1 Introduction

Let $\mathbf{x} = (x^1, \ldots, x^M)$ denote a sequence (e.g. a sequence of words), and let $\mathbf{y} = (y^1, \ldots, y^M)$ denote a corresponding "hidden" sequence that we believe explains or influences $\mathbf{x}$ somehow (e.g. a sequence of part-of-speech tags). For simplicity, we assume that $\mathbf{x}$ is encoded using D tokens, so each $x^j \in 1, ..., D$, and $\mathbf{y}$ is encoded using L tokens, so each $y^j \in 1, ..., L$. Thus there are D^M possible length-M sequences $\mathbf{x}$ and L^M possible length-M sequences of $\mathbf{y}$. Typically we refer to the elements of $\mathbf{x}$ as "observations" and the elements of $\mathbf{y}$ as "states" or "hidden states".

Our goal is to use a first-order hidden Markov model (HMM) to model the joint distribution $P(\mathbf{x}, \mathbf{y})$. An HMM models the joint distribution as follows:

$$P(\mathbf{x}, \mathbf{y}) = P(\texttt{End}|y^M) \prod_{j=1}^{M} P(x^j|y^j) P(y^j|y^{j-1}), \tag{1}$$

where `End` is a special state denoting the end of the hidden sequence and $y^0 := \texttt{Start}$ is always the special state denoting the start of the hidden sequence. Note that the `End` state is optional in HMMs, and can be ignored if you don't wish to model the probability of a sequence spontaneously ending — if we wanted to ignore `End` we would just replace $P(\texttt{End}|y^M)$ with 1 everywhere it appears below.

A closer look at (1) reveals that we have made two big independence assumptions about the sequences $\mathbf{x}$ and $\mathbf{y}$. The first is that observations in $\mathbf{x}$ are conditionally independent of each other given the hidden sequence $\mathbf{y}$:

$$P(\mathbf{x}|\mathbf{y}) = \prod_{j=1}^{M} P(x^j|y^j).$$

In other words, observation x^j in $\mathbf{x}$ only depends on state y^j, not on any other state in $\mathbf{y}$. The second is that $\mathbf{y}$ is entirely independent of $\mathbf{x}$, with each y^j depending only on $y^{<j}$.

Because in order to compute the entire joint distribution, our HMM model only needs to know the values of $P(x^j|y^j)$ and $P(y^j|y^{j-1})$ for all possible combinations of tokens, the total size of our HMM model is $(D \times L) + (L \times L) + 2L$:

Component	No. Parameters
$P(x^j\|y^j)$	$D \times L$
$P(y^j\|y^{j-1})$	$L \times L$
$P(y^1\|y^0)$	L
$P(\texttt{End}\|y^M)$	L

Note that the probabilities of entire sequences $P(\mathbf{x}, \mathbf{y})$ can often be very small (exponentially small in M), so it is usually more convenient to work in log-probability space:

$$\log P(\mathbf{x}, \mathbf{y}) = \log P(\texttt{End}|y^M) + \sum_{j=1}^{M} \left(\log P(x^j|y^j) + \log P(y^j|y^{j-1}) \right). \tag{2}$$

2 How to Make Predictions: The Viterbi Algorithm

Let's say we have a trained HMM model and a sequence $\mathbf{x}$, and we want to predict what hidden sequence $\mathbf{y}$ most likely produced $\mathbf{x}$. E.g. we have a sentence of words and we want to tag each word with a part-of-

speech tag. We make this prediction by selecting the $\mathbf{y}$ that maximizes $P(\mathbf{y}|\mathbf{x})$:

$$\begin{aligned}\operatorname{argmax}_{\mathbf{y}} P(\mathbf{y}|\mathbf{x}) &= \operatorname{argmax}_{\mathbf{y}} \log P(\mathbf{y}|\mathbf{x}) = \operatorname{argmax}_{\mathbf{y}} \log P(\mathbf{x},\mathbf{y})/P(\mathbf{x}) = \operatorname{argmax}_{\mathbf{y}} \log P(\mathbf{x},\mathbf{y}) \\ &= \operatorname{argmax}_{\mathbf{y}} \left[\log P(\texttt{End}|y^M) + \sum_{j=1}^{M} \left(\log P(x^j|y^j) + \log P(y^j|y^{j-1}) \right) \right]. \end{aligned} \tag{3}$$

Naively iterating over all possible $\mathbf{y}$s would take exponential time (w.r.t. the length $|\mathbf{x}| = M$). Luckily, we can use use a dynamic programming approach known as the Viterbi algorithm to efficiently compute (3).

