A Simple Recipe for EM Update Equations

Philip Resnik, University of Maryland
resnik@umd.edu

1 Overview

The family of expectation maximization (EM) algorithms is extremely useful in natural language processing, but unfortunately most descriptions get very technical very quickly. This document aims at conveying the intuitions, provides a useful recipe for deriving the update equations used in training, and gives an example of the recipe in action.

In this document, we will talk about the use of EM in cases where we are estimating probabilities that are multinomially distributed, according to the maximum likelihood criterion. This covers some common and interesting cases, for example training of the aggregate bigram model (Saul and Pereira 1997), hidden Markov models, and probabilistic context-free grammars.

A first key intuition is the idea behind *maximum likelihood* estimation (MLE): "good" values for the parameters of your model are likely to have led to the observed training data. Consider a simple coin-flipping (i.e. binomial) model, where there's just one parameter to estimate, namely the probability of heads. If you've observed 10 coin flips and heads came up 7 times, then the estimate $\hat{p}(\text{heads}) = 0.7$ is much more likely to have generated that training set than $\hat{p}(\text{heads}) = 0.2$. So, according to the maximum likelihood criterion, 0.7 is a better choice than 0.2 for the model's one parameter, given the training data you observed.

The second key intuition is that EM's main goal is to approximate the probabilities you *would* have calculated in the obvious way using MLE if the hidden variables had all been observable. The "obvious way" is just to count up the observable events and then normalize so that probabilities sum up to 1. EM is all about estimating what those observable counts *would* be, even if you don't get to see them directly.

A third key intuition is that EM is an *iterative algorithm* that starts with some initial probability estimates (e.g. chosen randomly), and successively improves those estimates by looking at data. You can think of this as a hill-climbing search, where at every point you take a step in the direction that leads upward most steeply. (In this case, your altitude represents the (log-)likelihood of the training data, given your current probability estimates.) If you keep doing this long enough, eventually you'll wind up being at the top of some hill.

2 A general recipe, and a few tricks

EM algorithms have the following general structure:

1. Set initial values for model parameters (i.e. probabilities) μ
2. E-step: Figure out *expected* counts of relevant events, where those events typically involve both observable and hidden values, using the current parameters μ to determine what's expected.
3. M-step: Use the expected counts to compute μ_{new}, a new set of parameters. For purposes of this document, we're assuming this is just a maximum likelihood estimate, so you get probabilities from counts just by normalizing the expected counts.
4. If the process has converged — i.e. if the (log-)likelihood of the data given μ is no longer increasing —or if we've done some maximum number of iterations, stop. Otherwise let $\mu = \mu_{new}$ and go back to the E-step for the next iteration.

My general recipe for turning this structure into a specific algorithm is as follows.

- For the probability you're interested in, write down the maximum likelihood estimate as if you could observe *all* values, including the ones that are officially hidden in the real model. Let's assume that this takes the form $\hat{p_i} = \frac{p(x_i)}{\sum_j p(x_j)}$ or something similar, i.e. the denominator just adds up the numerator over all possible values so that probabilities sum to one.
- Turn the numerator and denominator into expectations, by adding $E[\ ]$ around them, i.e. $\frac{E[p(x_i)]}{E[\sum_j p(x_j)]}$ or something similar.
- If we knew how to compute the numerator, then we we could easily compute the denominator, so now the question is just how to compute the numerator. We do this by turning the numerator into an expression that involves counts.
  - For observable counts, you can just use them directly.
  - For unobservable counts, compute an *expected* count.

Computing the expected counts usually involves the following trick:

- Trick #1: Expected counts. If your current estimate as to the probability of something happening is p, and the number of opportunities for it to happen is N, then the expected number of times it will happen is $p \times N$. For example, if you're currently estimating that a coin has $\hat{p}(\text{heads}) = 0.7$, and you flip it 100 times, then your expected count is $0.7 \times 100 = 70$. Notice that this can also be expressed as a sum over the individual opportunities: $\sum_{i=1}^{100} Pr(\text{result will be heads for the } i^{th} \text{ flip}) = \sum_{i=1}^{100} 0.7 = 70$.

