CSC2515 – Machine Learning Sam Roweis

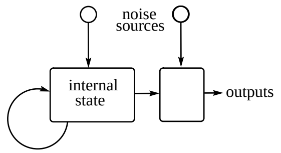
Lecture 10:

Markov and Hidden Markov Models

November 14, 2006

Probabilistic Models for Time Series

- Generative models for time-series:
  To get interesting variability need *noise*.
  To get correlations across time, need some system *state*.



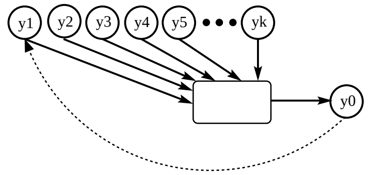
- Time: discrete
  States: discrete or continuous
  Outputs: discrete or continuous
- Today: discrete state
  similar to finite state automata; Moore/Mealy machines

Markov Models

- Use past as state. Next output depends on previous output(s):

$$\mathbf{y}_t = f[\mathbf{y}_{t-1}, \mathbf{y}_{t-2}, \ldots]$$

  order is number of previous outputs



- Add noise to make the system probabilistic:

$$p(\mathbf{y}_t | \mathbf{y}_{t-1}, \mathbf{y}_{t-2}, \ldots, \mathbf{y}_{t-k})$$

- Markov models have two problems:
  – need big order to remember past "events"
  – output noise is confounded with state noise

Learning Markov Models

- The ML parameter estimates for a simple Markov model are easy:

$$p(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_T) = \prod_{t=k+1}^{T} p(\mathbf{y}_t | \mathbf{y}_{t-1}, \mathbf{y}_{t-2}, \ldots, \mathbf{y}_{t-k})$$

$$\log p(\{\mathbf{y}\}) = \sum_{t=k+1}^{T} \log p(\mathbf{y}_t | \mathbf{y}_{t-1}, \mathbf{y}_{t-2}, \ldots, \mathbf{y}_{t-k})$$

- Each window of $k+1$ outputs is a training case for the model $p(\mathbf{y}_t | \mathbf{y}_{t-1}, \mathbf{y}_{t-2}, \ldots, \mathbf{y}_{t-k})$.
- Example: for discrete outputs (symbols) and a 2nd-order markov model we can use the multinomial model:

$$p(y_t = m | y_{t-1} = a, y_{t-2} = b) = \alpha_{mab}$$

  The maximum likelihood values for α are:

$$\alpha^*_{mab} = \frac{\mathrm{num}[t \; s.t. \; y_t = m, y_{t-1} = a, y_{t-2} = b]}{\mathrm{num}[t \; s.t. \; y_{t-1} = a, y_{t-2} = b]}$$

Aggregate Markov Models

- A first order Markov Model $p(y_2|y_1)$ is also called a *bigram* model.
- If there are a huge number N of possible symbols (e.g. words in English) we might need an enormously long sequence to estimate even such a simple model. But a *unigram* (0-order) is too simple...
- There is a very clever way to regularlize this model, which is to constrain the transition matrix $p_{ij} = p(y_2 = j|y_1 = i)$ to be *low rank*, e.g. rank at most K where $K \ll N$.
- This has an interpretation as a conditional latent variable (mixture) model with K "topics":

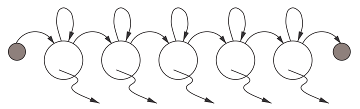
$$p_{ij} = p(y_2 = j|y_1 = i) = \sum_k p(y_2 = j|z = k)p(z = k|y_1 = i)$$

- The model can be trained very simply using EM and gives very good predictions even with only a modest amount of training data.

Hidden Markov Models (HMMs)

Add a latent (hidden) variable x_t to improve the model.