First, we introduce some notation. Let $\tilde{P}$ denote the log probability, e.g., $\tilde{P}(\mathbf{x},\mathbf{y}) = \log P(\mathbf{x},\mathbf{y})$. Define the "transition matrix" $\tilde{A}$ whose entries are

$$\tilde{A}_{a,b} = \tilde{P}(y^j = a | y^{j-1} = b)$$

and the "observation matrix" $\tilde{O}$ whose entries are

$$\tilde{O}_{w,z} = \tilde{P}(x^j = w | y^j = z).$$

Finally, for the key definition, let $\hat{\mathbf{y}}_a^j$ denote the length-j sequence of hidden states ending with $y^j = a$ that is most likely to have produced $\mathbf{x}^{1:j}$:

$$\hat{\mathbf{y}}_a^j = \left(\operatorname{argmax}_{\mathbf{y}^{1:j-1}} \tilde{P}(\mathbf{y}^{1:j-1} \oplus a, \mathbf{x}^{1:j}) \right) \oplus a, \tag{4}$$

where $\oplus$ denotes sequence concatenation or appending a token (it should be clear from context which one it is).

Now, the Viterbi algorithm: If we already knew $\hat{\mathbf{y}}_a^M$ for all different choices of hidden state a, then we could find the $\mathbf{y}$ that maximizes $P(\mathbf{y}|\mathbf{x})$ (which, recall, is the prediction we're looking for) by simply selecting the $\hat{\mathbf{y}}_a^M$ with highest value of $\tilde{P}(\hat{\mathbf{y}}_a^M, \mathbf{x})$. At the other end of things, computing each $\hat{\mathbf{y}}_a^1$ is trivial since by definition $\hat{\mathbf{y}}_a^1 = a$. To find $\hat{\mathbf{y}}_a^j$ for $j > 1$ we use the Viterbi Algorithm, which tells us how to find $\hat{\mathbf{y}}_a^j$ by just looping through all the $\hat{\mathbf{y}}_a^{j-1}$ and choosing the best one:

$$\begin{aligned}\hat{\mathbf{y}}_a^j &= \left(\operatorname*{argmax}_{\mathbf{y}^{1:j-1} \in \{\hat{\mathbf{y}}_1^{j-1}, \ldots, \hat{\mathbf{y}}_L^{j-1}\}} \tilde{P}(\mathbf{y}^{1:j-1} \oplus a, \mathbf{x}^{1:j}) \right) \oplus a \\ &= \left(\operatorname*{argmax}_{\mathbf{y}^{1:j-1} \in \{\hat{\mathbf{y}}_1^{j-1}, \ldots, \hat{\mathbf{y}}_L^{j-1}\}} \tilde{P}(\mathbf{y}^{1:j-1}, \mathbf{x}^{1:j-1}) + \tilde{P}(y^j = a | y^{j-1}) + \tilde{P}(x^j | y^j = a) \right) \oplus a \\ &= \left(\operatorname*{argmax}_{\mathbf{y}^{1:j-1} \in \{\hat{\mathbf{y}}_1^{j-1}, \ldots, \hat{\mathbf{y}}_L^{j-1}\}} \tilde{P}(\mathbf{y}^{1:j-1}, \mathbf{x}^{1:j-1}) + \tilde{A}_{a,y^{j-1}} + \tilde{O}_{x^j,a} \right) \oplus a. \end{aligned} \tag{5}$$

Computing each $\hat{\mathbf{y}}_a^j$ takes $\mathcal{O}(L)$ running time (enumerating over all $\hat{\mathbf{y}}_a^{j-1}$), and so the total running time of this approach is $\mathcal{O}(L^2 M)$. Note that we need to save the $\tilde{P}(\hat{\mathbf{y}}_a^j, \mathbf{x}^{1:j})$ that we compute along the way while computing each (5), since we use them to compute each $\hat{\mathbf{y}}_a^{j+1}$, and we need to save the $\hat{\mathbf{y}}_a^j$ in order to have something to output at the end. Again, note that we're operating with log-probabilities, because the actual probabilities would underflow for any reasonably-sized M.

3 How to Compute Marginal Probabilities: The Forward-Backward Algorithm

Let's say we once again have a trained HMM, and suppose we want to compute some marginal probability from our joint model, like $P(y^5 = 1, \mathbf{x} = 1423)$ or $P(y^3 = 2, y^4 = 2, \mathbf{x} = 334113)$. Why might we want to do this? Potentially because we are genuinely curious to know some marginal probability, but also because knowing these marginals will to let us do unsupervised training of HMMs later.

