Chapter 1

Language Modeling

(Course notes for NLP by Michael Collins, Columbia University)

1.1 Introduction

In this chapter we will consider the the problem of constructing a *language model* from a set of example sentences in a language. Language models were originally developed for the problem of speech recognition; they still play a central role in modern speech recognition systems. They are also widely used in other NLP applications. The parameter estimation techniques that were originally developed for language modeling, as described in this chapter, are useful in many other contexts, such as the tagging and parsing problems considered in later chapters of this book.

Our task is as follows. Assume that we have a *corpus*, which is a set of sentences in some language. For example, we might have several years of text from the New York Times, or we might have a very large amount of text from the web. Given this corpus, we'd like to estimate the parameters of a language model.

A language model is defined as follows. First, we will define $\mathcal{V}$ to be the set of all words in the language. For example, when building a language model for English we might have

$$\mathcal{V} = \{\text{the, dog, laughs, saw, barks, cat, } \ldots\}$$

In practice $\mathcal{V}$ can be quite large: it might contain several thousands, or tens of thousands, of words. We assume that $\mathcal{V}$ is a finite set. A *sentence* in the language is a sequence of words

$$x_1 x_2 \ldots x_n$$

where the integer n is such that $n \geq 1$, we have $x_i \in \mathcal{V}$ for $i \in \{1 \ldots (n-1)\}$, and we assume that x_n is a special symbol, STOP (we assume that STOP is not a

member of $\mathcal{V}$). We'll soon see why it is convenient to assume that each sentence ends in the STOP symbol. Example sentences could be

the dog barks STOP
the cat laughs STOP
the cat saw the dog STOP
the STOP
cat the dog the STOP
cat cat cat STOP
STOP
...

We will define $\mathcal{V}^\dagger$ to be the set of all sentences with the vocabulary $\mathcal{V}$: this is an infinite set, because sentences can be of any length.

We then give the following definition:

Definition 1 (Language Model) *A language model consists of a finite set $\mathcal{V}$, and a function $p(x_1, x_2, \ldots x_n)$ such that:*

1. *For any $\langle x_1 \ldots x_n \rangle \in \mathcal{V}^\dagger$, $p(x_1, x_2, \ldots x_n) \geq 0$*
2. *In addition,*
$$\sum_{\langle x_1 \ldots x_n \rangle \in \mathcal{V}^\dagger} p(x_1, x_2, \ldots x_n) = 1$$

Hence $p(x_1, x_2, \ldots x_n)$ is a probability distribution over the sentences in $\mathcal{V}^\dagger$.

As one example of a (very bad) method for learning a language model from a training corpus, consider the following. Define $c(x_1 \ldots x_n)$ to be the number of times that the sentence $x_1 \ldots x_n$ is seen in our training corpus, and N to be the total number of sentences in the training corpus. We could then define

$$p(x_1 \ldots x_n) = \frac{c(x_1 \ldots x_n)}{N}$$

This is, however, a very poor model: in particular it will assign probability 0 to any sentence not seen in the training corpus. Thus it fails to generalize to sentences that have not been seen in the training data. The key technical contribution of this chapter will be to introduce methods that do generalize to sentences that are not seen in our training data.

At first glance the language modeling problem seems like a rather strange task, so why are we considering it? There are a couple of reasons:

1. Language models are very useful in a broad range of applications, the most obvious perhaps being speech recognition and machine translation. In many applications it is very useful to have a good "prior" distribution $p(x_1 \dots x_n)$ over which sentences are or aren't probable in a language. For example, in speech recognition the language model is combined with an acoustic model that models the pronunciation of different words: one way to think about it is that the acoustic model generates a large number of candidate sentences, together with probabilities; the language model is then used to reorder these possibilities based on how likely they are to be a sentence in the language.

2. The techniques we describe for defining the function p, and for estimating the parameters of the resulting model from training examples, will be useful in several other contexts during the course: for example in hidden Markov models, which we will see next, and in models for natural language parsing.

1.2 Markov Models

We now turn to a critical question: given a training corpus, how do we learn the function p? In this section we describe *Markov models*, a central idea from probability theory; in the next section we describe *trigram language models*, an important class of language models that build directly on ideas from Markov models.

1.2.1 Markov Models for Fixed-length Sequences

Consider a sequence of random variables, $X_1, X_2, \dots, X_n$. Each random variable can take any value in a finite set $\mathcal{V}$. For now we will assume that the length of the sequence, n, is some fixed number (e.g., $n = 100$). In the next section we'll describe how to generalize the approach to cases where n is also a random variable, allowing different sequences to have different lengths.

Our goal is as follows: we would like to model the probability of any sequence $x_1 \dots x_n$, where $n \geq 1$ and $x_i \in \mathcal{V}$ for $i = 1 \dots n$, that is, to model the joint probability

$$P(X_1 = x_1, X_2 = x_2, \dots, X_n = x_n)$$

There are $|\mathcal{V}|^n$ possible sequences of the form $x_1 \dots x_n$: so clearly, it is not feasible for reasonable values of $|\mathcal{V}|$ and n to simply list all $|\mathcal{V}|^n$ probabilities. We would like to build a much more compact model.

In a first-order Markov process, we make the following assumption, which considerably simplifies the model:

$$P(X_1 = x_1, X_2 = x_2, \dots X_n = x_n)$$

$$= P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_1 = x_1, \ldots, X_{i-1} = x_{i-1}) \quad (1.1)$$

$$= P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_{i-1} = x_{i-1}) \quad (1.2)$$

The first step, in Eq. 1.1, is exact: by the chain rule of probabilities, *any* distribution $P(X_1 = x_1 \ldots X_n = x_n)$ can be written in this form. So we have made no assumptions in this step of the derivation. However, the second step, in Eq. 1.2, is not necessarily exact: we have made the assumption that for any $i \in \{2 \ldots n\}$, for any $x_1 \ldots x_i$,

$$P(X_i = x_i | X_1 = x_1 \ldots X_{i-1} = x_{i-1}) = P(X_i = x_i | X_{i-1} = x_{i-1})$$

This is a (first-order) *Markov assumption*. We have assumed that the identity of the i'th word in the sequence depends only on the identity of the previous word, x_{i-1}. More formally, we have assumed that the value of X_i is conditionally independent of $X_1 \ldots X_{i-2}$, given the value for X_{i-1}.

In a second-order Markov process, which will form the basis of trigram language models, we make a slightly weaker assumption, namely that each word depends on the previous *two* words in the sequence:

$$\begin{aligned} & P(X_i = x_i | X_1 = x_1, \ldots, X_{i-1} = x_{i-1}) \\ = \; & P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1}) \end{aligned}$$

It follows that the probability of an entire sequence is written as

$$\begin{aligned} & P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n) \\ = \; & \prod_{i=1}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1}) \end{aligned} \quad (1.3)$$

For convenience, we will assume that $x_0 = x_{-1} =$ * in this definition, where * is a special "start" symbol in the sentence.

1.2.2 Markov Sequences for Variable-length Sentences

In the previous section, we assumed that the length of the sequence, n, was fixed. In many applications, however, the length n can itself vary. Thus n is itself a random variable. There are various ways of modeling this variability in length: in this section we describe the most common approach for language modeling.

The approach is simple: we will assume that the n'th word in the sequence, X_n, is always equal to a special symbol, the STOP symbol. This symbol can only

appear at the end of a sequence. We use exactly the same assumptions as before: for example under a second-order Markov assumption, we have

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n) = \prod_{i=1}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1}) \tag{1.4}$$

for any $n \geq 1$, and for any $x_1 \ldots x_n$ such that $x_n =$ STOP, and $x_i \in \mathcal{V}$ for $i = 1 \ldots (n-1)$.

We have assumed a second-order Markov process where at each step we generate a symbol x_i from the distribution

$$P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

where x_i can be a member of $\mathcal{V}$, or alternatively can be the STOP symbol. If we generate the STOP symbol, we finish the sequence. Otherwise, we generate the next symbol in the sequence.

A little more formally, the process that generates sentences would be as follows:

1. Initialize $i = 1$, and $x_0 = x_{-1} =$ *
2. Generate x_i from the distribution
$$P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$
3. If $x_i =$ STOP then return the sequence $x_1 \ldots x_i$. Otherwise, set $i = i + 1$ and return to step 2.

Thus we now have a model that generates sequences that vary in length.

1.3 Trigram Language Models

There are various ways of defining language models, but we'll focus on a particularly important example, the trigram language model, in this chapter. This will be a direct application of Markov models, as described in the previous section, to the language modeling problem. In this section we give the basic definition of a trigram model, discuss maximum-likelihood parameter estimates for trigram models, and finally discuss strengths of weaknesses of trigram models.

1.3.1 Basic Definitions

As in Markov models, we model each sentence as a sequence of n random variables, $X_1, X_2, \ldots X_n$. The length, n, is itself a random variable (it can vary across different sentences). We always have $X_n = \text{STOP}$. Under a second-order Markov model, the probability of any sentence $x_1 \ldots x_n$ is then

$$P(X_1 = x_1, X_2 = x_2, \ldots X_n = x_n) = \prod_{i=1}^{n} P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1})$$

where we assume as before that $x_0 = x_{-1} = *$.

We will assume that for any i, for any x_{i-2}, x_{i-1}, x_i,

$$P(X_i = x_i | X_{i-2} = x_{i-2}, X_{i-1} = x_{i-1}) = q(x_i | x_{i-2}, x_{i-1})$$

where $q(w|u, v)$ for any (u, v, w) is a parameter of the model. We will soon see how to derive estimates of the $q(w|u, v)$ parameters from our training corpus. Our model then takes the form

$$p(x_1 \ldots x_n) = \prod_{i=1}^{n} q(x_i | x_{i-2}, x_{i-1})$$

for any sequence $x_1 \ldots x_n$.

This leads us to the following definition:

Definition 2 (Trigram Language Model) *A trigram language model consists of a finite set $\mathcal{V}$, and a parameter*

$$q(w|u, v)$$

for each trigram u, v, w such that $w \in \mathcal{V} \cup \{STOP\}$, and $u, v \in \mathcal{V} \cup \{*\}$. The value for $q(w|u, v)$ can be interpreted as the probability of seeing the word w immediately after the bigram (u, v). For any sentence $x_1 \ldots x_n$ where $x_i \in \mathcal{V}$ for $i = 1 \ldots (n-1)$, and $x_n = STOP$, the probability of the sentence under the trigram language model is

$$p(x_1 \ldots x_n) = \prod_{i=1}^{n} q(x_i | x_{i-2}, x_{i-1})$$

where we define $x_0 = x_{-1} = *$. □

For example, for the sentence

the dog barks STOP

we would have

p(the dog barks STOP) = q(the|*, *)×q(dog|*, the)×q(barks|the, dog)×q(STOP|dog, barks)

Note that in this expression we have one term for each word in the sentence (*the, dog, barks*, and *STOP*). Each word depends only on the previous two words: this is the trigram assumption.

The parameters satisfy the constraints that for any trigram u, v, w,

$$q(w|u, v) \geq 0$$

and for any bigram u, v,

$$\sum_{w \in \mathcal{V} \cup \{\text{STOP}\}} q(w|u, v) = 1$$

Thus $q(w|u, v)$ defines a distribution over possible words w, conditioned on the bigram context u, v.

The key problem we are left with is to estimate the parameters of the model, namely

$$q(w|u, v)$$

where w can be any member of $\mathcal{V} \cup \{\text{STOP}\}$, and $u, v \in \mathcal{V} \cup \{*\}$. There are around $|\mathcal{V}|^3$ parameters in the model. This is likely to be a very large number. For example with $|\mathcal{V}| = 10,000$ (this is a realistic number, most likely quite small by modern standards), we have $|\mathcal{V}|^3 \approx 10^{12}$.

1.3.2 Maximum-Likelihood Estimates

We first start with the most generic solution to the estimation problem, the *maximum-likelihood estimates*. We will see that these estimates are flawed in a critical way, but we will then show how related estimates can be derived that work very well in practice.

First, some notation. Define $c(u, v, w)$ to be the number of times that the trigram (u, v, w) is seen in the training corpus: for example, c(the, dog, barks) is the number of times that the sequence of three words *the dog barks* is seen in the training corpus. Similarly, define $c(u, v)$ to be the number of times that the bigram (u, v) is seen in the corpus. For any w, u, v, we then define

$$q(w|u, v) = \frac{c(u, v, w)}{c(u, v)}$$

As an example, our estimate for $q(\text{barks}|\text{the, dog})$ would be

$$q(\text{barks}|\text{the, dog}) = \frac{c(\text{the, dog, barks})}{\text{c(the, dog)}}$$

This estimate is very natural: the numerator is the number of times the entire trigram *the dog barks* is seen, and the denominator is the number of times the bigram *the dog* is seen. We simply take the ratio of these two terms.

Unfortunately, this way of estimating parameters runs into a very serious issue. Recall that we have a very large number of parameters in our model (e.g., with a vocabulary size of $10,000$, we have around 10^{12} parameters). Because of this, many of our counts will be zero. This leads to two problems:

- Many of the above estimates will be $q(w|u, v) = 0$, due to the count in the numerator being 0. This will lead to many trigram probabilities being systematically underestimated: it seems unreasonable to assign probability 0 to any trigram not seen in training data, given that the number of parameters of the model is typically very large in comparison to the number of words in the training corpus.

- In cases where the denominator $c(u, v)$ is equal to zero, the estimate is not well defined.

We will shortly see how to come up with modified estimates that fix these problems. First, however, we discuss how language models are evaluated, and then discuss strengths and weaknesses of trigram language models.

1.3.3 Evaluating Language Models: Perplexity

So how do we measure the quality of a language model? A very common method is to evaluate the *perplexity* of the model on some held-out data.

The method is as follows. Assume that we have some test data sentences $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$. Each test sentence $x^{(i)}$ for $i \in \{1 \ldots m\}$ is a sequence of words $x_1^{(i)}, \ldots, x_{n_i}^{(i)}$, where n_i is the length of the i'th sentence. As before we assume that every sentence ends in the STOP symbol.

It is critical that the test sentences are "held out", in the sense that *they are not part of the corpus used to estimate the language model*. In this sense, they are examples of new, unseen sentences.

For any test sentence $x^{(i)}$, we can measure its probability $p(x^{(i)})$ under the language model. A natural measure of the quality of the language model would be

the probability it assigns to the entire set of test sentences, that is

$$\prod_{i=1}^{m} p(x^{(i)})$$

The intuition is as follows: the higher this quantity is, the better the language model is at modeling unseen sentences.

The perplexity on the test corpus is derived as a direct transformation of this quantity. Define M to be the total number of words in the test corpus. More precisely, under the definition that n_i is the length of the i'th test sentence,

$$M = \sum_{i=1}^{m} n_i$$

Then the average log probability under the model is defined as

$$\frac{1}{M} \log_2 \prod_{i=1}^{m} p(x^{(i)}) = \frac{1}{M} \sum_{i=1}^{m} \log_2 p(x^{(i)})$$

This is just the log probability of the entire test corpus, divided by the total number of words in the test corpus. Here we use $\log_2(z)$ for any $z > 0$ to refer to the $\log$ with respect to base 2 of z. Again, the higher this quantity is, the better the language model.

The *perplexity* is then defined as

$$2^{-l}$$

where

$$l = \frac{1}{M} \sum_{i=1}^{m} \log_2 p(x^{(i)})$$

Thus we take the negative of the average log probability, and raise two to that power. (Again, we're assuming in this section that $\log_2$ is log base two). The perplexity is a positive number. The *smaller* the value of perplexity, the better the language model is at modeling unseen data.

Some intuition behind perplexity is as follows. Say we have a vocabulary $\mathcal{V}$, where $|\mathcal{V} \cup \{\text{STOP}\}| = N$, and the model predicts

$$q(w|u, v) = \frac{1}{N}$$

for all u, v, w. Thus this is the dumb model that simply predicts the uniform distribution over the vocabulary together with the STOP symbol. In this case, it can

be shown that the perplexity is equal to N. So *under a uniform probability model, the perplexity is equal to the vocabulary size*. Perplexity can be thought of as the effective vocabulary size under the model: if, for example, the perplexity of the model is 120 (even though the vocabulary size is say $10,000$), then this is roughly equivalent to having an effective vocabulary size of 120.

To give some more motivation, it is relatively easy to show that the perplexity is equal to

$$\frac{1}{t}$$

where

$$t = \sqrt[M]{\prod_{i=1}^{m} p(x^{(i)})}$$

Here we use $\sqrt[M]{z}$ to refer to the M'th root of z: so t is the value such that $t^M = \prod_{i=1}^{m} p(x^{(i)})$. Given that

$$\prod_{i=1}^{m} p(x^{(i)}) = \prod_{i=1}^{m} \prod_{j=1}^{n_i} q(x_j^{(i)} | x_{j-2}^{(i)}, x_{j-1}^{(i)})$$

and $M = \sum_i n_i$, the value for t is the *geometric mean* of the terms $q(x_j^{(i)} | x_{j-2}^{(i)}, x_{j-1}^{(i)})$ appearing in $\prod_{i=1}^{m} p(x^{(i)})$. For example if the perplexity is equal to 100, then $t = 0.01$, indicating that the geometric mean is 0.01.

One additional useful fact about perplexity is the following. If for any trigram u, v, w seen in test data, we have the estimate

$$q(w|u, v) = 0$$

then the perplexity will be ∞. To see this, note that in this case the probability of the test corpus under the model will be 0, and the average log probability will be $-\infty$. Thus if we take perplexity seriously as our measure of a language model, then we should avoid giving 0 estimates at all costs.

