Introduction to Computational Linguistics

Eugene Charniak and Mark Johnson

Contents

Chapter 1

Language modeling and probability

1.1 Introduction

Which of the following is a reasonable English sentence: '*I bought a rose*' or '*I bought arose*'[1]?

Of course the question is rhetorical. Anyone reading this text must have sufficient mastery of English to recognize that the first is good English while the second is not. Even so, when spoken the two sound the same, so a computer dictation system or *speech-recognition system* would have a hard time distinguishing them by sound alone. Consequently such systems employ a *language model*, which distinguishes the fluent (i.e., what a native English speaker would say) phrases and sentences of a particular natural language such as English from the multitude of possible sequences of words.

Virtually all methods for language modeling are probabilistic in nature. That is, instead of making a categorical (yes-or-no) judgment about the fluency of a sequence of words, they return (an estimate of) the probability of the sequence. The probability of a sequence is a real number between 0 and 1, and high-probability sequences are more likely to occur than low-probability ones. Thus a model that estimates the probability that a sequence of words is a phrase or sentence permits us to rank different sequences in terms of their fluency; it can answer the question with which we started this chapter.

In fact, probabilistic methods are pervasive in modern computational lin-

[1]©Eugene Charniak, Mark Johnson 2013

guistics, and all the major topics discussed in this book involve probabilistic models of some kind or other. Thus a basic knowledge of probability and statistics is essential and one of the goals of this chapter is to provide a basic introduction to them. In fact, the probabilistic methods used in the language models we describe here are simpler than most, which is why we begin this book with them.

1.2 A Brief Introduction to Probability

Take a page or two from this book, cut them into strips each with exactly one word, and dump them into an urn. (If you prefer you can use a trash can, but for some reason all books on probability use urns.) Now pull out a strip of paper. How likely is it that the word you get is '*the*'? To make this more precise, what is the probability that the word is '*the*'?

1.2.1 Outcomes, Events and Probabilities

To make this still more precise, suppose the urn contains 1000 strips of paper. These 1000 strips constitute the *sample space* or the set of all possible outcomes, which is usually denoted by Ω (the Greek letter omega). In discrete sample spaces such as this, each sample $x \in \Omega$ is assigned a *probability* $\mathrm{P}(x)$, which is a real number that indicates how likely each sample is to occur. If we assume that we are equally likely to choose any of the strips in the urn, then $\mathrm{P}(x) = 0.001$ for all $x \in \Omega$. In general, we require that the probabilities of the samples satisfy the following constraints:

1. $\mathrm{P}(x) \in [0,1]$ for all $x \in \Omega$, i.e., probabilities are real numbers between 0 and 1, and
2. $\sum_{x \in \Omega} \mathrm{P}(x) = 1$, i.e., the probabilities of all samples sum to one.

Of course, many of these strips of paper have the same word. For example, if the pages you cut up are written in English the word '*the*' is likely to appear on more than 50 of your strips of paper. The 1000 strips of paper each correspond to a different word *token* or occurrence of a word in the urn, so the urn contains 1000 word tokens. But because many of these strips contain the same word, the number of word *types* (i.e., distinct words) labeling these strips is much smaller; our urn might contain only 250 word types.

We can formalize the distinction between types and tokens by using the notion of a random *event*. Formally, an event E is a set of samples, i.e., $E \subseteq \Omega$, and the probability of an event is the sum of the probabilities of the samples that constitute it:

$$\mathrm{P}(E) \quad = \quad \sum_{x \in E} \mathrm{P}(x)$$

We can treat each word type as an event.

Example 1.1: Suppose E_{the} is the event of drawing a strip labeled '*the*', that $|E_{the}| = 60$ (i.e., there are 60 strips labeled '*the*') and that $\mathrm{P}(x) = 0.001$ for all samples $x \in \Omega$. Then $\mathrm{P}(E_{the}) = 60 \times 0.0001 = 0.06$.

1.2.2 Random Variables and Joint Probabilities

Random variables are a convenient method for specifying events. Formally, a random variable is a function from the sample space Ω to some set of values. For example, to capture the type-token distinction we might introduce a random variable W that maps samples to the words that appear on them, so that $W(x)$ is the word labeling strip $x \in \Omega$.

Given a random variable V and a value v, $\mathrm{P}(V{=}v)$ is the probability of the event that V takes the value v, i.e.:

$$\mathrm{P}(V{=}v) \quad = \quad \mathrm{P}(\{x \in \Omega : V(x) = v\})$$

Notation: It is standard to capitalize random variables and use lower-cased variables as their values.

Returning to our type-token example, $\mathrm{P}(W{=}\text{'}the\text{'}) = 0.06$. If the random variable intended is clear from the context, sometimes we elide it and just write its value, e.g., $\mathrm{P}(\text{'}the\text{'})$ abbreviates $\mathrm{P}(W{=}\text{'}the\text{'})$. Similarly, the value of the random variable may be elided if it is unspecified or clear from the context, e.g., $\mathrm{P}(W)$ abbreviates $\mathrm{P}(W{=}w)$ where w ranges over words.

Random variables are useful because they let us easily construct a variety of complex events. For example, suppose F is the random variable mapping each sample to the first letter of the word appearing in it and S is the random variable mapping samples to their second letters (or the space character if there is no second letter). Then $\mathrm{P}(F{=}\text{'}t\text{'})$ is the probability of the event in which the first letter is 't' and $\mathrm{P}(S=\text{'}h\text{'})$ is the probability of the event in which the second letter is 'h'.

Given any two events E_1 and E_2, the probability of their conjunction $\mathrm{P}(E_1, E_2) = \mathrm{P}(E_1 \cap E_2)$ is called the *joint probability* of E_1 and E_2; this is the probability of E_1 and E_2 occurring simultaneously. Continuing with our example, $\mathrm{P}(F=\text{'}t\text{'}, S=\text{'}h\text{'})$ is the joint probability that the first letter is 't' *and* that the second letter is 'h'. Clearly, this must be at least as large as P('*the*').

1.2.3 Conditional and marginal probabilities

Now imagine temporarily moving all the strips whose first letter is 'q' into a new urn. Clearly this new urn has a different distribution of words from the old one; for example, $\mathrm{P}(F=\text{'}q\text{'}) = 1$ in the sample contained in the new urn. The distributions of the other random variables change as well; if our strips of paper contain only English words then $\mathrm{P}(S=\text{'}u\text{'}) \approx 1$ in the new urn (because 'q' is almost always followed by 'u' in English).

Conditional probabilities formalize this notion of temporarily setting the sample set to a particular set. The *conditional probability* $\mathrm{P}(E_2 \mid E_1)$ is the probability of event E_2 given that event E_1 has occurred. (You can think of this as the probability of E_2 given that E_1 is the temporary sample set). $\mathrm{P}(E_2 \mid E_1)$ is defined as:

$$\mathrm{P}(E_2 \mid E_1) \;=\; \frac{\mathrm{P}(E_1, E_2)}{\mathrm{P}(E_1)} \quad \text{if } \mathrm{P}(E_1) > 0$$

and is undefined if $\mathrm{P}(E_1) = 0$. (If it is impossible for E_1 to occur then it makes no sense to speak of the probability of E_2 given that E_1 has occurred.) This equation relates the *conditional probability* $\mathrm{P}(E_2 \mid E_1)$, the *joint probability* $\mathrm{P}(E_1, E_2)$ and the *marginal probability* $\mathrm{P}(E_1)$. (If there are several random variables, then the probability of a random variable V on its own is sometimes called the *marginal probability* of V, in order to distinguish it from joint and conditional probabilities involving V.) The process of adding up the joint probabilities to get the marginal probability is called *marginalization*.

Example 1.2: Suppose our urn contains 10 strips of paper (i.e., our sample space Ω has 10 elements) that are labeled with four word types, and the frequency of each word is as follows:

word type	**frequency**
‘*nab*’	1
‘*no*’	2
‘*tap*’	3
‘*tot*’	4

Let F and S be random variables that map each strip of paper (i.e., sample) to the first and second letters that appear on them, as before. We start by computing the marginal probability of each random variable:

$$\begin{aligned} \mathrm{P}(F{=}\text{‘}n\text{’}) &= 3/10 \\ \mathrm{P}(F{=}\text{‘}t\text{’}) &= 7/10 \\ \\ \mathrm{P}(S{=}\text{‘}a\text{’}) &= 4/10 \\ \mathrm{P}(S{=}\text{‘}o\text{’}) &= 6/10 \end{aligned}$$

Now let’s compute the *joint probabilities* of F and S:

$$\begin{aligned} \mathrm{P}(F{=}\text{‘}n\text{’}, S{=}\text{‘}a\text{’}) &= 1/10 \\ \mathrm{P}(F{=}\text{‘}n\text{’}, S{=}\text{‘}o\text{’}) &= 2/10 \\ \mathrm{P}(F{=}\text{‘}t\text{’}, S{=}\text{‘}a\text{’}) &= 3/10 \\ \mathrm{P}(F{=}\text{‘}t\text{’}, S{=}\text{‘}o\text{’}) &= 4/10 \end{aligned}$$

Finally, let’s compute the *conditional probabilities* of F and S. There are two ways to do this. If we condition on F we obtain the following conditional probabilities:

$$\begin{aligned} \mathrm{P}(S{=}\text{‘}a\text{’} \mid F{=}\text{‘}n\text{’}) &= 1/3 \\ \mathrm{P}(S{=}\text{‘}o\text{’} \mid F{=}\text{‘}n\text{’}) &= 2/3 \\ \\ \mathrm{P}(S{=}\text{‘}a\text{’} \mid F{=}\text{‘}t\text{’}) &= 3/7 \\ \mathrm{P}(S{=}\text{‘}o\text{’} \mid F{=}\text{‘}t\text{’}) &= 4/7 \end{aligned}$$

On the other hand, if we condition on S we obtain the following conditional probabilities:

$$\begin{aligned} \mathrm{P}(F{=}\text{‘}n\text{’} \mid S{=}\text{‘}a\text{’}) &= 1/4 \\ \mathrm{P}(F{=}\text{‘}t\text{’} \mid S{=}\text{‘}a\text{’}) &= 3/4 \\ \\ \mathrm{P}(F{=}\text{‘}n\text{’} \mid S{=}\text{‘}o\text{’}) &= 2/6 \\ \mathrm{P}(F{=}\text{‘}t\text{’} \mid S{=}\text{‘}o\text{’}) &= 4/6 \end{aligned}$$

Newcomers to probability sometimes have trouble distinguishing between the conditional probabilities $\mathrm{P}(A \mid B)$ and $\mathrm{P}(B \mid A)$, seeing both as expressing a correlation between A and B. However, in general there is no reason to think that they will be the same. In Example 1.2

$$\begin{aligned}\mathrm{P}(S{=}\text{`}a\text{'} \mid F{=}\text{`}n\text{'}) &= 1/3\\ \mathrm{P}(F{=}\text{`}n\text{'} \mid S{=}\text{`}a\text{'}) &= 1/4\end{aligned}$$

The correct way to think about these is that the first says, "if we restrict consideration to samples where the first letter is '*n*', then the probability of that the second letter is '*a*' is 1/3." However, if we restrict consideration to those samples whose second letter is '*a*', then the probability that the first letter is '*n*' is 1/4. To take a more extreme example, the probability of a medical diagnosis D being "flu", given the symptom S being "elevated body temperature", is small. In the world of patients with high temperatures, most just have a cold, not the flu. But vice versa, the probability of high temperature given flu is very large.

1.2.4 Independence

Notice that in the previous example, the marginal probability $\mathrm{P}(S{=}\text{`}o\text{'})$ of the event $S{=}\text{`}o\text{'}$ differs from its conditional probabilities $\mathrm{P}(S{=}\text{`}o\text{'} \mid F{=}\text{`}n\text{'})$ and $\mathrm{P}(S{=}\text{`}o\text{'} \mid F{=}\text{`}t\text{'})$. This is because these conditional probabilities restrict attention to different sets of samples, and the distribution of second letters S differs on these sets. Statistical dependence captures this interaction. Informally, two events are dependent if the probability of one depends on whether the other occurs; if there is no such interaction then the events are independent.

Formally, we define independence as follows. Two events E_1 and E_2 are *independent* if and only if

$$\mathrm{P}(E_1, E_2) = \mathrm{P}(E_1)\,\mathrm{P}(E_2).$$

If $\mathrm{P}(E_2) > 0$ it is easy to show that this is equivalent to

$$\mathrm{P}(E_1 \mid E_2) = \mathrm{P}(E_1)$$

which captures the informal definition of independence above.

Two random variables V_1 and V_2 are independent if and only if all of the events $V_1{=}v_1$ are independent of all of the events $V_2{=}v_2$ for all v_1 and v_2. That is, V_1 and V_2 are independent if and only if:

$$\mathrm{P}(V_1, V_2) \;=\; \mathrm{P}(V_1)\,\mathrm{P}(V_2)$$

or equivalently, if $\mathrm{P}(V_2) > 0$,

$$\mathrm{P}(V_1 \mid V_2) \;=\; \mathrm{P}(V_1)$$

Example 1.3: The random variables F and S in Example 1.2 on page 10 are dependent because $\mathrm{P}(F \mid S) \neq \mathrm{P}(F)$. But if the urn contained only four strips of paper showing '*nab*', '*no*', '*tap*' and '*tot*', then F and S would be independent because $\mathrm{P}(F \mid S) = \mathrm{P}(F) = 1/2$, no matter what values F and S take.

1.2.5 Expectations of random variable

If a random variable X ranges over numerical values (say integers or reals), then its *expectation* or *expected value* is just a weighted average of these values, where the weight on value X is $\mathrm{P}(X)$. More precisely, the *expected value* $\mathrm{E}[X]$ of a random variable X is

$$\mathrm{E}[X] \;=\; \sum_{x \in \mathcal{X}} x\,\mathrm{P}(X{=}x)$$

where $\mathcal{X}$ is the set of values that the random variable X ranges over. Conditional expectations are defined in the same way as conditional probabilities, namely by restricting attention to a particular conditioning event, so

$$\mathrm{E}[\,X \mid Y{=}y\,] \;=\; \sum_{x \in \mathcal{X}} x\,\mathrm{P}(\,X{=}x \mid Y{=}y\,)$$

Expectation is a linear operator, so if $X_1, \ldots, X_n$ are random variables then

$$\mathrm{E}[\,X_1 + \ldots + X_n] \;=\; \mathrm{E}[X_1] + \ldots + \mathrm{E}[X_n]$$

Example 1.4: Suppose X is a random variable generated by throwing a fair six-sided die. Then the expected value of X is

$$\mathrm{E}[X] \;=\; \frac{1}{6} + \frac{2}{6} + \ldots + \frac{6}{6} \;=\; 3.5$$

Now suppose that X_1 and X_2 are random variables generated by throwing two fair six-sided dice. Then the expected value of their sum is

$$\mathrm{E}[X_1 + X_2] \;=\; \mathrm{E}[X_1] + \mathrm{E}[X_2] \;=\; 7$$

In this book we frequently deal with the expectation of an event V, such as that in some translation the English word '*cat*' is translated to the French word '*chat.*' In this case the "value" of the event occurring is one, and thus the expectation of how often the event occurs is simply the sum over each member of the sample space (in our case each pair English word, French word) of the probably of the event occurring. That is:

$$\begin{aligned}\mathrm{E}[V] &= \sum_{x\in\mathcal{X}} 1\cdot \mathrm{P}(X{=}x)\\ &= \sum_{x\in\mathcal{X}} \mathrm{P}(X{=}x)\end{aligned}$$

1.3 Modeling documents with unigrams

Let's think for a moment about *language identification* — determining the language in which a document is written. For simplicity, let's assume we know ahead of time that the document is either English or French, and we need only determine which; however the methods we use generalize to an arbitrary number of languages. Further, assume that we have a *corpus* or collection of documents we know are definitely English, and another corpus of documents that we know are definitely French. How could we determine if our unknown document is English or French?

If our unknown language document appeared in one of the two corpora (one corpus, two corpora), then we could use that fact to make a reasonably certain identification of its language. But this is extremely unlikely to occur; there are just too many different documents.

1.3.1 Documents as sequences of words

The basic idea we pursue here is to sort the documents in each of the corpora into smaller sequences and compare the pieces that occur in the unknown language document with those that occur in each of the two known language corpora.

It turns out that it is not too important what the pieces are, so long as there is likely to be a reasonable overlap between the pieces that occur in the unknown language document and the pieces that occur in the corresponding corpus. We follow most work in computational linguistics and take the pieces to be words, but nothing really hinges on this (e.g., you can also take them to be characters).

Thus, if $\mathcal{W}$ is the set of possible words then a document is a sequence $\boldsymbol{w} = (w_1, \ldots, w_n)$ where n (the length of the document) may vary and each piece $w_i \in \mathcal{W}$. (Note that we generally use bold-faced symbols (e.g., $\boldsymbol{w}$) to denote vectors or sequences.)

A few technical issues need to be dealt with when working on real documents. Punctuation and typesetting information can either be ignored or treated as pseudo-words (e.g., a punctuation symbol is regarded as a one-character word). Exactly how to break up a text into words can also be an issue: it is sometimes unclear whether something is one word or two: for example, is '*doesn't*' a single word or is it '*does*' followed by '*n't*'? In many applications it does not matter exactly what you do so long as you are consistent.

It also often simplifies matters to assume that the set of possible words is finite. We do so here. You might think this is reasonable — after all, a dictionary seems to list all the words in a language — but if we count things such as geographical locations and names (especially company and product names) as words (and there's no reason not to), then it is clear that new words are being coined all the time.

A standard way to define a finite set of possible words is to collect all the words appearing in your corpus and declare this set, together with a novel symbol (say '∗U∗', but any novel sequence of characters will do) called the *unknown word*, to be the set of possible words. That is, if $\mathcal{W}_0$ is the set of words that appear in your corpus, then the set of possible words is $\mathcal{W} = \mathcal{W}_0 \cup \{*\mathrm{U}*\}$. Then we preprocess every document by replacing every word not in $\mathcal{W}_0$ with '∗U∗'. The result is a document in which every word is a member of the finite set of possibilities $\mathcal{W}$.

1.3.2 Language models as models of possible documents

Now we set about building our language models. Formally a *language model* is a probability distribution over possible documents that specifies the probability of that document in the language (as opposed to all the other documents that one might encounter).

More specifically, let the sample space be the set of possible documents and introduce two random variables N and $\boldsymbol{W}$, where $\boldsymbol{W} = (W_1, \ldots, W_N)$ is the sequence of words in the document and N is its length (i.e., the number of words). A language model is then just a probability distribution $\mathrm{P}(\boldsymbol{W})$.

But which one? Ideally we would like $\mathrm{P}(\boldsymbol{W})$ to be the "true" distribution over documents $\boldsymbol{W}$, but we don't have access to this. (Indeed, it is not clear that it even makes sense to talk of a "true" probability distribution over English documents.) Instead, we assume we have a *training corpus* of documents $\boldsymbol{d}$ that contains representative samples from $\mathrm{P}(\boldsymbol{W})$ and that we use to learn $\mathrm{P}(\boldsymbol{W})$. We also assume that $\mathrm{P}(\boldsymbol{W})$ is defined by a specific mathematical formula or model. This model has some adjustable or free parameters $\boldsymbol{\theta}$, and by changing these we define different language models. We use $\boldsymbol{d}$ to *estimate* or set the values of the parameters $\boldsymbol{\theta}$ in such a way that the resulting language model $\mathrm{P}(\boldsymbol{W})$ is (we hope) close to the true distribution of documents in the language.

Notation: We designate model parameters with Greek letters. However, we use Greek letters for other things as well. The meaning should always be clear.

1.3.3 Unigram language models

Finding the best way to define a language model $\mathrm{P}(\boldsymbol{W})$ and to estimate its parameters $\boldsymbol{\theta}$ is still a major research topic in computational linguistics. In this section we introduce an extremely simple class called unigram language models. These are not especially good models of documents, but they can often be good enough to perform simple tasks such as language identification. (Indeed, one of lessons of computational linguistics is that a model doesn't always need to model everything in order to be able to solve many tasks quite well. Figuring out which features are crucial to a task and which can be ignored is a large part of computational linguistics research.)

A *unigram language model* assumes that each word W_i is generated independently of the other words. (The name "unigram" comes from the fact that the model's basic units are single words; in section 1.4.1 on page 25 we see how to define bigram and trigram language models, whose basic units are pairs and triples of words, respectively.) More precisely, a unigram language model defines a distribution over documents as follows:

$$\mathrm{P}(\boldsymbol{W}) = \mathrm{P}(N) \prod_{i=1}^{N} \mathrm{P}(W_i) \tag{1.1}$$

where $\mathrm{P}(N)$ is a distribution over document lengths and $\mathrm{P}(W_i)$ is the probability of word W_i. You can read this formula as an instruction for generating a document $\boldsymbol{W}$: first pick its length N from the distribution $\mathrm{P}(N)$ and then independently generate each of its words W_i from the distribution $\mathrm{P}(W_i)$. Models that can be understood as generating their data in this way are called *generative models*. The "story" of how we generate a document by first picking the length, etc., is called a *generative story* about how the document could have been created.

A unigram model assumes that the probability of a word $\mathrm{P}(W_i)$ does not depend on its position i in the document, i.e., $\mathrm{P}(W_i{=}w) = \mathrm{P}(W_j{=}w)$ for all i, j in $1, \ldots, N$. This means that all words are generated by the same distribution over words, so we only have one distribution to learn. This assumption is clearly false (why?), but it does make the unigram model simple enough to be estimated from a modest amount of data.

We introduce a parameter θ_w for each word $w \in \mathcal{W}$ to specify the probability of w, i.e., $\mathrm{P}(W_i{=}w) = \theta_w$. (Since the probabilities of all words must sum to one, it is necessary that the parameter vector $\boldsymbol{\theta}$ satisfy $\sum_{w \in \mathcal{W}} \theta_w = 1$.) This means that we can rewrite the unigram model (1.1) as follows:

$$\mathrm{P}(\boldsymbol{W}{=}\boldsymbol{w}) \quad = \quad \mathrm{P}(N) \prod_{i=1}^{N} \theta_{w_i}. \tag{1.2}$$

We have to solve two remaining problems before we have a fully specified unigram language model. First, we have to determine the distribution $\mathrm{P}(N)$ over document lengths N. Second, we have to find the values of the parameter vector $\boldsymbol{\theta}$ that specifies the probability of the words. For our language identification application we assume $\mathrm{P}(N)$ is the same for all languages, so we can ignore it. (Why can we ignore it if it is the same for all languages?)

1.3.4 Maximum likelihood estimates of unigram parameters

We now turn to estimating the vector of parameters $\boldsymbol{\theta}$ of a unigram language model from a corpus of documents $\boldsymbol{d}$. The field of statistics is concerned with problems such as these. Statistics can be quite technical, and while we believe every practicing computational linguist should have a thorough grasp of the field, we don't have space for anything more than a cursory discussion here. Briefly, a *statistic* is a function of the data (usually one that describes or summarizes some aspect of the data) and an *estimator* for a parameter is a statistic whose value is intended to approximate the value of that parameter. (This last paragraph is for those of you who have always wondered where in "probability and statistics" the first leaves off and the second begins.)

Returning to the problem at hand, there are a number of plausible estimators for $\boldsymbol{\theta}$, but since θ_w is the probability of generating word w, the "obvious" estimator sets θ_w to the relative frequency of word w in the corpus $\boldsymbol{d}$. In more detail, we imagine that our corpus is one long sequence of words (formed by concatenating all of our documents together) so we can treat $\boldsymbol{d}$ as a vector. Then the maximum likelihood estimator sets θ_w to:

$$\hat{\theta}_w = \frac{n_w(\boldsymbol{d})}{n_\circ(\boldsymbol{d})} \tag{1.3}$$

Notation: $n_w(\boldsymbol{d})$ is the number of times that word w occurs in $\boldsymbol{d}$ and $n_\circ(\boldsymbol{d}) = \sum_{w \in \mathcal{W}} n_w(\boldsymbol{d})$ is the total number of words in $\boldsymbol{d}$. More generally, $\circ$ as a subscript to n is always a count over all possibilities.

Notation: Maximum likelihood distributions are indicated by a "caret" over the parameter name.

Example 1.5: Suppose we have a corpus size $n_\circ(\boldsymbol{d}) = 10^7$. Consider two words, '*the*' and '*equilateral*' with counts $2{\cdot}10^5$ and 2, respectively. Their maximum likelihood estimates are 0.02 and $2{\cdot}10^{-7}$.

We shall see there are good reasons *not* to use this estimator for a unigram language model. But first we look more closely at its name — the *maximum likelihood* estimator for $\boldsymbol{\theta}$.

The maximum likelihood principle is a general method for deriving estimators for a wide class of models. The principle says: to estimate the value of a parameter $\boldsymbol{\theta}$ from data x, select the value $\hat{\boldsymbol{\theta}}$ of $\boldsymbol{\theta}$ that makes x as likely as possible. In more detail, suppose we wish to estimate the parameter $\boldsymbol{\theta}$ of a probability distribution $\mathrm{P}_{\boldsymbol{\theta}}(X)$ given data x (i.e., x is the observed value for X). Then the maximum likelihood estimate $\hat{\boldsymbol{\theta}}$ of $\boldsymbol{\theta}$ is value of $\boldsymbol{\theta}$ that maximizes the *likelihood function* $L_x(\boldsymbol{\theta}) = \mathrm{P}_{\boldsymbol{\theta}}(x)$. (The value of the likelihood function is equal to the probability of the data, but in a probability distribution the parameter $\boldsymbol{\theta}$ is held fixed while the data x varies, while in the likelihood function the data is fixed while the parameter is varied. To take a more concrete example, imagine two computer programs. One, F(date), takes a date and returns the lead article in the New York Times for that date. The second, G(newspaper), returns the lead article for June 2, 1946 for whatever newspaper is requested. These are very different programs, but they both have the property that F(6.2.1946)=G(NYT).)

To get the maximum likelihood estimate of $\boldsymbol{\theta}$ for the unigram model, we need to calculate the probability of the training corpus $\boldsymbol{d}$. It is not hard to show that the likelihood function for the unigram model (1.2) is:

$$L_{\boldsymbol{d}}(\boldsymbol{\theta}) = \prod_{w \in \mathcal{W}} \theta_w^{n_w(\boldsymbol{d})}$$

where $n_w(\boldsymbol{d})$ is the number of times word w appears in $\boldsymbol{d}$, and we have ignored the factor concerning the length of $\boldsymbol{d}$ because it does not involve $\boldsymbol{\theta}$ and therefore does not affect $\hat{\boldsymbol{\theta}}$. To understand this formula, observe that it contains a factor θ_w for each occurrence of word w in $\boldsymbol{d}$.

You can show using multivariate calculus that the $\hat{\boldsymbol{\theta}}$ that simultaneously maximizes L_D and satisfies $\sum_{w \in \mathcal{W}} \theta_w = 1$ is nothing other than the relative frequencies of each w, as given in (1.3).

Example 1.6: Consider the "document" (we call it ♡) consisting of the phrase '*I love you*' one hundred times in succession:

$$\begin{aligned} L_{\heartsuit}(\boldsymbol{\theta}) &= (\theta_i)^{n_i(\heartsuit)} \cdot (\theta_{love})^{n_{love}(\heartsuit)} \cdot (\theta_{you})^{n_{you}(\heartsuit)} \\ &= (\theta_i)^{100} \cdot (\theta_{love})^{100} \cdot (\theta_{you})^{100} \end{aligned}$$

The θ_ws in turn are all 100/300=1/3, so

$$\begin{aligned} L_{\heartsuit}(\boldsymbol{\theta}) &= (1/3)^{100} \cdot (1/3)^{100} \cdot (1/3)^{100} \\ &= (1/3)^{300} \end{aligned}$$

1.3.5 Sparse-data problems and smoothing

Returning to our example of language models for language identification, recall that our unknown document is very likely to contain words that don't occur in either our English or French corpora. It is easy to see that if w is a word that does not appear in our corpus $\boldsymbol{d}$ then the maximum likelihood estimate $\hat{\theta}_w = 0$, as $n_w(\boldsymbol{d}) = 0$. And this, together with Equation 1.2 on page 17, means that $\mathrm{P}(\boldsymbol{w}) = 0$ if the document $\boldsymbol{w}$ contains an unknown word.

To put this another way, in Section 1.3.1 we said that we would define $\mathcal{W}$, our vocabulary, as all the words in our training data plus $*$U$*$. By definition, $*$U$*$ does not appear in our training data, so the maximum likelihood estimate assigns it zero probability.

This is fatal in many applications, including our language identification task. Just because a word does not appear in our corpus $\boldsymbol{d}$ does not mean it cannot appear in the documents we want to classify. This is what is called a *sparse-data* problem: our training data $\boldsymbol{d}$ does not have the exact range of phenomena found in the documents we ultimately intend to analyze. And the problem is more general than unknown words: it turns out that maximum likelihood estimates $\hat{\boldsymbol{\theta}}$ have a tendency for *over-fitting*, i.e., modeling the training data $\boldsymbol{d}$ accurately but not describing novel data well at all. More specifically, maximum likelihood estimators select $\boldsymbol{\theta}$ to make the training data $\boldsymbol{d}$ as likely as possible, but for our language classification application we really want something else: namely, to make all the other documents we haven't yet seen from the same language as $\boldsymbol{d}$ as likely as possible.

The standard way to address over-fitting is by *smoothing*. If you think of a probability function as a landscape with peaks and valleys, smoothing is a kind of Robin-Hood process that steals mass from the rich peaks and gives it to the poor valleys, all the while ensuring that the resulting distribution still sums to one. In many computational linguistic applications, maximum likelihood estimators produce distributions that are too tightly tuned to their training data and smoothing often improves the performance of models on novel data. There are many ways to smooth and the precise method used can have a dramatic effect on the performance. But while there are many different ways to redistribute, in computational linguistics as in economics, only a few of them work well!

