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 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 probabilities have converged (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 $0.7 \times 100 = 70$.

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 aggregate bigram model

3.1 The model

Consider the following "hidden model" variation on a bigram model for word sequences (Saul and Pereira, 1997). As in the usual bigram model, we express the probability of an entire sequence $w_1 w_2 \dots w_T$ by

(1) $$p(w_1 w_2 \dots w_T) = p(w_1) \prod_{t=2}^{T} p(w_t|w_{t-1}).$$

However, the aggregate bigram model doesn't have any parameters $p(w_t|w_{t-1})$ for word-to-word transitions. Instead, we replace each $p(w_t|w_{t-1})$ as follows,

$$p(w_t|w_{t-1}) \quad = \quad \sum_{i=1}^{C} p(w_t|c_i)p(c_i|w_{t-1}) \tag{2}$$

so that the model is defined as:

$$p(w_1 w_2 \ldots w_T) = p(w_1) \prod_{t=2}^{T} \sum_{i=1}^{C} p(w_t|c_i)p(c_i|w_{t-1}). \tag{3}$$

In plain English, the "generative story" for this model is the following. Instead of generating the next word w_t based on the previous word w_{t-1}, as in a usual bigram model, we generate a "class" c based on w_{t-1}, and then we generate w_t based on c.[1] So the probability of choosing the next word w_t is a sum of probabilities, one for each hidden class. The probability each class c contributes to the choice of w_t, namely $p(w_t|c)p(c|w_{t-1})$, represents the joint probability of two events: picking c based on w_{t-1}, and then picking w_t based on c. Since c is hidden, we have to sum up over the different possibilities. Assuming there are C different classes, that means we sum up over $c_1, c_2, \ldots, c_C$. Intuitively, the hidden class c can be viewed as capturing the general properties of w_{t-1} that are relevant for generating the next word. Or you can think of "going through" hidden class c to get from w_{t-1} to w_t.

As you can see, there are two sets of parameters in this model. The first set is the word-to-class probabilities $p(c|w_i)$, and the second set is the class-to-word probabilities $p(w_j|c)$. (Both w_i and w_j range over the entire vocabulary, and c ranges over the set of C hidden classes.) Because of the way the model is structured, there's an EM algorithm for estimating these parameters that is much simpler than the Forward-Backward algorithm for HMMs, so it's a nice pedagogical example. In particular, there's no need at all for dynamic programming.[2]

In addition to making a nice example, why might this model be interesting? Let's let the size of the vocabulary be V, and, as above, suppose the number of classes is C. Notice that a regular bigram model has V^2 parameters (or more properly $V(V+1)$ if you include the parameters for starting the string with w_i for each w_i, but $O(V^2)$ in any case so let's not nitpick). The aggregate bigram model, which uses hidden variables, has $O(2CV)$ parameters. Minus nitpicking, the latter reaches the former when C exceeds $\frac{V}{2}$. For typical values like $C = 32$ and $V = 50000$, the difference is many orders of magnitude: 2.5×10^9 versus 3.2×10^6. Having many fewer parameters, the aggregate bigram model is likely to be less susceptible to problems of too-sparse training data, assuming we can train it.

3.2 Appying the recipe

So... How is this model trained? For the sake of consistent notation, let's use $N(x, y)$ as the notation for counts. For example, $N(w_i, w_j)$ would be the number of times w_i is followed by w_j.

Following our recipe, suppose the classes c were observable rather than hidden. Let's express the maximum likelihood estimate for the probability $p(w_j|c)$ in terms of the observable counts:

$$p(w_j|c) = \frac{N(w_j, c)}{\sum_{w'_j} N(w'_j, c)} \tag{4}$$

So, if you could actually *see* the choices of class c, you could calculate the conditional probability of going to a particular word w_j given that the previous class is some specific c.[3]

[1]Also, as usual, we can assume w_1 is always a special "start" word that always starts an observed sequence with probability 1.

[2]Intuitively, this is because the choice of the hidden class at every step depends only on what's observable, not any previous hidden states. More precisely, hidden classes are conditionally independent of each other given the intervening word.

[3]Note that the denominator can also be expressed as $N(c)$, though doing so would make it a little less obvious that the denominator is trivial to compute once we deal with the numerator.

