Hidden Markov models

Bayesian Methodology in Biostatistics (BST 249)

Jeffrey W. Miller

Department of Biostatistics
Harvard T.H. Chan School of Public Health

Outline

Outline

Introduction

- Hidden Markov models (HMMs) are a surprisingly powerful tool for modeling a wide range of sequential data, including:
  - speech
  - written text
  - genomic data
  - weather patterns
  - financial data
  - animal behaviors

- Dynamic programming enables tractable inference in HMMs, including:
  - finding the most probable sequence of hidden states using the Viterbi algorithm,
  - probabilistic inference using the forward-backward algorithm,
  - parameter estimation using the Baum–Welch algorithm.

Refresher on Markov chains

- Recall that $(Z_1, \ldots, Z_n)$ is a Markov chain if

$$Z_{t+1} \perp\!\!\!\perp (Z_1, \ldots, Z_{t-1}) \mid Z_t$$

  for each t.

- In other words, "the future is conditionally independent of the past given the present."

- This is equivalent to saying that the distribution respects the following directed graph:

$$\begin{tikzcd} Z_1 \arrow[r] & Z_2 \arrow[r] & Z_3 \arrow[r] & \cdots \arrow[r] & Z_n \end{tikzcd}$$

Refresher on Markov chains

- A Markov chain is a natural model to use for sequential data when the present state Z_t contains all of the information about the future that could be gleaned from $Z_1, \ldots, Z_t$.
- In other words, when Z_t is the "complete state" of the system.
- Oftentimes, however, we only get to observe an incomplete or noisy version of the state. In such cases, it is more natural to use a hidden Markov model.

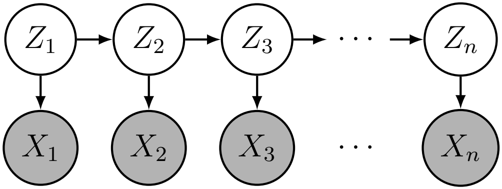
Outline

Hidden Markov models: Graphical model

- A hidden Markov model is a distribution

$$p(x_1, \ldots, x_n, z_1, \ldots, z_n)$$

that respects the following directed graph:

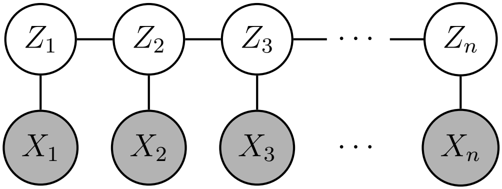


- In other words, it factors as

$$p(x_{1:n}, z_{1:n}) = p(z_1)p(x_1|z_1)\prod_{t=2}^{n} p(z_t|z_{t-1})p(x_t|z_t).$$

Hidden Markov models: Graphical model

- It turns out that in this case, it is equivalent to say that the distribution respects the following undirected graph:



- $Z_1, \dots, Z_n$ represent the "hidden states", and $X_1, \dots, X_n$ represent the observations.

- Assume that $Z_1, \dots, Z_n$ are discrete random variables taking finitely many possible values. For simplicity, let's denote these possible values by $1, \dots, m$. In other words, $Z_t \in \{1, \dots, m\}$.

Hidden Markov models: Transition matrix

- Assume the "transition probabilities"

$$T_{ij} = \mathbb{P}(Z_{t+1} = j \mid Z_t = i)$$

  do not depend on the time index t. This assumption is referred to as "time-homogeneity."

- The $m \times m$ matrix T in which entry (i, j) is T_{ij} is referred to as the "transition matrix."

- Note that every row of T must sum to 1. A nonnegative square matrix with this property is referred to as a "stochastic matrix".

Hidden Markov models: Emission distn, Initial distn

- Assume that the "emission distributions"

$$\varepsilon_i(x_t) = p(x_t \mid Z_t = i)$$

  do not depend on the time index t.

- While we assume the Z's are discrete, the X's may be either discrete or continuous, and may also be multivariate.

- The "initial distribution" π is the distribution of Z_1, that is, $\pi_i = \mathbb{P}(Z_1 = i)$.

Hidden Markov models: Example parameters

- Number of hidden states: $m = 2$, that is, $Z_t \in \{1, 2\}$
- Initial distribution: $\pi = (0.5, 0.5)$
- Transition matrix:
$$T = \begin{bmatrix} .9 & .1 \\ .2 & .8 \end{bmatrix}$$
- Emission distributions:
$$X_t \mid Z_t = i \ \sim \mathcal{N}(\mu_i, \sigma_i^2)$$
where $\mu = (-1, 1)$ and $\sigma = (1, 1)$.

Outline

Overview of HMM algorithms

- There are three main algorithms used for computation in HMMs:
  - the Viterbi algorithm,
  - the forward-backward algorithm, and
  - the Baum–Welch algorithm.
- These algorithms employ dynamic programming, which enables otherwise intractable calculations to be done efficiently.

