10601: Machine Learning **Spring 2013**

Homework 10 : Hidden Markov Model (HMM) for
Protein Secondary Structure Analysis

Handed Out: 17 Apr 2013 *Due:* 24 and 29 Apr 2013

TA-in-charge: Meghana `mkshirsa@andrew.cmu.edu`

- For `Makefile` instructions, please see Piazza.
- This homework involves a lot of elaborate coding, so start as early as you can!
- We have divided the deliverables into two parts which are due on two different dates.
- Question 5 is due on 24th Apr (Wednesday). Questions 6, 7 are due on 29th Apr (the following Monday).
- The deliverables for Question 5 (code, model, outputs) should all go into the handin folder `hw10a` and that for Questions 6, 7 should be in `hw10b`.

1 Overview

This is mainly a programming assignment that will test your understanding of the probabilistic model underlying HMM, the Viterbi algorithm and the Baum-Welch algorithm.

The choice of programming language is left to you. However, we are providing a lot of framework C++ code that you could use, but note that the TA cannot provide any debugging assistance if you choose to use the C++ code. The accompanying documentation and the comments in the code will help you understand what it does. Irrespective of what programming language you use, **you are required** to follow a framework - you will have to ensure there are methods or functions or modules in your code to do specific tasks. For example, your code should have a method `ComputeAlpha` that computes the forward probabilities. These requirements are mentioned at the end of the problem description in Section 8.

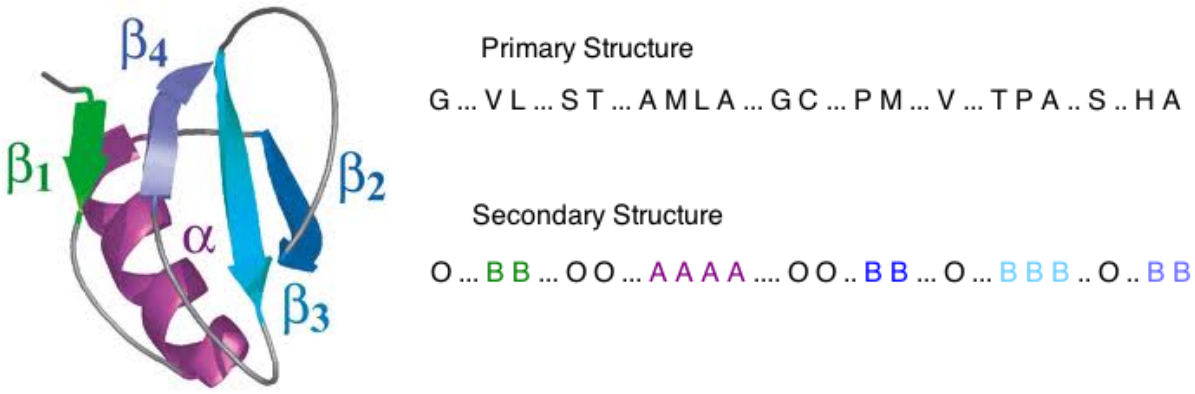
Information about the files provided:

- The C++ code along with documentation is in `c++_code.zip` and all data is in `data.zip`.
- In order to help debug your implementation, we are providing you some dummy input files. Each question will describe their use along with what the correct output should be. Compare your code's output with the correct output to know if it's working fine. These files are: `sampleseq1, sampleseq2, samplemod1, samplemod2, taggedsampleseq1`.
- Training and test data for the homework: `traintag, testtag, testseq`.

You are going to write code to perform three different tasks:

1. **Decoding (i.e Inference)**: 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 using a labeled/tagged sequence of symbols.
3. **Parameter estimation (i.e Training)**: Train an HMM in an unsupervised way based on a sequence of untagged symbols using *Baum-Welch algorithm.*

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



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. A common way of using an HMM is to 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).

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. Luckily, you do not really need to know much about biology in order to be able to understand some of these challenges.

The building blocks of any organism are *proteins*. A protein is molecule formed from a sequence of amino acids. There is a total of 20 distinct amino acids, each is typically represented by a distinct letter. The protein molecule is formed via a process called *folding*, in which the one-dimensional sequence of amino acids is converted to a three-dimensional structure. However, the mechanism by which a sequence of amino acid symbols is mapped to a particular 3-D shape is largely unknown. Two important aspects of a protein's molecular structure are: (a) the *primary (linear) structure* which refers to the amino acid sequence and (b) *secondary structures* which refer to repeating units or "basic" shapes that occur frequently in the 3-D structure. There are two types of secondary structure that are particularly common: the α-helix and the β-sheet which are formed from parts of the primary structure sequence. Thus, we can tag the amino acid sequence with three different tags: "**in-α-helix**", "**in-β-sheet**", and "**other**". A possible 3-D structure and the corresponding primary and secondary structures are shown in Figure 1.

Understanding the secondary structure of a protein is essential to understanding how the function of a protein is encoded. In this assignment, we will apply HMM to predict the secondary structure of a protein given the primary structure. 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. Here this interpretation makes less sense, because the amino acid sequence determines the secondary structure, rather than vice versa.