To compute such quantities, we first define for each hidden state a the vector $\alpha_a \in \mathbb{R}^M$ whose j^{th} entry corresponds to the unnormalized probability of observing the "prefix" $\mathbf{x}^{1:j}$ while also having that $y^j = a$. That is:

$$\alpha_a(j) \propto P(\mathbf{x}^{1:j}, y^j = a) = \sum_{\mathbf{y}^{1:j-1}} P(\mathbf{x}^{1:j}, \mathbf{y}^{1:j-1} \oplus a)$$

We similarly define a sequence of vectors $\beta_b \in \mathbb{R}^{M-1}$ whose j^{th} entry corresponds to the unnormalized probability of observing the "suffix" $\mathbf{x}^{j+1:M}$ given that the j^{th} state is $y^j = b$. That is:

$$\beta_b(j) \propto P(\mathbf{x}^{j+1:M} | y^j = b) = \sum_{\mathbf{y}^{j+1:M}} P(y^{j+1} | y^j = b) P(\mathbf{y}^{j+1:M}, \mathbf{x}^{j+1:M}).$$

(Note that sometimes people define α and β with superscripts in $j = 1, ..., M$ and entries indexed by the hidden states a; all together it's the same quantities, just with different indexing.)

If we had values for the α and β vectors, note that we could easily compute some useful marginal probabilities:

$$P(y^j = a, \mathbf{x}) = \frac{\alpha_a(j)\beta_a(j)}{\sum_{a'} \alpha_{a'}(j)\beta_{a'}(j)}, \tag{6}$$

$$P(y^j = a, y^{j+1} = b, \mathbf{x}) = \frac{\alpha_a(j) P(x^{j+1} | y^{j+1} = b) P(y^{j+1} = b | y^j = a) \beta_b(j+1)}{\sum_{a',b'} \alpha_{a'}(j) P(x^{j+1} | y^{j+1} = b') P(y^{j+1} = b' | y^j = a') \beta_{b'}(j+1)}. \tag{7}$$

But how can we compute $\alpha_a(j)$ and $\beta_b(j)$ efficiently, given that the equations for α and β above require summing over potentially exponentially many hidden subsequences? A pair of dynamic programming algorithms called the Forward and Backward algorithms, both similar to the Viterbi algorithm, will do the job. We initalize $\alpha_a(0)$ and $\beta_b(M)$ as:

$$\alpha_a(0) = \begin{cases} 1 & \text{if } a = \texttt{Start} \\ 0 & \text{otherwise} \end{cases}, \qquad \beta_b(M) = P(\texttt{End} | y^M = b).$$

Define $P_y(b|a) = P(y^{j+1} = b | y^j = a)$, and $P_x(w|z) = P(x^j = w | y^j = z)$. Similar to the Viterbi algorithm, we recursively define each $\alpha_a(j)$ as:

$$\alpha_a(j) = P_x(x^j | a) \sum_{a'} \alpha_{a'}(j-1) P_y(a | a'). \tag{8}$$

We can also recursively define each $\beta_b(j)$ as:

$$\beta_b(j) = \sum_{b'} \beta_{b'}(j+1) P_y(b' | b) P_x(x^{j+1} | b'). \tag{9}$$

(8) is known as the Forward Algorithm, and (9) is known as the Backward Algorithm.

Dealing With Numerical Instability. In practice, directly implementing (8) and (9) leads to numerical instability. But remembering that the entries of α and β need only be *proportional* to the probabilities they

represent, we can avoid overflow and underflow by renormalizing each $\alpha_a(j)$ and $\beta_b(j)$ after each step in (8) and (9), i.e.:

$$\tilde{\alpha}_a(j) = \frac{1}{C_\alpha^j}\left(P_x(x^j|a)\sum_{a'}\alpha_{a'}(j-1)P_y(a|a')\right). \tag{10}$$

and

$$\tilde{\beta}_b(j) = \frac{1}{C_\beta^j}\left(\sum_{b'}\beta_{b'}(j+1)P_y(b'|b)P_x(x^{j+1}|b')\right), \tag{11}$$

for some choice of C_α^j and C_β^j such that overflow and underflow do not happen. (The sum of the vector entries is a good choice.) Afterwards, we can compute (6) and (7) as:

$$P(y^j = a, \mathbf{x}) = \frac{\tilde{\alpha}_a(j)\tilde{\beta}_a(j)}{\sum_{a'}\tilde{\alpha}_{a'}(j)\tilde{\beta}_{a'}(j)}, \tag{12}$$

$$P(y^j = a, y^{j+1} = b, \mathbf{x}) = \frac{\tilde{\alpha}_a(j)P(x^{j+1}|y^{j+1}=b)P(y^{j+1}=b|y^j=a)\tilde{\beta}_b(j+1)}{\sum_{a',b'}\tilde{\alpha}_{a'}(j)P(x^{j+1}|y^{j+1}=b')P(y^{j+1}=b'|y^j=a')\tilde{\beta}_{b'}(j+1)}. \tag{13}$$

Relationship to Viterbi. Viterbi keeps track of the best sequence of length j that ends in some $y^j = a$, and also keeps track of the probability of that sequence. The Forward algorithm keeps track of the marginal probability of all sequences of length j that end in some $y^j = a$. Thus, the Viterbi takes the max whereas the Forward algorithm takes the sum. Because Viterbi doesn't sum, the probabilities of the single best sequence will shrink exponentially as the sequence grows, hence necessitating taking the log to ensure numerical stability. With the Forward-Backward algorithm, we keep track of the unnormalized probabilities, which can both overflow or underflow. But because the values are the product of a bunch of sums, you can normalize at each iteration and still maintain correctness.

4 How to Train: Counting and Baum-Welch

Supervised Training. In the supervised setting, we are given a training set of N training examples:

$$S = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N,$$

and our goal is to learn the parameters of the HMM to maximize the likelihood on S:

$$\text{argmax}\prod_{i=1}^N P(\mathbf{x}_i, \mathbf{y}_i).$$

In this case, training is very straightforward counting:

$$P(y^j = b|y^{j-1} = a) = \frac{\sum_{i=1}^N\sum_{k=1}^{M_i}\mathbf{1}_{[y_i^k=b\wedge y_i^{k-1}=a]}}{\sum_{i=1}^N\sum_{k=1}^{M_i}\mathbf{1}_{[y_i^{k-1}=a]}}.$$

$$P(x^j = w|y^j = a) = \frac{\sum_{i=1}^N\sum_{k=1}^{M_i}\mathbf{1}_{[x_i^k=w\wedge y_i^k=a]}}{\sum_{i=1}^N\sum_{k=1}^{M_i}\mathbf{1}_{[y_i^k=a]}}.$$

Unsupervised Training. In the unsupervised setting, we are given a training set of N training examples containing only the $\mathbf{x}$'s:

$$S = \{\mathbf{x}_i\}_{i=1}^N,$$

and the maximum likelihood problem is thus:

$$\text{argmax} \prod_{i=1}^N P(\mathbf{x}_i) = \text{argmax} \prod_{i=1}^N \sum_{\mathbf{y}} P(\mathbf{x}_i, \mathbf{y}).$$

If we knew the marginal distributions $P(y_i^j = a, \mathbf{x}_i)$ and $P(y_i^j = b, y_i^{j-1} = a, \mathbf{x}_i)$ that we learned how to compute with Forward-Backward, then we could use our training data to estimate the parameters of our HMM model as:

$$P(y^j = b | y^{j-1} = a) = \frac{\sum_{i=1}^N \sum_{k=1}^{M_i} P(y_i^k = b, y_i^{k-1} = a, \mathbf{x}_i)}{\sum_{i=1}^N \sum_{k=1}^{M_i} P(y_i^{k-1} = a, \mathbf{x}_i)}. \tag{14}$$

$$P(x^j = w | y^j = a) = \frac{\sum_{i=1}^N \sum_{k=1}^{M_i} \mathbf{1}_{[x_i^k = w]} P(y_i^k = a, \mathbf{x}_i)}{\sum_{i=1}^N \sum_{k=1}^{M_i} P(y_i^k = a, \mathbf{x}_i)}. \tag{15}$$

And of course if we knew the parameters of our HMM model, then we could run Forward-Backward to get the marginal distributions.

The EM or Baum-Welch algorithm solves this chicken-and-egg problem by iterating between the two computations until convergence. That is, the EM algorithm for unsupervised training of HMMs is

1. INIT: randomly initialize HMM model parameters
2. E-STEP: run Forward-Backward to compute marginal probabilities (6), (7) based on the current model parameters
3. M-STEP: update the model parameters based on the maximium likelihood estimate given the data (14), (15)
4. If not converged, repeat from Step 2