Overview: Viterbi algorithm

- In the Viterbi algorithm and the forward-backward algorithm, it is assumed that all of the parameters are known.

- In other words, the initial distribution π, transition matrix T, and emission distributions ε_i are all known.

- The Viterbi algorithm is an efficient method of finding a sequence $z_1^*, \ldots, z_n^*$ with maximal probability given $x_1, \ldots, x_n$, that is, finding

$$z_{1:n}^* \in \operatorname*{argmax}_{z_{1:n}} p(z_{1:n} | x_{1:n}).$$

- Naively maximizing over all sequences would take order nm^n time, whereas the Viterbi algorithm only takes nm^2 time.

Overview: Forward-backward algorithm

- The forward-backward algorithm enables one to efficiently compute many quantities given $x_{1:n}$, for example,
  - $\mathbb{P}(Z_t = i \mid x_{1:n})$ for each i and each t,
  - $\mathbb{P}(Z_t = i, Z_{t+1} = j \mid x_{1:n})$ for each i, j and each t,
  - $\mathbb{P}(Z_t \neq Z_{t+1} \mid x_{1:n})$ for each t,
  - etc.
- It also allows you to efficiently sample from $p(z_{1:n}|x_{1:n})$.

Overview: Baum–Welch algorithm

- The Baum–Welch algorithm is a method of estimating the parameters of an HMM.
- Specifically, Baum–Welch enables estimation of the initial distribution, transition matrix, and emission distributions.
- Baum–Welch uses expectation-maximization and the forward-backward algorithm.

Overview: Historical fun facts

- The term “dynamic programming” was coined by Richard Bellman in the 1940s, to describe his research on certain optimization problems that can be efficiently solved with recursions.
- In this context, “programming” means optimization.
- As I understand it, this terminology comes from the 1940s during which there was a lot of work on how to optimize military plans or “programs”, in the field of operations research.
- So, what is “dynamic” about it? There's a funny story on Wikipedia about why he called it “dynamic” programming:

  `https://en.wikipedia.org/wiki/Dynamic_programming#History`

Outline

Viterbi algorithm: Preliminaries

- Before we start, note the following facts.
- If $c \geq 0$ and $f(x) \geq 0$, then

$$\max_x cf(x) = c \max_x f(x)$$
$$\operatorname*{argmax}_x cf(x) = \operatorname*{argmax}_x f(x).$$

- Also note that

$$\max_{x,y} f(x,y) = \max_x \max_y f(x,y).$$

Viterbi algorithm: Preliminaries

- The goal of the Viterbi algorithm is to find

$$z^*_{1:n} \in \operatorname*{argmax}_{z_{1:n}} p(z_{1:n}|x_{1:n}).$$

- Since $p(x_{1:n})$ is constant with respect to $z_{1:n}$, this is equivalent to

$$z^*_{1:n} \in \operatorname*{argmax}_{z_{1:n}} p(x_{1:n}, z_{1:n}).$$

- Naively, this would take order nm^n time, since there are m^n sequences $z_{1:n}$ and computing $p(x_{1:n}, z_{1:n})$ takes order n time.

- The Viterbi algorithm provides a much faster way.

Viterbi algorithm: Computing the max (1/3)

- Before trying to find the argmax, let's think about the max:

$$M = \max_{z_{1:n}} p(x_{1:n}, z_{1:n}).$$

- Throughout the following derivation, we will assume $x_{1:n}$ is fixed, and will suppress it from the notation for clarity.

- Recall the the assumed factorization of $p(x_{1:n}, z_{1:n})$,

$$p(x_{1:n}, z_{1:n}) = p(z_1)p(x_1|z_1)p(z_2|z_1)p(x_2|z_2)\prod_{t=3}^{n} p(z_t|z_{t-1})p(x_t|z_t).$$

Viterbi algorithm: Computing the max (2/3)

$$
\begin{aligned}
M &= \max_{z_{1:n}} \underbrace{p(z_1)p(x_1|z_1)}_{\text{call this } \mu_1(z_1)} p(z_2|z_1)p(x_2|z_2) \prod_{t=3}^{n} p(z_t|z_{t-1})p(x_t|z_t) \\
&= \max_{z_{2:n}} \underbrace{\left(\max_{z_1} \mu_1(z_1)p(z_2|z_1)p(x_2|z_2) \right)}_{\text{call this } \mu_2(z_2)} \prod_{t=3}^{n} p(z_t|z_{t-1})p(x_t|z_t) \\
&= \max_{z_{3:n}} \underbrace{\left(\max_{z_2} \mu_2(z_2)p(z_3|z_2)p(x_3|z_3) \right)}_{\text{call this } \mu_3(z_3)} \prod_{t=4}^{n} p(z_t|z_{t-1})p(x_t|z_t) \\
&\vdots \\
&= \max_{z_{j:n}} \underbrace{\left(\max_{z_{j-1}} \mu_{j-1}(z_{j-1})p(z_j|z_{j-1})p(x_j|z_j) \right)}_{\text{call this } \mu_j(z_j)} \prod_{t=j+1}^{n} p(z_t|z_{t-1})p(x_t|z_t)
\end{aligned}
$$

