Assignment 9: Hidden Markov Model (HMM) for Protein Secondary Structure Analysis

10-601: Machine Learning - Spring 2011
Roni Rosenfeld
Carnegie Mellon University
TA in charge: Roger (wrhsiao@cs.cmu.edu)

Out on Monday, April 5.
Due on Tuesday, **April 19**, at 1:30pm.

1 Overview

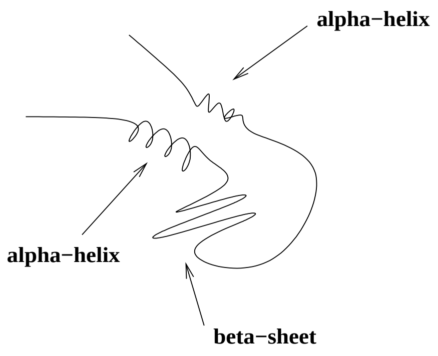
Hidden Markov Models (HMMs) are powerful statistical models for modeling sequential or time-series data, and have been successfully used in many tasks such as speech recognition, protein/DNA sequence analysis, robot control, and information extraction from text data. In this assignment, you will write much of the code responsible for training/learning an HMM and apply the learned HMM to predict the secondary structure of proteins.

This is mainly a programming assignment in C++, but in order to implement the algorithm correctly, you need to understand many details of the underlying probabilistic model. You need to understand the details of several algorithms, including the Viterbi algorithm for finding the most likely transition path and the Baum-Welch algorithm for parameter estimation. We will provide a great deal of "basic" code, so the expected knowledge of C++ is actually at the minimum[1]. You will be asked to "fill in" the required code for three different tasks (algorithms):

1. Compute the most likely path for an observed sequence of symbols from a given HMM using the Viterbi algorithm.
2. Estimate an HMM in a supervised way based on a sequence of symbols along with the *observed* state transitions.
3. Train an HMM in an unsupervised way based on just a sequence of symbols.

You will also be asked to use your HMM program to predict the secondary structure of proteins. A common way of using an HMM is to train an HMM and then find the most likely state transition path for a given sequence of observed symbols. Such an optimal path can be interpreted as providing a sequence of *tags* corresponding to each of the symbols. The tags can also be interpreted as specifying a way of *segmenting* the observed sequence with a segment corresponding to a consecutive sequence of identical tags (i.e. states).

[1] If, for some reason, you are not familiar with C++ or C at all, please talk to the TA (Roger) as soon as you can.



Primary structure

n e g d a a k e f ... ... d g t d i k g g k t g p n l y g a s e e f

A rough secondary structure(A for alpha−helix, B for beta−sheet, O for others)

O O ... O A A ... A O O B B ... B O O ... O A ... A O O ...O

Figure 1: A make-up example of the 3-D structure and corresponding primary and secondary structures of a protein

2 Prediction of Protein Secondary Structure

The problem of predicting protein structures is one of the many possible applications of HMM in the exciting emerging field of bioinformatics. One amazing fact in biology is that the development and functions of any organism, whether it's a human or a virus, are *completely* prescribed in deoxyribonucleic acid (DNA). However, little is known about how such an encoding mechanism actually works; indeed, it is a major challenge in biology and bioinformatics to understand this mechanism.

Surprisingly, you do not really need to know much about biology in order to be able to understand some of these challenges. Conceptually, DNA is a very long sequence of four basic symbols $\{A, T, G, C\}$, each representing a different simple *nucleic acid* unit. It is known that three consecutive nucleic acid units would completely determine an *amino acid*. There are totally 20 distinct amino acids, each is typically represented by a distinct letter. The mapping from the 64 nucleic acid triples to the 20 amino acids is fixed and completely known.