Finally, some intuition about "typical" values for perplexity. Goodman ("A bit of progress in language modeling", figure 2) evaluates unigram, bigram and trigram language models on English data, with a vocabulary size of $50,000$. In a bigram model we have parameters of the form $q(w|v)$, and

$$p(x_1 \dots x_n) = \prod_{i=1}^{n} q(x_i | x_{i-1})$$

Thus each word depends only on the previous word in the sentence. In a unigram model we have parameters $q(w)$, and

$$p(x_1 \dots x_n) = \prod_{i=1}^{n} q(x_i)$$

Thus each word is chosen completely independently of other words in the sentence. Goodman reports perplexity figures of around 74 for a trigram model, 137 for a bigram model, and 955 for a unigram model. The perplexity for a model that simply assigns probability $1/50,000$ to each word in the vocabulary would be $50,000$. So the trigram model clearly gives a big improvement over bigram and unigram models, and a huge improvement over assigning a probability of $1/50,000$ to each word in the vocabulary.

1.3.4 Strengths and Weaknesses of Trigram Language Models

The trigram assumption is arguably quite strong, and linguistically naive (see the lecture slides for discussion). However, it leads to models that are very useful in practice.

1.4 Smoothed Estimation of Trigram Models

As discussed previously, a trigram language model has a very large number of parameters. The maximum-likelihood parameter estimates, which take the form

$$q(w|u,v) = \frac{c(u,v,w)}{c(u,v)}$$

will run into serious issues with sparse data. Even with a large set of training sentences, many of the counts $c(u,v,w)$ or $c(u,v)$ will be low, or will be equal to zero.

In this section we describe *smoothed estimation methods*, which alleviate many of the problems found with sparse data. The key idea will be to rely on lower-order statistical estimates—in particular, estimates based on bigram or unigram counts—to "smooth" the estimates based on trigrams. We discuss two smoothing methods that are very commonly used in practice: first, *linear interpolation*; second, *discounting methods*.

1.4.1 Linear Interpolation

A linearly interpolated trigram model is derived as follows. We define the *trigram*, *bigram*, and *unigram* maximum-likelihood estimates as

$$q_{ML}(w|u,v) = \frac{c(u,v,w)}{c(u,v)}$$

$$q_{ML}(w|v) = \frac{c(v,w)}{c(v)}$$

$$q_{ML}(w) = \frac{c(w)}{c()}$$

where we have extended our notation: $c(w)$ is the number of times word w is seen in the training corpus, and $c()$ is the total number of words seen in the training corpus.

The trigram, bigram, and unigram estimates have different strengths and weaknesses. The unigram estimate will never have the problem of its numerator or denominator being equal to 0: thus the estimate will always be well-defined, and will always be greater than 0 (providing that each word is seen at least once in the training corpus, which is a reasonable assumption). However, the unigram estimate completely ignores the context (previous two words), and hence discards very valuable information. In contrast, the trigram estimate does make use of context, but has the problem of many of its counts being 0. The bigram estimate falls between these two extremes.

The idea in linear interpolation is to use all three estimates, by defining the trigram estimate as follows:

$$q(w|u,v) = \lambda_1 \times q_{ML}(w|u,v) + \lambda_2 \times q_{ML}(w|v) + \lambda_3 \times q_{ML}(w)$$

Here λ_1, λ_2 and λ_3 are three additional parameters of the model, which satisfy

$$\lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0$$

and

$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

Thus we take a weighted average of the three estimates.

There are various ways of estimating the λ values. A common one is as follows. Say we have some additional held-out data, which is separate from both our training and test corpora. We will call this data the *development data*. Define

$c'(u, v, w)$ to be the number of times that the trigram (u, v, w) is seen in the development data. It is easy to show that the log-likelihood of the development data, as a function of the parameters $\lambda_1, \lambda_2, \lambda_3$, is

$$\begin{aligned} L(\lambda_1, \lambda_2, \lambda_3) &= \sum_{u,v,w} c'(u, v, w) \log q(w|u, v) \\ &= \sum_{u,v,w} c'(u, v, w) \log \left(\lambda_1 \times q_{ML}(w|u, v) + \lambda_2 \times q_{ML}(w|v) + \lambda_3 \times q_{ML}(w)\right) \end{aligned}$$

We would like to choose our λ values to make $L(\lambda_1, \lambda_2, \lambda_3)$ as high as possible. Thus the λ values are taken to be

$$\arg \max_{\lambda_1, \lambda_2, \lambda_3} L(\lambda_1, \lambda_2, \lambda_3)$$

such that

$$\lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0$$

and

$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

Finding the optimal values for $\lambda_1, \lambda_2, \lambda_3$ is fairly straightforward (see section **??** for one algorithm that is often used for this purpose).

As described, our method has three smoothing parameters, λ_1, λ_2, and λ_3. The three parameters can be interpreted as an indication of the confidence, or weight, placed on each of the trigram, bigram, and unigram estimates. For example, if λ_1 is close to 1, this implies that we put a significant weight on the trigram estimate $q_{ML}(w|u, v)$; conversely, if λ_1 is close to zero we have placed a low weighting on the trigram estimate.

In practice, it is important to add an additional degree of freedom, by allowing the values for λ_1, λ_2 and λ_3 to vary depending on the bigram (u, v) that is being conditioned on. In particular, the method can be extended to allow λ_1 to be larger when $c(u, v)$ is larger—the intuition being that a larger value of $c(u, v)$ should translate to us having more confidence in the trigram estimate.

At the very least, the method is used to ensure that $\lambda_1 = 0$ when $c(u, v) = 0$, because in this case the trigram estimate

$$q_{ML}(w|u, v) = \frac{c(u, v, w)}{c(u, v)}$$

is undefined. Similarly, if both $c(u, v)$ and $c(v)$ are equal to zero, we need $\lambda_1 = \lambda_2 = 0$, as both the trigram and bigram ML estimates are undefined.

One extension to the method, often referred to as *bucketing*, is described in section 1.5.1. Another, much simpler method, is to define

$$\begin{aligned}\lambda_1 &= \frac{c(u,v)}{c(u,v)+\gamma} \\ \lambda_2 &= (1-\lambda_1) \times \frac{c(v)}{c(v)+\gamma} \\ \lambda_3 &= 1-\lambda_1-\lambda_2\end{aligned}$$

where $\gamma > 0$ is the only parameter of the method. It can be verified that $\lambda_1 \geq 0, \lambda_2 \geq 0$, and $\lambda_3 \geq 0$, and also that $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

Under this definition, it can be seen that λ_1 increases as $c(u,v)$ increases, and similarly that λ_2 increases as $c(v)$ increases. In addition we have $\lambda_1 = 0$ if $c(u,v) = 0$, and $\lambda_2 = 0$ if $c(v) = 0$. The value for γ can again be chosen by maximizing log-likelihood of a set of development data.

This method is relatively crude, and is not likely to be optimal. It is, however, very simple, and in practice it can work well in some applications.

1.4.2 Discounting Methods

We now describe an alternative estimation method, which is again commonly used in practice. Consider first a method for estimating a bigram language model, that is, our goal is to define

$$q(w|v)$$

for any $w \in \mathcal{V} \cup \{\text{STOP}\}$, $v \in \mathcal{V} \cup \{*\}$.

The first step will be to define *discounted counts*. For any bigram $c(v,w)$ such that $c(v,w) > 0$, we define the discounted count as

$$c^*(v,w) = c(v,w) - \beta$$

where β is a value between 0 and 1 (a typical value might be $\beta = 0.5$). Thus we simply subtract a constant value, β, from the count. This reflects the intuition that if we take counts from the training corpus, we will systematically over-estimate the probability of bigrams seen in the corpus (and under-estimate bigrams not seen in the corpus).

For any bigram (v,w) such that $c(v,w) > 0$, we can then define

$$q(w|v) = \frac{c^*(v,w)}{c(v)}$$

Thus we use the discounted count on the numerator, and the regular count on the denominator of this expression.

x	$c(x)$	$c^*(x)$	$\frac{c^*(x)}{c(\textit{the})}$
the	48		
the, dog	15	14.5	14.5/48
the, woman	11	10.5	10.5/48
the, man	10	9.5	9.5/48
the, park	5	4.5	4.5/48
the, job	2	1.5	1.5/48
the, telescope	1	0.5	0.5/48
the, manual	1	0.5	0.5/48
the, afternoon	1	0.5	0.5/48
the, country	1	0.5	0.5/48
the, street	1	0.5	0.5/48

Figure 1.1: An example illustrating discounting methods. We show a made-up example, where the unigram *the* is seen 48 times, and we list all bigrams (u, v) such that $u =$ *the*, and $c(u, v) > 0$ (the counts for these bigrams sum to 48). In addition we show the discounted count $c^*(x) = c(x) - \beta$, where $\beta = 0.5$, and finally we show the estimate $c^*(x)/c(\textit{the})$ based on the discounted count.

For any context v, this definition leads to some *missing probability mass*, defined as

$$\alpha(v) = 1 - \sum_{w:c(v,w)>0} \frac{c^*(v,w)}{c(v)}$$

As an example, consider the counts shown in the example in figure 1.1. In this case we show all bigrams (u, v) where $u =$ *the*, and $c(u, v) > 0$. We use a discounting value of $\beta = 0.5$. In this case we have

$$\sum_{w:c(v,w)>0} \frac{c^*(v,w)}{c(v)} = \frac{14.5}{48} + \frac{10.5}{48} + \frac{9.5}{48} + \frac{4.5}{48} + \frac{1.5}{48} + 5 \times \frac{0.5}{48} = \frac{43}{48}$$

and the missing mass is

$$\alpha(\textit{the}) = 1 - \frac{43}{48} = \frac{5}{48}$$

The intuition behind discounted methods is to divide this "missing mass" between the words w such that $c(v, w) = 0$.

More specifically, the complete definition of the estimate is as follows. For any v, define the sets

$$\mathcal{A}(v) = \{w : c(v, w) > 0\}$$

and

$$\mathcal{B}(v) = \{w : c(v, w) = 0\}$$

For the data in figure 1.1, for example, we would have

$\mathcal{A}(\textit{the}) = \{\textit{dog, woman, man, park, job, telescope, manual, afternoon, country, street}\}$

and $\mathcal{B}(\textit{the})$ would be the set of remaining words in the vocabulary.

Then the estimate is defined as

$$q_D(w|v) = \begin{cases} \frac{c^*(v,w)}{c(v)} & \text{If } w \in \mathcal{A}(v) \\ \alpha(v) \times \frac{q_{ML}(w)}{\sum_{w \in \mathcal{B}(v)} q_{ML}(w)} & \text{If } w \in \mathcal{B}(v) \end{cases}$$

Thus if $c(v, w) > 0$ we return the estimate $c^*(v, w)/c(v)$; otherwise we divide the remaining probability mass $\alpha(v)$ in proportion to the unigram estimates $q_{ML}(w)$.

The method can be generalized to trigram language models in a natural, recursive way: for any bigram (u, v) define

$$\mathcal{A}(u, v) = \{w : c(u, v, w) > 0\}$$

and

$$\mathcal{B}(u, v) = \{w : c(u, v, w) = 0\}$$

Define $c^*(u, v, w)$ to be the discounted count for the trigram (u, v, w): that is,

$$c^*(u, v, w) = c(u, v, w) - \beta$$

where β is again the discounting value. Then the trigram model is

$$q_D(w|u, v) = \begin{cases} \frac{c^*(u,v,w)}{c(u,v)} & \text{If } w \in \mathcal{A}(u, v) \\ \alpha(u, v) \times \frac{q_D(w|v)}{\sum_{w \in \mathcal{B}(u,v)} q_D(w|v)} & \text{If } w \in \mathcal{B}(u, v) \end{cases}$$

where

$$\alpha(u, v) = 1 - \sum_{w \in \mathcal{A}(u,v)} \frac{c^*(u, v, w)}{c(u, v)}$$

is again the "missing" probability mass. Note that we have divided the missing probability mass in proportion to the bigram estimates $q_D(w|v)$, which were defined previously.

The only parameter of this approach is the discounting value, β. As in linearly interpolated models, the usual way to choose the value for this parameter is by optimization of likelihood of a development corpus, which is again a separate set of data from the training and test corpora. Define $c'(u, v, w)$ to be the number of times that the trigram u, v, w is seen in this development corpus. The log-likelihood of the development data is

$$\sum_{u,v,w} c'(u, v, w) \log q_D(w|u, v)$$

where $q_D(w|u, v)$ is defined as above. The parameter estimates $q_D(w|u, v)$ will vary as the value for β varies. Typically we will test a set of possible values for β—for example, we might test all values in the set $\{0.1, 0.2, 0.3, \ldots, 0.9\}$—where for each value of β we calculate the log-likelihood of the development data. We then choose the value for β that maximizes this log-likelihood.

1.5 Advanced Topics

1.5.1 Linear Interpolation with Bucketing

In linearly interpolated models the parameter estimates are defined as

$$q(w|u, v) = q(w|u, v) = \lambda_1 \times q_{ML}(w|u, v) + \lambda_2 \times q_{ML}(w|v) + \lambda_3 \times q_{ML}(w)$$

where λ_1, λ_2 and λ_3 are smoothing parameters in the approach.

In practice, it is important to allow the smoothing parameters to vary depending on the bigram (u, v) that is being conditioned on—in particular, the higher the count $c(u, v)$, the higher the weight should be for λ_1 (and similarly, the higher the count $c(v)$, the higher the weight should be for λ_2). The classical method for achieving this is through an extension that is often referred to as "bucketing".

The first step in this method is to define a function Π that maps bigrams (u, v) to values $\Pi(u, v) \in \{1, 2, \ldots, K\}$ where K is some integer specifying the number of buckets. Thus the function Π defines a partition of bigrams into K different subsets. The function is defined by hand, and typically depends on the counts seen in the training data. One such definition, with $K = 3$, would be

$$\begin{aligned}
\Pi(u, v) &= 1 \quad \text{if } c(u, v) > 0 \\
\Pi(u, v) &= 2 \quad \text{if } c(u, v) = 0 \text{ and } c(v) > 0 \\
\Pi(u, v) &= 3 \quad \text{otherwise}
\end{aligned}$$

This is a very simple definition that simply tests whether the counts $c(u, v)$ and $c(v)$ are equal to 0.

Another slightly more complicated definition, which is more sensitive to frequency of the bigram (u, v), would be

$$\begin{aligned}
\Pi(u, v) &= 1 \quad \text{if } 100 \leq c(u, v) \\
\Pi(u, v) &= 2 \quad \text{if } 50 \leq c(u, v) < 100 \\
\Pi(u, v) &= 3 \quad \text{if } 20 \leq c(u, v) < 50 \\
\Pi(u, v) &= 4 \quad \text{if } 10 \leq c(u, v) < 20 \\
\Pi(u, v) &= 5 \quad \text{if } 5 \leq c(u, v) < 10 \\
\Pi(u, v) &= 6 \quad \text{if } 2 \leq c(u, v) < 5 \\
\Pi(u, v) &= 7 \quad \text{if } c(u, v) = 1 \\
\Pi(u, v) &= 8 \quad \text{if } c(u, v) = 0 \text{ and } c(v) > 0 \\
\Pi(u, v) &= 9 \quad \text{otherwise}
\end{aligned}$$

Given a definition of the function $\Pi(u, v)$, we then introduce smoothing parameters $\lambda_1^{(k)}, \lambda_2^{(k)}, \lambda_3^{(k)}$ for all $k \in \{1 \ldots K\}$. Thus each bucket has its own set of smoothing parameters. We have the constraints that for all $k \in \{1 \ldots K\}$

$$\lambda_1^{(k)} \geq 0, \lambda_2^{(k)} \geq 0, \lambda_3^{(k)} \geq 0$$

and

$$\lambda_1^{(k)} + \lambda_2^{(k)} + \lambda_3^{(k)} = 1$$

The linearly interpolated estimate will be

$$q(w|u,v) = q(w|u,v) = \lambda_1^{(k)} \times q_{ML}(w|u,v) + \lambda_2^{(k)} \times q_{ML}(w|v) + \lambda_3^{(k)} \times q_{ML}(w)$$

where

$$k = \Pi(u,v)$$

Thus we have crucially introduced a dependence of the smoothing parameters on the value for $\Pi(u,v)$. Thus each bucket of bigrams gets its own set of smoothing parameters; the values for the smoothing parameters can vary depending on the value of $\Pi(u,v)$ (which is usually directly related to the counts $c(u,v)$ and $c(v)$).

The smoothing parameters are again estimated using a development data set. If we again define $c'(u,v,w)$ to be the number of times the trigram u,v,w appears in the development data, the log-likelihood of the development data is

$$\begin{aligned}
& \sum_{u,v,w} c'(u,v,w) \log q(w|u,v) \\
= & \sum_{u,v,w} c'(u,v,w) \log \left(\lambda_1^{(\Pi(u,v))} \times q_{ML}(w|u,v) + \lambda_2^{(\Pi(u,v))} \times q_{ML}(w|v) + \lambda_3^{(\Pi(u,v))} \times q_{ML}(w) \right) \\
= & \sum_{k=1}^{K} \sum_{\substack{u,v,w: \\ \Pi(u,v)=k}} c'(u,v,w) \log \left(\lambda_1^{(k)} \times q_{ML}(w|u,v) + \lambda_2^{(k)} \times q_{ML}(w|v) + \lambda_3^{(k)} \times q_{ML}(w) \right)
\end{aligned}$$

The $\lambda_1^{(k)}, \lambda_2^{(k)}, \lambda_3^{(k)}$ values are chosen to maximize this function.

Chapter 2

Tagging Problems, and Hidden Markov Models

(Course notes for NLP by Michael Collins, Columbia University)

2.1 Introduction

In many NLP problems, we would like to model *pairs* of sequences. Part-of-speech (POS) tagging is perhaps the earliest, and most famous, example of this type of problem. In POS tagging our goal is to build a model whose input is a *sentence*, for example

the dog saw a cat

and whose output is a *tag sequence*, for example

D N V D N (2.1)

(here we use D for a determiner, N for noun, and V for verb). The tag sequence is the same length as the input sentence, and therefore specifies a single tag for each word in the sentence (in this example D for *the*, N for *dog*, V for *saw*, and so on).

