Chapter 3

Sequence Labeling and HMMs

3.1 Introduction

A *sequence-labeling problem* has as input a sequence of length n (where n can vary) $\boldsymbol{x} = (x_1, \ldots, x_n)$ and the output is another sequence $\boldsymbol{y} = (y_1, \ldots, y_n)$, also of length n, where each $y_i \in \mathcal{Y}$ is the "label" of x_i. Many interesting language-processing tasks can be cast in this framework.

Notation: Using $\boldsymbol{x}$ for the input sequence and $\boldsymbol{y}$ for the label sequence is fairly standard. Frequently the $\boldsymbol{x}$ is a sequence of words. In this case we may refer to the words as a *terminal* sequence as we do in parsing (see Section 4.1.1).

Part-of-speech tagging (abbreviated POS tagging): Each x_i in $\boldsymbol{x}$ is a word of the sentence, and each y_i in $\boldsymbol{y}$ is a *part of speech* (e.g., '*NN*' is common noun, '*JJ*' is adjective. etc.).

$\boldsymbol{y}$:	DT	JJ	NN	VBD	NNP	.
$\boldsymbol{x}$:	the	big	cat	bit	Sam	.

Noun-phrase chunking: Each x_i is a word in the sentence and its corresponding y_i indicates whether x_i is in the beginning, middle or end of a noun phrase (NP) chunk.

$\boldsymbol{y}$:	[NP	NP	NP]	_	[NP]	.
$\boldsymbol{x}$:	the	big	cat	bit	Sam	.

In this task, '*[NP*' labels the beginning of a noun phrase — the notation is intended to conveys the intuitive idea that the labeled word is the start of an NP. Similarly, '*[NP]*' labels a word that is both the start and end of a noun phrase.

Named entity detection: The elements of $\boldsymbol{x}$ are the words of a sentence, and $\boldsymbol{y}$ indicates whether they are in the beginning, middle or end of a noun phrase (NP) chunk that is the name of a person, company or location.

$\boldsymbol{y}$:	[CO	CO]	_	[LOC]	_	[PER]	_
$\boldsymbol{x}$:	XYZ	Corp.	of	Boston	announced	Spade's	resignation

Speech recognition: The elements of $\boldsymbol{x}$ are 100 msec. time slices of acoustic input, and those of $\boldsymbol{y}$ are the corresponding phonemes (i.e., y_i is the phoneme being uttered in time slice x_i). A *phoneme* is (roughly) the smallest unit of sound that makes up words.

In this chapter we introduce *hidden Markov models* (HMMs), a very elegant technique for accomplishing such tasks. HMMs were first used for speech recognition where i is a measure of time. Thus it is often the case that HMMs are thought of as marching through time — a metaphor we occasionally use below.

3.2 Hidden Markov models

Recall (ordinary) Markov models. A Markov model (e.g., a bigram model) generates a string $\boldsymbol{x} = (x_1, \ldots, x_n)$. As in Chapter 1, we imagine that the string is padded with a start symbol (or *start marker* $x_0 = \triangleright$ and an stop symbol (or marker) $x_{n+1} = \triangleleft$.

$$\begin{aligned} \mathrm{P}(\boldsymbol{x}) &= \prod_{i=1}^{n+1} \mathrm{P}(x_i \mid x_{i-1}) \\ &= \prod_{i=1}^{n+1} \Phi_{x_{i-1},x_i} \end{aligned}$$

Here $\Phi_{x,x'}$ is a parameter of the model specifying the probability that x is followed by x'. As before, note the Markov assumption that the next word depends only on the previous word.

In a *hidden* Markov model (HMM) we observe a string (or *observation sequence*$\boldsymbol{x}$, but in general its *label sequence* $\boldsymbol{y}$ is hidden (not observed). Just as in the Markov model above, we imagine that the label sequence $\boldsymbol{y}$ is padded with begin marker $y_0 = \triangleright$ and end marker $y_{n+1} = \triangleleft$. A HMM is a generative model that jointly generates both the label sequence $\boldsymbol{y}$ and the observation sequence $\boldsymbol{x}$. Specifically, the label sequence $\boldsymbol{y}$ is generated by a Markov model. Then the observations $\boldsymbol{x}$ are generated from the $\boldsymbol{y}$.

$$\begin{aligned}
\mathrm{P}(\boldsymbol{y}) &= \prod_{i=1}^{n+1} \mathrm{P}(y_i \mid y_{i-1}) \\
&= \prod_{i=1}^{n+1} \sigma_{y_{i-1},y_i} \\
\mathrm{P}(\boldsymbol{x}|\boldsymbol{y}) &= \prod_{i=1}^{n+1} \mathrm{P}(x_i \mid y_i) \\
&= \prod_{i=1}^{n+1} \tau_{y_i,x_i}
\end{aligned}$$

Notation: We use $\sigma_{y,y'}$ for the parameter estimating the probability that label y is followed by label y' and $\tau_{y,x}$ for the probability that label y generates output x. (Think of σ as state-to-*state* transition and τ as a state-to-*terminal* transition.)

We combine these two formulae as follows:

$$\begin{aligned}
\mathrm{P}(\boldsymbol{x},\boldsymbol{y}) &= \mathrm{P}(\boldsymbol{y})\,\mathrm{P}(\boldsymbol{x} \mid \boldsymbol{y}) \\
&= \prod_{i=1}^{n+1} \sigma_{y_{i-1},y_i}\ \tau_{y_i,x_i} \qquad (3.1)
\end{aligned}$$

So the generative story for an HMM goes like this: generate the next label y_i with probability $\mathrm{P}(y_i \mid y_{i-1})$ and then the next member of the sequence x_i with probabillity $\mathrm{P}(x_i \mid y_i)$.

In our study of HMMs we use three different visualizations. The first is the *Bayes-net representation* shown in Figure 3.1. In a Bayes net, the nodes are random variables and the edges between them indicate dependence relations. If we go back to the time-step metaphor for HMMs, this diagram can be thought of as follows: at each time i the HMM transitions between $Y_{i-1} = y$ and $Y_i = y'$, where the probability of the event is $\sigma_{y,y}$. The top row

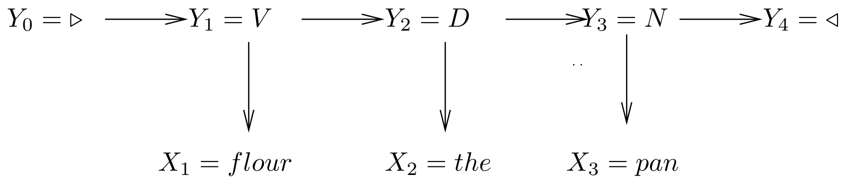

Figure 3.1: The Bayes-net representation of an HMM generating '*flour the pan*' from the labels '*V D N*'.

of arrows in Figure 3.1 indicates this dependence. Then (during the same time step) the HMM generates x_i according to the probability distribution for y_i. The row of downward arrows indicates this dependence.

A second representation for HMMs is that used to diagram *probabilistic automata*, as seen in Figure 3.2. The Bayes net representation emphasizes what is happening over time. In contrast, the automata representation can be thought of as showing what is going on "inside" the HMM "machine". That is, when looking at the automaton representation we refer to the label values as the *states* of the HMM. We use m for the number of states. The edges between one state and the next are labeled with the corresponding σ values. The state labeled with the beginning of sentence symbols $\triangleright$ is the initial state.

So in Figure 3.2 the edge between N and V has the probability of going from the first of these states to the second, $\sigma_{N,V}$, which is 0.3. Once we reach a new state we generate the next visible symbol with some probability associated with the state. In our automata notation the probabilities of each output symbol are written inside the state. So in our figure the probability of generating '*flour*' from the V state is 0.2.

Example 3.1: What is the probability of the sequence '*flour pan*' when the state sequence is $< \triangleright, V, N \triangleleft >$? (So '*flour pan*' is a command to coat the pan with flour.) That is, we want to compute

$$\mathrm{P}(< flour, pan >, < \triangleright, V, N, \triangleleft >)$$

From Equation 3.1 we see that

$$\begin{aligned} \mathrm{P}(< flour, pan >, < V, N, \triangleleft >) &= \sigma_{\triangleright,V}\ \tau_{V,flour}\ \sigma_{V,N}\ \tau_{N,pan}\ \sigma_{N,\triangleleft} \\ &= 0.3 \cdot 0.2 \cdot 0.3 \cdot 0.4 \cdot 0.4. \end{aligned}$$

It should be emphasised that here we are computing the *joint* probability of the labeling and the string.

