Hidden Markov Models

Contents

Jeffrey W. Miller (2016). *Lecture Notes on Advanced Stochastic Modeling.* Duke University, Durham, NC.

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, and many more applications. 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, and parameter estimation using the Baum–Welch algorithm.

1 Setup

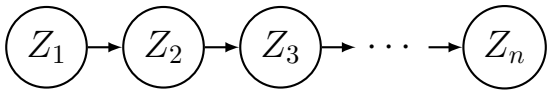
1.1 Refresher on Markov chains

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

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

  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:

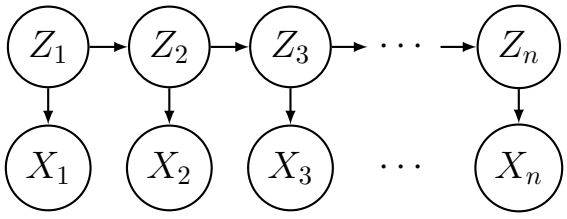


- A Markov chain is a natural model to use for sequential data when the present state Z_t contains all of the information about the future that could be gleaned from $Z_1, \ldots, Z_t$. In other words, when Z_t is the "complete state" of the system.

- If Z_t is sufficiently rich, then this may be a reasonable assumption, but oftentimes we only get to observe an incomplete or noisy version of Z_t. In such cases, a hidden Markov model is preferable.

1.2 Hidden Markov models

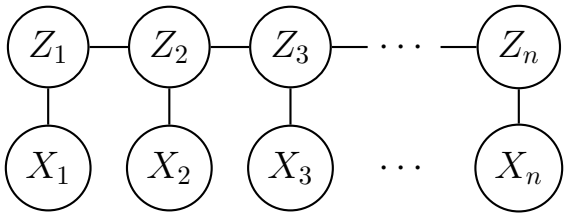
- A hidden Markov model is a distribution $p(x_1, \ldots, x_n, z_1, \ldots, z_n)$ that respects the following directed graph:



In other words, it factors as

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

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



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

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

- Assume that 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".)

- 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, $\pi_i = \mathbb{P}(Z_1 = i)$.

1.3 Example

- $m = 2$ hidden states, i.e., $Z_t \in \{1, 2\}$

- 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 $\sigma = (1, 1)$.

2 Overview of dynamic programming for HMMs

- There are three main algorithms used for inference in HMMs: the Viterbi algorithm, the forward-backward algorithm, and the Baum–Welch 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 a[1]

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

  Naively maximizing over all sequences would take order nm^n time, whereas the Viterbi algorithm only takes nm^2 time.
- The forward-backward algorithm enables one to efficiently compute a wide range of conditional probabilities 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.,

  and it also allows you to sample from the distribution on $z_{1:n}$ given $x_{1:n}$.
- The Baum–Welch algorithm is a method of estimating the parameters of an HMM (the initial distribution, transition matrix, and emission distributions), using expectation-maximization and the forward-backward algorithm.
- 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.

[1] The argmax is the set of maximizers, i.e., $x^* \in \operatorname{argmax}_x f(x)$ means that $f(x^*) = \max_x f(x)$.

3 Viterbi algorithm

- Before we start, note the following facts. If $c \geq 0$ and $f(x) \geq 0$, then $\max_x cf(x) = c \max_x f(x)$ and $\operatorname{argmax}_x cf(x) = \operatorname{argmax}_x f(x)$. Also note that $\max_{x,y} f(x,y) = \max_x \max_y f(x,y)$.

- The goal of the Viterbi algorithm is to find a

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

  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.

3.1 Computing the max

- 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.