We will use $x_1 \dots x_n$ to denote the input to the tagging model: we will often refer to this as a *sentence*. In the above example we have the length $n = 5$, and $x_1 =$ *the*, $x_2 =$ *dog*, $x_3 =$ *saw*, $x_4 =$ *the*, $x_5 =$ *cat*. We will use $y_1 \dots y_n$ to denote the output of the tagging model: we will often refer to this as the *state sequence* or *tag sequence*. In the above example we have $y_1 =$ D, $y_2 =$ N, $y_3 =$ V, and so on.

This type of problem, where the task is to map a sentence $x_1 \dots x_n$ to a tag sequence $y_1 \dots y_n$, is often referred to as a **sequence labeling problem**, or a **tagging problem**.

INPUT:
Profits soared at Boeing Co., easily topping forecasts on Wall Street, as their CEO Alan Mulally announced first quarter results.

OUTPUT:
Profits/N soared/V at/P Boeing/N Co./N ,/, easily/ADV topping/V forecasts/N on/P Wall/N Street/N ,/, as/P their/POSS CEO/N Alan/N Mulally/N announced/V first/ADJ quarter/N results/N ./.

KEY:

N = Noun
V = Verb
P = Preposition
Adv = Adverb
Adj = Adjective
...

Figure 2.1: A part-of-speech (POS) tagging example. The input to the model is a sentence. The output is a tagged sentence, where each word is tagged with its part of speech: for example `N` is a noun, `V` is a verb, `P` is a preposition, and so on.

We will assume that we have a set of *training examples*, $(x^{(i)}, y^{(i)})$ for $i = 1 \ldots m$, where each $x^{(i)}$ is a sentence $x_1^{(i)} \ldots x_{n_i}^{(i)}$, and each $y^{(i)}$ is a tag sequence $y_1^{(i)} \ldots y_{n_i}^{(i)}$ (we assume that the i'th example is of length n_i). Hence $x_j^{(i)}$ is the j'th word in the i'th training example, and $y_j^{(i)}$ is the tag for that word. Our task is to learn a function that maps sentences to tag sequences from these training examples.

2.2 Two Example Tagging Problems: POS Tagging, and Named-Entity Recognition

We first discuss two important examples of tagging problems in NLP, part-of-speech (POS) tagging, and named-entity recognition.

Figure 2.1 gives an example illustrating the part-of-speech problem. The input to the problem is a sentence. The output is a tagged sentence, where each word in the sentence is annotated with its part of speech. Our goal will be to construct a model that recovers POS tags for sentences with high accuracy. POS tagging is one of the most basic problems in NLP, and is useful in many natural language applications.

We will assume that we have a set of training examples for the problem: that is, we have a set of sentences paired with their correct POS tag sequences. As one example, the Penn WSJ treebank corpus contains around 1 million words (around 40,000 sentences) annotated with their POS tags. Similar resources are available in many other languages and genres.

One of the main challenges in POS tagging is *ambiguity*. Many words in English can take several possible parts of speech—a similar observation is true for many other languages. The example sentence in figure 2.1 has several ambiguous words. For example, the first word in the sentence, *profits*, is a noun in this context, but can also be a verb (e.g., in *the company profits from its endeavors*). The word *topping* is a verb in this particular sentence, but can also be a noun (e.g., *the topping on the cake*). The words *forecasts* and *results* are both nouns in the sentence, but can also be verbs in other contexts. If we look further, we see that *quarter* is a noun in this sentence, but it also has a much less frequent usage, as a verb. We can see from this sentence that there is a surprising amount of ambiguity at the POS level.

A second challenge is the presence of words that are rare, in particular words that are not seen in our training examples. Even with say a million words of training data, there will be many words in new sentences which have not been seen in training. As one example, words such as *Mulally* or *topping* are potentially quite rare, and may not have been seen in our training examples. It will be important to develop methods that deal effectively with words which have not been seen in training data.

In recovering POS tags, it is useful to think of two different sources of information. First, individual words have statistical preferences for their part of speech: for example, *quarter* can be a noun or a verb, but is more likely to be a noun. Second, the context has an important effect on the part of speech for a word. In particular, some sequences of POS tags are much more likely than others. If we consider POS trigrams, the sequence `D N V` will be frequent in English (e.g., in *the/D dog/N saw/V* . . .), whereas the sequence `D V N` is much less likely.

Sometimes these two sources of evidence are in conflict: for example, in the sentence

The trash can is hard to find

the part of speech for *can* is a noun—however, *can* can also be a modal verb, and in fact it is much more frequently seen as a modal verb in English.[1] In this sentence the context has overridden the tendency for *can* to be a verb as opposed to a noun.

[1]There are over 30 uses of the word "can" in this chapter, and if we exclude the example given above, in every case "can" is used as a modal verb.

INPUT: Profits soared at Boeing Co., easily topping forecasts on Wall Street, as their CEO Alan Mulally announced first quarter results.

OUTPUT: Profits soared at [Company Boeing Co.], easily topping forecasts on [Location Wall Street], as their CEO [Person Alan Mulally] announced first quarter results.

Figure 2.2: A Named-Entity Recognition Example. The input to the problem is a sentence. The output is a sentence annotated with named-entities corresponding to companies, location, and people.

Later in this chapter we will describe models for the tagging problem that take into account both sources of information—local and contextual—when making tagging decisions.

A second important example tagging problem is named entity recognition. Figure 2.2 gives an example. For this problem the input is again a sentence. The output is the sentence with entity-boundaries marked. In this example we assume there are three possible entity types: `PERSON`, `LOCATION`, and `COMPANY`. The output in this example identifies *Boeing Co.* as a company, *Wall Street* as a location, and *Alan Mulally* as a person. Recognising entities such as people, locations and organizations has many applications, and named-entity recognition has been widely studied in NLP research.

At first glance the named-entity problem does not resemble a tagging problem—in figure 2.2 the output does not consist of a tagging decision for each word in the sentence. However, it is straightforward to map named-entity recognition to a tagging problem. The basic method is illustrated in figure 2.3. Each word in the sentence is either tagged as not being part of an entity (the tag `NA`) is used for this purpose, as being the start of a particular entity type (e.g., the tag `SC`) corresponds to words that are the first word in a company, or as being the continuation of a particular entity type (e.g., the tag `CC` corresponds to words that are part of a company name, but are not the first word).

Once this mapping has been performed on training examples, we can train a tagging model on these training examples. Given a new test sentence we can then recover the sequence of tags from the model, and it is straightforward to identify the entities identified by the model.

2.3 Generative Models, and The Noisy Channel Model

In this chapter we will treat tagging problems as a supervised learning problem. In this section we describe one important class of model for supervised learning:

INPUT: Profits soared at Boeing Co., easily topping forecasts on Wall Street, as their CEO Alan Mulally announced first quarter results.

OUTPUT: Profits/NA soared/NA at/NA Boeing/SC Co./CC ,/NA easily/NA topping/NA forecasts/NA on/NA Wall/SL Street/CL ,/NA as/NA their/NA CEO/NA Alan/SP Mulally/CP announced/NA first/NA quarter/NA results/NA ./NA

KEY:

NA = No entity
SC = Start Company
CC = Continue Company
SL = Start Location
CL = Continue Location
. . .

Figure 2.3: Named-Entity Recognition as a Tagging Problem. There are three entity types: `PERSON`, `LOCATION`, and `COMPANY`. For each entity type we introduce a tag for the start of that entity type, and for the continuation of that entity type. The tag `NA` is used for words which are not part of an entity. We can then represent the named-entity output in figure 2.2 as a sequence of tagging decisions using this tag set.

the class of *generative models*. We will then go on to describe a particular type of generative model, *hidden Markov models*, applied to the tagging problem.

The set-up in supervised learning problems is as follows. We assume training examples $(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$, where each example consists of an input $x^{(i)}$ paired with a label $y^{(i)}$. We use $\mathcal{X}$ to refer to the set of possible inputs, and $\mathcal{Y}$ to refer to the set of possible labels. Our task is to learn a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that maps any input x to a label $f(x)$.

Many problems in natural language processing are supervised learning problems. For example, in tagging problems each $x^{(i)}$ would be a sequence of words $x_1^{(i)} \dots x_{n_i}^{(i)}$, and each $y^{(i)}$ would be a sequence of tags $y_1^{(i)} \dots y_{n_i}^{(i)}$ (we use n_i to refer to the length of the i'th training example). $\mathcal{X}$ would refer to the set of all sequences $x_1 \dots x_n$, and $\mathcal{Y}$ would be the set of all tag sequences $y_1 \dots y_n$. Our task would be to learn a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that maps sentences to tag sequences. In machine translation, each input x would be a sentence in the source language (e.g., Chinese), and each "label" would be a sentence in the target language (e.g., English). In speech recognition each input would be the recording of some utterance—perhaps pre-processed using a Fourier transform, for example—and each label is an entire sentence. Our task in all of these examples is to learn a function from inputs x to labels y, using our training examples $(x^{(i)}, y^{(i)})$ for $i = 1 \dots n$ as evidence.

One way to define the function $f(x)$ is through a *conditional model*. In this approach we define a model that defines the conditional probability

$$p(y|x)$$

for any x, y pair. The parameters of the model are estimated from the training examples. Given a new test example x, the output from the model is

$$f(x) = \arg\max_{y \in \mathcal{Y}} p(y|x)$$

Thus we simply take the most likely label y as the output from the model. If our model $p(y|x)$ is close to the true conditional distribution of labels given inputs, the function $f(x)$ will be close to optimal.

An alternative approach, which is often used in machine learning and natural language processing, is to define a *generative model*. Rather than directly estimating the conditional distribution $p(y|x)$, in generative models we instead model the *joint* probability

$$p(x, y)$$

over (x, y) pairs. The parameters of the model $p(x, y)$ are again estimated from the training examples $(x^{(i)}, y^{(i)})$ for $i = 1 \dots n$. In many cases we further decompose

the probability $p(x, y)$ as follows:

$$p(x, y) = p(y)p(x|y) \tag{2.2}$$

and then estimate the models for $p(y)$ and $p(x|y)$ separately. These two model components have the following interpretations:

- $p(y)$ is a *prior* probability distribution over labels y.
- $p(x|y)$ is the probability of generating the input x, given that the underlying label is y.

We will see that in many cases it is very convenient to decompose models in this way; for example, the classical approach to speech recognition is based on this type of decomposition.

Given a generative model, we can use Bayes rule to derive the conditional probability $p(y|x)$ for any (x, y) pair:

$$p(y|x) = \frac{p(y)p(x|y)}{p(x)}$$

where

$$p(x) = \sum_{y \in \mathcal{Y}} p(x, y) = \sum_{y \in \mathcal{Y}} p(y)p(x|y)$$

Thus the joint model is quite versatile, in that we can also derive the probabilities $p(x)$ and $p(y|x)$.

We use Bayes rule directly in applying the joint model to a new test example. Given an input x, the output of our model, $f(x)$, can be derived as follows:

$$\begin{aligned} f(x) &= \arg\max_y p(y|x) \\ &= \arg\max_y \frac{p(y)p(x|y)}{p(x)} \qquad (2.3) \\ &= \arg\max_y p(y)p(x|y) \qquad (2.4) \end{aligned}$$

Eq. 2.3 follows by Bayes rule. Eq. 2.4 follows because the denominator, $p(x)$, does not depend on y, and hence does not affect the $\arg\max$. This is convenient, because it means that we do not need to calculate $p(x)$, which can be an expensive operation.

Models that decompose a joint probability into into terms $p(y)$ and $p(x|y)$ are often called *noisy-channel* models. Intuitively, when we see a test example x, we assume that has been generated in two steps: first, a label y has been chosen with

probability $p(y)$; second, the example x has been generated from the distribution $p(x|y)$. The model $p(x|y)$ can be interpreted as a "channel" which takes a label y as its input, and corrupts it to produce x as its output. Our task is to find the most likely label y, given that we observe x.

In summary:

- Our task is to learn a function from inputs x to labels $y = f(x)$. We assume training examples $(x^{(i)}, y^{(i)})$ for $i = 1 \dots n$.
- In the noisy channel approach, we use the training examples to estimate models $p(y)$ and $p(x|y)$. These models define a joint (generative) model

$$p(x, y) = p(y)p(x|y)$$

- Given a new test example x, we predict the label

$$f(x) = \arg\max_{y \in \mathcal{Y}} p(y)p(x|y)$$

  Finding the output $f(x)$ for an input x is often referred to as the *decoding* problem.

2.4 Generative Tagging Models

We now see how generative models can be applied to the tagging problem. We assume that we have a finite vocabulary $\mathcal{V}$, for example $\mathcal{V}$ might be the set of words seen in English, e.g., $\mathcal{V} = \{$*the, dog, saw, cat, laughs*$, \dots\}$. We use $\mathcal{K}$ to denote the set of possible tags; again, we assume that this set is finite. We then give the following definition:

Definition 1 (Generative Tagging Models) *Assume a finite set of words $\mathcal{V}$, and a finite set of tags $\mathcal{K}$. Define $\mathcal{S}$ to be the set of all sequence/tag-sequence pairs $\langle x_1 \dots x_n, y_1 \dots y_n \rangle$ such that $n \geq 0$, $x_i \in \mathcal{V}$ for $i = 1 \dots n$, and $y_i \in \mathcal{K}$ for $i = 1 \dots n$. A* generative tagging model *is then a function p such that:*

1. *For any $\langle x_1 \dots x_n, y_1 \dots y_n \rangle \in \mathcal{S}$,*

$$p(x_1 \dots x_n, y_1 \dots y_n) \geq 0$$

2. *In addition,*

$$\sum_{\langle x_1 \dots x_n, y_1 \dots y_n \rangle \in \mathcal{S}} p(x_1 \dots x_n, y_1 \dots y_n) = 1$$

Hence $p(x_1 \ldots x_n, y_1 \ldots y_n)$ is a probability distribution over pairs of sequences (i.e., a probability distribution over the set $\mathcal{S}$).

Given a generative tagging model, the function from sentences $x_1 \ldots x_n$ to tag sequences $y_1 \ldots y_n$ is defined as

$$f(x_1 \ldots x_n) = \arg\max_{y_1 \ldots y_n} p(x_1 \ldots x_n, y_1 \ldots y_n)$$

where the $\arg\max$ *is taken over all sequences $y_1 \ldots y_n$ such that $y_i \in \mathcal{K}$ for $i \in \{1 \ldots n\}$. Thus for any input $x_1 \ldots x_n$, we take the highest probability tag sequence as the output from the model.* □

Having introduced generative tagging models, there are three critical questions:

- How we define a generative tagging model $p(x_1 \ldots x_n, y_1 \ldots y_n)$?
- How do we estimate the parameters of the model from training examples?
- How do we efficiently find

  $$\arg\max_{y_1 \ldots y_n} p(x_1 \ldots x_n, y_1 \ldots y_n)$$

  for any input $x_1 \ldots x_n$?

The next section describes how trigram hidden Markov models can be used to answer these three questions.

2.5 Trigram Hidden Markov Models (Trigram HMMs)

In this section we describe an important type of generative tagging model, a *trigram hidden Markov model*, describe how the parameters of the model can be estimated from training examples, and describe how the most likely sequence of tags can be found for any sentence.

2.5.1 Definition of Trigram HMMs

We now give a formal definition of trigram hidden Markov models (trigram HMMs). The next section shows how this model form is derived, and gives some intuition behind the model.

Definition 2 (Trigram Hidden Markov Model (Trigram HMM)) *A trigram HMM consists of a finite set $\mathcal{V}$ of possible words, and a finite set $\mathcal{K}$ of possible tags, together with the following parameters:*

- *A parameter*

$$q(s|u,v)$$

for any trigram (u,v,s) *such that* $s \in \mathcal{K} \cup \{STOP\}$*, and* $u,v \in \mathcal{K} \cup \{*\}$*. The value for* $q(s|u,v)$ *can be interpreted as the probability of seeing the tag* s *immediately after the bigram of tags* (u,v)*.*

- *A parameter*

$$e(x|s)$$

for any $x \in \mathcal{V}$*,* $s \in \mathcal{K}$*. The value for* $e(x|s)$ *can be interpreted as the probability of seeing observation* x *paired with state* s*.*

Define $\mathcal{S}$ *to be the set of all sequence/tag-sequence pairs* $\langle x_1 \dots x_n, y_1 \dots y_{n+1} \rangle$ *such that* $n \geq 0$*,* $x_i \in \mathcal{V}$ *for* $i = 1 \dots n$*,* $y_i \in \mathcal{K}$ *for* $i = 1 \dots n$*, and* $y_{n+1} = STOP$*.*

We then define the probability for any $\langle x_1 \dots x_n, y_1 \dots y_{n+1} \rangle \in \mathcal{S}$ *as*

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i|y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i|y_i)$$

where we have assumed that $y_0 = y_{-1} = *$*.* $\square$

As one example, if we have $n = 3$, $x_1 \dots x_3$ equal to the sentence *the dog laughs*, and $y_1 \dots y_4$ equal to the tag sequence `D N V STOP`, then

$$\begin{aligned} p(x_1 \dots x_n, y_1 \dots y_{n+1}) \quad = \quad & q(\texttt{D}|*,*) \times q(\texttt{N}|*,\texttt{D}) \times q(\texttt{V}|\texttt{D},\texttt{N}) \times q(\texttt{STOP}|\texttt{N},\texttt{V}) \\ & \times e(\textit{the}|\texttt{D}) \times e(\textit{dog}|\texttt{N}) \times e(\textit{laughs}|\texttt{V}) \end{aligned}$$

Note that this model form is a noisy-channel model. The quantity

$$q(\texttt{D}|*,*) \times q(\texttt{N}|*,\texttt{D}) \times q(\texttt{V}|\texttt{D},\texttt{N}) \times q(\texttt{STOP}|\texttt{N},\texttt{V})$$

is the prior probability of seeing the tag sequence `D N V STOP`, where we have used a second-order Markov model (a trigram model), very similar to the language models we derived in the previous lecture. The quantity

$$e(\textit{the}|\texttt{D}) \times e(\textit{dog}|\texttt{N}) \times e(\textit{laughs}|\texttt{V})$$

can be interpreted as the conditional probability $p(\textit{the dog laughs}|\texttt{D N V STOP})$: that is, the conditional probability $p(x|y)$ where x is the sentence *the dog laughs*, and y is the tag sequence `D N V STOP`.