Similarly, we can express the maximum likelihood estimate for the probability $p(c|w_i)$ in terms of counts, as follows:

$$p(c|w_i) = \frac{N(c, w_i)}{\sum_{c'} N(c', w_i)}. \tag{5}$$

That is, with observable classes, you could just count up how often this particular w_i generates c, and normalize.[4]

If you read about the EM algorithm for the aggregate bigram model (Saul and Pereira 1997), you don't get a derivation of the steps. The updating of the parameters is simply described as having the following M-step for the two sets of parameters.[5]

$$p_{\texttt{new}}(c|w_i) = \frac{\sum_{w_j} N(w_i, w_j) p(c|w_i, w_j)}{\sum_{c'} \sum_{w_j} N(w_i, w_j) p(c'|w_i, w_j)} \tag{6}$$

$$p_{\texttt{new}}(w_j|c) = \frac{\sum_{w_i} N(w_i, w_j) p(c|w_i, w_j)}{\sum_{w'_j} \sum_{w_i} N(w_i, w'_j) p(c|w_i, w'_j)} \tag{7}$$

The E-step is described as:

$$p(c|w_i, w_j) = \frac{p(w_j|c) p(c|w_i)}{\sum_{c'} p(w_j|c') p(c'|w_i)} \tag{8}$$

All this looks pretty imposing.

But the recipe should help to make things clearer. What's most important here is recognizing that although you want to assign probabilities involving c, you can only observe transitions from w_i to w_j. Therefore the most important question is this: if I counted $N(w_i, w_j)$ transitions from w_i to w_j, what proportion of those transitions would I *expect* to have gone through class c , even if I didn't get to observe that happening?

For particular c, w_i, and w_j, we can express that proportion as the value $p(c|w_i, w_j)$. That is, $p(c|w_i, w_j)$ is the probability that c was the hidden state that got used when "going from" w_i to w_j. Following the recipe, let's derive equation (8) by taking "observable" maximum likelihood estimation as the starting point.

- If we could observe the c, we would just compute

  $$p(c|w_i, w_j) = \frac{N(w_i, c, w_j)}{\sum_{c'} N(w_i, c', w_j)}. \tag{9}$$

  That value in the numerator is just the number of times we'd observe a transition from w_i to w_j, going through this particular state c on the way. The denominator adds that up over all the classes, guaranteeing that the probabilities always sum to 1.

- Since we can't observe the c, we provide an estimate in terms of expected counts.

  $$p(c|w_i, w_j) = \frac{E[N(w_i, c, w_j)]}{E[\sum_{c'} N(w_i, c', w_j)]}. \tag{10}$$

  So now we just need to figure out how to compute the expected counts.

- Now, the expression $N(w_i, c, w_j)$ represents the number of transitions from w_i to w_j that go through c. Using Trick #1, the expected value for this count is (a) the total number of *opportunities to go through c en route from w_i to w_j* times (b) the probability of the opportunity actually going through c.

[4] Again, expressing denominator as $N(w_i)$ is equivalent.

[5] Actually, I would consider the computations of the numerators and denominators in equations (6) and (7) to be part of the E-step, with the M-step just consisting in the division of numerator by denominator. But that's not too important here.

The number of opportunities (a) is observable; it's just $N(w_i, w_j)$.

For the probability (b), this is where the current estimates of the model probabilities come into play. We've got a current estimate for the probability of going from w_i to c; that's just our current value for $p(c|w_i)$. Similarly, we have a current estimate for the probability of going from c to w_j, namely our current value of $p(w_j|c)$. According to the definition of the model, the probability of going from w_i to w_j through c is the product of these two probabilities, so the probability (b) is $p(w_j|c)p(c|w_i)$.

- Putting these pieces together, our numerator and denominator in 10 can be expressed as:

$$p(c|w_i, w_j) = \frac{N(w_i, w_j)p(w_j|c)p(c|w_i)}{\sum_{c'} N(w_i, w_j)p(w_j|c')p(c'|w_i)}. \tag{11}$$

- The rest is simple algebra. We can pull $N(w_i, w_j)$ outside the sum in the denominator, and then cancel with the same value in the numerator, yielding the expression for $p(c|w_i, w_j)$ given in equation (8) and repeated here for convenience:

$$p(c|w_i, w_j) = \frac{p(w_j|c)p(c|w_i)}{\sum_{c'} p(w_j|c')p(c'|w_i)} \tag{12}$$

Now we are in a position to derive $p_{\texttt{new}}(w_j|c)$. The logic is quite similar.