- For reasons that will become clear in a second, define $\mu_1(z_1) = p(z_1)p(x_1|z_1)$. Writing out the factorization implied by the graphical model for an HMM,

$$p(x_{1:n}, z_{1:n}) = \underbrace{p(z_1)p(x_1|z_1)}_{\mu_1(z_1)} p(z_2|z_1)p(x_2|z_2) \prod_{t=3}^{n} p(z_t|z_{t-1})p(x_t|z_t),$$

  we have

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

$$= \max_{z_{j:n}} \Big(\underbrace{\max_{z_{j-1}} \mu_{j-1}(z_{j-1}) p(z_j|z_{j-1}) p(x_j|z_j)}_{\text{call this } \mu_j(z_j)} \Big) \prod_{t=j+1}^{n} p(z_t|z_{t-1}) p(x_t|z_t)$$

$$\vdots$$

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

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

- 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.

3.2 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 $z^*_{1:n}$.

- Let's augment step 2 in the algorithm above, by also recording a value of z_{j-1} attaining the maximum in the definition of $\mu_j(z_j)$; let's denote this z_{j-1} 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}.

- 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 $j = n, n-1, \ldots, 2$,
$$\begin{aligned} \mu_j(z^*_j) &= \max_{z_{j-1}} \mu_{j-1}(z_{j-1}) p(z^*_j|z_{j-1}) p(x_j|z^*_j) \\ &= \mu_{j-1}(z^*_{j-1}) p(z^*_j|z^*_{j-1}) p(x_j|z^*_j). \end{aligned}$$

- 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.

- In theory, this provides an algorithm for computing $z_{1:n}^*$. However, in practice, the algorithm above doesn't work! *What?!? Why? And why did we work so hard deriving it then?* The reason why the algorithm fails is very subtle—we will discuss this next—and fortunately, there is an easy fix.

3.3 Fixing arithmetic underflow/overflow by using logs

- Except for rather short sequences in which n is relatively small, say, a couple hundred or so, the algorithm above will fail due to the fact that we are trying to represent numbers that are too small (or too large) for the computer to handle. And what's worse, you will usually receive no warning or error that something has gone wrong. Basically, the issue is that (in most programming languages), there is a limit on how small (or large) of a number can be represented. (For example, in a couple of languages that I use, the lower limit seems to be around 10^{-323}, and the upper limit around 10^{308}.) Anything smaller (or larger) than this will be considered to be exactly zero (or infinity). This is referred to as "arithmetic underflow" (or "arithmetic overflow").

- Unfortunately, in the algorithm described above, we will regularly encounter very very small numbers, because we are multiplying together a large number of probabilities, and arithmetic underflow is very likely to occur. (It is also possible for arithmetic 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 is a trick that works in a lot of other problems as well.

- Denote $\ell = \log p$, e.g., $\ell(z_1) = \log p(z_1)$, $\ell(z_t|z_{t-1}) = \log p(z_t|z_{t-1})$, and $\ell(x_t|z_t) = \log p(x_t|z_t)$.

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

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

  and

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

- The reason why it works is because log is order-preserving. This implies that $f_j(z_j) = \log \mu_j(z_j)$, and consequently, that choosing $\alpha_j(z_j)$ in this way is equivalent to the earlier definition.

4 Forward-backward algorithm

- Just as in the Viterbi algorithm, 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.

- Our derivation of the algorithm will necessarily involve a lot of notation and indices, and may appear to be complicated, but despite appearances, it is actually very simple. The details of the algorithm are not important—what is important is to understand how the algorithm is derived.

- So, how is the algorithm derived? To me, the simplest way to think about it is to ask the question: How can we efficiently compute the normalization constant? In this case, since $p(z_{1:n}|x_{1:n}) = p(x_{1:n}, z_{1:n})/p(x_{1:n})$, and $p(x_{1:n}, z_{1:n})$ is easy to compute, 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. This is especially true in dynamic programming.

- 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, and derive a recursion for computing $p(x_{1:j}, z_j)$ 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, and derive a recursion for computing $p(x_{j+1:n}|z_j)$ 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.
- 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

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

(here, we are using the conditional independence properties implied by the undirected graphical model) 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 $j = 1, \ldots, n-1$. (Note that $p(z_{j+1}|z_j, x_{1:n})$ can be easily computed from $p(z_j, z_{j+1}|x_{1:n})$.)