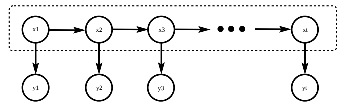
- HMM $\equiv$ " probabilistic function of a Markov chain":
  1. 1st-order Markov chain generates hidden state sequence (path):
  $$\mathsf{P}(x_{t+1} = j|x_t = i) = S_{ij} \qquad \mathsf{P}(x_1 = j) = \pi_j$$
  2. A set of output probability distributions $\mathbf{A}_j(\cdot)$ (one per state) converts state path into sequence of observable symbols/vectors
  $$\mathsf{P}(\mathbf{y}_t = y|x_t = j) = \mathbf{A}_j(\mathbf{y})$$


(state transition diagram)

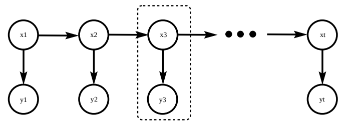
- Even though hidden state seq. is 1st-order Markov, the output process is not Markov of *any* order
[ex. 1111121111311121111131...]

Links to Other Models

- You can think of an HMM as:
A Markov chain with stochastic measurements.



or
A mixture model with states coupled across time.

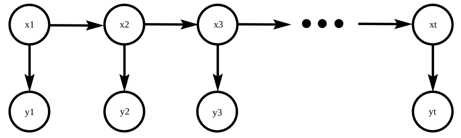


- The future is independent of the past given the present.
However, conditioning on all the observations couples hidden states.

Applications of HMMs

- Speech recognition.
- Language modeling.
- Information retrieval.
- Motion video analysis/tracking.
- Protein sequence and genetic sequence alignment and analysis.
- Financial time series prediction.
- ...

HMM Model Equations



- Hidden states $\{x_t\}$, outputs $\{\mathbf{y}_t\}$
  Joint probability factorizes:

$$\mathsf{P}(\{x\},\{\mathbf{y}\}) = \prod_{t=1}^{T} \mathsf{P}(x_t|x_{t-1})\mathsf{P}(\mathbf{y}_t|x_t)$$
$$= \pi_{x_1} \prod_{t=1}^{T-1} S_{x_t,x_{t+1}} \prod_{t=1}^{T} A_{x_t}(\mathbf{y}_t)$$

- NB: Data are *not* i.i.d. Everything is coupled across time.
- Three problems: computing probabilities of observed sequences, inference of hidden state sequences, learning of parameters.

Probability of an Observed Sequence

- To evaluate the probability $\mathsf{P}(\{\mathbf{y}\})$, we want:

$$\mathsf{P}(\{\mathbf{y}\}) = \sum_{\{x\}} \mathsf{P}(\{x\},\{\mathbf{y}\})$$
$$\mathsf{P}(\text{observed sequence}) = \sum_{\text{all paths}} \mathsf{P}(\text{ observed outputs , state path })$$

- Looks hard! (#paths $= N^T$). But joint probability factorizes:

$$\mathsf{P}(\{\mathbf{y}\}) = \sum_{x_1}\sum_{x_2}\cdots\sum_{x_T}\prod_{t=1}^{T} \mathsf{P}(x_t|x_{t-1})\mathsf{P}(\mathbf{y}_t|x_t)$$
$$= \sum_{x_1} \mathsf{P}(x_1)\mathsf{P}(\mathbf{y}_1|x_1) \sum_{x_2} \mathsf{P}(x_2|x_1)\mathsf{P}(\mathbf{y}_2|x_2) \cdots$$
$$\sum_{x_T} \mathsf{P}(x_T|x_{T-1})\mathsf{P}(\mathbf{y}_T|x_T)$$

- By moving the summations inside, we can save a lot of work.

The forward (α) recursion

- We want to compute:

$$L = \mathsf{P}(\{\mathbf{y}\}) = \sum_{\{x\}} \mathsf{P}(\{x\},\{\mathbf{y}\})$$

- There exists a clever "forward recursion" to compute this huge sum very efficiently. Define $\alpha_j(t)$:

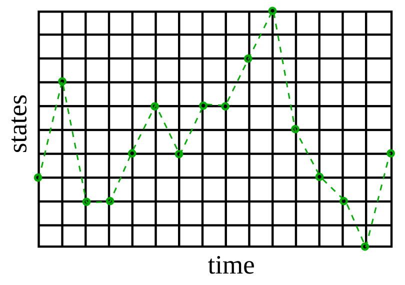
$$\alpha_j(t) = \mathsf{P}(\ \mathbf{y}_1^t\ ,\ x_t = j\ )$$
$$\alpha_j(1) = \pi_j \mathbf{A}_j(\mathbf{y}_1) \qquad \text{induction to the rescue...}$$
$$\alpha_k(t+1) = \{\sum_j \alpha_j(t) S_{jk}\} A_k(\mathbf{y}_{t+1})$$