A popular method for smoothing the maximum likelihood estimator in (1.3) is to add a positive number α_w called a *pseudo-count* to each word w's empirical frequency $n_w(\boldsymbol{d})$ in (1.3), readjusting the denominator so that the revised estimates of $\boldsymbol{\theta}$ still sum to 1. (You can view the pseudo-counts $\boldsymbol{\alpha}$ as counts coming from hypothetical or pseudo-data that are added to the counts $\boldsymbol{n}(\boldsymbol{d})$ that we get from the real data.) This means that our smoothed maximum likelihood estimator $\widetilde{\boldsymbol{\theta}}$ is:

$$\widetilde{\theta}_w = \frac{n_w(\boldsymbol{d}) + \alpha_w}{n_\circ(\boldsymbol{d}) + \alpha_\circ} \tag{1.4}$$

where $\alpha_\circ = \sum_{w \in \mathcal{W}} \alpha_w$ is the sum over all words of the pseudo-counts. With this estimator, $\widetilde{\theta}_w$ for all w is always greater than zero even if $n_w(\boldsymbol{d}) = 0$, so long as $\alpha_w > 0$. This smoothing method is often called *add-alpha smoothing*.

Notation: A tilde over parameter values indicates that the parameters define a smoothed distribution.

It is standard to *bin* or group words into equivalence classes and assign the same pseudo-count to all words in the same equivalence class. This way we have fewer pseudo-count parameters to estimate. For example, if we group all the words into one single equivalence class, then there is a single pseudo-count value $\alpha = \alpha_w$ that is used for all $w \in \mathcal{W}$ and only a single parameter need be estimated from our held-out data. This works well for a surprising number of applications. Indeed, often the actual value of α is not critical and we just set it to one. (This is called *add-one smoothing* or alternatively *Laplace smoothing.*)

Example 1.7: Let us assume that all w get the same smoothing constant. In this case Equation 1.4 simplifies to;

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

Suppose we set $\alpha = 1$ and we have $|\mathcal{W}| = 100,000$ and $n_\circ(\boldsymbol{d}) = 10^7$. As in Example 1.5, the two words '*the*' and '*equilateral*' have counts $2{\cdot}10^5$ and 2, respectively. Their maximum likelihood estimates again are 0.02 and $2{\cdot}10^{-7}$. After smoothing, the estimate for '*the*' hardly changes

$$\widetilde{\theta}_{\text{the}} = \frac{2{\cdot}10^5 + 1}{10^7 + 10^5} \approx 0.02$$

while the estimate for '*equilateral*' goes up by 50%:

$$\widetilde{\theta}_{\text{equilateral}} = \frac{2+1}{10^7 + 10^5} \approx 3{\cdot}10^{-7}$$

Of course, we now face another estimation problem, because we need to specify the values of the pseudo-counts $\boldsymbol{\alpha}$. It is reasonable to try to estimate these from our data $\boldsymbol{d}$, but in this case maximum likelihood estimations are of no use: they are just going to set $\boldsymbol{\alpha}$ to zero!

On reflection, this should be no surprise: if w does not occur in the data $\boldsymbol{d}$, then the maximum likelihood estimator sets $\hat{\theta}_w = 0$ because to do otherwise would "waste" probability on w. So if we select $\boldsymbol{\alpha}$ in our smoothed estimator to maximize the likelihood of $\boldsymbol{d}$ then $\alpha_w = 0$ for exactly the same reason. In summary, setting $\theta_w = 0$ when $n_w(\boldsymbol{d}) = 0$ is an eminently sensible thing to do if we are concerned only with maximizing the likelihood of our data $\boldsymbol{d}$. The problem is that our data $\boldsymbol{d}$ is *not* representative of the occurrence or non-occurrence of any specific word w in brand new documents.

One standard way to address this problem is to collect an additional corpus of training documents $\boldsymbol{h}$, the *held-out corpus*, and use them to set the smoothing pseudo-count parameters $\boldsymbol{\alpha}$. This makes sense because one of the primary goals of smoothing is to correct, or at least ameliorate, problems caused by sparse data, and a second set of training data $\boldsymbol{h}$ gives us a way of saying just how bad $\boldsymbol{d}$ is as a representative sample. (The same reasoning tells us that $\boldsymbol{h}$ should be "fresh data", disjoint from the primary training data $\boldsymbol{d}$.)

In fact, it is standard at the onset of a research project to split the available data into a primary training corpus $\boldsymbol{d}$ and a secondary held-out training corpus $\boldsymbol{h}$, and perhaps a third *test set* $\boldsymbol{t}$ as well, to be used to evaluate different models trained on $\boldsymbol{d}$ and $\boldsymbol{h}$. For example, we would want to evaluate a language identification system on its ability to discriminate novel documents, and a test set $\boldsymbol{t}$ disjoint from our training data $\boldsymbol{d}$ and $\boldsymbol{h}$ gives us a way to do this. Usually $\boldsymbol{d}$ is larger than $\boldsymbol{h}$ as it needs to estimate many parameters (for the unigram model, the total number of word types), while typically $\boldsymbol{h}$ sets very few (one if all the α's are the same). Similarly the we need less testing than training data. A standard split might be 80% for training and 10% each for the held-out and testing sets.

1.3.6 Estimating the smoothing parameters

We now describe how to use a held-out corpus $\boldsymbol{h}$ to estimate the pseudo-counts $\boldsymbol{\alpha}$ of the smoothed maximum likelihood estimator described on page 20. We treat $\boldsymbol{h}$, like $\boldsymbol{d}$, as a long vector of words obtained by concatenating all of the documents in the held-out corpus. For simplicity we assume that all the words are grouped into a single bin, so there is a single pseudo-count $\alpha = \alpha_w$ for all words w. This means that our smoothed maximum likelihood estimate (1.4) for $\boldsymbol{\theta}$ simplifies to Equation 1.5.

Example 1.8: Suppose our training data $\boldsymbol{d}$ is $\heartsuit$ from Example 1.6 and the held-out data $\boldsymbol{h}$ is $\heartsuit'$, which consists of eight copies of '*I love you*' plus one copy each of '*I can love you*' and '*I will love you*'. When we preprocess the held-out data both '*can*' and '*will*' become ∗U∗, so $\mathcal{W} = \{$ i love you ∗U∗$\}$. We let $\alpha = 1$.

Now when we compute the likelihood of $\heartsuit'$ our smoothed θs are as follows:

$$\begin{aligned}
\widetilde{\theta}_{\mathrm{i}} &= \frac{100+1}{300+4} \\
\widetilde{\theta}_{\mathrm{love}} &= \frac{100+1}{300+4} \\
\widetilde{\theta}_{\mathrm{you}} &= \frac{100+1}{300+4} \\
\widetilde{\theta}_{*\mathrm{U}*} &= \frac{1}{300+4}
\end{aligned}$$

These are then substituted into our normal likelihood equation.

We seek the value $\hat{\alpha}$ of α that maximizes the likelihood $L_{\boldsymbol{h}}$ of the *held-out* corpus $\boldsymbol{h}$ for reasons explained in section 1.3.5 on page 19.

$$\begin{aligned}
\hat{\alpha} &= \underset{\alpha}{\operatorname{argmax}}\, L_{\boldsymbol{h}}(\alpha) \\
L_{\boldsymbol{h}}(\alpha) &= \prod_{w \in \mathcal{W}} \left(\frac{n_w(\boldsymbol{d}) + \alpha}{n_\circ(\boldsymbol{d}) + \alpha|\mathcal{W}|} \right)^{n_w(\boldsymbol{h})} \qquad (1.5)
\end{aligned}$$

Argmax (pronounced just as it is spelled) is the abbreviation of "argument maximum". $\arg\max_x f(x)$ has as its value the value of x that makes $f(x)$ as large as possible. Consider the function $f(x) = 1 - x^2$. Just a bit of thought is required to realize that the maximum of $f(x)$ is 1 and occurs when $x = 0$, so $\arg\max_x 1 - x^2 = 0$.

With that out of the way, let us return to the contemplation of Equation 1.5. This formula simply says that the likelihood of the held-out data is the product of the probability of each word token in the data. (Make sure you see this.) Again we have ignored the factor in $L_{\boldsymbol{h}}$ that depends on the length of $\boldsymbol{h}$, since it does not involve α. If you plug in lots of values for α you find that this likelihood function has a single peak. (This could have been predicted in advance.) Thus you can try out values to home in on the best value. A *line-search* routine (such as *Golden Section search*) does this for you efficiently. (Actually, the equation is simple enough that $\hat{\alpha}$ can be found analytically.) But be aware that the likelihood of even a moderate-sized corpus can become extremely small, so to avoid underflow you should compute and optimize the *logarithm* of the likelihood rather than the likelihood itself.

1.4 Contextual dependencies and *n*-grams

The previous section described unigram language models, in which the words (or whatever the basic units of the model are) are each generated as independent entities. This means that unigram language models have no way of capturing contextual dependencies among words in the same sentence or document. For a comparatively simple task like language identification this may work fine. But remember the example at the start of this chapter where a speech recognition system would want to distinguish the fluent '*I saw a rose*' from the disfluent '*I saw arose.*'

In fact, there are a large number of different kinds of contextual dependencies that a unigram model does not capture. There are clearly dependencies between words in the same sentence that are related to syntactic and other structure. For example, '*students eat bananas*' is far more likely than '*bananas eat students*', mainly because students are far more likely to be eaters than eaten while the reverse holds for bananas, yet a unigram model would assign these two sentences the same probability.

There are also dependencies that hold intersententially as well as intrasententially. Some of these have to do with topicality and discourse structure. For example, the probability that a sentence contains the words '*court*' or '*racquet*' is much higher if a preceding sentences contains '*tennis*'. And while the probability of seeing any given name in the second half of a random document is very low, the probability of seeing a name in the second half of a document *given* that it has occurred in the first half of that document is generally many times higher (i.e., names are very likely to be repeated).

Methods have been proposed to capture all these dependencies (and many more), and identifying the important contextual dependencies and coming up with language models that capture them well is still one of the central topics in computational linguistics. In this section we describe how to capture one of simplest yet most important kinds of contextual dependencies: those that hold between a word and its neighbors. The idea is that we slide a window of width n words over the text, and the overlapping sequences of length n that we see through this window are called *n-grams*. When n=2 the sequences are called *bigrams* and when n=3 the sequences are called *trigrams*. The basic idea is illustrated for n=2 in Figure 1.1.

It turns out that a surprising number of contextual dependencies are visible in an n-word window even for quite small values of n. For example, a bigram language model can distinguish '*students eat bananas*' from

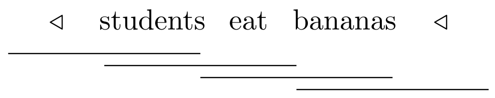

Figure 1.1: The four bigrams extracted by a bigram language from the sentence '*students eat bananas*', padded with '◁' symbols at its beginning and end.

'*bananas eat students*'. This is because most linguistic dependencies tend to be local in terms of the word string, even though the actual dependencies may reflect syntactic or other linguistic structure.

For simplicity we focus on explaining bigram models in detail here (i.e., n=2), but it is quite straightforward to generalize to larger values of n. Currently most language models are built with $n = 3$ (i.e., trigrams), but models with n as high as 7 are not uncommon. Of course, sparse-data problems become more severe as n grows, and the precise smoothing method used can have a dramatic impact on how well the model generalizes.

1.4.1 Bigram language models

A bigram language model is a generative model of sequences of tokens. In our applications the tokens are typically words and the sequences are sentences or documents. Informally, a bigram language model generates a sequence one word at a time, starting with the first word and then generating each succeeding word conditioned on the previous one.

We can derive the bigram model via a sequence of equations and simplifying approximations. Ignoring the length of $\boldsymbol{W}$ for the moment, we can decompose the joint probability of the sequence $\boldsymbol{W}$ into a product of conditional probabilities (this operation is called the *chain rule* and is used many times in this text):

$$\begin{aligned} \mathrm{P}(\boldsymbol{W}) &= \mathrm{P}(W_1, \ldots, W_n) \\ &= \mathrm{P}(W_1)\mathrm{P}(W_2|W_1)\ldots\mathrm{P}(W_n|W_{n-1}, \ldots, W_1) \end{aligned} \tag{1.6}$$

where n is the length of $\boldsymbol{W}$. Now we make the so-called *Markov assumption*, which is that

$$\mathrm{P}(W_i \mid W_{i-1}, \ldots, W_1) \;=\; \mathrm{P}(W_i \mid W_{i-1}) \tag{1.7}$$

for all positions $i \in 2, \ldots, N$. Putting (1.6) and (1.7) together, we have:

$$\mathrm{P}(\boldsymbol{W}) \quad = \quad \mathrm{P}(W_1) \prod_{i=2}^{n} \mathrm{P}(W_i \mid W_{i-1}) \tag{1.8}$$

In addition, we assume that $\mathrm{P}(W_i \mid W_{i-1})$ does not depend on the position i, i.e., that $\mathrm{P}(W_i \mid W_{i-1}) = \mathrm{P}(W_j \mid W_{j-1})$ for all $i, j \in 1, \ldots n$.

We can both simplify the notation and generate the length of $\boldsymbol{W}$ if we imagine that each sequence is surrounded or padded with special *stop symbols* '$\triangleleft$' that appear nowhere else in the string. What we mean by this is that if $\boldsymbol{w} = (w_1, \ldots, w_n)$ then we define $w_0 = w_{n+1} = \triangleleft$. We generate the first word in the sentence with the context '$\triangleleft$' (i.e., $\mathrm{P}(W_1) = \mathrm{P}(W_1 | \triangleleft)$, and stop when we generate another '$\triangleleft$', which marks the end of the sequence. The stop symbol '$\triangleleft$' thus has a rather strange status: we treat '$\triangleleft$' as a token, i.e., $\triangleleft \in \mathcal{W}$, even though it never appears in any sequence we generate.

Notation: The $\triangleleft$ symbol can be used for both the start and end of a sequence. However, sometimes we find it clearer to use $\triangleright$ for the start. Nothing depends on this.

In more detail, a *bigram language model* is defined as follows. If $\boldsymbol{W} = (W_1, \ldots, W_n)$ then

$$\mathrm{P}(\boldsymbol{W}) \quad = \quad \prod_{i=1}^{n+1} \mathrm{P}(W_i \mid W_{i-1}) \tag{1.9}$$

where

$$\mathrm{P}(W_i{=}w' \mid W_{i-1}{=}w) \quad = \quad \Theta_{w,w'} \text{ for all } i \text{ in } 1, 2, \ldots, n+1$$

and $\boldsymbol{\Theta} = \{\Theta_{w,w'} : w, w' \in \mathcal{W}\}$ is a matrix of parameters specifying the conditional probability of w' given it is preceded by w. Just as in the unigram model, we assume that these conditional probabilities are time-invariant, i.e., they do not depend on i directly. Because probabilities must sum to one, it is necessary that $\sum_{w' \in \mathcal{W}} \Theta_{w,w'} = 1$ for all $w \in \mathcal{W}$.

Notation: We have many situations in which the parameters of a model are conditional probabilities. Here the parameter is the conditional probability of word given the previous word. Naturally such parameters have two subscripts. We order the subscripts so that the first is the conditioning event (here the previous word (w)) and the second is the conditioned event (w').

Example 1.9: A bigram model assigns the following probability to the string '*students eat bananas*'.

$$\begin{aligned}\mathrm{P}(\text{`}\textit{students eat bananas}\text{'}) &= \Theta_{\triangleleft,students} \quad \Theta_{students,eat} \\ &\quad\; \Theta_{eat,bananas} \quad \Theta_{bananas,\triangleleft}\end{aligned}$$

In general, the probability of a string of length n is a product of $n+1$ conditional probabilities, one for each of the n words and one for the end-of-sentence token '$\triangleleft$'. The model predicts the length of the sentence by predicting where '$\triangleleft$' appears, even though it is not part of the string.

This generalizes to n-gram models as follows. In an n-gram model each word W_i is generated conditional on the $n-1$ word sequence $(W_{i-n+1} \ldots W_{i-1})$ that precedes it, and these conditional distributions are time-invariant, i.e., they don't depend on i. Intuitively, in an n-gram model $(W_{i-n+1} \ldots W_{i-1})$ form the *conditioning context* that is used to predict W_i.

Before going on, a typical misconception should be nipped in its bud. Most students seeing the bigram (or n-gram) model for the first time think that it is innately directional. That is, we start at the beginning of our sentence and condition each word on the previous one. Somehow it would seem very different if we started at the end and conditioned each word on the subsequent one. But, in fact, *we would get exactly the same probability*! You should show this for, say, '*students eat bananas*'. (Write down the four conditional probabilities involved when we use forward bigram probabilities, and then backward ones. Replace each conditional probability by the definition of a conditional probability. Then shuffle the terms.)

It is important to see that bigrams are not directional because of the misconceptions that follow from thinking the opposite. For example, suppose we want a language model to help a speech recognition system distinguish between '*big*' and '*pig*' in a sentence '*The big/pig is ...*'. Students see that one can make the distinction only by looking at the word *after* big/pig, and think that our bigram model cannot do it because somehow it only looks at the word before. But as we have just seen, this cannot be the case because we would get the same answer either way. More constructively, even though both of big/pig are reasonably likely after '*the*', the conditional probabilities P(is|big) and P(is|pig) are very different and presumably strongly bias a speech-recognition model in the right direction.

(For the more mathematically inclined, it is not hard to show that under a bigram model, for all i between 1 and n:

$$\mathrm{P}(W_i{=}w_i \mid W_{i-1}{=}w_{i-1}, W_{i+1}{=}w_{i+1}) \quad = \quad \frac{\Theta_{w_{i-1},w_i}\,\Theta_{w_i,w_{i+1}}}{\sum_{w\in\mathcal{W}} \Theta_{w_{i-1},w}\,\Theta_{w,w_{i+1}}}$$

which shows that a bigram model implicitly captures dependencies from both the left and right simultaneously.)

1.4.2 Estimating the bigram parameters $\boldsymbol{\Theta}$

The same basic techniques we used for the unigram model extend to the bigram and higher-order n-gram models. The maximum likelihood estimate selects the parameters $\widehat{\boldsymbol{\Theta}}$ that make our training data $\boldsymbol{d}$ as likely as possible:

$$\widehat{\Theta}_{w,w'} \quad = \quad \frac{n_{w,w'}(\boldsymbol{d})}{n_{w,\circ}(\boldsymbol{d})}$$

where $n_{w,w'}(\boldsymbol{d})$ is the number of times that the bigram w, w' appears anywhere in $\boldsymbol{d}$ (including the stop symbols '$\triangleleft$'), and $n_{w,\circ}(\boldsymbol{d}) = \sum_{w'\in\mathcal{W}} n_{w,w'}(\boldsymbol{d})$ is the number of bigrams that begin with w. (Except for the odd cases of start and stop symbols, $n_{w,\circ}(\boldsymbol{d}) = n_w(\boldsymbol{d})$, i.e., the number of times that w appears in $\boldsymbol{d}$). Intuitively, the maximum likelihood estimator is the obvious one that sets $\Theta_{w,w'}$ to the fraction of times w' was seen immediately following w.

Example 1.10: Looking at the $\heartsuit'$ corpus again, we find that

$$\begin{aligned}
\widehat{\Theta}_{\triangleleft,i} &= 1\\
\widehat{\Theta}_{i,love} &= 0.8\\
\widehat{\Theta}_{i,will} &= 0.1\\
\widehat{\Theta}_{i,can} &= 0.1\\
\widehat{\Theta}_{can,love} &= 1\\
\widehat{\Theta}_{will,love} &= 1\\
\widehat{\Theta}_{love,you} &= 1\\
\widehat{\Theta}_{you,i} &= 0.9\\
\widehat{\Theta}_{you,\triangleleft} &= 0.1
\end{aligned}$$

The sparse-data problems we noticed with the unigram model in S ection 1.3.5 become more serious when we move to the bigram model. In general, sparse-data problems get worse as we work with n-grams of larger size; if we think of an n-gram model as predicting a word conditioned on the $n-1$ word sequence that precedes it, it becomes increasingly common that the conditioning $n-1$ word sequence occurs only infrequently, if at all, in the training data $\boldsymbol{d}$.

For example, consider a word w that occurs only once in our training corpus $\boldsymbol{d}$ (such word types are extremely common). Then $n_{w,w'}(\boldsymbol{d}) = 1$ for exactly one word w' and is 0 for all other words. This means that the maximum likelihood estimator is $\widehat{\Theta}_{w,w'} = 1$, which corresponds to predicting that w can be followed only by w'. The problem is that we are effectively estimating the distribution over words that follow w from the occurrences of w in $\boldsymbol{d}$, and if there are only very few such occurrences then these estimates are based on very sparse data indeed.

Just as in the unigram case, our general approach to these sparse-data problems is to smooth. Again, a general way to do this is to add a pseudo-count $\beta_{w,w'}$ to the observations $n_{w,w'}$ and normalize, i.e.:

$$\widetilde{\Theta}_{w,w'} = \frac{n_{w,w'}(\boldsymbol{d}) + \beta_{w,w'}}{n_{w,\circ}(\boldsymbol{d}) + \beta_{w,\circ}} \tag{1.10}$$

where $\beta_{w,\circ} = \sum_{w' \in \mathcal{W}} \beta_{w,w'}$. While it is possible to follow what we did for the unigram model and set $\beta_{w,w'}$ to the same value for all $w, w' \in \mathcal{W}$, it is usually better to make $\beta_{w,w'}$ proportional to the smoothed unigram estimate $\widetilde{\theta}_{w'}$; this corresponds to the assumption that, all else equal, we're more likely to see a high-frequency word w' following w than a low-frequency one. That is, we set $\beta_{w,w'}$ in (1.10) as follows:

$$\beta_{w,w'} = \beta \, \widetilde{\theta}_{w'}$$

where β is a single adjustable constant. Plugging this back into (1.10), we have:

$$\widetilde{\Theta}_{w,w'} = \frac{n_{w,w'}(\boldsymbol{d}) + \beta \, \widetilde{\theta}_{w'}}{n_{w,\circ}(\boldsymbol{d}) + \beta} \tag{1.11}$$

Note that if β is positive then $\beta_{w,w'}$ is also, because $\widetilde{\theta}_{w'}$ is always positive. This means our bigram model will not assign probability zero to any bigram, and therefore the probability of all strings are strictly positive.

Example 1.11: Suppose w is '*redistribute*' and we consider two possible next words w', '*food*' and '*pears*', with (assumed) smoothed unigram probabilities 10^{-4} and 10^{-6} respectively. Let β be 1.

Suppose we have never seen the word '*redistribute*' in our corpus. Thus $n_{w,w'}(\boldsymbol{d}) = n_{w,\circ}(\boldsymbol{d}) = 0$ (why?). In this case our estimate of the bigram probabilities reverts to the unigram probabilities.

$$\begin{aligned}
\widetilde{\Theta}_{\text{redistribute,food}} &= \frac{0 + 10^{-4}}{0 + 1} \\
\widetilde{\Theta}_{\text{redistribute,pears}} &= \frac{0 + 10^{-5}}{0 + 1}
\end{aligned}$$

If we have seen '*redistribute*' (say 10 times) and '*redistribute food*' once we get:

$$\begin{aligned}
\widetilde{\Theta}_{\text{redistribute,food,}} &= \frac{1 + 10^{-4}}{10 + 1} \\
\widetilde{\Theta}_{\text{redistribute,pears}} &= \frac{0 + 10^{-5}}{10 + 1}
\end{aligned}$$

The first is very close to the maximum likelihood estimate of 1/10, while the second goes down to about 10^{-6}.

We can estimate the bigram smoothing constant β in the same way we estimated the unigram smoothing constant α, namely by choosing the $\hat{\beta}$ that maximizes the likelihood of a held-out corpus $\boldsymbol{h}$. (As with the unigram model, $\boldsymbol{h}$ must differ from $\boldsymbol{d}$, otherwise $\hat{\beta} = 0$.)

It is easy to show that the likelihood of the held-out corpus $\boldsymbol{h}$ is:

$$L_{\boldsymbol{h}}(\beta) = \prod_{w,w' \in \mathcal{W}} \widetilde{\Theta}_{w,w'}^{n_{w,w'}(\boldsymbol{h})} \tag{1.12}$$

where $\widetilde{\Theta}_{w,w'}$ is given by (1.10), and the product in (1.12) need only range over the bigrams (w, w') that actually occur in $\boldsymbol{h}$. (Do you understand why?) Just as in the unigram case, a simple line search can be used to find the value $\hat{\beta}$ of β that optimizes the likelihood (1.12).

1.4.3 Implementing n-gram language models

It usually simplifies things to assign each word type its own unique integer identifier, so that the corpora $\boldsymbol{d}$ and $\boldsymbol{h}$ can be represented as integer vectors, as can their unigram counts $\boldsymbol{n}(\boldsymbol{d})$ and $\boldsymbol{n}(\boldsymbol{h})$.

Typically, the n-grams extracted from a real corpus (even a very large one) are sparse in the space of possible n-word sequences. We can take advantage of this by using hash tables or similar sparse maps to store the bigram counts $n_{w,w'}(\boldsymbol{d})$ for just those bigrams that actually occur in the data. (If a bigram is not found in the table then its count is zero.) The parameters $\hat{\theta}_w$ and $\hat{\Theta}_{w,w'}$ are computed on the fly.

As we mentioned earlier, because the probability of a sequence is the product of the probability of each of the words that appear in it, the probability of even just a moderately long sequence can become extremely small. To avoid underflow problems, it is wise to compute the logarithm of these probabilities. For example, to find the smoothing parameters α and β you should compute the logarithm of the likelihoods rather than just the likelihoods themselves. In fact, it is standard to report the *negative* logarithm of the likelihood, which is a positive number (why?), and smaller values of the negative log likelihood correspond to higher likelihoods.

1.4.4 Kneser-Ney Smoothing

In many situations, bigram and trigram language models definitely included, best practice is to use *Kneser-Ney smoothing* (KN). We first specify how it computed, and then look at it more closely to understand why it works so well.

Remember where we left off with bigram language models. In Equation 1.11. smoothing was accomplished by adding the terms $\beta\,\widetilde{\theta}_{w'}$ and β to the numerator and denominator respectively, where $\widetilde{\theta}_{w'}$ is the unigram probability estimate for w'.

In KN we change this to

$$\bar{\Theta}_{w,w'} \quad = \quad \frac{n_{w,w'}(\boldsymbol{d}) + \beta\,\kappa_{w'}}{n_{w,\circ}(\boldsymbol{d}) + \beta}$$

That is, we replace the smooth unigram estimate of w', by a new parameter $\kappa_{w'}$. We compute κ from our training data, where $k_{w'}(\boldsymbol{d})$ is the number of different word *types* that precede w' in $\boldsymbol{d}$ and we set

$$\kappa_w = \frac{k_w(\boldsymbol{d})}{k_\circ(\boldsymbol{d})}.$$

Our use of the dot notation here is the same as in previous notation,

$$k_\circ(\boldsymbol{d}) = \sum_{w'} k_{w'}(\boldsymbol{d})$$

and thus the denominator is a normalization factor and κ_w is a probability distribution.

Now for the intuition. We start with the observation that the more often you depend on smoothing to prevent zero probabilities, the larger the smoothing parameter should be. So one would require a larger β to mix in unigrams if we had 100,000 words of data than if we had, say 10^7 words. By "require" here we mean of course that the log-likelihood function for our data would have a maximum value with larger β.

Now consider two words, say '*Francisco*' and '*report*', and assume that they both occur fifty times in our corpus. Thus their smoothed unigram probabilities are the same. But suppose we see two new bigrams, one ending in '*Francisco*', the other in '*report.*' If we use unigrams to smooth, these new bigrams will themselves get the same probability. (Make sure you see why.)

But assigning them equal probability is a very bad estimate. In fact, new bigrams ending in '*report*' should be much more common than those ending in '*Francisco.*' Or to put it another way, we said that a smoothing parameter should be in rough proportion to how often we need to use it to prevent zeros in the data. But we almost never need to back off because of a new word preceding '*Francisco*' because, to a first approximation, all the word tokens preceding it are of the same word type: '*San*'.

This is not true for '*report.*' In its fifty tokens we might see it preceded by '*the*' and '*a*', say, ten times each, but we will have many words that have preceded it only once, e.g., '*financial,*' '*government,*' and '*dreadful.*' This is certain to continue when we look at the test data, where we might encounter '*January*', '*I.B.M.*', etc. KN is designed to capture this intuition by backing off not to the unigram probability, but to a number proportional to the number of different types than can precede the word.

Example 1.12: Suppose in a million word corpus we have forty thousand word types. Some plausible k numbers for '*Francisco*' and '*report*' might be:

$$\begin{aligned} k_{\text{Francisco}}(\boldsymbol{d}) &= 2 \\ k_{\text{report}}(\boldsymbol{d}) &= 32 \\ k_{\circ}(\boldsymbol{d}) &= 100,000 \end{aligned}$$

We will set β to one. A new bigram probability ending in '*Francisco*', say '*Pedro Francisco*' would get the probability estimate:

$$\begin{aligned}\bar{\Theta}_{\text{Pedro,Fransisco}} &= \frac{0+1 \circ 2}{0+1 \cdot 100000} \\ &= \frac{1}{50000}\end{aligned}$$

whereas the equivalent number for '*January report*' is 1/3125.

If you are anything like your authors, you still might be having trouble getting your head around KM smoothing. Another good way to understand it is to look at how your algorithms would change if you converted from smoothing with unigrams to KN. Suppose you have a function that, given a bigram, updates the training parameters, e.g., addBigram(w_{i-1}, w_i). It would look something like this:

1. $n_{w_{i-1},w_i} += 1$
2. $n_{w_i} += 1$
3. $n_{w_\circ} += 1$

Here is the function for KN smoothing:

1. If $n_{w_{i-1},w_i} = 0$
   (a) $k_{w_i} += 1$
   (b) $k_{w_\circ} += 1$
2. $n_{w_{i-1},w_i} += 1$
3. $n_{w_i} += 1$

That is, instead of incrementing the back-off counters for every new word, you do it only when you have not seen its bigram before.

One last point. Just as we previously smoothed unigram counts with α to account for unknown words, we need to do so when using KN. That is, rather than κ_w we use $\widetilde{\kappa}_w$, where

$$\widetilde{\kappa}_w = \frac{k_w(\boldsymbol{d}) + \alpha}{k_\circ(\boldsymbol{d}) + \alpha|\mathcal{W}|}$$

1.4.5 The noisy channel model

We first noted the need for language modeling in conjunction with speech recognition. We did this in an intuitive way. Good speech recognition needs to distinguish fluent from disfluent strings in a language, and language models can do that.