Viterbi algorithm: Computing the max (3/3)

- Continuing in this way, we end up with

$$M = \max_{z_n} \mu_n(z_n).$$

- Therefore, we can compute M via the following algorithm:

  1. For each $z_1 = 1, \ldots, m$, compute $\mu_1(z_1) = p(z_1)p(x_1|z_1)$.
  2. For each $j = 2, \ldots, n$, for each $z_j = 1, \ldots, m$, compute
  $$\mu_j(z_j) = \max_{z_{j-1}} \mu_{j-1}(z_{j-1})p(z_j|z_{j-1})p(x_j|z_j).$$
  3. Compute $M = \max_{z_n} \mu_n(z_n)$.

Viterbi algorithm: Computation time

- How much time does this take, as a function of m and n?
- Step 1 takes order m time.
- In step 2, for each j and each z_j, it takes order m time to compute $\mu_j(z_j)$.
- So, overall, step 2 takes nm^2 time.
- Step 3 takes order m time.
- Thus, altogether, the computation takes order nm^2 time.

Viterbi algorithm: Computing the argmax

- Okay, so now we know how to compute the max, M.
- But who cares about the max? What we really want is the argmax!
- More precisely, we want to find a sequence $z^*_{1:n}$ maximizing $p(x_{1:n}, z_{1:n})$.
- It turns out that in the algorithm above, we've basically already done all the work required to find such a $z^*_{1:n}$.

Viterbi algorithm: Computing the argmax (1/3)

- Augment step 2 in the algorithm above by also recording a value of z_{j-1} attaining the maximum in the definition of $\mu_j(z_j)$; denote this value by $\alpha_j(z_j)$.

- In other words, in addition to computing $\mu_j(z_j)$, we are going to define $\alpha_j(z_j)$ to be any value such that

$$\alpha_j(z_j) \in \operatorname*{argmax}_{z_{j-1}} \mu_{j-1}(z_{j-1}) p(z_j | z_{j-1}) p(x_j | z_j).$$

- Note that this doesn't really require any additional computation—we already have to loop over z_{j-1} to compute $\mu_j(z_j)$, so to get $\alpha_j(z_j)$ we just need to record one of the maximizing values of z_{j-1}.

Viterbi algorithm: Computing the argmax (2/3)

- Now, choose any z_n^* such that $\mu_n(z_n^*) = \max_{z_n} \mu_n(z_n)$, and for $j = n, n-1, \ldots, 2$ successively, let $z_{j-1}^* = \alpha_j(z_j^*)$.
- That gives us a sequence $z_{1:n}^*$, but how do we know that this sequence attains the maximum? Note that $\mu_n(z_n^*) = M$ and for each $j = n, n-1, \ldots, 2$,

$$\begin{aligned}\mu_j(z_j^*) &= \max_{z_{j-1}} \mu_{j-1}(z_{j-1}) p(z_j^* | z_{j-1}) p(x_j | z_j^*) \\ &= \mu_{j-1}(z_{j-1}^*) p(z_j^* | z_{j-1}^*) p(x_j | z_j^*).\end{aligned}$$

Viterbi algorithm: Computing the argmax (3/3)

- Therefore, plugging in this expression for $\mu_j(z_j^*)$ repeatedly,

$$\begin{aligned}
M &= \mu_n(z_n^*) \\
&= \mu_{n-1}(z_{n-1}^*)p(z_n^*|z_{n-1}^*)p(x_n|z_n^*) \\
&= \mu_{n-2}(z_{n-2}^*)p(z_{n-1}^*|z_{n-2}^*)p(x_{n-1}|z_{n-1}^*)p(z_n^*|z_{n-1}^*)p(x_n|z_n^*) \\
&\vdots \\
&= \mu_j(z_j^*)\prod_{t=j+1}^{n} p(z_t^*|z_{t-1}^*)p(x_t|z_t^*) \\
&\vdots \\
&= p(z_1^*)p(x_1|z_1^*)\prod_{t=2}^{n} p(z_t^*|z_{t-1}^*)p(x_t|z_t^*) \\
&= p(x_{1:n}, z_{1:n}^*).
\end{aligned}$$

So, $z_{1:n}^*$ is indeed a maximizer.