- Notation: $x_a^b \equiv \{x_a, \ldots, x_b\}$; $\mathbf{y}_a^b \equiv \{\mathbf{y}_a, \ldots, \mathbf{y}_b\}$
- This enables us to easily (cheaply) compute the desired likelihood L since we know we must end in some possible state:

$$L = \sum_k \alpha_k(T)$$

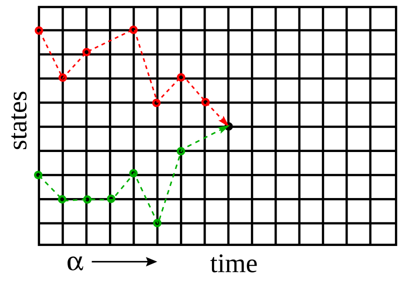
Bugs on a Grid

- Naive algorithm:
  1. start bug in each state at t=1 holding value 0
  2. move each bug forward in time by making copies of it and incrementing the value of each copy by the probability of the transition and output emission
  3. go to 2 until all bugs have reached time T
  4. sum up values on all bugs



Bugs on a Grid - Trick

- Clever recursion:
  adds a step between 2 and 3 above which says: at each node, replace all the bugs with a single bug carrying the sum of their values



- This is exactly dynamic programming.

where

$$\alpha_j(t) = \mathsf{P}(\, \mathbf{y}_1^t \,,\, x_t = j\,)$$
$$\beta_j(t) = p(\mathbf{y}_{t+1}^T \,|\, x_t = j\,)$$
$$\gamma_i(t) = p(x_t = i \,|\, \mathbf{y}_1^T)$$

Inference of Hidden States

- What if we we want to estimate the hidden states given observations? To start with, let us estimate a single hidden state:

$$p(x_t|\{\mathbf{y}\}) = \gamma(x_t) = \frac{p(\{\mathbf{y}\}|x_t)p(x_t)}{p(\{\mathbf{y}\})}$$
$$= \frac{p(\mathbf{y}_1^t|x_t)p(\mathbf{y}_{t+1}^T|x_t)p(x_t)}{p(\mathbf{y}_1^T)}$$
$$= \frac{p(\mathbf{y}_1^t, x_t)p(\mathbf{y}_{t+1}^T|x_t)}{p(\mathbf{y}_1^T)}$$
$$p(x_t|\{\mathbf{y}\}) = \gamma(x_t) = \frac{\alpha(x_t)\beta(x_t)}{p(\mathbf{y}_1^T)}$$

Forward-Backward Algorithm

- We compute these quantities efficiently using another recursion. Use total prob. of all paths going through state i at time t to compute the *conditional* prob. of being in state i at time t:

$$\gamma_i(t) = \mathsf{P}(x_t = i \,|\, \mathbf{y}_1^T)$$
$$= \alpha_i(t)\beta_i(t)/L$$

where we defined:

$$\beta_j(t) = \mathsf{P}(\mathbf{y}_{t+1}^T \,|\, x_t = j\,)$$

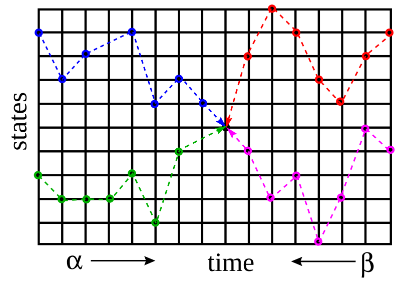
- There is also a simple recursion for $\beta_j(t)$:

$$\beta_j(t) = \sum_k S_{jk} A_k(\mathbf{y}_{t+1}) \beta_k(t+1)$$
$$\beta_j(T) = 1$$

- $\alpha_i(t)$ gives total *inflow* of prob. to node (t, i)
  $\beta_i(t)$ gives total *outflow* of prob.