The building blocks of any organism are *proteins*. A protein is nothing but a sequence of amino acids. One part (subsequence) of the whole DNA sequence would determine a sequence of amino acids, i.e., one type of protein, while another would determine another protein. There are usually many different kinds of proteins encoded in a long DNA sequence, but not all parts of a DNA sequence encode proteins – some are *non-encoding regions*. A protein can be represented as a sequence of amino acid symbols, and this is called the *primary (linear) structure* of the protein. In the three-dimensional space, a protein would *fold* into certain shape, and the shape affects its function. Interestingly, the three-dimensional structure of a protein is also determined by its primary structure, in the sense that one type of proteins generally prefers certain shape. However, how a sequence of amino acid symbols exactly determines a particular 3-D shape is largely unknown. There are two types of "basic" shapes that occur frequently in the 3-D structure of proteins: the α-helix and the β-sheet. Thus, at a level above the primary structure, we can tag the amino acid sequence with three different tags: "**in-α-helix**", "**in-β-sheet**", and "**other**". This is called the *secondary structure* of a protein. A possible 3-D structure and the corresponding primary and secondary structures are shown in Figure 1.

Understanding the structure of a protein is essential to understanding how the function of a protein is encoded in the amino acid sequence. In this assignment, we will apply HMM to predict the secondary structure of a protein. Our "observations" will be the amino acid sequences of the proteins. Notice that this is slightly unusual in the following sense: oftentimes we think of the observations as being "caused" by the state. (For instance, think of a robot that has noisy sensors trying to predict the real state of the world.) Here this makes less sense, because if anything, the amino acid sequence determines the secondary structure, rather than vice versa. This, however, will not prevent us from applying the method.

We will only consider some specific proteins that are known to have no $\beta - sheet$ in their secondary structures[2], so the secondary structure only involves two tags: either **in-α** or **other**. More specifically, we will use the Golem dataset that Muggleton and Sternberg used in their research [1]. It is available at

`http://www.doc.ic.ac.uk/~shm/utube.html#golem`.

You can visit the website to know more about this dataset. You can also easily find a lot of tutorial-like material on the Internet about bioinformatics in general.

The original data involves 12 training proteins and 4 testing proteins. Each protein sequence has a known segmentation that indicates where the α-helix is. We suspect that these proteins are known to have only *alpha*-helix, so the whole sequence can be seen as containing two types of alternating regions: either within an α-helix or outside one. To make it easy to finish this assignment, we concatenated all the training (testing) sequences into one long training (testing) sequence. They are stored in two files: `traintag` and `testtag`. Each file is a sequence of pairs. Each pair is on a separate line with the first character denoting the amino acid and the second the region tag (0 for within α-helix and 1 for outside). The file `testseq` has only the sequence of amino acids without the tag information. This is the file we will use for testing. We will run the Viterbi algorithm with an HMM to "decode" this sequence, i.e., to identify which part belongs to an α-helix and which does not. The predicted segmentation can be compared with the true tags in `testtag` to compute the prediction accuracy. For this assignment, we will use the simplest measure of prediction accuracy, which is defined as the percentage of tags that are correct. There is a perl script in the directory (eval.pl) that will compute this accuracy for you. The usage is

```
% eval.pl testtag result
```

3 Summary of the HMM Algorithms

The online slides on the course web site provide an excellent introduction to the HMM and the related algorithms. There are also some other excellent references on HMMs. For example, [2] and [3] are two classic references. Moreover, you can also find lots of online tutorials on HMM from the Internet (e.g., there is one at [4] and a Wikipedia entry at [5]).. Here we will summarize the HMM algorithms that are relevant to this assignment, so as to make the assignment essentially self-contained. We will also cover some of the *differences* in the formulation of the HMM and the scaling of quantities such as the forward and backward probabilities in order to avoid underflow.

The main difference between the formulation given here and that in the course slides is the

[2]This is our understanding of the data set. No verification was made.

initial state distribution. In the course slides, it is assumed that we have both a *fixed* starting state and a *fixed* ending state. Here, we will allow any state to be a starting state and any state to be an ending state[3]. Thus, we will introduce another set of parameters $\Pi = \{\pi_i\}$ for the probability of being in each state at the beginning.