- Applying our recipe, if we could observe everything we'd compute

$$p_{\texttt{new}}(w_j|c) = \frac{N(w_j, c)}{\sum_{w'_j} N(w'_j, c)}.$$

- But the c are unobserved, so we need to use expectations based on the current probabilities. This leads us to wrap expectations around the numerator and denominator:

$$p_{\texttt{new}}(w_j|c) = \frac{E[N(w_j, c)]}{E[\sum_{w'_j} N(w'_j, c)]}. \tag{13}$$

- The numerator describes the number of times we go to w_j from any w_i, *and* do so via c. We can express this as a sum of counts over all the w_i we could have come from, i.e.

$$N(w_j, c) = \sum_{w_i} N(w_i, c, w_j).$$

- We again apply Trick #1, expressing our expected value for that total count in terms of (a) the number of observed opportunities to go through c (which is the number of times we go from w_i to w_j, also known as $N(w_i, w_j)$) and (b) the probability of such an opportunity actually going through c (which is $p(c|w_i, w_j)$). In other words,

$$E[N(w_j, c)] = \sum_{w_i} N(w_i, w_j) \times p(c|w_i, w_j) \tag{14}$$

- Thus, the only remaining missing piece is the problem of figuring out the value of $p(c|w_i, w_j)$ given our current probability estimates. But we've already solved that in eq. (8)! So putting equation (14) together with equation (13), we wind up with the update in equation (7), repeated here for convenience:

$$p_{\texttt{new}}(w_j|c) = \frac{\sum_{w_i} N(w_i, w_j)p(c|w_i, w_j)}{\sum_{w'_j} \sum_{w_i} N(w_i, w'_j)p(c|w_i, w'_j)} \tag{15}$$

Here's a summary. The numerator is an expected count of the number of times we go from c to w_j. Based on what we observe in the training data, there were $\sum_{w_i} N(w_i, w_j)$ opportunities to go from c to j. Based on our *current* probability estimates, the fraction of them actually going through c is $p(c|w_i, w_j)$. If you multiply the number of opportunities times the probability of it actually happening, then, *voila!*, you have the expected number of times it will happen. The denominator just sums that up over all the possible destinations $w_{j'}$ you could have gone to from c, so that the conditional probabilities sum to 1.

4 Let's do it again!

As an exercise *before* reading the rest of this section, see if you can follow the same process we just went through in order to explain why equation 6 is the way to update $p_{\texttt{new}}(c|w_i)$.

Here's the answer. The logic is identical. Moving from observable to expectations, you get

$$p(c|w_i) = \frac{E[N(c, w_i)]}{E[\sum_{c'} N(c', w_i)]}.$$

To get the joint count you want in the numerator, you can sum $N(w_i, c, w_j)$ over w_j. So now the problem is again reduced to estimating the count $N(w_i, c, w_j)$ by multiplying the number of observed opportunities, $N(w_i, w_j)$, times the probability of going from w_i to w_j through class c, hence $N(w_i, w_j)p(c|w_i, w_j)$. This yields equation (6), repeated here for convenience:

$$p_{\texttt{new}}(c|w_i) = \frac{\sum_{w_j} N(w_i, w_j)p(c|w_i, w_j)}{\sum_{c'} \sum_{w_j} N(w_i, w_j)p(c'|w_i, w_j)}. \tag{16}$$

Notice one minor difference is in the normalization of the expected counts: since we're conditioning on w_i, the denominator must sum the numerator over classes c'.

5 An intuition about what EM is doing

Here's a useful way to visualize what happens during EM training. Imagine that each transition from every w_i to every c is a pipe made out of some flexible material, and the same for each transition from the cs to the w_js.

The width of each pipe corresponds to its probability: when $p(c|w_i)$ is large, for example, the corresponding pipe has a wide diameter, permitting lots of liquid to flow through it. When a probability is small, the pipe is very narrow, like a straw.

The data you're training on can be imagined as liquid that goes through the pipes. Each iteration of EM – that is, each update to get new probabilities in equations (6) and (7) – can be pictured as pushing the liquid training set through the pipes. When the liquid goes from, say, w_7 to w_{15}, we don't get to see how it gets split up through the different classes c in between them; that's hidden. But if there's a lot of data going through a narrow pipe, say from w_7 to c_{1234}, then that pipe will get forced a little wider. On the other hand, if you don't see some w_i very much in the training data, then none of the pipes involving it are likely to have much liquid going through them, no matter how wide they were to begin with; therefore they shrink.

After many iterations of pushing data through the model, the various pipes have widened and narrowed to a stable set of diameters. Pushing the same data through again and again will no longer lead to any changes. This set of pipe diameters (i.e. probabilities) represents a local maximum for the likelihood of the data given the model. (Or, if you're lucky and the model is convex, a global maximum.)

References

L. Saul and F. Pereira, "Aggregate and mixed-order Markov models for statistical language processing". In Proceedings of the 2nd International Conference on Empirical Methods in Natural Language Processing, 1997.