An Evening with...
EM

A Tutorial by Dirk Hovy
USC's Information Sciences Institute
dirkh@isi.edu

October 20, 2012

The EM algorithm (Dempster/Laird/Rubin, 1977) is one of the most widely used unsupervised learning methods in NLP. It is important to have a solid understanding of its properties and limitations in order to use it well. However, it is quite complex and a bit tricky, and can easily be confusing for the beginner. There are a lot of terms, a lot of implementations, and a lot of Greek letters to add to the confusion. This tutorial tries to avoid the confusion by providing the step-by-step implementation of a concrete case (with pseudocode and a Python implementation), highlighting only the necessary terms and providing layman's explanations of the underlying concepts.

1 Introduction

Before we look at what EM is, let's look at some of the common misconceptions about EM, and clear up **what it is *not***:

what it is *not*

- a model (it's a way to optimize one)
- a silver bullet (you have to be careful what to use it for)
- magic (really, it's just code...)

So, after we got that out of the way, let's see what EM actually is. EM is a form of **unsupervised learning**. It is not so much an algorithm, but rather a class of algorithms that use a 2-step procedure (E step, M step) to train a **generative model**. Generative models are joint probabilities ($P(x,y)$) that explain how the data was, well, generated (*discriminative* models, on the other hand, are simply weight vectors that explain $P(x|y)$). EM tries to find the parameters of a model which best explain the observed data. Or: "If only I knew X, I could estimate Y. If only I knew Y, I could estimate X." We will see this sentence a couple of times.

unsupervised learning

generative model

Throughout the tutorial, there will be little questions to see whether you are still awake. You can just ignore them and read on, or try to solve them and feel good when you look up the answer at the end. Don't beat yourself up when you got one wrong, though!

Also, do not get scared by complicated-looking formulas. We will translate them into plain English, and the mathematical notation will become just the shorthand it was meant to be.

2 Preliminaries

In order to understand EM, we need to look at probabilities and graphical models. If you are not quite sure what they are, fear not, we will explain them in the next sections. If you are already familiar with the concepts, you can skip ahead to Section 3.

2.1 Probabilities

We will be talking about probabilities a lot, so we will review some basics. If you want to get deeper into it, look at the relevant sections in Manning/Schütze (2000) and Russell/Norvig (2003).

Since we do NLP, let's look at the probability of words. Say we have a corpus of 100 sentences, x is "unsupervised" and occurs in 20 sentences, and y is "learning" and occurs in 50 sentences.

Probabilities are basically counts that have been normalized. $P(x)$, or the probability of seeing a sentence with "unsupervised", is simply the count of sentences with "unsupervised" (20) divided by the number of all sentences in our corpus/text (100). $P(x)$

$$P(unsupervised) = \frac{count(sentences\ w.\ "unsupervised")}{number\ of\ all\ sentences} = \frac{20}{100} = \frac{1}{5} = 0.2 \quad (1)$$

$P(x,y)$ is a **joint probability**, i.e., how likely is it that we see x and y together, that is the words in one sentence ("unsupervised learning" or "learning unsupervised", or even separated by other words, order does not matter in this case). Say we have 10 sentences that contain both words, then joint probability

$$P(unsupervised, learning) = \frac{count(sentences\ w.\ "unsupervised"\ and\ "learning")}{number\ of\ all\ sentences} \quad (2)$$

$$= \frac{10}{100} = \frac{1}{10} = 0.1 \quad (3)$$

$P(y|x)$ is a **conditional probability**, i.e., how likely is it to see y after having seen x, or "learning" in a sentence that contains "unsupervised". We can compute this as conditional probability

$$P(y|x) = \frac{P(x,y)}{P(x)} = \frac{\frac{count(sentences\ w.\ "unsupervised"\ and\ "learning")}{number\ of\ all\ sentences}}{\frac{count(sentences\ w.\ "unsupervised")}{number\ of\ all\ sentences}} \quad (4)$$

$$= \frac{count(sentences\ w.\ "unsupervised"\ and\ "learning")}{count(sentences\ w.\ "unsupervised")} \quad (5)$$

$$= \frac{10}{20} = \frac{1}{2} = 0.5 \quad (6)$$

Note that order *does* matter in this case!

QUESTION: What would be the corresponding probability formulation for seeing "unsupervised" in a sentence that contains "learning"?[1]

2.1.1 Where Probabilities Come From

You might have wondered where the probabilities we are gonna use come from. That is a very good question, and there are two answers. The good, scientifically satisfying answer is “data". We can use the statistics collected by statistical institutions (now you know what they are good for), or we can get them by counting and dividing from (preferably large amounts of) data. How likely is ist that it is sunny in LA? Get the weather data of the last 30 years, count the days marked *sunny* and divide by the total number of days. The larger your sample, the more accurate your probability.

The other, less satisfying answer is “elaborated guessing". Some things cannot be measured, or there is simply no data (some things are just too rare. The only exception to this rule is baseball. There are statistics for everything in baseball). What is the probability of being eaten by a tiger in Montana? Probably very small (luckily), but what number is “very small"? We can make up a number, but it might not be accurate.

The good news is that we can use EM to make up numbers and then adjust them until our model explains the data best.

2.2 Graphical Models

Graphical Models are a nice way of visualizing probabilistic models, and also to express the dependencies that hold between the individual elements. There are several types of graphical models, but the ones we are interested in (and the ones we mean when we use the term) are **Bayes Nets** and **Hidden Markov Models** (HMMs). Graphical models consist of two elements, **nodes and arcs**.

Bayes Nets
Hidden Markov Models
nodes and arcs

The nodes are **random variables**. Random variables are events that have some probability, and come in different flavors. If a random variable has exactly two values, like $\{on, off\}$ or $\{true, false\}$, they are **binary** (in the latter case boolean). If they have a list of values (something like $\{red, green, blue\}$ or $\{chocolate, vanilla, strawberry, pistacchio\}$), they are **discrete**. If they have numbers as values, they are called **continuous**. The probabilities associated with each of the values of a random variable sum up to 1.0, i.e., the values $\{red, green, blue\}$ for the variable *COLOR* could be $\{0.2, 0.5, 0.3\}$ or $\{0.33, 0.33, 0.33\}$, but not $\{0.8, 0.4, 0.7\}$.

random variables
binary
discrete
continuous

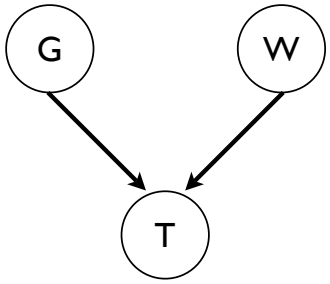

Figure 1: A simple graphical model with three random variables

Arcs are the directed links between the random variables, and you can think of them as causal relations (there are other kinds, but it is easiest this way). They denote what influence the parent has on the child node. This influence is expressed by a conditional probability. Thus in a network like the one in Figure 1, we can say how likely it is that traffic (T) is *bad*, given that the weather (W) *raining*. A node

X can have several parents, which means that its value is influenced by several factors (traffic could also be influenced by a Lakers game, G). If there are no links between two variables, then they are independent of one another, i.e., whether the Lakers play or not luckily has no influence on the weather W (the examples in this section are largely influenced by Russell/Norvig 2003).

2.2.1 Bayes Nets

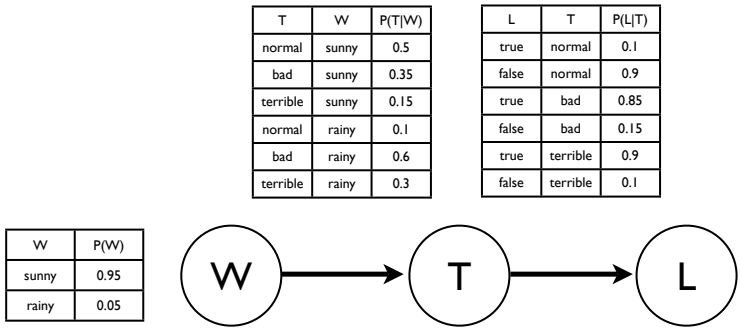

Figure 2: A Bayes Net with three variables

If we combine several nodes in a network, we call it a Bayes Net. Let's look at a very simple example (Figure 2), inspired by Russell/Norvig (2003). Say we have three random variables, namely the weather (W), with values $\{sunny, rainy\}$; traffic (T), which can be $\{normal, bad, terrible\}$; and whether we are late for a meeting ($L : \{true, false\}$).