Formally, we define an HMM as a 5-tuple (S, V, Π, A, B), where $S = \{s_0, s_1, ..., s_{N-1}\}$ is a finite set of N states, $V = \{v_0, v_1, ..., v_{M-1}\}$ 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

$$\sum_{i=0}^{N-1} \pi_i = 1$$
$$\sum_{j=0}^{N-1} a_{ij} = 1 \text{ for } i = 0, 1, 2, ..., N-1$$
$$\sum_{k=0}^{M-1} b_i(v_k) = 1 \text{ for } i = 0, 1, 2, ..., N-1$$

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, $q^* = \text{argmax}_q\, p(q|O, \lambda)$.
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 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 (also known as the forward-backward 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.

3.1 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:

[3]In principle, one can always add a special starting state and an ending state to an HMM, but then we will need to deal with the constraints on these two states so that there is no incoming link to the starting state and no out-going link from the ending state.

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

2. For $1 \leq t < T$, $\alpha_{t+1}(i) = b_i(o_{t+1}) \sum_{j=0}^{N-1} \alpha_t(j) a_{ji}$ (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 $p(O|\lambda) = \sum_{i=0}^{N-1} \alpha_T(i)$, since we may end at any of the N states. This is slightly different from the course slides, where $p(O|\lambda) = \alpha_T(s_N)$ as s_N is assumed to be the only ending state. Also note that there are $N+1$ states in the course slides, whereas we have N states.

3.2 The Backward Algorithm

The α values computed using the forward algorithm 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 $1 \leq t < T$, $\beta_t(i) = \sum_{j=0}^{N-1} \beta_{t+1}(j) a_{ij} b_j(o_{t+1})$ (Any state s_j can be the state from which o_{t+1} is generated)

It is also easy to see that $p(O|\lambda) = \sum_{i=0}^{N-1} \alpha_1(i)\beta_1(i)$; in fact, $p(O|\lambda) = \sum_{i=0}^{N-1} \alpha_t(i)\beta_t(i)$ for any t $1 \leq t \leq T$.

3.3 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 $1 \leq t < T$, $VP_{t+1}(i) = \max_{0 \leq j < N} VP_t(j) a_{ji} b_i(o_{t+1})$ and $q^*_{t+1}(i) = q^*_t(k).(i)$, where $k = \text{argmax}_{0 \leq j < N} VP_t(j) a_{ji} b_i(o_{t+1})$ and "." is a concatenation operator of states to form a path.

And, of course, $q^* = q^*_T = q^*_T(k)$, where $k = \text{argmax}_{0 \leq i < N} VP_T(i)$.

3.4 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 the Baum-Welch algorithm or forward-backward 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 $\gamma_t(i) = \sum_{j=0}^{N-1} \xi_t(i,j)$, and also, for $t = 1, ..., T$,

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=0}^{N-1} \alpha_t(j)\beta_t(j)} \tag{1}$$

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

$$\begin{aligned} \xi_t(i,j) &= \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{\sum_{j=0}^{N-1} \alpha_t(j)\beta_t(j)} && (2) \\ &= \frac{\gamma_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{\beta_t(i)} && (3) \end{aligned}$$

The updating formulas for all the parameters are:

- $\pi'_i = \gamma_1(i)$
- $a'_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{j=0}^{N-1} \sum_{t=1}^{T-1} \xi_t(i,j)}$
- $b_i(v_k)' = \frac{\sum_{t=1, o_t = v_k}^{T} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)}$

4 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 get 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 $\alpha_t(i)$ such 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,

$$\sum_{i=0}^{N-1} \hat{\alpha}_t(i) = 1$$

and we want

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

so, $\prod_{k=1}^{t} \eta_k = \frac{1}{\sum_{i=0}^{N-1} \alpha_t(i)} = \frac{1}{p(O(t)|\lambda)}$, and in particular, $\prod_{k=1}^{T} \eta_k = \frac{1}{\sum_{i=0}^{N-1} \alpha_T(i)} = \frac{1}{p(O|\lambda)}$. Thus, $\hat{\alpha}_t(i) = p(q_t = s_i | O(t), \lambda)$, compared with $\alpha_t(i) = p(O(t), q_t = s_i|\lambda)$.

