

Project Option 1: The world's simplest HMM-based genefinder

October 9, 2007

* This just gives you some idea about the project. Feel free to change the experiments or add anything you think interesting. Adapted from Sean Eddy's Computational Biology course at WashU.

The base composition of most genome sequences is not homogeneous. In particular, GC composition can vary regionally. In the human genome, for instance, there are "CpG islands" which show a very strong statistical signal and which tend to mark the 5' end of many genes. Therefore the problem of segmenting a genome sequence into regions of different compositions arises naturally. One way to segment a genome is with HMM algorithms.

Let's assume that a genome can be modeled as a two-state HMM, with two states A and B. Assume that the parameters of this HMM are as follows. State A has an AT-biased emission distribution 0.35, 0.15, 0.15, 0.35 for the probabilities of A,C,G,T. State B has a GC-biased emission distribution 0.15, 0.35, 0.35, 0.15. State A switches to state B with probability 0.001. State B switches back to state A with probability 0.01. Assume that the initial distribution for the HMM is uniform; 0.5, 0.5 for the two states. The file `example.fa` contains a simulated 1 Mb genome sequence, generated by this HMM.

The parameters is saved in a file `example.hmm` with the following format:

```
2 4 ACGT
.5 .5
.999 .001 .35 .15 .15 .35
.01 .99 .15 .35 .35 .15
```

where the first row shows the number of states, number of symbols, and the symbols themselves. The second row specifies the initial probability for each state. The remaining lines are the transition and emission probabilities. Each line represents a state, and the first K columns on each line are the transition probability for a K -state HMM, followed by emission probabilities of all possible symbols.

1 HMM model I/O and sequence generator

Implement a HMM class that provides the following functions:

- (1) Read a .HMM file to initialize parameters.
- (2) Output a HMM model with the format shown above.
- (3) Generate a sequence according to the model parameters, and output the sequence together with the correct state path (using a format similar to Viterbi decoding output below).
- (4) Read in a sequence and a state path to estimate the parameters.

Use your program to generate 1Mb sequence (or different lengths to test how your program performs with different lengths of sequences) according to the above HMM model. Then use your program to estimate parameters to check that they are close to your true model.

2 Viterbi Decoding

Implement a Viterbi parser for the HMM above, including the initialization, matrix fill, and traceback stages. Your program should output the decoding as a series of segments, e.g.:

```
1    153    A
154  252    B
253  1651   A
```

Do some crude calculation of your decoding accuracy. (It's probably more meaningful to estimate the accuracy for state A and state B separately, because there are more symbols in A than in B).

3 Posterior decoding

Implement a posterior decoding parser for the HMM above, including forward, backward, and the forward/backward calculation to obtain the probability that each residue i in the sequence is in state A versus state B. For decoding you can treat all positions where the posterior probability of state B is $i = 0.5$ as in state B, and the rest as in A. But it's probably a better idea to remove short stretches of B's. What is your accuracy with and without the filter?

4 Expectation/maximization

Use your forward and backward implementations (above), implement an expectation maximization algorithm to estimate the parameters of an HMM from randomized starting parameters. (You can assume that the initial distribution is still uniform - you only have one starting point, so it's not really useful to try to estimate the initial distribution from the data. Just estimate the transition

and emission probabilities of the two-state HMM.) Run your EM algorithm on the randomly generated genome sequence in `example.fa`. Does your algorithm find the same parameters that actually generated the genome sequence?

To test whether EM convergence, sum the difference between the new and old parameter set after each iteration. When the difference is less than some small number (say 0.001), it is usually safe to assume convergence. You can also print out the parameters every 10 iterations to visually observe the convergence. Another way to check convergence is to print out $\log(P(seq))$, which can be obtained by the forward algorithm. This quantity should be increasing until it reaches a local optimal (otherwise you've probably got a bug.)

Try several sets of initial parameters, and observe whether the program always converge to the same parameters.

5 A real application: identifying structural RNA genes in *Methanococcus*

The *Methanococcus jannaschii* genome has an interesting property. On average, the base composition of the genome is strongly AT-biased. Some of its genes, though – the structural RNA genes – are strongly GC-biased. It turns out that the GC composition of bulk genomes appears to vary freely, but the GC composition of structural RNA genes is strongly correlated with the normal growth temperature of the organism (we think this is because structural RNAs have to maintain a base paired secondary structure; thus, at high temperature, they drive towards high GC because folded high GC sequences have higher thermostability).

The parameters above (and in the `example.hmm`) are pretty close to what you get from estimating a two-state HMM on *M. jannaschii* bulk genome and structural RNA genes, with state A as the bulk genome and state B as the RNA genes.

A slightly modified copy of the complete *M. jannaschii* genome will be posted on the course website.

(1) Run your viterbi and posterior decoding parsers on the real *M. jannaschii* genome. How many RNA regions do you detect? A file containing the positions of known transfer RNA genes is available on the course website. How many of these are detected by the Viterbi parser and the posterior parser?

(2) Run your EM algorithm on the *M. jannaschii* genome. Do you estimate about same parameters as above, or does your EM algorithm find a different kind of segmentation pattern in the genome?

Note: Numerics – Use log probabilities for both versions.

In Viterbi this is completely straightforward: products of probabilities become sums of log probs; max is max. In the forward/backward algorithms, however, you need sums of products of probabilities. Mathematically, $\log(x + y) = \log(\exp(\log x) + \exp(\log y))$, but there's some danger of underflow/loss of precision. Instead, do something like this (C syntax):

```
/*
 * Given two probabilities x and y, represented by their logs lx, ly,
 * return the log of their sum log(x+y) = log(exp(lx) + exp(ly)).
 *
 * Assume log(0) is represented by NaN.
 *
 * The "lx > ly" trick is some protection from underflow:
 * log(a+b) = log(a(1+b/a)) = log(a)+log(1+b/a),
 * which will be most accurate when b/a < 1.
 */
double log_of_sum_of_logs(double lx, double ly){
    if(isnan(lx)) return ly;
    if(isnan(ly)) return lx;
    if(lx > ly) {
        return lx + log(1+exp(ly-lx));
    } else {
        return ly + log(1+exp(lx-ly));
    }
}
```