Fixing numerical underflow/overflow by using logs

- In theory, this provides an algorithm for computing $z^*_{1:n}$.
- However, in practice, the algorithm above will fail due to numerical underflow/overflow.
- The problem is that we are multiplying together a large number of probabilities, leading to very very small numbers that the computer will just round off to zero in most programming languages.
- It is also possible for overflow to occur if the x's are continuous since densities can be larger than 1.
- The standard solution to this problem is to work with logs. This works in a lot of other problems as well.

Fixing numerical underflow/overflow by using logs

- Denote $\ell = \log p$, for instance,

$$\begin{aligned}\ell(z_1) &= \log p(z_1)\\ \ell(z_t|z_{t-1}) &= \log p(z_t|z_{t-1})\\ \ell(x_t|z_t) &= \log p(x_t|z_t).\end{aligned}$$

- The algorithm above works if we use $f_j(z_j)$ in place of $\mu_j(z_j)$, where

$$\begin{aligned}f_1(z_1) &= \ell(z_1) + \ell(x_1|z_1)\\ f_j(z_j) &= \max_{z_{j-1}} \Big(f_{j-1}(z_{j-1}) + \ell(z_j|z_{j-1}) + \ell(x_j|z_j)\Big),\end{aligned}$$

and

$$\alpha_j(z_j) \in \operatorname*{argmax}_{z_{j-1}} \Big(f_{j-1}(z_{j-1}) + \ell(z_j|z_{j-1}) + \ell(x_j|z_j)\Big).$$

- This implies that $f_j(z_j) = \log \mu_j(z_j)$, and thus, choosing $\alpha_j(z_j)$ in this way is equivalent to the earlier definition.

Individual activity: Check your understanding

Answer these questions individually (5 minutes):
`https://forms.gle/kcoB5bQ6TMBZJhLr5`

Outline

Forward-backward algorithm: Preliminary remarks

- In the forward-backward algorithm, it is assumed that the initial distribution π, the transition matrix T, and the emission distributions ε_i, are known.
- The structure of the algorithm is very similar to the first part of the Viterbi algorithm, except that it involves sums instead of maxs.
- Despite the somewhat complicated derivation, the algorithm is actually quite simple.
- The details of the algorithm are not important—what is important is to understand how the algorithm is derived.

Forward-backward algorithm: Preliminary remarks

- So, how is the algorithm derived? To me, the simplest way to think about it is to ask: How can we efficiently compute the normalization constant?

- In this case, since $p(z_{1:n}|x_{1:n}) = p(x_{1:n}, z_{1:n})/p(x_{1:n})$, the normalization constant is $p(x_{1:n})$.

- The key is to look at the expression for the normalization constant, and try to find recursive formulas that would enable you to compute it efficiently.

- Typically this involves summing over variables sequentially.

- For some reason, it seems that for a wide range of inferential problems, once you know how to efficiently compute the normalization constant, you have "cracked" the problem, and can compute pretty much anything you want.

Forward-backward algorithm: Overview

- The forward-backward algorithm consists of two parts:
  1. In the forward algorithm, we sum over $z_1, z_2, \ldots, z_n$, in that order, to compute $p(x_{1:j}, z_j)$ for each $z_j = 1, \ldots, m$ and each $j = 1, \ldots, n$.
  2. In the backward algorithm, we sum over $z_n, z_{n-1}, \ldots, z_1$, in that order, to compute $p(x_{j+1:n}|z_j)$ for each $z_j = 1, \ldots, m$ and each $j = 1, \ldots, n$.
- There are multiple ways of defining the forward and backward algorithms, all of which are essentially equivalent. So, the details may vary from source to source.
- The forward and backward algorithms each take order nm^2 time.

Forward-backward algorithm: Overview

- Once we have our hands on $p(x_{1:j}, z_j)$ and $p(x_{j+1:n}|z_j)$ for each z_j and each j, we can compute lots of stuff, such as

$$p(z_j|x_{1:n}) \propto p(x_{1:n}, z_j) = p(x_{1:j}, z_j)p(x_{j+1:n}|z_j)$$

  and

$$\begin{aligned} p(z_j, z_{j+1}|x_{1:n}) &\propto p(x_{1:n}, z_j, z_{j+1}) \\ &= p(x_{1:j}, z_j)p(z_{j+1}|z_j)p(x_{j+1}|z_{j+1})p(x_{j+2:n}|z_{j+1}), \end{aligned}$$

  which are used in the Baum–Welch algorithm.

- These can also be used to sample from $p(z_{1:n}|x_{1:n})$, by first sampling from $p(z_1|x_{1:n})$, then from $p(z_{j+1}|z_j, x_{1:n})$ for each $j = 1, \dots, n-1$.

- Note that $p(z_{j+1}|z_j, x_{1:n})$ can be easily computed from $p(z_j, z_{j+1}|x_{1:n})$.