4.1 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. $\hat{\alpha}_1(i) = \eta_1 \pi_i b_i(o_1)$, where $\eta_1 = \frac{1}{\sum_{k=0}^{N-1} \pi_k b_k(o_1)}$

2. For $1 \leq t < T$, $\hat{\alpha}_{t+1}(i) = \eta_{t+1} b_i(o_{t+1}) \sum_{j=0}^{N-1} \hat{\alpha}_t(j) a_{ji}$, where $\eta_{t+1} = \frac{1}{\sum_{k=0}^{N-1} b_k(o_{t+1}) \sum_{j=0}^{N-1} \hat{\alpha}_t(j) a_{jk}}$

4.2 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 η's which were already computed in the normalized forward algorithm. That is,

1. $\hat{\beta}_T(i) = \beta_T(i) = 1$

2. For $1 \leq t < T$, $\hat{\beta}_t(i) = \eta_{t+1} \sum_{j=0}^{N-1} \hat{\beta}_{t+1}(j) a_{ij} b_j(o_{t+1})$

4.3 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 $t = 1, ..., T$,

$$\gamma_t(i) = \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{\sum_{j=0}^{N-1} \hat{\alpha}_t(j)\hat{\beta}_t(j)} \tag{4}$$

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

$$\xi_t(i,j) = \frac{\hat{\alpha}_t(i)a_{ij}b_j(o_{t+1})\eta_{t+1}\hat{\beta}_{t+1}(j)}{\sum_{j=0}^{N-1} \hat{\alpha}_t(j)\hat{\beta}_t(j)} \tag{5}$$

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

Now that we can compute the γ's and ξ's using the normalized α's and β's, we can update the parameters using exactly the same updating formulas as we see in the previous section. That is,

- $\pi'_i = \gamma_1(i)$
- $a'_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{j=0}^{N-1} \sum_{t=1}^{T-1} \xi_t(i,j)}$
- $b_i(v_k)' = \frac{\sum_{t=1, o_t=v_k}^{T} \gamma_t(i)}{\sum_{t=1}^{T} \gamma_t(i)}$

You might have already noticed that the updating formulas generally involve the accumulation of some expected counts (expressed in terms of γ's and ξ's) over all the possible time points $t = 1, 2, ..., T$, and the expected counts at each time point t depend only on the $\hat{\alpha}_t(i)$'s, $\hat{\beta}_t(i)$'s, $\hat{\beta}_{t+1}(i)$'s, and η_{t+1}, in addition to the model parameters. Thus, in the most efficient implementation of Baum-Welch algorithm, such accumulation can be done at the same time of computing the $\hat{\beta}$'s. In this assignment, however, we will implement the accumulation of counts as a separate pass.

5 The HMM Code Provided

We have provided a significant amount of code in C++ for this assignment. This includes three files { `hmm.hpp, hmm.cpp, main.cpp`} and a makefile. The code and the necessary sequence data are all in `http://cs.cmu.edu/~roni/10601-s11/assignments/hw9/basecode.tar.gz`. The code is complete except that several key functions in hmm.cpp are not implemented. Your task is to implement them based on the algorithms specified in the previous section. **Important:** You must implement the *normalized* version of the formulas as described in Section 4.

The program can be run in three different modes, depending on the option given.

1. With option `-d`, it *decodes* a given sequence, i.e., computes the most likely path of state transitions given the sequence. The required input includes a *model file* and a *sequence file* (without tags).

2. With option `-c`, it *counts* the relevant events in a given *tagged* sequence and saves the model in the model file given. That is, it estimates the HMM parameters in a *supervised* way. The required input includes the number of states, a *model file name* and a *tagged sequence file*.