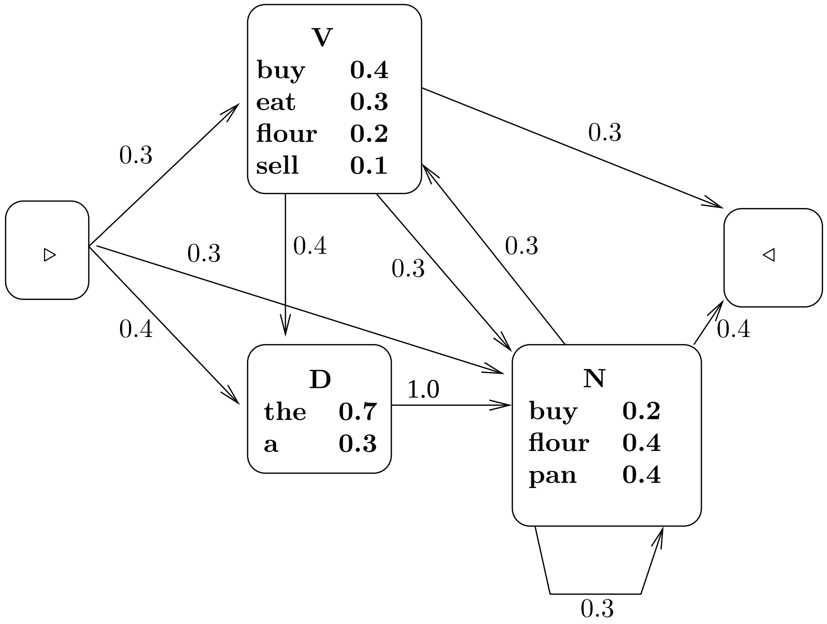

Figure 3.2: Example of HMM for POS tagging '*flour pan*', '*buy flour*'

The third of our visual representations is the *trellis representation.* The Bayes net representation shows what happens over time, and the automata representation shows what is happening inside the machine. The trellis representation shows both. Here each node is a pair (y, i) where $y \in \mathcal{Y}$ is a hidden label (or state) and $i \in 0, \ldots, n+1$ is a position of a word. So at the bottom of Figure 3.3 we have the sequence of words going from left to right. Meanwhile we have (almost) a separate row for each state (label). (We let the start and stop states share a row.) The edge from $(y, i-1)$ to (y', i) has weight $\sigma_{y,y'}\, \tau_{y',x_i}$, which is the probability of moving from state y to y' and emitting x_i.

In the sections that follow we look at several algorithms for using HMMs. The most important, *Viterbi decoding*, comes first. The Viterbi algorithm finds the most probable sequence of hidden states that could have generated the observed sequence. (This sequence is thus often called the *Viterbi labeling.*) The next two, which find the total probability of an observed string according to an HMM and find the most likely state at any given point, are less useful. We include them to motivate two very important ideas: forward and backward probabilities. In the subsequent section, the use of forward

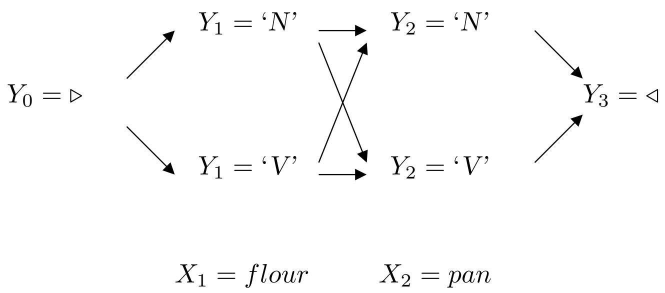
$Y_1 = \text{'}N\text{'}$
$Y_2 = \text{'}N\text{'}$
$Y_0 = \triangleright$
$Y_3 = \triangleleft$
$Y_1 = \text{'}V\text{'}$
$Y_2 = \text{'}V\text{'}$
$X_1 = flour$
$X_2 = pan$

Figure 3.3: The trellis representation of an HMM generating '*flour pan*'

and backward probabilities is required for adapting the EM algorithm to HMMs.

3.3 Most likely labels and Viterbi decoding

In our introduction to HMMs we saw that many interesting problems such as POS tagging can be thought of as labeling problems. We then introduced HMMs as a way to represent a labeling problem by associating, probabilistically, a label (or state) Y_i with each input X_i. However, actually to *use* an HMM for, say, POS tagging, we need to solve the following problem: given an an HMM $(\boldsymbol{\sigma}, \boldsymbol{\tau})$ and an observation sequence $\boldsymbol{x}$, what is the *most likely* label sequence $\hat{\boldsymbol{y}}$?

$$\begin{aligned} \hat{\boldsymbol{y}} &= \operatorname*{argmax}_{\boldsymbol{y}} \mathrm{P}(\boldsymbol{y} \mid \boldsymbol{x}) \\ &= \operatorname*{argmax}_{\boldsymbol{y}} \mathrm{P}(\boldsymbol{x}, \boldsymbol{y}) \end{aligned}$$

Simply put, the problem of POS tagging is: given a sequence of words, find most likely tag sequence.

In principle we could solve this by enumerating all possible $\boldsymbol{y}$ and finding the one that maximizes $\mathrm{P}(\boldsymbol{x}, \boldsymbol{y})$. Unfortunately, the number of possible $\boldsymbol{y}$ grows *exponentially* with length of sentence n. Assume for the sake of argument that every word in English has two and only two possible tags. Then a string of length one has two possible sequences, a sequence of two words has $2{\cdot}2$ possible state sequences, and a sequence of n words has 2^n state sequences. One of your authors looked at the *New York Times* on the day he wrote this paragraph and the first sentence of the lead article had 38 words. 2^{38} is approximately 10^{12}, a trillion. As you can imagine, the naive algorithm is not a practical option.

The solution is *dynamic programming.* Dynamic programming is the technique of saving partial solutions to problems to avoid computing them over and over again. The particular algorithm is the Viterbi Algorithm, discovered by Andrew Viterbi in 1967. It is often called the Viterbi *decoder* for much the same reason that doing actual machine translation is called "decoding".

To begin, let's solve a simpler problem, finding the probability $\mathrm{P}(\boldsymbol{x}, \hat{\boldsymbol{y}})$ of the most likely solution $\hat{\boldsymbol{y}}$.

Notation: $\boldsymbol{y}_{i,j} = (y_i, \ldots, y_j)$ is the subsequence of $\boldsymbol{y}$ from y_i to y_j.

As we do frequently in both HMM's and (in the next chapter) probabilistic context-free grammars, we need to solve a slightly more general probelm — finding the probability of the most likely solution for the prefix of $\boldsymbol{x}$ up to position i that ends in state y,

$$\mu_y(i) = \max_{\boldsymbol{y_{1,i}}} \mathrm{P}(x_{1,i}, Y_i = y). \tag{3.2}$$

The basic idea is that we compute the $\mu_y(i)$s starting on left and working our way to the right. At the far left,

$$\mu_{\triangleright}(0) = 1.0 \tag{3.3}$$

That is, the maximum probability of "ending up" at the start state at time zero is 1.0 (since there is no other option).

We next go from time $i-1$ to i as follows:

$$\mu_y(i) = \max_{y'=1}^{m} \mu_{y'}(i-1)\sigma_{y',y}\tau_{y,x_i} \tag{3.4}$$

Eventually we will derive this, but first let's get a feeling for why it is true. Figure 3.4 shows a piece of a trellis in detail. In particular we are looking at the possible transitions from the m states at time $i-1$ to the particular state y at time i. At the bottom we show that the HMM outputs are $X_{i-1} = x_{i-1}$ and $X_i = x_i$. Each of the arcs is labeled with the state-to-state transition probability $\sigma_{y_i,y}$.

Each of the Y_{i-1} is labeled with the maximum probability the string could have and end up in that state, $\mu_y(i-1)$. These have already been computed. Now, the label subsequence of maximum probability at time i must have come about by first going through, say, $Y_{i-1} = y'$ and then

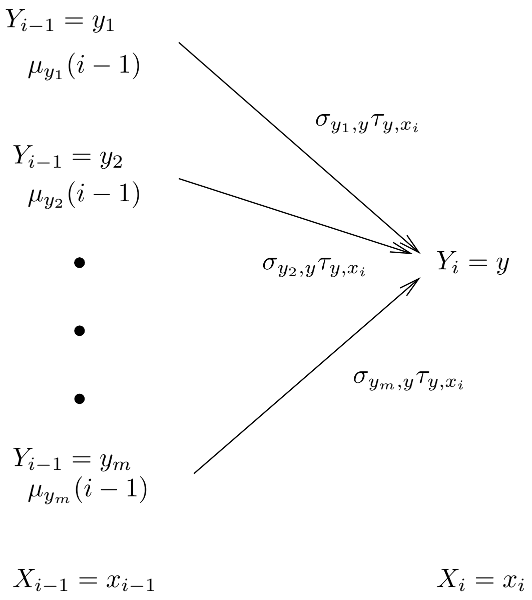

Figure 3.4: A piece of a trellis showing how to compute $\mu_y(i)$ if all of the $\mu_{y'}(i-1)$ are known.

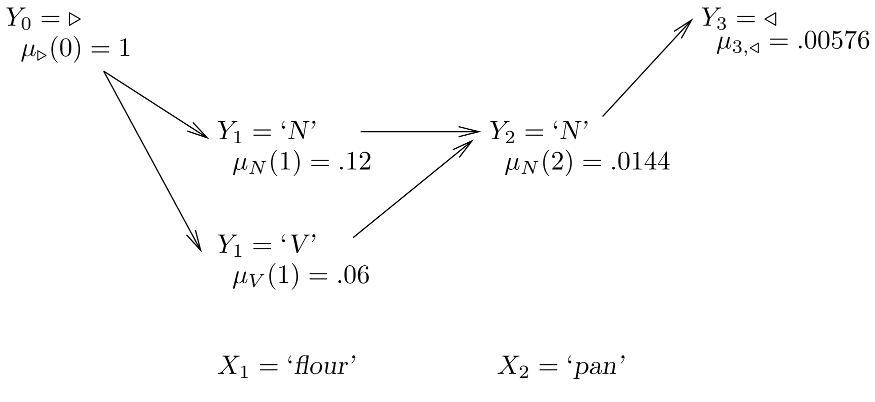

Figure 3.5: The trellis for ‘*flour pan*’ showing the $\mu_y(i)$’s

ending up at y_i. Then for this path the probability at time i is the maximum probability at $Y_{i-1} = y'$, times the transition probability to state y. Thus Equation 3.4 says to find the $Y_{i-1} = y'$ that maximizes this product.

