the model is "hidden."

The principle we used to determine hidden states in an HMM was the same as the principle used for sequence alignment. In alignment, we had an exponential number of possible sequences; in the HMM matching problem, we have an exponential number of possible parse sequences, i.e., choices of generating states. Indeed, in an HMM with k states, at each position we can be in any of k states; hence, for a sequence of length n, there are k^n possible parses. As we have seen, in both cases we nonetheless avoid actually doing exponential work by using dynamic programming.

HMMs present several problems of interest beyond simply finding the optimal parse sequence, however. So far, we have discussed the **Viterbi decoding** algorithm for finding the single optimal path that could have generated a given sequence, and scoring (i.e., computing the probability of) such a path. We also discussed the **Forward** algorithm for computing the **total probability** of a given sequence being generated by a particular HMM over all possible state paths that could have generated it; the method is yet another application of dynamic programming. One motivation for computing this probability is the desire to measure the accuracy of a model. Being able to compute the total probability of a sequence allows us to compare alternate models by asking the question: "Given a portion of a genome, how likely is it that each HMM produced this sequence?"

Although we now know the Viterbi decoding algorithm for finding the single optimal path, we will talk about another notion of decoding known as **posterior decoding**, which finds the most likely state at any position of a sequence (given the knowledge that our HMM produced the entire sequence). The posterior decoding algorithm will apply both the forward algorithm and the closely related **backward algorithm**. After this discussion, we will pause for an aside on **encoding "memory"** in a Markov chain before moving on to cover **learning**

algorithms: that is, given a sequence and an HMM structure, how to choose parameters for that HMM which best describe the sequence.

1 The backward algorithm

In analogy with the forward algorithm discussed last time, we may define a backward algorithm with only slight changes. This time, given a starting position in the sequence and the current state of the HMM—i.e., a certain square in the dynamic programming table—we compute the total probability of reaching the end of the sequence, taken over all paths leaving the chosen square.

In other words, we calculate:

$$b_k(i) = P(x_{i+1}...x_N|\pi_i = k)$$

This can be done iteratively using a recurrence similar to the forward recurrence; we do not elaborate on the details, but they are provided in the lecture slides. In this case, after the algorithm finishes, we can obtain the total probability of the HMM generating the given sequence by summing over the states in the first column.

One might wonder why the backward algorithm is useful given that we already know how to compute total probability using the forward algorithm. One simple reason is that the backward algorithm provides a check against numerical errors, but more interestingly, combining our knowledge from the forward and backward algorithms allows us to find the probability of the HMM being in a given state at any position of the sequence. This is called posterior decoding, which is the topic of the next section.

2 Posterior decoding

In contrast to the Viterbi optimal sequence of states, which finds the most likely path of states that could have generated the input sequence X, posterior decoding finds a sequence π of states such that at each individual position i, the decoded state π_i is the most likely (given the HMM and the assumption it generated X). Note that posterior decoding is thus not a "path decoding" algorithm: the "path" it produces will be less likely than the Viterbi optimal path; indeed, it may even be nonsensical, for example if two consecutive states have a zero transition probability.

Formally, the difference between Viterbi and posterior decoding is that in position i, the Viterbi algorithm calculates

$$\pi_i^* = i\text{-th position of } \operatorname{argmax}_\pi P(X, \pi)$$

whereas posterior decoding calculates

$$\pi\hat{}_i = \operatorname{argmax}_k P(\pi_i = k|X),$$

where

$$P(\pi_i = k|X) = \sum_{\pi:\{\pi_i=k\}} P(X,\pi) \cdot \frac{1}{P(X)}.$$

The term $\sum_{\pi:\{\pi_i=k\}} P(X,\pi)$ expresses the sum of the probabilities of all paths producing X that pass through state k in position i.