Forward-Backward Algorithm

- $\alpha_i(t)$ gives total *inflow* of prob. to node (t, i)
  $\beta_i(t)$ gives total *outflow* of prob.



- Bugs again: we just let the bugs run forward from time 0 to t and backward from time T to t.
- In fact, we can just do one forward pass to compute all the $\alpha_i(t)$ and one backward pass to compute all the $\beta_i(t)$ and then compute any $\gamma_i(t)$ we want. Total cost is $O(M^2T)$.

Likelihood from Forward-Backward Algorithm

- Since $\sum_{x_t} \gamma(x_t) = 1$, we can compute the likelihood at *any* time using the results of the $\alpha - \beta$ recursions:

$$L = p(\{\mathbf{y}\}) = \sum_{x_t} \alpha(x_t)\beta(x_t)$$

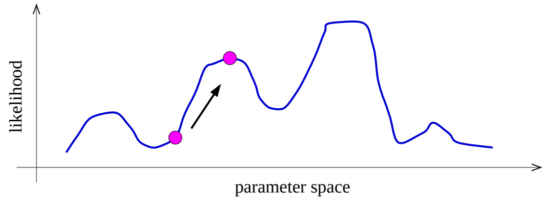
- In the forward calculation we proposed originally, we did this at the final timestep $t = T$:

$$L = \sum_{x_T} \alpha(x_T)$$

  because $\beta_T = 1$.
- This is a good way to check your code!

Baum-Welch (EM) Training

1. Intuition: if only we *knew* the true state path then ML parameter estimation would be trivial.
2. But: can *estimate* state path using the DP trick.
3. *Baum-Welch algorithm* (special case of EM): estimate the states, then compute params, then re-estimate states, and so on ...
4. This works and we can *prove* that it always improves likelihood.
5. However: finding the ML parameters is NP hard, so initial conditions matter a lot and convergence is hard to tell.



Parameter Estimation with EM

- Complete log likelihood:

$$\log p(x, y) = \log\{\pi_{x_1} \prod_{t=1}^{T-1} S_{x_t,x_{t+1}} \prod_{t=1}^{T} A_{x_t}(\mathbf{y}_t)\}$$

$$= \log\{\prod_i \pi_i^{[x_1^i]} \prod_{t=1}^{T-1} \prod_j S_{ij}^{[x_t^i, x_{t+1}^j]} \prod_{t=1}^{T} \prod_k A_k(\mathbf{y}_t)^{[x_t^k]}\}$$

$$= \sum_i [x_1^i] \log \pi_i + \sum_{t=1}^{T-1} \sum_j [x_t^i, x_{t+1}^j] \log S_{ij} + \sum_{t=1}^{T} \sum_k [x_t^k] \log A_k(\mathbf{y}_t)$$

  where the indicator $[x_t^i] = 1$ if $x_t = i$ and 0 otherwise
- Statistics we need from the E-step are: $p(x_t|\{\mathbf{y}\})$ and $p(x_t, x_{t+1}|\{\mathbf{y}\})$.
- We saw how to get single time marginals $p(x_t|\{\mathbf{y}\})$, but what about two-frame estimates $p(x_t, x_{t+1}|\{\mathbf{y}\})$?

Two-frame inference

- Need the cross-time statistics for adjacent time steps:

$$\xi_{ij} = p(x_t = i, x_{t+1} = j | \{\mathbf{y}\})$$

- This can be done by rewriting:

$$\begin{aligned} p(x_t, x_{t+1}|\{\mathbf{y}\}) &= p(x_t, x_{t+1}, \{\mathbf{y}\})/p(\{\mathbf{y}\}) \\ &= p(x_t, \mathbf{y}_1^t)p(x_{t+1}, \mathbf{y}_{t+1}^T | x_t, \mathbf{y}_1^t)/L \\ &= p(x_t, \mathbf{y}_1^t)p(x_{t+1}|x_t)p(\mathbf{y}_{t+1}|x_{t+1})p(y_{t+2}^T|x_{t+1})/L \\ &= \alpha_i(t)S_{ij}\mathbf{A}_j(\mathbf{y}_{t+1})\beta_j(t+1)/L \\ &= \xi_{ij} \end{aligned}$$

- This is the expected number of transitions from state i to state j that begin at time t, given the observations.
- It can be computed with the same α and β recursions.