Example 3.2: Figure 3.5 shows the (almost) complete $\mu_y(i)$ computation for the '*flour pan*' sentence according to the HMM shown in Figure 3.5. We ignore the state '*D*'. Since '*D*' can generate only the words '*a*' and '*the*' and neither of these words appear in our "sentence", all paths going through '*D*' must have zero probability and thus cannot be maximum probability paths.

As already mentioned, the computation starts at the left by setting $\mu_{\triangleright}(0) = 1$. Moving to word 1, we need to compute $\mu_N(1)$ and $\mu_V(1)$. These are particularly easy to calculate. The μs require finding the maximum over all states at position 0. But there is only one such state, namely $\triangleright$. Thus we get:

$$\begin{aligned}\mu_N(1) &= \mu_{\triangleright}(0)\ \tau_{\triangleright,N}\ \sigma_{N,flour}\\ &= 1 \circ 0.3 \circ 0.4\\ &= 0.12\end{aligned}$$

A similar situation holds for $\mu_V(1)$ except that the probability of flour as a verb is only 0.2 so the value at the verb node is 0.06.

We now move on to $i = 2$. Here there is only one possible state with a nonzero value, namely '*N*'. However, $Y_2 =$ '*N*' has two possible predecessors at $i = 1$, '*N*' and '*V*'. When we do the calculations we find that the maximum path probability at $i = 1$ comes from '*N*' with values

$$\begin{aligned}\mu_N(2) &= \mu_N(1)\ \tau_{N,N}\ \sigma_{N,pan}\\ &= 0.12 \circ 0.3 \circ 0.4\\ &= 0.0144\end{aligned}$$

We leave the calculation at $i = 3$ to the reader.

At each stage we need look backward only one step because the new maximum probability must be the continuation from the maximum probability at one of the previous states. Thus the computation for any one state at time i requires m processing steps, one for each possible previous state. Since there are $n + 1$ time steps (one for each word plus one step for $\triangleleft$), the total time is proportional to $m^2(n + 1)$. The runtime of the Viterbi algorithm is linear in n, the length of the string. This is much better than the obvious algorithm, which as we saw takes exponential time.

At this point, we hope that Equation 3.4 showing how to reduce the μ calculation at i to the μ calculations at $i-1$ seems reasonably intuitive. But we now prove it formally for those readers who like things reduced to hard mathematics.

$$\begin{aligned}
\mu_y(i) &= \max_{y_{0,i}} \mathrm{P}(x_{1,i}, y_{0,i-1}, Y_i = y) \\
&= \max_{y_{0,i}} \mathrm{P}(x_{1,i-1}, y_{0,i-2}, Y_{i-1} = y', x_i, Y_i = y) \\
&= \max_{y_{0,i}} \mathrm{P}(x_{1,i-1}, y_{0,i-2}, Y_{i-1} = y') \\
&\qquad \mathrm{P}(Y_i = y \mid x_{1,i-1}, y_{0,i-2}, Y_{i-1} = y') \\
&\qquad \mathrm{P}(x_i \mid x_{1,i-1}, y_{0,i-2}, Y_i = y, Y_{i-1} = y'))
\end{aligned}$$

We start by expanding things out. In the second line above, we separate out the x and y at position $i-1$. We then use the chain rule to turn the single probability into a product of three probabilities. Next we reduce things down.

$$\begin{aligned}
\mu_y(i) &= \max_{y_{0,i}} \mathrm{P}(x_{1,i-1}, y_{0,i-2} Y_{i-1} = y')\mathrm{P}(Y_i = y \mid Y_{i-1} = y')\mathrm{P}(x_i \mid y) \\
&= \max_{y_{0,i}} \mathrm{P}(x_{1,i-1}, y_{0,i-2} Y_{i-1} = y')\sigma_{y',y}\tau_{y,x_i} \\
&= \max_{y_{0,i}} \left(\max_{y_{0,i-1}} \mathrm{P}(x_{1,i-1}, y_{0,i-2}, Y_{i-1} = y') \right) \sigma_{y',y}\tau_{y,x_i} \\
&= \max_{y_{0,i}} \mu_{y'}(i-1)\sigma_{y_{i-1},y_i}\tau_{y_i,x_i} \\
&= \max_{y'=1}^{m} \mu_{y'}(i-1)\sigma_{y',y}\tau_{y,x_i}
\end{aligned}$$

In the top line we simplify by noting that the next state is dependent only on the last state, and the next output just on the next state. In line 2, we substitute the corresponding model parameters for these two probabilities. In line 3, since the first term does not involve the last state y_i, we look for the maximum over $y_{0,i-1}$. Then in line 4 we note that the previous term in the parentheses is simply the definition of $\mu_{y'}(i-1)$ and make the substitution. Finally, in line 5 we replace the maximum over the entire y sequence with simply the maximum over y' since it is the only variable left in the equation. Phew!

It's easy to extend this algorithm so that it finds the most likely label sequence $\hat{\boldsymbol{y}}$ as well as its probability. The basic idea is to keep a *back pointer* $\rho_y(i)$ that indicates which y' maximizes (3.4) for each i and y, i.e.:

$$\rho_y(i) \;=\; \underset{y'}{\operatorname{argmax}} \quad \mu_{y'}(i-1)\; \sigma_{y',y}\; \tau_{y,x_i}, \qquad i = 2, \ldots, n$$

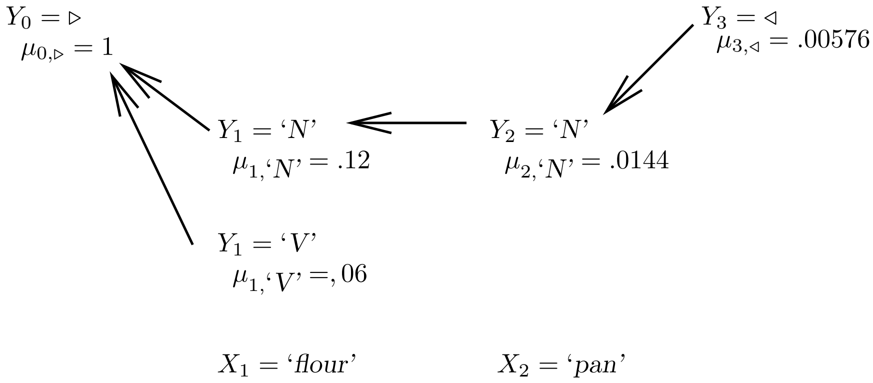

Figure 3.6: Figure 3.5 with back pointers added.

We can then read off the most likely sequence $\hat{\boldsymbol{y}}$ from right to left as follows:

$$\begin{aligned} \hat{y}_{n+1} &= \triangleleft \\ \hat{y}_i &= \rho_{\hat{y}_{i+1}}(i+1) \end{aligned}$$

That is, the last state in the maximum probability sequence must be $\triangleleft$. Then the most likely state at i is indicated by the back pointer at the maximum probability state at $i+1$.

Figure 3.6 shows Figure 3.5 plus the back pointers.

3.4 Finding sequence probabilities with HMMs

In the last section we looked at finding the most probable hidden sequence of states for a given HMM output. In this section we want to compute the total probability of the output. As in the last section, at first glance this problem looks intractable as it seems to require summing over all possible hidden sequences. But again dynamic programming comes to the rescue.

We wish to compute the probability of generating the observed sequence and ending up in the state $\triangleleft$.

$$\mathrm{P}(x_{1,n+1}, Y_{n+1} = \triangleleft)$$

We do this by computing something slightly more general, the so-called *forward probability*, $\alpha_y(i)$. This is the probability of the HMM generating

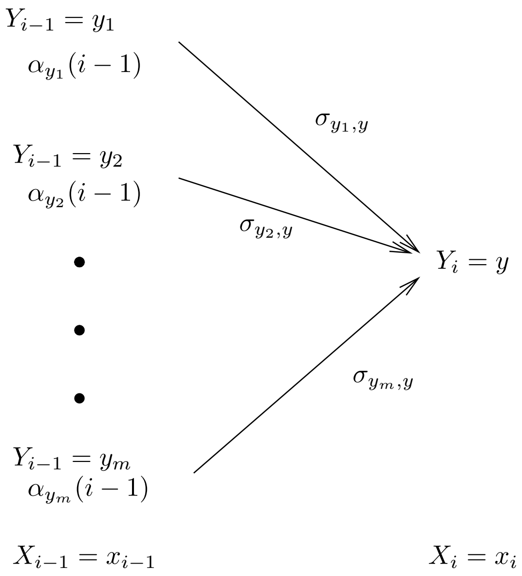

Figure 3.7: Piece of a trellis showing how forward probabilities are calculated

the symbols $x_{1,i}$ and ending up in state y_i.

$$\alpha_y(i) \quad = \quad \mathrm{P}(x_{1,i}, Y_i = y) \tag{3.5}$$

This is more general because the number we care about is simply:

$$\mathrm{P}(x_{1,n+1}, Y_{n+1} = \triangleleft) = \alpha_{\triangleleft}(n+1)$$

This follows immediately from the definition of the forward probability.

The reason why we recast the problem in this fashion is that forward probabilities can be computed in time linear in the length of the sequence. This follows from a mild recasting of the problem in the last section where we looked at the maximum path probability to any point. Here we compute sum rather than max. As before, we start the computation at the left-hand side of the trellis with

$$\alpha_{\triangleright}(0) = 1 \tag{3.6}$$

Also as before, the key intuition for the recursion step comes from looking at a piece of the trellis, this time shown in Figure 3.7. We assume here that all the $\alpha_{y'}(i-1)$ have been computed and we now want to compute $\alpha_y(i)$.