2.5.2 Independence Assumptions in Trigram HMMs

We now describe how the form for trigram HMMs can be derived: in particular, we describe the independence assumptions that are made in the model. Consider a pair of sequences of random variables $X_1 \dots X_n$, and $Y_1 \dots Y_n$, where n is the length of the sequences. We assume that each X_i can take any value in a finite set $\mathcal{V}$ of *words*. For example, $\mathcal{V}$ might be a set of possible words in English, for example $\mathcal{V} = \{$*the, dog, saw, cat, laughs*$, \dots\}$. Each Y_i can take any value in a finite set $\mathcal{K}$ of possible *tags*. For example, $\mathcal{K}$ might be the set of possible part-of-speech tags for English, e.g. $\mathcal{K} = \{\texttt{D}, \texttt{N}, \texttt{V}, \dots\}$.

The length n is itself a random variable—it can vary across different sentences—but we will use a similar technique to the method used for modeling variable-length Markov processes (see chapter **??**).

Our task will be to model the joint probability

$$P(X_1 = x_1 \dots X_n = x_n, Y_1 = y_1 \dots Y_n = y_n)$$

for any observation sequence $x_1 \dots x_n$ paired with a state sequence $y_1 \dots y_n$, where each x_i is a member of $\mathcal{V}$, and each y_i is a member of $\mathcal{K}$.

We will find it convenient to define one additional random variable Y_{n+1}, which always takes the value STOP. This will play a similar role to the STOP symbol seen for variable-length Markov sequences, as described in the previous lecture notes.

The key idea in hidden Markov models is the following definition:

$$\begin{aligned} & P(X_1 = x_1 \dots X_n = x_n, Y_1 = y_1 \dots Y_{n+1} = y_{n+1}) \\ = & \prod_{i=1}^{n+1} P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \prod_{i=1}^{n} P(X_i = x_i | Y_i = y_i) \end{aligned} \quad (2.5)$$

where we have assumed that $y_0 = y_{-1} =$ *, where * is a special start symbol.

Note the similarity to our definition of trigram HMMs. In trigram HMMs we have made the assumption that the joint probability factorizes as in Eq. 2.5, and in addition we have assumed that for any i, for any values of y_{i-2}, y_{i-1}, y_i,

$$P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) = q(y_i | y_{i-2}, y_{i-1})$$

and that for any value of i, for any values of x_i and y_i,

$$P(X_i = x_i | Y_i = y_i) = e(x_i | y_i)$$

We now describe how Eq. 2.5 is derived, in particular focusing on independence assumptions that have been made in the model. First, we can write

$$\begin{aligned} & P(X_1 = x_1 \dots X_n = x_n, Y_1 = y_1 \dots Y_{n+1} = y_{n+1}) \\ = & P(Y_1 = y_1 \dots Y_{n+1} = y_{n+1}) \times P(X_1 = x_1 \dots X_n = x_n | Y_1 = y_1 \dots Y_{n+1} = y_{n+1}) \end{aligned} \quad (2.6)$$

This step is exact, by the chain rule of probabilities. Thus we have decomposed the joint probability into two terms: first, the probability of choosing tag sequence $y_1 \ldots y_{n+1}$; second, the probability of choosing the word sequence $x_1 \ldots x_n$, conditioned on the choice of tag sequence. Note that this is exactly the same type of decomposition as seen in noisy channel models.

Now consider the probability of seeing the tag sequence $y_1 \ldots y_{n+1}$. We make independence assumptions as follows: we assume that for any sequence $y_1 \ldots y_{n+1}$,

$$P(Y_1 = y_1 \ldots Y_{n+1} = y_{n+1}) = \prod_{i=1}^{n+1} P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1})$$

That is, we have assumed that the sequence $Y_1 \ldots Y_{n+1}$ is a second-order Markov sequence, where each state depends only on the previous two states in the sequence.

Next, consider the probability of the word sequence $x_1 \ldots x_n$, conditioned on the choice of tag sequence, $y_1 \ldots y_{n+1}$. We make the following assumption:

$$\begin{aligned} & P(X_1 = x_1 \ldots X_n = x_n | Y_1 = y_1 \ldots Y_{n+1} = y_{n+1}) \\ = & \prod_{i=1}^{n} P(X_i = x_i | X_1 = x_1 \ldots X_{i-1} = x_{i-1}, Y_1 = y_1 \ldots Y_{n+1} = y_{n+1}) \\ = & \prod_{i=1}^{n} P(X_i = x_i | Y_i = y_i) \end{aligned} \tag{2.7}$$

The first step of this derivation is exact, by the chain rule. The second step involves an independence assumption, namely that for $i = 1 \ldots n$,

$$P(X_i = x_i | X_1 = x_1 \ldots X_{i-1} = x_{i-1}, Y_1 = y_1 \ldots Y_{n+1} = y_{n+1}) = P(X_i = x_i | Y_i = y_i)$$

Hence we have assumed that the value for the random variable X_i depends only on the value of Y_i. More formally, the value for X_i is conditionally independent of the previous observations $X_1 \ldots X_{i-1}$, and the other state values $Y_1 \ldots Y_{i-1}, Y_{i+1} \ldots Y_{n+1}$, given the value of Y_i.

One useful way of thinking of this model is to consider the following stochastic process, which generates sequence pairs $y_1 \ldots y_{n+1}$, $x_1 \ldots x_n$:

1. Initialize $i = 1$ and $y_0 = y_{-1} =$ *.
2. Generate y_i from the distribution
$$q(y_i | y_{i-2}, y_{i-1})$$
3. If $y_i =$ STOP then return $y_1 \ldots y_i$, $x_1 \ldots x_{i-1}$. Otherwise, generate x_i from the distribution
$$e(x_i | y_i),$$
set $i = i + 1$, and return to step 2.

2.5.3 Estimating the Parameters of a Trigram HMM

We will assume that we have access to some training data. The training data consists of a set of examples where each example is a sentence $x_1 \dots x_n$ paired with a tag sequence $y_1 \dots y_n$. Given this data, how do we estimate the parameters of the model? We will see that there is a simple and very intuitive answer to this question.

Define $c(u, v, s)$ to be the number of times the sequence of three states (u, v, s) is seen in training data: for example, $c(\text{V, D, N})$ would be the number of times the sequence of three tags V, D, N is seen in the training corpus. Similarly, define $c(u, v)$ to be the number of times the tag bigram (u, v) is seen. Define $c(s)$ to be the number of times that the state s is seen in the corpus. Finally, define $c(s \rightsquigarrow x)$ to be the number of times state s is seen paired sith observation x in the corpus: for example, $c(\text{N} \rightsquigarrow \text{dog})$ would be the number of times the word *dog* is seen paired with the tag N.

Given these definitions, the *maximum-likelihood* estimates are

$$q(s|u, v) = \frac{c(u, v, s)}{c(u, v)}$$

and

$$e(x|s) = \frac{c(s \rightsquigarrow x)}{c(s)}$$

For example, we would have the estimates

$$q(\texttt{N}|\texttt{V}, \texttt{D}) = \frac{c(\texttt{V}, \texttt{D}, \texttt{N})}{c(\texttt{V}, \texttt{D})}$$

and

$$e(\textit{dog}|\texttt{N}) = \frac{c(\texttt{N} \rightsquigarrow \textit{dog})}{c(\texttt{N})}$$

Thus estimating the parameters of the model is simple: we just read off counts from the training corpus, and then compute the maximum-likelihood estimates as described above.

In some cases it is useful to smooth our estimates of $q(s|u, v)$, using the techniques described in chapter **??** of this book, for example defining

$$q(s|u, v) = \lambda_1 \times q_{ML}(s|u, v) + \lambda_2 \times q_{ML}(s|v) + \lambda_3 \times q_{ML}(s)$$

where the q_{ML} terms are maximum-likelihood estimates derived from counts in the corpus, and $\lambda_1, \lambda_2, \lambda_3$ are smoothing parameters satisfying $\lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0$, and $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

One problem with these estimates is that the value for $e(x|s)$ will be unreliable if the word x is infrequent: worse still, we have $e(x|s) = 0$ if the word x is not seen in the training data. A solution to this problem is described in section 2.7.1.

2.5.4 Decoding with HMMs: the Viterbi Algorithm

We now turn to the problem of finding the most likely tag sequence for an input sentence $x_1 \dots x_n$. This is the problem of finding

$$\arg \max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

where the $\arg \max$ is taken over all sequences $y_1 \dots y_{n+1}$ such that $y_i \in \mathcal{K}$ for $i = 1 \dots n$, and $y_{n+1} =$ STOP. We assume that p again takes the form

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i | y_i) \tag{2.8}$$

Recall that we have assumed in this definition that $y_0 = y_{-1} =$ *, and $y_{n+1} =$ STOP.

The naive, brute force method would be to simply enumerate all possible tag sequences $y_1 \dots y_{n+1}$, score them under the function p, and take the highest scoring sequence. For example, given the input sentence

the dog barks

and assuming that the set of possible tags is $\mathcal{K} = \{\texttt{D}, \texttt{N}, \texttt{V}\}$, we would consider all possible tag sequences:

D	D	D	STOP
D	D	N	STOP
D	D	V	STOP
D	N	D	STOP
D	N	N	STOP
D	N	V	STOP
...			

and so on. There are $3^3 = 27$ possible sequences in this case.

For longer sentences, however, this method will be hopelessly inefficient. For an input sentence of length n, there are $|\mathcal{K}|^n$ possible tag sequences. The exponential growth with respect to the length n means that for any reasonable length sentence, brute-force search will not be tractable.

The Basic Algorithm

Instead, we will see that we can efficiently find the highest probability tag sequence, using a dynamic programming algorithm that is often called *the Viterbi*

algorithm. The input to the algorithm is a sentence $x_1 \dots x_n$. Given this sentence, for any $k \in \{1 \dots n\}$, for any sequence $y_{-1}, y_0, y_1, \dots, y_k$ such that $y_i \in \mathcal{K}$ for $i = 1 \dots k$, and $y_{-1} = y_0 = *$, we define the function

$$r(y_{-1}, y_0, y_1, \dots, y_k) = \prod_{i=1}^{k} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^{k} e(x_i | y_i) \tag{2.9}$$

This is simply a truncated version of the definition of p in Eq. 2.8, where we just consider the first k terms. In particular, note that

$$\begin{aligned} p(x_1 \dots x_n, y_1 \dots y_{n+1}) &= r(*, *, y_1, \dots, y_n) \times q(y_{n+1} | y_{n-1}, y_n) \\ &= r(*, *, y_1, \dots, y_n) \times q(\text{STOP} | y_{n-1}, y_n) \end{aligned} \tag{2.10}$$

It will be convenient to use $\mathcal{K}_k$ for $k \in \{-1 \dots n\}$ to denote the set of allowable tags at position k in the sequence: more precisely, define

$$\mathcal{K}_{-1} = \mathcal{K}_o = \{*\}$$

and

$$\mathcal{K}_k = \mathcal{K} \quad \text{for } k \in \{1 \dots n\}$$

Next, for any $k \in \{1 \dots n\}$, for any $u \in \mathcal{K}_{k-1}$, $v \in \mathcal{K}_k$, define $S(k, u, v)$ to be the set of sequences $y_{-1}, y_0, y_1, \dots, y_k$ such that $y_{k-1} = u$, $y_k = v$, and $y_i \in \mathcal{K}_i$ for $i \in \{-1 \dots k\}$. Thus $S(k, u, v)$ is the set of all tag sequences of length k, which end in the tag bigram (u, v). Define

$$\pi(k, u, v) = \max_{\langle y_{-1}, y_0, y_1, \dots, y_k \rangle \in S(k,u,v)} r(y_{-1}, y_0, y_1, \dots, y_k) \tag{2.11}$$

Thus $\pi(k, u, v)$ is the maximum probability for any sequence of length k, ending in the tag bigram (u, v).

We now observe that we can calculate the $\pi(k, u, v)$ values for all (k, u, v) efficiently, as follows. First, as a base case define

$$\pi(0, *, *) = 1$$

Next, we give the recursive definition.

Proposition 1 *For any $k \in \{1 \dots n\}$, for any $u \in \mathcal{K}_{k-1}$ and $v \in \mathcal{K}_k$, we can use the following recursive definition:*

$$\pi(k, u, v) = \max_{w \in \mathcal{K}_{k-2}} (\pi(k-1, w, u) \times q(v | w, u) \times e(x_k | v)) \tag{2.12}$$

This definition is recursive because the definition makes use of the $\pi(k-1, w, u)$ values computed for shorter sequences. This definition will be key to our dynamic programming algorithm.

How can we justify this recurrence? Recall that $\pi(k, u, v)$ is the highest probability for any sequence $y_{-1} \dots y_k$ ending in the bigram (u, v). Any such sequence must have $y_{k-2} = w$ for some state w. The highest probability for any sequence of length $k-1$ ending in the bigram (w, u) is $\pi(k-1, w, u)$, hence the highest probability for any sequence of length k ending in the trigram (w, u, v) must be

$$\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v)$$

In Eq. 2.12 we simply search over all possible values for w, and return the maximum.

Our second claim is the following:

Proposition 2

$$\max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \max_{u \in \mathcal{K}_{n-1}, v \in \mathcal{K}_n} (\pi(n, u, v) \times q(\textit{STOP}|u, v)) \tag{2.13}$$

This follows directly from the identity in Eq. 2.10.

Figure 2.4 shows an algorithm that puts these ideas together. The algorithm takes a sentence $x_1 \dots x_n$ as input, and returns

$$\max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

as its output. The algorithm first fills in the $\pi(k, u, v)$ values in using the recursive definition. It then uses the identity in Eq. 2.13 to calculate the highest probability for any sequence.

The running time for the algorithm is $O(n|\mathcal{K}|^3)$, hence it is linear in the length of the sequence, and cubic in the number of tags.

The Viterbi Algorithm with Backpointers

The algorithm we have just described takes a sentence $x_1 \dots x_n$ as input, and returns

$$\max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

as its output. However we'd really like an algorithm that returned the highest probability sequence, that is, an algorithm that returns

$$\arg \max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

Input: a sentence $x_1 \dots x_n$, parameters $q(s|u,v)$ and $e(x|s)$.
Definitions: Define $\mathcal{K}$ to be the set of possible tags. Define $\mathcal{K}_{-1} = \mathcal{K}_0 = \{*\}$, and $\mathcal{K}_k = \mathcal{K}$ for $k = 1 \dots n$.
Initialization: Set $\pi(0, *, *) = 1$.
Algorithm:

- For $k = 1 \dots n$,
  - For $u \in \mathcal{K}_{k-1}$, $v \in \mathcal{K}_k$,

$$\pi(k, u, v) = \max_{w \in \mathcal{K}_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

- **Return** $\max_{u \in \mathcal{K}_{n-1}, v \in \mathcal{K}_n} (\pi(n, u, v) \times q(\text{STOP}|u, v))$

Figure 2.4: The basic Viterbi Algorithm.

for any input sentence $x_1 \dots x_n$.

Figure 2.5 shows a modified algorithm that achieves this goal. The key step is to store backpointer values $bp(k, u, v)$ at each step, which record the previous state w which leads to the highest scoring sequence ending in (u, v) at position k (the use of backpointers such as these is very common in dynamic programming methods). At the end of the algorithm, we unravel the backpointers to find the highest probability sequence, and then return this sequence. The algorithm again runs in $O(n|\mathcal{K}|^3)$ time.

2.6 Summary

We've covered a number of important points in this chapter, but the end result is fairly straightforward: we have derived a complete method for learning a tagger from a training corpus, and for applying it to new sentences. The main points were as follows:

- A trigram HMM has parameters $q(s|u,v)$ and $e(x|s)$, and defines the joint probability of any sentence $x_1 \dots x_n$ paired with a tag sequence $y_1 \dots y_{n+1}$ (where $y_{n+1} = \text{STOP}$) as

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i|y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i|y_i)$$

Input: a sentence $x_1 \dots x_n$, parameters $q(s|u,v)$ and $e(x|s)$.
Definitions: Define $\mathcal{K}$ to be the set of possible tags. Define $\mathcal{K}_{-1} = \mathcal{K}_0 = \{*\}$, and $\mathcal{K}_k = \mathcal{K}$ for $k = 1 \dots n$.
Initialization: Set $\pi(0, *, *) = 1$.
Algorithm:

- For $k = 1 \dots n$,
  - For $u \in \mathcal{K}_{k-1}$, $v \in \mathcal{K}_k$,

$$\pi(k, u, v) = \max_{w \in \mathcal{K}_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

$$bp(k, u, v) = \arg \max_{w \in \mathcal{K}_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

- Set $(y_{n-1}, y_n) = \arg\max_{u \in \mathcal{K}_{n-1}, v \in \mathcal{K}_n} (\pi(n, u, v) \times q(\text{STOP}|u, v))$
- For $k = (n-2) \dots 1$,

$$y_k = bp(k+2, y_{k+1}, y_{k+2})$$

- **Return** the tag sequence $y_1 \dots y_n$

Figure 2.5: The Viterbi Algorithm with backpointers.

