Today’s Lecture

• Parameter estimation
  – Viterbi training

• Forward & forward/backward algorithms
Hidden Markov Model

observed symbols

A → G → C → A → T

unobserved states

0 → π₁ → π₂ → π₃ → ... → πₙ → 0
WDAG for 3-state HMM, length $n$ sequence

weights are emission probabilities $e_k(b_i)$ for $i^{th}$ residue $b_i$

weights are transition probabilities $a_{kl}$

$b_{i-1}$ position $i-1$

$b_i$ position $i$

$b_{i+1}$ position $i+1
• **Paths** through graph from begin node to end node correspond to **sequences of states**

• **Product weight** along path
  
  = **joint probability** of state sequence & observed symbol sequence

• **Highest-weight path** = **highest probability state sequence**

• **Sum of (product) path weights, over all paths,**
  
  = **probability of observed sequence**

• **Sum of (product) path weights over**
  
  – all paths going through a particular node, or
  
  – all paths that include a particular edge,

  **divided by** prob of observed sequence,

  = **posterior probability** of that edge or node
Complexity

• $= O(|V|+|E|)$, i.e. total # nodes and edges.
• # nodes = $2ns + 2$
  – where $n =$ sequence length,
  – $s =$ # states.
• # edges = $(n - 1)s^2 + ns + 2s$

• So overall complexity is $O(ns^2)$
  – (actually $s^2$ can be reduced to # ‘allowed’ transitions between states – depends on model topology).
HMM Parameter Estimation

• Suppose parameter values (transition & emission probs) unknown
• Need to estimate from set of training sequences
• **Maximum likelihood** (ML) estimation (= choice of param vals to maximize prob of data) is preferred
  – optimality properties of ML estimates discussed in Ewens & Grant
Hidden Markov Model

observed symbols

unobserved states

$e_{\pi_1}(A)$

$e_{\pi_2}(G)$

$e_{\pi_3}(C)$

$\ldots e_{\pi_i}(A)$

$\ldots e_{\pi_n}(T)$

$\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \pi_3 \rightarrow \pi_i \rightarrow \pi_{i+1} \rightarrow \pi_n \rightarrow 0$
Parameter estimation when state sequence is known

• When underlying state sequence for each training sequence is known,
  – e.g.: site model

then ML estimates are given by:

  – emission probabilities:
    \[ \hat{e}_k(b) = \frac{\text{# times symbol } b \text{ emitted by state } k}{\text{# times state } k \text{ occurs}}. \]

  – transition probabilities:
    \[ \hat{a}_{kl} = \frac{\text{# times state } k \text{ followed by state } l}{\text{# times state } k \text{ occurs}}. \]

  – in denominator above, *omit occurrence at last position of sequence* (for transition probabilities)
    • But include it for emission probs

  – can include pseudocounts, to incorporate prior expectations/avoid small sample overfitting (Bayesian justification)
Parameter estimation when state sequence unknown

- **Viterbi training**
  1. choose starting parameter values
  2. find highest weight paths (Viterbi) for each sequence
  3. estimate new emission and transition probs as above, assuming Viterbi state sequence is true
  4. iterate steps 2 and 3 until convergence
     - not guaranteed to occur – but nearly always does
  5. does *not* necessarily give ML estimates, but often are reasonably good
Hidden Markov Model

observed symbols

unobserved states

0 → π₁ → π₂ → π₃ → ... → πᵢ → πᵢ₊₁ → πₙ → 0

e_{π₁}(A) e_{π₂}(G) e_{π₃}(C) ... e_{πᵢ}(A) ... e_{πₙ}(T)

a_{0 π₁} a_{π₁ π₂} a_{π₂ π₃} a_{π₃ π₄} ... a_{πᵢ πᵢ₊₁}
More algorithms

• Can also use dynamic programming to find
  – sum of all product path weights
    = "forward algorithm" for probability of observed sequence
  – sum of all product path weights through particular node or particular edge
    = "forward/backward algorithm" to find posterior probabilities

• Now must use product weights and non-log-transformed probabilities
  – because need to *add* probabilities
• In each case, compute successively for each node (by increasing depth: left to right)
  – the sum of the weights of all paths ending at that node.
  – N.B. paths are constrained to begin at the begin node!

• In forward/backward algorithm,
  – work through all nodes a second time, in opposite direction
    • i.e. in reverse graph – constraining paths to start in rightmost column of nodes
For each vertex $v$, let $f(v) = \sum_{\text{paths } p \text{ ending at } v} \text{weight}(p)$, where $\text{weight}(p) = \text{product} \text{ of edge weights in } p$. Only consider paths starting at ‘begin’ node.

Compute $f(v)$ by dynam. prog: $f(v) = \sum_i w_i f(v_i)$, where $v_i$ ranges over the parents of $v$, and $w_i = \text{weight of the edge from } v_i \text{ to } v$.

Similarly for $b(v) = \sum_p \text{beginning at } v \text{ weight}(p)$

The paths $\text{beginning} \text{ at } v$ are the ones $\text{ending} \text{ at } v$ in the reverse (or inverted) graph.
\( f(v)b(\nu) = \) sum of the path weights of all paths \textit{through} \( v \).

\( f(\nu')wb(v) = \) sum of the path weights of all paths \textit{through the edge} \((\nu',v)\)
• Numerical issues: multiplying many small values can cause underflow. Remedies:

  – *Scale* weights to be close to 1 (affects all paths by same constant factor – which can be multiplied back later); or

  – (where possible) use *log weights*, so can add instead of multiplying.

  – see Rabiner & Tobias Mann links on web page

    • & will discuss further in discussion section