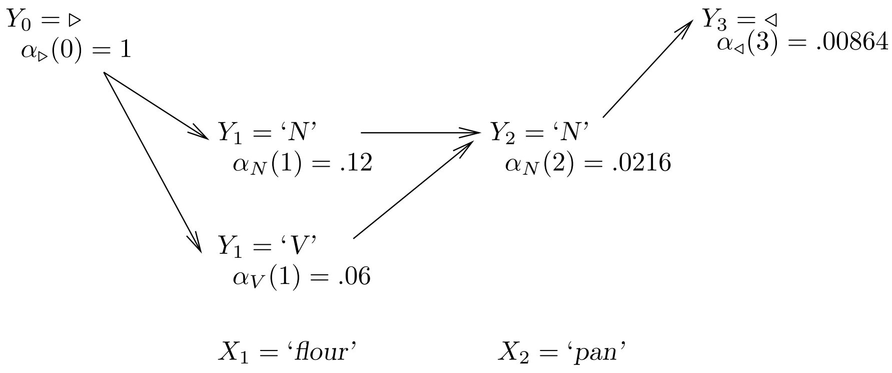

Figure 3.8: Forward probability calculations for '*flour pan*' HMM

This time we could reach $Y_i = y$ from any of the previous states, so the total probability at Y_i is the sum from each of the possible last states. The total path probability coming from one of these states is thus:

$$\alpha_{y\prime}(i-1)\sigma_{y\prime,y}\tau_{y,x_i}$$

This is the probability of first getting to y', times the transition probability of then getting to y, times the probability of generating x_i. Doing the sum gives us:

$$\alpha_y(i) = \sum_{y'} \alpha_{y'}(i-1)\sigma_{y',y}\tau_{y,x_i} \tag{3.7}$$

Example 3.3: Figure 3.8 shows the computation of forward probabilities for our '*flour pan*' example. At position zero $\alpha_{\triangleright}(0)$ is always one. At position one only one previous state is possible, so the sum in Equation 3.7 is a sum over one prior state. The most interesting calculation is for $Y_2 =$ 'N'. From the higher path into it (from 'N') we get 0.12 (the previous forward probability) times 0.3 (the transition probability) times 0.4 (the probability of generating the word '*pan*', for 0.0144. In much the same way, the lower path contributes 0.06 times 0.3 times 0.4=0.0072. So the forward probability at this node is 0.0216.

Deriving Equation 3.7 is relatively straightforward: add $y\prime$ through reverse marginalization, reorder terms, and replace the terms by the corresponding α, σ, and τs.

$$\begin{aligned}
\alpha_y(i) &= \mathrm{P}(x_{1,i}, Y_i = y)\\
&= \sum_{y'} \mathrm{P}(x_{1,i-1}, Y_{i-1} = y', Y_i = y, x_i)\\
&= \sum_{y'} \mathrm{P}(x_{1,i-1}, y')\mathrm{P}(y \mid y', x_{1,i-1})\mathrm{P}(x_i \mid y, y', x_{1,i-1})\\
&= \sum_{y'} \mathrm{P}(x_{1,i-1}, y')\mathrm{P}(y \mid y')\mathrm{P}(x_i \mid y)\\
&= \sum_{y'} \alpha_{y'}(i-1)\sigma_{y',y}\tau_{y,x_i}
\end{aligned}$$

3.5 Backward probabilities

In this section we introduce *backward probabilities*. . . We care about backward probabilities primarily because we need them for our polynomial time EM algorithm for estimating HMM parameters. However, for the purposes of this section we motivate them in a simpler fashion.

Suppose we are building an HMM part-of-speech tagger, and we intend to evaluate it by asking what percentage of the words are assigned the correct part of speech. Clearly the way to do this is to pick at each word the part of speech that maximizes the following probability:

$$\mathrm{P}(Y_i = y \mid \boldsymbol{x}).$$

At first glance, one might think that the Viterbi algorithm does this. However, as the following example shows, this is not the case.

Example 3.4: Let us suppose the sentence '*Flour pans like lightning*' has the following three possible part-of-speech sequences along with their associated probabilities:

Flour	pans	like	lightning	
N	V	A	N	0.11
N	N	V	N	0.1
N	N	A	N	0.05

It is immediately obvious that the first of these is the Viterbi analysis and thus has the most likely sequence of tags. However, let us now ask: given the entire sentence, what is the mostly likely tag for each word? The answer is the third one! To see this, first note that there is no competition at the first or fourth words. For the second word, the paths that go through '*N*' sum to 0.15, while those that go through '*V*' sum to 0.11. For position 3 we have '*V*' at 0.1 and '*A*' at 0.16.

To efficiently compute $\mathrm{P}(Y_i = y \mid \boldsymbol{x})$ we introduce *backward probabilities* defined as follows:

$$\beta_y(i) \quad = \quad \mathrm{P}(x_{i+1,n+1} \mid Y_i = y)$$

That is, $\beta_y(i)$ is the probability of generating the outputs from x_{i+1} to the end of the sentence assuming you start the process with $Y_i = y$.

At first glance this seems a pretty arbitrary definition, but it has all sorts of nice properties. First, as we show, it too can be computed using dynamic programing in time linear in the length of the sentence. Secondly, if we know both the forward and backward probabilities, we can easily compute $\mathrm{P}(Y_i = y \mid \boldsymbol{x})$:

$$\mathrm{P}(Y_i = y \mid \boldsymbol{x}) = \frac{\alpha_y(i)\beta_y(i)}{\alpha_{\triangleleft}(n+1)}. \tag{3.8}$$

This may be one case where the best way to come to believe this is just to see the derivation.

$$\begin{aligned}
\mathrm{P}(Y_i = y \mid \boldsymbol{x}) \quad &= \quad \frac{\mathrm{P}(x_{1,n+1}, Y_i = y)}{\mathrm{P}(x_{1,n+1})} \\
&= \quad \frac{\mathrm{P}(x_{1,i}, Y_i = y)\mathrm{P}(x_{i+1,n+1} \mid x_{1,i}, Y_i = y)}{\mathrm{P}(x_{1,n+1})} \\
&= \quad \frac{\mathrm{P}(x_{1,i}, Y_i = y)\mathrm{P}(x_{i+1,n+1} \mid Y_i = y)}{\mathrm{P}(x_{1,n+1})} \\
&= \quad \frac{\alpha_y(i)\beta_y(i)}{\alpha_{\triangleleft}(n+1)}
\end{aligned}$$

We start out with the definition of conditional probability. In the second line we rearrange some terms. The third line simplifies the last probability in the numerator using the fact that in a Markov model, once we know the state $Y_i = y$, anything that happened before that is independent of what comes after, and the last line substitutes the forward and backward probabilities for their definitions. We also use the fact that $\alpha_{\triangleleft}(n+1) = \mathrm{P}(\boldsymbol{x})$.

As with forward probabilities, we can compute the β values incrementally. However, there is a reason these are called *backward* probabilities. Now we start at the end of the string and work backward. First, at $i = n+1$,

$$\beta_{\triangleleft}(n+1) = 1. \tag{3.9}$$

At position $n+1$ there is nothing left to generate but the empty string. Its probability given we are in state $\triangleleft$ is 1.

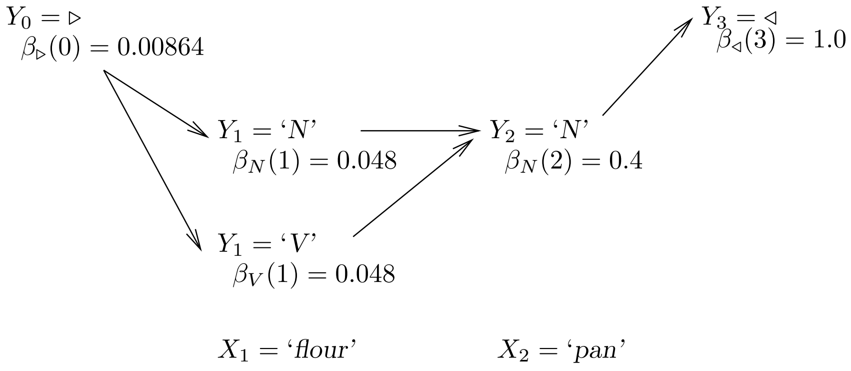

Figure 3.9: Computing backward probabilities for the ‘*flour pan*’ example

We now show that if we can compute the βs at position $i+1$, then we can compute it at i.

$$\begin{aligned}
\beta_y(i) &= \mathrm{P}(x_{i+1,n+1} \mid Y_i = y) \\
&= \sum_{y'} \mathrm{P}(Y_{i+1} = y', x_{i+1}, x_{i+2,n+1} \mid Y_i = y) \\
&= \sum_{y'} \mathrm{P}(y' \mid y)\mathrm{P}(x_{i+1} \mid y, y')\mathrm{P}(x_{i+2,n+1} \mid y, y', x_{i=1}) \\
&= \sum_{y'} \mathrm{P}(y' \mid y)\mathrm{P}(x_{i+1} \mid y')\mathrm{P}(x_{i+2,n+1} \mid y') \\
&= \sum_{y'} \sigma_{y,y'}\tau_{y',x_{i+1}}\beta_{y'}(i+1) \qquad (3.10)
\end{aligned}$$

The idea here is very much like what we saw for the forward probabilities. To compute $\beta_y(i)$, we sum over all the states that could follow $Y_i = y_i$. To be slightly more specific, consider $Y_{i+1} = y'$. We assume we know the probability of generating everything after that, $\beta_{y'}(i+1)$. To get $\beta_y(i)$, we also need to multiply in the probability of the symbol generated at $i+1$, $\tau_{y',x_{i+1}}$, and the probability of getting to y' from y, $\sigma_{y,y'}$.