- Given a training corpus from which we can derive counts, the maximum-likelihood estimates for the parameters are

$$q(s|u,v) = \frac{c(u,v,s)}{c(u,v)}$$

and

$$e(x|s) = \frac{c(s \rightsquigarrow x)}{c(s)}$$

- Given a new sentence $x_1 \dots x_n$, and parameters q and e that we have estimated from a training corpus, we can find the highest probability tag sequence for $x_1 \dots x_n$ using the algorithm in figure 2.5 (the Viterbi algorithm).

2.7 Advanced Material

2.7.1 Dealing with Unknown Words

Recall that our parameter estimates for the emission probabilities in the HMM are

$$e(x|s) = \frac{c(s \rightsquigarrow x)}{c(s)}$$

where $c(s \rightsquigarrow x)$ is the number of times state s is paired with word x in the training data, and $c(s)$ is the number of times state s is seen in training data.

A major issue with these estimates is that for any word x that is not seen in training data, $e(x|s)$ will be equal to 0 for all states s. Because of this, for any test sentence $x_1 \dots x_n$ that contains some word that is never seen in training data, it is easily verified that

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = 0$$

for *all* tag sequences $y_1 \dots y_{n+1}$. Thus the model will completely fail on the test sentence. In particular, the $\arg\max$ in

$$\arg \max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1}) = 0$$

will not be useful: every tag sequence will have the same, maximum score, of 0.

This is an acute problem, because however large our training data, there will inevitably be words in test sentences that are never seen in training data. The vocabulary size for English, for example, is very large; and new words are always being encountered in test data. Take for example the sentence used in figures 2.2 and 2.3:

Profits soared at Boeing Co., easily topping forecasts on Wall Street, as their CEO Alan Mulally announced first quarter results.

In this sentence it is quite likely that the word *Mulally* has not been seen in training data. Similarly, *topping* is a relatively infrequent word in English, and may not have been seen in training.

In this section we describe a simple but quite effective solution to this problem. The key idea is to map low frequency words in training data, and in addition words seen in test data but never seen in training, to a relatively small set of pseudo-words. For example, we might map the word *Mulally* to the pseudo-word `initCap`, the word *1990* to the pseudo-word `fourDigitNum`, and so on. Here the pseudo-word `initCap` is used for any word whose first letter is a capital, and whose remaining letters are lower case. The pseudo-word `fourDigitNum` is used for any four digit number.

Figure 2.6 shows an example set of pseudo-words, taken from [**?**], who applied an HMM tagger to the problem of named entity recognition. This set of pseudo-words was chosen by hand, and was clearly chosen to preserve some useful information about the spelling features of different words: for example capitalization features of words, and a sub-division into different number types (one of the entity classes identified in this work was dates, so it is useful to distinguish different types of numbers, as numbers are often relevant to dates).

Once a mapping from words to pseudo-words is defined we procede as follows. Define $f(x)$ to be the function that maps a word x to its pseudo-word $f(x)$. We define some count cut-off γ: a typical value for γ might be $\gamma = 5$. For any word seen in training data less than γ times, we simply replace the word x by its pseudo-word $f(x)$. This mapping is applied to words in both training and test examples: so words which are never seen in training data, but which are seen in test data, are also mapped to their pseudo-word. Once this mapping has been performed, we can estimate the parameters of the HMM in exactly the same way as before, with some of our words in training data now being pseudo-words. Similarly, we can apply the Viterbi algorithm for decoding with the model, with some of the words in our input sentences being pseudo-words.

Mapping low-frequency words to pseudo-words has the effect of "closing the vocabulary": with this mapping, every word in test data will be seen at least once in training data (assuming that each pseudo-word is seen at least once in training, which is generally the case). Thus we will never have the problem that $e(x|s) = 0$ for some word x in test data. In addition, with a careful choice for the set of pseudo-words, important information about the spelling of different words will be preserved. See figure 2.7 for an example sentence before and after the mapping is applied.

Word class	Example	Intuition
twoDigitNum	90	Two digit year
fourDigitNum	1990	Four digit year
containsDigitAndAlpha	A8956-67	Product code
containsDigitAndDash	09-96	Date
containsDigitAndSlash	11/9/89	Date
containsDigitAndComma	23,000.00	Monetary amount
containsDigitAndPeriod	1.00	Monetary amount,percentage
othernum	456789	Other number
allCaps	BBN	Organization
capPeriod	M.	Person name initial
firstWord	first word of sentence	no useful capitalization information
initCap	Sally	Capitalized word
lowercase	can	Uncapitalized word
other	,	Punctuation marks, all other words

Figure 2.6: The mapping to pseudo words used by [**?**] Bikel et. al (1999).

A drawback of the approach is that some care is needed in defining the mapping to pseudo-words: and this mapping may vary depending on the task being considered (for example different mappings might be used for named-entity recognition versus POS tagging). In a later chapter we will see what is arguably a cleaner solution to the problem of low frequency and unknown words, building on ideas from log-linear models.

Profits/NA soared/NA at/NA Boeing/SC Co./CC ,/NA easily/NA topping/NA forecasts/NA on/NA Wall/SL Street/CL ,/NA as/NA their/NA CEO/NA Alan/SP Mulally/CP announced/NA first/NA quarter/NA results/NA ./NA

⇓

`firstword`/NA soared/NA at/NA `initCap`/SC Co./CC ,/NA easily/NA `lowercase`/NA forecasts/NA on/NA `initCap`/SL Street/CL ,/NA as/NA their/NA CEO/NA Alan/SP `initCap`/CP announced/NA first/NA quarter/NA results/NA ./NA

NA = No entity
SC = Start Company
CC = Continue Company
SL = Start Location
CL = Continue Location
...

Figure 2.7: An example of how the pseudo-word mapping shown in figure 2.6 is applied to a sentence. Here we are assuming that *Profits*, *Boeing*, *topping*, *Wall*, and *Mullaly* are all seen infrequently enough to be replaced by their pseudo word. We show the sentence before and after the mapping.

Probabilistic Context-Free Grammars (PCFGs)

Michael Collins

1 Context-Free Grammars

1.1 Basic Definition

A context-free grammar (CFG) is a 4-tuple $G = (N, \Sigma, R, S)$ where:

- N is a finite set of non-terminal symbols.
- Σ is a finite set of terminal symbols.
- R is a finite set of rules of the form $X \rightarrow Y_1 Y_2 \ldots Y_n$, where $X \in N$, $n \geq 0$, and $Y_i \in (N \cup \Sigma)$ for $i = 1 \ldots n$.
- $S \in N$ is a distinguished start symbol.

Figure 1 shows a very simple context-free grammar, for a fragment of English. In this case the set of non-terminals N specifies some basic syntactic categories: for example S stands for "sentence", NP for "noun phrase", VP for "verb phrase", and so on. The set Σ contains the set of words in the vocabulary. The start symbol in this grammar is S: as we will see, this specifies that every parse tree has S as its root. Finally, we have context-free rules such as

$$\texttt{S} \rightarrow \texttt{NP VP}$$

or

$$\texttt{NN} \rightarrow \texttt{man}$$

The first rule specifies that an S (sentence) can be composed of an NP followed by a VP. The second rule specifies that an NN (a singular noun) can be composed of the word man.

Note that the set of allowable rules, as defined above, is quite broad: we can have any rule $X \rightarrow Y_1 \ldots Y_n$ as long as X is a member of N, and each Y_i for

$i = 1 \ldots n$ is a member of either N or Σ. We can for example have "unary rules", where $n = 1$, such as the following:

$$\begin{aligned} \texttt{NN} &\rightarrow \texttt{man} \\ \texttt{S} &\rightarrow \texttt{VP} \end{aligned}$$

We can also have rules that have a mixture of terminal and non-terminal symbols on the right-hand-side of the rule, for example

$$\begin{aligned} \texttt{VP} &\rightarrow \texttt{John Vt Mary} \\ \texttt{NP} &\rightarrow \texttt{the NN} \end{aligned}$$

We can even have rules where $n = 0$, so that there are no symbols on the right-hand-side of the rule. Examples are

$$\begin{aligned} \texttt{VP} &\rightarrow \epsilon \\ \texttt{NP} &\rightarrow \epsilon \end{aligned}$$

Here we use ϵ to refer to the empty string. Intuitively, these latter rules specify that a particular non-terminal (e.g., VP), is allowed to have no words below it in a parse tree.

1.2 (Left-most) Derivations

Given a context-free grammar G, a left-most derivation is a sequence of strings $s_1 \ldots s_n$ where

- $s_1 = \texttt{S}$. i.e., s_1 consists of a single element, the start symbol.
- $s_n \in \Sigma^*$, i.e. s_n is made up of terminal symbols only (we write Σ^* to denote the set of all possible strings made up of sequences of words taken from Σ.)
- Each s_i for $i = 2 \ldots n$ is derived from s_{i-1} by picking the left-most non-terminal X in s_{i-1} and replacing it by some β where $X \rightarrow \beta$ is a rule in R.

As one example, one left-most derivation under the grammar in figure 1 is the following:

- $s_1 = \texttt{S}$.
- $s_2 = \texttt{NP VP}$. (We have taken the left-most non-terminal in s_1, namely S, and chosen the rule $\texttt{S} \rightarrow \texttt{NP VP}$, thereby replacing S by NP followed by VP.)

N = {S, NP, VP, PP, DT, Vi, Vt, NN, IN}
S = S
Σ = {sleeps, saw, man, woman, dog, telescope, the, with, in}

$R =$

S	→	NP	VP
VP	→	Vi	
VP	→	Vt	NP
VP	→	VP	PP
NP	→	DT	NN
NP	→	NP	PP
PP	→	IN	NP

Vi	→	sleeps
Vt	→	saw
NN	→	man
NN	→	woman
NN	→	telescope
NN	→	dog
DT	→	the
IN	→	with
IN	→	in

Figure 1: A simple context-free grammar. Note that the set of non-terminals N contains a basic set of syntactic categories: S=sentence, VP=verb phrase, NP=noun phrase, PP=prepositional phrase, DT=determiner, Vi=intransitive verb, Vt=transitive verb, NN=noun, IN=preposition. The set Σ is the set of possible words in the language.

- s_3 = `DT NN VP`. (We have used the rule `NP` → `DT NN` to expand the left-most non-terminal, namely `NP`.)
- s_4 = `the NN VP`. (We have used the rule `DT` → `the`.)
- s_5 = `the man VP`. (We have used the rule `NN` → `man`.)
- s_6 = `the man Vi`. (We have used the rule `VP` → `Vi`.)
- s_7 = `the man sleeps`. (We have used the rule `Vi` → `sleeps`.)

It is very convenient to represent derivations as *parse trees*. For example, the above derivation would be represented as the parse tree shown in figure 2. This parse tree has `S` as its root, reflecting the fact that s_1 = `S`. We see the sequence `NP VP` directly below `S`, reflecting the fact that the `S` was expanded using the rule `S` → `NP VP`; we see the sequence `DT NN` directly below the `NP`, reflecting the fact that the `NP` was expanded using the rule `NP` → `DT NN`; and so on.

A context-free grammar G will in general specify a set of possible left-most derivations. Each left-most derivation will end in a string $s_n \in \Sigma^*$: we say that s_n

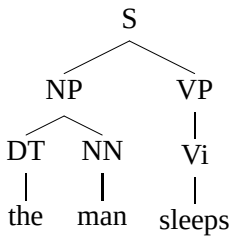


Figure 2: A derivation can be represented as a parse tree.

is the *yield* of the derivation. The set of possible derivations may be a finite or an infinite set (in fact the set of derivations for the grammar in figure 1 is infinite).

The following definition is crucial:

- A string $s \in \Sigma^*$ is said to be in the *language* defined by the CFG, if there is at least one derivation whose yield is s.

2 Ambiguity

Note that some strings s may have more than one underlying derivation (i.e., more than one derivation with s as the yield). In this case we say that the string is *ambiguous* under the CFG.

As one example, see figure 3, which gives two parse trees for the string *the man saw the dog with the telescope*, both of which are valid under the CFG given in figure 1. This example is a case of prepositional phrase attachment ambiguity: the prepositional phrase (`PP`) *with the telescope* can modify either *the dog*, or *saw the dog*. In the first parse tree shown in the figure, the `PP` modifies *the dog*, leading to an `NP` *the dog with the telescope*: this parse tree corresponds to an interpretation where the dog is holding the telescope. In the second parse tree, the `PP` modifies the entire `VP` *saw the dog*: this parse tree corresponds to an interpretation where the man is using the telescope to see the dog.

Ambiguity is an astonishingly severe problem for natural languages. When researchers first started building reasonably large grammars for languages such as English, they were surprised to see that sentences often had a very large number of possible parse trees: it is not uncommon for a moderate-length sentence (say 20 or 30 words in length) to have hundreds, thousands, or even tens of thousands of possible parses.

As one example, in lecture we argued that the following sentence has a surprisingly large number of parse trees (I've found 14 in total):

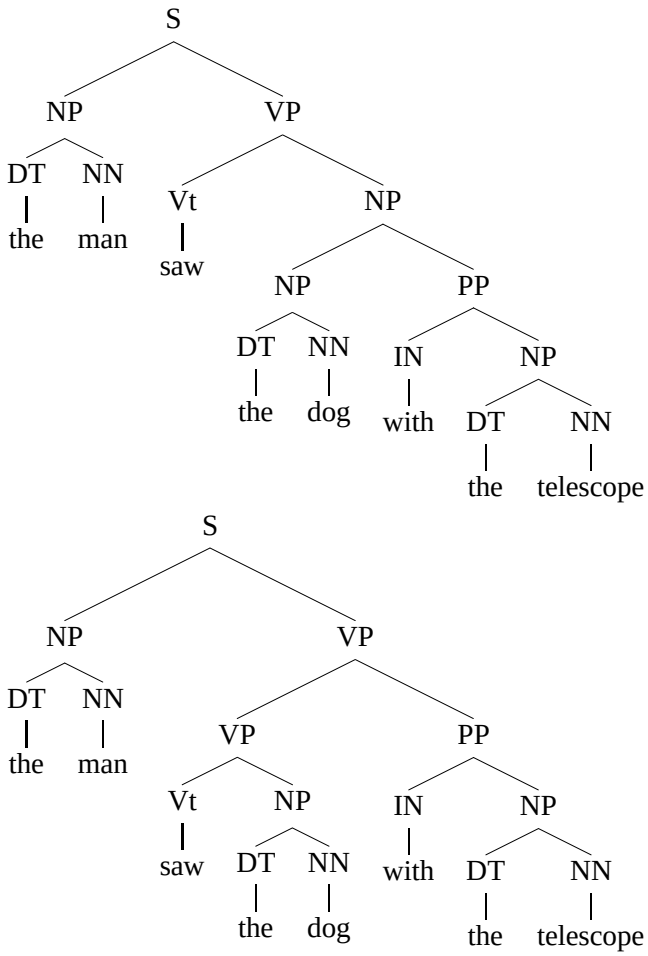


Figure 3: Two parse trees (derivations) for the sentence *the man saw the dog with the telescope*, under the CFG in figure 1.

She announced a program to promote safety in trucks and vans

Can you find the different parse trees for this example?

3 Probabilistic Context-Free Grammars (PCFGs)

3.1 Basic Definitions

Given a context-free grammar G, we will use the following definitions:

- $\mathcal{T}_G$ is the set of all possible left-most derivations (parse trees) under the grammar G. When the grammar G is clear from context we will often write this as simply $\mathcal{T}$.
- For any derivation $t \in \mathcal{T}_G$, we write $\mathsf{yield}(t)$ to denote the string $s \in \Sigma^*$ that is the yield of t (i.e., $\mathsf{yield}(t)$ is the sequence of words in t).
- For a given sentence $s \in \Sigma^*$, we write $\mathcal{T}_G(s)$ to refer to the set

$$\{t : t \in \mathcal{T}_G, \mathsf{yield}(t) = s\}$$

  That is, $\mathcal{T}_G(s)$ is the set of possible parse trees for s.
- We say that a sentence s is *ambiguous* if it has more than one parse tree, i.e., $|\mathcal{T}_G(s)| > 1$.
- We say that a sentence s is *grammatical* if it has at least one parse tree, i.e., $|\mathcal{T}_G(s)| > 0$.

The key idea in probabilistic context-free grammars is to extend our definition to give a *probability distribution over possible derivations*. That is, we will find a way to define a distribution over parse trees, $p(t)$, such that for any $t \in \mathcal{T}_G$,

$$p(t) \geq 0$$

and in addition such that

$$\sum_{t \in \mathcal{T}_G} p(t) = 1$$

At first glance this seems difficult: each parse-tree t is a complex structure, and the set $\mathcal{T}_G$ will most likely be infinite. However, we will see that there is a very simple extension to context-free grammars that allows us to define a function $p(t)$.

Why is this a useful problem? A crucial idea is that once we have a function $p(t)$, we have a ranking over possible parses for any sentence in order of probability. In particular, given a sentence s, we can return

$$\arg \max_{t \in \mathcal{T}_G(s)} p(t)$$

as the output from our parser—this is the most likely parse tree for s under the model. Thus if our distribution $p(t)$ is a good model for the probability of different parse trees in our language, we will have an effective way of dealing with ambiguity.

This leaves us with the following questions:

- How do we define the function $p(t)$?
- How do we learn the parameters of our model of $p(t)$ from training examples?
- For a given sentence s, how do we find the most likely tree, namely

$$\arg \max_{t \in \mathcal{T}_G(s)} p(t)?$$

  This last problem will be referred to as the *decoding* or *parsing* problem.

In the following sections we answer these questions through defining *probabilistic context-free grammars* (PCFGs), a natural generalization of context-free grammars.