In fact, we can recast the speech recognition problem in a way that makes language models not just convenient but (almost) inevitable. From a probabilistic point of view, the task confronting a speech recognition system is to find the most likely string S in a language given the acoustic signal A. We write this formally as follows:

$$\arg\max_S \mathrm{P}(S|A) \tag{1.13}$$

We now make the following transformations on this term:

$$\begin{aligned}\arg\max_S \mathrm{P}(S|A) &= \arg\max_S \frac{\mathrm{P}(S)\mathrm{P}(A|S)}{\mathrm{P}(A)} \\ &= \arg\max_S \mathrm{P}(S)\mathrm{P}(A|S) \qquad (1.14)\end{aligned}$$

If you ignore the "$\arg\max_S$" in the first line, this is just *Bayes' law*. You should be able to derive it by starting with the definition of conditional probability and then using the chain law on the numerator.

In the second line we drop the $\mathrm{P}(A)$ in the denominator. We can do this because as we vary the S to find the maximum probability the denominator stays constant. (It is, remember, just the sound signal we took in.)

Now consider the two terms on the right-hand side of Equation 1.14. The first is our language model, the second is called the *acoustic model*. That a language model term arises so directly from the definition of the speech-recognition problem is what we meant when we said that language modeling is almost inevitable. It could be avoided, but all serious speech-recognition systems have one.

This set of transformations has its own name, the *noisy channel model*, and it is a staple of NLP. Its name refers to it is origins in communication theory. There a signal goes in at one end of a communication channel and comes out at the other slightly changed. The process that changes it is called *noise*. We want to recover the clean message C given the noisy message N. We do so using the noisy channel model:

$$\mathrm{P}(C|N) \propto \mathrm{P}(C)\mathrm{P}(N|C) \Rightarrow \arg\max_C \mathrm{P}(C|N) = \arg\max_C \mathrm{P}(C)\mathrm{P}(N|C)$$

The first term is is called a *source model* (a probabilistic model of the input, or source), while the second is called a *channel model* (a model of how noise affects the communication channel). When we talk about speech recognition we replace these terms with the more specific *language model* and *acoustic model*.

1.5 Exercises

Exercise 1.1: In what follows w and w' are word types and $\boldsymbol{d}$ is a end-of-sentence padded corpus. True or false:

1. If $n_w(\boldsymbol{d}) = 1$ then there exists exactly one w' such that $n_{w,w'} = 1$.
2. If there is exactly one w, w' pair such that $n_{w,w'} = 1$, then $n_w(\boldsymbol{d}) = 1$.
3. Under an unsmoothed unigram model with parameters $\boldsymbol{\theta}$ trained from a corpus $\boldsymbol{c}$, if $L_{\boldsymbol{d}}(\boldsymbol{\theta}) = 0$ then there must be a w such that $n_w(\boldsymbol{d}) > 0$ and $n_w(\boldsymbol{c}) = 0$.

Exercise 1.2: Consider a corpus $\boldsymbol{d}$ comprising the following sentences:

The dog bit the man
The dog ate the cheese
The mouse bit the cheese
The mouse drank coffee
The man drank tea

Assuming each sentence is padded with the stop symbol on both sides and ignoring capitalization, compute the smoothed bigram probability of "The mouse ate the potato". Use $\alpha = 1$ and $\beta = 10$. Show how you compute the smoothed bigram probabilities for all five words.

Exercise 1.3: Consider the bigram model described by the following equations:

$$\mathrm{P}(\boldsymbol{W}) = \prod_{i=1}^{n+1} \mathrm{P}(W_i \mid W_{i-2})$$

We assume here that our sentences are padded with two stop symbols. Explain why this model should not do as well as the one presented in this chapter.

Exercise 1.4: Once you have seen how a bigram model extends a unigram model, it should not be too difficult to imagine how a trigram model would work. Write down equations for a soothed trigram language model.

Exercise 1.5: A common method of smoothing is *deleted interpolation.* For a bigram model, rather than Equation 1.11 we would use the following:

$$\widetilde{\Theta}_{w,w'} = \lambda \cdot \frac{n_{w,w'}(\boldsymbol{d})}{n_{w,\circ}(\boldsymbol{d})} + (1-\lambda)\widetilde{\theta}_{w'}$$

1. Show that if $1 \geq \lambda \geq 0$, $\sum_{w'} \widetilde{\Theta}_{w,w'} = 1$ for all w.
2. As with α, we can set λ by maximizing the likelihood of held-out data. As the amount of training data goes up, would we expect λ to increase or decrease?

Exercise 1.6: True or false:

- In a bigram model the best value for alpha is independent of the value for beta (for beta's greater than zero naturally.)
- The best value for beta is independent of the value of alpha.

Explain.

1.6 Programming problem

Problem 1.1: **Distinguishing good from bad English**

Write a program that can, at least to some degree, distinguish between real English and the output of a very bad French-to-English machine translation system. The basic idea is that you create a language model for English, and then run it on both the good and bad versions of sentences. The one that is assigned the higher probability is declared to be the real sentence. This assignment guides you through the steps involved in doing this.

The `/course/cs146/asgn/langmod/data/` directory contains the data we use to train and test the language models. (This data comes from the Canadian Hansard's, which are parliamentary proceedings and appear in both English and French. Here we just use the English.) These files have one sentence per line and have been tokenized, i.e., split into words. We train our language model using `english-senate-0.txt` as the main training data. We also need held-out training data to set the smoothing parameters. `english-senate-2` is the test data. Set the alphas and betas using the development data, `english-senate-1.txt`. Your model assigns a probability to each sentence, including the ◁padding at the end.

1. Create a smoothed unigram language model with the smoothing parameter $\alpha = 1$. Compute the log probability of our language-model test data `english-senate-2.txt`.

2. Now set the unigram smoothing parameter α to optimize the likelihood of the held-out data as described in the text. What values of α do you find? Repeat the evaluation described in the previous step using your new unigram models. The log probability of the language-specific test data should increase.

3. Now try distinguishing good from bad English. In `good-bad.txt` we have pairs of sentences, first the good one, then the bad. Each sentence is on its own line, and each pair is separated from the next with a blank line. Try guessing which is which using the language model. This should not work very well.

4. Now construct smoothed bigram models as described in the text, setting $\beta = 1$, and repeat the evaluations, first on the testing text to see what log probability you get, and then on the good and bad sentences.

5. Set the bigram smoothing parameter β as described in the text, and repeat both evaluations. What values of β maximize the likelihood of the held-out data?

6. Lastly, use the smoothed bigram model to determine good/bad sentences.

1.7 Further Reading

If you turn to the end of this book you will not find a list of references. In research papers such lists have two purposes — they are pointers to further reading, and they help give credit where credit is due. In textbooks, however, this second point is much less important. Furthermore, the topics in a book such as this have been covered in hundreds or thousands, of papers.

Giving the reader guides to further reading is still important, but these days tools such as Google Scholar work very well for this task. They are far from perfect, but it seems to us that they are more reliable than lists of references based upon your authors' memories. Thus we assume you have a tool like Google Scholar, and our goal now is to make sure you have the right key words to find important papers in various subdisciplines of language modeling (LM). Here are a few.

Naturally the best key words are *language modeling.* Also *unigram, bigram, Kneser Ney*

Adaptation in LM is changing the probability of words depending on their context. If in reading an article you encounter the words '*money*' and '*bank*', you are more likely to see the word '*stock*' than '*piano.*' You are also more likely to see '*money*' and '*bank*' a second time, in which case it might be a good idea to store recently seen words in a *cache.* Such adaptation is one kind of *long-range dependency* in LM.

A quite different approach are models based on *neural nets.*

A very good smoothing method that does not require held-out data is *Good-Turing* smoothing.

LM is used in many tasks other than speech recognition. *Information retrieval* is a general term for finding information from documents relating to some topic or question. It is often useful to characterize it in terms of LM. Other tasks that involve LM are *spam detection* and *hand-writing recognition.* However, after speech recognition, the most prominent application is surely *machine translation*, the topic of the next chapter.

Chapter 2

Machine Translation

In the early 1960s the philosopher Bar-Hillel published a famous attack on work in *machine translation* or *MT*. He made two main points: first, MT required a machine to understand the sentence to be translated, and second, we were so far from designing programs that could understand human language that we should put off MT into the indefinite future.

On the first point, Bar-Hillel's argument was conclusive. He simply pointed out that even a very simple example, like translating "The baby picked up a pen", is difficult because '*pen*' has two meanings (or, in NLP terminology, *word senses*): "writing instrument" and "animal/baby pen". If the target language (the language into which we are translating) does not have a word with exactly these two senses, it is necessary to *disambiguate* '*pen*'. (To "disambiguate" is to make unambiguous, in this case to decide which word sense was intended.) Word-sense disambiguation is an ongoing research issue. Furthermore, once we see one example like this it is not hard to think of many more.

Thus, perfect MT does require understanding the text, and thus such a program is still in the future. But perhaps the really difficult MT problems do not occur very often. That seems to be the case: readable MT, MT with only a few debilitating mistakes, now seems within our grasp. Certainly the progress over the last ten-fifteen years has been phenomenal. That this progress has been largely fueled by the new statistical approaches is one of the best selling points for statistical NLP. In this chapter we look at (elementary) MT from the statistical perspective. In doing so we also hope to motivate some of the mathematical techniques used throughout this book, particularly the *expectation maximization algorithm* (*EM* for short).

The key idea behind statistical MT is quite simple. If we want to translate between, say, French and English we first obtain a French-English *parallel corpus* — a text in which each sentence expressed in French is *aligned* with an English sentence meaning the same thing. The first such corpus to be used for MT was the so-called *Canadian Hansard's* — the proceedings of the Canadian parliament, which by Canadian law must be published in both English and French no matter which language was used by the speakers in parliament. (It is called *Hansard's* after the British printer who first published the proceedings of the British parliament.)

Now suppose we want to know which English word (or words) are the translations of the French word '*pain*'. (The most common translation is '*bread*'.) To make this concrete, let us further suppose that our "corpus" consisted of the following French/English pairs:

J 'ai acheté du pain	I bought some bread
J 'ai acheté du beurre	I bought some butter
Nous devons manger le pain blanc	We must eat the white bread

(In French '*j'ai*' is a single word. However when tokenizing French is it useful to split it in two, much as on page 15 we discussed splitting '*doesn't*' into '*does*' and '*n't*'.)

As we go through the corpus looking for French sentences with '*pain*' in them, we check the words in the corresponding English sentence. Here we arranged it so that the word '*bread*' is the only word common between the first and last sentences. In general things will not be so easy, but it is not hard to believe that you will find '*bread*' occurring with great regularity as a possible translation of '*pain*', and that at the very least it would become one of your top candidates.

Statistical MT follows from this simple observation.

2.1 The fundamental theorem of MT

Now let us start at the beginning and develop this idea more precisely.

Notation: Here $\boldsymbol{F}$ is a random variable denoting a French (or foreign) sentence, with $\boldsymbol{f}$ being a possible value, and $\boldsymbol{E}$ is a random variable denoting an English sentence. We use M for the length of $\boldsymbol{F}$, so $\boldsymbol{F} =< F_1, \dots F_m >$. Similarly, L is the length of $\boldsymbol{E} =< E_1 \dots E_l >$. We also typically use j to index over English sentences and k over French.

1. We must eat the white bread
2. We must eat the bread white
3. We eat must the bread white.

Figure 2.1: Some possible translations of “Nous devons manger le pain blanc”

From a probabilistic point of view, MT can be formalized as finding the most probable translation $\boldsymbol{e}$ of a foreign language string $\boldsymbol{f}$, which is

$$\arg\max_{\boldsymbol{e}} \mathrm{P}(\boldsymbol{e} \mid \boldsymbol{f}).$$

As noted in 1.4.5, the *noisy-channel model* is often a good way to approach “argmax”-type problems, and we do this here:

$$\arg\max_{\boldsymbol{e}} \mathrm{P}(\boldsymbol{e} \mid \boldsymbol{f}) = \arg\max_{\boldsymbol{e}} \mathrm{P}(\boldsymbol{e})\mathrm{P}(\boldsymbol{f} \mid \boldsymbol{e}). \tag{2.1}$$

This equation is often called the *fundamental theorem of machine translation*. The first term on the right is a *language model*, as discussed in Chapter 1. The second term is the translation model. It encodes the procedure for turning English strings into French ones.

At first glance, this second term looks counterintuitive. On the left we have the term $P(\boldsymbol{e} \mid \boldsymbol{f})$ and we turn this into a problem requiring that we estimate $P(\boldsymbol{f} \mid \boldsymbol{e})$. Given that it is just as hard to translate from English into French as the other way around, it is not obvious that the noisy-channel model has gained us much.

Nevertheless, this factorization is useful because the translation and language models capture different kinds of dependencies, and (2.1) tells us how these should be combined. To see this, let us consider the third sentence in our fake “corpus” in the introduction: “Nous devons manger le pain blanc”. Consider the several possible English translations in Figure 2.1. The first is the correct translation, the second is the word-by-word translation, and the last permutes the second two words instead of the last two.

In our current state of knowledge, our translation models are very poor at ordering the words in the translation and at picking the best words for a particular context. Thus it would not be surprising if the translation model picked the incorrect literal translation in Figure 2.1 as the best and the other two as equally likely variants. On the other hand, the overlapping windows of even a simple trigram language model should have no problem assigning the first a comparatively high probability and the others dramatically lower ones. Thus by multiplying the two probabilities, our program has a much better chance of getting the correct result.

A second important advantage of the noisy-channel formulation is that, while the translation model $\mathrm{P}(\boldsymbol{F}|\boldsymbol{E})$ needs to be trained from parallel data (which always is in short supply), the language model $\mathrm{P}(\boldsymbol{E})$ can be trained from monolingual data, which is plentiful. Thus it permits us to train our translation system from a wider range of data, and simply adding more training data usually results in more accurate translations (all other factors equal).

In the last chapter we covered language modeling. Here we start with the translation model.

2.2 The IBM Model 1 noisy-channel model

We now turn to a very simple model for $\mathrm{P}(\boldsymbol{F}|\boldsymbol{E})$ known as IBM model 1, so called because it was the first of five ever more complex MT models defined by a group at IBM in the early 1990s.

This model makes a number of simplifying assumptions that more complex models remove. These assumptions mean that model 1 is not a particularly accurate channel model, but it is very simple and easy to train. (We show how to relax one of these assumptions when we explain IBM model 2 in 2.3.)

IBM model 1 assumes that each French word f_k is the translation of exactly one English word in $\boldsymbol{e}$, say e_j. That is, we assume f_k is independent of all the other words in $\boldsymbol{e}$ given the word e_j.

This assumption is less restrictive than it may seem at first. We don't insist that $k = j$, so the French words don't need to be in the same order as the English words they correspond to. We also don't require a one-to-one mapping between English and French words, so each English word can correspond to zero, one, or several French words. We can also give French words some of the same flexibility by assuming that each English sentence $\boldsymbol{e}$ contains an additional invisible *null word* (also called a *spurious word*) '$*$N$*$' that generates words in $\boldsymbol{f}$ that aren't translations of any actual word in $\boldsymbol{e}$. (The $*$N$*$ is assumed to be e_0, the "zeroth" word of the English sentence.)

None of the IBM models give up the "one English word" assumption, but more recent work does. This work, to be discussed at the end of this chapter, assumes that multiword phrases, rather than individual words, are the basic atomic units of translation.

We formalize this word-to-word translation idea in terms of word alignments. A *word alignment* $\boldsymbol{a}$ from a French sentence $\boldsymbol{f}$ of length m to an English sentence $\boldsymbol{e}$ of length l is a vector of length m where e_{a_k} is the English word that translates to f_k. That is, a_k is a position in $\boldsymbol{e}$, i.e., an integer between 0 and l.

Example 2.1: In our toy corpus in the introduction,

J 'ai acheté du pain	I bought some bread
J 'ai acheté du beurre	I bought some butter
Nous devons manger le pain blanc	We must eat the white bread

the alignment for the first sentence pair is

$< 1, 0, 2, 3, 4 >$

Note how the second French word aligns with the zeroth English word, the spurious word. The '*ai*' is needed to express past tense in French, and there is no word corresponding to it in the English. Otherwise the translation is word-for-word.

The correct alignment for the third sentence is:

$< 1, 2, 3, 4, 6, 5 >$

Again the alignment is close to word-for-word, except that it switches the order of between '*pain blanc*' and '*white bread*'.

To give another example, suppose the person who translated this into French had deleted '*devons*', the translation of '*must*'. Then we would have an alignment

$< 1, 3, 4, 6, 5 >$

Notice the alignment vector is now of length five rather than six, and none of the French words align to '*must*'.

We introduce alignments into our probability equations using marginalization. That is:

$$\mathrm{P}(\boldsymbol{f} \mid \boldsymbol{e}) \; = \; \sum_{\boldsymbol{a}} \mathrm{P}(\boldsymbol{f}, \boldsymbol{a} \mid \boldsymbol{e}) \qquad (2.2)$$

As explained in Chapter 1, if you have joint probability $\mathrm{P}(C, D)$ you can sum over all possible values of D to get the probability of C. Here, we obtain the probability $\mathrm{P}(\boldsymbol{f}|\boldsymbol{e})$ by marginalizing over A. We then separate $\boldsymbol{f}$ and $\boldsymbol{a}$ using the chain rule:

$$\mathrm{P}(\boldsymbol{f} \mid \boldsymbol{e}) \;=\; \sum_{\boldsymbol{a}} \mathrm{P}(\boldsymbol{a} \mid \boldsymbol{e})\mathrm{P}(\boldsymbol{f} \mid \boldsymbol{a}, \boldsymbol{e}) \tag{2.3}$$

It is useful to introduce a bit of terminology. Our parallel corpus gives us just the English and French words, E, F. We say that these random variables are *visible variables*, since we see their values in our data. On the other hand, the alignment random variable $\boldsymbol{A}$ is not visible: it is a *hidden variable* because our parallel corpus is aligned only with respect to sentences, not words, so we do not know the correct value of $\boldsymbol{A}$. Hidden variables are useful when they give us a useful way to think about our data. We see in ensuing sections how $\boldsymbol{A}$ does this.

Example 2.2: Some examples of $\mathrm{P}(\boldsymbol{a} \mid \boldsymbol{e})$ are in order. So suppose we are aligning an $\boldsymbol{e}/\boldsymbol{f}$ pair in which both are of length six. English and French have very similar word order so, all else being equal, the most probable alignment might be $< 1, 2, 3, 4, 5, 6 >$. In our "the white bread" example we came close to this except the order of the last two words was exchanged, $< 1, 2, 3, 4, 6, 5 >$. This should have a probability lower than 1 to 6 in order, but still relatively high.

At the other extreme, it is easy to make up alignments that should have very low probability when we are aligning two six-word sentences. Consider $< 6, 5, 4, 3, 2, 1 >$ — the French sentence is ordered in the reverse of the English. An even sillier alignment would be $< 1, 1, 1, 1, 1, 1 >$. This says that the entire French translation is based upon the first word of the English, and the rest of the English was completely ignored. There is nothing in the definition of an alignment function that forbids this.

Even though many silly alignments should be assigned *very* low probabilities, IBM model 1 assumes that all the ℓ^m possible word alignments of $\boldsymbol{e}$ and $\boldsymbol{f}$ are equally likely! Thus removing this assumption is one of the major improvements of more sophisticated noisy-channel models for machine translation. In particular, this is done in *IBM model 2* discussed in section 2.3.

Nevertheless, IBM model 1 does remarkably well, so let's see how the equal-probability assumption plays out. We now plug the IBM model 1 assumption into (2.3) and get our "model 1 equation".

First. note that once we condition on a, we can break apart the probability of $\boldsymbol{f}$ into the probabilities for each word in f, f_k. That is, we can now say

$$\mathrm{P}(\boldsymbol{f} \mid \boldsymbol{e}, \boldsymbol{a}) \;=\; \prod_{k=1}^{m} \mathrm{P}(f_k \mid e_{a(k)}).$$

Substituting this into equation 2.3 gives us:

$$\begin{aligned} \mathrm{P}(\boldsymbol{f} \mid \boldsymbol{e}) &= \sum_{\boldsymbol{a}} \mathrm{P}(\boldsymbol{a} \mid \boldsymbol{e}) \prod_{k=1}^{m} \mathrm{P}(f_k \mid e_{a(k)}) \\ &= \mathrm{P}(m \mid \ell)\ell^{-m} \sum_{\boldsymbol{a}} \prod_{k=1}^{m} \mathrm{P}(f_k \mid e_{a(k)}). \end{aligned} \tag{2.4}$$

In (2.4) we replaced $\mathrm{P}(\boldsymbol{a} \mid \boldsymbol{e})$ (the probability for alignment a) by $\mathrm{P}(m \mid \ell)\ell^{-m}$. First. note that there are ℓ^m possible alignments, and they all have the same probability. This probability is ℓ^{-m}. However, initially we are only "given" (condition on) $\boldsymbol{e}$ from which we can get l, its length. Thus we must "guess" (assign a probability to) m, the length of $\boldsymbol{f}$. Hence the term $\mathrm{P}(m \mid l)$.

Example 2.3: Suppose $\boldsymbol{f}$ = '*Pas fumer*' and $\boldsymbol{e}$ = '*No smoking*'. Here $l = m = 2$, so the probability of any one alignment is $\frac{1}{4}\mathrm{P}(M = 2 \mid L = 2)$. Next consider the length term. A plausible value for $\mathrm{P}(2 \mid 2)$ would be 0.4. That is, a substantial fraction of French two-word sentences get translated into two-word English ones. So the product of the terms outside the summation in 2.4 would be 0.1.

Equation 2.4 describes a generative model in that, given an English sentence $\boldsymbol{e}$, it tells us how we can generate a French sentence $\boldsymbol{f}$ along with its associated probability. As you may remember, in Chapter 1 we talked about a generative "story" as a good way to explain equations that describe generative models. Here is the associated generative story .

Given a sentence $\boldsymbol{e}$ of length ℓ, we can generate a French sentence $\boldsymbol{f}$ of length m using the following steps and their associated probabilities.

1. Choose the length m of the French sentence $\boldsymbol{f}$. This gives us the $\mathrm{P}(m \mid l)$ in Equation 2.4.

2. Choose a word alignment $\boldsymbol{a}$ at random from the ℓ^m possible alignments. This give us the ℓ^{-m} portion.

3. Generate each French word $F_k, k = 1, \ldots, m$ from the English word e_{a_k} it is aligned to with probability $\mathrm{P}(f_k \mid e_{a(k)})$. (This gives us $\prod_{k=1}^{m} \mathrm{P}(f_k \mid e_{a(k)})$.)

This gives one way to generate $\boldsymbol{f}$ from $\boldsymbol{e}$ — the way corresponding to the chosen alignment $\boldsymbol{a}$. We can generate $\boldsymbol{f}$ in lots of ways, each one corresponding to a different alignment. In Equation 2.4 we sum over all of these to get the total probability of $\mathrm{P}(\boldsymbol{f}|\boldsymbol{e})$.

From this story we see immediately that the IBM model 1 channel model has two sets of parameters. One set is the estimate of the conditional probability $\mathrm{P}(m|\ell)$ of generating a French sentence of length m from an English sentence of length ℓ. The most straightforward way to get this is to use the maximum likelihood estimator. First go through the parallel corpus and count how often an English sentence of length ℓ is paired with a French sentence of length m. In our usual way, we call this $n_{\ell,m}$. Then to get the maximum likelihood estimation we divide this by how often an English length ℓ appears in the corpus, $n_{\ell,\circ}$. We call these parameters $\eta_{\ell,m} = n_{\ell,m}/n_{\ell,\circ}$.

The second set of parameters is the conditional probabilities $\mathrm{P}(f|e)$ of generating the French word f given that it is aligned to the English word e, henceforth referred to as $\tau_{e,f}$.

After substituting the model parameter estimations into our IBM model 1 equation, we get this:

$$\mathrm{P}(\boldsymbol{f} \mid \boldsymbol{e}) = \eta_{\ell,m} \ell^{-m} \sum_{\boldsymbol{a}} \prod_{k=1}^{m} \tau_{e_{a(k)},f_k}. \tag{2.5}$$

All the terms of this equation are either known or easily estimated except for the τ's. We turn to this next.

Example 2.4: If $\boldsymbol{f}$ is '*J'ai acheté du pain*' then $\mathrm{P}(\boldsymbol{f} \mid \boldsymbol{e})$ will be the same if $\boldsymbol{e}$ is '*I bought some bread*' or '*Bought I some bread*'. Basically, this is a consequence of the assumption that all alignments are equally likely. More immediately, we can see this from Equation 2.5. Both English versions have $\ell = 4$ so $\eta_{\ell,m}\ell^{-m}$ is the same for both translations. Then in the first case $\boldsymbol{a}_1 =< 1, 0, 2, 3, 4 >$ while $\boldsymbol{a}_2 =< 2, 0, 1, 3, 4 >$. With these alignments the product of $\tau_{e_{a(k)},f_k}$ will be the same, differing only in the order of multiplication.

On the other hand, consider the two possible translations '*I bought some bread*' and '*I bought bread*'. (In most cases the second of these would be the preferred translation because the '*du*' in the French version is grammatically obligatory. It

J 'ai acheté du pain	$< 1, 0, 2, 3, 4 >$	I bought some bread
J 'ai acheté du beurre	$< 1, 0, 2, 3, 4 >$	I bought some butter
Nous devons manger le pain blanc	$< 1, 2, 3, 4, 6, 5 >$	We must eat the white bread

Figure 2.2: A word-aligned version of our toy corpus

does not have any meaning of its own and thus need not be translated.) Now the translation model probabilities will differ. First, for comparable sentences French tends to use more words than English by about 10%, so while $\eta_{4,4}$ will probably be larger than $\eta_{3,4}$ the difference will be small. The next term, however, ℓ^{-m}, will be larger for the shorter English sentence (4^{-4} vs. 3^{-4}). The difference in word probabilities will be due to the difference in alignments of '*du*' in the French to '*some*' in the first English version and ∗N∗ in the second. Thus the first has the term $\tau_{some,du}$ and the second $\tau_{*\mathrm{N}*,du}$. Both these translation pairs are reasonably likely, but in our model $\tau_{*\mathrm{N}*,du}$ will typically be smaller because the null word has quite a few translations but '*some*' has fewer, and thus each one will be more probable. Thus we would typically expect the translation model to slightly prefer the more wordy English. On the other hand, a good English language model would distinctly prefer the shorter English, and this would most likely tip the balance in favor of '*I bought bread*'.

2.2.1 Estimating IBM model 1 parameters with EM

We attack the τ problem by noting that it would be easy to estimate $\boldsymbol{\tau}$ from a *word-aligned parallel corpus*, i.e., a corpus that specifies the word alignment $\boldsymbol{a}$ for the English-French sentences. Figure 2.2 shows such an alignment for our toy corpus.

Note how the alignment allows us to read off how often each French word is paired with each English, and from that it is easy to get a maximum likelihood estimate of all the $\tau_{e,f}$. The maximum likelihood estimator for $\tau_{e,f}$ is just the number of times e aligns to f divided by the number of times e aligns with anything. Or, more formally:

$$n_{e,f}(\boldsymbol{a}) \;=\; \sum_{k:f_k=f} [\![e_{a_k} = e]\!] \,. \tag{2.6}$$

where $[\![condition]\!]$ is the indicator function for *condition*, i.e., it is 1 if *condition* is true and 0 otherwise. Then the maximum likelihood estimator $\boldsymbol{\tau}$ is just the relative frequency, i.e.:

$$\hat{\tau}_{e,f} = \frac{n_{e,f}(\boldsymbol{a})}{n_{e,\circ}(\boldsymbol{a})} \tag{2.7}$$

where $n_{e,\circ} = \sum_f n_{e,f}(\boldsymbol{a})$ is the number of times that e is aligned to any French word in $\boldsymbol{a}$ (this is *not* necessarily the same as the number of times e appears in the training corpus). These two equations combine to say, in effect: go through the corpus counting how often e aligns with f and then take the maximum likelihood estimate to get the corresponding probability.

Example 2.5: In Figure 2.2 all the τ's are 1, because each English word is aligned to a unique French word. Suppose that we made a mistake and aligned '*bread*' with '*must*' in the third sentence. Then $\tau_{bread,pain}$ would be $\frac{1}{2}$ because bread is aligned with pain once, and bread is paired with anything two times.

Now, let us suppose that the folks who created our word-aligned corpus were not always confident that they could align the words correctly and that when they were unsure they labeled alignments with probabilities: e.g., they think e_2 aligns with f_2, but it might be e_1, so they assign a probability of 0.9 to the first and 0.1 to the second. What do we do when the alignments are not yes or no, but more or less confident?

There are several possibilities. One is simply to ignore any alignment with confidence less than some threshold, say 0.8. If we had more word-aligned data than we could use, this would be a reasonable thing to do. However, this is never the case, so if there is useful information in our data we should try to extract it. And there is a lot of useful information in alignments even when they are quite uncertain. Suppose the aligner knows that in some ten-word sentence f_1 aligns with either e_1 or e_2, but is completely unsure which, so they both get probability 0.5. Note that without this information we were much more uncertain. In IBM model 1, since all alignments are equally likely, those two alignments would have had probability 0.1 each (since there were ten words in the sentence). So even a 50-50 split is useful.