New Parameters are just Ratios of Frequency Counts

- Initial state distribution: expected #times in state i at time 1:

$$\hat{\pi}_i = \gamma_i(1)$$

- Expected #transitions from state i to j which begin at time t:

$$\xi_{ij}(t) = \alpha_i(t)S_{ij}\mathbf{A}_j(\mathbf{y}_{t+1})\beta_j(t+1)/L$$

so the estimated transition probabilities are:

$$\hat{S}_{ij} = \sum_{t=1}^{T-1} \xi_{ij}(t) \Bigg/ \sum_{t=1}^{T-1} \gamma_i(t)$$

- The output distributions are the expected number of times we observe a particular symbol in a particular state:

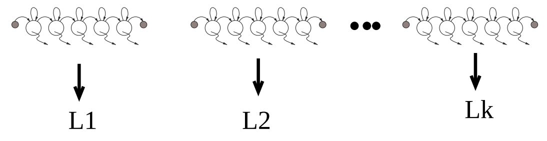
$$\hat{A}_j(y) = \sum_{t|\mathbf{y}_t=y} \gamma_j(t) \Bigg/ \sum_{t=1}^{T} \gamma_j(t)$$

Viterbi Decoding

- The numbers $\gamma_j(t)$ above gave the probability distribution over all states at any time.
- By choosing the state $\gamma_*(t)$ with the largest probability at each time, we can make an "average" state path. This is the path with the *maximum expected number of correct states*.
- But it *is not* the single path with the highest likelihood of generating the data. In fact it may be a path of probability zero!
- To find the single best path, we do *Viterbi decoding* which is just Bellman's dynamic programming algorithm applied to this problem.
- The recursions look the same, except with $\max$ instead of $\sum$.
- Bugs once more: same trick except at each step kill all bugs but the one with the highest value at the node.
- There is also a modified Baum-Welch training based on the Viterbi decode. Like K-means instead of mixtures of Gaussians.

Using HMMs for Recognition

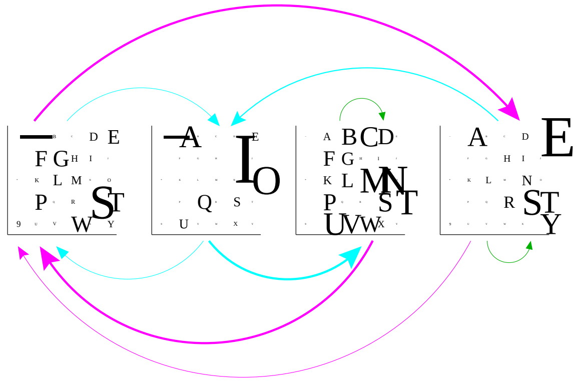
- Use many HMMs for recognition by:
  1. training one HMM for each class (requires *labelled* training data)
  2. evaluating probability of an unknown sequence under each HMM
  3. classifying unknown sequence: HMM with highest likelihood



- This requires the solution of two problems:
  1. Given model, evaluate prob. of a sequence.
     (We can do this exactly & efficiently.)
  2. Give some training sequences, estimate model parameters.
     (We can find the local maximum of parameter space nearest our starting point.)

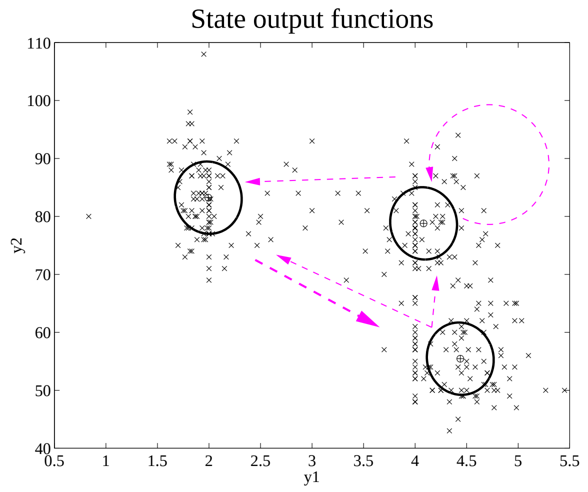
Symbol HMM Example

- Character sequences (discrete outputs)