3. With option `-t`, it *trains* an HMM with a raw sequence without tags, and saves the model in the model file given. That is, it estimates the HMM parameters in an *unsupervised* way. The required input includes the number of states, a *model file name* and a *raw sequence file*.

In any case, the program recognizes the following options:

- `-m filename` specifies a model file
- `-s filename` specifies a sequence file (tagged or untagged)
- `-n integer` specifies the desired number of states

You need to implement or complete the implementation of the following functions in the file `hmm.cpp`.

1. `void Decode(char *seqFile)`: This function should read in a sequence and compute the most likely state path using the Viterbi algorithm. It should print out a tagged sequence similar to `testtag`.

2. `void CountSequence(char *seqFile)`: This function should read in a tagged sequence file and estimate all the HMM parameters based on the counting of the corresponding events in the tagged sequence.

3. `void ComputeAlpha (int *seq, int seqLength)`: This function computes the normalized α values using the forward algorithm. `seq` has the sequence of observed symbols, and is an array of length `seqLength`. It is 0-indexed, that is, the first observed symbol o_1 is in `seq[0]` and the last observed symbol o_T is in `seq[seqLength-1]`. You can think of `seqLength` as T in our formulas.

4. `void ComputeBeta (int *seq, int seqLength)`: This function computes the normalized β values using the backward algorithm. `seq` is the same as in `ComputeAlpha`.

5. `void AccumulateCounts (int *seq, int seqLength)`: This function computes the γ's and ξ's and accumulates the counts for all the parameters. `seq` is the same as in `ComputeAlpha`.

The following is a brief explanation of how an HMM is represented in the provided code.

An HMM is represented by the class `Model`. We assume a fixed vocabulary that consists of all the 94 "readable/printable" ascii characters that are typically on a keyboard. More precisely, the symbol set is fixed as all the 94 ascii characters from `[33,!]` to `[126, ~]`. When a symbol is not observed in the data, its count is automatically set to zero, effectively excluded from the model.

The number of state is stored in a variable N. The parameters of an N-state HMM are represented by the following arrays:

1. Initial state probability (`I`)

   `I` is an array of length N (0-indexed), with `I[s]` representing the initial probabiltiy of being at state `s`.

2. State transition probability matrix (`A`)

   `A` is a 0-indexed two-dimensional array ($N \times N$), with `A[i][j]` representing the probability of going to state `j` from state `i`.

3. Output probability matrix (`B`)

   `B` is a 0-indexed two-dimensional array ($M \times N$), with `B[o][s]` representing the probability of generating output symbol `o` at state `s`. M is the number of unique symbols (currently 94, stored in `SYMNUM`). Note that the observed character can *not* be used directly as an index to access the entries of matrix `B`; it must be normalized by subtracting the "lowest" character `baseChar` defined in the class `Model`. For example, to find out the probability of generating `'C'` from state 1, use `B['C'-baseChar][1]`.

More specific instructions on how to implement them can be found in the embedded documentation in the provided code. In all the cases, run "`gmake`" to recompile the code after you change the code.

6 Finding the Most Likely State Path (Viterbi Algorithm) [30 points]

(a) (5 points) Give an explanation of your HMM. Especially, how many states do you have? What do the states represent? What is the observation vocabulary? What parameters does your HMM have?

(b) (10 points) Complete the implementation of the `Decode` function. The function is mostly implemented except the part of Viterbi iteration for growing the most likely path. Fill in the appropriate code to make it work. The frame code provided in the `Decode` function is meant to minimize your work on implementing the Viterbi algorithm, but you should feel free to change any part of the provided code in the `Decode` function, including implementing the whole function on your own. Using logarithm to avoid underflow.

Test your code using the two example model files and sequence files provided. That is, to run the following command:

```
% hmm -d -m samplemod1 -s sampleseq1 > sampletag1
% hmm -d -m samplemod2 -s sampleseq2 > sampletag2
```

Your program should tag every "`a`" with "0" and every "`b`" with "1" for `sampleseq1`, and all the first eight symbols with "0" and the last eight symbols with "1" in the case of `sampleseq2`.