QUESTION: What are the types of the random variables W, T, and L?[2]

From pop culture we know that it never rains in southern California, and from our meterological data (see section 2.1.1) we know that *never* means 5%. So the probabilities for W are $(0.95, 0.05)$. If we talk about the probability of a specific outcome of the variables values, we write $P(W = sunny) = 0.95$ or shorter $P(sunny) = 0.95$.

If it rains, traffic tends to get worse, and if traffic is bad, we are more likely to be late for our meeting. If it is *sunny*, the traffic behaves different than when it is *rainy*, so we have to specify the probability of each value of T for each value of W. We do that in a table, where each column is a value for a variable, T and W. Notice that the rows with the same value for W have to sum up to 1.0. You can imagine that each value for weather is a state you are in, and the different values for T are options you can choose from. Some options are more likely than others, but all probability is distributed between them (thus summing to 1.0). You cannot choose something that is not there.

Whether I am late for a meeting (L) in turn depends on the state of the traffic (T), so we have to specify another table with probabilities for each value of L given each value of T. Again you can see that with worse traffic, our chances of being late increase.

Using the Bayes Net, we can now compute how likely we are to be late if the weather is bad but traffic is normal, and other interesting things.

2.2.2 Hidden Markov Models

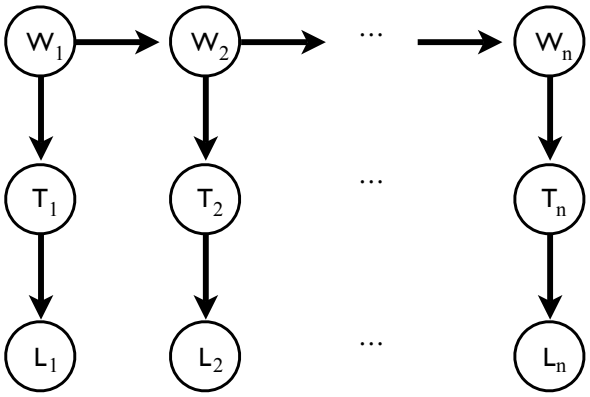

Figure 3: A Hidden Markov Model of the Bayes Net above

Things change over time, but they migh tbe connected. Tomorrow's weather does not just happen, it actually depends on the weather of today. If we want to capture this, we can include another kind of conditional probabilities, namely the ones expressing how a random variable changes over time. This is the Markov part of HMMs. To make things easier, we assume that each state depends only on the previous one, not all previous states. This is one of the so-called **Markov properties**. In order to make it a *hidden* Markov model, we assume that the random variable we are actually interested in is unobservable, but related to something we can observe.

Markov properties

Using our example from above, we have the following scenario: one year from now, we want to get the sequence of sunny and rainy days that occurred (see Figure 3). We do not remember the weather (W is hidden), but we do have our diary, in which we noted for each day whether we were late or not (L is our observed variable, and it is dependent on W). We just copy the Bayes net from above for each day, and add the new transition probabilities $P(W_t|W_{t-1})$ between each of the Bayes nets. The probability means "how likely is it to be $\{rainy, sunny\}$ today if it was $\{rainy, sunny\}$ yesterday".

In this case, we do have another hidden variable, T, but it is not necessary for HMMs. The important part is that whether I am late one day does not depend on whether I was late the day before, but on the weather on that day. Also, the traffic of today is independent of yesterday's traffic. This is why there are no arcs between the T and L variables, only the W nodes. This is another Markov property, that the observations (here, L) are independent of one another. We guesstimated the probabilities $P(T|W)$ and $P(L|T)$ based on intuitions or data (we will later say, we initialized them), and we could now use EM to adjust them to reflect observations, using our diary as data and reconstructing the weather one year ago.

3 Example Uses of EM

Let's start with a practical NLP example of what EM is good for. Say you have a context free grammar (CFG), and would like to attach probabilities to each rule like $P(S \rightarrow NP\ VP|S)$ to say how likely it is that S goes to $NP\ VP$ (vs. that S goes to $S\ CC\ S$).

QUESTION: How do you get the probabilities?[3]

That's easy enough, but it presupposes that we do have some suitable treebank. What if we do not have a treebank, only plain text? This case is far more common... No problem, if you just had some rule probabilities, you could generate a treebank by applying the most likely rules to produce the text.

QUESTION: What would you use to do that?[4]

Oh wait, dangit! To get those rule probabilities, you'd have to have a treebank to collect them from. But that's what we wanted in the first place. This is a pretty circular problem! "If only I knew the rule probabilities, I could make a treebank from text. If only I had a treebank, I could compute the rule probabilities..."

EM can help us solve this problem. But how?

k-means

If you know ***k*-means** clustering, you already know how EM works! If not, don't despair. Here is an example: We want to divide the green points in Figure 4 into a red and a blue cluster. If

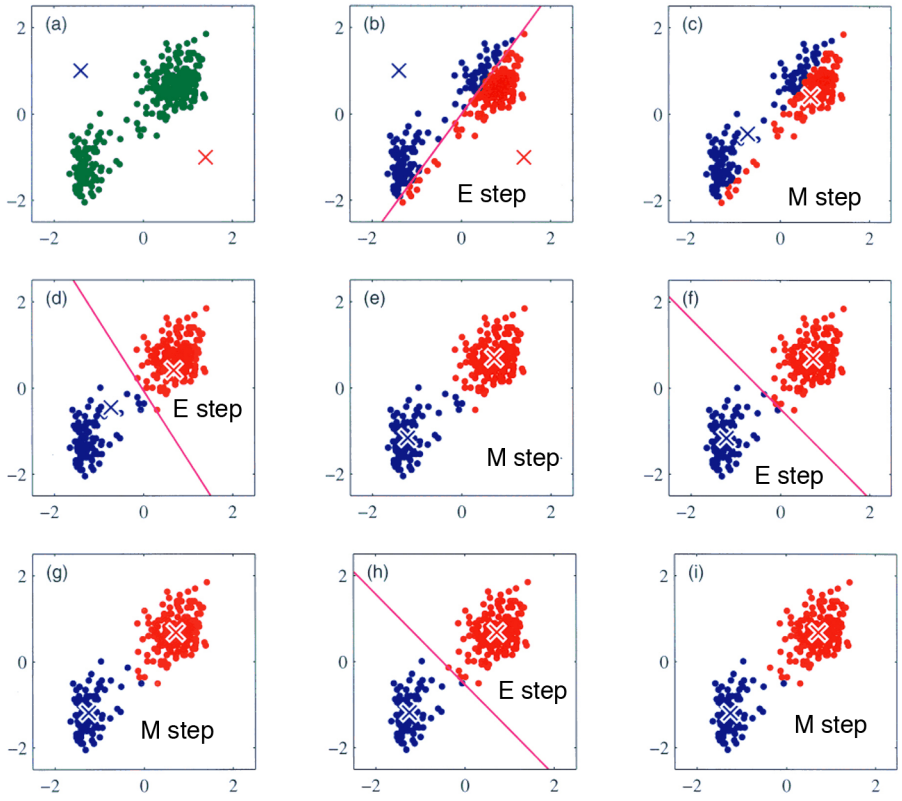

Figure 4: *k*-means clustering. Modified from Bishop 2006

only we knew the cluster centroids, we could assign the data points to the closest clusters. If only we knew which clusters the data points belong to, we could compute their centroids... Sounds somehow familiar? Again, we have a circular problem. And here is how we gonna solve it:

We start by randomly placing cluster centroids on the graph (a). Then, we assign each data point to a cluster (b).

QUESTION: How do we assign the points to a cluster?[5]

Then, we compute the centers of those new clusters and move the centroids to that position (c).