Forward algorithm (1/3)

- To derive the forward algorithm, we will write out the expression for $p(x_{1:n})$, rewrite it in terms of a sequence of sums over $z_1, \ldots, z_n$, and identify certain recursive formulas.

- Recall that the joint distribution factors as

$$p(x_{1:n}, z_{1:n}) = p(z_1)p(x_1|z_1)p(z_2|z_1)p(x_2|z_2)\prod_{t=3}^{n} p(z_t|z_{t-1})p(x_t|z_t)$$

and

$$p(x_{1:n} = \sum_{z_{1:n}} p(x_{1:n}, z_{1:n}).$$

Forward algorithm (2/3)

$$p(x_{1:n}) = \sum_{z_{1:n}} \underbrace{p(z_1)p(x_1|z_1)}_{\text{call this } s_1(z_1)} p(z_2|z_1)p(x_2|z_2) \prod_{t=3}^{n} p(z_t|z_{t-1})p(x_t|z_t)$$

$$= \sum_{z_{2:n}} \underbrace{\left(\sum_{z_1} s_1(z_1)p(z_2|z_1)p(x_2|z_2) \right)}_{\text{call this } s_2(z_2)} \prod_{t=3}^{n} p(z_t|z_{t-1})p(x_t|z_t)$$

$$= \sum_{z_{3:n}} \underbrace{\left(\sum_{z_2} s_2(z_2)p(z_3|z_2)p(x_3|z_3) \right)}_{\text{call this } s_3(z_3)} \prod_{t=4}^{n} p(z_t|z_{t-1})p(x_t|z_t)$$

$$\vdots$$

$$= \sum_{z_{j:n}} \underbrace{\left(\sum_{z_{j-1}} s_{j-1}(z_{j-1})p(z_j|z_{j-1})p(x_j|z_j) \right)}_{\text{call this } s_j(z_j)} \prod_{t=j+1}^{n} p(z_t|z_{t-1})p(x_t|z_t)$$

Forward algorithm (3/3)

- Continuing in this way, we end up with $p(x_{1:n}) = \sum_{z_n} s_n(z_n)$.
- This suggests the following algorithm:

1. For each $z_1 = 1, \ldots, m$, compute $s_1(z_1) = p(z_1)p(x_1|z_1)$.
2. For each $j = 2, \ldots, n$, for each $z_j = 1, \ldots, m$, compute

$$s_j(z_j) = \sum_{z_{j-1}} s_{j-1}(z_{j-1})p(z_j|z_{j-1})p(x_j|z_j).$$

3. $p(x_{1:n}) = \sum_{z_n} s_n(z_n)$.

Forward algorithm: Using it for inference

- In theory, this allows us to compute the normalization constant in order nm^2 time (although as in the case of the Viterbi algorithm, there are numerical underflow/overflow issues—stay tuned).

- The real utility of the algorithm, though, is not that it allows us to compute the normalization constant, but that it gives us the intermediate quantities $s_j(z_j)$. How can we interpret these quantities? It turns out that

$$s_j(z_j) = \sum_{z_{1:j-1}} p(x_{1:j}, z_{1:j}) = p(x_{1:j}, z_j).$$

- As described earlier, when these are combined with the results of the backward algorithm, they can be used to compute many other useful things.

Forward algorithm: Using it for prediction

- Suppose we are interested in inferring the value of z_j based on the observations $x_{1:j}$ (i.e., "online" prediction).

- This can be done using the results of the forward algorithm, since
$$p(z_j|x_{1:j}) \propto p(x_{1:j}, z_j) = s_j(z_j).$$

- Similarly, we can predict x_{j+1} given $x_{1:j}$ using

$$\begin{aligned} p(x_{j+1}|x_{1:j}) \propto p(x_{1:j}, x_{j+1}) &= \sum_{z_j, z_{j+1}} p(x_{1:j}, x_{j+1}, z_j, z_{j+1}) \\ &= \sum_{z_j, z_{j+1}} p(x_{1:j}, z_j) p(z_{j+1}|z_j) p(x_{j+1}|z_{j+1}). \end{aligned}$$

Backward algorithm

- The backward algorithm is derived similarly to the forward algorithm, except that we sum the variables in the reverse order, $z_n, \dots, z_1$.

- This leads to the following algorithm (I will leave the derivation to you):

  1. For each $z_n = 1, \dots, m$, define $r_n(z_n) = 1$.
  2. For each $j = n-1, n-2, \dots, 1$, for each $z_j = 1, \dots, m$, compute
  $$r_j(z_j) = \sum_{z_{j+1}} p(z_{j+1}|z_j)p(x_{j+1}|z_{j+1})r_{j+1}(z_{j+1}).$$
  3. $p(x_{1:n}) = \sum_{z_1} p(z_1)p(x_1|z_1)r_1(z_1)$

Backward algorithm

- The backward algorithm takes order nm^2 time.