We are now poised to introduce the first of two key insights of the so-called *Expectation Maximization algorithm, EM.* In a case like the above we split the alignment counts $N_{e,f}$ according to their probabilities. In the 50-50 split case both n_{e_1,f_1} and n_{e_2,f_1} get a 0.5 "count". When we split counts like this we call them *fractional counts* or *partial counts.*

English	French	Sentences			$\tau^2_{e,f}$
		1	2	3	
bought	pain	1/2			1/4
bought	acheté	1/2	1/2		1/2
bread	pain	1/2		1/2	1/2
bread	acheté	1/2			1/4
bought	beurre		1/2		1/4
butter	acheté		1/2		1/2
butter	beurre		1/2		1/2
eat	pain			1/2	1/2
eat	manger			1/2	1/2
bread	manger			1/2	1/4

Figure 2.3: Partial counts for English-French word pairs in three sentences

Example 2.6: Suppose we have some annotators align our three-sentence parallel corpus. For some reason they align '*acheté*' with equal probability to both '*bought*' and '*bread*' in the first sentence. Similarly, '*pain*' is also aligned with equal probability to these two. The same sort of confusion occurs in the second sentence, except now it is '*acheté/beurre*' and '*bought/butter*'. In the third sentences it is '*manger/pain*' and '*eat/bread*'. Figure 2.3 shows the partial counts we get for these word pairs. For example, the first line starting with '*bought*' indicates that it is aligned with '*pain*' 0.5 times in sentence one, and not at all in the other two sentences. Looking down the columns, we see that each sentence is responsible for a total of two counts. (Because we are assuming all the other words are unambiguous we are ignoring them. In effect assuming that the sentences only include the ambiguous words.)

The last column gives the second iteration $\tau^2_{e,f}$ that is computed from these partial counts. (We occasionally use superscripts on parameter values to indicate the iteration in which they apply. In the first iteration all of the τs were equal and the new τ^2's are used in the second iteration.) For example,

$$\begin{aligned}\tau^2_{bought,pain} &= \frac{n_{bought,pain}}{n_{bought,\circ}} \\ &= \frac{1/2}{1/2+1/2+1/2+1/2} = 1/4.\end{aligned}$$

Again, note that by this point EM is preferring '*bought/acheté*' over any of the other translations of '*bought*', and similarly for '*bread*' and '*pain*'. On the other hand, the translations for '*butter*' and '*eat*' have not been clarified. This is because each of these words appears only in a single sentence, so there is no way to get disambiguation from other sentences.

In the EM algorithm we take this to an extreme and pretend that initially our annotaters gave each $\boldsymbol{f}$-$\boldsymbol{e}$ alignment an equal probability. So for a ten-word English sentence each fractional count would be 0.1. (Actually, it need not necessarily be uniform, but when you don't know anything, uniform is most often the best thing to try.) Then after going through the corpus summing up the fractional counts, we set the τ parameters to the maximum likelihood estimates from the counts, just as if they were "real".

Now we introduce the second key idea. The parameters we have just obtained should be much better than our original assumption that everything was equally likely, so we repeat the process, but now using the new probabilities. And so on. That is to say, EM is an iterative algorithm in which we start with a very bad (e.g., uniform) distribution of our τs and at each iteration replace them with the ML estimate from the previous iteration. In Section 2.2.3, after seeing that EM actually works, we give an analysis of what EM is doing from a mathematical point of view.

To make this complete we just need the equation for computing fractional counts when we have just probabilistic information about who is aligned with whom. For the moment we give it without mathematical justification:

$$n_{e_j,f_k} += \frac{\tau_{e_j,f_k}}{p_k} \tag{2.8}$$

where

$$p_k = \sum_j \tau_{e_j,f_k} \tag{2.9}$$

That is, within each sentence and for each French word f_k, we add to our running total of n_{e_j,f_k} the term on the right-hand side of Equation 2.8. (We implicitly assume that j runs over only the English words in the sentence aligned with $\boldsymbol{f}$.) This term is our estimate for the probability that f_k is the translation of our English word e_j divided by the total probability that it translates any of the English words in the corresponding sentence (p_k).

Actually, what we are computing here is the expected number of times our generative model aligns f_k with e_j given our data, that is,

$$E[n_{e,f} \mid \boldsymbol{e}, \boldsymbol{f}].$$

We come back to this point in Section 2.2.3 where we derive Equation 2.8 from first principles.

English	French	Sentences			$\tau^2_{e,f}$
		1	2	3	
bought	pain	1/2			2/11
bought	acheté	1/2	1/2		7/11
bread	pain	1/2		1/2	7/11
bread	acheté	1/2			2/11
bought	beurre		1/3		2/11
butter	acheté		1/2		3/7
butter	beurre		2/3		4/7
eat	pain			1/2	3/7
eat	manger			2/3	4/7
bread	manger			1/3	2/11

Figure 2.4: Partial counts for English-French word pairs on the second iteration

Example 2.7: The previous example (see Figure 2.3) followed the EM algorithm though its first iteration, culminating in a new $\boldsymbol{\tau}$. Here we go though the second iteration (see Figure 2.4). As before, we go though the sentences computing $n_{e,f}$ for each word pair. Again, to take a single example, consider $N_{bought,acheté}$ for the first sentence.

$$\begin{aligned} p_{acheté} &= \sum_j \tau_{j,acheté} \\ &= \tau_{bread,acheté} + \tau_{bought,acheté} \\ &= 1/4 + 1/2 \\ &= 3/4 \end{aligned}$$

$$\begin{aligned} n_{bought,achete} &= \frac{\tau_{bought,acheté}}{p_{acheté}} \\ &= \frac{1/2}{3/4} \\ &= 2/3 \end{aligned}$$

Thus we arrive at the EM algorithm for estimating the τ parameters of IBM model 1 shown in Figure 2.5 To keep things simple we have written the algorithm as if there were only one French/English sentence pair, and show how we compute the expectations for that single sentence. In reality we also iterate on all sentence pairs, to sum the expectations.

```
1. Pick positive initial values for $\tau_{e,f}$ for all English words $e$ and all French
   words $f$ (equal is best).

2. For $i = 1, 2, \ldots$ until convergence (see below) do:

   (a) E-step:
       Set $n_{e,f} = 0$ for all English words $e$ and French words $f$.
       For each French word position $k = 1, \ldots, m$ do:
          i. Set $p_k = \sum_{j=0}^{l} \tau_{e_j,f_k}$, where $j$ are the positions of the English
             words in the same sentence pair as $f_k$.
         ii. For each $0 \leq j \leq l$, increment $n_{e_j,f_k}$ += $\tau_{e_j,f_k}/p_k$
       ($n_{e,f}$ now contains the expected number of times $e$ aligns with $f$)
   (b) M-step:
       Set $\tau_{e,f} = n_{e,f}/n_{e,\circ}$, where $n_{e,\circ} = \sum_f n_{e,f}$.
```

Figure 2.5: EM algorithm for IBM model 1 inference

A few points about this algorithm. First, we can now understand why this is called the *Expectation Maximization* algorithm. Each iteration has two steps. We use the previous estimate $\boldsymbol{\tau}^{(i-1)}$ of the parameters to compute the *expected value* of the statistics $n_{e,f}$. Then we set $\boldsymbol{\tau}^{(i)}$ to the maximum likelihood estimate using those expected values.

Second, the algorithm says to iterate until "convergence." The EM algorithm finds values of the hidden parameters that correspond to a local maximum of the likelihood.

$$L_{\boldsymbol{d}}(\boldsymbol{\Phi}) \;\propto\; \prod_{k=1}^{m} p_k. \tag{2.10}$$

That is the likelyhood of the data is a constant times the product of the p_ks where the constant is dependent only on the English strings. After a few iterations neither the likelihood nor the $\boldsymbol{\tau}$s change much from one iteration to the next. Since we need to compute the p_k's in the inner loop anyway, keeping track of the likelihood at each iteration is a good way to measure this "convergence". We set a threshold, say 1%, and when the likelihood changes less than this threshold from one iteration to the next we stop. (As you might expect, multiplying all of these probabilities together produces a *very* small number. Thus it is better to compute the log of the likelihood by adding the $\log p_k$'s.)

Also, while in general EM is guaranteed only to find a local maximum, in our present application the likelihood of Equation 2.10 is unimodal — it only has one maximum. Therefore in this case we find a global maximum.

Example 2.8: Figures 2.3 and 2.4 followed EM through two iterations on a simple example, where all the alignments were certain except those for '*pain*', '*acheté*', '*buerre*',and '*manger*'. Let us compute the likelihood of our data with the τ we computed after the first and second iterations to check that the likelihood of our training corpus is indeed increasing. Equation 2.10 tells us to take the product of the p_k for the position k of every French word in the corpus. First note that $p_k = 1$ for all French words k that we assumed were correctly and uniquely aligned, e.g. '*j*', '*ai*' '*le*', etc. So these can be ignored. All we need to do is look at the p_ks for all occurrences of the four French words that were originally labeled ambiguously. However, rather than do them all, let's just consider the $p_{acheté}$ in the first sentence. First for the $\boldsymbol{\tau}$ after the first iteration:

$$\begin{aligned} p_{acheté} &= \tau_{acheté,bought} + \tau_{acheté,bread} \\ &= 1/2 + 1/4 \\ &= 3/4. \end{aligned}$$

For the $\boldsymbol{\tau}$ we get after the second iteration

$$\begin{aligned} p_{acheté} &= 2/3 + 1/3 \\ &= 1. \end{aligned}$$

So this p_k increases. As it turns out, they all either increase or remain the same.

2.2.2 An extended example

Figure 2.6 shows the estimated IBM model 1 parameters $\tau_{e,pain}$ for several English words e, trained on some sections of the Canadian Hansard's. We show these values after several different iterations of EM.

In the first iteration there is good news and bad news. The good news is that $\tau_{bread,pain}$ comes out with a relatively high value. For example, '*bread*' at 0.04 is considerably higher than '*spirit*' at 0.001. However, the program does not do too well distinguishing '*bread*' from related words that appear along with bread, such as '*baked*', not to mention some seemingly random words like '*drudgery*', which at 0.048 seems way too high. And, of course, the

English word	Iteration 1	Iteration 2	Iteration 19	Iteration 20
bread	0.042	0.138	0.3712	0.3710
drudgery	0.048	0.055	0.0	0.0
enslaved	0.048	0.055	0.0	0.0
loaf	0.038	0.100	0.17561	0.17571
spirit	0.001	0.0	0.0	0.0
mouths	0.017	0.055	0.13292	0.13298

Figure 2.6: Probabilities for English words translating as '*pain*'

probability for '*bread*' in Figure 2.6 is much too low. The actual probability of translating '*bread*' as '*pain*' is close to 1, say 0.9.

We now move on to a second iteration. Now the probability of '*pain*' given '*bread*' has gone up by better than a factor of three (from 0.042 to 0.138). Spirit fell off the map, but '*drudgery*' and '*enslaved*' went up, though only slightly.

Having done this twice, there is nothing stopping us from repeating this process ad infinitum, except after a while there is not much change from one iteration to the next. After twenty iterations of EM '*pain*' is the translation of '*bread*' 37% of the time. This is very good compared to the first iteration, but it is still too low. '*Drudgery*' and '*enslaved*' have properly gone to zero. '*Loaf*' can be translated as '*pain*', but the probability of 15% is probably too high. This is because '*loaf of bread*' would usually be translated as just '*pain.*' '*Mouths*' is definitely an artifact of some funny sentences somewhere.

We added an extra level of precision to the numbers for iterations 19 and 20 to show that the changes from one iteration are becoming quite small.

2.2.3 The mathematics of IBM 1 EM

The EM algorithm follows from a theorem of statistics which goes roughly as follows. Suppose you have a generative model that depends on hidden parameters (in our case, the τs). Initially make nonzero guesses about the values of these parameters. Imagine actually generating all of the data according to these parameters. In so doing, each of the τs will be used an estimated number of times E_τ. After going through our data we reset the τ's to their maximum likelihood value according to these estimates. This can be iterated. The theorem states that as you iterate the τs will monotonically approach a set that gives a local maximum for the likelihood of the data. (It is also possible to get stuck in a saddle-point, but we ignore that here.)

A quick comparison of the above description with the EM algorithm as shown on Page 52 should convince you that it is correct, *provided that what we there labeled the e-step indeed computes expectation for the τ's.* That is what we prove in this section.

We first start with a clear formulation of what the expectation is that we are computing at each iteration i:

$$n_{e,f}^{(i)} \;=\; E_{\boldsymbol{\tau}^{(i-1)}}[n_{e,f} \mid \boldsymbol{e}, \boldsymbol{f}] \;=\; \sum_{\boldsymbol{a}} n_{e,f}(\boldsymbol{a})\, \mathrm{P}_{\boldsymbol{\tau}^{(i-1)}}(\boldsymbol{a} \mid \boldsymbol{e}, \boldsymbol{f}). \tag{2.11}$$

The first equality makes it clear that at iteration i we compute the expectation according to the $\boldsymbol{\tau}$ we obtained at the previous iteration. (For the first iteration we just use the initial, all equal τs.) The second equality comes from the definition of expectation. Here $n_{e,f}(\boldsymbol{a})$ is the number of times f is aligned with e in the alignment $\boldsymbol{a}$ defined in Equation 2.6 and repeated here:

$$n_{e,f}(\boldsymbol{a}) \;=\; \sum_{k:f_k=f} [\![e_{a_k} = e]\!]\,.$$

So the right hand side says what we would expect (excuse the pun), that the expectation is the number of times we align e and f in an alignment times the probability of that alignment.

Unfortunately, the computation is intractable as expressed above because it sums over all alignments. This number grows exponentially in the length of $\boldsymbol{a}$. We now show how the non-exponential computation we gave in our EM algorithm is equivalent.

We do this in two steps. First we show how the expectation can be reformulated as follows:

$$E_{\boldsymbol{\tau}}[n_{e,f} \mid \boldsymbol{e}, \boldsymbol{f}] \;=\; \sum_{k:f_k=f} \sum_{j:e_j=e} \mathrm{P}_{\boldsymbol{\tau}}(A_k = j \mid \boldsymbol{e}, \boldsymbol{f}). \tag{2.12}$$

Note that in this equation there is no iteration over all $\boldsymbol{a}$. Instead, it tells us to march through each French word position k computing the sum on the right. This is what we do when the EM algorithm says "For each French word position k ...". Also, note that it is intuitively quite reasonable: it says that the expected number of times the English word type e is aligned with French word type f is the sum of the probabilities of any English token e_j of type e being aligned with any French token f_k of type f.

Then we show that for IBM model 1:

$$\mathrm{P}_{\boldsymbol{\tau}}(A_k = j \mid \boldsymbol{e}, \boldsymbol{f}) \quad = \quad \frac{\tau_{e_j, f_k}}{\sum_{j'=0}^{l} \tau_{e_{j'}, f_k}}. \qquad (2.13)$$

The j' in the denominator ranges over the word positions in the corresponding English sentence.

In many applications EM is quite sensitive to its initial parameter values. But, as remarked above, the likelihood function (2.10) for IBM model 1 is unimodal, which means that in this particular case it is not that important how $\boldsymbol{\tau}$ is initialized. You can initialize it with random positive values, but setting them all equal is typically best.

So now we turn our attention to the two equations (2.12) and (2.13) used in the derivation of the EM algorithm above. Combining (2.6) and (2.11) and reordering the summation, we have:

$$\begin{aligned} E[n_{e,f}|\boldsymbol{e},\boldsymbol{f}] &= \sum_{\boldsymbol{a}} \sum_{k:f_k=f} [\![e_{a_k} = e]\!] \; \mathrm{P}(\boldsymbol{a} \mid \boldsymbol{e}, \boldsymbol{f}) \\ &= \sum_{k:f_k=f} \sum_{\boldsymbol{a}} [\![e_{a_k} = e]\!] \; \mathrm{P}(a_k \mid \boldsymbol{e}, \boldsymbol{f}) \; \mathrm{P}(\boldsymbol{a}_{-k} \mid \boldsymbol{e}, \boldsymbol{f}, a_k). \end{aligned}$$

where $\boldsymbol{a}_{-k}$ is the vector of word alignments for all words *except* f_k (i.e., $\boldsymbol{a}_{-k}$ is $\boldsymbol{a}$ with a_k removed). Splitting the sum over $\boldsymbol{a}$ into a sum over a_k and $\boldsymbol{a}_{-k}$ and rearranging the terms, we have:

$$\begin{aligned} E[n_{e,f}|\boldsymbol{e},\boldsymbol{f}] &= \sum_{k:f_k=f} \sum_{a_k} \sum_{\boldsymbol{a}_{-k}} [\![e_{a_k} = e]\!] \; \mathrm{P}(a_k \mid \boldsymbol{e}, \boldsymbol{f}) \; \mathrm{P}(\boldsymbol{a}_{-k} \mid \boldsymbol{e}, \boldsymbol{f}, a_k) \\ &= \sum_{k:f_k=f} \sum_{a_k} \Big([\![e_{a_k} = e]\!] \; \mathrm{P}(a_k \mid \boldsymbol{e}, \boldsymbol{f}) \; \sum_{\boldsymbol{a}_{-k}} \mathrm{P}(\boldsymbol{a}_{-k} \mid \boldsymbol{e}, \boldsymbol{f}, a_k) \Big) \\ &= \sum_{k:f_k=f} \sum_{a_k} [\![e_{a_k} = e]\!] \; \mathrm{P}(a_k \mid \boldsymbol{e}, \boldsymbol{f}) \\ &= \sum_{k:f_k=f} \sum_{j:e_j=e} \mathrm{P}(A_k{=}j \mid \boldsymbol{e}, \boldsymbol{f}). \end{aligned}$$

We get the third line here because $\sum_{\boldsymbol{a}_{-k}} \mathrm{P}(\boldsymbol{a}_{-k} \mid \boldsymbol{e}, \boldsymbol{f}, a_k) = 1$. All the terms we are conditioning on are constants as far as the summation is concerned and the sum of any probability distribution over all possible outcomes is 1. Next, notice that nothing in this version depends on the entire $\boldsymbol{a}$, just a_k. From this the final line follows, and this is equation 2.12.

When we turn to IBM model 2 we reuse this result, so we need to point out several aspects of the equation that will become important. First, the words that we are translating appear only in the summation. What we actually compute is the probability of the alignment. The words tell us then which $n_{e,f}$ bin gets incremented.

Second, we note that only one IBM model 1 assumption was used to get here: that each French word is generated by zero or one English words. It is this assumption that allows us to characterize all the important hidden information in the alignment vector random variable $\boldsymbol{A}$. Or, to put it another way, alignments would not make much sense if several English words were allowed to combine to form a French one. On the other hand, we did not use the second major assumption, that all alignments have equal probability. Thus Equation 2.12's reuse is allowed when the latter assumption is relaxed.

Turning to Equation 2.13, we first use the postulate that IBM model 1 alignments of French word f_k do not depend on the alignments of any other French words, so:

$$\mathrm{P}(A_k{=}j \mid \boldsymbol{e}, \boldsymbol{f}) = \mathrm{P}(A_k{=}j \mid \boldsymbol{e}, f_k).$$

By Bayes' rule we have:

$$\begin{aligned}\mathrm{P}(A_k{=}j \mid \boldsymbol{e}, f_k) &= \frac{\mathrm{P}(f_k \mid A_k{=}j, \boldsymbol{e})\,\mathrm{P}(A_k{=}j \mid \boldsymbol{e})}{\mathrm{P}(f_k \mid \boldsymbol{e})} &(2.14)\\ &= \frac{\mathrm{P}(f_k \mid A_k{=}j, e_j)\,\mathrm{P}(A_k{=}j \mid \boldsymbol{e})}{\sum_{j'}\mathrm{P}(f_k \mid A_k{=}j', e_{j'})\,\mathrm{P}(A_k{=}j' \mid \boldsymbol{e})}. &(2.15)\end{aligned}$$

In IBM model 1 all alignments are equally likely, i.e., $\mathrm{P}(A_k{=}j|\boldsymbol{e}) = \mathrm{P}(A_k{=}j'|\boldsymbol{e})$, so:

$$\mathrm{P}(A_k{=}j \mid \boldsymbol{e}, \boldsymbol{f}) = \frac{\mathrm{P}(f_k \mid A_k{=}j, e_j)}{\sum_{j'}\mathrm{P}(f_k \mid A_k{=}j', e_{j'})} = \frac{\tau_{e_j,f_k}}{\sum_{j'}\tau_{e_{j'},f_k}}. \qquad (2.16)$$

which is Equation 2.8. Note that we have just used the all-alignments-are-equiprobable assumption.

2.3 IBM model 2

So far we have assumed that the probabilities of all alignments $\boldsymbol{a}$ are the same. As this is not a very good assumption, IBM model 2 replaces it with the assumption that a word at position k in the source language (French) will be moved to position j in the target with probability $\mathrm{P}(A_i = j|k, l, m)$, where as before l and m are the lengths of the English and French sentences respectively. Once we add these probabilities to our equations, we let EM estimate them, just like the word translation probabilities.

To see precisely how this works, let us go back and remember where the offending assumption (all alignments are equally likely) was introduced into our equations:

$$\begin{aligned} P(\boldsymbol{f} \mid \boldsymbol{e}) &= \sum_{\boldsymbol{a}} P(\boldsymbol{f}, \boldsymbol{a} \mid \boldsymbol{e}) \\ &= \sum_{\boldsymbol{a}} P(\boldsymbol{a} \mid \boldsymbol{e})\, P(\boldsymbol{f} \mid \boldsymbol{a}, \boldsymbol{e}) \\ &= \mathrm{P}(m \mid l) l^{-m} \sum_{\boldsymbol{a}} P(\boldsymbol{f} \mid \boldsymbol{a}, \boldsymbol{e}). \end{aligned} \tag{2.17}$$

It is in the last equation that we replaced $P(\boldsymbol{a} \mid \boldsymbol{e})$ with $\mathrm{P}(m \mid l)l^{-m}$. We back out of this assumption by reverting to the previous equation, so we now need to compute $\mathrm{P}(\boldsymbol{a} \mid \boldsymbol{e})$. We get our formula for this as follows:

$$\begin{aligned} P(\boldsymbol{a} \mid \boldsymbol{e}) &= \mathrm{P}(m \mid l)\; P(\boldsymbol{a} \mid l, m) \\ &= \mathrm{P}(m \mid l) \prod_{i=1}^{m} \mathrm{P}(A_i = j \mid i, l, m). \end{aligned}$$

The first line assumes that the only thing the alignment probability takes from the English sentence is its length. The second line assumes that each alignment probability is independent of the others, and thus the probability of the total alignment is just the product of the individual alignment probabilities.

So IBM model 2 needs a new set of parameters $\delta_{j,k,l,m}$ that are estimates of $P(A_k = j \mid k, l, m)$, the probability that the French word in position k is aligned with the English word in position j given the position of the French word (k) and the lengths of the two sentences (l, m). These parameters are often called the *distortion probabilities*, whence the use of the Greek letter δ:

$$P(\boldsymbol{a} \mid \boldsymbol{e}) = \eta_{l,m} \prod_{k=1}^{m} \delta_{j,k,l,m}. \tag{2.18}$$

Example 2.9: Suppose we have the two sentences '*The white bread*' and '*Le pain blanc*' with the alignment $a =< 1, 3, 2 >$. Then

$$P(\boldsymbol{a} \mid \boldsymbol{e}) \;=\; \eta_{3,3} \cdot \delta_{1,1,3,3} \cdot \delta_{2,3,3,3} \cdot \delta_{3,2,3,3}. \tag{2.19}$$

Example 2.10: Suppose we have a French sentence of length 10 and we are looking at the second word. Our English sentence is of length 9. The distortion parameter for the case that $A_2 = 2$ is $\delta_{2,2,9,10}$. A reasonable value of this parameter would be around 0.2. We would expect this to be much larger than say, $\delta_{7,2,10,9}$, which might be, say, 0.01.

Example 2.11: Some combinations of δ parameters cannot correspond to any real situation and thus have probability zero, e.g. $\delta_{20,2,10,9}$. The second French word cannot be aligned to the twentieth English one when the English sentence has only nine words. We condition on the sentence lengths so as not to assign probability mass to such parameters.

In practice it is often convenient to index distortion probabilities on both word positions and length of the French sentence, but not the English When we actually use them to translate we will know the former but not the latter, at least not until the translation is completed.

Example 2.12: The assumption that the alignment probabilities are independent of each other is not a very good one. To give a specific example, both French and English allow prepositional phrases like '*last Tuesday*' to migrate to the beginning of a sentence. In English we have:

I will write the letter next week.
Next week I will write the letter.

Think about the "alignment" of these two sentences (i.e., think of the first one as the French sentence). The correct alignment is $< 3, 4, 5, 6, 1, 2 >$. In general, the probability that $A_6 = 2$, as here, is low. However, if we knew that $A_5 = 1$, then it is much more likely. So distortion probabilities are not really independent of each other. In section 3.7 we show a clever way to handle these dependencies. For now we simply note that, although assuming that alignment probabilities are mutually independent is not great, it is certainly better than assuming they are all equal.

If we now substitute our alignment probability from Equation 2.18 into Equation 2.18, we get:

$$\begin{aligned} P(\boldsymbol{f} \mid \boldsymbol{e}) &= \sum_{\boldsymbol{a}} P(\boldsymbol{a} \mid \boldsymbol{e})\, P(\boldsymbol{f} \mid \boldsymbol{a}, \boldsymbol{e}) \\ &= \eta_{l,m} \sum_{\boldsymbol{a}} \prod_{k=1}^{m} \delta_{a_k,k,l,m}\, \tau_{e_{a_k},f_i}. \end{aligned} \tag{2.20}$$

Note how similar this new equation is to our IBM model 1 equation, Equation 2.5. Before we multiplied all of the translation probabilities together. Now we do the same thing, but multiply in one alignment probability for each translation probability. Furthermore, this similarity carries over to our fractional count equation. The new equation replacing Equation 2.9 makes the identical transformation:

$$n_{e_{a_k},f_k} += \frac{\tau_{e_{a_k},f_k}\ \delta_{a_k,k,l,m}}{\sum_{j'=0}^{\ell} \tau_{e_{j'},f_k}\ \delta_{j',k,l,m}}. \tag{2.21}$$

Thus to learn IBM model 2 parameters we do the same thing we did for IBM 1, with some small differences. First, we need to pick initial values for δ as well (all equal as before). Second, to get the fractional counts we multiply in the δs as in Equation 2.21. Last, the fractional counts thus obtained are used to get new counts not only for the τ but for the δ as well.

Once we have the proof that we have correctly computed model 1 expected counts, only a small change will show that model 2 is correct as well. Note that the assumption that all alignments were equal came in only at the very last step, going from Equation 2.15 to Equation 2.16. But for model 2 we have:

$$\begin{aligned} \mathrm{P}(A_k{=}j|\boldsymbol{e},m) &\neq \mathrm{P}(A_k{=}j'|\boldsymbol{e},m) \\ &= \delta_{j,k,l,m}. \end{aligned}$$

Substituting the δs into Equation 2.16, we get Equation 2.21 and we are done.

As we have already noted, in general EM is not guaranteed to find estimates of the parameters that maximize the probability of the data. It may find a local maximum but not the global one. As remarked in Section 2.2.3, IBM 1 has only has one maximum and thus we are guaranteed to find it as long as none of the initial estimates are zero. This is not true for IBM 2 — the initial estimates matter.

In practice, we start EM ignoring the δs. That is, we set them all equal, but do not update them for several iterations — long enough for the τs to be "burned in" with reasonable values. Only then do we start collecting fractional counts for the δs and resetting them in the E-step. In practice this tends to find very good estimates.

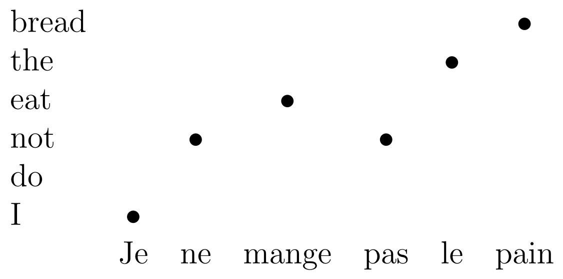

Figure 2.7: A graph of the alignment for the $(f)/(e)$ pair '*Je ne mange pas le pain*' and '*I do not eat the bread*'

2.4 Phrasal machine translation

Beyond IBM model 2 lie 3, 4, and 5. While they are important, certainly the most important difference between the MT we have presented and MT as it is practiced today is the introduction of phrases into machine translation — something not found in any of the IBM models.

You remember that IBM models 1 and 2 share the assumption that each French word is triggered by exactly one English word. This is captured by the alignments we introduced early in our discussion. Each alignment ties a French word to one, and only one English word (though it may be the null word).

Unfortunately, this assumption is far from correct. Consider the following $(f)/(e)$ pair:

Je ne mange pas le pain
I do not eat the bread

The best alignment is:

$< 1, 3, 4, 3, 5, 6 >$

'*Je*', '*mange*', '*le*', and '*pain*' align with '*I*', '*eat*', '*the*' and '*bread*' respectively. Both '*ne*' and '*pas*' align with '*not*'. Figure 2.7 shows a graph of this alignment, with the French words along the x axis and the English along the y axis. We have put a dot at the (x, y) position when $a_x = y$.

Notice that nothing aligns with the English word '*do*'. This is problematic when we want to decode, and here is where phrasal alignments kick in. We start the same way, but then we also do the reverse alignment — align each

bread						•+
the					•+	
eat			•+			
not		•		•+		
do		+				
I	•+					
	Je	ne	mange	pas	le	pain

Figure 2.8: Both English to French and French to English alignments

English word to exactly one French word. We do this by creating a second noisy-channel model, this time for computing $\mathrm{P}(\boldsymbol{e} \mid \boldsymbol{f})$ rather than the $\mathrm{P}(\boldsymbol{f} \mid \boldsymbol{e})$ we have been computing so far. Nothing in the IBM models cares about which language is which, so in principle this should be as simple as flipping the sentence pairs in the parallel corpus.

Going back to our '*do*' example, an English-to-French alignment might be:

$$< 1, 2, 4, 3, 5, 6 >$$

This is similar to the French-to-English, except this time we are forced to align '*do*' with something. In our example we have chosen to align it with '*ne*'. (Another possibility would be to align it to the null French word. This choice is not critical to our discussion.)

In Figure 2.8 we superpose the two alignments, representing the second alignment with the +'s. The figure clarifies a few things. First, '*Je/I*', '*le/the*', and '*pain/bread*' are in good shape. In each case there is a one-for-one alignment, and furthermore the alignment matches up with the words on at least one side. By this we mean that '*Je*' and '*I*' are both preceded by the start of sentence, '*le*' and '*the*' are both followed by words that are themselves aligned ('*pain*' and '*bread*'), and these last two words are preceded by aligned words and followed by the end of the sentence. This suggests that these words are reasonably translated in a one-to-one fashion.