For this assignment our dataset has only some specific proteins that are known to have no $\beta - sheet$ in their secondary structures[1], 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].

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

It is available at

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

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. The whole sequence can be seen as containing two types of alternating regions: either within an α-helix or outside it. We concatenated all the 16 sequences (both training and test) into one long 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, 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. Two classic references on HMMs are: [2] and [3]. Here we will summarize the HMM algorithms that are relevant to this assignment, to make it 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 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[2]. 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

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

[2] 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.

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:

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

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

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

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

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

The updating formulas for all the parameters are:

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

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 $1 \le t < T$, $\hat{\alpha}_{t+1}(i) = \eta_{t+1}b_i(o_{t+1})\sum_{j=0}^{N-1}\hat{\alpha}_t(j)a_{ji}$, 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 Finding the Most Likely State Path (Viterbi Algorithm) [30 points]

(a) (15 points) Complete the implementation of the `Decode` function. In the C++ code, this 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. Use logarithm to avoid underflow.

Test your code by using the two sample model files: `samplemod1`, `samplemod2` to tag the sequence files `sampleseq1` and `sampleseq2` respectively. Save the output into two files `sampletag1` and `sampletag2` and compare with the correct output described next. If correct, 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`.

(b) (5 points) If the results are as expected for these two examples, great! Continue to run the decoding algorithm on the real protein sequence: `testseq` using the HMM model `mod.test`. Save the resulting labels in a file `result5b`. Evaluate the prediction accuracy of `result5b` with "eval.pl". What is the accuracy? The segmentation accuracy should be above 0.7. If not, your implementation is probably incorrect.

(c) (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 `model5c`.

6 Supervised Training of HMM [25 points]

(a) (10 points) Complete the implementation of the function `CountSequence` which computes the following counts. If using the C++ code, update the corresponding counters.

- How many times is the start 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, do not forget to compute 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` and save the model in a file `model6a`. 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 tagged protein data file `testtag` and test the model on the raw sequence `testseq`. Save your output tags on the test sequence in a file called `result6b1`. Evaluate `result6b1` with `eval.pl`. What is the prediction accuracy? It should be very similar to what you got in 5(b).

Now train an HMM using the tagged protein data in `traintag` and test the HMM model on the raw sequence `testseq`. Save the output tags in a file `result6b2`. Evaluate `result6b2` 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 `result6b2` compared with `result6b1`? Which one has a better accuracy? Why?

(c) (5 points) Compare the model you got in part 6(a) and `samplemod2`. Are they similar? Is this expected? How are they different? Give an example of sequence for which the two models would output a different decoding (i.e result from Viterbi are different). Name the sequence file `seq6c`.

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

(a) (30 points) Implement Baum-Welch algorithm. You will need to implement the functions `ComputeAlpha`, `ComputeBeta`, and `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.

In the C++ code provided, the high level process, e.g., the iterations, is already implemented in the function `UpdateHMM`. 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`. Does the learned model capture some of the patterns in `sampleseq2`? Repeat the command above 20 times. Do you get the same model 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 call it `bestmod7b`. Compare it with `samplemod2`. Are they similar? Remember that since we are doing *unsupervised* training, state "0" and "1" may switch their roles. Now, using `bestmod7b` decode `sampleseq2` and save the resulting labels in `result7b`. Did you get the same tagging as `sampletag2` which was your output in Q 5(a)?

(c) (10 points) Now, train the HMM on the protein data in `testseq` (there are no labels). After training, also output the labels produced on this sequence. 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 your result. The usage is exactly the same as `eval.pl`.

Repeat the training 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? Save the best model into a file `bestmod7c` and the resultant output into a file `result7c`. 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 `result6b1` and `result6b2`? 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.
- What to hand in:
  1. Make sure your code also compiles and runs as expected on the Andrew UNIX servers. Your code will be tested with some new examples.
  2. Copy the following result files to your appropriate handin directory (either `hw10a` or `hw10b` based on the question):
     - 5b: result5b
     - 5c: model5c
     - 6a: model6a
     - 6b: result6b1, result6b2
     - 6c: seq6c
     - 7b: bestmod7b, result7b
     - 7c: bestmod7c, result7c
     - All other written solutions.
  3. Remember to explain your results. And don't forget to add comments in your code.

8 Framework with which your code should comply

The following functions, methods should exist in your code to perform the appropriate process. The input to each function has also been indicated.

- `Decode`: Input: `seqFile`. This function should read in a sequence and compute the most likely state path using the Viterbi algorithm.
- `CountSequence`: Input: `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.
- `ComputeAlpha`: Input: `obsFile`. This function computes the normalized α values for a given observation sequence using the forward algorithm.
- `ComputeBeta`: Input: `obsFile`. This function computes the normalized β values using the backward algorithm.
- `AccumulateCounts`: Input: `obsFile`. This function computes the γ's and ξ's and accumulates the counts for all the parameters for a given sequence of observed symbols.

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.