A second trick is often also useful:

- Trick #2: conditional and joint probabilities. Sometimes what you want involves *conditional* probabilities — for example, $p(x_i|O)$ where O is the observed training data. But many models are expressed in terms of *generating* the observed data, which means it's easier to talk about *joint* probabilities. For example, x_i might be a generative step, and $p(x_i, O)$ might be the probability of taking that step in the process of generating the observed output. For example, x_i could represent taking a transition in an HMM, as part of generating the observed word sequence. The handy trick is just to apply the definition of conditional probability, $p(x_i|O) = \frac{p(x_i,O)}{p(O)}$. By computing the right hand side to figure out the left hand side, you've often got something easier to work with in the numerator, because it's a joint rather than conditional probability. Moreover, often you've often already figured out a solution for the denominator; for example, $p(O)$ is solved by the Forward algorithm in the case of HMMs.

3 An example: the forward-backward algorithm

We can apply this recipe in order to get the well known forward-backward algorithm for estimating the parameters of an HMM.

As background, recall a few things.

- An HMM model μ consists of three sets of parameters: $\langle \pi, A, B \rangle$, where π_i is the probability of the HMM starting in state i, each cell a_{ij} in matrix A is the probability of a transition from state i to state j, and each cell b_{j,w_k}in matrix B is the probability of emitting symbol w_k from state j.

- Given an HMM μ, there is an efficient dynamic programming algorithm to compute *forward* probabilities $\alpha_t(i)$, that is, the probability of being in state i at time t when generating observed string of symbols $o_1, o_2, \ldots, o_t$. You can use this algorithm to easily solve the first fundamental question for HMMs, how to compute $Pr(O|\mu)$; that is, the probability assigned by HMM μ to an entire string $O = o_1, o_2, \ldots, o_T$. (Just let $t = T$, the length of the whole string, and sum up $\alpha_t(i)$ over all the states.) Note that another way to write $Pr(O|\mu)$ is $Pr_\mu(O)$, which will be useful later.
- Given an HMM μ, there is also an efficient dynamic programming algorithm to compute *backward* probabilities $\beta_t(j)$, that is, the probability of generating observed string of symbols $o_{t+1}, \ldots, o_T$ if you've started from state j at time t.

  Intuitively, if you have an entire string $O = o_1, o_2, \ldots, o_T$, the forward probability $\alpha_t(i)$ is the probability of getting from the start to state i at time t while generating O, and the backward probability is the probability of getting from state i to the end.

With those things in mind, let's apply our recipe to one critical piece of the problem: how do we compute the new values for the A parameters in μ_{new}? That is, assuming we've got a current estimate μ that includes the parameters a_{ij}, how do we compute the new estimates $\hat{a}_{ij}$? We describe this for a single pair i, j and of course then this needs to be done for all of them.

First, let's pretend that everything is observable. The maximum likelihood estimate is trivial:

$$\hat{a}_{ij} = Pr(i \rightarrow j|O) \quad = \quad \frac{c(i \rightarrow j|O)}{\sum_{j'} c(i \rightarrow j'|O)} \tag{1}$$

We just observe the number of times that the model took a transition from i to j while generating O, and divide by the number of times it took a transition from i to *anywhere*. (The denominator can also be expressed simply as $c(i)$.)

Second, we turn the numerator and denominator into *expected* counts:

$$Pr(i \rightarrow j|O) \quad \approx \quad \frac{E_\mu\left[c(i \rightarrow j|O)\right]}{E_\mu\left[\sum_{j'} c(i \rightarrow j'|O)\right]}. \tag{2}$$

Crucially, notice that the expectations are taken with respect to our *current* estimates of the model parameters, i.e. with respect to μ.[1]

The denominator is trivial if you know how to compute the numerator, so we'll focus on how to compute that.