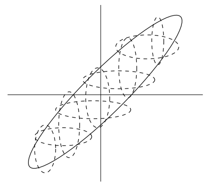
Mixture HMM Example

- Geyser data (continuous outputs)



Regularizing HMMs

- Two problems:
  - for high dimensional outputs, lots of parameters in each $\mathbf{A}_j(\mathbf{y})$
  - with many states, transition matrix has many^2 elements
- First problem: full covariance matrices in high dimensions or discrete symbol models with many symbols have *lots* of parameters. To estimate these accurately requires a lot of training data. Instead, we often use mixtures of diagonal covariance Gaussians.

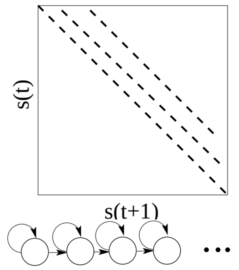


- For discrete data, we can use mixtures of base rates.
- We can also tie parameters across states.

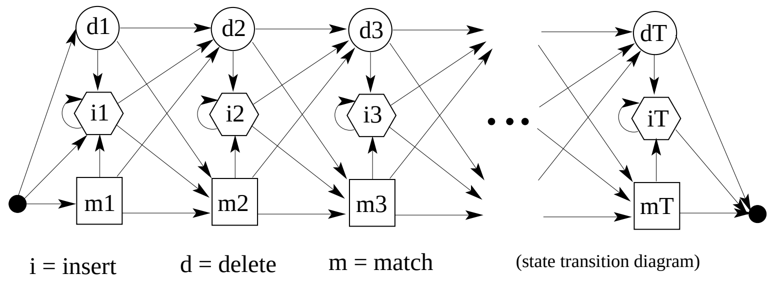
Regularizing Transition Matrices

- One way to regularize large transition matrices is to *constrain* them to be relatively *sparse*: instead of being allowed to transition to *any* other state, each state has only a few possible successor states.
- For example if each state has at most p possible next states then the cost of inference is $O(pKT)$ and the number of parameters is $O(pK + KM)$ which are both *linear* in the number of states K.

An extremely effective way to constrain the transitions is to *order* the states in the HMM and allow transitions only to *states that come*
- *later in the ordering*. Such models are known as "linear HMMs", "chain HMMs" or "left-to-right HMMs". Transition matrix is upper-diagonal (usually only has a few bands).



Profile (String-Edit) HMMs



- A "profile HMM" or "string-edit" HMM is used for probabilistically matching an observed input string to a stored template pattern with possible insertions and deletions.
- Three kinds of states: match, insert, delete.
  m_n – use position n in the template to match an observed symbol
  i_n – insert extra symbol(s) observations after template position n
  d_n – delete (skip) template position n

HMM Practicalities

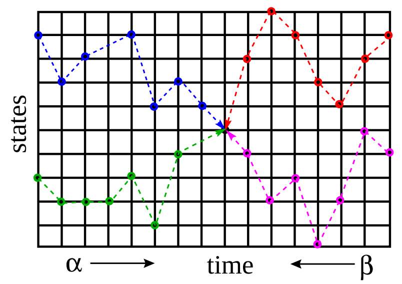
- If you just implement things as I have described them, *they will not work at all*. Why? Remember `logsum`...
- Numerical scaling: the probability values that the bugs carry get tiny for big times and so can easily underflow. Good rescaling trick:

$$\rho_t = \mathsf{P}(\mathbf{y}_t|\mathbf{y}_1^{t-1}) \qquad \alpha(t) = \tilde{\alpha}(t)\prod_{t'=1}^{t}\rho_{t'}$$