Algorithm : KMEANS(k)

for each $c \in k$
  do $centroid[c] = random(x,y)$

while not *converged*
do
  comment: E step
  for each $c_i \in clusters$
    do $c_i =$ COMPUTECLUSTERMEMBERS(i)
  comment: M step
  for each $f_j \in centroids$
    do $f_j =$ COMPUTECENTER(c_j)

Figure 5: Pseudocode for the k-means algorithm, adapted from Manning/Schütze 2000

As we can see, we alternate between assigning the points to clusters and computing new centroids. Once the centroids stop moving around, we are done.

EM works similarly, and in fact is a "soft" version of k-means. Instead of assigning each point to just one cluster (hard clustering), EM will attach a probability to the membership of a point in each cluster ($P(cluster|point)$). A data point can thus belong to several clusters (though with different probabilities).

Probably the most well-known NLP task for EM is **labeling** unannotated text. We want to find $\arg\max_t P(t|w)$, i.e., given some words w, what is the best tag sequence t? Or: "If only I knew the right tag sequence, I could compute their probabilities... If only I knew the tag probabilities, I could tag the words." This is going to be our running example, and I will use t for a tag sequence and w for a word sequence. The subscript $_i$ denotes the position in the respective sequence (w_i is the i^{th} word). labeling

As mentioned in the first section, there are several algorithms to do EM (especially the E step), and the one we will be using is the **Forward-Backward** (or Baum-Welch) algorithm for labeling (there are other algorithms for other tasks). Forward-Backward

4 The Goal

What we want from EM is the model **parameters** that best explain the observed data.If we plot how well a model configuration explains the data over all possible parameter configurations, we get the graph in Figure 6. parameters

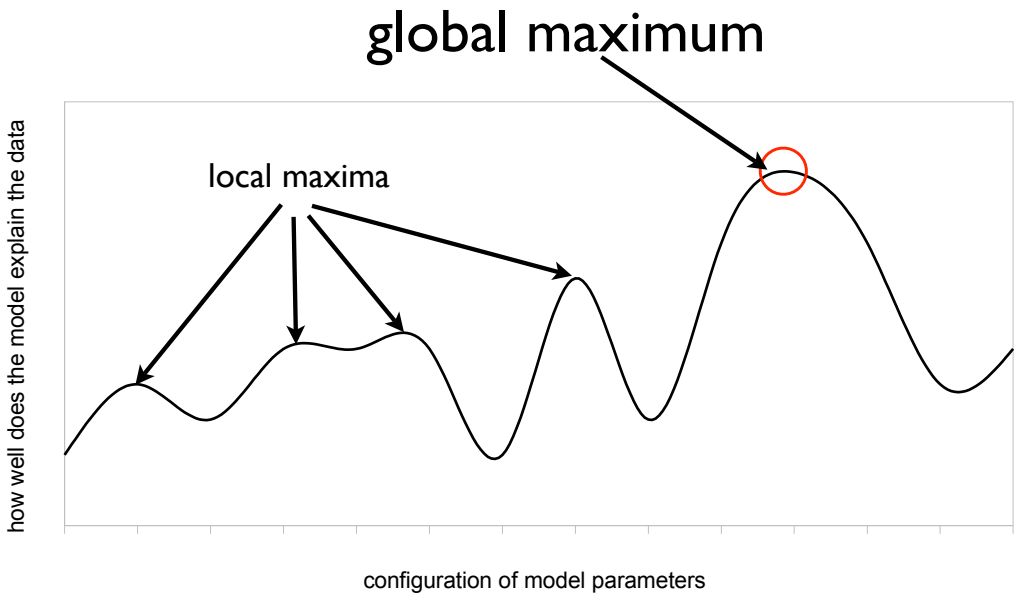

Figure 6: What we try to optimize with EM

The configuration we want is the one at the highest peak. As you can see, there are several smaller peaks. These are local maxima (they are a bit of a hassle, but we come back to that later).

Let's first look at how we get those parameters. There will be a few formulas involved, but don't be intimidated, they are easier than they look.

Remember that EM models a joint probability distribution $P(x,y)$. One can write that as

$$P(x,y) = P(y) * P(x|y) = P(x) * P(y|x) \tag{7}$$

The last two parts are the same because we don't care about the order of x and y. To see why that is

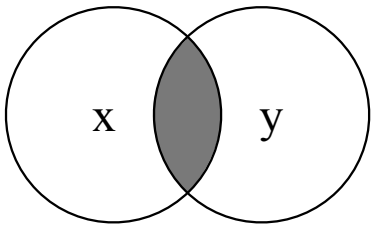

Figure 7: Venn diagram for $P(x,y)$

so, look at the diagram in Figure 7. $P(x,y)$ is the grey area. To get that, we can just look at x and the part of it that overlaps with y. This is $P(x) * P(y|x)$. You could call this very x-centric. If you don't like

that, you can get the same result by looking at y and the part of it that overlaps with x. That would be $P(y) * P(x|y)$.

In our case, however, we don't want to model xs and ys, but the probability of a word and tag sequence occurring together, $P(w,t)$. So let's substitute x and y for w and t and see what we get. Something like

$$P(w,t) = P(t) * P(w|t) = P(w) * P(t|w) \tag{8}$$

This says that the probability of the word and tag sequence together ($P(w,t)$) is equal to the probability of the tag sequence $P(t)$ times the probability that we generate the words from that tag sequence ($P(w|t)$). It is also equal to the probability of seeing those words ($P(w)$) times the probability of turning those words into that tag sequence ($P(t|w)$). Both are equivalent to $P(w,t)$, and that fact will come in handy.

What we want to maximize is the last part, $P(t|w)$ (i.e., what is the best tag sequence for the sentence we see), since we have the words and want to know the tags. Let's take the last two parts and move a few things around to get $P(t|w)$ alone.

QUESTION: How do we do that? Don't peak ahead...[6]

$$\frac{P(t) * P(w|t)}{P(w)} = P(t|w) \tag{9}$$

Ok, that's something. Take a deep breath and make sure you followed this.

Since we observe w (the sentence), we can say that $P(w)$ is 1.0. In that case we can forget about that denominator! So what we want to optimize ($P(t|w)$) is simply

$$P(t|w) = P(t) * P(w|t) \tag{10}$$

Much cleaner, hmm?

Let's look at that $P(t)$. The probability of seeing the whole tag sequence $t_1, t_2, ..., t_n$ ($P(t)$) is really just the product of seeing each of the tags following another tag. We write that as

$$P(t) = P(t_1) * P(t_2|t_1) * P(t_3|t_2) * ... * P(t_n|t_{n-1}) \tag{11}$$

or for short

$$P(t) = P(t_1) * \prod_{i=2}^{n} P(t_i|t_{i-1}) \tag{12}$$

Ok, so we played with the formula, made it cleaner, and dissected $P(t)$. But how does that help us? Good question.

We wanted to find the parameters for our model, and now we have them: $P(t)$ and $P(w|t)$. So we put it all together, and what we want to optimize is finally the product of our two parameters.

$$P(t|w) = P(t_1) * P(w_1|t_1) * \prod_{i=2}^{n} P(t_i|t_{i-1}) * P(w_i|t_i) \tag{13}$$

By the way: parameters come in two flavors: **free parameters** (these are the ones we want EM to optimize) and **fixed parameters** (we already know these and don't want EM to change them). You can make all parameters free or just some!

free parameters

fixed parameters

QUESTION: What happened if you fixed all parameters?[7]

5 Implementation

In order to compute the parameters, we have to develop a data structure that allows us to manipulate them. We will use a Hidden Markov Model to represent the model and lattice-based dynamic programming to compute and manipulate the probabilities. The following sections walk through the individual parts and explain them in detail using pseudocode. You can find a Python implementation (not optimized for performance) in the Appendix (page 23).

5.1 HMM as Lattice

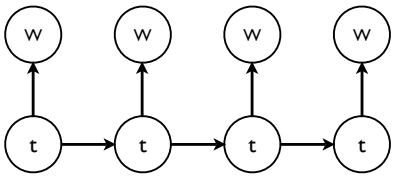

Figure 8: The Hidden Markov Model of the tagging task

The idea of our HMM (see Figure 8): What we can see (the words) was generated by something we cannot see (the tag sequence). This is our **generative story**. Sounds strange? Just wait...

generative story

Tags are connected by **transition probabilities** $P(t_i|t_{i-1})$, and emit words with **emission probabilities** $P(w_i|t_i)$. These look strangely familiar... We could also say "Assume the first word was generated by the first tag, how likely is it the next word was generated by the next tag, and how likely is it that that tag followed the first?" This translates to a sequence of conditional probabilities that we already know from earlier (aren't you glad now we went through all those equations?):

transition probabilities

emission probabilities

$$P(t|w) = \prod_{i=1}^{n} P(t_i|t_{i-1}) * P(w_i|t_i) \tag{14}$$

Again, this just means "the most likely tag sequence given a sentence is computed by concatenating the most likely tags that can emit those words".