Example 3.5: Figure 3.9 shows the backward probabilities for the ‘*flour pan*’ example. As it should, $\beta_{\triangleleft}(3) = 1.0$. Looking at the previous state, $\beta_V(2) = 0.4$. This is the product of $\beta_{\triangleleft}(3) = 1.0$, $\tau_{\triangleleft,\triangleleft} = 1$, and $\sigma_{V,\triangleleft} = 0.4$.

Next we can compute P($Y_2 =$ '*N*' | '*flour pan*'):

$$\begin{aligned}\mathrm{P}(Y_2 = V \mid \text{'}\textit{flour pan}\text{'}) &= \\ &= \frac{0.216 \cdot 0.4}{0.00864} \\ &= 1.0.\end{aligned}$$

This is as it should be, since there is no alternative to $Y_2 =$ '*V*'. In the same way, we can compute the probability of being in states '*N*' and '*V*' in position one as 2/3 and 1/3 respectively.

When computing forward and backward probabilities there are a few computations uou can use to check that you are doing it correctly. First, as can be verified from Figures 3.8 and 3.9, $\alpha_{\triangleleft}(n+1) = \beta_{\triangleleft}(0)$, and both are equal to the total probability of the string according to the HMM.

Second, for any i,

$$\sum_{y\prime} \mathrm{P}(Y_i = y\prime \mid \boldsymbol{x}) = 1.$$

That is, the total probability of being in any of the states given $\boldsymbol{x}$ must sum to one.

3.6 Estimating HMM parameters

This section describes how to estimate the HMM parameters $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ from training data that consists of output strings $\boldsymbol{x}$ and their labels $\boldsymbol{y}$ (in the case of visible training data) or output strings $\boldsymbol{x}$ alone (if the labels are hidden). In both cases, we treat the entire training data as one or two long strings. In practice, it is standard actually to break the data down into individual sentences. But the math is basically the same in both cases.

3.6.1 HMM parameters from visible data

In this section we assume that our training data is *visible* or *fully observed*, i.e., it consists of the HMM output $\boldsymbol{x} = (x_1, \ldots, x_n)$ (e.g., words) and their corresponding labels $\boldsymbol{y} = (y_1, \ldots, y_n)$ (e.g., parts of speech tags). The likelihood is then:

$$\begin{aligned}L(\boldsymbol{\sigma}, \boldsymbol{\tau}) &= \mathrm{P}(\boldsymbol{x}, \boldsymbol{y}) \\ &= \prod_{i=1}^{n+1} \sigma_{y_{i-1}, y_i} \tau_{y_i, x_i}.\end{aligned}$$

In this case, the maximum likelihood estimates for the parameters $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ are just their relative frequencies:

$$\begin{aligned}\hat{\sigma}_{y,y'} &= \frac{n_{y,y'}(\boldsymbol{y})}{n_{y,\circ}(\boldsymbol{y})}\\ \hat{\tau}_{y,x} &= \frac{n_{y,x}(\boldsymbol{x},\boldsymbol{y})}{n_{y,\circ}(\boldsymbol{x},\boldsymbol{y})}\end{aligned}$$

where

$$\begin{aligned}n_{y,y'}(\boldsymbol{y}) &= \sum_{i=1}^{n+1} [[y_{i-1}=y, y_i=y']] \,,\\ n_{y,x}(\boldsymbol{x},\boldsymbol{y}) &= \sum_{i=1}^{n} \llbracket y_i=y, x_i=x \rrbracket \,.\end{aligned}$$

That is, $n_{y,y'}(\boldsymbol{y})$ is the number of times that a label y' follows y, and $n_{y,x}(\boldsymbol{x},\boldsymbol{y})$ is the number of times that a label y labels an observation x.

In practice you want to smooth $\hat{\boldsymbol{\tau}}$ to deal with sparse-data problems such as unknown and low-frequency words. This is similar to what we did for language models in Section 1.3.5.

3.6.2 HMM parameters from hidden data

In this case our training data consists only of the output strings $\boldsymbol{x}$, and we are not told the labels $\boldsymbol{y}$; they are *invisible* or *hidden*. We can still write the likelihood, which (as usual) is the probability of the data.

$$\begin{aligned}L(\boldsymbol{\sigma},\boldsymbol{\tau}) &= \mathrm{P}(\boldsymbol{x}) \qquad (3.11)\\ &= \sum_{\boldsymbol{y}} \mathrm{P}(\boldsymbol{x},\boldsymbol{y})\\ &= \sum_{\boldsymbol{y}} \left(\prod_{i=1}^{n} \sigma_{y_{i-1},y_i} \tau_{y_i,x_i} \right)\end{aligned}$$

where the variable $\boldsymbol{y}$ in the sum ranges over all possible label sequences. The number of such label sequences grows exponentially in the length of the string $\boldsymbol{x}$, so it is impractical to enumerate the $\boldsymbol{y}$ except for very short strings $\boldsymbol{x}$.

There is no simple closed-form expression for the maximum likelihood estimates for $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ when the label sequence is not given, but since this is a hidden data problem we can use the expectation maximization algorithm.

Recall the general idea of the EM algorithm. We are given observations $\boldsymbol{x}$ and assume we have estimates $\boldsymbol{\sigma}^{(0)}, \boldsymbol{\tau}^{(0)}$ of the true parameters $\boldsymbol{\sigma}, \boldsymbol{\tau}$. We use these estimates to find estimates for how often each of the HMM transitions are taken while processing $\boldsymbol{x}$. We then estimate $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ from these expected values using the maximum likelihood estimator described in the previous section, producing new estimates $\boldsymbol{\sigma}^{(1)}$ and $\boldsymbol{\tau}^{(1)}$. That is,

$$\begin{aligned}\sigma^{(1)}_{y,y'} &= \frac{\mathrm{E}[n_{y,y'} \mid \boldsymbol{x}]}{\mathrm{E}[n_{y,\circ} \mid \boldsymbol{x}]} \\ \tau^{(1)}_{y,x} &= \frac{\mathrm{E}[n_{y,x} \mid \boldsymbol{x}]}{\mathrm{E}[n_{y,\circ} \mid \boldsymbol{x}]}\end{aligned}$$

where the expected counts

$$\begin{aligned}\mathrm{E}[n_{y,y'} \mid \boldsymbol{x}] &= \sum_{\boldsymbol{y}} n_{y,y'}(\boldsymbol{y}) \mathrm{P}(\boldsymbol{y} \mid \boldsymbol{x}), \text{ and} \\ \mathrm{E}[n_{y,x} \mid \boldsymbol{x}] &= \sum_{\boldsymbol{y}} n_{y,x}(\boldsymbol{x}, \boldsymbol{y}) \mathrm{P}(\boldsymbol{y} \mid \boldsymbol{x})\end{aligned}$$

are calculated using the probability distribution defined by $\boldsymbol{\sigma}^{(0)}$ and $\boldsymbol{\tau}^{(1)}$.

In theory we could calculate these expectations by explicitly enumerating all possible label sequences $\boldsymbol{y}$, but since the number of such sequences grows exponentially with the length of $\boldsymbol{x}$, this is infeasible except with extremely small sequences.

3.6.3 The forward-backward algorithm

This section presents a dynamic programming algorithm known as the *forward-backward algorithm* for efficiently computing the expected counts required by the EM algorithm. It is so named because the algorithm requires the computating both the forward and backward probabilities.

First we consider $\mathrm{E}[n_{y,x} \mid \boldsymbol{x}]$, the expectation of how often state y generates symbol x. Here we actually compute something more precise than we require here, namely the expectation that y generates x *at each point in the string*. Once these are computed, we get the expectations we need for the M-step by summing over all positions i. Given the forward and backward probabilities this turns out to be quite easy.

$$\begin{aligned}\mathrm{E}[n_{i,y,x} \mid \boldsymbol{x}] &= [[X_i = x]] \, \mathrm{P}(Y_i = y \mid \boldsymbol{x}) && (3.12) \\ &= [[X_i = x]] \frac{\alpha_y(i)\beta_y(i)}{\alpha_{\triangleleft}(n+1)} && (3.13)\end{aligned}$$

$$\begin{tikzcd}
\substack{Y_i = y_i \\ \alpha_{y_i}(i)} \arrow[rr, "\sigma_{y_i,y_{i+1}}"] & & \substack{Y_{i+1} = y_{i+1} \\ \beta_{y_{i+1}}(i+1)} \\
X_i = x_1 & & \substack{X_{i+1} = x_{i+1} \\ \tau_{y_{i+1},x_{i+1}}}
\end{tikzcd}$$

Figure 3.10: Detailed look at a transition with forward and backward probabilities

The first line says that if at position i if we do not generate x, then the expected number of times here is zero, and otherwise it is the probability that we were in state y when we generated x. The second line follows from Equation 3.8 in which we showed how to compute this probability from our forward and backward probabilities.