3.2 Definition of PCFGs

Probabilistic context-free grammars (PCFGs) are defined as follows:

Definition 1 (PCFGs) *A PCFG consists of:*

1. *A context-free grammar* $G = (N, \Sigma, S, R)$.
2. *A parameter*

   $$q(\alpha \rightarrow \beta)$$

   for each rule $\alpha \rightarrow \beta \in R$. *The parameter* $q(\alpha \rightarrow \beta)$ *can be interpreted as the conditional probabilty of choosing rule* $\alpha \rightarrow \beta$ *in a left-most derivation, given that the non-terminal being expanded is* α. *For any* $X \in N$, *we have the constraint*

   $$\sum_{\alpha \rightarrow \beta \in R : \alpha = X} q(\alpha \rightarrow \beta) = 1$$

   In addition we have $q(\alpha \rightarrow \beta) \geq 0$ *for any* $\alpha \rightarrow \beta \in R$.

Given a parse-tree $t \in \mathcal{T}_G$ containing rules $\alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2, \ldots, \alpha_n \rightarrow \beta_n$, the probability of t under the PCFG is

$$p(t) = \prod_{i=1}^{n} q(\alpha_i \rightarrow \beta_i)$$

□

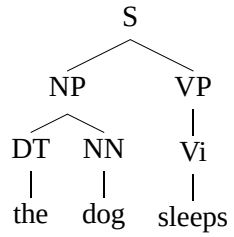
Figure 4 shows an example PCFG, which has the same underlying context-free grammar as that shown in figure 1. The only addition to the original context-free grammar is a parameter $q(\alpha \rightarrow \beta)$ for each rule $\alpha \rightarrow \beta \in R$. Each of these parameters is constrained to be non-negative, and in addition we have the constraint that for any non-terminal $X \in N$,

$$\sum_{\alpha \rightarrow \beta \in R : \alpha = X} q(\alpha \rightarrow \beta) = 1$$

This simply states that for any non-terminal X, the parameter values for all rules with that non-terminal on the left-hand-side of the rule must sum to one. We can verify that this property holds for the PCFG in figure 4. For example, we can verify that this constraint holds for $X = \texttt{VP}$, because

$$\begin{aligned} \sum_{\alpha \rightarrow \beta \in R : \alpha = \texttt{VP}} q(\alpha \rightarrow \beta) &= q(\texttt{VP} \rightarrow \texttt{Vi}) + q(\texttt{VP} \rightarrow \texttt{Vt NP}) + q(\texttt{VP} \rightarrow \texttt{VP PP}) \\ &= 0.3 + 0.5 + 0.2 \\ &= 1.0 \end{aligned}$$

To calculate the probability of any parse tree t, we simply multiply together the q values for the context-free rules that it contains. For example, if our parse tree t is



then we have

$$\begin{aligned} p(t) = \; & q(\texttt{S} \rightarrow \texttt{NP VP}) \times q(\texttt{NP} \rightarrow \texttt{DT NN}) \times q(\texttt{DT} \rightarrow \texttt{the}) \times q(\texttt{NN} \rightarrow \texttt{dog}) \times \\ & q(\texttt{VP} \rightarrow \texttt{Vi}) \times q(\texttt{Vi} \rightarrow \texttt{sleeps}) \end{aligned}$$

Intuitively, PCFGs make the assumption that parse trees are generated stochastically, according to the following process:

N = {S, NP, VP, PP, DT, Vi, Vt, NN, IN}
S = S
Σ = {sleeps, saw, man, woman, dog, telescope, the, with, in}

$R, q =$

S	→	NP	VP	1.0
VP	→	Vi		0.3
VP	→	Vt	NP	0.5
VP	→	VP	PP	0.2
NP	→	DT	NN	0.8
NP	→	NP	PP	0.2
PP	→	IN	NP	1.0

Vi	→	sleeps	1.0
Vt	→	saw	1.0
NN	→	man	0.1
NN	→	woman	0.1
NN	→	telescope	0.3
NN	→	dog	0.5
DT	→	the	1.0
IN	→	with	0.6
IN	→	in	0.4

Figure 4: A simple probabilistic context-free grammar (PCFG). In addition to the set of rules R, we show the parameter value for each rule. For example, $q(\texttt{VP} \rightarrow \texttt{Vt NP}) = 0.5$ in this PCFG.

- Define $s_1 = S$, $i = 1$.
- While s_i contains at least one non-terminal:
  - Find the left-most non-terminal in s_i, call this X.
  - Choose one of the rules of the form $X \rightarrow \beta$ from the distribution $q(X \rightarrow \beta)$.
  - Create s_{i+1} by replacing the left-most X in s_i by β.
  - Set $i = i + 1$.

So we have simply added probabilities to each step in left-most derivations. The probability of an entire tree is the product of probabilities for these individual choices.

3.3 Deriving a PCFG from a Corpus

Having defined PCFGs, the next question is the following: how do we derive a PCFG from a corpus? We will assume a set of training data, which is simply a set

of parse trees $t_1, t_2, \ldots, t_m$. As before, we will write $\mathsf{yield}(t_i)$ to be the yield for the i'th parse tree in the sentence, i.e., $\mathsf{yield}(t_i)$ is the i'th sentence in the corpus.

Each parse tree t_i is a sequence of context-free rules: we assume that every parse tree in our corpus has the same symbol, S, at its root. We can then define a PCFG (N, Σ, S, R, q) as follows:

- N is the set of all non-terminals seen in the trees $t_1 \ldots t_m$.
- Σ is the set of all words seen in the trees $t_1 \ldots t_m$.
- The start symbol S is taken to be S.
- The set of rules R is taken to be the set of all rules $\alpha \rightarrow \beta$ seen in the trees $t_1 \ldots t_m$.
- The maximum-likelihood parameter estimates are

$$q_{ML}(\alpha \rightarrow \beta) = \frac{\text{Count}(\alpha \rightarrow \beta)}{\text{Count}(\alpha)}$$

  where $\text{Count}(\alpha \rightarrow \beta)$ is the number of times that the rule $\alpha \rightarrow \beta$ is seen in the trees $t_1 \ldots t_m$, and $\text{Count}(\alpha)$ is the number of times the non-terminal α is seen in the trees $t_1 \ldots t_m$.

For example, if the rule $\texttt{VP} \rightarrow \texttt{Vt NP}$ is seen 105 times in our corpus, and the non-terminal VP is seen 1000 times, then

$$q(\texttt{VP} \rightarrow \texttt{Vt NP}) = \frac{105}{1000}$$

3.4 Parsing with PCFGs

A crucial question is the following: given a sentence s, how do we find the highest scoring parse tree for s, or more explicitly, how do we find

$$\arg \max_{t \in \mathcal{T}(s)} p(t) \quad ?$$

This section describes a dynamic programming algorithm, *the CKY algorithm*, for this problem.

The CKY algorithm we present applies to a restricted type of PCFG: a PCFG where which is in Chomsky normal form (CNF). While the restriction to grammars in CNF might at first seem to be restrictive, it turns out not to be a strong assumption. It is possible to convert any PCFG into an equivalent grammar in CNF: we will look at this question more in the homeworks.

In the next sections we first describe the idea of grammars in CNF, then describe the CKY algorithm.

N = {S, NP, VP, PP, DT, Vi, Vt, NN, IN}
S = S
Σ = {sleeps, saw, man, woman, dog, telescope, the, with, in}

$R, q =$

S	$\rightarrow$	NP	VP	1.0
VP	$\rightarrow$	Vt	NP	0.8
VP	$\rightarrow$	VP	PP	0.2
NP	$\rightarrow$	DT	NN	0.8
NP	$\rightarrow$	NP	PP	0.2
PP	$\rightarrow$	IN	NP	1.0

Vi	$\rightarrow$	sleeps	1.0
Vt	$\rightarrow$	saw	1.0
NN	$\rightarrow$	man	0.1
NN	$\rightarrow$	woman	0.1
NN	$\rightarrow$	telescope	0.3
NN	$\rightarrow$	dog	0.5
DT	$\rightarrow$	the	1.0
IN	$\rightarrow$	with	0.6
IN	$\rightarrow$	in	0.4

Figure 5: A simple probabilistic context-free grammar (PCFG) in Chomsky normal form. Note that each rule in the grammar takes one of two forms: $X \rightarrow Y_1 \ \ Y_2$ where $X \in N, Y_1 \in N, Y_2 \in N$; or $X \rightarrow Y$ where $X \in N$, $Y \in \Sigma$.

3.4.1 Chomsky Normal Form

Definition 2 (Chomsky Normal Form) *A context-free grammar $G = (N, \Sigma, R, S)$ is in Chomsky form if each rule $\alpha \rightarrow \beta \in R$ takes one of the two following forms:*

- $X \rightarrow Y_1 Y_2$ *where* $X \in N, Y_1 \in N, Y_2 \in N$.
- $X \rightarrow Y$ *where* $X \in N$, $Y \in \Sigma$.

Hence each rule in the grammar either consists of a non-terminal X rewriting as exactly two non-terminal symbols, $Y_1 Y_2$; or a non-terminal X rewriting as exactly one terminal symbol Y. □

Figure 5 shows an example of a PCFG in Chomsky normal form.

3.4.2 Parsing using the CKY Algorithm

We now describe an algorithm for parsing with a PCFG in CNF. The input to the algorithm is a PCFG $G = (N, \Sigma, S, R, q)$ in Chomsky normal form, and a sentence

$s = x_1 \ldots x_n$, where x_i is the i'th word in the sentence. The output of the algorithm is

$$\arg \max_{t \in \mathcal{T}_G(s)} p(t)$$

The CKY algorithm is a dynamic-programming algorithm. Key definitions in the algorithm are as follows:

- For a given sentence $x_1 \ldots x_n$, define $\mathcal{T}(i, j, X)$ for any $X \in N$, for any (i, j) such that $1 \leq i \leq j \leq n$, to be the set of all parse trees for words $x_i \ldots x_j$ such that non-terminal X is at the root of the tree.

- Define

$$\pi(i, j, X) = \max_{t \in \mathcal{T}(i,j,X)} p(t)$$

  (we define $\pi(i, j, X) = 0$ if $\mathcal{T}(i, j, X)$ is the empty set).

Thus $\pi(i, j, X)$ is the highest score for any parse tree that dominates words $x_i \ldots x_j$, and has non-terminal X as its root. The score for a tree t is again taken to be the product of scores for the rules that it contains (i.e. if the tree t contains rules $\alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2, \ldots, \alpha_m \rightarrow \beta_m$, then $p(t) = \prod_{i=1}^{m} q(\alpha_i \rightarrow \beta_i)$).

Note in particular, that

$$\pi(1, n, S) = \arg \max_{t \in \mathcal{T}_G(s)}$$

because by definition $\pi(1, n, S)$ is the score for the highest probability parse tree spanning words $x_1 \ldots x_n$, with S as its root.

The key observation in the CKY algorithm is that we can use a recursive definition of the π values, which allows a simple bottom-up dynamic programming algorithm. The algorithm is "bottom-up", in the sense that it will first fill in $\pi(i, j, X)$ values for the cases where $j = i$, then the cases where $j = i + 1$, and so on.

The base case in the recursive definition is as follows: for all $i = 1 \ldots n$, for all $X \in N$,

$$\pi(i, i, X) = \begin{cases} q(X \rightarrow x_i) & \text{if } X \rightarrow x_i \in R \\ 0 & \text{otherwise} \end{cases}$$

This is a natural definition: the only way that we can have a tree rooted in node X spanning word x_i is if the rule $X \rightarrow x_i$ is in the grammar, in which case the tree has score $q(X \rightarrow x_i)$; otherwise, we set $\pi(i, i, X) = 0$, reflecting the fact that there are no trees rooted in X spanning word x_i.

The recursive definition is as follows: for all (i, j) such that $1 \leq i < j \leq n$, for all $X \in N$,

$$\pi(i, j, X) = \max_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \ldots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z)) \quad (1)$$

The next section of this note gives justification for this recursive definition.

Figure 6 shows the final algorithm, based on these recursive definitions. The algorithm fills in the π values bottom-up: first the $\pi(i, i, X)$ values, using the base case in the recursion; then the values for $\pi(i, j, X)$ such that $j = i + 1$; then the values for $\pi(i, j, X)$ such that $j = i + 2$; and so on.

Note that the algorithm also stores *backpointer* values $bp(i, j, X)$ for all values of (i, j, X). These values record the rule $X \rightarrow YZ$ and the split-point s leading to the highest scoring parse tree. The backpointer values allow recovery of the highest scoring parse tree for the sentence.

3.4.3 Justification for the Algorithm

As an example of how the recursive rule in Eq. 2 is applied, consider parsing the sentence

$$x_1 \ldots x_8 = \textit{the dog saw the man with the telescope}$$

and consider the calculation of $\pi(3, 8, \texttt{VP})$. This will be the highest score for any tree with root VP, spanning words $x_3 \ldots x_8 =$ *saw the man with the telescope*. Eq. 2 specifies that to calculate this value we take the $\max$ over two choices: first, a choice of a rule $\texttt{VP} \rightarrow YZ$ which is in the set of rules R—note that there are two such rules, $\texttt{VP} \rightarrow \texttt{Vt NP}$ and $\texttt{VP} \rightarrow \texttt{VP PP}$. Second, a choice of $s \in \{3, 4, \ldots 7\}$. Thus we will take the maximum value of the following terms:

$$\begin{aligned}
&q(\texttt{VP} \rightarrow \texttt{Vt NP}) \times \pi(3, 3, \texttt{Vt}) \times \pi(4, 8, \texttt{NP}) \\
&q(\texttt{VP} \rightarrow \texttt{VP PP}) \times \pi(3, 3, \texttt{VP}) \times \pi(4, 8, \texttt{PP}) \\
&q(\texttt{VP} \rightarrow \texttt{Vt NP}) \times \pi(3, 4, \texttt{Vt}) \times \pi(5, 8, \texttt{NP}) \\
&q(\texttt{VP} \rightarrow \texttt{VP PP}) \times \pi(3, 4, \texttt{VP}) \times \pi(5, 8, \texttt{PP}) \\
&q(\texttt{VP} \rightarrow \texttt{Vt NP}) \times \pi(3, 5, \texttt{Vt}) \times \pi(6, 8, \texttt{NP}) \\
&q(\texttt{VP} \rightarrow \texttt{VP PP}) \times \pi(3, 5, \texttt{VP}) \times \pi(6, 8, \texttt{PP}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
&q(\texttt{VP} \rightarrow \texttt{Vt NP}) \times \pi(3, 7, \texttt{Vt}) \times \pi(8, 8, \texttt{NP}) \\
&q(\texttt{VP} \rightarrow \texttt{VP PP}) \times \pi(3, 7, \texttt{VP}) \times \pi(8, 8, \texttt{PP})
\end{aligned}$$

Input: a sentence $s = x_1 \dots x_n$, a PCFG $G = (N, \Sigma, S, R, q)$.
Initialization:
For all $i \in \{1 \dots n\}$, for all $X \in N$,

$$\pi(i, i, X) = \begin{cases} q(X \rightarrow x_i) & \text{if } X \rightarrow x_i \in R \\ 0 & \text{otherwise} \end{cases}$$

Algorithm:

- For $l = 1 \dots (n-1)$
  - For $i = 1 \dots (n-l)$
    - Set $j = i + l$
    - For all $X \in N$, calculate

$$\pi(i, j, X) = \max_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \dots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z))$$

and

$$bp(i, j, X) = \arg \max_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \dots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z))$$

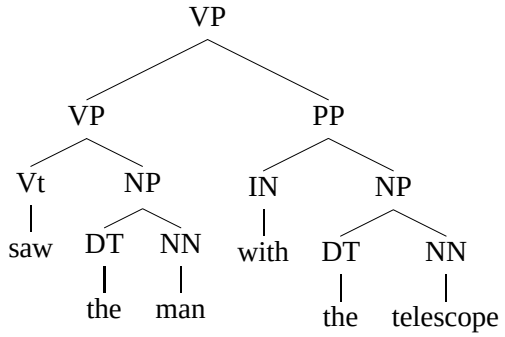
Output: Return $\pi(1, n, S) = \max_{t \in \mathcal{T}(s)} p(t)$, and backpointers bp which allow recovery of $\arg\max_{t \in \mathcal{T}(s)} p(t)$.

Figure 6: The CKY parsing algorithm.

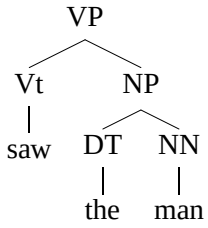
How do we justify this recursive definition? The key observation is that any tree t rooted in X, spanning words $x_i \ldots x_j$, must consist of the following:

- A choice of some rule $X \rightarrow Y\ Z \in R$, at the top of the tree.
- A choice of some value $s \in \{i \ldots j-1\}$, which we will refer to as the "split point" of the rule.
- A choice of a tree rooted in Y, spanning words $x_i \ldots x_s$, call this tree t_1
- A choice of a tree rooted in Z, spanning words $x_{s+1} \ldots x_j$, call this tree t_2.
- We then have
$$p(t) = q(X \rightarrow Y\ Z) \times p(t_1) \times p(t_2)$$
I.e., the probability for the tree t is the product of three terms: the rule probability for the rule at the top of the tree, and probabilities for the sub-trees t_1 and t_2.

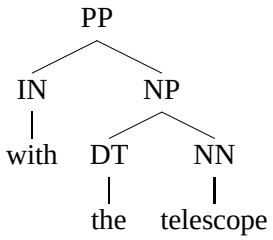
For example, consider the following tree, rooted in `VP`, spanning words $x_3 \ldots x_8$ in our previous example:



In this case we have the rule `VP` $\rightarrow$ `VP PP` at the top of the tree; the choice of split-point is $s = 5$; the tree dominating words $x_3 \ldots x_s$, rooted in `VP`, is



and the tree dominating words $x_{s+1} \ldots x_8$, rooted in `PP`, is



The second key observation is the following:

- *If the highest scoring tree rooted in non-terminal X, and spanning words $x_i \dots x_j$, uses rule $X \rightarrow Y \ \ Z$ and split point s, then its two subtrees must be: 1) the highest scoring tree rooted in Y that spanns words $x_i \dots x_s$; 2) the highest scoring tree rooted in Z that spans words $x_{s+1} \dots x_j$.*

The proof is by contradiction. If either condition (1) or condition (2) was not true, we could always find a higher scoring tree rooted in X, spanning words $x_i \dots x_j$, by choosing a higher scoring subtree spanning words $x_i \dots x_s$ or $x_{s+1} \dots x_j$.