Since $P(X)$ is independent of k, dividing by it does not affect the argmax, so we can write:

$$\pi\hat{}_i = \text{argmax}_k \sum_{\pi:\{\pi_i=k\}} P(X,\pi)$$

Recall that $P(X)$ was calculated by the forward algorithm using the recursion for

$$f_k(i) = P(x_1...x_i, \pi_i = k)$$

given on the right-middle slide of page 3. A simple way to calculate $\sum_{\pi:\{\pi_i=k\}} P(X,\pi)$ would be to use this same recursion, except at the $i+1$ step instead of summing over all states just use state k. The problem with this is that if we wanted to calculate $\pi\hat{}$ we would need to do this calculation for all i and all k.

The key observation is that

$$\sum_{\pi:\{\pi_i=k\}} P(X,\pi) = P(\pi_i = k, X) = P(x_1...x_i, \pi_i = k)P(x_{i+1}...x_N|\pi_i = k) = f_k(i) \cdot b_k(i)$$

so we can obtain all of these sums at once by running the forward and backward algorithms and multiplying the values in the corresponding cells! Thus, computing the optimal $\pi\hat{}_i$ amounts only to testing each of the K possible states and choosing the one with maximum probability.

Which of Viterbi or posterior decoding is better for real applications? The answer depends partly on how likely the Viterbi optimal path is (i.e., on how much of the probability mass is concentrated on it compared to other paths): if the optimal path is only marginally better than alternatives, posterior decoding is probably more useful. Additionally, the application itself plays a crucial role in determining the algorithm to apply. For example, if a biologist had identified a possible promoter region but wanted to be more sure before running experiments, he or she might wish to apply posterior decoding to estimate the likelihood (taking into account all possible paths) that the region of interest was indeed a promoter region. There are other examples, however, in which Viterbi decoding is more appropriate. Some applications even combine the two algorithms.

3 Encoding memory in an HMM

Before moving on to discussing learning, we pause to describe an application of HMMs requiring a new trick: incorporating "memory" into a Markov model by increasing the number of states. First we present a motivating biological example.

Recall that in the previous lecture, we built an HMM with two states: a “high-CG” promoter state with elevated probabilities of emitting Cs and Gs, and a background state with uniform probabilities of emitting all nucleotides. Thus, our previous model, which we will call HMM1, characterized promoters simply by abundance of Cs and Gs. However, in reality the situation is more complicated. As we shall see, a more accurate model, which we call HMM2, would instead monitor the abundance of CpG pairs, i.e., C and G nucleotides on the same strand (separated by a phosphate on the DNA backbone, hence the name “CpG”). Note that we call these CpGs simply to avoid confusion with bonded complementary C–G bases (on opposing strands).

Some biological background will shed light on the reason for introducing HMM2. In the genome, not only the sequence of bases, but also their *methylation states* determine the biological processes that act on DNA. That is, a methyl group can be added to a nucleotide, thus marking it for future reference. Proteins that later bind to the DNA (e.g., for transcription or replication purposes) “notice” such modifications that have been made to particular bases. To elaborate on one interesting example, methylation allows error-checking in the DNA replication process. Recall that replication is performed by “unzipping” the DNA polymer and then rebinding complementary bases to each strand, resulting in a pair of new double-stranded DNA helices. If, during this process, a discrepancy is found between an old strand of DNA and a new strand (i.e., a pair of bases is not complementary), then the old—correct—base is identified by its being methylated. Thus, the new, unmethylated base can be removed and replaced with a base actually complementary to the original one.

Now, a side effect of methylation is that in a CpG pair, a methylated C has a high chance of mutating to a T. Hence, dinucleotide CpGs are rare throughout the genome. However, in active promoter regions, methylation is suppressed, resulting in a greater abundance of CpG dinucleotides. Such regions are called CpG islands. These few hundred to few thousand base-long islands thus allow us to identify promoters with greater accuracy than classifying by abundance of Cs and Gs alone.

All of this raises the question of how to encode dinucleotides in a Markov model, which by definition is “memoryless”: the next state depends only on the current state via predefined transition probabilities. Fortunately, we can overcome this barrier simply by increasing the number of states in our Markov model (thus increasing the information stored in a single state). That is, we encode in a state all the information we need to remember in order to know how to move to the next state. In general, if we have an “almost HMM” that determines transition probabilities based not only on the single previous state but also on a finite history (possibly of both states and emissions), we can convert this to a true memoryless HMM by increasing the number of states.