How do we compute the expected counts in the numerator? First, let's apply Trick #1 and turn counts into probabilities. While we generate $O = o_1 \ldots o_T$, there are T opportunities to transition from state i to state j. so we can re-express the numerator as follows:

$$E_\mu\left[c(i \rightarrow j|O)\right] \quad = \quad \sum_{t=1}^{T} Pr_\mu(i \rightarrow j \text{ at time t}|O). \tag{3}$$

By taking this transition "at time t", I mean that the HMM is in state i at time t, and in state j at time $t+1$.[2]

Great, now we've expressed our expected count in terms of probabilities in our current estimate μ. The next step is to apply Trick #2:

$$\sum_{t=1}^{T} Pr_\mu(i \rightarrow j \text{ at time t}|O) \quad = \quad \sum_{t=1}^{T} \frac{Pr_\mu(i \rightarrow j \text{ at time t}, O)}{\mathrm{Pr}_\mu(O)}. \tag{4}$$

[1]Note that sometimes we call these expected counts *fractional* counts. That's because the expected value might not be a whole number: we're taking an observed event and distributing probability associated with it over a bunch of unobserved events, each of which only gets a fractional part of the credit.

[2]Yes, there may be an off-by-one error here, since the HMM never transitions out of state T. These issues are usually solved by having a special start state for time 0, and a special end state for time $T+1$. Let's not worry about that here.

Why do we do this? Because, as noted above, we already know how to compute the $\Pr_\mu(O)$ in the denominator, using the forward algorithm. And we also know how to take a sum $\sum_{t=1}^{T}$. That means all we really need to figure out now is how to compute

$$Pr_\mu(i \rightarrow j \text{ at time t}, O), \tag{5}$$

which is the joint probability of generating string O *and* taking a transition from state i at time t to state j at time $t+1$.

Here's where the independence assumptions built into the HMM model make life easy. The joint probability we're interested in is the product of four probabilities that, by definition, are independent:

- The probability of being in state i at the point where you've already generated symbols $o_1 \ldots o_t$. Recall that this is just $\alpha_t(i)$, computed using our current estimated model μ.
- The probability of choosing j as the next state after i, i.e. as the state for time $t+1$. This is just our current estimate of the transition probability a_{ij} according to μ.
- The probability of emitting the observed symbol o_{t+1} at time $t+1$. This is just our current estimate of the the emission probability $b_{j,o_{t+1}}$ according to μ.
- The probability of proceeding from state j at time $t+1$ all the way to the end while generating the rest of O. This is just the backward probability $\beta_{t+1}(j)$.

If you plug in the product of those four things as the value of (5), and continue on backwards through the derivation, you should wind up with the following:

$$p_t(i,j) = \frac{\alpha_t(i)\, a_{ij}\, b_{j,o_{t+1}}\, \beta_{t+1}(j)}{Pr_\mu(O)} \tag{6}$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T} p_t(i,j)}{\sum_{j'=1}^{N} \sum_{t=1}^{T} p_t(i,j)} \tag{7}$$

To make things notationally simpler, I've written $p_t(i,j)$ for the *expected count* of a transition from state i at time t to state j at time $t+1$. In some textbooks this is written using a Greek letter I can't pronounce. But if you adjust for notation, this should be the same thing everyone gets.

Now that you've got (6) and (7), it's easy to see why this is called the forward-backward algorithm. At each iteration, first you do a forward pass of dynamic programming, which gives you the α values and, summing over the states at time T, the value for $Pr_\mu(O)$. Then you do a backward pass, to compute the β values. At that point you need simply iterate over all pairs i, j to compute and plug in $\hat{a}_{ij}$ for each one.

Finally, don't forget why we're doing this. The whole point of this is that at every iteration of the EM algorithm, we're going to replace $\mu = \langle \pi, A, B \rangle$ with $\mu_{new} = \langle \hat{\pi}, \hat{A}, \hat{B} \rangle$. What we've done by deriving (7) is to define how to compute the cells in $\hat{A}$. Or, to use a standard bit of terminology, we've derived the EM *update equation* for the A matrix. A similar derivation, actually rather simpler, will get you the update equation for B by following the same recipe.