4.1 Forward algorithm

- 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.
- First, define $s_1(z_1) = p(z_1)p(x_1|z_1)$, for reasons that will become clear shortly. Then the joint distribution factors as

$$p(x_{1:n}, z_{1:n}) = s_1(z_1)p(z_2|z_1)p(x_2|z_2)\prod_{t=3}^{n} p(z_t|z_{t-1})p(x_t|z_t),$$

and

$$\begin{aligned} p(x_{1:n}) &= \sum_{z_{1:n}} p(x_{1:n}, z_{1:n}) \\ &= \sum_{z_{2:n}} \Big(\underbrace{\sum_{z_1} s_1(z_1)p(z_2|z_1)p(x_2|z_2)}_{\text{call this } s_2(z_2)} \Big) \prod_{t=3}^{n} p(z_t|z_{t-1})p(x_t|z_t) \\ &= \sum_{z_{3:n}} \Big(\underbrace{\sum_{z_2} s_2(z_2)p(z_3|z_2)p(x_3|z_3)}_{\text{call this } s_3(z_3)} \Big) \prod_{t=4}^{n} p(z_t|z_{t-1})p(x_t|z_t) \\ &\vdots \end{aligned}$$

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

$$\vdots$$

$$= \sum_{z_n} s_n(z_n).$$

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

- 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 arithmetic 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? If you think about how they are defined, and recall the directed graphical model, it's pretty straightforward to see 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.

- Additionally, if 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 could predict x_{j+1} given $x_{1:j}$ using

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

4.2 Backward algorithm

- The backward algorithm is derived similarly to the forward algorithm, except that we sum the variables in the reverse order, $z_n, \ldots, z_1$.
- This leads to the following algorithm (I will leave the derivation to you):
  1. For each $z_n = 1, \ldots, m$, define $r_n(z_n) = 1$.
  2. For each $j = n-1, n-2, \ldots, 1$, for each $z_j = 1, \ldots, m$, 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. $p(x_{1:n}) = \sum_{z_1} p(z_1) p(x_1|z_1) r_1(z_1)$
- This 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 arithmetic underflow/overflow. As a consequence, in practice, it is necessary to work with log values. We address this next.

4.3 The log-sum-exp trick

- In practice, naively implementing the forward and backward algorithms as described above will not work, due to arithmetic underflow or overflow.
- As with the Viterbi algorithm, the solution is to use logs, but things are a bit trickier now. In the Viterbi algorithm, we were able to interchange the log and the max, but in the forward and backward algorithms, we cannot interchange the log and the sum.
- For example, in the case of the forward algorithm, if we define $g_j(z_j) = \log s_j(z_j)$, then
$$\begin{aligned} g_j(z_j) = \log s_j(z_j) &= \log \sum_{z_{j-1}} s_{j-1}(z_{j-1}) p(z_j|z_{j-1}) p(x_j|z_j) \\ &= \log \sum_{z_{j-1}} \exp\big(g_{j-1}(z_{j-1}) + \ell(z_j|z_{j-1}) + \ell(x_j|z_j)\big) \end{aligned}$$
  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, and when we try to compute $\exp(-5000)$, most programming languages will consider this to be exactly equal to 0.

- The solution is to use what is sometimes called 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 arithmetic underflow/overflow when computing $\exp(a_i - b)$, and it turns out that this is enough to resolve the issue very satisfactorily.

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

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

  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 $a_3 = -3656$. Then 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 $\exp(-600)$. I'd say that's close enough, wouldn't you?

- 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 $b + \log \sum_{i=1}^{m} \exp(a_i - b)$.

5 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. Baum–Welch is a special case of a general class of algorithms referred to as "expectation-maximization", or EM, for short.

- Baum–Welch is an iterative algorithm in which the forward and backward algorithms are used at each iteration.

5.1 Expectation-maximization

- The goal of EM is to find a maximum likelihood estimate (MLE) or maximum a posteriori (MAP) estimate in models involving hidden/latent/unobserved variables/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. It turns out that this is quite common, and consequently, EM is a popular technique for maximum likelihood estimation in complex models.