QUESTION: Why do we need $P(t_i|t_{i-1})$ in there? Why don't we just take the best tag for each word ($P(t|w)$) and be done?[8]

We can model HMMs as a matrix/**lattice** (or automaton) of tags and words, as in Figure 9, by replacing each random variable in the HMM with all possible values and drawing all possible arcs between them. This can be tricky, and translating from a graphical model to a lattice takes some getting used to. lattice

In this case, we want to label "Mice like cheese", and have an alphabet of only two tags, N and V. It is important to specify all the tags you want to use. We start from a designated start state and from there choose one of the tags with the respective probability $P(t)$. From each of those possible tag states, we can emit a word with the respective probability $P(w|t)$. Those are the horizontal lines in our lattice. Then, we choose the next tag with some probability $P(t_i|t_{i-1})$. Those are the crossing lines in the lattice.

To visualize this, we list all tags as rows. If we have a **dictionary** that tells us that some words can only have certain tags, we simply set all other $P(w|t)$ for this word to 0 (here, we could set ($P(mice|verb) = 0$) and omit the arcs. If we don't have a such a dictionary, we have to assume that all words can be emitted by all tags and let EM figure it out. dictionary

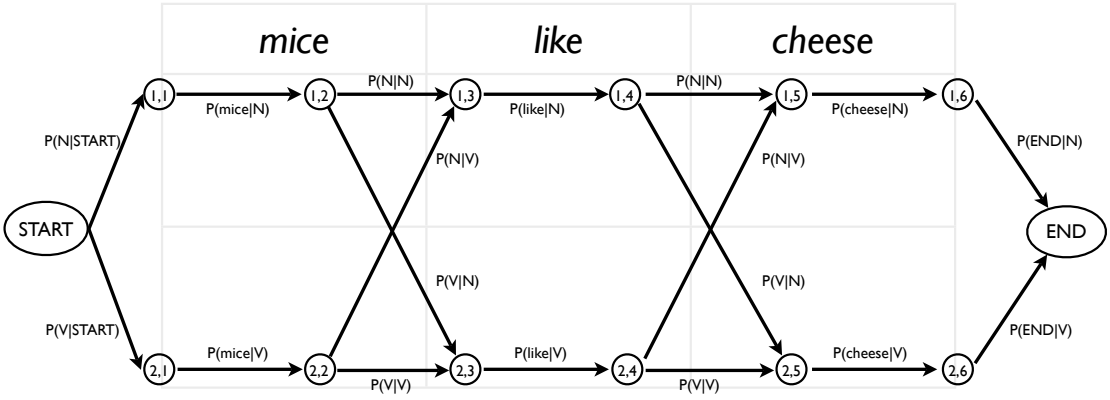

Figure 9: The lattice for the sentence "Mice like cheese" and two possible tags, N and V

Ultimately, we want to learn which tags follow one another, say V usually comes after N, and which words have which tags, e.g., *like* is most of the times a verb, never a noun. Only expressed as probabilities: what is $P(like|V)$? So how do we assign those values? We reward good parameters, i.e., transitions that increase $P(sentence)$, and we decrease bad ones. As a first step, instead of just taking whole counts of how often we see a transition, we "weigh" them by how likely the resulting sentence was ($P(sentence)$). This is called **fractional counts**. fractional counts

QUESTION: What do you think is $P(sentence)$?[9]

We could try to just generate all possible taggings of a sentence (see Figure 10) and count how often we see N following V and "cheese" was tagged as N, and then sum them all up to get $P(sentence)$. But there is a problem...

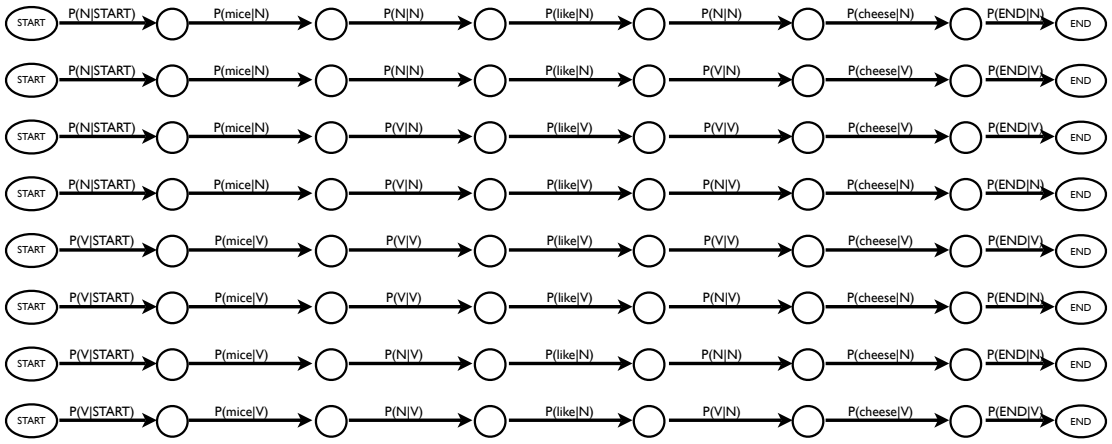

Figure 10: Naive listing of all possible tag sequences

Say each word has on avg. 2.5 tags, and a sentence has 17 words, like this one. That's 2.5^{17}, or $5,820,766$ possible paths through that lattice. For just one sentence! Imagine a sentence with 50 words... Clearly, we cannot afford to do that. We will have to do something else. And that something is **dynamic programming**, see Section 5.2.

dynamic programming

An aside: since we multiply a lot of transitions to get through the lattice, the numbers can quickly become very small. To deal with this, you can use the logarithm of the probabilities.

The smaller a number gets, the larger its negative log will be. Since the range of probabilities is between 0 and 1.0, this corresponds an interval between negative infinity and 0 in log world.

If you do use logarithms, all multiplications shown here become additions (which is slightly faster), and all additions have to be log-additions, a special computation that unfortunately is relatively slow, and works like this:

Algorithm : LOGADDITION(x,y)

$m = \min(x,y)$
$big = 10^{30}$
if $y - x > \log(big)$
  then return (y)
  else if $x - y > \log(big)$
  then return (x)
  else return $(m + \log(\exp(x-m) + \exp(y-m)))$

Figure 11: Adding logs, adapted from Manning/Schütze 2000

In the remainder of the paper, I use "normal" probabilities (not logs) since it would clutter the code.

5.2 Dynamic Programming

Back to our question: we wanted to know what $P(like|V)$ is. This now turns into the question of how likely it is that we end up at the node that has $P(like|V)$ as outgoing transition (in our example node $(2,3)$). And once we took that transition, what is the probability from the node we reach (i.e., $(2,4)$) to the end?

Since we have modeled the HMM as a lattice, we can use dynamic programming techniques. This allows us to compute how likely it is to arrive at each node (with the Forward algorithm), and to get to the end from there (with the Backward algorithm).

S this is where we use the Forward-Backward algorithm! It consists of two parts.

QUESTION: What are the names of those parts?[10]

We use Forward-Backward in order to efficiently compute for each sentence how often we see each transition and what the probability of that sentence is. We need both for the fractional counts. Forward-Backward is the E step in our EM implementation: we compute the expected counts given the current model parameters.

QUESTION: What is the equivalent in clustering?[11]

When you model the lattice, it is a good idea to use a matrix or some such data structure in your implementation, so you can access the nodes directly.