None of the other words have this property. In particular, the relation between '*ne mange pas*' and '*do not eat*' is messy, and the words at their ends do not match up nicely with the words that precede/follow them. On the other hand, suppose we treat '*ne mange pas*' as a single phrase that should be translated into '*do not eat*' as a whole, not word for word. While the words within it do not match up very well, the phrases '*ne mange pas*' and '*do not eat*' match well in exactly the same way that the '*good*' words do: they are preceded and followed by good matches. That is the second key idea of phrasal MT — use graphs like that in Figure 2.8 as clues that some sequence of words should be aligned not individually but as phrases. We then keep alignment counts for these phrases, just as we did for individual words. Finally, at decoding time we treat such phrases as if they were single words.

Unfortunately, this simple version of phrasal MT does not work very well. We show here two ways in which we have grossly oversimplified things. First, the example in Figure 2.8 was deliberately chosen to be simple. In real examples, the graph of alignment pairs can get very messy and our vague description of how to find phrases would end up, say, aligning entire sentences as a single phrase. Second, even if we are lucky and never get phrases longer than, say, four words, the number of four-word phrases is very, very large, and there are not sufficient parallel corpora for us to collect statistics on even a fraction of what can occur. So we do not see merely '*ne mange pas*' and '*do not eat*'. Virtually any French verb and its English translation could be substituted here. What we really need to do is recognize that there is a pattern of the form '*ne* `french_verb` *pas*' that becomes '*do not* `English_verb`'. This is exactly what a real implementation does, but exactly how it works differs from system to system.

Actually, what we ought to do is to consider all possible phrasal translations and then, say, use EM to pick out good ones, just as we did for single words. Unfortunately this is not possible. Suppose we have a FE pair with each sentence thirty words long. How many phrasal pairs are there? Well, if we assume that phrases are always made up of contiguous words, there are 2^{29} possible phrases for the English, and the same number for the French. (Show this!) If each English phrase can align with any of the French ones, we get 2^{29} squared possibilities, or 2^{58} combinations. This is a seriously big number. So any method, such as EM, that requires simultaneously storing all expectations is out of the question. We come back to this point near the end of the book.

2.5 Decoding

There is, of course, one major problem with our MT efforts so far. We can gather probabilities, but we don't know how to use them to translate anything.

In this section we rectify this situation. However, rather than calling this section "translation," we call it "decoding," as this is what it is called in the MT community. The reason for this odd nomenclature is the origin of our fundamental theorem of MT, Equation 2.1, which is a particular instance of the noisy-channel model. As its name implies, the noisy-channel model was invented to overcome problems of noise on a communication channel. The key to all solutions to this problem is to encode the message in some way so as to make it possible to reconstruct the message even if noise obliterates portions of it. The process of finding the message from the coded version sent over the channel is, naturally enough, called "decoding." Since when viewed as a noisy-channel problem MT imagines that a message originally written in our target, say English, arrived written in the source, say French, the problem of getting the English from the French is thus a "decoding" problem.

The MT decoding problem is, in its most general form, very difficult. Consider again the MT fundamental equation (2.1) repeated here:

$$\arg\max_{\boldsymbol{e}} P(\boldsymbol{e} \mid \boldsymbol{f}) = \arg\max_{\boldsymbol{e}} P(\boldsymbol{e})P(\boldsymbol{f} \mid \boldsymbol{e}).$$

We have now shown how to get the required language model $P(\boldsymbol{e})$ and translation model $P(\boldsymbol{f} \mid \boldsymbol{f})$, but solving the equation requires iterating over all possible English sentences — a daunting task. (For those of you who have taken a computational theory class, the problem is NP-hard.)

2.5.1 Really dumb decoding

As a general rule it is almost always worth doing something really simple before trying to do something very hard. Sometimes the dumb method works and you save a lot of work. More often it doesn't, but you gain some insight into the problem.

Here is our first feeble attempt. All this time we have been using EM to estimate $P(\boldsymbol{f} \mid \boldsymbol{e})$. Let's redo the programs to compute $P(\boldsymbol{e} \mid \boldsymbol{f})$. We then take our French sentence and for each French word substitute the English word $t(f_i)$ which we define as

$$t(f_i) = \arg\max_{e} P(e \mid f_i).$$

So $t(f_i)$ is a function from a French word f_i to the English word e that is its most likely translation out of context. Then the translation of the sentence $\boldsymbol{f}$ is the concatenation of $t(f_i)$ for $1 \leq i \leq m$. Now the τ's estimate $P(\boldsymbol{e} \mid \boldsymbol{f})$ rather than $P(\boldsymbol{f} \mid \boldsymbol{e})$. Finding these is *really* easy. Just take the program you had earlier, and switch the input files! That's it.

Unfortunately the results are pretty bad.

Example 2.13: Looking at the output of such a very simple decoder is a good way to appreciate the complexities of MT. We start with an example where things go relatively well, and then descend rapidly into the realm of unfunny machine-translation jokes. (The classic, which is surely apocryphal, is that the Russian equivalent of "The spirit is willing but the flesh is weak" came out as "The vodka is good but the meat is rotten".)

English:	That is quite substantial .
French:	Ce est une somme considerable .
MT output:	That is a amount considerable .

Here the program got it right except for the reversal of '*amount considerable*'.

English:	I disagree with the argument advanced by the minister .
French:	Je ne partage pas le avis de le ministre .
MT output:	I not share not the think of the Minister .

'*Je ne partage pas*' means '*I do not share.*' However, the split '*ne*' and '*pas*' both get translated as '*not*', and the '*do*' in the English version, which from the French point of view would be spurious, is not included. '*Avis*' means '*opinion*', but one can easily imagine that it gets translated into a phrase like '*I think*'.

English:	I believe he is looking into the matter .
French:	Je crois que il est en train de etudier la question .
MT output:	I think that he is in doing of look the question .

'*En train de*' means '*in the process of*' but comes out as '*in doing of*'.

English:	My question very briefly is this .
French:	Voici tres brièvement de quoi il se agit .
MT output:	: very briefly of what he is point .

We thought ending with '*voici*' ('*this*') being translated as a colon would be a nice touch.

2.5.2 IBM model 2 decoding

So really simple techniques do not work and we must turn to complicated ones. To make things more concrete, we consider a decoder for IBM model 2. Unfortunately, while there are standard IBM models 1 to 5, there is no such thing as a standard decoder for any of them. What we present is as typical as any, but we cannot say anything stronger.

In one sense, decoding is simple. If we substitute Equation 2.20 into the MT fundamental theorem, we get:

$$\arg\max_{\boldsymbol{e}} \eta_{l,m} \sum_{\boldsymbol{a}} \prod_{k=1}^{m} \delta_{a_k,k,l,m}\ \tau_{e_{a_k},f_i}. \tag{2.22}$$

Thus decoding "reduces" to testing every possible English sentence and seeing which one is the arg max required in Equation 2.22.

Of course, this is not possible. Even if we restrict consideration to English sentences no longer than the French one, the number of such sentences grows exponentially with the lengths of the sentences involved. Furthermore, the problem is "NP hard" — it is (almost certainly) inherently exponential.

We have then a search problem — finding the needle in the haystack. Fortunately, we computational linguists are not the only people with search problems. A whole branch of artificial intelligence is devoted to them — *heuristic search*. More formally, we have a problem (constructing a sentence) that requires many steps to complete (selecting English words). There may be many possible solutions, in which case we have an evaluation metric for the quality of the solution. We need to search the space of partial solutions ($\boldsymbol{S}$) until we find the optimal total solution (the $\boldsymbol{e}$ with highest $\mathrm{P}(\boldsymbol{e} \mid \boldsymbol{f})$. That is, we have an algorithm something like that in Figure 2.9.

While this is very sketchy, several assumptions are built into it. First, we need to recognize a full translation when we see it. This is simple if our partial solutions record which French words have been assigned an English one. When they all have we are done. Second, we assume that the first solution we find is either the best or tied for the best. We come back to this point in a moment. Third, we assume that partial solutions fill in English words from left to right. That is, we never try to guess, say, the second word of the translation until the partial solution has already guessed the first. Nearly all MT decoding algorithms do things this way because it makes computing a n-gram language model $\mathrm{P}(\boldsymbol{E})$ easy.

```
1. add the empty partial solution (no words translated) to $\boldsymbol{S}$
2. repeat until termination
    (a) pick the partial solutions $s \in \boldsymbol{S}$ with maximum $h(s)$
    (b) if $s$ is a full solution, return it and stop
    (c) remove $s$ from $\boldsymbol{S}$
    (d) make copies of $s$ ($s'$) each with a different next possible English
        word $e'$
    (e) add all $s'$ to $S$
```

Figure 2.9: A* algorithm for IBM model 2 decoding

Lastly, we have assumed some function $h(s)$ that picks the partial solution to do next. There is no perfect way of always picking the best s to work on next, so h is a *heuristic* — a method that is not perfect but is much better than chance. Finding a good heuristic is the critical step — thus the name "heuristic search." A simple (but not very good) method would be next to expand the solution with the highest probability according to Equation 2.22. We come back to this later.

This algorithm is an instantiation of a general method named "A*" (pronounced a-star). In A* $h(s')$ is expected to return an estimate of the best solution emanating from s'. Note that if the heuristic is optimistic the first solution found is always the (or "a") best one. The reasoning is that when a complete solution is most highly ranked, all the incomplete ones have optimistic scores, and thus were we to expand them their values could only go down.

To make this all work we define a state s as a three-tuple $< h, \boldsymbol{e}, \boldsymbol{f} >$. Here h is s's heuristic score, $\boldsymbol{e}$ is the vector of English words, and $\boldsymbol{f}$ is a set of the French words that have been translated so far. Note that while $\boldsymbol{e}$ starts at the beginning of the English, $\boldsymbol{f}$ will not be in any particular order since the first English words need not align to the first French ones.

Actually, in our haste we said that we add a new English word to s. This is not necessary. What is necessary is that we add a new French word f to $\boldsymbol{f}$. We could decide to align f to an existing member of $\boldsymbol{e}$ since this is allowed in both IBM models 1 and 2. When $\boldsymbol{f}$ includes all of the words in the sentence to be translated, we have a full solution.

The really difficult part of all of this is finding a good heuristic. An obvious one is the probability estimate for the solution so far according to Equation 2.22. This is better than picking an s at random, but is is still very poor. Note that this value gets exponentially smaller with the length of the partial solution. This means, for example, that the algorithm would keep considering alternative ways to put together two-word sequences to avoid lowering the probability by looking at a third word. Or again, we said that $h(s)$ is an estimate of the quality (probability) of the best final state descending from s. The probability of the current sequence is a very poor estimate of this.

Example 2.14: Suppose that the correct translation of some sentence is "This will reverberate for a while." '*Reverberate*' is an uncommon word, and the bigram probability P('*reverberate*' | *will*) is low, say 10^{-5}. Thus the probability of the English string so far will go down expeditiously as soon as it is translated. With the probability-so-far heuristic the program would try to put off translating it as long as possible, perhaps heading toward a solution like "For a while this will reverberate."

So finding a good heuristic is the heart of the MT decoding problem and the literature is large. One method, *stack decoding*, gives up on the idea of always looking at the best partial solution, and instead keeps multiple sets of solutions, one for each length English string. The algorithm then always divides its time between the lengths so that the longer ones are always being expanded.

Alternatively, we can find a better h. According to Equation 2.22, for each English word (roughly) there will be a probability τ of it generating a French word, a distortion δ depending on relative word positions, and a bigram probability $\mathrm{P}(e_i \mid e_{i-1})$ for the language model. To keep the heuristic optimistic, we look for a better maximum values for all of these.

Consider better estimates for the probabilities of future δ's. If s has j English words already selected, then any remaining French word f_k will eventually be paired with an English word with position $j' > j$. Thus a maximum for the delta associated with f_k is $\max_{j'>j} \delta_{j',k}$.

Example 2.15: Suppose we are translating '*Je mange le pain blanc.*' and s has translated the first two words as '*I eat*'. The next English word will come from one of the last three French. Thus it must have as a δ either $\delta_{3,3}$ (if we chose to translate '*le*'), $\delta_{3,4}$ (for '*pain*'), or $\delta_{3,5}$. Typically $\delta_{3,3}$ would be the largest, perhaps 0.3. Thus 0.3 will be multiplied in when computing h.

The same sorts of reasoning can then be applied to the other factors that make up $\mathrm{P}(\boldsymbol{f} \mid \boldsymbol{e})$, i.e. the τ's and bigram probabilities.

The alert reader might be slightly worried about the line in Figure 2.9 that says "make copies of s (s') each with a different next possible English word e'". How many next English words are there? A very large number could make the algorithm impractical. Take another look at Figure 2.4. There we see three English words with significant probabilities of translating into '*pain*'. Thus if '*pain*' is a still untranslated word, there are at least these three words which could somehow come next. Of most concern is '*mouths*'. We can be reasonably confident that there are few real English translations of '*pain*', but how many relatively unrelated words show up?

Fortunately, we did not pick the words in the table at random. Rather, we picked a few that would show interesting situations. Words like '*mouths*' are, in fact, pretty rare. Furthermore, if our heuristic is a good one, even these might not cause that much of a problem. Suppose every untranslated f suggests, on average, 20 possible e'. So a 20-word sentence would start with 400 s' on the first iteration. This might sound bad, but if most of them have very low h, they simply drop to the bottom of $\boldsymbol{S}$ and we never consider them again. And if h is any good, this is what should happen to '*mouths*' as a translation of '*pain.*' While $\tau_{mouths,\,pain}$ might be too high, any good estimate of the bigram probability of '*mouths*' should be quite small. This in turn would give a low h.

2.6 Exercises

Exercise 2.1: Consider the following English sentences:

The dog drank the coffee.
The man drank the soup.
The coffee pleased the man.

We have a parallel corpus of the same sentences in Shtetlish, a language spoken in a tiny town somewhere in the Carpathians:

Pes ten kava ta pil.
Muzh ten polevka ta pil.
Kava ta muzh ten libil.

Give the correct alignment for the sentences

> *The soup pleased the dog.*
> *Polevka ta pes ten libil.*

Exercise 2.2: Compute the values of $n_{eat,mange}$ and $\tau_{eat,mange}$ after the first and second iterations of EM given the following training data:

Elle mange du pain	*She eats bread*
Il mange du boef	*He eats beef*

Exercise 2.3: In Section 2.2.1 we noted that $n_{e,\circ}$ in general will not be the same as the number of times the word e appears in the corpus. Explain.

Exercise 2.4: Would $< 0, 0, 0 >$ be a legal alignment for a French sentence of length three? (Extra credit: discuss the philosophical implications.)

Exercise 2.5: We suggest starting EM with all $\tau_{e,f}$'s the same. However, as long as they are the same (and non-zero) any one value works as well as any other. That is, after the first M-step the τs you get will not be a function of the τs you stated with. For example, you could initialize all τs to one. Prove this. However, althought the τs at the end of the M-step would not vary, there would be some other number we suggest you compute that *would.* What is it?

Exercise 2.6: On the first iteration of IBM model 1 training, the word that would align the most often with '*pain*' is almost certainly going to be '*the.*' Why? Yet in Figure 2.6 we did not mention '*the*' as a translation of '*pain.*' Why is that?

Exercise 2.7: In our derivation of Equation 2.13 we stated that since all alignments are equally probable in IBM model 1 it follows that

$$\mathrm{P}(A_k{=}j|\boldsymbol{e}) = \mathrm{P}(A_k{=}j'|\boldsymbol{e}). \tag{2.23}$$

Actually, this deserves a bit more thought. Note that it is *not* the case that

$$\mathrm{P}(A_k{=}j|\boldsymbol{e},\boldsymbol{f}) = \mathrm{P}(A_k{=}j'|\boldsymbol{e},\boldsymbol{f}). \tag{2.24}$$

First, explain why Equation 2.24 is false. Then explain why Equation 2.23 is nevertheless true.

Exercise 2.8: Consider the two IBM model 2 distortion probabilities $\delta_{1,2,10,11}$ and $\delta_{1,2,11,11}$. Will these be close in value, or more than a factor of 10 different? Why?

Exercise 2.9: We noted that it can be a good idea to make distortion probabilities depend only on word positions and the length of the French sentence. This has some negative consequences, however. Show that during training, such a model will either (a) assign a zero probability to a legal alignment, or (b) assign nonzero probabilities to impossible alignments.

2.7 Programming problems

Problem 2.1: **Machine Translation with Very Dumb Decoding**

Write an MT program based upon IBM model 1 and our very dumb decoder, which simply goes through the incoming French sentence and for each word f_i outputs $\arg\max_{e_j} P(e_j \mid f_i)$.

Start by building an IBM model 1 parameter estimation program. It takes as arguments two input files, one for the French half of the parallel corpus, one for the English half. Make sure that your program does not care which is which, so that by switching the order of the files your program will switch between computing $P(e_j \mid f_i)$ and $P(f_i \mid e_j)$.

To initialize EM we assign all translation probabilities the same value. As noted in Exercise 2.5 you can simply set them to 1. Equation 2.9 tells us how to compute the fractional counts for each word (the e-step), and then at the end of each iteration Equation 2.7 does the m-step. Ten iterations or so should be sufficient for all the parameter values to settle down.

Before writing the decoder part of this assignment, print out some probabilities for French words with reasonably obvious English translations. Foremost should be punctuation. If you do not know any French, you can also look at the French for words that look like English words and occur several times. Many French words have been adopted into English.

Finally, the decoder should be just a few lines of code. Two details. Given the (low) quality of our decoder, only very simple sentences have any chance of being comprehensible after translation. Only translate French sentences of length ten or less. Also, when your program encounters French words it has never seen before, just pass them through to the English output without change.

Use `English-senate-0.txt` and `french-senate-0.txt` as the training data. We are not tuning any smoothing parameters, so there is no particular need for held-out data. Translate `french-senate-2.txt` and save your translation to a new file.

Problem 2.2: MT with a Simple Noisy-Channel Decoder

Now we'll try to do better. For this assignment use the IBM model 1 decoder from the last programming assignment, but now use it to compute the reverse translation probabilities, $P(f_i \mid e_j)$. In addition, use the English bigram language model from the programing assignment in Chapter 1. The only thing that changes is our decoder.

In this version we again translate the French one word at a time. Now, however, rather than maximizing $P(e_j \mid f_i)$, we maximize $P(e_j \mid e_{j-1})P(f_i \mid e_j)$. As we are going left to right one word at a time, we will know the previous English word at each step. (Set the zeroth word to $\triangleright$.)

Use the same training and test data as in Problem 1.

Problem 2.3: Evaluation

Evaluate the two different translators according to the *F-score* of their output. The F-score is a standard measure of accuracy defined as the *harmonic mean* of *precision* and *recall*:

$$F = 2\frac{precision \cdot recall}{precision + recall}. \tag{2.25}$$

Precision is the number of correct results divided by the number of all returned results, and recall is the number of correct results divided by the number of results that should have been returned. In this case, we are counting the number of individual word tokens translated correctly. You may consider a translation of the word "pain" in `french-senate-2.txt` to "bread" as correct if "bread" occurs anywhere in the corresponding sentence in `english-senate-2.txt`. The total number of returned results is, of course, the word-token count of your translated output file (which will be exactly the same as that of `french-senate-2.txt`), and the number of results that should have been returned is the word-token count of `english-senate-2.txt`.

Which translation looks better to you (this is a rather subjective question)? Which gives the better F-score? Why? What does this tell us about our two different decoders? What about our chosen method of evaluation?

Include the answers to these questions in your README.

2.8 Further reading

For the most part, the best key words for MT are the obvious ones used in this chapter. However, if you want to download the *Canadian Hansard Corpus*, use this as the search key in a typical search engine; Google Scholar will list only papers citing the corpus. Another corpus is the *Europarl Corpus*.

You might want to try out an *open-source machine translation* system.

A recent trend is *grammar-based machine translation*, also called *hierarchical phrase-based* MT.

The most frequent out-of-vocabulary items in MT are names of people and places. The best way to handle them is to learn how to *transliterate* them to get English equivalents.

One big problem is translation of a *low-resource language* — a language for which there are few parallel corpora. One way to do this is to use a *bridging language* — a language similar to the low-resource target for which we have much more data. For example, to help translate Portuguese, use data from Spanish, or for Slovak, use Czech.

Finally, we would be remiss if we did not mention one of the best tutorials on MT, Kevin Knight's *MT Workbook*. The material covered is much the same as that in this chapter, but from a slightly different point of view. It is also enjoyable for Kevin's wacky sense of humor.

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 begin marker $x_0 = \triangleright$ and an end 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 $\boldsymbol{x}$, 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

$$\begin{tikzcd}
Y_0 = \triangleright \arrow[r] & Y_1 = V \arrow[r] \arrow[d] & Y_2 = D \arrow[r] \arrow[d] & Y_3 = N \arrow[r] \arrow[d] & Y_4 = \triangleleft \\
 & X_1 = flour & X_2 = the & X_3 = pan &
\end{tikzcd}$$

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

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 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

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

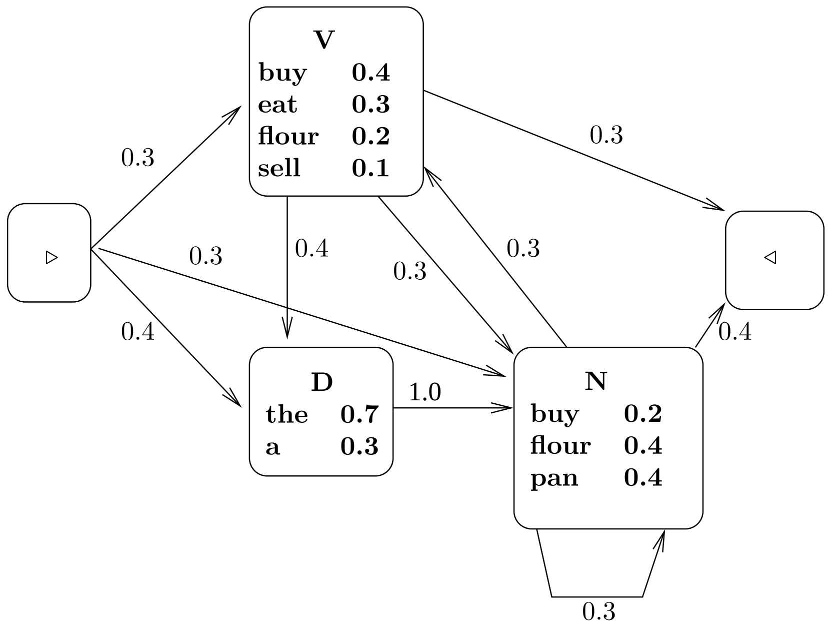

Figure 3.2: Example of HMM for POS tagging '*flour pan*', '*buy flour*'

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

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

$$\begin{tikzcd}
 & Y_1 = \text{‘}N\text{’} \arrow[r] \arrow[rdd] & Y_2 = \text{‘}N\text{’} \arrow[dr] & \\
Y_0 = \triangleright \arrow[ur] \arrow[dr] & & & Y_3 = \triangleleft \\
 & Y_1 = \text{‘}V\text{’} \arrow[r] \arrow[ruu] & Y_2 = \text{‘}V\text{’} \arrow[ur] & \\
 & X_1 = flour & X_2 = pan &
\end{tikzcd}$$

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

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 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 2·2 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" — historically, it was used as an algorithm for decoding noisy signals.

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,

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

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 $X_{i-1} = x_{i-1}$

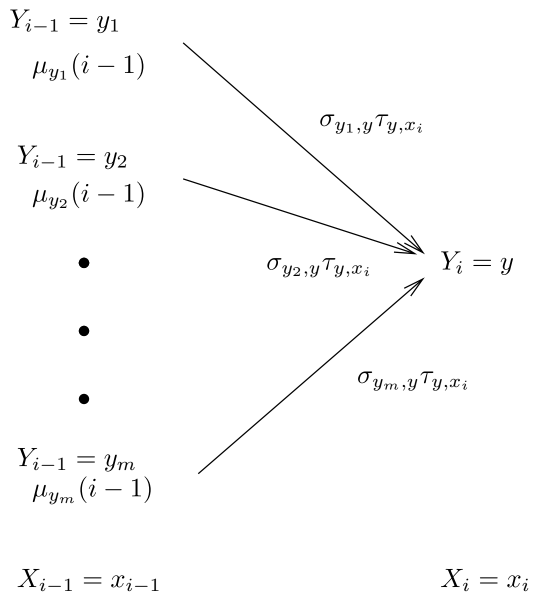

Figure 3.4: A piece of a trellis showing how to compute $\mu_y(i)$ if all of the $\mu_{y'}(i-1)$ are known.

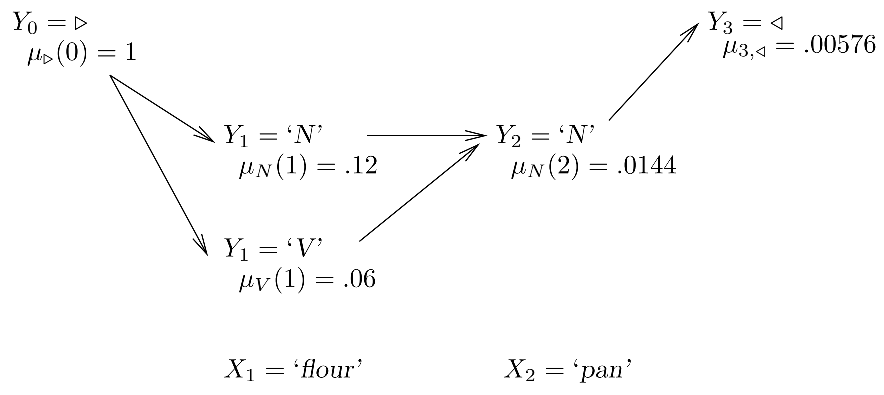

Figure 3.5: The trellis for '*flour pan*' showing the $\mu_y(i)$'s

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 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.

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

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.

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

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 $\mu_{y\prime}(i-1)$ 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.:

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

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.

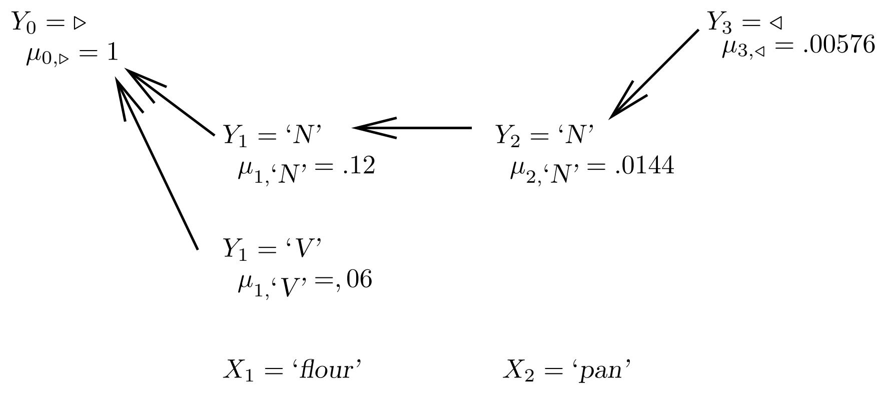
$Y_0 = \triangleright$
$\mu_{0,\triangleright} = 1$
$Y_3 = \triangleleft$
$\mu_{3,\triangleleft} = .00576$
$Y_1 = $ '*N*'
$\mu_{1,\text{'}N\text{'}} = .12$
$Y_2 = $ '*N*'
$\mu_{2,\text{'}N\text{'}} = .0144$
$Y_1 = $ '*V*'
$\mu_{1,\text{'}V\text{'}} =, 06$
$X_1 = $ '*flour*'
$X_2 = $ '*pan*'

Figure 3.6: Figure 3.5 with back pointers added.

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 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.

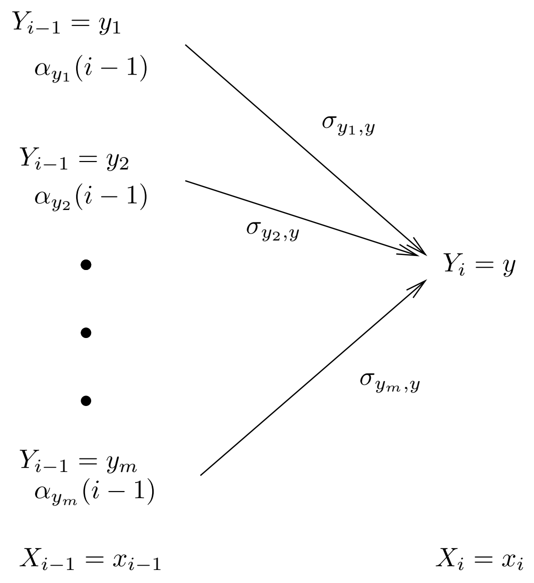

Figure 3.7: Piece of a trellis showing how forward probabilities are calculated

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 $\alpha_y(i)$. 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:

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

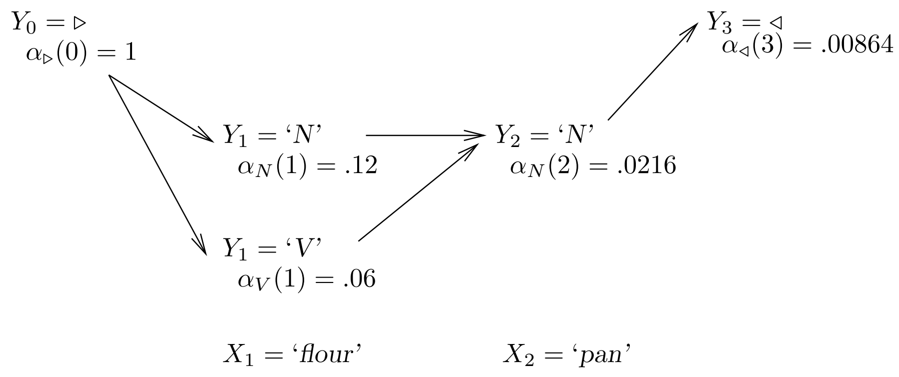

Figure 3.8: Forward probability calculations for '*flour pan*' HMM

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

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

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

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

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 $\alpha_{\lhd}(n+1) = \mathrm{P}(\boldsymbol{x})$.