In our particular example, we wish to create HMM2 such that dinucleotide CpGs have high frequency in promoter states and low frequency in the background. To do this, we use eight states rather than just two, with each state encoding both the promoter/background state as well as the current emission. We thus need to specify an 8×8 table of transition probabilities. (Note, however, that each emission probability from a given state is now either 0 or 1, since the state already contains the information of which nucleotide to emit.) Details of this construction are included in the lecture slides. (But note that on the middle-right slide on page 4 it incorrectly states that the emissions probabilities are distinct for the + and - states. It is actually the *transition* probabilities that are different – the emission probabilities are just 0 and 1 in both cases.)

Although an 8 state model would normally have 64 transition probabilities the slides show only 32; transition probabilities between + and - states are not included. Since transitions to and from CpG islands are relatively rare it would require a very large amount of training data to infer what all of these probabilities are. A simplification is to assume that all transition probabilities from a + state to a - state are the same, regardless of the nucleotides between which we are transitioning, and to similarly assume a single probability for transitions from a - to a + state. In that case we would need these two transition parameters in addition to the 16 shown on the slides. If our assumptions turn out to be biologically incorrect then more parameters would be needed to have an accurate model.

During lecture someone pointed out that the transition probabilities given in the table at the middle-left slide of page 4 are different depending on which strand you are moving along. For example, when one strand transitions from T to C, the other transitions from G to A, yet the numbers in the table are quite different (.355 and .161). This seemed to imply that a different HMM is needed to predict CpG islands depending on which strand you are moving along. However, this reasoning was not correct. For a single HMM to work on either strand it must produce the same frequency of TpCs on one strand as GpAs on the other. However, the frequency of TpC's is not simply the probability of transitioning from T to C. Rather this transition probability must be multiplied by the frequency of T. We calculate based on the transition probabilities in the '+' table that this HMM would produce As and Ts with frequency about 15% and Cs and Gs with frequency about 35%. This calculation is in Appendix 1. When this is taken into account the probabilities on the two strands are within around 7% of each other. For example, the frequency of TpCs is 0.15*0.355 = 0.0525, while the frequency of GpAs is 0.35*0.161 = 0.05635. We will chalk this difference up to sampling error.

4 Learning

The final topic of this lecture is learning, i.e., having a machine figure out how to model data. The word 'learning' must be taken with a grain of salt. We do not give a machine raw data and nothing else and expect it to figure out how to model this data from scratch. Rather, we must use our intuition and knowledge of biology to come up with the general framework of the model. Then we ask the machine to choose parameters for our model.

More specifically, given a training sequence and an HMM with known layout but unspecified parameters, how does one infer the best choice of parameters? This general class of problems splits naturally into two subclasses based on a whether or not we know the hidden states in our training data.

For example, if we have a hidden Markov model to distinguish coding regions of genes from other stretches of DNA, we might train our model with a sequence in which coding genes have already been identified using other means. We would supply this information for supervised learning. On the other hand, we might be working with a DNA sequence for an organism that has not had any genes classified and that we expect would have different parameter values from any that have already been classified. In that case, we could use unsupervised learning to infer the parameters without our having to specify the genes in our training data.

4.1 Supervised learning

In this case we assume we are given the layout of the HMM, an emitted sequence X, and additionally the *actual sequence* of Markov states π that produced X. In other words, our training data is *labeled*). We wish to find the transition and emission probabilities a and e maximizing the probability $P(X, \pi)$ of generating the sequence X and labels π.