Now let's look back at our recursive definition:

$$\pi(i, j, X) = \max_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \dots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z))$$

We see that it involves a search over rules possible rules $X \rightarrow YZ \in R$, and possible split points s. For each choice of rule and split point, we calculate

$$q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z)$$

which is the highest scoring tree rooted in X, spanning words $x_i \dots x_j$, with this choice of rule and split point. The definition uses the values $\pi(i, s, Y)$ and $\pi(s+1, j, Z)$, corresponding to the two highest scoring subtrees. We take the $\max$ over all possible choices of rules and split points.

3.4.4 The Inside Algorithm for Summing over Trees

We now describe a second, very similar algorithm, which sums the probabilities for all parse trees for a given sentence, thereby calculating the probability of the sentence under the PCFG. The algorithm is called *the inside algorithm*.

The input to the algorithm is again a PCFG $G = (N, \Sigma, S, R, q)$ in Chomsky normal form, and a sentence $s = x_1 \dots x_n$, where x_i is the i'th word in the sentence. The output of the algorithm is

$$p(s) = \sum_{t \in \mathcal{T}_G(s)} p(t)$$

Here $p(s)$ is the probability of the PCFG generating string s.

We define the following:

- As before, for a given sentence $x_1 \ldots x_n$, define $\mathcal{T}(i, j, X)$ for any $X \in N$, for any (i, j) such that $1 \leq i \leq j \leq n$, to be the set of all parse trees for words $x_i \ldots x_j$ such that non-terminal X is at the root of the tree.

- Define
$$\pi(i, j, X) = \sum_{t \in \mathcal{T}(i,j,X)} p(t)$$
(we define $\pi(i, j, X) = 0$ if $\mathcal{T}(i, j, X)$ is the empty set).

Note that we have simply replaced the $\max$ in the previous definition of π, with a sum.

In particular, we have

$$\pi(1, n, S) = \sum_{t \in \mathcal{T}_G(s)} p(t) = p(s)$$

Thus by calculating $\pi(1, n, S)$, we have calculated the probability $p(s)$.

We use a very similar recursive definition to before. First, the base case is as follows: for all $i = 1 \ldots n$, for all $X \in N$,

$$\pi(i, i, X) = \begin{cases} q(X \rightarrow x_i) & \text{if } X \rightarrow x_i \in R \\ 0 & \text{otherwise} \end{cases}$$

The recursive definition is as follows: for all (i, j) such that $1 \leq i < j \leq n$, for all $X \in N$,

$$\pi(i, j, X) = \sum_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \ldots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s + 1, j, Z)) \quad (2)$$

Figure 7 shows the algorithm based on these recursive definitions. The algorithm is essentially identical to the CKY algorithm, but with $\max$ replaced by a sum in the recursive definition. The π values are again calculated bottom-up.

Input: a sentence $s = x_1 \dots x_n$, a PCFG $G = (N, \Sigma, S, R, q)$.
Initialization:
For all $i \in \{1 \dots n\}$, for all $X \in N$,

$$\pi(i, i, X) = \begin{cases} q(X \rightarrow x_i) & \text{if } X \rightarrow x_i \in R \\ 0 & \text{otherwise} \end{cases}$$

Algorithm:

- For $l = 1 \dots (n-1)$
  - For $i = 1 \dots (n-l)$
    - Set $j = i + l$
    - For all $X \in N$, calculate

$$\pi(i, j, X) = \sum_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \dots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z))$$

Output: Return $\pi(1, n, S) = \sum_{t \in \mathcal{T}(s)} p(t)$

Figure 7: The inside algorithm.

Lexicalized Probabilistic Context-Free Grammars

Michael Collins

1 Introduction

In the previous lecture notes we introduced probabilistic context-free grammars (PCFGs) as a model for statistical parsing. We introduced the basic PCFG formalism; described how the parameters of a PCFG can be estimated from a set of training examples (a "treebank"); and derived a dynamic programming algorithm for parsing with a PCFG.

Unfortunately, the basic PCFGs we have described turn out to be a rather poor model for statistical parsing. This note introduces *lexicalized PCFGs*, which build directly on ideas from regular PCFGs, but give much higher parsing accuracy. The remainder of this note is structured as follows:

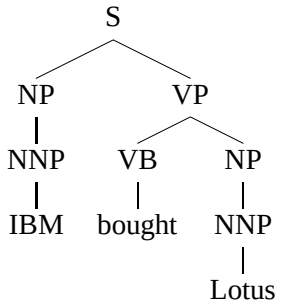
- In section 2 we describe some weaknesses of basic PCFGs, in particular focusing on their lack of sensitity to lexical information.
- In section 3 we describe the first step in deriving lexicalized PCFGs: the process of adding lexical items to non-terminals in treebank parses.
- In section 4 we give a formal definition of lexicalized PCFGs.
- In section 5 we describe how the parameters of lexicalized PCFGs can be estimated from a treebank.
- In section 6 we describe a dynamic-programming algorithm for parsing with lexicalized PCFGs.

2 Weaknesses of PCFGs as Parsing Models

We focus on two crucial weaknesses of PCFGs: 1) lack of sensitivity to lexical information; and 2), lack of sensitivity to structural preferences. Problem (1) is the underlying motivation for a move to lexicalized PCFGs. In a later lecture we will describe extensions to lexical PCFGs that address problem (2).

2.1 Lack of Sensitivity to Lexical Information

First, consider the following parse tree:



Under the PCFG model, this tree will have probability

$$q(\text{S} \rightarrow \text{NP VP}) \times q(\text{VP} \rightarrow \text{V NP}) \times q(\text{NP} \rightarrow \text{NNP}) \times q(\text{NP} \rightarrow \text{NNP})$$
$$\times q(\text{NNP} \rightarrow IBM) \times q(\text{Vt} \rightarrow bought) \times q(\text{NNP} \rightarrow Lotus)$$

Recall that for any rule $\alpha \rightarrow \beta$, $q(\alpha \rightarrow \beta)$ is an associated parameter, which can be interpreted as the conditional probability of seeing β on the right-hand-side of the rule, given that α is on the left-hand-side of the rule.

If we consider the lexical items in this parse tree (i.e., *IBM*, *bought*, and *Lotus*), we can see that the PCFG makes a very strong independence assumption. Intuitively the identity of each lexical item depends only on the part-of-speech (POS) above that lexical item: for example, the choice of the word *IBM* depends on its tag *NNP*, but does not depend directly on other information in the tree. More formally, the choice of each word in the string is conditionally independent of the entire tree, once we have conditioned on the POS directly above the word. This is clearly a very strong assumption, and it leads to many problems in parsing. We will see that lexicalized PCFGs address this weakness of PCFGs in a very direct way.

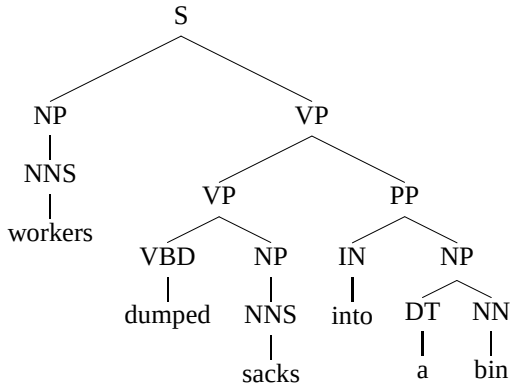
Let's now look at how PCFGs behave under a particular type of ambiguity, prepositional-phrase (PP) attachment ambiguity. Figure 1 shows two parse trees for the same sentence that includes a PP attachment ambiguity. Figure 2 lists the set of context-free rules for the two parse trees. A critical observation is the following: **the two parse trees have identical rules, with the exception of** `VP -> VP PP` **in tree (a), and** `NP -> NP PP` **in tree (b).** It follows that the probabilistic parser, when choosing between the two parse trees, will pick tree (a) if

$$q(\text{VP} \rightarrow \text{VP PP}) > q(\text{NP} \rightarrow \text{NP PP})$$

and will pick tree (b) if

$$q(\text{NP} \rightarrow \text{NP PP}) > q(\text{VP} \rightarrow \text{VP PP})$$

(a)



(b)

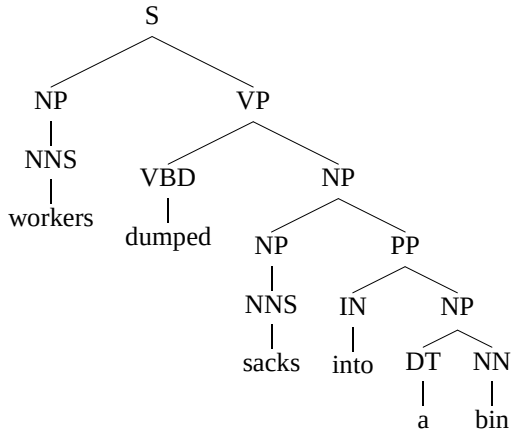


Figure 1: Two valid parses for a sentence that includes a prepositional-phrase attachment ambiguity.

(a)

Rules
S → NP VP
NP → NNS
VP → VP PP
VP → VBD NP
NP → NNS
PP → IN NP
NP → DT NN
NNS → workers
VBD → dumped
NNS → sacks
IN → into
DT → a
NN → bin

(b)

Rules
S → NP VP
NP → NNS
NP → NP PP
VP → VBD NP
NP → NNS
PP → IN NP
NP → DT NN
NNS → workers
VBD → dumped
NNS → sacks
IN → into
DT → a
NN → bin

Figure 2: The set of rules for parse trees (a) and (b) in figure 1.

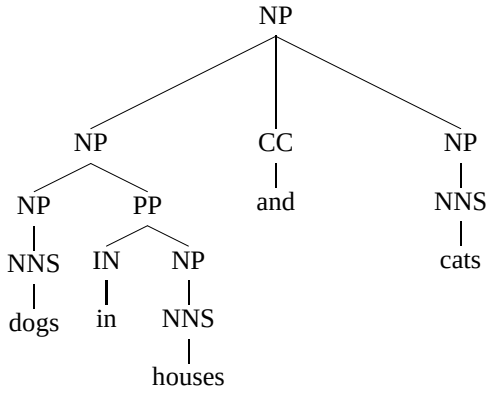
Notice that this decision is *entirely independent of any lexical information (the words) in the two input sentences*. For this particular case of ambiguity (NP vs VP attachment of a PP, with just one possible NP and one possible VP attachment) the parser will always attach PPs to VP if $q(\text{VP} \rightarrow \text{VP PP}) > q(\text{NP} \rightarrow \text{NP PP})$, and conversely will always attach PPs to NP if $q(\text{NP} \rightarrow \text{NP PP}) > q(\text{VP} \rightarrow \text{VP PP})$.

The lack of sensitivity to lexical information in this particular situation, prepositional-phrase attachment ambiguity, is known to be highly non-optimal. The lexical items involved can give very strong evidence about whether to attach to the noun or the verb. If we look at the preposition, *into*, alone, we find that PPs with *into* as the preposition are almost nine times more likely to attach to a VP rather than an NP (this statistic is taken from the Penn treebank data). As another example, PPs with the preposition *of* are about 100 times more likely to attach to an NP rather than a VP. But PCFGs ignore the preposition entirely in making the attachment decision.

As another example, consider the two parse trees shown in figure 3, which is an example of coordination ambiguity. In this case it can be verified that the two parse trees have identical sets of context-free rules (the only difference is in the order in which these rules are applied). Hence a PCFG will assign identical probabilities to these two parse trees, again completely ignoring lexical information.

In summary, the PCFGs we have described essentially generate lexical items as an afterthought, conditioned only on the POS directly above them in the tree. This is a very strong independence assumption, which leads to non-optimal decisions being made by the parser in many important cases of ambiguity.

(a)



(b)

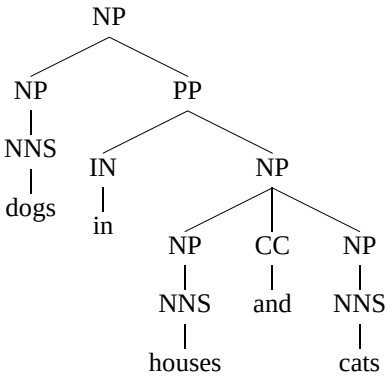


Figure 3: Two valid parses for a noun-phrase that includes an instance of coordination ambiguity.

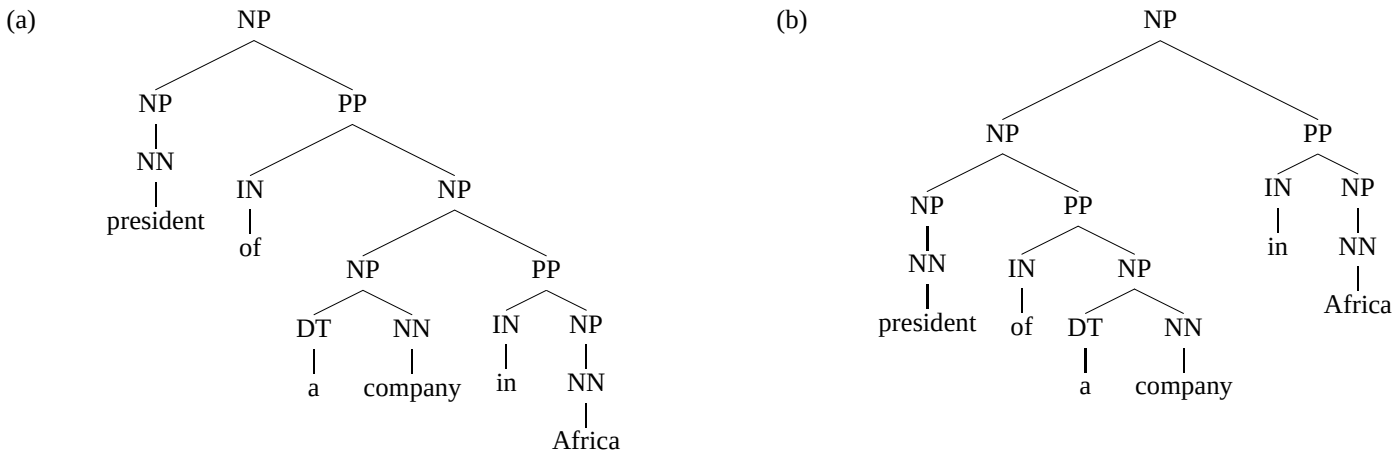


Figure 4: Two possible parses for *president of a company in Africa*.

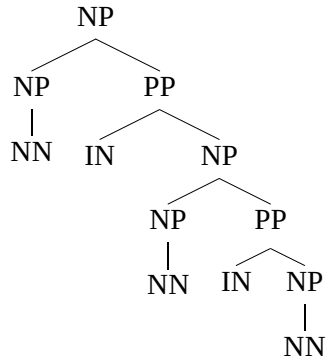
2.2 Lack of Sensitivity to Structural Preferences

A second weakness of PCFGs is their lack of sensitivity to structural preferences. We illustrate this weakness through a couple of examples.

First, consider the two potential parses for *president of a company in Africa*, shown in figure 4. This noun-phrase again involves a case of prepositional-phrase attachment ambiguity: the PP *in Africa* can either attach to *president* or *a company*. It can be verified once again that these two parse trees contain exactly the same set of context-free rules, and will therefore get identical probabilities under a PCFG.

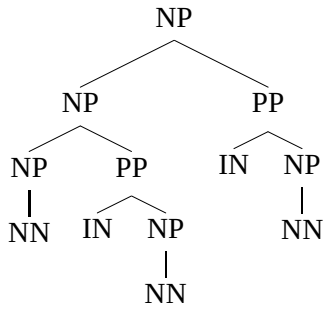
Lexical information may of course help again, in this case. However another useful source of information may be basic statistics about structural preferences (preferences that ignore lexical items). The first parse tree involves a structure of the following form, where the final PP (*in Africa* in the example) attaches to the most recent NP (*a company*):

(1)



This attachment for the final PP is often referred to as a *close attachment*, because the PP has attached to the closest possible preceding NP. The second parse structure has the form

(2)



where the final PP has attached to the further NP (*president* in the example).

We can again look at statistics from the treebank for the frequency of structure 1 versus 2: structure 1 is roughly twice as frequent as structure 2. So there is a fairly significant bias towards close attachment. Again, we stress that the PCFG assigns identical probabilities to these two trees, because they include the same set of rules: hence the PCFG fails to capture the bias towards close-attachment in this case.

There are many other examples where close attachment is a useful cue in disambiguating structures. The preferences can be even stronger when a choice is being made between attachment to two different verbs. For example, consider the sentence

John was believed to have been shot by Bill

Here the PP *by Bill* can modify either the verb *shot* (Bill was doing the shooting) or *believe* (Bill is doing the believing). However statistics from the treebank show that when a PP can attach to two potential verbs, it is about 20 times more likely

to attach to the most recent verb. Again, the basic PCFG will often give equal probability to the two structures in question, because they contain the same set of rules.

3 Lexicalization of a Treebank

We now describe how lexicalized PCFGs address the first fundamental weakness of PCFGs: their lack of sensitivity to lexical information. The first key step, described in this section, is to *lexicalize* the underlying treebank.

Figure 5 shows a parse tree before and after lexicalization. The lexicalization step has replaced non-terminals such as `S` or `NP` with new non-terminals that include lexical items, for example `S(questioned)` or `NP(lawyer)`.