The Forward Algorithm

In the **forward pass**, we compute a new lattice with the same dimensions as the original one, which contains for each node the sum of all possible paths that lead up to there (see Figure 13). These values are also called **alphas**. $\alpha[i,j]$ denotes the probability of all paths up to node (i,j). $\alpha[START]$ is always 1.0. Each subsequent α is just the sum of all transitions arriving there, each multiplied by the α of the node where it (the transition) originated.

forward pass

alphas

$\alpha[END]$ is the sum of all paths through the lattice, which is equal to $P(sentence)$. $P(data)$ is the sum of all $P(sentence)$ in the data. In each iteration, just add up all the $\alpha[END]$ of the sentences. Remember, $P(data)$ has to increase with each iteration, or there is something wrong! EM guarantees that the likelihood of the data increases at each iteration over the data. Outputting $P(data)$ is thus a good way of debugging your code: if it does not increase, something went wrong...

Algorithm : FORWARD(*instance*)

comment: each word has substitution and transition probabilities and thus 2 nodes

comment: the length of the lattice is thus 2 * no of words in the instance

comment: populate first column

for each $j \in tags$

do $\begin{cases} \alpha[j,0] = P(tags[j]|'START') \\ \alpha[j,1] = \alpha[j,0] * P(word1|tags[j]) \end{cases}$

comment: walk the lattice

for $i \in (2,\ |words|-2,\ i+=2)$

do $\begin{cases} \textbf{for each } j \in tags \\ \quad \textbf{do} \begin{cases} \textbf{for each } k \in tags \\ \quad \textbf{do } \{\alpha[j,i] += P(tags[j]|tags[k]) * \alpha[k,i-1] \\ \alpha[j,i+1] = \alpha[j,i] * P(words[i/2]|tags[j]) \end{cases} \end{cases}$

comment: compute $\alpha[END]$

for $j \in tags$

do $\alpha[END] += \alpha[j,|words|-1] * P('END'|tags[j])$

Figure 12: Pseudocode for the Forward algorithm

The Backward Algorithm

The **backward pass** is almost the same as the forward pass, just backwards (note how the direction of the arrows is reversed in Figure 14). Again, we compute a new lattice, which contains for each node the sum of all possible paths that lead from that node to the end. These values are called **betas**. $\beta[i,j]$ denotes the summed probability of all paths from node (i,j) to the end. This time, however, we start at the end. $\beta[END]$ is always 1.0.

backward pass

betas

A useful property for debugging is the fact that $\beta[START] = \alpha[END]$.

QUESTION: Why are they the same?[12]

$$\alpha_{START} = 1.0$$
$$\cdots$$
$$\alpha_{1,3} = \alpha_{1,2} * P(N|N) + \alpha_{2,2} * P(N|V)$$

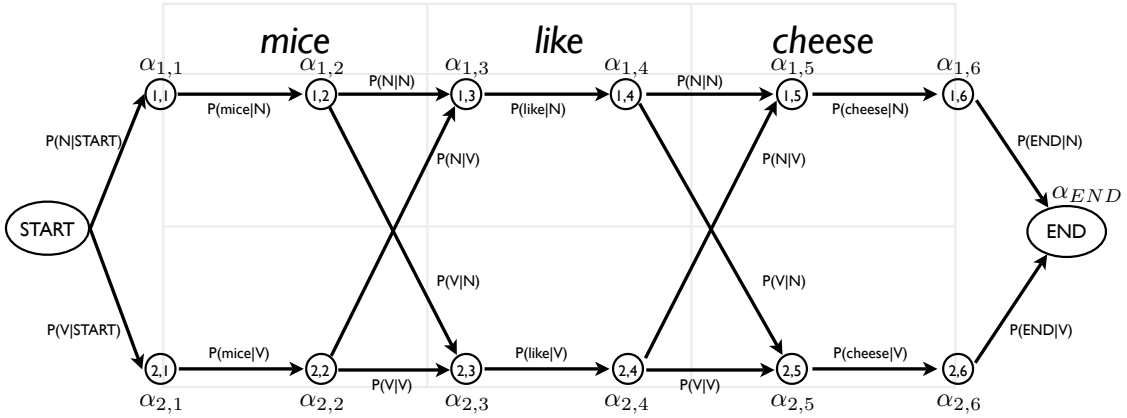

Figure 13: Computing the alphas in the Forward pass

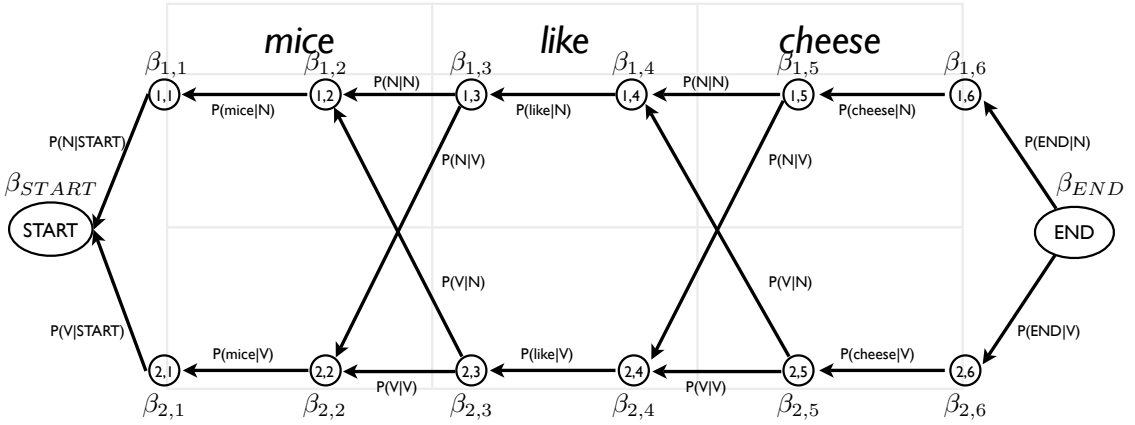

Figure 14: Computing the betas in the Backward pass

Algorithm : BACKWARD(*instance*)

comment: each word has substitution and transition probabilities and thus 2 nodes

comment: the length of the lattice is thus 2 * no of words in the instance

comment: start at the end

for each $j \in tags$

do $\begin{cases} \beta[j, 2*|words|] = P('END'|tags[j]) \\ \beta[j, 2*|words|-1] = \beta[j, 2*|words|] * P(words[last]|tags[j]) \end{cases}$

for $i \in (2*|words|-2,\ 0,\ i-=\ 2)$

do $\begin{cases} \textbf{for each } j \in tags \\ \quad \textbf{do} \begin{cases} \textbf{for each } k \in tags \\ \quad \textbf{do } \beta[j,i] += P(tags[k]|tags[j]) * \beta[k, i+1] \\ \beta[j, i-1] = \beta[j,i] * P(words[i/2]|tags[j]) \end{cases} \end{cases}$

comment: optionally, one can compute $\beta[END]$. It should $= \alpha[START]$

for each $j \in range(noOfTags)$

do $\beta[START] += \beta[j,0] * P(tags[j], 'START')$

Figure 15: Pseudocode for the Backward algorithm

Collecting Fractional Counts

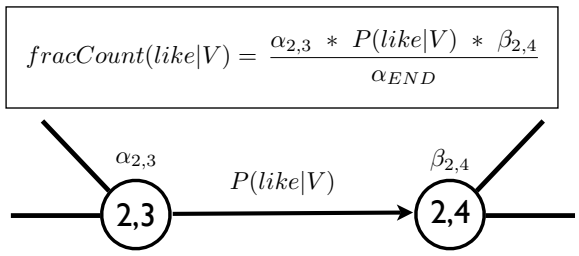

Figure 16: Collecting fractional counts for $P(like|V)$

Once we have the alphas and betas, it is easy to compute for each transition how much it contributes to $P(sentence)$. So, once more, with conviction, how good is $P(like|V)$? Remember, we have to know the likelihood of all possible paths arriving at node $(2,3)$, and the probability – once we have taken the transition – from $(2,4)$ to the end. See Figure 16.

We used the forward algorithm to get the probability of arriving at node $(2,3)$, and the backward algorithm to compute how likely it is from node $(2,4)$ to the end. We divide that by the likelihood of the sentence ($= \alpha[END]$), et voila!