This version of learning is easy: since we already know the path π of states the Markov chain took, our intuition suggests that we should choose parameters proportional to the observed frequencies of the corresponding events occurring. With i denoting position, k and l denoting states, and x denoting a base, we optimally set

$$e(k, x) = \frac{\text{\# of times } \pi_i = k \text{ and } x \text{ emitted}}{\text{\# of times } \pi_i = k}$$

and

$$a(k, l) = \frac{\text{\# of times } \pi_{i-1} = k \text{ and } \pi_i = l}{\text{\# of times } \pi_{i-1} = k}.$$

An example is given in the slide at the top-right of page 7. The counts are slightly different depending on whether you consider the start and end to be separate states. If we ignore the start and end then, for example, $a(B, P) = 0.25$ since of the 4 B states (we are excluding the final one) there is exactly 1 that transitions to a P state. If we considered the end to be a separate state then $a(B, P)$ would be 0.2, since we consider the final B state as transitioning to the end state. Similarly, we calculate $e(B, C) = 0.4$ because 2 of the 5 B states emit a C.

We can use Lagrange multipliers to verify our intuition that the transition and emission probabilities a and e obtained by counting maximize the probability $P(X, \pi)$ of generating the sequence X and labels π. This calculation is done appendix 2.

In addition the initial state probabilities of the HMM need to be specified. We choose these as the total frequency of each state in the training data.

A problem with the method of setting probabilities using our training data is that counts of some transitions and emissions might be 0 even though the actual probability is small but non-zero. If we do not correct this our HMM will consider these transitions and emissions to be impossible, which could significantly distort the results. To fix this we add 'pseudocounts' which are small non-zero counts. For example, we could set these counts to be 1, or we could make an estimate based on the a-priori probabilities.

4.2 Unsupervised learning

Now we assume we are given the HMM layout and sequence X and wish to find parameters a, e maximizing $P(X) = \sum_{\pi} P(X, \pi)$. Note that X is now *unlabeled*: that is, we have no knowledge of π to begin with.

The idea is we start with some initial guess for the parameter values, use those parameter values to obtain values (or probabilities) of the hidden states, use those to calculate new parameter values, and iterate until convergence.

Our initial guess for the parameters might be guided by other knowledge. For example, when working with a DNA sequence we might start with parameter values from a related species for which we have an annotated genome (for which parameters could be obtained using supervised learning). Or, we might just use arbitrary initial parameters. (The slide on Viterbi Training on page 8 describes initialization as 'Same'. This is a reference to initialization in the Baum-Welch algorithm slide on page 10.)

The lecture presented two methods of implementing unsupervised learning, Viterbi Training and the Baum-Welch Algorithm. Although Baum-Welch is theoretically more sound Viterbi training is simpler so we discuss it first.

Rather than maximizing $P(X) = \sum_{\pi} P(X, \pi)$, at each step Viterbi training uses π^*, the most-likely path. Since there are other paths and π^* usually only captures some of the probability this is not the same as maximizing $P(X)$, which would require looking at all paths.

In Viterbi training we use the Viterbi algorithm to find π^* given the current parameter values. We then use this path to estimate the coefficients a, e in the same way we used the known path in supervised learning – by counting emissions and transitions from each state. These counts, adjusted by pseudocounts, are then used to calculate new parameter values, and we repeat until convergence.

Side note: Posterior decoding can be used instead of the Viterbi algorithm, provided you use non-zero parameters so all transitions are legal. This still uses only one path for counting, but it is $\pi\hat{}$ instead of π^*.

In supervised learning and in Viterbi training we count frequencies in a single path to estimate our parameters a and e. But for particular parameter values the model determines probabilities for every state at each position, not just a single state. How can we take all of these probabilities into account when estimating our parameters, rather than just a single value at each position? That is what the Baum-Welch algorithm does.

The basic framework of the Baum-Welch algorithm is the same as for Viterbi training. In both cases we start with an initial guess for parameter values, use those to gather information about the hidden states, and then use that information to calculate new parameter values. The difference is that rather than determining the counts of transitions and emissions, $A_{kl}, E_k(b)$ by counting actual transitions and emissions in a particular path we do so by adding up probabilities over all possible paths.