5.1.1 The EM algorithm

- 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 $\theta_{\text{MLE}} \in \operatorname{argmax}_\theta p_\theta(x)$, where $p_\theta(x) = \sum_z p_\theta(x, z)$. We will assume that Z is discrete.
- Algorithm:
  1. Initialize θ_1.
  2. For $k = 1, 2, \dots$ until some convergence criterion is met,
     (a) E-step: Compute the function
     $$Q(\theta, \theta_k) = \mathbb{E}_{\theta_k}\big(\log p_\theta(X, Z) \mid X = x\big) = \sum_z \big(\log p_\theta(x, z)\big) p_{\theta_k}(z|x).$$
     (b) M-step: Solve for $\theta_{k+1} \in \operatorname{argmax}_\theta Q(\theta, \theta_k)$.
- In practice, we will be able to represent $Q(\theta, \theta_k)$ analytically as a function of θ, once we have computed certain coefficients that depend on θ_k and x. Furthermore, we will often be able to analytically 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 will sometimes cause EM to get stuck.
- The "E" in E-step stands for expectation, and the "M" in M-step stands for maximization.

5.1.2 Pros and cons

- Nice features of EM:
  - We are guaranteed that $p_{\theta_{k+1}}(x) \geq p_{\theta_k}(x)$ for each k, in other words, the likelihood of θ_{k+1} is guaranteed to be at least as high as the likelihood of θ_k.
  - The algorithm tends to work well in practice.
- Unfortunate features of EM:
  - The algorithm is 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 is specialized for models in which $p_\theta(x, z)$ is an exponential family.

5.2 Baum–Welch (EM for HMMs)

- 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).

5.2.1 The E-step

- In the E-step, we need to compute $Q(\theta, \theta_k)$. Let's take a closer look at this to see how we might do it. Recall that:

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

- By the factorization implied by the directed graphical model for an HMM,

$$\begin{aligned}\log p_\theta(x, z) &= \log p_\theta(z_1) + \sum_{t=2}^{n} \log p_\theta(z_t|z_{t-1}) + \sum_{t=1}^{n} \log p_\theta(x_t|z_t) \\ &= \sum_{i=1}^{m} \mathbb{1}(z_1 = i) \log \pi_i + \sum_{t=2}^{n} \sum_{i=1}^{m} \sum_{j=1}^{m} \mathbb{1}(z_{t-1} = i, z_t = j) \log T_{ij} \\ &\quad + \sum_{t=1}^{n} \sum_{i=1}^{m} \mathbb{1}(z_t = i) \log f_{\varphi_i}(x_t).\end{aligned}$$

- 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. Further, the expectation of an indicator function is equal to the probability of the event in the indicator—for example, $\mathbb{E}_{\theta_k}\big(\mathbb{1}(Z_t = i) \mid X = x\big) = \mathbb{P}_{\theta_k}(Z_t = i \mid X = x)$. Consequently,

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

- To simplify the notation, let's define

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

- 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). How can we compute the γ's and β's? Well, do they look familiar at all? That's right, they are precisely the quantities that we saw earlier could be computed using the results of the forward-backward algorithm! Consequently, for any given θ_k, we can use the forward-backward algorithm to efficiently compute the γ's and β's.

5.2.2 The M-step

- 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 to establish concavity and possibly other regularity conditions, but we will ignore these details and just focus on the big picture for now.

- 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}.

- Next, consider $\pi = (\pi_1, \ldots, \pi_m)$. 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,

$$\pi_i = \frac{\gamma_{1i}}{\sum_{j=1}^m \gamma_{1j}}.$$

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

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

5.2.3 Altogether now, with feeling

- Putting all these pieces together, then, 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:
     (a) E-step: Compute the γ's and β's using the forward-backward algorithm, given the current values of π, T, φ.
     (b) M-step: Update π, T, and φ using the formulas above involving the γ's and β's.