A Brief Note on the Hidden Markov Models (HMMs)

ChengXiang Zhai

Department of Computer Science
University of Illinois at Urbana-Champaign

March 16, 2003

1 Introduction

Hidden Markov Models (HMMs) are powerful statistical models for modeling sequential or time-series data, and have been successfully used in many tasks such as speech recognition, protein/DNA sequence analysis, robot control, and information extraction from text data.

There are many good references on HMMs. For example, [1] and [2] are two classic references. You may also find some online tutorials on HMM from the Internet. In this note, we briefly summarize the HMM algorithms with notations that are consistent with our course assignment, and also discribe in detail how to scale quantities such as the forward and backward probabilities in order to avoid underflow when implementing HMMs.

There are two different formulations of HMMs depending on whether to "generate" the output from a *state* or a *transition*. As a probabilistic model, there is no essential difference, though. We assume that an output symbol is always generated from a state and that any state can be a starting state or ending state.

2 Definition of HMM

Formally, we define an HMM as a 5-tuple (S, V, Π, A, B), where $S = \{s_1, ..., s_N\}$ is a finite set of N states, $V = \{v_1, ..., v_M\}$ is a set of M possible symbols in a vocabulary, $\Pi = \{\pi_i\}$ are the initial state probabilities, $A = \{a_{ij}\}$ are the state transition probabilities, $B = \{b_i(v_k)\}$ are the output or emission probabilities. We use $\lambda = (\Pi, A, B)$ to denote all the parameters. The meaning of each parameter is as follows:

- π_i - the probability that the system starts at state i at the beginning
- a_{ij} - the probability of going to state j from state i
- $b_i(v_k)$ - the probabillity of "generating" symbol v_k at state i

Clearly, we have the following constraints

$$\sum_{i=1}^{N} \pi_i = 1$$
$$\sum_{j=1}^{N} a_{ij} = 1 \text{ for } i = 1, 2, ..., N$$
$$\sum_{k=1}^{M} b_i(v_k) = 1 \text{ for } i = 1, 2, ..., N$$

3 Three basic problems

The three problems associated with an HMM are:

1. **Evaluation:** Evaluating the probability of an observed sequence of symbols $O = o_1 o_2 ... o_T$ ($o_i \in V$), given a particular HMM, i.e., $p(O|\lambda)$.

2. **Decoding:** Finding the most likely state transition path associated with an observed sequence. Let $q = q_1 q_2 ... q_T$ be a sequence of states. We want to find $q^* = \text{argmax}_q \; p(q, O|\lambda)$, or equivalently, $q^* = \text{argmax}_s \; p(q|O, \lambda)$.

3. **Training:** Adjusting all the parameters λ to maximize the probability of generating an observed sequence, i.e., to find $\lambda^* = \text{argmax}_\lambda \; p(O|\lambda)$.

The first problem is solved by using the forward and backward iterative algorithms. The second problem is solved by using the Viterbi algorithm, also an iterative algorithm to "grow" the best path by sequentially considering each observed symbol. The last problem is solved by the Baum-Welch algorithm (an EM algorithm), which uses the forward and backward probabilities to update the parameters iteratively.

Below, we give the formulas for each of the step of the computation.

4 The Forward Algorithm

Define $\alpha_t(i) = p(o_1, ..., o_t, q_t = s_i|\lambda)$ as the probability that all the symbols up to time point t have been generated and the system is in state s_i at time t. The α's can be computed using the following recursive procedure:

1. $\alpha_1(i) = \pi_i b_i(o_1)$ (Initially in state s_i and generating o_1)

2. For $1 \leq t < T$, $\alpha_{t+1}(i) = b_i(o_{t+1}) \sum_{j=1}^{N} \alpha_t(j) a_{ji}$ (Generate o_{t+1}, and we can arrive at state s_i from any of the previous state s_j with probability a_{ji})

Note that $\alpha_1(i)$, ..., $\alpha_T(i)$ correspond to the T observed symbols.
It is not hard to see that $p(O|\lambda) = \sum_{i=0}^{N} \alpha_T(i)$, since we may end at any of the N states.

5 The Backward Algorithm

The α values computed using the forward algorithms are sufficient for solving the first problem, i.e., computing $p(O|\lambda)$. However, in order to solve the third problem, we will need another set of probabilities – the β values.

Define $\beta_t(i) = p(o_{t+1}, ..., o_T|q_t = s_i, \lambda)$ as the probability of generating all the symbols *after* time t, given that the system is in state s_i at time t. Just like the α's, the β's can also be computed using the following backward recursive procedure:

1. $\beta_T(i) = 1$ (We have no symbol to generate and we allow each state to be a possible ending state)
2. For $1 \leq t < T$, $\beta_t(i) = \sum_{j=1}^{N} \beta_{t+1}(j) a_{ij} b_j(o_{t+1})$ (Any state s_j can be the state from which o_{t+1} is generated)

It is also easy to see that $p(O|\lambda) = \sum_{i=1}^{N} \alpha_1(i)\beta_1(i)$; in fact, $p(O|\lambda) = \sum_{i=1}^{N} \alpha_t(i)\beta_t(i)$ for any t $1 \leq t \leq T$.