- What is the interpretation of the values $r_j(z_j)$?

- Similarly to before, using the directed graphical model,

$$r_j(z_j) = \sum_{z_{j+1:n}} p(x_{j+1:n}, z_{j+1:n}|z_j) = p(x_{j+1:n}|z_j).$$

- As in the case of the Viterbi algorithm, both the forward and backward algorithm suffer from the same issue with underflow/overflow.

- As a consequence, in practice, it is necessary to work with logs. We address this next.

Outline

The log-sum-exp trick

- Consider the forward algorithm. Defining $g_j(z_j) = \log s_j(z_j)$, we have

$$g_j(z_j) = \log s_j(z_j) = \log \sum_{z_{j-1}} s_{j-1}(z_{j-1}) p(z_j|z_{j-1}) p(x_j|z_j)$$
$$= \log \sum_{z_{j-1}} \exp\left(g_{j-1}(z_{j-1}) + \ell(z_j|z_{j-1}) + \ell(x_j|z_j)\right)$$

  denoting $\ell = \log p$ as before.

- The issue is that $g_{j-1}(z_{j-1}) + \ell(z_j|z_{j-1}) + \ell(x_j|z_j)$ is typically going to have very large magnitude (usually negative, but possibly positive), say, -5000 or so.

- When we try to compute $\exp(-5000)$, most programming languages will round this off to be exactly equal to 0.

- The solution is to use the "log-sum-exp trick".

The log-sum-exp trick

- To simplify the notation a bit, let's suppose we would like to compute $\log \sum_{i=1}^m \exp(a_i)$. Note that for any $b \in \mathbb{R}$,

$$\begin{aligned}\log \sum_{i=1}^m \exp(a_i) &= \log \sum_{i=1}^m \exp(a_i - b) \exp(b) \\ &= \log \Big(\exp(b) \sum_{i=1}^m \exp(a_i - b) \Big) \\ &= b + \log \sum_{i=1}^m \exp(a_i - b).\end{aligned}$$

- The key is to choose $b = \max_i a_i$.

- Then, even if all of the a_i's have large magnitude, at least some of the shifted values $a_i - b$ will not result in underflow/overflow when computing $\exp(a_i - b)$, and it turns out that this is enough to solve the issue.

The log-sum-exp trick

- For example, if $a_1 = -3060$, $a_2 = -3056$, and $a_3 = -3071$, we will have $b = -3056$, so

$$b + \log \sum_{i=1}^{m} \exp(a_i - b) = -3056 + \log\left(e^{-4} + e^{0} + e^{-15}\right)$$

which is no problem to compute.

- It can (and usually will) happen that for some i's, $a_i - b$ will be a large negative number.

- For instance, suppose that in the example above we had $a_3 = -3656$. The third term in the sum will be $\exp(-600)$, which the computer will usually treat as exactly 0.

- However, the other two terms will still be fine, and the error introduced will be negligible—the error will be on the order of $\exp(-600)$.

The log-sum-exp trick

- There is one other issue that we need to take care of when using the log-sum-exp trick.
- Specifically, if $b = \infty$ or $b = -\infty$, then $a_i - b = \infty - \infty$ for one or more a_i's, and this will lead to NaN's in most programming languages.
- This is easily resolved by returning b if $b \in \{-\infty, \infty\}$, and otherwise, returning $b + \log \sum_{i=1}^{m} \exp(a_i - b)$.

Group activity: Check your understanding

Go to breakout rooms and work together to answer these questions:
https://forms.gle/cucQw9HvjXNj44Kh7

(Three people per room, randomly assigned. 15 minutes.)

Outline

Baum–Welch algorithm

- So far, we have been assuming that all of the HMM parameters are known (the initial distribution π, the transition matrix T, and the emission distributions ε_i).
- The Baum–Welch algorithm provides a way to estimate these parameters.
- Specifically, it is a special case of the expectation-maximization (EM) algorithm.
- Baum–Welch is an iterative algorithm in which the forward and backward algorithms are used at each iteration.

Refresher on expectation-maximization (EM)

- The goal of EM is to find a maximum likelihood estimate (MLE) or maximum a posteriori (MAP) estimate in models involving latent variables or missing data.
- The tricky thing about models with hidden variables is that the likelihood is often quite complicated and multimodal, making it difficult to maximize.
- Even with EM, we are not guaranteed to find a global maximum. However, the advantage of EM over standard optimization routines is that it exploits the structure of the model in a way that make the optimization computationally efficient.
- EM is designed for cases in which the "complete data" (that is, the observed data along with the hidden data) is modeled as an exponential family.