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 $i = n+1$,

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

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

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

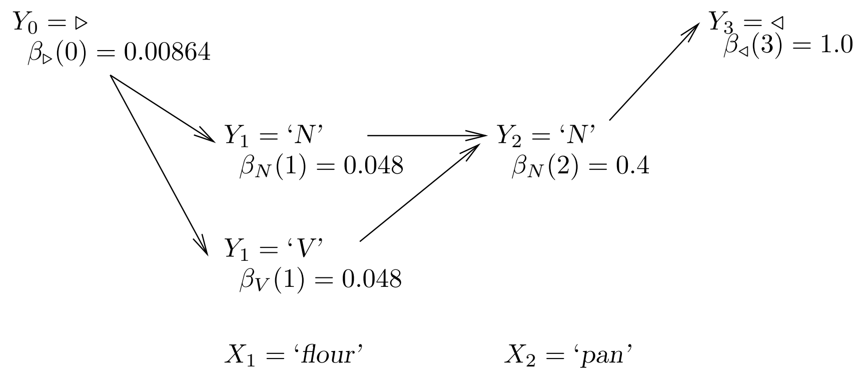

Figure 3.9: Computing backward probabilities for the '*flour pan*' example

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 $Y_{i+1} = y\prime$. We assume we know the probability of generating everything after that, $\beta_{y\prime}(i+1)$. To get $\beta_y(i)$, we also need to multiply in the probability of the symbol generated at $i+1$, $\tau_{y\prime,x_{i+1}}$, and the probability of getting to $y\prime$ from y, $\sigma_{y,y\prime}$.

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*'):

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

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

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

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 precessing $\boldsymbol{x}$. 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.

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

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 $y\prime$ 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

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

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

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.

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

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 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 $\delta_{j,k,l,m}$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

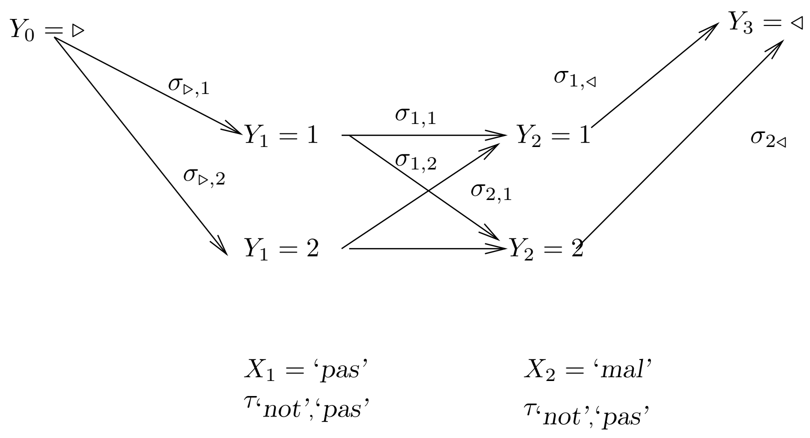
$Y_0 = \triangleright$
$Y_3 = \triangleleft$
$\sigma_{\triangleright,1}$
$\sigma_{\triangleright,2}$
$\sigma_{1,1}$
$\sigma_{1,2}$
$\sigma_{2,1}$
$\sigma_{1,\triangleleft}$
$\sigma_{2\triangleleft}$
$Y_1 = 1$
$Y_1 = 2$
$Y_2 = 1$
$Y_2 = 2$
$X_1 =$ '*pas*'
$\tau_{\text{'not','pas'}}$
$X_2 =$ '*mal*'
$\tau_{\text{'not','pas'}}$

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

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 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:

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

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.

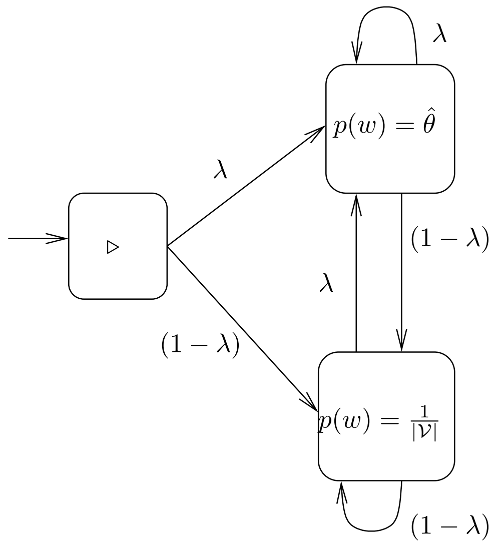

Figure 3.12: The HMM coresponding to Equation 3.18

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 $\mathrm{P}_{\hat{\boldsymbol{\theta}}}(W)$ and the *uniform distribution* $\mathrm{P}_1(W)$ that assigns the same probability to each word $w \in \mathcal{W}$.

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

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:

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

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 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 either state A or B, 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, and the probabilitity of getting to those states is the same for both A and B. 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 A to A (which is the same as the transition B to A, is

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

When we end up in state B we have:

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

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 A when generating "the" is $(.08/(.08 + 2.010^{-6})) \approx 1.0$, and for transition to state B is $2.010^{-6}/(.08+2.010^{-6}) \approx 2.510{-}5$. For $*U*$the expectation of transitioning to A is zero $(= 0/(0 + 210^{-6}))$ and has an expectation of one for transitioning to state B $(= 210^{-6}/(0 + 210^{-6}))$.

3.9 Part-of-speech induction

Part-ofspeech (POS) tagging is a standard example for HMM 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.

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 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

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

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 ws 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 this grouping does pretty well. 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 $\triangleright$ and $\triangleleft$).

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

(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 that goes through state a at position i assigns zero probability to 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? 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 $*$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, handwriting recognition, land-mine detection, you name it.

There are many HMM variations. Perhaps the most important of these is the *conditional random field* (CRF) model which in many situations outperforms HMMs. CRFs are discriminative rather than generative models. That is, they try to estimate the probability of the hidden variables directly ($\mathrm{P}(\boldsymbol{y} \mid \boldsymbol{x})$) rather than computing the joint probability $\mathrm{P}(\boldsymbol{y}, \boldsymbol{x})$ as do HMMs. More generally, there is a wide variety of *discriminatively trained HMMs.*

Factorial HMMs are models in which several HMMs can be thought of as simultaniously generating the visible string (also called *coupled HMMs*).

As HMMs are a staple in the speech recognition community, there is a large literature on *continuous HMMs* — where the outputs are real numbers representing, e.g., the energy/frequency of an acoustic signal.

Chapter 4

Parsing and PCFGs

4.1 Introduction

In natural languages like English, words combine to form phrases, which can themselves combine to form other phrases. For example, in the sentence "Sam thinks Sandy likes the book", the words '*the*' and '*book*' combine to form the noun phrase (NP) '*the book*', which combines with the verb '*likes*' to form the verb phrase (VP) '*likes the book*', which in turn combines with '*Sandy*' to form the embedded clause or sentence (S) '*Sandy likes the book*'. Parsing, which is the process of recovering this kind of structure from a string of words, is the topic of this chapter.

4.1.1 Phrase-structure trees

It's natural to represent this kind of recursive structure as a *tree*, as in Figure 4.1. They are called *phrase-structure trees* because they show how the words and phrases combine to form other phrases.

Notice that trees such as this are usually drawn upside-down, with the *root node* at the top of the tree. The *leaf nodes* of the tree (at the bottom, of course!), which are labeled with words, are also known as *terminal nodes*. The sequence of labels on the terminal nodes is called the *terminal yield* or just the *yield* of the tree; this is the string that the tree describes. The nodes immediately dominating (i.e., above) the terminal nodes are called *preterminal nodes*; they are labeled with the word's part-of-speech (the same parts-of-speech that we saw in the previous chapter). The *phrasal nodes* are above the preterminals; they are labeled with the *category* of the phrase (e.g.,

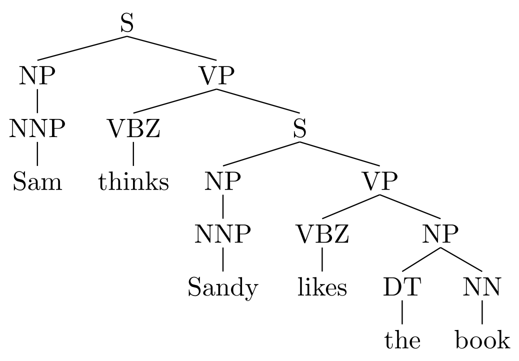

Figure 4.1: A phrase-structure tree for the sentence “Sam thinks Sandy likes the book”. The labels on the nonterminal nodes indicate the category of the corresponding phrase, e.g., ‘*the book*’ is an ‘*NP*’ (Noun Phrase).

‘*NP*’, ‘*VP*’, etc.). As you would expect, the *nonterminal nodes* are all of the nodes in the tree except the terminal nodes.

Parsing is the process of identifying this kind of structure for a sentence. It is one of the best-understood areas of computational linguistics. Arguably *the* best. The literature is very large, and, at least if you are parsing newspaper articles, several good parsers are downloadable from the web. However, there is something of a disconnect between the research effort, and clearly usable results. For example, parsing would seem to be useful in machine translation, especially when translating between languages with radically different word orders. One might try to view translation as a kind of “rotation” of phrase structure trees (viewed as a kind of mobile sculpture). In the last couple of years this has been shown to work, and now major English-Japanese MT programs use this approach.

This success to some degree justifies the previous ephasis on parsing, but there was never much doubt (at least among “parsing people”) that the area would one day come into wide usage. The reason is simple: people can understand endless new sentences. We conclude from this that we must understand by building the meaning of whole sentences out of the meaning of sentence parts. Syntactic parsing (to a first approximation) tells us what those parts are, and roughly in what order they combine. (Examples like Figure 4.1 should make this seem plausable.) Unfortunately we computational linguists know little of “meanings” and how they combine. When we

```
(S (NP (NNP Sam))
   (VP (VBZ thinks)
       (S (NP (NNP Sandy))
          (VP (VBZ likes)
              (NP (DT the)
                  (NN book))))))
```

Figure 4.2: The phrase structure tree of Figure 4.1 for '*Sam thinks Sandy likes the book*' in bracket notation. Note that the indentation is purely for aesthetic reasons; the structure is indicated by the opening and closing brackets.

do, the importance of parsing will be much more obvious.

It's convenient to have a standard (one dimensional) notation for writing phrase structure trees, and one common one is based on the bracketted expressions of the programming language Lisp. Figure 4.2 gives the same tree as in Figure 4.1 in bracket notation.

4.1.2 Dependency trees

The phrase-structure tree does not directly indicate all of the structure that the sentence has. For example, most phrases consist of a *head* and zero or more *dependents*. Continuing with the previous example, '*likes*' might be analysed as the head of the VP '*likes the book*' and the S '*Sandy likes the book*', where the NPs '*the book*' and '*Sandy*' are both dependents of '*likes*'.

A *dependency tree* makes these dependencies explicit. The nodes in the dependency tree are the words of the sentence, and there is an arc from each head to the heads of all of its dependent phrases. Figure 4.3 depicts a dependency tree for '*Sam thinks Sandy likes the book*'. Sometimes the dependency arcs are labeled to indicate the type of dependency involved; e.g., '*SUBJ*' indicates a subject, '*OBJ*' indicates a direct object and '*COMP*' indicates a verbal complement.

Of course the phrase-structure tree and the dependency tree for a sentence are closely related. For example, it is straightforward to map a phrase-structure tree to an unlabeled dependency tree if one can identify the head of each phrase.

There is much more to say about both phrase structure and dependency

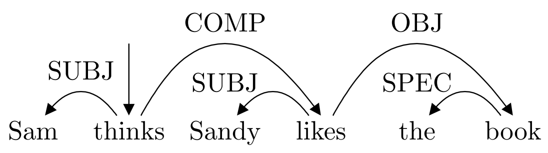

Figure 4.3: A dependency tree for the sentence "Sam thinks Sandy likes the book". The labels on the arcs indicate the type of dependency involved, e.g., '*Sandy*' is the subject of '*likes*'.

structure, but we only have space to make a few comments here. Most importantly, there is no agreement on the correct grammar even for a well-studied language like English. For example, what's the structure of a sentence like '*the more, the merrier*'?

It is not even clear that trees are the best representation of phrase structure or dependency structure. For example, some sentences have *discontinuous phrases*, which often correspond to *nonprojective dependency structures* (i.e., dependency structures with crossing dependencies). While these are more common in languages with relatively free word order such as Czech, there are English constructions which plausibly involve such discontinuities. However, as we will see in this chapter, there are such significant computational advantages to only considering structures that can be represented as trees that it is common to do so.

Example 4.1: Figure 4.4. shows an English example involving nonprojective structure. The phrase '*a book with a red cover*' is discontinuous in this example because it is interrupted by '*yesterday*'. Such cases are called "nonprojective" because when projected onto the plane they shows up as "crossing lines" in the phrase structure tree and the dependency structure.

In the face of puzzles and disagreements there are two ways to go, try to resolve them, or ignore them. Work in statistical parsing has mostly done the latter. Dedicated linguists and computational linguists have roughed out grammars for some languages and then hired people to apply their grammar to a corpus of sentences. The result is called a *tree bank*. Researchers then use the tree bank as the basis for a parser in the usual way of creating a train/development/test split and accept whatever the tree bank says as the gold standard. To the degree that most grammatical formalisms tend to

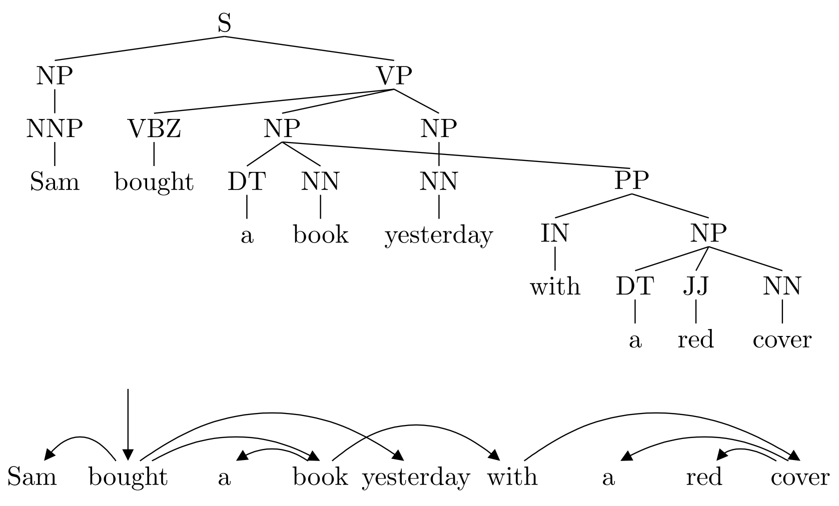

Figure 4.4: A discontinuous structure in English

capture the same regularities this can still be a successful strategy even if no one formalism is widely preferred over the rest. This seems to be the case.

4.2 Probabilistic context-free grammars

Probabilistic Context-Free Grammars (PCFGs), are a simple model of phrase-structure trees. We start by explaining what a formal language and a grammar are, and then present Context-Free Grammars and PCFGs (their probabilistic counterpart).

4.2.1 Languages and grammars

A formal language is a mathematical abstraction of a language. It is defined in terms of a *terminal vocabulary*, which is a finite set $\mathcal{V}$ of *terminal symbols* (or terminals for short), which are the atomic elements out of which the expressions of the language are constructed. For example $\mathcal{V}$ might be a set of English words, or it could be the characters 'a'–'z' (e.g., if we wanted to model the way that words are built out of letters).

Given a set $\mathcal{V}$ of terminals, $\mathcal{W} = \mathcal{V}^\star$ is the set of all finite sequences or strings whose elements are members of $\mathcal{V}$. ($\mathcal{V}^\star$ also includes the *empty string*

$$\begin{array}{rcl}
\mathcal{V} & = & \{\text{book}, \text{likes}, \text{Sandy}, \text{Sam}, \text{the}, \text{thinks}\} \\
\mathcal{N} & = & \{\text{DT}, \text{NNP}, \text{NP}, \text{S}, \text{VBZ}, \text{VP}\} \\
S & = & \text{S} \\
\mathcal{R} & = & \left\{ \begin{array}{ll}
\text{DT} \rightarrow \text{the} & \text{NN} \rightarrow \text{book} \\
\text{NNP} \rightarrow \text{Sam} & \text{NNP} \rightarrow \text{Sandy} \\
\text{NP} \rightarrow \text{NNP} & \text{NP} \rightarrow \text{DT NN} \\
\text{S} \rightarrow \text{NP VP} & \text{VBZ} \rightarrow \text{likes} \\
\text{VBZ} \rightarrow \text{thinks} & \text{VP} \rightarrow \text{VBZ NP} \\
\text{VP} \rightarrow \text{VBZ S} &
\end{array} \right\}
\end{array}$$

Figure 4.5: A context-free grammar which generates the phrase-structure tree depicted in Figure 4.1. The start symbol is 'S'.

ϵ). A *language* is a subset of $\mathcal{W}$. A *grammar* is a finite specification of a language. (A language can contain an infinite number of strings, or even if it is finite, it can contain so many strings that it is not practical to list them all). A *probabilistic language* is a probability distribution over $\mathcal{W}$, and a *probabilistic grammar* is a finite specification of a probabilistic language. (We will often drop the "probabilistic" when clear from the context).

A grammar may also provide other information about the strings in the language. For example, it is common in computational linguistics to use probabilistic grammars whose support is $\mathcal{W}$ (i.e., they assign non-zero probability to every string in $\mathcal{W}$), and whose primary purpose is to associate strings with their phrase-structure trees.

4.2.2 Context-free grammars

A context-free grammar is perhaps the simplest possible model of phrase-structure trees. Formally, a *context-free grammar* (CFG) is a quadruple $G = (\mathcal{V}, \mathcal{N}, S, \mathcal{R})$, where $\mathcal{V}$ and $\mathcal{N}$ are disjoint finite sets of *terminal symbols* and *nonterminal symbols* respectively, $S \in \mathcal{N}$ is a distinguished nonterminal called *start symbol*, and $\mathcal{R}$ is a finite set of *rules* or *productions*. A rule $A \rightarrow \beta$ consists of a parent nonterminal $A \in \mathcal{N}$ and children $\beta \in (\mathcal{N} \cup \mathcal{V})^\star$. Figure 4.5 contains an example of a context-free grammar.

Productions of the form $A \rightarrow \epsilon$, where ϵ is the empty string, are called *ep-*

S
NP VP
NNP VP
Sam VP
Sam VBZ S
Sam thinks S
Sam thinks NP VP
Sam thinks Sandy VP
Sam thinks Sandy VBZ NP
Sam thinks Sandy likes NP
Sam thinks Sandy likes DT NN
Sam thinks Sandy likes the NN
Sam thinks Sandy likes the book

Figure 4.6: A derivation of '*Sam thinks Sandy likes the book*' using the context-free grammar presented in Figure 4.5 on page 114.

silon rules. A CFG that does not contain any epsilon rules is called *epsilon-free*. While everything we say about CFGs in this chapter generalizes to CFGs that contain epsilon productions, they do complicate the mathematical and computational treatment, so for simplicity we will assume that our grammars are epsilon-free.

Context-free grammars were originally thought of as a rewriting system, which explains the otherwise curious nomenclature and notation. The rewriting process starts with a string that contains only the start symbol 'S'. Each rewriting step consists of replacing some occurence of a nonterminal A in the string with β, where $A \to \beta \in \mathcal{R}$ is a rule in the grammar. The rewriting process terminates when the string contains no nonterminals, so it is not possible to apply any more rewriting rules. The set of strings that can be produced in this way is the language that the grammar specifies or *generates*.

Example 4.2: Figure 4.6 gives a derivation of the string "Sam thinks Sandy likes the book" using the grammar of Figure 4.5.

The derivational view of context-free grammars also explains why they are called "context-free". A context-sensitive grammar differs from a context-free one in that the rules come with additional restrictions on the contexts in which they can apply.

Even though historically context-free grammars were first described as rewriting systems, it is probably more useful to think of them as specifying or generating a set of phrase-structure trees. The rules of a context-free grammar can be seen as specifying the possible *local trees* that the tree can contain, where a local tree is a subtree that consists of a parent node and its sequence of children nodes.

In more detail, a context-free grammar $G = (\mathcal{V}, \mathcal{N}, S, \mathcal{R})$ *generates* a tree t iff t's root node is labeled S, its leaf nodes are labeled with terminals from $\mathcal{V}$, and for each local tree ℓ in t, if ℓ's parent node is labeled A and its children are labeled β, then $\mathcal{R}$ contains a rule $A \to \beta$. The set of all trees that G generates is $\mathcal{T}_G$, where we will drop the subscript when clear from the context. G generates a string $w \in \mathcal{W}$ iff G generates a tree that has w as its terminal yield.

Example 4.3: The CFG in Figure 4.5 on page 114 generates the phrase-structure tree depicted in Figure 4.1 on page 110, as well as an infinite number of other trees, including trees for rather bizarre sentences such as '*the book thinks Sam*' as well as impeccable ones such as '*Sandy likes Sam*'.

Parsing is the process of taking a CFG G and a string w and returning the subset $\mathcal{T}_G(w)$ of the trees in $\mathcal{T}_G$ that have w as their yield. Note that $\mathcal{T}_G(w)$ can contain an infinite number of trees if $\mathcal{R}$ has epsilon productions or unary productions (i.e., productions of the form $A \to B$ for $A, B \in N$), and even if $\mathcal{T}_G(w)$ is finite its size can grow exponentially with the length of w, so we may be forced to return some kind of finite description of $\mathcal{T}_G(w)$ such as the *packed parse forest* of section 4.3.

One way to think about a CFG is as a kind of "plugging system". We imagine that our terminal and nonterminal symbols are different plug shapes, and that our goal is to somehow connect up a sequence of terminal plugs to a single socket labeled with the start symbol S. Each rule $A \to \beta \in \mathcal{R}$ is a kind of adaptor (maybe a bit like a surge suppressor board) that has a sequence of sockets β and a plug A. When parsing our goal is to find a way of connecting up all of the terminals via the rules such that there are no unconnected plugs or sockets, and everything winds up plugged into the start socket S.

4.2.3 Probabilistic context-free grammars

Probabilistic Context-Free Grammars (PCFGs) extend context-free grammars by associating a probability $\rho_{A\to\beta}$ with each rule $A \to \beta \in \mathcal{R}$ in the

$$\begin{array}{llll}
\rho_{\mathrm{DT}\to\mathrm{the}} = 1.0 & & \rho_{\mathrm{NN}\to\mathrm{book}} = 1.0 \\
\rho_{\mathrm{NNP}\to\mathrm{Sam}} = 0.7 & & \rho_{\mathrm{NNP}\to\mathrm{Sandy}} = 0.3 \\
\rho_{\mathrm{NP}\to\mathrm{NNP}} = 0.2 & & \rho_{\mathrm{NP}\to\mathrm{DT\ NN}} = 0.8 \\
\rho_{\mathrm{S}\to\mathrm{NP\ VP}} = 1.0 & & \rho_{\mathrm{VBZ}\to\mathrm{likes}} = 0.4 \\
\rho_{\mathrm{VBZ}\to\mathrm{thinks}} = 0.6 & & \rho_{\mathrm{VP}\to\mathrm{VBZ\ NP}} = 0.9 \\
\rho_{\mathrm{VP}\to\mathrm{VBZ\ S}} = 0.1 & &
\end{array}$$

Figure 4.7: The rule probability vector $\boldsymbol{\rho}$ which, when combined with CFG in Figure 4.5 on page 114, specifies a PCFG which generates the tree depicted in Figure 4.1 on page 110 with probability approximately 3×10^{-4}.

grammar. Informally, $\rho_{A\to\beta}$ is the conditional probability that the nonterminal A expands to β. The probability of a tree generated by the PCFG is just the product of the probabilities of the rules used to derive that tree. PCFGs are generative probability models in the sense we described in Section 1.3.3. The above description of how trees are generated in a CFG is their generative story.

More formally, a PCFG G is a quintuple $G = (\mathcal{V}, \mathcal{N}, S, \mathcal{R}, \boldsymbol{\rho})$ where $(\mathcal{V}, \mathcal{N}, S, \mathcal{R})$ is a CFG and $\boldsymbol{\rho}$ is a vector of real numbers in $[0, 1]$ that satisfies:

$$\sum_{A\to\beta\in\mathcal{R}_A} \rho_{A\to\beta} = 1 \qquad (4.1)$$

where $\mathcal{R}_A = \{A \to \beta \in R\}$ is the set of all rules in $\mathcal{R}$ whose parent is A. This condition is natural if we interpret $\rho_{A\to\beta}$ as the conditional probability of A expanding to β. It simply says that the probabiltities of all of the rules expanding A sum to one. Figure 4.7 gives an example of $\boldsymbol{\rho}$ for the CFG presented earlier in Figure 4.5 on page 114.

A PCFG G defines a probability distribution $\mathrm{P}_G(T)$ over trees $T \in \mathcal{T}_G$ as follows:

$$\mathrm{P}_G(T = t) = \prod_{A\to\beta\in R} {\rho_{A\to\beta}}^{n_{A\to\beta}(t)}$$

where $n_{A\to\beta}(t)$ is the number of times the local tree with parent labeled A and children labeled β appears in t. (This is equivalent to saying that the probability of a tree is the product of the probabilities of all the local trees that make it up.) If $\mathcal{T}_G(w)$ is the set of trees generated by G with yield w,

$$\begin{aligned}
\mathcal{V} &= \{\mathrm{x}\}\\
\mathcal{N} &= \{\mathrm{S}\}\\
S &= \mathrm{S}\\
\mathcal{R} &= \{\mathrm{S} \to \mathrm{S}\ \mathrm{S} \qquad \mathrm{S} \to \mathrm{x}\}\\
\boldsymbol{\rho} &= (\rho_{\mathrm{S}\to\mathrm{S}\,\mathrm{S}} = 0.7,\ \rho_{\mathrm{S}\to\mathrm{x}} = 0.3)
\end{aligned}$$

Figure 4.8: A PCFG for which the tree "probabilities" $\mathrm{P}(t)$ do not sum to 1. Informally, this is because this grammar puts non-zero mass on infinite trees.

then we define $\mathrm{P}_G(w)$ to be the sum of the probability of the trees in $\mathcal{T}_G(w)$, i.e.,

$$\mathrm{P}_G(W = w) \;=\; \sum_{t \in \mathcal{T}_G(w)} \mathrm{P}(T = t) \tag{4.2}$$

Finally, note that P_G may not define a properly normalized probability distribution on $\mathcal{T}$ or $\mathcal{W}$. Figure 4.8 presents a simple PCFG for which $\sum_{t \in \mathcal{T}} \mathrm{P}(t) < 1$; intuitively this is because this grammar puts its mass on "infinite trees". Fortunately it has been shown that such cases cannot occur for a very wide class of grammars, including all the ones we allude to in this book.

4.2.4 HMMs as a kind of PCFG

PCFGs are strictly more expressive than the HMMs we saw in the previous chapter. That is, for each HMM there is a PCFG that generates the same language (with the same probabilities) as the HMM. This parallelism goes further; the algorithms we presented for HMMs in the last chapter can be viewed as special cases of the algorithms for PCFGs that we present below.

Recall that an HMM H is defined in terms of a set of states $\mathcal{Y}$, a state-to-state transition matrix $\boldsymbol{\sigma}$, where $\sigma_{y,y'}$ is the probability of a transition from y to y', and a state-to-output matrix $\boldsymbol{\tau}$, where $\tau_{y,v}$ is the probability of emitting output v from state y. Given an HMM G, we can define a PCFG G_H that generates the same probabilistic language as H as follows. Let $\mathcal{Y}'$ be the set of HMM states except for the begin and end states '$\triangleright$' and '$\triangleleft$'. Then set

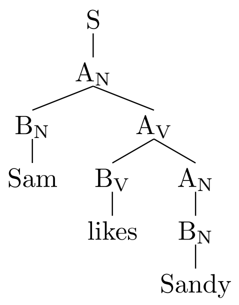

Figure 4.9: A parse tree generated by the PCFG corresponding to the HMM presented in the previous chapter.

$G_H = (\mathcal{V}, \mathcal{N}, S, \mathcal{R}, \rho)$ where $\mathcal{V}$ is the same terminal vocabulary as the HMM, $\mathcal{N} = \{S\} \cup \{A_y, B_y : y \in \mathcal{Y}'\}$ where A_y and B_y are unique symbols distinct from S and each element of $\mathcal{V}$. $\mathcal{R}$ consists of all rules of the form $S \to A_y$, $A_y \to B_y\ A_{y'}$, $A_y \to B_y$ and $B_y \to v$, for all $y, y' \in \mathcal{Y}'$ and $v \in \mathcal{V}$, and $\boldsymbol{\rho}$ is defined as follows:

$$\begin{array}{lcllcl} \rho_{S \to A_y} & = & \sigma_{\triangleright,y} & \rho_{A_y \to B_y\ A_{y'}} & = & \sigma_{y,y'} \\ \rho_{A_y \to B_y} & = & \sigma_{y,\triangleleft} & \rho_{B_y \to v} & = & \tau_{y,v} \end{array}$$

Example 4.4: Figure 4.9 shows a parse tree (writen in the form normal for a parse tree) generated from the PCFG corresponding to the example HMM presented in Section 3.3 on page 80. However, the corespondence to HMMs is perhaps clearer if we write the parse tree "on its side" as in Figure 4.10. This figure should also make the above probability rules clearer — e.g, why $\rho_{S \to A_y} = \sigma_{\triangleright,y}$.