6 The Viterbi algorithm

The Viterbi algorithm is a dynamic programming algorithm that computes the most likely state transition path given an observed sequence of symbols. It is actually very similar to the forward algorithm, except that we will be taking a "max", rather than a "$\sum$", over all the possible ways to arrive at the current state under consideration. However, the formal description of the algorithm inevitablely involves some cumbersome notations.

Let $q = q_1 q_2 ... q_T$ be a sequence of states. We want to find $q^* = \text{argmax}_q\ p(q|O, \lambda)$, which is the same as finding $q^* = \text{argmax}_q\ p(q, O|\lambda)$, since $p(q, O|\lambda) = p(q|O, \lambda)p(O|\lambda)$ and $p(O|\lambda)$ does not affect our choice of q.

The Viterbi algorithm "grows" the optimal path q^* gradually while scanning each of the observed symbols. At time t, it will keep track of *all* the optimal paths ending at each of the N different states. At time $t + 1$, it will then update these N optimal paths.

Let q_t^* be the optimal path for the subsequence of symbols $O(t) = o_1 ... o_t$ up to time t, and $q_t^*(i)$ be the most likely path ending at state s_i given the subsequence $O(t)$, Let $VP_t(i) = p(O(t), q_t^*(i)|\lambda)$ be the probability of following path $q_t^*(i)$ and generating $O(t)$. Thus, $q_t^* = q_t^*(k)$ where $k = arg\max_i VP_t(i)$ and $q^* = q_T^*$. The Viterbi algorithm is as follows:

1. $VP_1(i) = \pi_i b_i(o_1)$ and $q_1^*(i) = (i)$
2. For $1 \leq t < T$, $VP_{t+1}(i) = \max_{1 \leq j \leq N} VP_t(j) a_{ji} b_i(o_{t+1})$ and $q_{t+1}^*(i) = q_t^*(k).(i)$, where $k = \text{argmax}_{1 \leq j \leq N} VP_t(j) a_{ji} b_i(o_{t+1})$ and "." is a concatenation operator of states to form a path.

And, of course, $q^* = q_T^* = q_T^*(k)$, where $k = \text{argmax}_{1 \leq i \leq N} VP_T(i)$.

7 The Baum-Welch Algorithm

Note that the last problem, i.e., finding $\lambda^* = \text{argmax}_\lambda p(O|\lambda)$, is precisely a maximum likelihood estimation problem. If we can also observe the actual state transition path that has been followed to generate the symbols that we observed, then, the estimation would be extremely simple – We simply count the corresponding events and compute the relative frequency. However, the transition path is not observed. As a result, the maximum likelihood estimate, in general, can not be found analytically. Fortunately, just like in the case of the mixture model of k Gaussians discussed in the class, we can also use an EM algorithm for HMM (called Baum-Welch algorithm). Like in all other cases of applying an EM algorithm, we will start with some random guess of the parameter values. At each iteration, we compute the expected probability of all possible hidden state transition paths, and then re-estimate all the parameters based on the expected counts of the corresponding events. The process is repeated until the likelihood converges.

The updating formulas can be expressed in terms of the α's and β's together with the current parameter values. However, it is much easier to understand, if we introduce two other notations. Define $\gamma_t(i) = p(q_t = s_i|O, \lambda)$ as the probability of being at state s_i at time t given our observations, and $\xi_t(i,j) = p(q_t = s_i, q_{t+1} = s_j|O, \lambda)$ as the probability of going through a transition from state i to j at time t given the observations. We have $\gamma_t(i) = \sum_{j=1}^N \xi_t(i,j)$, and also, for $t = 1, ..., T$,

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^N \alpha_t(j)\beta_t(j)} \tag{1}$$

For $t = 1, ..., T-1$, $\xi_t(i,j)$ is given by

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{j=1}^N \alpha_t(j)\beta_t(j)} \tag{2}$$

$$= \frac{\gamma_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\beta_t(i)} \tag{3}$$

The updating formulas for all the parameters are:

- $\pi_i' = \gamma_1(i)$
- $a_{ij}' = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{j=1}^N \sum_{t=1}^{T-1} \xi_t(i,j)}$
- $b_i(v_k)' = \frac{\sum_{t=1, o_t=v_k}^{T} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)}$

8 Implementation of the HMM algorithms

The formulas given above can be implemented as they are, but the code would only be useful for a very short sequence. This is because many quantities would quickly get extremely small as the sequence gets longer. There are generally two ways to deal with the problem: working on the logarithm domain or normalization. Taking logarithm would convert the product of small quantities into a sum of logarithm of the quantities. This would easily work for the Viterbi algorithm. It can also be used for computing the α's and β's. Unfortunately, it is not helpful for computing the γ's, since it would involve a sum of all the $\alpha_t(i)$'s, which would force us to work out of the logarithm domain. Below we present a normalization method that can neatly solve the problem of underflow.