Refresher on expectation-maximization (EM)

- Observed data: $x = (x_1, \dots, x_n)$.
- Model: $(X, Z) \sim p_\theta(x, z)$. Here, z represents some collection of unobserved variables.
- For example, in an HMM, $z = (z_1, \dots, z_n)$ represents the hidden states.
- EM works best when $p_\theta(x, z)$ is an exponential family.
- Goal: Find
$$\theta_{\mathsf{MLE}} \in \operatorname*{argmax}_\theta p_\theta(x)$$
where $p_\theta(x) = \sum_z p_\theta(x, z)$.
- We will assume that Z is discrete.

Refresher on expectation-maximization (EM)

- Algorithm:
  1. Initialize θ_1.
  2. For $k = 1, 2, \ldots$ until some convergence criterion is met,
     2.1 E-step: Compute the function
     $$\begin{aligned} Q(\theta, \theta_k) &= \mathrm{E}_{\theta_k}\big(\log p_\theta(X, Z) \mid X = x\big) \\ &= \sum_z \big(\log p_\theta(x, z)\big) p_{\theta_k}(z|x). \end{aligned}$$
     2.2 M-step: Solve for $\theta_{k+1} \in \mathrm{argmax}_\theta\, Q(\theta, \theta_k)$.
- In practice, we will often be able to analytically compute and maximize $Q(\theta, \theta_k)$.
- It is usually a good idea to introduce some randomization into the initialization, since hand-picked values of θ_1 sometimes cause EM to get stuck.

Expectation-maximization: Pros and cons

- Advantages of EM:
  - We are guaranteed that $p_{\theta_{k+1}}(x) \geq p_{\theta_k}(x)$ for each k, that is, the likelihood increases (or at least, doesn't decrease).
  - The algorithm tends to work well in practice.
- Disadvantages of EM:
  - Not guaranteed to converge to a global maximum.
  - Maximum likelihood can "overfit". A partial solution to this is that EM can be modified to try to find a MAP estimate instead of an MLE.
  - EM can be slow to converge. There are variations and extensions of the algorithm to improve the convergence rate.
  - EM works best for models in which $p_\theta(x, z)$ is an exponential family.

Baum–Welch algorithm

- In an HMM, the parameter θ specifies π, T, and ε_i for each i.
- Let's suppose that the emission distribution $\varepsilon_i(x)$ belongs to some family of distributions $f_{\varphi_i}(x)$ with parameter φ_i.
- For example, if the emission distributions are normal, then we could define $\varphi_i = (\mu_i, \sigma_i^2)$ and $\varepsilon_i(x) = f_{\varphi_i}(x) = \mathcal{N}(x|\mu_i, \sigma_i^2)$.
- Recall that $\pi_i = \mathbb{P}(Z_1 = i)$ and $T_{ij} = \mathbb{P}(Z_{t+1} = j \mid Z_t = i)$.
- With these conventions, the HMM is parameterized by $\theta = (\pi, T, \varphi)$, where $\varphi = (\varphi_1, \ldots, \varphi_m)$.
- We will assume that there are no functional relationships among π, T, and $\varphi_1, \ldots, \varphi_m$, so that we can maximize with respect to each of them separately.

Baum–Welch algorithm: E-step (1/3)

- In the E-step, we need to compute $Q(\theta, \theta_k)$. Recall that:

$$Q(\theta, \theta_k) = \mathrm{E}_{\theta_k}\big(\log p_\theta(X, Z) \mid X = x\big).$$

- By the factorization assumed in an HMM,

$$\log p_\theta(x, z) = \log p_\theta(z_1) + \sum_{t=2}^{n} \log p_\theta(z_t | z_{t-1}) + \sum_{t=1}^{n} \log p_\theta(x_t | z_t)$$

$$= \sum_{i=1}^{m} \mathrm{I}(z_1 = i) \log \pi_i + \sum_{t=2}^{n} \sum_{i=1}^{m} \sum_{j=1}^{m} \mathrm{I}(z_{t-1} = i, z_t = j) \log T_{ij}$$

$$+ \sum_{t=1}^{n} \sum_{i=1}^{m} \mathrm{I}(z_t = i) \log f_{\varphi_i}(x_t).$$

- The only places where z appears in this expression are in the indicator functions, so when we take the expectation of Z given $X = x$, the expectation moves through and hits only these indicators.

Baum–Welch algorithm: E-step (2/3)

- Further, the expectation of an indicator function is equal to the probability of the event in the indicator—for example, $\mathrm{E}_{\theta_k}\big(\mathrm{I}(Z_t = i) \mid X = x\big) = \mathbb{P}_{\theta_k}(Z_t = i \mid X = x)$.