Grammars such as these are called *right-linear* ("linear" means that every rule contains at most one nonterminal symbol, and "right-linear" means that that this nonterminal appears rightmost in the list of the rule's children). We've just seen that every HMM generates the same language as some right-linear PCFG, and it's possible to show that every right-linear PCFG generates the same language as some HMM, so HMMs and right-linear PCFGs can express the same class of languages.

$$\begin{tikzcd}
\mathrm{S} \arrow[r] & \mathrm{A_N} \arrow[r] \arrow[d] & \mathrm{A_V} \arrow[r] \arrow[d] & \mathrm{A_N} \arrow[d] \\
 & \mathrm{B_N} \arrow[d] & \mathrm{B_V} \arrow[d] & \mathrm{B_N} \arrow[d] \\
 & \text{Sam} & \text{likes} & \text{Sandy}
\end{tikzcd}$$

$$\begin{tikzcd}
\triangleleft \arrow[r] & \mathrm{N} \arrow[r] \arrow[d] & \mathrm{V} \arrow[r] \arrow[d] & \mathrm{N} \arrow[r] \arrow[d] & \triangleleft \\
 & \text{Sam} & \text{likes} & \text{Sandy} &
\end{tikzcd}$$

Figure 4.10: The same parse tree writen on its side, along with the Bayes net for the corresponding HMM

4.2.5 Binarization of PCFGs

Many of the dynamic programming algorithms for CFGs and PCFGs require that their rules be in a special form that permits efficient processing. We will say that a (P)CFG is *binarized* iff all of its productions are all instances of the following schemata:

$$\begin{array}{lcll}
A \to v & : & A \in \mathcal{N}, v \in \mathcal{V} & \text{(terminal rules)} \\
A \to B\ C & : & A, B, C \in \mathcal{N} & \text{(binary rules)} \\
A \to B & : & A, B \in \mathcal{N} & \text{(unary rules)}
\end{array}$$

Grammars in which all rules are either terminal rules or binary rules are said to be in *Chomsky normal form*. It turns out that for every PCFG G without epsilon rules there is another PCFG G' in Chomsky normal form that generates the same language as G.

Binarized PCFGs are less restrictive that Chomsky normal form because they also permit unary rules (in addition to terminal and binary rules). It turns out that every epsilon-free PCFG G has a corresponding binarized PCFG G' that generates the same language as G, and that the trees generated by G' correspond 1-to-1 with the trees of G. This means that we can map the trees generated by G' to trees generated by G. So in our algorithms below, given a grammar G we find an equivalent binarized grammar G' which we use in algorithm, and then map its output back to the trees of original grammar G.

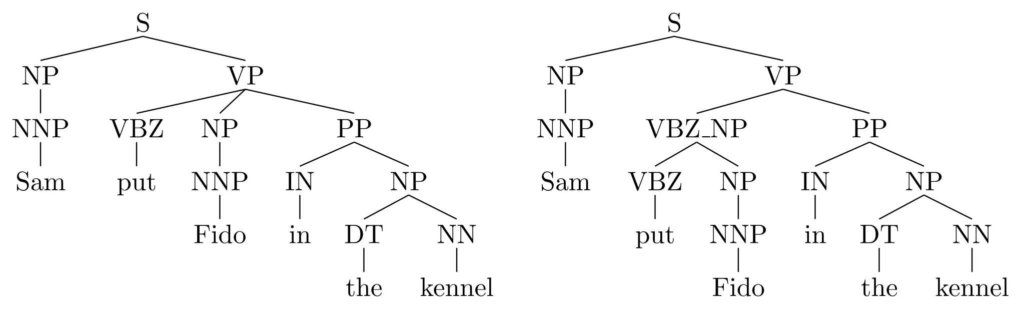

Figure 4.11: A tree generated by a non-binary CFG, and a tree generated by its binarized counterpart

There are many ways of converting an arbitrary PCFG G to an equivalent binarized PCFG G', but we only describe one of the simplest algorithms here. The key idea is to replace a rule with three or more symbols on the right with several binary rules that a ccomplish the same thing. So the rule '$A \to B\ C\ D$' would be replaced by

A → B_C D
B_C → C D

The new symbol '*B_C*' can only expand one way, so whenever we use the first of these rules, we always end up with the correct three symbols in our parse.

More formally the binarized PCFG $G' = (\mathcal{V}, \mathcal{N}', S, \mathcal{R}', \rho')$ has the same terminal vocabulary and start symbol as $G = (\mathcal{V}, \mathcal{N}, S, \mathcal{R}, \rho)$, but G' has many more nonterminal symbols and rules than G does, e.g., B_C. So the non-terminal set $\mathcal{N}'$ consist of $\mathcal{N}$ plus all *non-empty proper prefixes* of the rules of G. (A proper prefix of a string β is a prefix of β that does not include all of β). We'll write $\beta_{1:i}$ for a symbol concatingating $\beta_1, \ldots, \beta_i$ of β. This method is called *right-branching binarization* because all of the branching occurs along the right-hand-side spine.

Figure 4.11 depicts a tree generated by a non-binarized CFG as well as its binarized counterpart.

More precisely, the binarized grammar G' is defined as follows:

$$\mathcal{N}' = \mathcal{N} \cup \{\beta_{1:i} : A \to \beta \in \mathcal{R}, 1 < i < |\beta|\}$$

$$\mathcal{R}' = \left\{ \begin{array}{lcl} A \to v & : & A \to v \in \mathcal{R} \\ A \to B & : & A \to B \in \mathcal{R} \\ A \to \beta_{1:n-1}\ \beta_n & : & A \to \beta \in \mathcal{R}, |\beta| > 1 \\ \beta_{1:i} \to \beta_{1:i-1}\ \beta_i & : & A \to \beta \in \mathcal{R}, 1 < i < |\beta| \end{array} \right\}$$

$$\boldsymbol{\rho}' = \left\{ \begin{array}{lcll} \rho'_{A\to v} & = & \rho_{A\to v} & : \ A \to v \in \mathcal{R} \\ \rho'_{A\to B} & = & \rho_{A\to B} & : \ A \to B \in \mathcal{R} \\ \rho'_{A\to\beta_{1:n-1}\ \beta_n} & = & \rho_{A\to\beta} & : \ A \to \beta \in \mathcal{R} \\ \rho'_{\beta_{1:i}\to\beta_{1:i-1}\ \beta_i} & = & 1 & : \ A \to \beta \in \mathcal{R}, 1 < i < |\beta| \end{array} \right\}$$

Unary and binary rules probabilities go through untouched. Interestingly, larger rules retain their exactly probabilities as well. So the probability for, say, NP $\to$ DT_JJ_JJ NN ("the large yellow block") is unchanged from that of the original rule (NP $\to$ DT JJ JJ NN). The probability of the extra rules introduced by the process, e.g., (DT_JJ_JJ $\to$ DT_JJ JJ) are all one. This makes sense since there is only one thing that DT_JJ_JJ can expand into, and thus the probability that it will expand that way is one. This means that the sequence of rule expansions that are required in the binarization case have the same probability as the single probability in the unbinarized grammar. Also note that different $n$ary rules may share the helper binarized grammar rules. e.g., the rule (NP $\to$ DT JJ JJ NNS) (NNS is plural common noun) has the binarization (NP $\to$ DT_JJ_JJ NN), and make use of the rules for expanding DT_JJ_JJ.

Lastly, two point that we come back to in Section 4.7. First, as should already be obvious, binarization dramatically increases the size of the non-terminal vocabulary. Secondly, for the particular binarization method we choose, a new binary non-terminals may never appear as the second non-terminal in a binary rule, only the first. This has implications for efficient parsing.

4.3 Parsing with PCFGs

This section describes how to parse a string $w \in \mathcal{W}$ with a PCFG G. In practice most probabilistic grammars are designed to generate most or all strings in $\mathcal{W}$, so it's not interesting to ask whether w is generated by G.

Similarly, the set $\mathcal{T}_G(w)$ will typically include a very large number of trees, many of which will have extremely low probability. Instead, we typically want to find the most likely tree $\hat{t}(w)$ for a string w, or perhaps the set of the m most-likely trees (i.e., the m trees in $\mathcal{T}_G(w)$ with maximum probability).

$$\begin{aligned}\hat{t}(w) &= \operatorname*{argmax}_{t \in \mathcal{T}_G(w)} \mathrm{P}_G(T = t \mid W = w) \\ &= \operatorname*{argmax}_{t \in \mathcal{T}_G(w)} \mathrm{P}_G(T = t, W = w)\end{aligned}$$

(However, if we intend to use G as a *language model*, i.e., to predict the probability of a string w, we want to implicitly sum over all $\mathcal{T}_G(w)$ to calculate $\mathrm{P}_G(w)$, as in equation 4.2).

If w is very short and G is sufficiently simple we might be able to enumerate $\mathcal{T}_G(w)$ and identify $\hat{t}(w)$ exhaustively. However, because $\mathcal{T}_G(w)$ can be infinite (and even when it is finite, its size can be exponential in the length of w), in general this is not a practical method.

Instead, we show how to find $\hat{t}(w)$ using a dynamic programming algorithm that directly generalizes the algorithm used to find the most likely state sequence in an HMM given in Section 3.3 on page 80. This algorithm requires that G be a binarized PCFG. We described in the previous section how to map an arbitrary PCFG to an equivalent binarized one, so from here on we will assume that G is binarized. (Note that there are efficient dynamic programming parsing algorithms that do not require G to be binarized, but these usually perform an implicit binarization "on the fly").

It actually simplifies CFG parsing algorithms to use "computer science" zero-based indexing for strings. That is, a string $\boldsymbol{w}$ of length n is made up of elements indexed from 0 to $n-1$, i.e., $\boldsymbol{w} = (w_0, w_1, \ldots, w_{n-1})$. A subsequence or "slice" is defined to be $\boldsymbol{w}_{i,j} = (w_i, \ldots, w_{j-1})$, i.e., it goes up to, but doesn't include element w_j. This in turn is most easily visualized if you think of the indicies as enumerating the positions *between* the words.

Example 4.5: The sentence "Sam likes Sandy" with the indicies 0 to 3 would look like this:

```
      Sam     likes     Sandy
   0        1         2          3
```

So, e.g., $w_{1,3}$ = "likes Sandy".

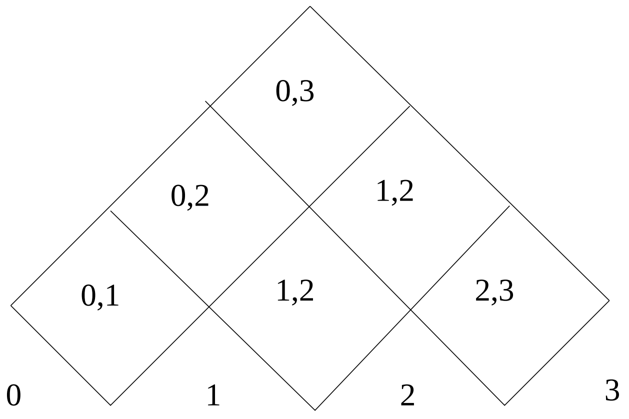

Figure 4.12: Chart for a string of length three

Just as in the HMM case, we first address the problem of finding the probability $\mu(\boldsymbol{w}) = \max_{t \in \mathcal{T}(w)} \mathrm{P}(t)$ of the most likely parse $\hat{t}(\boldsymbol{w})$ of $\boldsymbol{w}$. Then we describe a method for reading out the most likely tree $\hat{t}(\boldsymbol{w})$ from the dynamic programming parsing table used to compute $\mu(\boldsymbol{w})$.

For a given nonterminal $A \in \mathcal{N}$, let $\mu_A(i,k)$ be the probability of the most likely parse of $\boldsymbol{w_{i,k}}$ when the subtree is rooted in A. Given a string $\boldsymbol{w}$ to parse, we recursively compute $\mu_A(i,k)$ for all for each $A \in \mathcal{N}$ and $0 \leq i < j \leq |\boldsymbol{w}|$. (We use the positions i to k because in a moment we introduce a position j, $i < j < k$.)

The traditional visualization for this is called a *chart*. Charts are two dimensional arrays of (i,k) Figure 4.12 shows an empty chart for a three word string, e.g., “Sam likes Sandy”. Each diamond is a *cell* of the chart, and coresponds to possible span of words. So the bottom left-hand cell will be filled with terminal and non-terminals that span the words zero to one — e.g., the first word of the sentence “Sandy”. In the same way, the top center cell spans the words zero to three. It is always the top center that contains the start symbol if the string is in the language of the PCFG we are using. Note that the instructions to fill the chart from bottom up coresponds to the pseudo-code:

```
1. for $l = 1$ to $L$
    (a) for $s = 0$ to $L - l$
        i. fill cell$(s, s + l)$
```

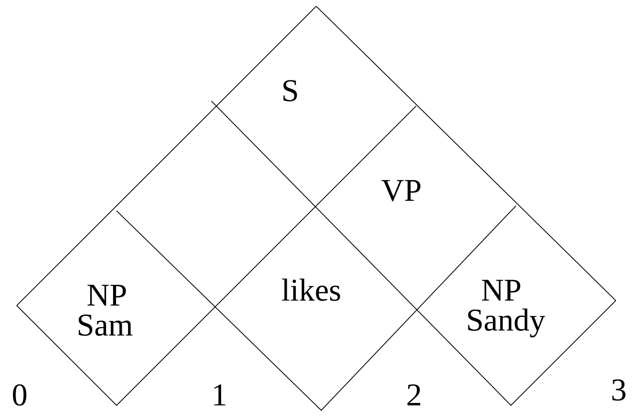

Figure 4.13: Chart appropriately filled for "Sam likes Sandy"

Filling a cell coresponds to finding the possible terminal and non-terminals that can span each terminal string (e.g., the third dimension) and determining their $\mu_A(i,k)$. We want to fill bottom up because filling a cell requires the entries for all of the cells beneath it. Figure 4.13 shows our chart when the grammar of Figure 4.5 Note that charts are what we earlier refered to as *packed parse forests* in that they compress an exponential number of possible trees into a data structure of size N^2, where N is the length of the sentence.

We begin with the case where the grammar is in Chomsky normal form (CNF), i.e., all rules are terminal or binary, and consider unary rules later. Our algorithm computes the values of $\mu_A(i,k)$ from smaller substrings to larger ones. Consider the substrings of $\boldsymbol{w}$ of length 1, i.e., where $k = i+1$. If the grammar is in CNF then these can only be generated by a terminal rule, as the yield of any tree containing a binary branch must contain at least two words. Therefore for $i = 0, \ldots, |\boldsymbol{w}| - 1$:

$$\mu_A(i, i+1) \quad = \quad \rho_{A \to w_i} \tag{4.3}$$

Now we turn to the substrings $\boldsymbol{w}_{i,k}$ of length greater than 1, so $k - i > 1$. All of their parse trees must start with a binary branching node, which must be generated by some rule $(A \to B\ C) \in \mathcal{R}$. This means that there must be a subtree with root labeled B and yield $\boldsymbol{w}_{i,j}$ and another subtree with root labeled C and yield $\boldsymbol{w}_{j,k}$. This yields the following equation for $\boldsymbol{\mu}$ when

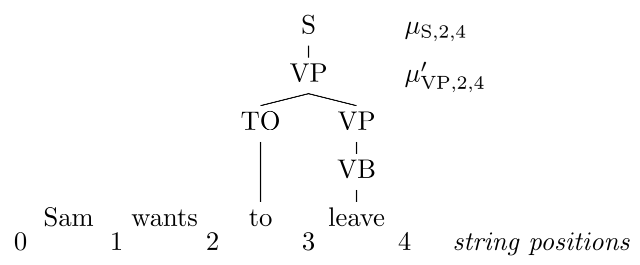

Figure 4.14: This figure shows the relationship between $\boldsymbol{\mu}'$ (which only considers trees starting with a terminal or binary rule) and $\boldsymbol{\mu}$ (which considers all trees).

$k - i > 1$,

$$\mu_A(i,k) \quad = \quad \max_{\substack{j\,:\,i<j<k \\ A\to B\ C\in\mathcal{R}_A}} \quad \rho_{A\to B\ C}\ \mu_B(i,j)\ \mu_C(j,k) \tag{4.4}$$

In words, we look for the combination of rule and mid-point which gives us the maximum probability for (A) spanning i, k. These probabilities are simply the probabilities of the two sub-components times the probability of the rule that joints them to form an A.

Equations 4.3 and 4.4 can be used bottom-up (working from shorter sub-strings to longer) to fill in the table $\boldsymbol{\mu}$. After the table is complete, the probability of the most likely parse is $\mu_S(0, |w|)$. Finding the most likely parse is straightforward if you associate each entry $\mu_A(i,k)$ with "backpointers" to the $\mu_B(i,j)$ and $\mu_C(j,k)$ that maximize (4.4).

Unfortunately unary rules complicate this simple picture somewhat. If we permit unary rules the subtree responsible for maximizing $\mu_A(i,k)$ can consist of a chain of up to $|\mathcal{N}|-1$ unary branching nodes before the binary or terminal rule is applied. We handle this by introducing $\mu'_A(i,k)$, which is the probability of the most likely parse rooted of $\boldsymbol{w}_{i,k}$ whose root is labeled A and expands with a binary or terminal rule. Figure 4.14 shows the relationship between $\boldsymbol{\mu}$ and $\boldsymbol{\mu}'$. Then:

$$\mu'_A(i,i+1) = \rho_{A\to w_i} \tag{4.5}$$

$$\mu'_A(i,k) = \max_{\substack{j\,:\,i<j<k\\ A\to B\,C\in\mathcal{R}_A}} \rho_{A\to B\;C}\;\mu_B(i,j)\;\mu_C(j,k) \tag{4.6}$$

$$\mu_A(i,k) = \max\left(\mu'_A(i,k),\, \max_{A\to B\in\mathcal{R}_A} \rho_{A\to B}\;\mu_B(i,k)\right) \tag{4.7}$$

Again, the computation needs to be arranged to fill in shorter substrings first. For each A, i and j combination, (4.5) or (4.6) only needs to be applied once. But unless we know that the grammar constrains the order of nonterminals in unary chains, (4.7) may have to be repeatedly applied up to $|\mathcal{N}|-1$ times for each i,j combination, sweeping over all unary rules, because each application adds one additional story to the unary chain.

There are some obvious optimizations here. For example, no nonterminal produced by binarization can ever appear in a unary chain, so they can be ignored during the construction of unary chains. If there are no unary chains of height ℓ for a particular i,j then there are no chains of height $\ell+1$, so there's no point looking for them. One can optimize still further: there's no point in considering a unary rule $A\to B$ to build a chain of height $\ell+1$ unless there is a unary chain with root B of height ℓ. And there's no need to actually maintain two separate tables $\boldsymbol{\mu}'$ and $\boldsymbol{\mu}$: one can first of all compute $\mu'_A(i,k)$ using (4.5) or (4.6), and then update those values to $\mu_A(i,k)$ using (4.7).

On the other hand, it possible that you already have, say, a constituent Z built from a chain of length two, but then find a better way to build it with a chain of height three. Even if this is the only thing added at three, you have to go on to four, because there could be some other constituent that will now use the more proable Z to improve its probability. To put it another way, you keep going until no non-terminal increases its μ.

4.4 Estimating PCFGs

This section describes methods for estimating PCFGs from data. We consider two basic problems here. First, we consider the supervised case where the training data consists of parse trees. Then we consider the unsupervised case where we are given a CFG (i.e., we are told the rules but not their probabilities) and have to estimate the rule probabilities from a set of strings generated by the CFG. (Learning the rules themselves, rather than just their probabilities, from strings alone is still very much an open research problem).

4.4.1 Estimating PCFGs from parse trees

This section considers the problem: given a sequence $\boldsymbol{t} = (t_1, \ldots, t_n)$ of parse trees, estimate the PCFG $\hat{G}$ that might have produced $\boldsymbol{t}$. The nonterminals $\mathcal{N}$, terminals $\mathcal{V}$, start symbol S and set of rules $\mathcal{R}$ used to generate $\boldsymbol{t}$ can be read directly off the local trees of $\boldsymbol{t}$, so all that remains is to estimate the rule probabilities $\boldsymbol{\rho}$. We'll use the Maximum Likelihood principle to estimate these. This supervised estimation problem is quite straight-forward because the data is fully observed, but we go through it explicitly here because it serves as the basis of the Expectation-Maximization unsupervised algorithm discussed in the next section.

It's straightforward to show that the likelihood of $\boldsymbol{t}$ is:

$$\begin{aligned} L_{\boldsymbol{t}}(\boldsymbol{\rho}) &= \mathrm{P}_{\boldsymbol{\rho}}(\boldsymbol{t}) \\ &= \prod_{A \rightarrow \beta \in \mathcal{R}} \rho_{A \rightarrow \beta}^{n_{A \rightarrow \beta}(\boldsymbol{t})} \end{aligned} \tag{4.8}$$

where $n_{A \rightarrow \beta}(\boldsymbol{t})$ is the number of times that the local tree $A \rightarrow \beta$ appears in the sequence of trees $\boldsymbol{t}$.

Since $\boldsymbol{\rho}$ satisfies the normalization constraint (4.1) on page 117, (4.8) is a product of multinomials, one for each nonterminal $A \in \mathcal{N}$. Using fairly simple analysis one can show that the maximum likelihood estimate is:

$$\hat{\rho}_{A \rightarrow \beta} = \frac{n_{A \rightarrow \beta}(\boldsymbol{t})}{n_A(\boldsymbol{t})} \tag{4.9}$$

where $n_A(\boldsymbol{t}) = \sum_{A \rightarrow \beta \in \mathcal{R}_A} n_{A \rightarrow \beta}(\boldsymbol{t})$ is the number of nodes labeled A in $\boldsymbol{t}$.

4.4.2 Estimating PCFGs from strings

We now turn to the much harder problem of estimating the rule probabilities $\boldsymbol{\rho}$ from strings, rather than parse trees. Now we face an estimation problem with hidden variables. As we have done with other hidden variable estimation problems, we will tackle this one using Expectation-Maximization (EM).

The basic idea is that given a training corpus $\boldsymbol{w}$ of strings and an initial estimate $\boldsymbol{\rho}^{(0)}$ of the rule probabilities, at least conceptually we use $\boldsymbol{\rho}^{(0)}$ to compute the distribution $\mathrm{P}(\boldsymbol{t}|\boldsymbol{w})$ over possible parses for $\boldsymbol{w}$, and from that distribution compute the expected value $\mathrm{E}[n_{A\to\beta}|\boldsymbol{w}]$ of the statistics needed in the MLE equation (4.9) to produce an improved estimate $\boldsymbol{\rho}$. We iterate this process, ultimately converging on at least a local maximum of the likelihood.

The key quantity we need to compute in the EM algorithm are the expected rule counts:

$$\begin{aligned}\mathrm{E}[n_{A\to\beta} \mid \boldsymbol{w}] &= \sum_{\boldsymbol{t}\in\mathcal{T}_G(\boldsymbol{w})} n_{A\to\beta}(\boldsymbol{t})\ \mathrm{P}(\boldsymbol{t} \mid \boldsymbol{w}) \\ &= \frac{1}{\mathrm{P}(w)} \sum_{\boldsymbol{t}\in\mathcal{T}_G(\boldsymbol{w})} n_{A\to\beta}(\boldsymbol{t})\ \mathrm{P}(\boldsymbol{t})\end{aligned}$$

If the sentences in $\boldsymbol{w}$ are all very short it may be practical to enumerate all of their parses and compute these expectations this way. But in general this will not be possible, and we will need to use a more efficient algorithm. The rest of this section describes the *Forward-Backward algorithm*, which is a dynamic programming algorithm for computing these expectations for binarized grammars.

To keep matters simple, we will focus on computing the expected counts from a single string $\boldsymbol{w} = (w_0, \ldots, w_{n-1})$. In practice we would compute the expected counts for each sentence separately, and sum them to get the expectations for the corpus as a whole.

4.4.3 The inside-outside algorithm for CNF PCFGs

Just as we did for parsing, we'll first describe the algorithm for grammars in Chomsky Normal Form (i.e., without unary rules), and then describe the extension required to handle unary rules.

Generalizing what we did for HMMs, we first describe how to compute something more specific than the expectations we require, namely the expected number $\mathrm{E}[n_{A\to BC}(i,j,k)|\boldsymbol{w}]$ of times a rule $A \to B\ C$ to expand an A spanning from i to k into a B spanning from i to j and a C spanning from j to k. The expectations we require are obtained by summing over all string positions i, j and k.

The quantities we need to compute are the *inside* and *outside* "probabilities" $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$ respectively. These generalize the Backward and the Forward

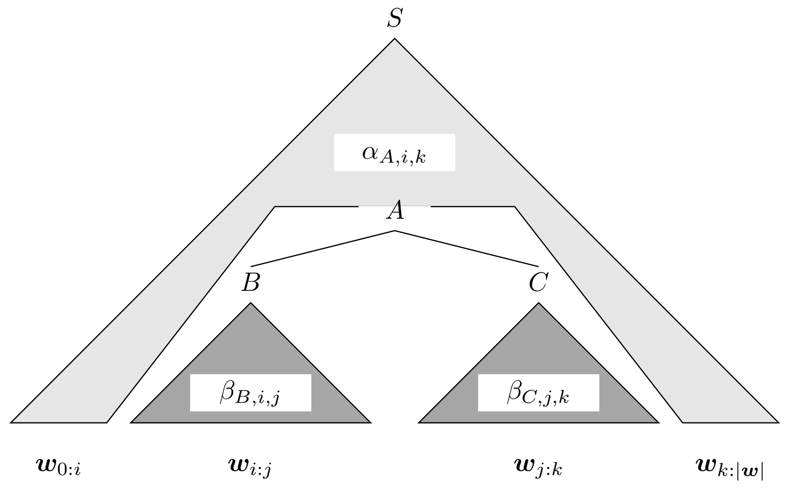

Figure 4.15: The parts of the tree contributing to the inside probabilities $\beta_A(i,j)$ and the outside scores $\alpha_A(i,j)$, as used to calculate $\mathrm{E}[n_{A\to B\,C}(i,j,k)]$.

probabilities used in the EM algorithm for HMMs, which is why we use the same symbols for them. The scare quotes are there because in grammars with unary rules the outside "probabilities" can be larger than one, and because of this we will refer to outside *scores* instead of "probabilities". Figure 4.15 depicts the parts of a tree that contribute to the inside probabilities and outside scores.

The inside probability $\beta_A(i,j)$ is the probability of an A expanding to $\boldsymbol{w}_{i,j}$, which is the sum of the probability of all trees with root labeled A and yield $\boldsymbol{w}_{i,j}$, i.e.,

$$\begin{aligned}\beta_A(i,j) &= \mathrm{P}_A(\boldsymbol{w}_{i:j}) \\ &= \sum_{t\in\mathcal{T}_A(\boldsymbol{w}_{i:j})} \mathrm{P}(t)\end{aligned}$$

The outside socre $\alpha_A(i,j)$ is somewhat harder to understand. It is the sum of the probability of all trees whose yield is $\boldsymbol{w}_{0,i}A\boldsymbol{w}_{j:|\boldsymbol{w}|}$, i.e., the string $\boldsymbol{w}$ with $\boldsymbol{w}_{i,j}$ replaced by A. It's called the outside probability because it counts only the part of the tree outside of A. Intuitively, it is the sum of the probability of all trees generating $\boldsymbol{w}$ that include an A expanding to $\boldsymbol{w}_{i,j}$, not including the subtree dominated by A.

The reason why the outside scores can be greater than one with unary rules is that the tree fragments being summed over aren't disjoint, i.e., a tree fragment and its extension wind up being counted. Consider the grammar with rules $\mathrm{S} \rightarrow \mathrm{S}$ and $\mathrm{S} \rightarrow x$, with $\rho_{\mathrm{S}\rightarrow\mathrm{S}} = \rho_{\mathrm{S}\rightarrow x} = 0.5$. The outside trees whose probability is summed to compute $\alpha_{\mathrm{S}}(0,1)$ consist of unary chains whose nodes are all labeled S, so $\alpha_{\mathrm{S}}(0,1) = 1 + 1/2 + 1/4 + \ldots = 2$. On the other hand, $\beta_{\mathrm{S}}(0,1)$ involves summing the probability of similar unary chains which terminate in an x, so $\beta_{\mathrm{S}}(0,1) = 1/2 + 1/4 + \ldots = 1$ as expected (since 'x' is the only string this grammar generates). For this section, however we are assuming Chomsky Normal Form grammars, which do not have unay rules. This simplifies things. In particular the outside scores are now probabilities.

Since the grammar is in Chomsky Normal Form, $\boldsymbol{\beta}$ satisfies the following:

$$\begin{aligned}
\beta_A(i,i+1) &= \rho_{A\rightarrow w_i} \\
\beta_A(i,k) &= \sum_{\substack{A\rightarrow B\ C \in \mathcal{R}_A \\ j\,:\, i<j<k}} \rho_{A\rightarrow B\ C}\ \beta_B(i,j)\ \beta_C(j,k) \text{ if } k-i>1
\end{aligned}$$

These can be used to compute $\boldsymbol{\beta}$ in a bottom-up fashion by iterating from smaller to larger substrings, just like we did for $\boldsymbol{\mu}$ in section 4.3 on page 122. It's easy to see that $\beta_S(0,|\boldsymbol{w}|) = \mathrm{P}(\boldsymbol{w})$, i.e., the probability of the string $\boldsymbol{w}$.