Our idea of normalization is to normalize $\alpha_t(i)$ so that, $\hat{\alpha}_t(i)$, the normalized $\alpha_t(i)$, would be proportional to $\alpha_t(i)$ and sum to 1 over all possible states. That is,

$$\sum_{i=1}^{N} \hat{\alpha}_t(i) = 1$$

and we want

$$\hat{\alpha}_t(i) = \prod_{k=1}^{t} \eta_k \alpha_t(i)$$

so, $\prod_{k=1}^{t} \eta_k = \frac{1}{\sum_{i=1}^{N} \alpha_t(i)} = \frac{1}{p(O(t)|\lambda)}$, and in particular, $\prod_{k=1}^{T} \eta_k = \frac{1}{\sum_{i=1}^{N} \alpha_T(i)} = \frac{1}{p(O|\lambda)}$. Thus, $\hat{\alpha}_t(i) = p(q_t = s_i|O(t), \lambda)$, compared with $\alpha_t(i) = p(O(t), q_t = s_i|\lambda)$.

8.1 The Normalized Forward Algorithm

The $\hat{\alpha}$'s can be computed in the same way as the α's, only we do a normalization at each step.

1. $\hat{\alpha}_1(i) = \frac{\pi_i b_i(o_1)}{\sum_{k=1}^{N} \pi_k b_i(o_1)}$
2. For $1 \leq i < T$, $\hat{\alpha}_{t+1}(i) = \frac{b_i(o_{t+1}) \sum_{j=1}^{N} \hat{\alpha}_t(j) a_{ji}}{\sum_{k=1}^{N} b_k(o_{t+1}) \sum_{j=1}^{N} \hat{\alpha}_t(j) a_{jk}}$

8.2 The Normalized Backward Algorithm

The β's are normalized using the *same normalizers* as used for the α's. That is, $\hat{\beta}_t(i) = \beta_t(i) \prod_{k=t+1}^{T} \eta_k$, (and $\hat{\beta}_T(i) = \beta_T(i)$) , so that we see $\hat{\alpha}_t(i)\hat{\beta}_t(i) = \prod_{k=1}^{T} \eta_k \alpha_t(i)\beta_t(i) = \frac{\alpha_t(i)\beta_t(i)}{p(O|\lambda)}$. Note that, unlike in the case of $\hat{\alpha}$'s, here, we are *not* requiring that the $\hat{\beta}$'s sum to one over all the states at any time point.

We can compute the $\hat{\beta}$'s in exactly the same way as computing the β's, only we will normalize the β's with the already computed η's form the normalized forward algorithm. That is,

1. $\hat{\beta}_T(i) = \beta_T(i) = 1$
2. For $1 \leq t < T$, $\hat{\beta}_t(i) = \eta_{t+1} \sum_{j=1}^{N} \hat{\beta}_{t+1}(j) a_{ij} b_j(o_{t+1})$

8.3 The Updating Formulas Using Normalized α's and β's

We now give the updating formulas in terms of the normalized α's and β's.

The γ's can be computed in the same way, because the normalizers are cancelled. For $t = 1, ..., T$,

$$\gamma_t(i) = \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{\sum_{j=1}^{N} \hat{\alpha}_t(j)\hat{\beta}_t(j)} \tag{4}$$

For $t = 1, ..., T-1$, The ξ's are computed with a slightly different formula:

$$\xi_t(i,j) = \frac{\hat{\alpha}_t(i) a_{ij} b_j(o_{t+1}) \eta_{t+1} \hat{\beta}_{t+1}(j)}{\sum_{j=1}^{N} \hat{\alpha}_t(j)\hat{\beta}_t(j)} \tag{5}$$

$$= \frac{\gamma_t(i) a_{ij} b_j(o_{t+1}) \eta_{t+1} \hat{\beta}_{t+1}(j)}{\hat{\beta}_t(i)} \tag{6}$$

Once we can compute the γ's and ξ's using the normalized α's and β's, we can use the same updating formulas, that is:

- $\pi_i' = \gamma_1(i)$
- $a_{ij}' = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{j=1}^{N} \sum_{t=1}^{T-1} \xi_t(i,j)}$
- $b_i(v_k)' = \frac{\sum_{t=1, o_t = v_k}^{T} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)}$

References

[1] L.R. Rabiner and B.H. Juang. An introduction to hidden markov models. In IEEE ASSP Magazine, 1986. pp. 4–16.

[2] Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. Proc. IEEE, 77 (2), 257-286.