(c) (5 points) If the results are as expected for these two examples, great! Continue to run the decoding algorithm on the real protein sequence. Do the following:

```
% hmm -d -m mod.test -s testseq > result6c
```

"mod.test" is an HMM constructed based on the tags available on the testing sequence. Evaluate the prediction accuracy with "eval.pl". What is the accuracy? The segmentation accuracy should be above 0.7. If not, your implementation is probably incorrect.

(d) (10 points) Take a look at the model specified in `samplemod1` and `samplemod2`. Explain briefly why the program should tag the two sequences as it did. In particular explain briefly why it tagged the first `b` with "0" and the last `a` with "1" in `sampleseq2` even though the output probability distribution given state "0" strongly favors `a` and the output probability distribution given state "1" strongly favors `b`. Modify `samplemod2` so that, when decoding `sampleseq2`, the program would tag all symbols with "0" except the last two `b`'s which would be tagged as "1". Name the modified model file as `samplemod6d`.

7 Supervised Training of HMM [25 points]

(a) (10 points) Complete the implementation of the function `CountSequence`. This function is mostly implemented except the part of counting the relevant events for estimating the HMM parameters. Fill in the appropriate code so that it will count the following with the corresponding counters.

- How many times it starts with state s_i (use `ICounter`)
- How many times a transition (from state s_i to s_j) has happened (use `ACounter`)
- How many times character c is generated from state s_i (use `BCounter`)

For each of the three counters, change the counter as well as the corresponding normalizer (i.e., INorm, ANorm, and BNorm). (These normalizers are to normalize the counts in order to obtain probabilities. They have nothing to do with the normalization discussed in Section 4, which is to avoid underflow.) Take a look at the function `UpdateParameter()`, if you are not sure how these counters and normalizers are used in the provided code.

Test your code using the tagged sequence (`taggedsampleseq1`) by running the following command: (`taggedsampleseq1` should be identical to the `sampletag2` that you got.)

```
% hmm -c -n 2 -m samplesupmod7a -s taggedsampleseq1
```

Take a look at `samplesupmod7a`. If your implementation is correct, you should get a model that would tend to generate "a" when at state "0" and tend to generate "b" when at state "1".

(b) (10 points)

Now train an HMM using the testing protein data and test the HMM on the raw testing sequence. That is, do the following

```
% hmm -c -n 2 -m mod7b.test -s testtag
% hmm -d -m mod7b.test -s testseq > result7b1
```

Evaluate `result7b1` with `eval.pl`. What is the prediction accuracy? It should be very similar to what you got in 6(b).

Now train an HMM using the *training* protein data and test the HMM on the raw *testing* sequence. That is, do the following

```
% hmm -c -n 2 -m mod7b.train -s traintag
% hmm -d -m mod7b.train -s testseq > result7b2
```

Evaluate `result7b2` with `eval.pl`. What is the prediction accuracy? It should be above 0.55. If not, you may not have implemented the algorithm correctly. How is `result7b2` compared with `result7b1`? Which one has a better accuracy? Why?

(c) (5 points) Compare `samplesupmod7a` and `samplemod2`. Are they similar? Is this expected? How are they different? Give an example of sequence for which `samplesupmod7a` would give a different decoding than `samplemod2` (i.e., When running with option `-d`, you will get different results). Name the sequence file `sampleseq7c`.

8 Unsupervised Training of HMM (Baum-Welch Algorithm) [45 points]

(a) (30 points) In this part of the assignment, you are asked to complete the implementation of training an HMM with the Baum-Welch algorithm. The high level process, e.g., the iterations, is already implemented in the function `UpdateHMM`. You will need to implement function `ComputeAlpha`, `ComputeBeta`, and complete function `AccumulateCounts` by inserting appropriate code for computing the γ's and ξś and the code for accumulating the expected counts for the three groups of parameters according to the formulas given in Section 4.