Next we consider the expectation of a transition from y to $y/$ at point i:

$$\mathrm{E}[n_{i,y,y'} \mid \boldsymbol{x}] = \mathrm{P}(Y_i = y, Y_{i+1} = y' \mid \boldsymbol{x}). \tag{3.14}$$

This says that the expectation of making the transition from y to y' at point i is the probability, given the visible data, that at i we are in state y and at $i+1$ we are in state y'. By the definition of conditional probability we can rewrite this as:

$$\mathrm{P}(Y_i = y, Y_{i+1} = y' \mid x_{1,n+1}) = \frac{\mathrm{P}(Y_i = y, Y_{i+1} = y', x_{1,n+1})}{\mathrm{P}(x_{1,n+1})} \tag{3.15}$$

We already know how to compute the denominator — it is just the total probability of the string, $\alpha_{\triangleleft}(n+1)$ — so we concentrate on the numerator. This is the probability of our HMM generating the sentence, but restricted to paths that go through the states $Y_i = y_i$ and $Y_{i+1} = y_{i+1}$. This situation is shown in Figure 3.10 along with the forward and backward probabilities at these positions in the trellis.

Now we claim — as is proved shortly — that:

$$\mathrm{P}(Y_i = y, Y_{i+1} = y', x_{1,n+1}) = \alpha_y(i)\sigma_{y,y'}, \tau_{y',x_{i+1}}\beta_{y'}(i+1) \tag{3.16}$$

Here, the left-hand side is the numerator in Equation 3.15, the probability of generating the entire sentence restricting consideration to paths going through the transition from $Y_i = y_i$ to $Y_{i+1} = y_{i+1}$. To understand the right-hand side, look again at Figure 3.10. Computing this probability requires first getting the probability of arriving at $Y_i = y_i$. This is the forward probability, α. Then we need the probability of getting from y_i to y_{i+1} (the σ term) while generating x_{i+1} (the τ term). Finally, given we are now at $Y_{i+1} = y_{i+1}$, we need the probability of generating the rest of the sentence (the backward probability).

Substituting Equation 3.16 into Equation 3.15 gives us:

$$\begin{aligned} \mathrm{E}[n_{i,y,y'} \mid \boldsymbol{x}] &= \mathrm{P}(Y_i = y, Y_{i+1} = y' \mid x_{1,n+1}) \\ &= \frac{\alpha_y(i)\sigma_{y,y'}, \tau_{y'x_{i+1}}\beta_{y'}(i+1)}{\alpha_{\triangleleft}(n+1)} \end{aligned} \tag{3.17}$$

We now prove the critical part of this.

$$\begin{aligned} \mathrm{P}(Y_i = y, \quad Y_{i+1} = y', \quad x_{1,n+1}) &= \mathrm{P}(x_{1,i}, Y_i = y, Y_{i+1} = y', x_{i+1}, x_{i+2,n+1}) \\ = \quad & \mathrm{P}(x_{1,i}, y)\mathrm{P}(y' \mid x_{1,i}, y)\mathrm{P}(x_{i+1} \mid y', x_{1,i}, y) \\ & \quad \mathrm{P}(x_{i+2,n+1} \mid y', x_{1,i+1}, y) \\ = \quad & \mathrm{P}(x_{1,i}, y)\mathrm{P}(y' \mid y)\mathrm{P}(x_{i+1} \mid y')\mathrm{P}(x_{i+2,n+1} \mid y') \\ = \quad & \mathrm{P}(x_{1,i}, y)\sigma_{y,y'}, \tau_{y'x_{i+1}}\mathrm{P}(x_{i+2,n+1} \mid y') \\ = \quad & \alpha_y(i)\sigma_{y,y'}, \tau_{y'x_{i+1}}\beta_{y'}(i+1) \end{aligned}$$

This sequence of transformations is similar in spirit to the one we used to derive our recursive formula for μ. Here we first rearrange some terms. Next we use the chain rule, and then apply the Markov assumptions to the various probabilities. Last, we substitute the forward and backward probabilities for their definitions.

3.6.4 The EM algorithm for estimating an HMM

At this point we have shown how to compute the forward and backward probabilities and how to use them to compute the expectations we need for the EM algorithm. Here we put everything together in the forward-backward algorithm.

We start the EM algorithm with parameter estimates $\boldsymbol{\sigma}^{(0)}$ and $\boldsymbol{\tau}^{(0)}$. Each iteration $t = 0, 1, 2, \ldots$ consists of the following steps:

1. Set all expected counts $\mathrm{E}[n_{y,y'}|\boldsymbol{x}]$ and $\mathrm{E}[n_{y,x}|\boldsymbol{x}]$ to zero

2. (E-step) For each sentence

   (a) Using $\boldsymbol{\sigma}^{(t)}$ and $\boldsymbol{\tau}^{(t)}$, calculate the forward probabilities $\boldsymbol{\alpha}$ and the backward probabilities $\boldsymbol{\beta}$ using the recursions (3.7–3.6) and (3.10–3.9).

(b) Calculate the expected counts $\mathrm{E}[n_{i,y,y'}|\boldsymbol{x}]$ and $\mathrm{E}[n_{i,y,x}|\boldsymbol{x}]$ using (3.17) and (3.13) respectively, and accumulate these into $\mathrm{E}[n_{y,y'}|\boldsymbol{x}]$ and $\mathrm{E}[n_{y,x}|\boldsymbol{x}]$.

3. (M-step) Compute $\boldsymbol{\sigma}^{(t+1)}$ and $\boldsymbol{\tau}^{(t+1)}$ using these expected counts

3.6.5 Implementing the EM algorithm for HMMs

There are a few things to keep in mind when implementing the algorithm just described. If all the values in $\boldsymbol{\sigma}^{(0)}$ and $\boldsymbol{\tau}^{(0)}$ are the same then the HMM is completely symmetric and the expected counts for the states are all the same. It's standard to break this symmetry by adding a small random number (or *noise*) to each of the values in $\boldsymbol{\sigma}^{(0)}$ and $\boldsymbol{\tau}^{(0)}$. So rather than setting them all to v, set them to rv for, random r such that $0.95 \leq r \leq 1.05$.

Another way to think of this is to imagine that we are attempting to learn word parts of speech from unlabeled data. Initially we have one state in the HMM for each POS-tag. The HMM must eventually learn that, say, $\tau_{N,dog}$ should be high, while $\tau_{V,dog}$ should be much lower (the verb '*dog*' means roughly the same thing as '*follow*', but it is not common). But naturally, as far as the HMM is concerned, it does not have states N, or V, it just has, say, 50 states representing 50 parts of speech. If we start with all the probabilities equal, then there is no reason to associate any particular state with any particular tag. This situation is called a *saddle point*. While the likelihood of the data is low with these probabilities, there is no reason for EM to move in any particular direction, and in fact, it does not change any of the probabilities. This is much like the philosophical example known as *Buridan's ass*. Buridan's ass is standing exactly halfway between two exactly equal bales of hay. It cannot decide which way to go because it is equally beneficial to go either way, and eventually it starves to death. If we break the symmetry by moving either bale by a very small amount, the ass survives.

As we mentioned at the end of Section 3.5, there are several relations between forward and backward probabilities that you should compute to check for bugs. Here is another one:

$$\sum_y \alpha_i(y)\,\beta_i(y) = \mathrm{P}(\boldsymbol{x})\text{for all } i = 1, \ldots$$

Be sure to understand how this follows directly from Equation 3.8.

3.7 MT parameters from forward-backward

Section 2.3 showed how IBM model 2 improves over model 1 insofar as it no longer assigns equal probability to all alignments. Instead, the probability of an alignment was the product of $\delta_{k,l,m.j}$s where l and m are the lengths of the English and French sentences and the delta gives us the probability of the kth French word being aligned with the jth English and thus model 2 still incorporates the assumption that each alignment decision is independent of the others.

Generally this is a bad assumption. Words in sentences are bundled into phrases, and typically if one word of a phrase gets moved during translation the others will go along with it. For example, both English and French allow prepositional phrases to appear at the beginning or ending of a sentence:

Columbus discovered America in 1492
In 1492 Columbus discovered America

If we were to align these two sentences we would get $< 4, 5, 1, 2, 3 >$. The model 2 probability would be much too low because, e.g., the probability of aligning position 2 in the second sentence with position 5 in the first would be small, even though the previous word had been "moved" to position 4 and thus the next, being in position 5, should be high. A better model would instead condition an alignment a_k on the alignment of the previous French word a_{k-1}.

HMMs are a natural fit for this problem. Our program creates a new HMM specifically for for each training E/F pair. Figure 3.11 shows an HMM created to learn the translation parameters for translating "Not bad" to "Pas mal."

The hidden HMM states are the alignment positions, so each one is dependent on the previous. The visible sequence is the French sentence. Thus $\sigma_{j,k}$ is the probability that the next French word will align with the kth English word, given that the previous word aligned with the jth. The emission probabilities $\tau_{y,x}$ are the probability that e_y will generate y_x, the French word.