For A_{kl}, instead of counting the number of places where state k is immediately followed by state l, we add up, for all positions i, the probability that position i is in state k and position $i+1$ is in state l . To do this we must add up the probabilities of all paths that pass through states k and l at positions i and $i+1$, respectively.

This is very similar to the sum we calculated in posterior decoding, except we are specifying the states at two consecutive positions instead of one. As in posterior decoding it uses f and b calculated by the forward and backward algorithms, but instead of just taking $f_k(i) \cdot b_k(i)$ it must take $f_k(i) \cdot b_k(i+1)$ times an intermediate term that has to do with the transition from position i to position $i+1$. This calculation is shown on the lower left slide of page 9. These must be summed over all positions i to calculate A_{kl}. A similar method is used to calculate $E_k(b)$, again using f and b.

A result of Baum-Welch *guarantees* convergence. A formal proof can be obtained by using the principle of expectation maximization.

Although this method is guaranteed to converge, it could converge to a *local* rather than *global* maximum. To work around this it is recommended that the whole procedure be repeated many times with different initial guesses for the unknown parameters, and use the one that converges to the best result.

5 Appendix 1

In the discussion of CpG islands above, there was a need to calculate the frequencies of states given the transition probabilities. We can do that as follows. Suppose the probability of transitioning from state k to state l is given by a_{kl} and the probability of being in state k is p_k. We get to be in a particular state, k, at some position in the sequence by being at some state, l, at the previous position in the sequence and then transitioning to state k. In terms of probabilities, that means that $p_k = \sum_l a_{lk} p_l$. In other words, p is an eigenvector with eigenvalue 1 of the transpose of the transition matrix. Calculating this eigenvector for the '+' matrix of transitions for CpG islands (slides page 4), yields an eigenvector of approximately $(1, 2.35, 2.35, 1)$ times an arbitrary constant. Adjusting to make the probabilities add up to 1 gives $p = (.15, .35, .35, .15)$

6 Appendix 2

We now use Lagrange multipliers to verify that during supervised learning the transition and emission probabilities a and e obtained by counting maximize the probability $P(X, \pi)$ of generating the sequence X and labels π.

We are looking for

$$\text{argmax}_{a,e} P(X,\pi) = \text{argmax}_{a,e} \prod_i e_{\pi_i}(x_i) \cdot a_{\pi_{i-1}\pi_i} = \text{argmax}_{a,e} \sum_i log(e_{\pi_i}(x_i)) + log(a_{\pi_{i-1}\pi_i})$$

Since the first term in the sum only depends on e, and the second only depends on a, we are looking for:

$$\text{argmax}_e \sum_i log(e_{\pi_i}(x_i))$$

and

$$\text{argmax}_a \sum_i log(a_{\pi_{i-1}\pi_i})$$

We will verify our intuitive answer only for the second sum but the same method works for the first one as well.

Let

$$C_{kl} = \#\text{ of times } \pi_{i-1} = k \text{ and } \pi_i = l$$

Then we can rewrite our expression for a as:

$$\text{argmax}_a \sum_{k,l} C_{kl} log(a_{kl})$$

Fix a particular value of k. Since state k transitions to exactly one other state, we know that $\sum_l a_{kl} = 1$. We want to find the values of $a_{k1}, a_{k2}, ..., a_{kK}$ that maximize $\sum_l C_{kl} log(a_{kl})$ given this constraint. From the method of Lagrange multipliers, we know this maximum occurs at a local extremum of $\sum_l C_{kl} log(a_{kl}) - \lambda \cdot \sum_l a_{kl}$ for some value of λ. Taking derivatives with respect to a_{kl} for each l and setting them all to 0, we get:

$$\frac{C_{kl}}{a_{kl}} = \lambda$$

$$a_{kl} = \frac{C_{kl}}{\lambda}$$

$$\sum_l \frac{C_{kl}}{\lambda} = 1, \text{ so } \lambda = \sum_l C_{kl}$$

$$a_{kl} = \frac{C_{kl}}{\sum_l C_{kl}} = a(k,l)$$

as was to be demonstrated.