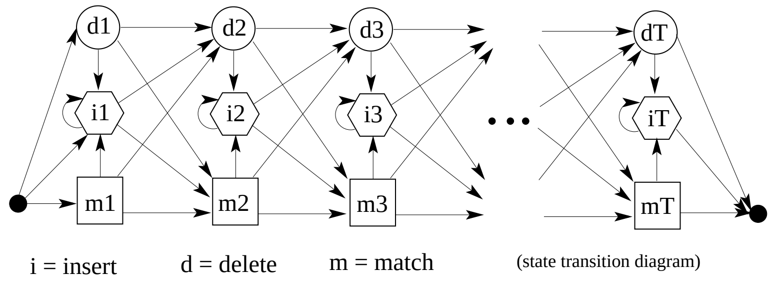
  (of course you could always use `logsum` but that's less efficient)
- Multiple observation sequences: can be dealt with by averaging numerators and averaging denominators in the ratios given above.
- Initialization: mixtures of base rates or mixtures of Gaussians
- Generation of new sequences. Just roll the dice!
- Sampling a single state sequence from the posterior $p(\{x\}|\{\mathbf{y}\})$. Harder...but possible. (can you think of how?)

Computational Costs in HMMs

- The number of parameters in the model was $O(K^2 + KM)$ for K states and M output symbols (or dimensions).
- Recall the forward-backward algorithm for inference of state probabilities $p(x_t|\{\mathbf{y}\})$.
- The storage cost of this procedure was $O(KT + K^2)$ for K states and a sequence of length T.
- The time complexity was $O(K^2T)$.



Profile HMMs have Linear Costs



- number of states $=$ 3(length_template)
- Only insert and match states can generate output symbols.
- Once you visit or skip a match state you can never return to it.
- At most 3 destination states from any state, so S_{ij} very sparse.
- Storage/Time cost *linear* in #states, not quadratic.
- State variables and observations no longer in sync.
  (e.g. y1:m1 ; d2 ; y2:i2 ; y3:i2 ; y4:m3 ; ...)

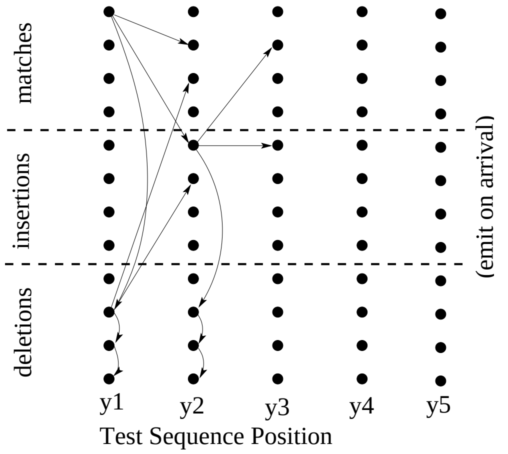
Some HMM History

- Markov ('13) and later Shannon ('48,'51) studied *Markov chains*.
- Baum et. al (BP'66, BE'67, BS'68, BPSW'70, B'72) developed much of the theory of "probabilistic functions of Markov chains".
- Viterbi ('67) (now Qualcomm) came up with an efficient optimal decoder for state inference.
- Applications to speech were pioneered independently by:
  - Baker ('75) at CMU (now Dragon)
  - Jelinek's group ('75) at IBM (now Hopkins)
  - communications research division of IDA (Ferguson '74 unpublished)
- Dempster, Laird & Rubin ('77) recognized a general form of the Baum-Welch algorithm and called it the *EM* algorithm.
- A landmark open symposium in Princeton ('80) hosted by IDA reviewed work till then.

DP for Profile HMMs

- How do we fill in the numbers for a DP grid using a string-edit HMM?
- Almost the same as normal except:
  - Now the grid is 3 times its normal height.
  - It is possible to move down without moving right if you move into a deletion state.