The HMM in Figure 3.11 starts in state $\triangleright$. Then for the first French word the HMM transitions into state Y_1 or Y_2. To transition into, say, Y_2 at position one is to guess that the first French word '*Pas*' is generated by (aligned with) the second English ('*bad*'). If at position two the HMM goes to Y_1 then the second French word is aligned with the first English one. The σ parameters are the probabilities of these transitions. Note that because this HMM was devised for this particular example, both states at position

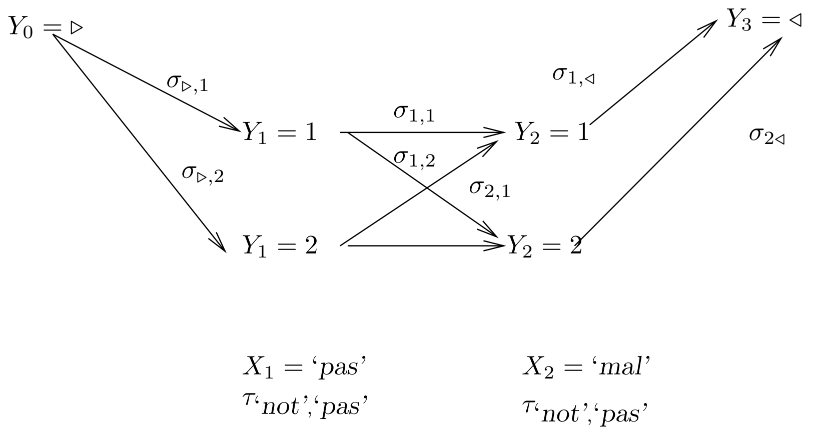
$Y_0 = \triangleright$
$\sigma_{\triangleright,1}$
$\sigma_{\triangleright,2}$
$Y_1 = 1$
$Y_1 = 2$
$\sigma_{1,1}$
$\sigma_{1,2}$
$\sigma_{2,1}$
$Y_2 = 1$
$Y_2 = 2$
$\sigma_{1,\triangleleft}$
$\sigma_{2\triangleleft}$
$Y_3 = \triangleleft$
$X_1 =$ '*pas*'
$\tau_{\text{'not','pas'}}$
$X_2 =$ '*mal*'
$\tau_{\text{'not','pas'}}$

Figure 3.11: An HMM to learn parameters for '*Pas mal*' to '*Not bad*'

one can only generate the first word of this French sentence, namely '*Pas*' and the probability of its doing so are the translation probabilities — e.g., $\tau_{bad,Pas}$.

However, even though this HMM was created for this particular training example, the parameters that are learned are general. So, for example, the two sentences here are both of length two. Thus the transition from, say, state Y_1 at position one to Y_2 at position two will add its expectation to $\delta_{1,2,2,2}$. These are the new distortion parameters for the HMM translation model. This particular parameter is the probability of a French word being aligned with English word 2, given that the previous French was aligned with English word 1 and both sentences are of length two. Other examples in the parallel corpus where both sentences are of length two will also contribute their expectations to this δ.

Because the assumptions behind this model are much closer to reality than those underlying IBM 2, the parameters learned by this model are better than those of IBM 2, and the math is only slightly more complicated. So now MT systems start by estimating τs from IBM 1, and then improve on these using the HMM approach.

There is one minor flaw in the model as we have presented it. What if the previous word was aligned with the null English word? What should our new $\sigma_{0,y}$ be? One solution would to use the last non-null position. A slightly more complicated solution would be to have a second set of English "positions" that are used for this situation. So, e.g., $\sigma_{3',4}$ would be the probability that the next French word will align with e_4 given that the previous word was aligned with e_0 and the last non-null alignment was with e_3.

3.8 Smoothing with HMMs

In Chapter 1 we encountered the problem of smoothing language-modeling probability distributions in the presence of sparse data. In particular, since a maximum-likelihood distribution assigns a zero probability to any word not seen in the training data, we explored the smoothed distribution obtained by assigning a pseudo-count α to the count of every possible word. (In practice we had only one unseen word, ∗U∗, but this need not be the case.) If we gave all words the same pseudo-count we arrived at the following equation:

$$\mathrm{P}_{\widetilde{\boldsymbol{\theta}}}(W{=}w) \quad = \quad \widetilde{\theta}_w \ = \ \frac{n_w(\boldsymbol{d})+\alpha}{n_\circ(\boldsymbol{d})+\alpha|\mathcal{W}|}$$

Then to find an appropriate α we looked at the probability a particular lambda would assign to some held-out data (the *likelihood* of the data). Last we suggested using line-search to maximize the likelihood.

In this section we show how using HMMs allow us to find these parameters more directly. First we note that our smoothing equation can be rewritten as a *mixture model* — specifically as a mixture of the maximum-likelihood distribution $\mathrm{P}_{\widehat{\boldsymbol{\theta}}}(W)$ and the *uniform distribution* $\mathrm{P}_1(W)$ that assigns the same probability to each word $w \in \mathcal{W}$.

$$\begin{aligned}\mathrm{P}_{\widetilde{\boldsymbol{\theta}}}(W{=}w) \quad &= \quad \widetilde{\theta}_w \ = \ \frac{n_w(\boldsymbol{d})+\alpha}{n_\circ(\boldsymbol{d})+\alpha|\mathcal{W}|} \\ &= \quad \lambda\frac{n_w(\boldsymbol{d})}{n_\circ(\boldsymbol{d})} + (1-\lambda)\frac{1}{\mathcal{W}} \\ &= \quad \lambda \mathrm{P}_{\widehat{\boldsymbol{\theta}}}(W) + (1-\lambda)\mathrm{P}_1(W) \end{aligned} \tag{3.18}$$

where the uniform distribution is $\mathrm{P}_1(w) = 1/|\mathcal{W}|$ for all $w \in \mathcal{W}$ and the *mixing parameter* $\lambda \in [0,1]$ satisfies:

$$\lambda \quad = \quad \frac{n_\circ(\boldsymbol{d})}{n_\circ(\boldsymbol{d})+\alpha|\mathcal{W}|}$$

To see that this works, substitute the last equation into (3.18).

Now consider the HMM shown in Figure 3.12. It has the property that Equation 3.18 gives the probability this HMM assigns to the data. Intuitiviely it says that at any point one generates the next symbol either by

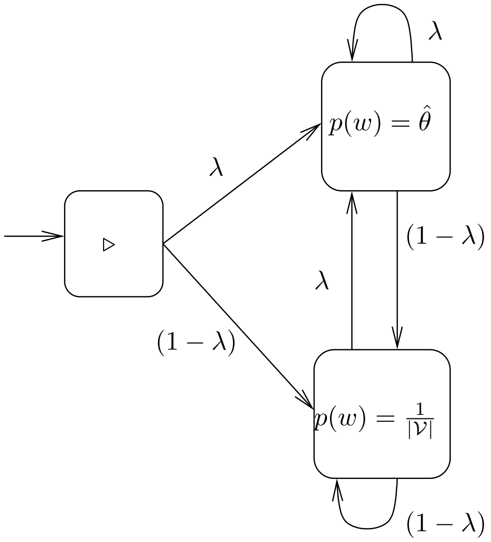

Figure 3.12: The HMM coresponding to Equation 3.18

choosing to go to the top sate with probability λ and then generating the word according to the maximum likelihood extimate $\hat{\theta}_w$, or to the bottom one and assigning a probability according to the uniform distribution $\frac{1}{\mathcal{W}|}$.

Example 3.6: Suppose we set λ to 0.8, the next word is "the", $\hat{\theta}_{the} = .08$ and number of words in our vocabularay is 10000. Then generating "the" via the top state has a probability of 0.16 while via the lower one has probability $2.0 \cdot 10^{-6}$. For our unknown word, $*$U$*$, the probabilities are zero and $2.0 \cdot 10^{-6}$.

Since λ is the transition probability of an HMM, we can now use Equation 3.17 iteratively to find a value that maximizes the probability of the data.

While this will work fine, for this HMM it is overkill. A bit of reflection will show that we can simplify the math enormously. We needed forward and backward probabilities because the state transltion used affects not only the current x, but the y we end up in after the transition, and this in turn, means that the rest of the string will get a different probability according to the model. But this is *not* the case for the HMM of Figure 3.12. Yes, we end up in different states, but this has no effect on the probability of the rest of the string. The next word will also either be generated by one of the two states (top and bottom, or T and B) and the probabilitity of getting to those states is the same no matter where we started. So the only difference in expectation is that due to the probability of the word we generate at this step. So the increment in exepectaions due to the ith word for the transition from T to T (which is the same as the transition from B to T) is

$$\mathrm{E}[n_{i,T,T} \mid \boldsymbol{x}] = \mathrm{E}[n_{i,B,T} \mid \boldsymbol{x}] = \frac{\lambda\hat{\theta}_{w_i}}{\lambda\hat{\theta}_{w_i} + (1-\lambda)\frac{1}{|\mathcal{W}|}}.$$

When we end up in the lower state we have:

$$\mathrm{E}[n_{i,T,B} \mid \boldsymbol{x}] = \mathrm{E}[n_{i,B,B} \mid \boldsymbol{x}] = \frac{(1-\lambda)\frac{1}{|\mathcal{W}|}}{\lambda\hat{\theta}_{w_i} + (1-\lambda)\frac{1}{|\mathcal{W}|}}$$

Example 3.7: Again suppose we set λ to 0.8, the next word is “the”, $\hat{\theta}_{the} = .08$ and number of words in our vocabularay is 10000. The expectation for the transition to state T when generating “the” is $(.08/(.08+2.0\cdot 10^{-6})) \approx 1.0$, and for transition to state B is $2.010^{-6}/(.08 + 2.0 \cdot 10^{-6}) \approx 2.510{-}5$. For $*$U$*$the expectation of transitioning to T is zero $(= 0/(0 + 2 \cdot 10^{-6}))$ and has an expectation of one for transitioning to state B $(= 210^{-6}/(0 + 2 \cdot 10^{-6}))$.

3.9 Part-of-speech induction

Part-of-speech (POS) tagging is a standard example for HMM’s and in the next section we discuss building a tagger when one has training data on which to base the HMM parameters. Typically this works quite well, with accuracies in the middle 90% range.

Here we discuss using EM and the forward backward algorithm for doing POS tagging when we do not have supervision — just plain English text. This works much less well. The model we discuss here achieves about 60% accuracy. The purpose of this section is to understand this difference, and more generally, to give the reader some idea of when the EM algorithm might not give you what you want.