Algorithm : COLLECTCOUNTS()

comment: collect the fractional counts of all transition in the current example

for each $i \in tags$

do
- $counts[tags[i],'START'] += (1.0 * P(tags[i]|'START') * \beta[i,0])/\alpha[END]$
- $counts['END',tags[i]] += (\alpha[i,2*|words|] * P('END'|tags[i]) * 1.0)/\alpha[END]$
- **for each** $j \in words$
  - **do**
    - **comment:** substitution counts
    - $counts[words[j],tags[i]] += (\alpha[i,j*2] * P(words[j]|tags[i]) * \beta[i,(j*2)+1])/\alpha[END]$
    - **do**
      - **comment:** transition counts
      - **for each** $k, \in tags$
        - **do** $counts[tags[k],tags[i]] += (\alpha[i,(j*2)+1] * P(tags[k]|tags[i]) * \beta[k,(j+1)*2])/ >>$ $\alpha[END]$

Figure 17: Pseudocode for collecting fractional counts, '»' denotes line breaks

The M Step

Computing alphas and betas and collecting the fractional counts for all free parameter transitions over all examples is the **E step**. This, as the name suggests, is one half of Forward-Backward EM, and in this case the bigger half. The **M step** is comparatively trivial: after having gone through all the data, we just normalize our fractional counts to get probabilities back (remember, probabilities are normalized counts).

E step

M step

QUESTION: What do you need to compute conditional probabilities from counts?[13]

Algorithm : NORMALIZECOUNTS()

comment: normalize the fractional counts to get probabilities

comment: get total counts for each tag

for each $i \in tags$

do
- $totalTransition['START'] += counts[tags[i],'START']$
- **for each** $j \in tags$
  - **do** $totalTransition[tags[i]] += counts[tags[j],tags[i]]$
- **for each** $j \in lemmas$
  - **do** $totalSubstitution[tags[i]] += counts[words[j],tags[i]]$

comment: divide fractional counts by totals

for each $i \in tags$

do
- $P(tag|'START') = counts[tags[i],'START']/totalTransition['START']$
- **for each** $j \in tags$
  - **do** $P(tags[j]|tags[i]) = counts[tags[j],tags[i]]/totalTransition[tags[i]]$
- **for each** $j \in words$
  - **do** $P(words[j]|tags[i]) = counts[words[j],tag[i]]/totalSubstitution[tags[i]]$

Figure 18: Pseudocode for normalizing counts to get the new parameters

5.3 Putting It All Together

If we put the E step and the M step together, we end up with the Forward-Backward EM algorithm!

Algorithm : FORWARD-BACKWARDEM()

while not *converged*

do
- $dataLikelihood = 0.0$
- **for each** $instance \in instances$
  - **do**
    - FORWARD($instance$)
    - BACKWARD($instance$)
    - COLLECTCOUNTS()
    - $dataLikelihood += \alpha[END]$
- NORMALIZECOUNTS()

Figure 19: Pseudocode for the Forward-Backward EM algorithm

The convergence criterion in this case is how much the data likelihood $P(data)$ has improved since the last iteration. Once it starts to flatten out, we can assume that we reached a maximum on our curve and stop. It is also customary to set a maximum number of iterations (50) and stop even if EM has not converged, to avoid overfitting.

And that's it. No magic, no silver bullets, just counting and normalizing. EM will adjust all the free parameters to get the maximum data likelihood, and you can then use those probabilities to label data using the Viterbi algorithm.

5.4 Example Run

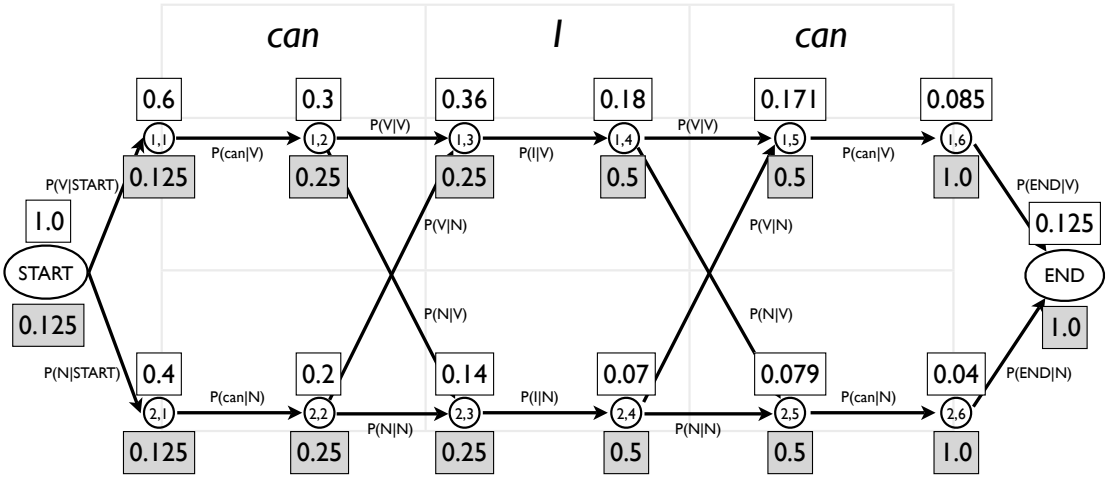

Figure 20: α and β values for example run

If you want to check whether your implementation is working, here is a little toy example. Let's take the tags "N" and "V", and the words *can* and *I* and initialize our model with the following numbers (this example is based on one used by Kevin Knight.)

Transitions:

$$P(V|V) = 0.6$$
$$P(N|V) = 0.4$$
$$P(V|N) = 0.9$$
$$P(N|N) = 0.1$$
$$P(V|START) = P(V) = 0.6$$
$$P(N|START) = P(N) = 0.4$$

Emissions:

$$P(can|V) = 0.5$$

$$P(I|V) = 0.5$$
$$P(can|N) = 0.5$$
$$P(I|N) = 0.5$$

As our only observed instance we use the sentence "can I can". Your αs and βs after the first iteration should look like they do in Figure 20, where the α values for each node are shown on a white background above the nodes, and β values with a grey background below the nodes. The arrows point forward, but you can safely ignore them.

6 Finally...

CONGRATULATIONS! You have made it through. You now know EM and can go and try it out...

You do not have to be constrained to HMMs with hidden inputs, like we used here. Maybe you do not have sequential data. In that case the HMM becomes a Bayesian network (which is an HMM without the transitions).

Maybe you know both the inputs and outputs, but not the hidden variable X that connects them . In that case you just want to know the conditional probabilities $P(X|input)$ and $P(output|X)$ (in our diary example above, you might also have the weather reports from that time and want to compute the traffic probabilities). EM can help you in those cases, too.

If you want to apply EM to other problems, always start with the inputs and outputs. What are the observed variables? Are the inputs hidden, or can they be observed, too? Is it a sequential model or a network? Write down the joint probability $P(x,y)$ and see how you can factor it. Drawing a graphical model helps.

Think about how you can express the parameters as conditional probabilities. Do you need additional (hidden) variables? What is your E step, what is your M step, what is the data you iterate over? Once you have figured all this out, you can start to look into implementations. Do not focus on implementation details like Forward-Backward in the beginning.

7 Troubleshooting

There are some problems that can arise, and it is good to be aware of them and know how to deal with, or, better, avoid them.

Problem: What we maximize – $P(data)$ – is often not what we want to evaluate (say, the tagging accuracy as compared to gold data)

Solution: Design your model with this in mind and try to formulate the problem so that the two criteria are similar. Keep test data around and evaluate your models on it.

Problem: **Local maxima**. EM improves $P(data)$ at each iteration, but can get stuck in local maxima. Remember the graph from section 4: there are many suboptimal parameter configurations at which EM stops because it can not improve from there. The result is not the best model, though.

Local maxima

Solution: Restart EM 50 times or more with random initializations, and remember the model that got the best data likelihood. Each time you restart, you start at a different point along the curve, and hopefully eventually at one that leads to the global optimum.