There are more concrete instructions on how to implement each of the three functions within the code. You might find it useful (for debugging) to uncomment the function call to print out the α's and the β's in the function `UpdateHMM`.

(b) (5 points)

Test your implementation by training on `sampleseq2`

```
% hmm -t -n 2 -m sampleunsupmod8b -s sampleseq2
```

Take a look at `sampleunsupmod8b`. Does the learned model capture some of the patterns in sampleseq2? Repeat the command above 20 times. Do you get the same model in `sampleunsupmod8b` and the same likelihood every time? Explain why you might get a different model with a different likelihood sometimes.

Now choose a model with the highest likelihood from the 20 runs, and Compare the corresponding `sampleunsupmod8b` with `samplemod2`. Are they similar? Remember that since we are doing *unsupervised* training, state "0" and "1" may switch their roles. Now, use this `sampleunsupmod8b` to decode `sampleseq2`, i.e., do,

```
% hmm -d -m sampleunsupmod8b -s sampleseq2 > result8b
```

What do you get? Did you get the same tagging as `sampletag2`, which is produced by using `samplemod2`.

(c) (10 points)

Now, train the HMM on the protein testing data (i.e., assuming no labelled training data).

```
% hmm -t -n 2 -m mod8c.test -s testseq
% hmm -d -m mod8c.test -s testseq>result8c
```

Note that since this is unsupervised training, the two states (0 and 1) may pick up their roles fairly arbitrarily, i.e., either state 0 or state 1 can mean within an α-helix. So, you need to evaluate the results in two different ways: assuming either 0 or 1 to be within an α-helix. For such evaluation, we provided a different perl script `evalflip.pl`. You should use `evalflip.pl` to evaluate `result8c`. The usage is exactly the same as `eval.pl`.

Repeat this process at least 10 times. Do you get the same likelihood, the same model, and the same accuracy every time? Record and report the accuracy each time. What is the *best* of all the recorded accuracy numbers? How is this best accuracy compared with the accuracy of assigning the same tag to all the symbols in `testseq`, which is 0.5216 when evaluated using `evalflip.pl`? How is this best accuracy compared with `result7b1` and `result7b2`? Take a look at the segmentation provided by this best model, and visually compare it with the true segmentation in `testtag`. Does the trained model seem to capture some of the true segment boundaries?

Notes:

- This is a big programming assignment and can take more time than you think, so start early. Contact Roger as soon as you feel you may need help, or if you have any questions.
- What to hand in:
  1. Make sure your code also compiles and runs as expected on the Andrew UNIX servers (ssh to unix.andrew.cmu.edu).
  2. Copy the modified `hmm.cpp` to the afs handin directory (i.e., `/afs/andrew.cmu.edu/course/10/601/s11/handin/YOUR_ANDREW_ID`/hw9). You must make sure it compiles with the provided `hmm.hpp` and `main.cpp`. Your code will be tested with some new examples.
  3. Copy the following files to your handin directory:
     - 6b: sampletag1,sampletag2
     - 6c: result6c
     - 6d: samplemod6d
     - 7a: samplesupmod7a
     - 7b: result7b1, result7b2, mod7b.test, mod7b.train
     - 7c: sampleseq7c
     - 8b: sampleunsupmod8b, result8b
     - 8c: mod8c.test giving the highest accuracy, result8c
  4. All other written solutions should be turned in as hard copies at the beginning of the class on the due date.
  5. Remember to explain your results. And don't forget to add comments in your code.

References

[1] Muggleton S., King R.D., and Sternberg M.J.E. (1992). Predicting protein secondary structure using inductive logic programming. in Protein Engineering, 5:647–657.

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

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

[4] http://www.cs.brown.edu/research/ai/dynamics/tutorial/Documents/HiddenMarkovModels.html.

[5] http://en.wikipedia.org/wiki/Hidden_Markov_model.