Hidden Markov Models

Contents

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

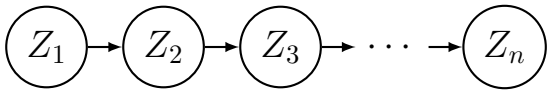
1.1 Refresher on Markov chains

- 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:

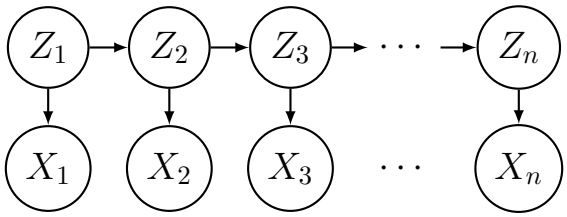


- A Markov chain is a natural model to use for sequential data when the present state Z_t contains all of the information about the future that could be gleaned from $Z_1, \ldots, Z_t$. In other words, when Z_t is the "complete state" of the system.

- 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

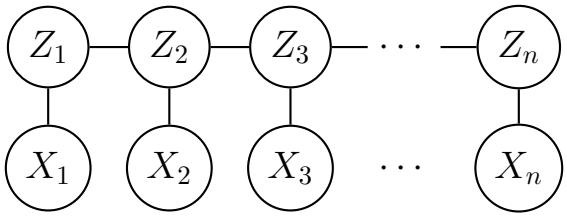
- A hidden Markov model is a distribution $p(x_1, \ldots, x_n, z_1, \ldots, z_n)$ that respects the following directed graph:



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" $T(i, j) = \mathbb{P}(Z_{t+1} = j \mid Z_t = i)$ 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 $T(i, j)$ 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.
- 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.

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

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

- 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

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

- 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

(to do)