Problem: EM does not care about semantics! Whether a label makes sense or not is irrelevant, as long as it explains the data! It uses this **weird tag** that only occurs once or twice, like FW (foreign word). weird tag

Solution: Use a dictionary that constrains EM's $P(w|t)$ choices to possible ones (all others will be 0). If you do completely unsupervised training (no dictionary), the labels become meaningless, and the task is more like clustering. Each label becomes a cluster. In this case, you have to afterwards map each of the clusters to a tag in order to label data and get the accuracy. One mapping is **many-to-1** (see Johnson 2007). Also, keep your data clean. EM wants to use everything. Yes, also that weird tag that got in by accident... By assigning it to a frequent word, EM actually even thinks it has done a good job. many-to-1

Problem: EM uses unlikely tags too frequently, i.e. we have a rather **flat label distribution**. In language, however, most probability mass should go to one or two cases, the rest becomes less and less likely, i.e., we have a Zipfian distribution. flat label distribution

Solution: Use smoothing and a dictionary. Additionally, you might want to try techniques like L_0 normalization Vaswani/Pauls/Chiang 2010 or Bayesian inference with Gibbs sampling and sparse priors.

Figure 21: Label distribution in EM and in reality

Problem: EM changes good **initial parameters** to make them worse. initial parameters

Solution: Fix as many parameters as you can! That will guide EM to only optimize the right things. Also, add pseudo-counts before normalization to make changes less dramatic. The higher the pseudo-counts, the smaller the normalization effect. You can see in Table 1 how the normalized values of two parameters change when using different pseudo-counts. If you do not want to fix anything random restarts can help to find a good starting point.

Parameter	fractional counts	pseudo-count	total count t	resulting probabilities
(x|t) (y|t)	1.1 3.5	0.0	4.6	P(x|t) = 1.1/4.6 = 0.24 P(y|t) = 3.5/4.6 = 0.76
		0.5	5.6	P(x|t) = 1.6/5.6 = 0.29 P(y|t) = 4.0/5.6 = 0.71
		1.0	6.6	P(x|t) = 2.1/6.6 = 0.32 P(y|t) = 4.5/6.6 = 0.68

Table 1: Influence of pseudo-counts

Problem: The resulting model is **overfitting** the training data ("Look, I can explain this data perfectly! And nothing else. . . "). overfitting

Solution: Again, use smoothing (add n to the fractional counts before normalizing) and stop after a maximum amount of iterations (usually 50). This will prevent the data from being exactly modeled.

Problem: **Ties**. All transition options leaving from one node are equally good. EM doesn't take a stance and just leaves all of them as is. Ties

Solution: Start out randomly to avoid ties, and do restarts. This way, you can break the ties.

8 Useful Reading

If you want to read the original, go for Dempster/Laird/Rubin (1977).

Rabiner/Juang (1986) is a general overview over parameter estimation, but very math-heavy. Manning/Schütze (2000) has a chapter on EM, based on clustering, but with an eye on other NLP applications. One of the most famous implementations for NLP is the tagging paper by Merialdo (1994), which also gives you a good idea about the various parameters you can set.

If you want to know more about the Forward-Backward procedure of calculating alphas and betas, check out Jason Eisner's tutorial (Eisner, 2002), including a spreadsheet with the changing values.

The second edition of the AI handbook (Russell/Norvig, 2003, 724 – 733) has a comprehensive section about EM.

Kevin Knight has written a very compelling introduction (Knight, 2009a). It is mainly about Bayesian Inference, but explains EM very nicely. You might also want to check out his tutorials for Carmel (Knight, 2009b), a software that helps you implement graphical models as automata and train them with EM. Also, the workbook for MT (Knight, 1999) contains a useful section on EM.

9 Further...

Vaswani/Pauls/Chiang 2010 show how using L_0 **normalization** can lead to smaller models and a sparser distribution, which improves language related tasks a lot, because it creates a more Zipfian distribution (see Hovy et al. 2011 for another application). L_0 normalization

If you want to get deeper into the matter, you could look into **EM with features** (Berg-Kirkpatrick et al., 2010). Instead of just using conditional probabilities, which often cannot capture EM with features

useful properties (or only when encoded as additional states), you can add all the features you like in discriminative models (like word suffixes or capitalization) and still do unsupervised learning.

Or explore **structural EM**, which not only learns the parameter values, but also how many parameters there should be. structural EM

If you do any of these things, or if you have questions, comments, or criticism, send me a mail—I'd be dead curious to know!

Acknowledgements

Thanks to Kevin Knight and David Chiang for the basics, Congxing Cai, Karl-Moritz Hermann, and Eduard Hovy for useful comments, and Ashish Vaswani, Victoria Fossum, and Taylor Berg-Kirkpatrick for many enlightening discussions.

References

BERG-KIRKPATRICK, TAYLOR ET AL. (2010): *Painless Unsupervised Learning with Features*. In: North American Chapter of the Association for Computational Linguistics..

BISHOP, C.M. (2006): *Pattern recognition and machine learning*. New York: Springer.

DEMPSTER, ARTHUR P./LAIRD, NAN M./RUBIN, DONALD B. (1977): *Maximum likelihood from incomplete data via the EM algorithm*. In: Journal of the Royal Statistical Society. Series B (Methodological), 39, Nr. 1, 1–38.

EISNER, JASON (2002): *An interactive spreadsheet for teaching the forward-backward algorithm*. In: Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1. Association for Computational Linguistics, 10–18.

HOVY, DIRK ET AL. (2011): *Unsupervised Discovery of Domain-Specific Knowledge from Text*. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies. Portland, Oregon, USA: Association for Computational Linguistics ⟨URL: `http://www.aclweb.org/anthology/P11-1147`⟩, 1466–1475.

JOHNSON, MARK (2007): *Why doesn't EM find good HMM POS-taggers*. In: Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)., 296–305.

KNIGHT, K. (1999): *A statistical MT tutorial workbook*. JHU Summer Workshop, 1999.

KNIGHT, K. (2009a): *Bayesian Inference with Tears*. No address in, http://www.isi.edu/natural-language/people/bayes-with-tears.pdf.

KNIGHT, K. (2009b): *Training Finite-State Transducer Cascades with Carmel*. 2009b.

MANNING, C.D./SCHÜTZE, H. (2000): *Foundations of statistical natural language processing*. MIT Press.

MERIALDO, BERNARD (1994): *Tagging English text with a probabilistic model*. In: Computational linguistics, 20, Nr. 2, 155–171.

RABINER, L./JUANG, B. (1986): *An introduction to hidden Markov models*. In: IEEE ASSp Magazine, 3, Nr. 1 Part 1, 4–16.

RUSSELL, S.J./NORVIG, P. (2003): *Artificial intelligence: a modern approach*. 2nd edition. Upper Saddle River, NJ: Prentice Hall.

VASWANI, ASHISH/PAULS, ADAM/CHIANG, DAVID (2010): *Efficient optimization of an MDL-inspired objective function for unsupervised part-of-speech tagging*. In: Proceedings of the ACL 2010 Conference Short Papers. Association for Computational Linguistics, 209–214.

A Python Implementation

```
import scipy as sp
from collections import defaultdict
import time

sp.set_printoptions( precision = 3 )

# CONSTANTS
NINF = float('-1e303')
LINF = sp.log(NINF)
SMOOTHING = sp.log(0.1)
HARDNESS = 1.0/1.0
THRESHOLD = 0.00001
MAX_NUM_ITERATIONS = 40
logadd = sp.logaddexp

# parameters
start = defaultdict(lambda: LINF)
# P(x|y) = emissions[y][x] - this allows to sum over all values given y
emissions = defaultdict(lambda: defaultdict(lambda: LINF))
transitions = defaultdict(lambda: defaultdict(lambda: LINF))

# initialize
start['V'] = sp.log(0.6)
start['N'] = sp.log(0.4)

emissions['V']['I'] = sp.log(0.5)
emissions['V']['can'] = sp.log(0.5)
emissions['N']['I'] = sp.log(0.5)
emissions['N']['can'] = sp.log(0.5)

transitions['V']['V'] = sp.log(0.6)
transitions['V']['N'] = sp.log(0.4)
transitions['N']['N'] = sp.log(0.1)
transitions['N']['V'] = sp.log(0.9)

tags = ['V', 'N']

examples = [
    ['can', 'I', 'can'],
    [ 'I', 'can', 'can']
]