eg: template length=4, test sequence length=5

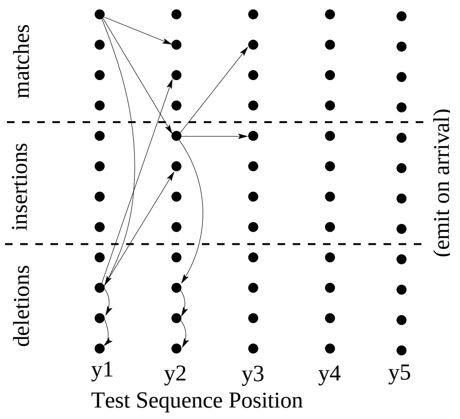


Profile HMM DP Grid

$C_{x \to x'} = -\log S_{x,x'} - \log A_{x'}(\mathbf{y}_t)$ if x' is match or insert

$C_{x \to x'} = -\log S_{x,x'}$ if x' is a delete state

State $x \in \{m_n, i_n, d_n\}$ has nonzero transition probabilities only to states $x' \in \{m_{n+1}, i_n, d_{n+1}\}$.



Forward-Backward for Profile HMMs

- The equations for the delete states in profile HMMs need to be modified slightly, since they don't emit any symbols.
- For delete states k, the forward equations become:
$$\alpha_k(t) = \sum_j \alpha_j(t) S_{jk}$$
which should be evaluated after the insert and match state updates.
- For all states, the backward equations become:
$$\beta_k(t) = \sum_{i \in \text{match,ins}} S_{ki} \beta_i(t+1) A_i(\mathbf{y}_{t+1}) + \sum_{j \in \text{del}} S_{kj} \beta_j(t)$$
which should be evaluated first for delete states k; then for the rest.
- The gamma equations remain the same:
$$\gamma_i(t) = p(x_t = i \mid \mathbf{y}_1^T) = \alpha_i(t)\beta_i(t)/L$$
- Notice that each summation above contains only three terms, regardless of the number of states!

Initializing Forward-Backward for Profile HMMs

- The initialization equations for Profile HMMs also need to be fixed up, to reflect the fact that the model can only begin in states m_1, i_1, d_1 and can only finish in states m_N, i_N, d_N.
- In particular, $\pi_j = 0$ if j is not one of m_1, i_1, d_1.
- When initializing $\alpha_k(1)$, delete states k have zeros, and all other states have the product of the transition probabilities through only delete states up to them, plus the final emission probability.
- When initializing $\beta_k(T)$, the same kind of adjustment must be made.

HMM Pseudocode

- Forward-backward including scaling tricks

```
$q_j(t) = \mathbf{A}_j(\mathbf{y}_t)$

$\alpha(1) = \pi .* q(1)$                $\rho(1) = \sum \alpha(1)$    $\alpha(1) = \alpha(1)/\rho(1)$
$\alpha(t) = (S' * \alpha(t-1)) .* q(t)$    $\rho(t) = \sum \alpha(t)$    $\alpha(t) = \alpha(t)/\rho(t)$    $[t = 2 : T]$

$\beta(T) = 1$
$\beta(t) = S * (\beta(t+1) .* q(t+1)/\rho(t+1)$                 $[t = (T-1) : 1]$

  $\xi = 0$
  $\xi = \xi + S .* (\alpha(t) * (\beta(t+1) .* q(t+1))')/\rho(t+1)$    $[t = 1 : (T-1)]$

  $\gamma = (\alpha .* \beta)$

    $\log \mathsf{P}(\mathbf{y}_1^T) = \sum \log(\rho(t))$
```

HMM Pseudocode

- Baum-Welch parameter updates

```
$\delta_j = 0$    $\hat{S}_{ij} = 0$    $\hat{\pi} = 0$    $\hat{A} = 0$

  for each sequence, run forward backward to get $\gamma$ and $\xi$ , then
  $\hat{S} = \hat{S} + \xi$    $\hat{\pi} = \hat{\pi} + \gamma(1)$    $\delta = \delta + \sum_t \gamma(t)$
  $\hat{A}_j(\mathbf{y}) = \sum_{t|\mathbf{y}_t = y} \gamma_j(t)$    or    $\hat{A} = \hat{A} + \sum_t \mathbf{y}_t \gamma(t)$

$\hat{S}_{ij} = \hat{S}_{ij} / \sum_k \hat{S}_{ik}$    $\hat{\pi} = \hat{\pi} / \sum \hat{\pi}$    $\hat{A}_j = \hat{A}_j / \delta_j$
```