- Consequently,

$$\begin{aligned} Q(\theta, \theta_k) = &\sum_{i=1}^{m} \mathbb{P}_{\theta_k}(Z_1 = i \mid x) \log \pi_i \\ &+ \sum_{t=2}^{m} \sum_{i=1}^{m} \sum_{j=1}^{m} \mathbb{P}_{\theta_k}(Z_{t-1} = i, Z_t = j \mid x) \log T_{ij} \\ &+ \sum_{t=1}^{n} \sum_{i=1}^{m} \mathbb{P}_{\theta_k}(Z_t = i \mid x) \log f_{\varphi_i}(x_t). \end{aligned}$$

- To simplify the notation, let's define

$$\begin{aligned} \gamma_{ti} &= \mathbb{P}_{\theta_k}(Z_t = i \mid x) \\ \beta_{tij} &= \mathbb{P}_{\theta_k}(Z_{t-1} = i, Z_t = j \mid x). \end{aligned}$$

Baum–Welch algorithm: E-step (3/3)

- With this notation, we have

$$Q(\theta, \theta_k) = \sum_{i=1}^{m} \gamma_{1i} \log \pi_i + \sum_{t=2}^{n} \sum_{i,j=1}^{m} \beta_{tij} \log T_{ij} + \sum_{t=1}^{n} \sum_{i=1}^{m} \gamma_{ti} \log f_{\varphi_i}(x_t).$$

- Now, if we could compute the γ's and β's, then we would have a nice analytical expression for $Q(\theta, \theta_k)$ (as a function of θ).

- The γ's and β's are precisely the quantities that we saw earlier could be computed using the results of the forward-backward algorithm!

- Thus, for any given θ_k, we can use the forward-backward algorithm to efficiently compute the γ's and β's.

Baum–Welch algorithm: The M-step (1/4)

- For the M-step, we need to find a value of θ maximizing $Q(\theta, \theta_k)$.
- Fortunately, it turns out that we can often do this analytically.
- To fully justify all of the steps below, we would need some regularity conditions, but we will ignore these details and just focus on the big picture for now.

Baum–Welch algorithm: The M-step (2/4)

- First, to maximize with respect to φ_i, if the family (f_φ) is sufficiently nice (and often it is), we will be able to simply take the gradient with respect to φ_i, set it equal to zero, and solve for φ_i.

- In other words, find the value of φ_i such that

$$0 = \nabla_{\varphi_i} Q(\theta, \theta_k) = \sum_{t=1}^{n} \gamma_{ti} \big(\nabla_{\varphi_i} \log f_{\varphi_i}(x_t) \big).$$

- Note that the derivative kills off all the terms in our expression for $Q(\theta, \theta_k)$ except for $\sum_{t=1}^{n} \gamma_{ti} \log f_{\varphi_i}(x_t)$.

- The value of φ_i satisfying this equation can be thought of as a weighted MLE, in which data point x_t has weight γ_{ti}.

Baum–Welch algorithm: The M-step (3/4)

- Next, consider π. Things are slightly trickier now, since we need to maximize subject to the constraint that $\sum_{i=1}^{m} \pi_i = 1$.
- Fortunately, we can do this analytically using the method of Lagrange multipliers, as follows.
- Denoting the Lagrange multiplier by λ, we set the derivative of the Lagrangian equal to zero, apply the constraint, and solve for π:

$$0 = \frac{\partial}{\partial \pi_i}\Big(Q(\theta, \theta_k) - \lambda \sum_{j=1}^{m} \pi_j\Big) = \frac{\gamma_{1i}}{\pi_i} - \lambda$$

$$\implies \quad \lambda \pi_i = \gamma_{1i} \quad \implies \quad \lambda = \lambda \sum_{i=1}^{m} \pi_i = \sum_{i=1}^{m} \gamma_{1i},$$

therefore, $\pi_i = \dfrac{\gamma_{1i}}{\sum_{j=1}^{m} \gamma_{1j}}$.

Baum–Welch algorithm: The M-step (4/4)

- Finally, for T, we need to maximize subject to the constraint that $\sum_{j=1}^{m} T_{ij} = 1$ for each i.
- As with π, we can do this analytically using Lagrange multipliers.
- If you work this out, you get

$$T_{ij} = \frac{\sum_{t=2}^{n} \beta_{tij}}{\sum_{t=2}^{n} \sum_{j=1}^{m} \beta_{tij}} = \frac{\sum_{t=2}^{n} \beta_{tij}}{\sum_{t=1}^{n-1} \gamma_{ti}}.$$

Altogether now, with feeling

- Putting all these pieces together, the Baum–Welch algorithm proceeds as follows:

  1. Randomly initialize π, T, and $\varphi = (\varphi_1, \ldots, \varphi_m)$.
  2. Iteratively repeat the following two steps, until convergence:

     2.1 E-step: Compute the γ's and β's using the forward-backward algorithm, given the current values of π, T, φ.

     2.2 M-step: Update π, T, and φ using the formulas above involving the γ's and β's.