```

```
43
44 converged = False
45 old_data_likelihood = sp.log(0.0)
46 iteration = 1
47
48 # repeat until convergence
49 # (i.e., difference of data-log likelihood < THRESHOLD or MAX_NUM_ITERATIONS)
50 while not converged:
51     print '=' * 20
52     print "iteration %s" % (iteration)
53     print '=' * 20
54     data_likelihood = sp.log(1.0)
55
56     # clear counts
57     start_counts = defaultdict(lambda: SMOOTHING)
58     emission_counts = defaultdict(lambda: defaultdict(lambda: SMOOTHING))
59     transit_counts = defaultdict(lambda: defaultdict(lambda: SMOOTHING))
60
61     # go through examples
62     for example in examples:
63         starttime = time.time()
64         N = len(example) * 2
65         M = len(tags)
66
67         ################
68         # forward pass #
69         ################
70         # alpha[i,j] = the probability of ending up with tag i at word j
71         alpha = sp.ones((M,N)) * NINF
72
73         for i, tag in enumerate(tags):
74             if tag in start:
75                 alpha[i][0] = start[ tag ]
76                 if example[0] in emissions[tag]:
77                     alpha[i][1] = alpha[i][0] + emissions[tag][ example[0] ]
78
79         for j in range(2, N, 2):
80             for i, tag1 in enumerate(tags):
81                 for k, tag2 in enumerate(tags):
82                     if tag1 in transitions[tag2]:
83                         # alpha[i,j] += P(t1|t2) * alpha[k, j-1]
84                         if alpha[i, j] == NINF:
85                             alpha[i,j] = transitions[tag2][tag1] + alpha[k,j-1]
86                         else:
87                             alpha[i,j] = logadd(alpha[i][j],
```

```
                                                    transitions[tag2][tag1]+alpha[k,j-1])
            if example[j/2] in emissions[tag1]:
                # alpha[i, j+1] = alpha[i, j] * P(word_j|tag1)
                alpha[i,j+1] = alpha[i,j] + emissions[tag1][ example[j/2] ]

    print "ALPHA:"
    print sp.exp(alpha)

    #################
    # backward pass #
    #################
    beta = sp.ones((M,N)) * NINF

    # initialize beta from the back
    for i, tag in enumerate(tags):
        beta[i][-1] = sp.log(1.0)
        if example[-1] in emissions[tag]:
            beta[i][-2] = beta[i][-1] + emissions[tag][ example[-1] ]

    for j in range(N-3,0,-2):
        for i, tag1 in enumerate(tags):
            for k, tag2 in enumerate(tags):
                if tag2 in transitions[tag1]:
                    if beta[i,j] == NINF:
                        beta[i,j] = transitions[tag1][tag2] + beta[k,j+1]
                    else:
                    beta[i,j] = logadd(beta[i,j],
                                            transitions[tag1][tag2] + beta[k,j+1])
            if example[j/2] in emissions[tag1]:
                beta[i, j-1] = beta[i, j] + emissions[tag1][example[j/2]]

    print "BETA:"
    print sp.exp(beta)
    print


    # sum of all paths through example
    example_likelihood = reduce(logadd, alpha[:, -1])
    print "example likelihood: ", sp.exp(example_likelihood)

    # add example likelihood to data likelihood
    data_likelihood += example_likelihood


    ##############################
```

```
133         # collect fractional counts #
134         ############################
135         for i, tag1 in enumerate(tags):
136             if tag1 in start:
137                 # if no smoothing, start from scratch, otherwise just add up
138                 if start_counts[tag1] == sp.log(0.0):
139                     start_counts[tag1] = start[tag1] + beta[i, 0] - example_likelihood
140                 else:
141                     start_counts[tag1]=logadd(start_counts[tag1],
142                                               start[tag1]+beta[i,0]-example_likelihood)
143 
144             for j, word in enumerate(example):
145                 if word in emissions[tag1]:
146                     if emission_counts[tag1][word] == sp.log(0.0):
147                         emission_counts[tag1][word] = alpha[i, j*2] +
148                                                       emissions[tag1][word] +
149                                                       beta[i, (j*2)+1] -
150                                                       example_likelihood
151                     else:
152                         emission_counts[tag1][word]=logadd(emission_counts[tag1][word],
153                                                            alpha[i, j*2] +
154                                                            emissions[tag1][word] +
155                                                            beta[i, (j*2)+1] -
156                                                            example_likelihood)
157 
158                 for k, tag2 in enumerate(tags):
159                     if j < len(example)-1 and tag2 in transitions[tag1]:
160                         if transit_counts[tag1][tag2] == sp.log(0.0):
161                             transit_counts[tag1][tag2] = alpha[i, j*2+1] +
162                                                          transitions[tag1][tag2] +
163                                                          beta[k, (j+1)*2] -
164                                                          example_likelihood
165                         else:
166                             transit_counts[tag1][tag2]=logadd(transit_counts[tag1][tag2],
167                                                          alpha[i, j*2+1] +
168                                                          transitions[tag1][tag2] +
169                                                          beta[k, (j+1)*2] -
170                                                          example_likelihood)
171 
172         print "%.5fsec" % (time.time()-starttime)
173     print
174 
175     ####################
176     # normalize counts #
177     ####################
```

```
    start_count_total = reduce(logadd, start_counts.values())
    print "New START probabilities:"
    print '-' * 20
    for tag in start:
        start[tag] = start_counts[tag] - start_count_total
        print 'P(%s) = %.2f' % (tag, sp.exp(start[tag]))

    print "New EMISSION probabilities:"
    print '-' * 20
    for tag, words in emissions.iteritems():
        emission_tag_total = reduce(logadd, emission_counts[tag].values())
        for word in words:
            emissions[tag][word] = emission_counts[tag][word] - emission_tag_total
            print 'P(%s|%s) = %.2f' % (word,tag, sp.exp(emissions[tag][word]))

    print "New TRANSITION probabilities:"
    print '-' * 20
    for tag, tag_successors in transitions.iteritems():
        transition_tag_total = reduce(logadd, transit_counts[tag].values())
        for tag_successor in tag_successors:
            transitions[tag][tag_successor] = transit_counts[tag][tag_successor]
                                              - transition_tag_total
            print 'P(%s|%s) = %.2f' % (tag_successor,
                                       tag,
                                       sp.exp(transitions[tag][tag_successor]))

    print

    print "CHANGE:"
    print 'old: %.5f (%s)' % (sp.exp(old_data_likelihood), old_data_likelihood)
    print 'new: %.5f (%s)' % (sp.exp(data_likelihood), data_likelihood)
    change = -(sp.exp(old_data_likelihood) - sp.exp(data_likelihood))
    print "change: ",change
    assert change > 0.0
    if change <= THRESHOLD or iteration > MAX_NUM_ITERATIONS:
        converged = True

    old_data_likelihood = data_likelihood
    iteration += 1
```

Notes

[1]ANSWER: $P(x|y) = \frac{P(x,y)}{P(y)}$

[2]ANSWER: W is discrete, T is discrete, but binary, and L is boolean, and thus also binary.

[3]ANSWER: We count the occurrences of each rule $S \rightarrow NP\ VP$ in a treebank and normalize by the number of times we have seen S.

[4]ANSWER: Exactly, CKY parsing... You are getting good at this!

[5]ANSWER: We compute the distance between the cluster centroids and each point and assign it to the closest centroid.

[6]ANSWER: We divide $P(t) * P(w|t) = P(w) * P(t|w)$ by $P(w)$

[7]ANSWER: You already had your model and would not need EM...

[8]ANSWER: Because words tend to be ambiguous. What is the most likely tag for "can"? It depends. It could be NOUN, VERB, or AUX. But what is the most likely tag if we see "the can"? Still VERB or AUX? Probably not... The context disambiguates it. That's why we want the transition probabilities.

[9]ANSWER: The probability that we end up with this sentence if we ran our model in generation mode.

[10]ANSWER: Well, Forward and Backward... What did you think? Though you could of course call Forward "Baum" and Backward "Welch", but that seems a little unfair to Mr. Welch.

[11]ANSWER: Assigning each data point to one of the centroids (= the current model parameters).

[12]ANSWER: Because in both cases we walked through all possible paths of the lattice, summing them up.

[13]ANSWER: The count of the two events together and the counts of the given part.