The outside probability $\boldsymbol{\alpha}$ are somewhat more complex.

$$\begin{aligned}
\alpha_A(0,|\boldsymbol{w}|) &= 1 \text{ if } A = \mathrm{S} \text{ and } 0 \text{ otherwise} && (4.10) \\
\alpha_C(j,k) &= \sum_{\substack{A\rightarrow B\, C \in \mathcal{R} \\ i\,:\, 0\leq i<j}} \rho_{A\rightarrow B\ C}\ \alpha_A(i,k)\ \beta_B(i,j) \\
&\quad + \sum_{\substack{A\rightarrow C\ D \in \mathcal{R} \\ \ell\,:\, k<\ell\leq|\boldsymbol{w}|}} \rho_{A\rightarrow C\ D}\ \alpha_A(j,\ell)\ \beta_D(k,\ell) && (4.11)
\end{aligned}$$

In order to understand these equations, it helps to recognize that if t is a tree generated by a grammar in Chomsky Normal Form then every nonterminal node in t is either the root node, and therefore has its α specified by (4.10), or is either a right child or a left child of some other nonterminal, and (4.11) sums over these alternatives. In more detail, (4.11) says that

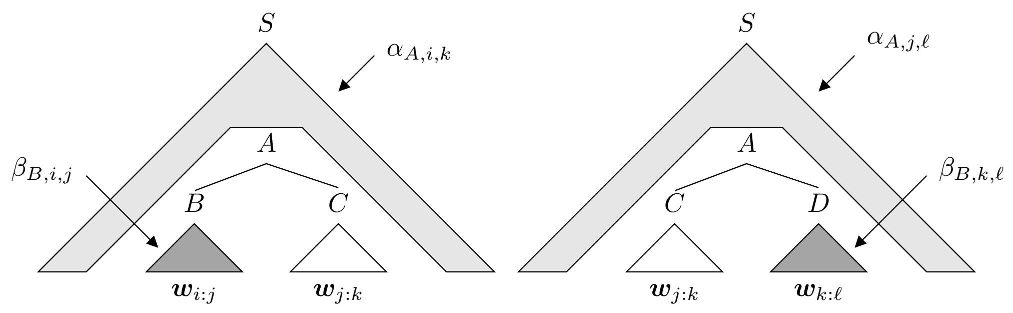

Figure 4.16: The two cases corresponding to the terms in the sum in (4.11) on page 131. Note that the insides of siblings B and D are outside C but inside A, so their inside probabilities are multiplied by the outside scores of A and the rule that connects them.

the outside score for the smaller constituent C consists of the sum of the outside scores for the larger constituent A times the inside probability of its sibling (either B or D) times the probability of the rule that connects them. Figure 4.16 depicts the structures concerned in (4.11).

These two equations can be used to compute $\boldsymbol{\alpha}$, iterating from longer strings to shorter. The first equation initializes the procedure by setting the outside probability for the root S node to 1 (all other labels have outside probability 0).

Once we have computed $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, the expected counts are given by:

$$\mathrm{E}[n_{A \to w_i}(i, i+1) \mid \boldsymbol{w}] = \frac{\alpha_{A,i,i+1}\ \rho_{A \to w_i}}{\mathrm{P}(\boldsymbol{w})} \tag{4.12}$$

$$\mathrm{E}[n_{A \to B\ C}(i, j, k) \mid \boldsymbol{w}] = \frac{\alpha_{A,i,k}\ \rho_{A \to B\ C}\ \beta_{B,i,j}\ \beta_{C,j,k}}{\mathrm{P}(\boldsymbol{w})} \tag{4.13}$$

The expected rule counts that we need to reestimate $\boldsymbol{\rho}$ are obtained by summing over all combinations of string positions, i.e.,

$$\begin{aligned}
\mathrm{E}[n_{A \to w} \mid \boldsymbol{w}] &= \sum_{i=0}^{|\boldsymbol{w}|-1} \mathrm{E}[n_{A \to w}(i, i+1) \mid \boldsymbol{w}] \\
\mathrm{E}[n_{A \to B\ C} \mid \boldsymbol{w}] &= \sum_{\substack{i,j,k: \\ 0 \le i < j < k \le |\boldsymbol{w}|}} \mathrm{E}[n_{A \to B\ C}(i, j, k) \mid \boldsymbol{w}]
\end{aligned}$$

4.4.4 The inside-outside algorithm for binarized grammars

This section sketches how the inside-outside algorithm can be extended to binarized grammars with unary rules. The key to this is calculating the scores (i.e., the product of rule probabilities) of all unary chains with root labeled A and root labeled B for all $A, B \in \mathcal{N}$.

In the computation of the most likely parse $\hat{t}(\boldsymbol{w})$ in section 4.3 on page 122 we explicitly enumerated the relevant unary chains, and were certain that the process would terminate because we could provide an upper bound on their length. But here we want to sum over all unary chains, including the low probability ones, and if the grammar permits *unary cycles* (i.e., nontrivial unary chains whose root and leaf have the same label) then there are infinitely many such chains.

Perhaps surprisingly, it turns out that there is a fairly simple way to compute the sum of the probability of all possible unary chains using matrix inversion. We start by defining a matrix $\mathbf{U} = \mathbf{U}_{A,B}$ whose indices are nonterminals $A, B \in \mathcal{N}$ of the binarized grammar G. (In order to use standard matrix software you'll need to map these to natural numbers, of course).

We set $\mathbf{U}_{A,B} = \rho_{A \to B}$ to be the probability of a unary chain of length one starting with A and ending with B. Then the probability of chains of length two is given by $\mathbf{U}^2$ (i.e., the probability of a chain of length two starting with an A and ending with a B is $\mathbf{U}^2_{A,B}$), and in general the probability of chains of length k is given by $\mathbf{U}^k$. Then the sum $\mathbf{S}$ of the probabilities of all such chains is given by:

$$\begin{aligned} \mathbf{S} &= \mathbf{1} + \mathbf{U} + \mathbf{U}^2 + \ldots \\ &= (\mathbf{1} - \mathbf{U})^{-1} \end{aligned}$$

where $\mathbf{1}$ is the identity matrix (corresponding to unary chains of length zero) and the superscript '-1' denotes matrix inversion.

With the matrix $\mathbf{S}$ in hand, we now proceed to calculate the inside probabilities $\boldsymbol{\beta}$ and outside scores $\boldsymbol{\alpha}$. Just as we did in section 4.3 on page 122, we introduce auxiliary matrices $\boldsymbol{\beta}'$ and $\boldsymbol{\alpha}'$ that only sum over trees whose root expands with a terminal or binary rule, and then compute their unary closure.

The equations for the inside probabilites are as follows:

$$
\begin{aligned}
\beta'_A(i,i+1) &= \rho_{A\to w_i} \\
\beta'_A(i,k) &= \sum_{\substack{A\to B\ C\in\mathcal{R}_A\\ j\,:\,i<j<k}} \rho_{A\to B\ C}\ \beta_B(i,j)\ \beta_C(j,k) \text{ if } k-i>1 \\
\beta_A(i,j) &= \sum_{B\in\mathcal{N}} \mathbf{S}_{A,B}\ \beta'_B(i,j)
\end{aligned}
$$

Note that the equation for $\boldsymbol{\beta}$ is *not* recursive, so it only needs to be computed once for each combination of A, i and j (unlike the corresponding unary closure equation for $\boldsymbol{\mu}$).

The equations for the outside scores are also fairly simple extensions of those for Chomsky Normal Form grammars, except that the unary chains grow *downward*, of course.

$$
\begin{aligned}
\alpha'_A(0,|\boldsymbol{w}|) &= 1 \text{ if } A=\mathrm{S} \text{ and } 0 \text{ otherwise} \\
\alpha'_C(j,k) &= \sum_{\substack{A\to B\,C\in\mathcal{R}\\ i\,:\,0\le i<j}} \rho_{A\to B\ C}\ \alpha_A(i,k)\ \beta_B(i,j) \\
&\quad + \sum_{\substack{A\to C\ D\in\mathcal{R}\\ \ell\,:\,k<\ell\le|\boldsymbol{w}|}} \rho_{A\to C\ D}\ \alpha_A(j,\ell)\ \beta_D(k,\ell) \\
\alpha_B(i,j) &= \sum_{A\in\mathcal{N}} \mathbf{S}_{A,B}\ \alpha'_A(i,j)
\end{aligned}
$$

Interestingly, the equations (4.12–4.13) we gave on page 132 are correct for binarized grammars as well, so all we need is the corresponding equation for unary rules.

$$
\mathrm{E}[n_{A\to B}(i,j) \mid \boldsymbol{w}] = \frac{\alpha_A(i,j)\ \rho_{A\to B}\ \beta_B(i,j)}{\mathrm{P}(\boldsymbol{w})} \tag{4.14}
$$

4.5 Scoring Parsers

The point of a parser is to produce "correct" trees. In this section we describe how the field measures this, and ultimately assigns a single number, say, 0.9 (out of 1.0) to grade a parsers performance. Rather than do this in the

```
(ROOT
 (SQ (MD Would)
   (NP (NNS participants))
   (VP (VP (VB work) (ADVP (JJ nearby)))
       (CC or)
       (VP (VP (VB live) (PP (IN in) (NP (NN barracks))))
           (CC and)
           (VP (VB work))
             (PP (IN on) (NP (JJ public) (NNS lands)))))
    (. ?)))
```

Figure 4.17: A correct tree as specified by the tree bank

abstract, however, it would be more interesting to do this for a particular parser, say one built directly using the technology we described above.

To start we first need to introduce the *Penn treebank*, a corpus of about 40,000 sentences (one million words) from the Wall-Street Journal, each assigned a phrase structure by hand. Figure 4.17 is a (slightly shortened) tree from this corpus. You might notice the start symbol is not S, but *ROOT* because many strings in the tree bank are not complete sentences. Also notice that punctuation is included in the tree. It is a good idea to parsing with punctuation left in, as it gives good clues to the correct parse structure, and parsers uniformly work better with the punctuation than without it.

After binarizing the trees we can find Maximum Likelyhood rule probabilities as described in Section 4.4.1. We then build a basic parser as layed out above, and parse a test set we separated in advance. The question we now answer is, having done this, what is the parser's "score" and what does it mean?

The standard numerical measure of parser performance is something called *labeled precision/recall f-measure.* It is a number between zero and 1, with 1 being perfect. We now unpack this name.

First we find the total number of correct phrasal constituents in all of the test set. So in Figure 4.17 there are 13 phrasal labels (and fourteen pretermals). We do not count ROOT, because it is impossible to get wrong, so we have 12. A test-set of a thousand sentences might have a total of twenty five thousand such nodes. We then count how many the parser got correct. If it produced exactly the above tree, it would be twelve, but in fact,

```
(ROOT
 (SQ (MD Would)
   (NP (NNS participants))
   (VP (VB work)
       (ADJP (JJ nearby) (CC or) (JJ live))
        (PP (IN in)
            (NP (NP (NN barracks) (CC and) (NN work))
                (PP (IN on) (NP (JJ public) (NNS lands))))))
   (. ?)))
```

Figure 4.18: An incorrect tree for the example in Figure 4.17

it produces the tree in Figure 4.18 How many nodes are correct there?

To assign a number, we first need to define what makes a constituent in one tree the "same" as one in another tree with the same yield. Our formal definition is that two trees are the same if and only if they have the same label, starting point in the string, and ending point in the string. Note that according to this definition we cound as correct the VP(2,13) found in Figures 4.17 and 4.18 even though they do not have the same sub-constituents. The idea is that in such a case the sub-constituents will be marked wrong, but we do now allow the mistakes at that level to count against the VP one level up. The required label identity of the two constituents is what makes this "*labeled* precision/recall f-measure" So here there are four correct constituents in Figure 4.18 out of eight total.

So let C be the total constituents correct, G be the number in the treebank tree, and H the number in the parser tree. We define *precision* (P) as C/H and *recall* (R) is C/G. F-measure(F) is the harmonic mean of the two $(2{\cdot}P{\cdot}R)/(P+R)$. Note that if $P = R$ then $P = R = F$, or equalently F is the average of P and R. But as P and R diverge, F tends to the smaller of the two. So suppose our parser produces a tree with only one phrasal category, and it is correct, then $P = 1$, $R = 1/13$, and $F \approx 0.16$

When you build the just-described parser and test it we get $C = 18151$,$G = 27007$, and $H = 25213$, so $F = .695$ This means that almost one out of three constituents are incorrect. The example of Figure 4.18 was deliberatly picked to show you how bad things can get. The parser decided to use "work" as a noun, and "live" as an adjective, and these make for unusual combinations, such as one saying, in effect, that the participants are working "nearby or live" presumably as opposed to far-away or dead.

Fortunately, this is a very simple model, and statistical parsing has moved way beyond this. A modern parser gets about .90 F, and in particular gets the above example completely correct.

4.6 Estimating better grammars from treebanks

Context-free grammars are called "context-free" because the choice of rule expanding each non-terminal is independent of everything already generated in the tree except for the label on the parent being expanded. You can think of this label as classifying the rest of the tree into one of $|\mathcal{N}|$ states. Or to put it another way, a context-free symbol is an information barrior between what is going on outside the constituent and its inside. Because a CFG only constrains the labels on nodes in local trees, if we want to capture a a longer-range dependency between nonlocal parts of the tree, it must be broken down into a sequence of local dependencies between between the labels on nodes in the same local trees. The nonterminal labels then serve as "communication channels" responsible for propagating the dependency through the appropriate sequence of local trees. This process would start with a single state such as NP, and end up with several variants, NP1, NP2, etc. Thus the process is called *state-splitting.*

The classic example of state-splitting is *parent annotation.* where the parent category of each nonterminal is encoded in its label. So, if the string is a complete sentence the parse starts out with `(ROOT (S ...))` But treebank sentences under ROOT (almost) always end in a final punctuation mark, and thus these S's are quite different from, e.g., subordinate clases. We can better model this is we have the symbol "S$\hat{\mathrm{R}}$OOT" — an S underneath ROOT. In fact there are a lot of situations where parent annotation allows us to capture important regularities. For example, subject noun phrases, e.g., NP$\hat{\mathrm{S}}$, are more likely to be pronouns than object NPs (NP$\hat{\mathrm{V}}$P). Figure 4.19 also shows the result of parent-annotating the S and NP nonterminals.

Another important way to use state splitting is suggested by the fact that many words prefer to appear in phrases expanded with particular rules. For example, '*put*' prefers to appear under a VP that also contains an NP and PP (as in the example in Figure 4.11 on page 121), while '*sleep*' prefers

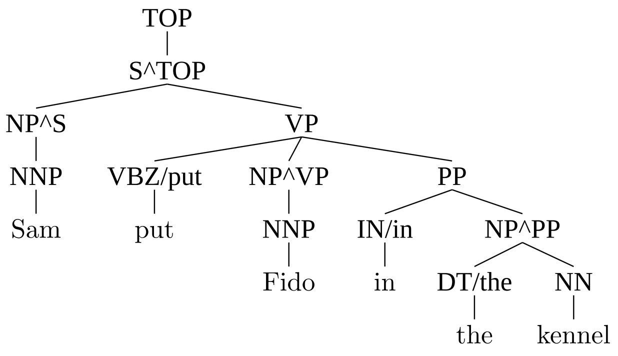

Figure 4.19: The tree produced by a state-splitting tree-transformation of the one depicted in Figure 4.11 on page 121. All verb, preposition and determiner preterminals are lexicalized, and S and NP nonterminals are parent-annotated. A new start symbol TOP has been introduced because not all trees in the treebank have a root labeled S.

to appear without any following NP (i.e., '*Sam slept'* is a much better sentence than '*Sam slept Fido in the kennel'*, but '*Sam put Fido in the kennel'* is much better than '*Sam put'*). However, in PCFGs extracted from the Penn Treebank and similar treebanks, there is a preterminal intervening between the word and the phrase (VBZ), so the grammar cannot capture this dependency. However, we can modify the grammar so that it does by splitting the preterminals so that they encode the terminal word. This is sometimes called "lexicalization". (In order to avoid sparse data problems in estimation you might choose to do this only for a small number of high-frequency words). For grammars estimated from a treebank training corpus, one way to do this is to transform the trees from which the grammar is estimated. Figure 4.19 shows this as well.

A second way to handle sparse data problems cause by state splitting is to smooth, as we did in Chapter 1. So we might estimate the probability of a parent anotated constituent by blending it with the unannoted version. So if $R_1 = VP^{VP} \rightarrow VBDNP^{VP}$ we could smooth it with $R_2 = VP \rightarrow VBDNP$ as follows:

$$\widetilde{P}_{R_1} = \frac{n_{R_1}(\boldsymbol{d}) + \beta\, \widetilde{P}_{R_2}}{n_{VB^{VB}}(\boldsymbol{d}) + \beta}$$

4.7 Programming A Parser

Section 4.3 layed out the basics of probabilistic CFG parsing. When we left it, it looked something like this: The basic data structure is the grammar, a set of rules where a rule r is the fourtuple $< lhs, rhs1, rhs2, prob >$. The chart is an N by N array of Cells. A Cell is a set of Constituents, and a constituent is a four tuple $< label, \rho_1, \rho_2.\mu >$. A constituent coresponding to a terminal symbol would have $\mu = 1$ and $\rho_1 = \rho_x = NULL$. (No back pointters).

The parsing algorithm thus is this:

Function: parse($w_{0,L}$)

```
1. read in binarized tree-bank grammar.
2. for $\ell = 1$ to $L$
   (a) for $s = 0$ to $(L - \ell)$
       i. fill $\mathcal{C}(s, s + \ell)$
3. if $Root(0, N) \in \mathcal{C}(0, N)$
   (a) return $debinarized(\rho_{Root(0,n)})$
   (b) else return NULL
```

Function: fill($\mathcal{C}(i, k)$)

```
1. if $k = (i + 1)$
   (a) add constituent $c =< word, NULL, NULL, 1 >$ to $\mathcal{C}$
2. for $j = (i + 1)$ to $(k - 1)$
   (a) for $c_1 \in \mathcal{C}(i, j)$ and $c_2 \in \mathcal{C}(j, k)$ and rule $r =< l, lab_{c_1}, lab_{c_2}, p >$
       i. create constituent $c =< lhs, c_1, c_2, \mu_n = (p_r{\cdot}\mu_{c_1}{\cdot}\mu c_2) >$
       ii. If there is no $c' \in$cell with $lab_{c'} = lhs_r$ and $\mu_{c'} \geq \mu_n$
       iii. add $n$ to $\mathcal{C}(i, k)$.
3. While adding new or higher $\mu$ constituents to $\mathcal{C}(j, k)$
   (a) for $c \in \mathcal{C}$ and $r =< lhs, lab_c, NULL, p >$
```

```
i. create constituent $n =< lhs_r, c, NULL, \mu_n = (\mu_c \cdot p_r) >$
ii. if there is no $c' \in \mathcal{C}$ $lab_c = lab_{c'}$ and $\mu_{c'} >= \mu_n$
    A. add n to cell.
```

In words, we fill up the chart always working on smaller spans before larger ones for which they are a substring. The for each cell with a span of one we first add the word from the string. Then for larger spans we use the binarized grammar to create new constituents from pairs of smaller constituents. And for any size cell, we add constituents from unary rules repeatedly untill there are no more to add.

So far in this book we have refrained from delving below the typical level for psuedo-code. But here there is one issue that can drastically affect performance, and will be far from obvious to the reader. It has to do with the triple loop inside FILL, to the effect "for all rules, left constituents and right constituents" when you think about this you realize that these three loops can be in any order. That is, we could first find a left-constituent, then find rules that match this left constituent, and finally look for the rule's right constituent in the chart. If we find it we apply to rule to create a new constituent of the type specified by the left-hand side of the rule. But we could do this in the reverse order, first right, then rule, then left – or first left, then right, then rule. And it turns out that this can make a *big* difference in the speed of your program. In particular, do not use the ordering left, right, rule.

To see this, we first note that for a normal Chomsky normal form implementation, the average cell has about 750 constituents. This means if the two loops over constituents go on the outside, the loop in which we consider rules is executed 750·750, e.g., about a half million, times. This will slow you down.

Now at first 750 may seem excessive. The Penn-Treebank only has about thirty phrasal constituents. Where did the other 720 come from? The answer, of course, is that they are all symbols created by binarization, e.g., $DT_J J_N N$. In fact, there are about fourty binarized constituents in a call for every "regular" one.

But there is some good news here. As we noted at the end of Section 4.3, the right-most symbol in a binary rule can never be a binarization non-terminal. That means there are not 750 * 750 possible combinations, to build, but only say, 750*30, where 30 is the number of original phrasal non-terminals.

If we use an ordering with rule selection in the middle we automatically get this efficiency. Suppose we start by iterating over possible left-hand side constituents. As we just noted, there will be on average about 750 of them. We now take the first of these, say with the label DT_NN. Next we look for rules of the form X $\rightarrow DT_NNY$. Finally we look for Y's. But we are guaranteed there can be a maximum 30 of these, since no binary constituents will every appear in this position within a rule. (For left-branching binarization. The opposite would be true if we had used right-branching.)

That is it. It is no harder doing rule selection in the middle, than last, and it is much faster.

4.8 Exercises

Exercise 4.1: Suppose we added the following rules to the grammar of Figure 4.5:

$$\text{NP} \rightarrow \text{NP CC NP} \quad \text{VBZ} \rightarrow \text{like} \quad \text{CC} \rightarrow \text{and}$$

Show the parse tree for the sentence "Sandy and Sam like the book".

Exercise 4.2: Show a dependency tree for the sentence in the previous exercise using Figure 4.3 as a template. Assume that the head of a conjoyned '*NP*' is the '*CC*', and its dependents are the noun-phrase conjuncts (the subordinate noun-phrases). Do not worry about the labels on arcs.

Exercise 4.3: Fill in the chart of Figure 4.13 with (a) two constituents which are missing, and (b) the μ values for all constituents according to the probabilities of Figure 4.7.

Exercise 4.4: Consider the probability of a string of N words $P(w_{1,N})$ according to a PCFG. Is this equal to $\alpha_{ROOT}(1, N)$, $\beta_{ROOT}(1, N)$, both, or neither?

Exercise 4.5: Suppose the third word of a string of length $N > 3$ is "apple" where $\rho_{X \rightarrow apple}$ is 0.001 when X is NOUN, but zero for any other part of speech. Which of the following is correct? $\alpha_{NOUN}(2, 3)$ is equal to $P(w_{1,N})$, it is $1000P(w_{1,N})$, it is .001, it is 1.0? What about $\beta_{NOUN}2, 3$?

4.9 Programming Assignment

Write a parser and evaluate its accuracy. You'll use the following files:

1. `wsj2-21.blt`, which contains binarized rules and their counts extracted from sections 2–21 of the Penn WSJ treebank. All words that appear less than 5 times have been replaced with `*UNK*`. A new root node labeled `TOP` has been inserted. Also, preterminal symbols that consist entirely of punctuation characters have had a caret '^' appended to them, to distinguish them from the identical terminal symbols. You'll need to remove this from the parse trees you finally produce (`munge-trees` can do this if you want).

2. `wsj24.tree`, which contains the Penn WSJ treebank trees for section 24. Terminals not appearing in `wsj2-21.blt` have been replaced with `*UNK*`. You should use this with the EVALB scoring program to evaluate your

3. `wsj24.txt`, which contains the terminal strings of `wsj24.tree`.

4. The EVALB parser scoring program and instructions.

Note that the EVALB program returns various types of results. One is just the POS tag accuracy of the parser. This should be in the mid %90 range, much like an HMM tagger. The result we car about the the labeled precision recall F-measure. This will be much lower — about %70.

Chapter 5

Topic Modeling and PLSA

This is just the beginings of a chapter. The point is to give you enough information (when suplimented by the lectures) to do the fifth programming assignment

5.1 Topic Modeling

Notably absent in this book is any discussion of the *meaning* of language. For example, the entire point of translation is to capture the meaning of a sentence written in one language in a second sentence written in another, yet the models we created did not deal with the meanings of the words at all. (Though they did rely on the fact that the sentences pairs of the requisite parallel corpus meant the same thing.) Indeed, the success of MT has been due to our ability to translate without the program knowing what it is talking about.

A large fraction of current CL research is trying change this, but textbooks such as this restrict themselves to areas where there is some consensus on how things should work, and as far as meanings are concerned these are few and far between. This chapter is concerned with one of these few: topic modeling.

Intuitively texts such as, e.g., newspaper articles, are "about" something — their *topic*. So consider the following

> It was the surgical head nurse who turned the South African hospital upside down. She needed a gynecologist, and it made perfect sense to her to choose Dr. E.T. Mokgokong, who would

> soon become deputy head of obstetrics and gynecology at the Univeristy of Natal. Except that it was 20 years ago at the height of apartheid and the nurse was white and Dr. Mokgokong is black. "She caused complete pandemonium in the hospital," the doctor recalled.

If asked to pick words from the article that best express what it is about we would pick, say, "apartheid", "South Africa", "gynecologist", as opposed to, say, "pandemonium". Furthermore, we would not hesitate to say that this articles is about apartheid and is not about World-War II, or the United States Congress.

This chapter is concerned with *topic modeling*, writing programs that get at this notion of "aboutness."

5.2 Probabilistic Latent Semantic Analysis

In particular we look at a formal model called *Probabilistic Latent Semantic Analysis* or PLSA. PLSA models have much in common with the language models we delt with in Chapter 1 in so far as they allow us to assign a probability to a text, but instead of basing the probability of a word on the previous word, we base it on the topics of the document. If our corpus consists of M "documents" (e.g., newspaper articles), each with l_d words, then the probability of our corpus c is the product of the document probabilities

$$\mathrm{P}c = \prod_{d=1}^{M} \mathrm{P}(w_{1,l_d}^{d}) \tag{5.1}$$

where w_{1,l_d}^{d} is the sequence of words w_{1,l_d}in document d. The probability of the words is

$$\mathrm{P}(w_{1,l_d}^{d}) = \prod_{i=1}^{l_d} \sum_{t=1}^{N} \mathrm{P}(T_i = t \mid d)\mathrm{P}(w_i \mid t). \tag{5.2}$$

We do not bother to generate the length of the documents, nor how many they are — just the words. We choose in advance how many different topics N are allowed. We will choose, say fifty. The model is then parameterized by two sets of parameters:

$$\begin{aligned} \mathrm{P}(T = t \mid D = d) &= \delta_{d,t} \\ \mathrm{P}(W_i = w \mid T_i = t) &= \tau_{t,w} \end{aligned}$$

The generative story for this model is as follows: for each word we first pick a topic according to a document specific distribution $\delta_{d,t}$, and then pick a word from this topic according to the distribution $\tau_{t,w}$. So the δs specify the topics of the documents, and the τs specify the words that are used in a topic. The probably of the data is the sum over all ways it could be generated.

Here we are not interested in language modeling, but determining topics, so we ignore words that are not topic specific. In particular we use a *stopword* list — a list of words to ignore. Such lists were originally created for information retrieval purposes. Stopwords appearing in the query were ignored. These words include function words such as prepositions, pronouns, modal verbs, etc. For our purposes we also remove as some very common/general words such as "large", "big", "slowly", etc.

Example 5.1: A fifty topic PLSA model run on 1000 newspaper articles, including the one which starts with the above four sentences, produced the following δ parameters for the article:

Topic	$\mathrm{P}(t \mid d)$
21	0.221112
33	0.138694
40	0.113522
46	0.496539

These four topics account for just above % 96 of the probability mass. All of the others are very close to zero. Figure 5.1 shows words associated with five topics, three associated with this article, two not.

Topic 46 is about South Africa and apartheid, while topics 33 and 40 split up different aspects of medicine — 33 more about health-care, and 40 about medical treatment with a hint of law. Note that in principle *all* words have some probability of appearing in an article, about any given topic, but most will have very low probability for that topic. Here the words we show are those that maximize the smoothed probability of the topic given the word

$$\widetilde{\mathrm{P}}(t_k \mid w_i) = \frac{n_{k,i}(c) + \alpha}{n_{k,\circ}(c) + N\alpha} \tag{5.3}$$

where we set α to 5.

Topic No.	Words
1	white house watkins clinton helicopter david trip staff officials golf camp military office president course cost duffy chief presidential washington
2	u.n. rwanda united government bosnia war troops nations bosnian peace serbs military forces rebels force army peacekeeping president somalia boutros-ghali
33	health care percent people children estate tax taxes black trust plan family doctors quebec political money voters public capital jackson
40	drug patients king cancer nih breast percent treatment disease medical researchers panel fda report court study fiau public cases jury
46	africa south government mandela development countries black international nations aid african country housing program apartheid economic national lane political president

Figure 5.1: The words associated with five of the fifty topics created by a PLSA model of one thousand. Tokens from a 500-word stopword list were removed, and all words were lower-cased. newspaper articles

5.3 Learning PLSA parameters

We again turn to EM to learn the necessary parameters. As before, we first consider learning with a training corpus in which every word in our corpus c is marked with its associated topic. The same word may have a different topic in different documents. In particular, note that the word "president" appears in topics 1, 2, and 46 in Figure 5.1. With such a corpus we can set our parameters to their maximum-likelihood estimates:

$$\begin{aligned}\delta_{d,t} &= \frac{n_{d,t}(c)}{n_{d,\circ}}(c) \\ \tau_{t,w} &= \frac{n_{t,w}(c)}{n_{t,\circ}}(c).\end{aligned}$$

Since we do not have labeled data, we use EM. The algorithm is as follows:

1. Pick positive initial values for $\tau_{t,w}$ for all topics t and words w (equal is fine).

2. Pick positive vales for $\delta_{d,t}$ for all documents d and topics t. They should be almost equal, but with about 2% randomness to avoid the saddle point.

3. For $i = 1, 2, \ldots$ until convergence do:

   (a) *E-step:*
   Set $n_{d,t}$ and $n_{t,w}$ to zero for all documents d, topics t, and words w
   For each word w (that appears in its document d) do:

      i. Set $p = \sum_{t'=0}^{N} \delta_{d,t'}\tau_{t',w}$,
      ii. Set $q = \frac{\delta_{d,t}\tau_{t,w}}{p}$
      iii. Set both $n_{d,t}$ and $n_{t,w}$ += q.

   (b) *M-step:*
   Set $\delta_{d,t} = n_{d,t}/n_{d,\circ}$ and $\tau_{t,w} = n_{t,w}/n_{t,\circ}$.

It very similar to the EM algorithm for IBM model one.

5.4 Programming assignment

The data for this assignment is in news1000.txt The format of the file is as follows:

> wordCount
> word-1 word-2 ... word-12
> word-13 ... word-24
> ... word-wordCount

This is repeated 1000 times, once for each news article. The words are all of the words in the articles that (a) appear at least 5 times and in two different articles, and (b) do not appear in a stopword list of common non-topical words.

Your assignment is to create a PLSA model with 50 topics. To demonstrate that your program works it should output the following facts:

1. The log likelihood of the data at convergence. (We will define convergence as the point where the log-likelihood changes by less than 1% between one iteration and the next.)

2. the probabilities of topics for article 17 (as a cross check, it has 367 words).

3. the most probable 15 words w for each topic t according to the following formula:

$$\widetilde{\mathrm{P}}(w \mid t) = \frac{n_{t,w} + \alpha}{n_{t,\circ} + \alpha|\mathcal{W}|}.$$

   Set $\alpha = 5$.