The model is straightforward. We start the HMM with all our τs and σs approximately equal. We say “approximately” because of the saddle point problem discussed in Section 3.6.5. Also, because our testing data is labeled with 45 parts of speech, we start with 45 states. We then apply the forward-backward algorithm to adjust the parameters to fit the data. In so doing EM will drive the parameters apart.

The first step in understanding the low score is to look at the problem from the machine’s point of view. In essence, it will achieve the maximum likelihood for the training data when the words in each class are as similar

7	DET	The “ But In It He A And For That They As At Some This If
18	.	. ? ! ... in Activity Daffynition of -RCB- to -RRB- students
6	NNP	Mr. New American Wall National John Dow Ms. Big Robert
36	NNP	Corp. Inc. & York Co. 's Exchange Stock Street Board Bank
8	VBD	said says is say 's and reported took think added was makes
45	VBD	about at rose up fell down was closed net only a income dropped
19	CD	1 few 100 2 10 15 50 11 4 500 3 30 200 5 20 two
44	NN	year share week month 1988 months Friday 30 1987 September
5	DET	a in this last next Oct. “ Nov. late Sept. fiscal one early recent
32	DET	the a an its his their any net no some this one that another
42	DET	the its a chief his this “ other all their each an which such U.S.

Figure 3.13: Some HMM states and the most common words in them

as possible based upon the neighboring words. But similar in what way? There is nothing saying that the classes have to look anything like POS classes. Actually, from this point of view 60% is pretty impressive.

Figure 3.13 shows 11 of the 45 resulting states using forward-backward and repeating until the log-likelihood of the data barely increases at each iteration. This table may be a bit overwhelming at first glance, but it is worth study because it shows all the creative ways EM can raise the probability of the data, many with only marginal relation to part-of-speech tagging.

The first column is the state number — as just discussed this is arbitrary. Which words end up in which numbered states depends only on how the states are randomly initialized. By the way, this also means that different random initializations will give different accuracies, and the classes will be differently organized. Remember, EM is a hill-climbing algorithm and the hill it climbs depends on where it starts. The accuracy, however, never moves much from the 60% figure.

The second column gives the part-of-speech tag that is most common for the words in this set. We shown two sets each where the most common is ‘*NNP*’ (proper noun) and ‘*NN*’ (common noun) and four for ‘*DET*’ (determiner). In fact, eight of our states were assigned to DET. We will come back to this point.

Finally, each row has up to fifteen words that are “common” for this state. In particular, for each state s we found the fifteen words w that maximized the term

$$\frac{n_{s,w} + \alpha}{n_{\circ,w} + 45\alpha}.$$

If we did not have the add-α smoothing terms this equation would find the w's that are most probable given the state. These would mostly be words that only appear once or twice, and only in s. These, however, would not really be indicative of what the state really "means.". Smoothing with α (we used $\alpha = 5$) prevents this.

The first row in Table 3.13 (class 7) is made up of words and punctuation that begin sentences — e.g. words w for which $\sigma_{\triangleright,w}$ is large. It is assigned to the class DET just because "The" happens to be the most common word in the class by far. Symmetrically, class 18 is words that end sentences. The class of final punctuation marks is named '*.*' so that is the part of speech assignment for class 18. But we can see that the HMM is more interested in things that end sentences than things that are final punctuation because the "right round bracket" symbol is in the class. (It is written '*-RRB-*' rather than '*)*' because in the Penn Treebank parentheses are used to annotate trees.)

Continuing down the list, classes 6 and 36 are both assigned to proper nouns (NNP), but 6 is made up of words that typically start names, while 36 end names. Similarly we show two classes of past-tense verbs, each of which show a sort of within-class semantic similarity. Class 8 has many acts of thinking and saying, while Class 45 has things that happen to stock prices. As you should now expect, the HMM groups them for more prosaic reasons — the first are often followed by open-quotation marks, the second by numbers (as in "fell 19 %").

We leave classes 44 and 19 to the reader, and move on to three more classes that all get associated with determiners. The first of these (5) only has one DET, namely '*a*', but the other words have no common part of speech, and '*a*'s are ubiquitous, so it controls the class. In fact, these happen to be words that precede those in class 44 (mostly date words). But classes 32 and 42 have such overlap of determiners that it is hard to see why they are separate. An examination of the state transitions show that class 32 determiners are followed by common nouns, while those in 42 are followed by adjectives.

The point of this exercise is to disabuse the reader of the idea that maximizing the likelihood of the data will necessary make good things happen. It also should convince you that the forward-backward algorithm is really good at what it does: it will come up with ways to increase probability that we would never think of.

There are more successful approaches to unsupervised POS induction — currently the best achieve in the middle 70% range. As you might expect given the eight classes assigned to determiners, one very effective technique is to require each word type to appear in only one class. This is somewhat limiting, but most words have only one possible POS tag, and those with more than one typically have only one that is common (e.g., '*can*'), so mistakes due to such a restriction are comparatively rare.

3.10 Exercises

Exercise 3.1: The following tagging model has the two words '*boxes*' and '*books*', and the two POS tags '*noun*' and '*verb*' (plus the end-of-sentence delimiters ⊳ and ⊲).

$$\begin{array}{lll}
\mathrm{P}(noun \mid \triangleright) = 1/2 & \mathrm{P}(verb \mid \triangleright) = 1/2 & \mathrm{P}(boxes \mid noun) = 1/2 \\
\mathrm{P}(noun \mid noun) = 1/2 & \mathrm{P}(verb \mid noun) = 1/6 & \mathrm{P}(boxes \mid verb) = 3/4 \\
\mathrm{P}(noun \mid verb) = 1/2 & \mathrm{P}(verb \mid verb) = 1/6 & \mathrm{P}(books \mid noun) = 1/2 \\
\mathrm{P}(\triangleleft \mid noun) = 1/3 & \mathrm{P}(\triangleleft \mid verb) = 1/3 & \mathrm{P}(books \mid verb) = 1/4
\end{array}$$

(a) What is the *total probability* of the output sequence '*boxes books*'? Show your work. (b) What is the probability of the *most likely tag sequence* for '*boxes books*'? Show your work.

Exercise 3.2: Suppose for some HMM when applied to the sequence $\boldsymbol{x}$, $\mathrm{P}(\boldsymbol{x} \mid \boldsymbol{y}) = 0$ for all $\boldsymbol{y} = < \cdots, Y_i = a, \cdots >$. That is, any sequence of states $\boldsymbol{y}$ that goes through state a at position i has zero joint probability with the string $\boldsymbol{x}$. Does it follow that $\tau_{a,x_i} = 0$?

Exercise 3.3: Example 3.4 shows that the Viterbi sequence of labels is not always the sequence of labels that individually have the highest label probability. Note that in this example no single labeling has the majority of the total sentence probability (i.e., the probability of the sentence is more than twice the probability of the Viterbi labeling). Is this a coincidence of the numbers we picked, or a necessary feature of such examples? To put this another way, if the Viterbi sequence is not the same as the sequence of individually hightest probability labels, does this imply that the probability of Viterbi sequence is less than the total probability of all the other possible sequences? Explain.

Exercise 3.4: What does the presence of the word "Daffynition" in class 18 in Figure 3.13 suggest about how it is used in the *Wall Street Journal*?

3.11 Programming problems

Problem 3.1: Part-of-speech tagging using HMMs

The data directory contains files `wsj2-21.txt` and `wsj22.txt`. Each file contains one sentence per line, where each line is a sequence of pairs consisting of a word and its part of speech. Take a look at the files so you know what the precise format is. `wsj22.txt` has been pre-filtered so that words that don't appear in `wsj2-21.txt` have been replaced with the unknown word symbol $*\mathrm{U}*$.

The assignment directory contains the script templates `tag` and `score`. Follow the directions in the templates carefully.

1. Find maximum-likelihood estimates for the parameters $\hat{\boldsymbol{\sigma}}$ and $\hat{\boldsymbol{\tau}}$ from the file `wsj2-21.txt`. Note that you'll have to smooth the parameter estimates for $\tau_{y,*\mathrm{U}*}$; at this stage you can just give these a pseudo-count of 1.

2. Implement the Viterbi decoding algorithm and find the most likely tag sequence $\hat{\boldsymbol{y}}$ for each sentence $\boldsymbol{x}$ in `wsj22.txt`. Compute the percentage of words for which their Viterbi tag is in fact the correct tag.

3. Now we'll try to find a better estimate for $\tau_{y,\cdot\mathtt{UNK}\cdot}$. We note that words that appear once in our training data `wsj2-21.txt` are only one occurrence away from not occurring at all. So suppose we go through our training data and change all words that only appear once to $*\mathrm{U}*$. We can now compute $\tau_{y,*\mathrm{U}*}$ just like everything else and there is no need to smooth. Also note that, say, $\tau_{NN,*\mathrm{U}*}$ can (and will) differ from $\tau_{DT,*\mathrm{U}*}$. In general this should increase accuracy because it correctly models the fact that an $*\mathrm{U}*$ is more likely to be a noun than a determiner. Implement this and report your new accuracy. (One problem with this is that you lose the part-of-speech tags for all these words. Fortunately they are rare words, so the odds are that few of them will appear in your test data. However, if this bothers you, you can count each word with one token twice, once as itself, once as $*\mathrm{U}*$.)

3.12 Further reading

HMMs are used not just in computational linguistics, but in signal processing, computational biology, analysis of music, hand-writing recognition, land-mine detection, you name it.