The remainder of this section describes exactly how trees are lexicalized. First though, we give the underlying motivation for this step. The basic idea will be to replace rules such as

$$\text{S} \rightarrow \text{NP VP}$$

in the basic PCFG, with rules such as

$$\text{S(questioned)} \rightarrow \text{NP(lawyer) VP(questioned)}$$

in the lexicalized PCFG. The symbols S(questioned), NP(lawyer) and VP(questioned) are new non-terminals in the grammar. Each non-terminal now includes a lexical item; the resulting model has far more sensitivity to lexical information.

In one sense, nothing has changed from a formal standpoint: we will simply move from a PCFG with a relatively small number of non-terminals (`S`, `NP`, etc.) to a PCFG with a much larger set of non-terminals (`S(questioned)`, `NP(lawyer)` etc.) This will, however, lead to a radical increase in the number of rules and non-terminals in the grammar: for this reason we will have to take some care in estimating the parameters of the underlying PCFG. We describe how this is done in the next section.

First, however, we describe how the lexicalization process is carried out. The key idea will be to identify for each context-free rule of the form

$$X \rightarrow Y_1 \; Y_2 \; \ldots \; Y_n$$

an index $h \in \{1 \ldots n\}$ that specifies the *head* of the rule. The head of a context-free rule intuitively corresponds to the "center" or the most important child of the rule.[1] For example, for the rule

$$\text{S} \rightarrow \text{NP VP}$$

[1] The idea of heads has a long history in linguistics, which is beyond the scope of this note.

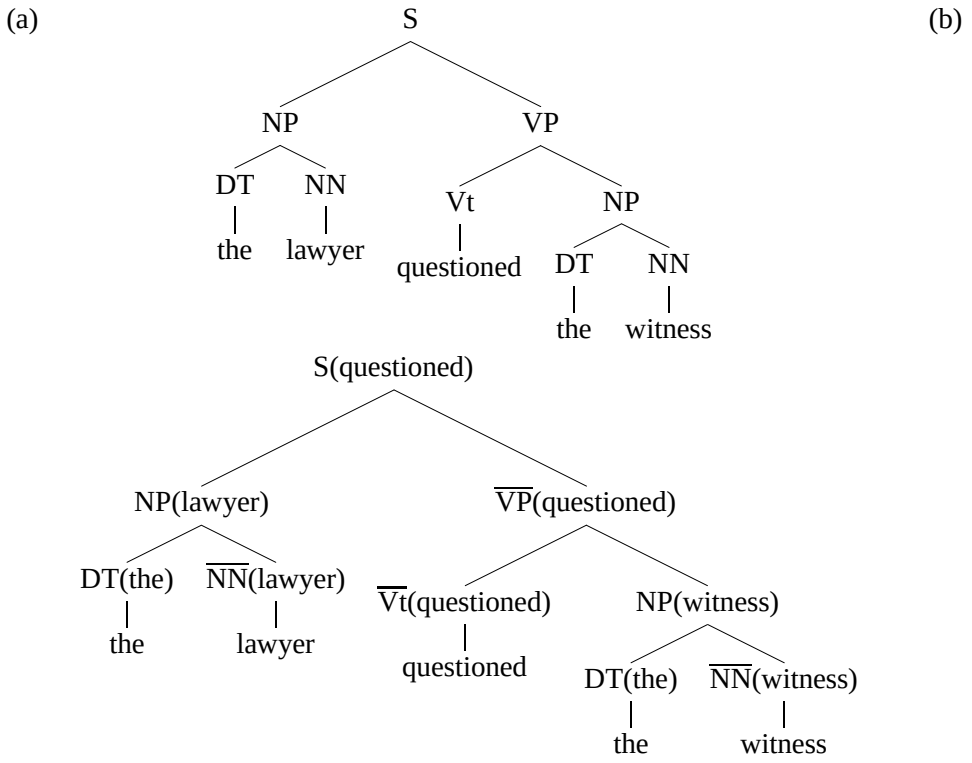


Figure 5: (a) A conventional parse tree as found for example in the Penn treebank. (b) A lexicalized parse tree for the same sentence. Note that each non-terminal in the tree now includes a single lexical item. For clarity we mark the head of each rule with an overline: for example for the rule `NP` $\rightarrow$ `DT NN` the child `NN` is the head, and hence the `NN` symbol is marked as $\overline{\texttt{NN}}$.

the head would be $h = 2$ (corresponding to the VP). For the rule

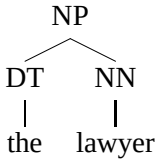
$$\text{NP} \rightarrow \text{NP PP PP PP}$$

the head would be $h = 1$ (corresponding to the NP). For the rule

$$\text{PP} \rightarrow \text{IN NP}$$

the head would be $h = 1$ (corresponding to the IN), and so on.

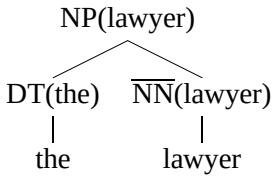
Once the head of each context-free rule has been identified, lexical information can be propagated bottom-up through parse trees in the treebank. For example, if we consider the sub-tree



and assuming that the head of the rule

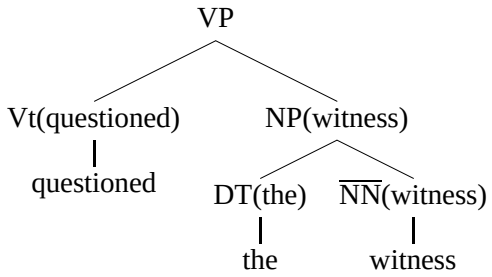
$$\text{NP} \rightarrow \text{DT NN}$$

is $h = 2$ (the NN), the lexicalized sub-tree is

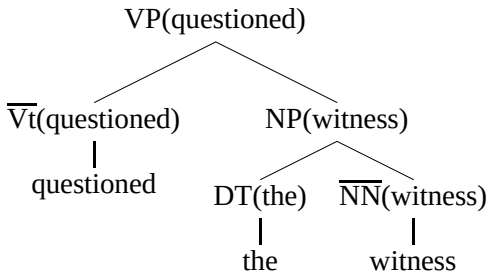


Parts of speech such as DT or NN receive the lexical item below them as their head word. Non-terminals higher in the tree receive the lexical item from their head child: for example, the NP in this example receives the lexical item lawyer, because this is the lexical item associated with the NN which is the head child of the NP. For clarity, we mark the head of each rule in a lexicalized parse tree with an overline (in this case we have $\overline{\text{NN}}$). See figure 5 for an example of a complete lexicalized tree.

As another example consider the VP in the parse tree in figure 5. Before lexicalizing the VP, the parse structure is as follows (we have filled in lexical items lower in the tree, using the steps described before):



We then identify `Vt` as the head of the rule VP $\rightarrow$ Vt NP, and lexicalize the tree as follows:



In summary, once the head of each context-free rule has been identified, lexical items can be propagated bottom-up through parse trees, to give lexicalized trees such as the one shown in figure 5(b).

The remaining question is how to identify heads. Ideally, the head of each rule would be annotated in the treebank in question: in practice however, these annotations are often not present. Instead, researchers have generally used a simple set of rules to automatically identify the head of each context-free rule.

As one example, figure 6 gives an example set of rules that identifies the head of rules whose left-hand-side is `NP`. Figure 7 shows a set of rules used for `VPs`. In both cases we see that the rules look for particular children (e.g., `NN` for the `NP` case, `Vi` for the `VP` case). The rules are fairly heuristic, but rely on some linguistic guidance on what the head of a rule should be: in spite of their simplicity they work quite well in practice.

4 Lexicalized PCFGs

The basic idea in lexicalized PCFGs will be to replace rules such as

$$S \rightarrow NP\ VP$$

with lexicalized rules such as

$$S(examined) \rightarrow NP(lawyer)\ VP(examined)$$

If the rule contains NN, NNS, or NNP:
    Choose the rightmost NN, NNS, or NNP

Else If the rule contains an NP: Choose the leftmost NP

Else If the rule contains a JJ: Choose the rightmost JJ

Else If the rule contains a CD: Choose the rightmost CD

Else Choose the rightmost child

Figure 6: Example of a set of rules that identifies the head of any rule whose left-hand-side is an NP.

If the rule contains Vi or Vt: Choose the leftmost Vi or Vt

Else If the rule contains a VP: Choose the leftmost VP

Else Choose the leftmost child

Figure 7: Example of a set of rules that identifies the head of any rule whose left-hand-side is a VP.

Thus we have replaced simple non-terminals such as `S` or `NP` with lexicalized non-terminals such as `S(examined)` or `NP(lawyer)`.

From a formal standpoint, nothing has changed: we can treat the new, lexicalized grammar exactly as we would a regular PCFG. We have just expanded the number of non-terminals in the grammar from a fairly small number (say 20, or 50) to a much larger number (because each non-terminal now has a lexical item, we could easily have thousands or tens of thousands of non-terminals).

Each rule in the lexicalized PCFG will have an associated parameter, for example the above rule would have the parameter

$$q(\text{S(examined)} \rightarrow \text{NP(lawyer) VP(examined)})$$

There are a very large number of parameters in the model, and we will have to take some care in estimating them: the next section describes parameter estimation methods.

We will next give a formal definition of lexicalized PCFGs, in Chomsky normal form. First, though, we need to take care of one detail. Each rule in the lexicalized PCFG has a non-terminal with a head word on the left hand side of the rule: for example the rule

$$\text{S(examined)} \rightarrow \text{NP(lawyer) VP(examined)}$$

has `S(examined)` on the left hand side. In addition, the rule has two children. One of the two children must have the same lexical item as the left hand side: in this example `VP(examined)` is the child with this property. To be explicit about which child shares the lexical item with the left hand side, we will add an annotation to the rule, using $\rightarrow_1$ to specify that the left child shares the lexical item with the parent, and $\rightarrow_2$ to specify that the right child shares the lexical item with the parent. So the above rule would now be written as

$$\text{S(examined)} \rightarrow_2 \text{NP(lawyer) VP(examined)}$$

The extra notation might seem unneccessary in this case, because it is clear that the second child is the head of the rule—it is the only child to have the same lexical item, *examined*, as the left hand side of the rule. However this information will be important for rules where both children have the same lexical item: take for example the rules

$$\text{PP(in)} \rightarrow_1 \text{PP(in) PP(in)}$$

and

$$\text{PP(in)} \rightarrow_2 \text{PP(in) PP(in)}$$

where we need to be careful about specifying which of the two children is the head of the rule.

We now give the following definition:

Definition 1 (Lexicalized PCFGs in Chomsky Normal Form) *A lexicalized PCFG in Chomsky normal form is a 6-tuple $G = (N, \Sigma, R, S, q, \gamma)$ where:*

- *N is a finite set of non-terminals in the grammar.*
- *Σ is a finite set of lexical items in the grammar.*
- *R is a set of rules. Each rule takes one of the following three forms:*
  1. *$X(h) \rightarrow_1 Y_1(h)\ Y_2(m)$ where $X, Y_1, Y_2 \in N$, $h, m \in \Sigma$.*
  2. *$X(h) \rightarrow_2 Y_1(m)\ Y_2(h)$ where $X, Y_1, Y_2 \in N$, $h, m \in \Sigma$.*
  3. *$X(h) \rightarrow h$ where $X \in N$, $h \in \Sigma$.*
- *For each rule $r \in R$ there is an associated parameter*

  $$q(r)$$

  The parameters satisfy $q(r) \geq 0$, and for any $X \in N, h \in \Sigma$,

  $$\sum_{r \in R : LHS(r) = X(h)} q(r) = 1$$

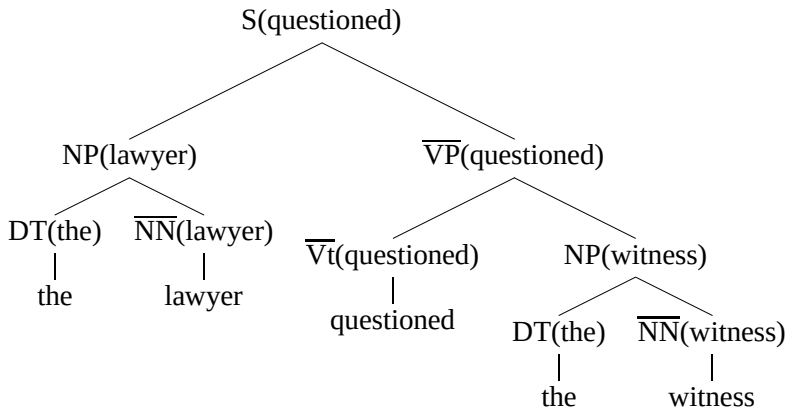
  where we use $LHS(r)$ to refer to the left hand side of any rule r.
- *For each $X \in N$, $h \in \Sigma$, there is a parameter $\gamma(X, h)$. We have $\gamma(X, h) \geq 0$, and $\sum_{X \in N, h \in \Sigma} \gamma(X, h) = 1$.*

Given a left-most derivation $r_1, r_2, \ldots r_N$ under the grammar, where each r_i is a member of R, the probability of the derivation is

$$\gamma(LHS(r_1)) \times \prod_{i=1}^{N} q(r_i)$$

□

As an example, consider the parse tree in figure 5, repeated here:



In this case the parse tree consists of the following sequence of rules:

S(questioned) $\rightarrow_2$ NP(lawyer) VP(questioned)
NP(lawyer) $\rightarrow_2$ DT(the) NN(lawyer)
DT(the) $\rightarrow$ the
NN(lawyer) $\rightarrow$ lawyer
VP(questioned) $\rightarrow_1$ Vt(questioned) NP(witness)
NP(witness) $\rightarrow_2$ DT(the) NN(witness)
DT(the) $\rightarrow$ the
NN(witness) $\rightarrow$ witness

The probability of the tree is calculated as

$\gamma(\text{S, questioned})$
$\times q(\text{S(questioned)} \rightarrow_2 \text{NP(lawyer) VP(questioned)})$
$\times q(\text{NP(lawyer)} \rightarrow_2 \text{DT(the) NN(lawyer)})$
$\times q(\text{DT(the)} \rightarrow \text{the})$
$\times q(\text{NN(lawyer)} \rightarrow \text{lawyer})$
$\times q(\text{VP(questioned)} \rightarrow_1 \text{Vt(questioned) NP(witness)})$
$\times q(\text{NP(witness)} \rightarrow_2 \text{DT(the) NN(witness)})$
$\times q(\text{DT(the)} \rightarrow \text{the})$
$\times q(\text{NN(witness)} \rightarrow \text{witness})$

Thus the model looks very similar to regular PCFGs, where the probability of a tree is calculated as a product of terms, one for each rule in the tree. One difference is that we have the $\gamma(\text{S, questioned})$ term for the root of the tree: this term can be interpreted as the probability of choosing the nonterminal S(questioned) at the

root of the tree. (Recall that in regular PCFGs we specified that a particular non-terminal, for example S, always appeared at the root of the tree.)

5 Parameter Estimation in Lexicalized PCFGs

We now describe a method for parameter estimation within lexicalized PCFGs. The number of rules (and therefore parameters) in the model is very large. However with appropriate smoothing—using techniques described earlier in the class, for language modeling—we can derive estimates that are robust and effective in practice.

First, for a given rule of the form

$$X(h) \rightarrow_1 Y_1(h)\ Y_2(m)$$

or

$$X(h) \rightarrow_2 Y_1(m)\ Y_2(h)$$

define the following variables: X is the non-terminal on the left-hand side of the rule; H is the head-word of that non-terminal; R is the rule used, either of the form $X \rightarrow_1 Y_1\ Y_2$ or $X \rightarrow_2 Y_1\ Y_2$; M is the modifier word.

For example, for the rule

$$\text{S(examined)} \rightarrow_2 \text{NP(lawyer) VP(examined)}$$

we have

$$\begin{aligned} X &= \text{S} \\ H &= \text{examined} \\ R &= \text{S} \rightarrow_2 \text{NP VP} \\ M &= \text{lawyer} \end{aligned}$$

With these definitions, the parameter for the rule has the following interpretation:

$$\begin{aligned} & q(\text{S(examined)} \rightarrow_2 \text{NP(lawyer) VP(examined)}) \\ = \ & P(R = \text{S} \rightarrow_2 \text{NP VP}, M = \text{lawyer} | X = \text{S}, H = \text{examined}) \end{aligned}$$

The first step in deriving an estimate of $q(\text{S(examined)} \rightarrow_2 \text{NP(lawyer) VP(examined)})$ will be to use the chain rule to decompose the above expression into two terms:

$$\begin{aligned} & P(R = \text{S} \rightarrow_2 \text{NP VP}, M = \text{lawyer} | X = \text{S}, H = \text{examined}) \\ = \ & P(R = \text{S} \rightarrow_2 \text{NP VP} | X = \text{S}, H = \text{examined}) \qquad (3) \\ & \times P(M = \text{lawyer} | R = \text{S} \rightarrow_2 \text{NP VP}, X = \text{S}, H = \text{examined}) \qquad (4) \end{aligned}$$

This step is exact, by the chain rule of probabilities.

We will now derive separate smoothed estimates of the quantities in Eqs. 3 and 4. First, for Eq. 3 define the following maximum-likelihood estimates:

$$q_{ML}(\text{S} \rightarrow_2 \text{NP VP}|\text{S, examined}) = \frac{\text{count}(R = \text{S} \rightarrow_2 \text{NP VP}, X = \text{S}, H = \text{examined})}{\text{count}(X = \text{S}, H = \text{examined})}$$

$$q_{ML}(\text{S} \rightarrow_2 \text{NP VP}|\text{S}) = \frac{\text{count}(R = \text{S} \rightarrow_2 \text{NP VP}, X = \text{S})}{\text{count}(X = \text{S})}$$

Here the count$(\ldots)$ expressions are counts derived directly from the training samples (the lexicalized trees in the treebank). Our estimate of

$$P(R = \text{S} \rightarrow_2 \text{NP VP}|X = \text{S}, H = \text{examined})$$

is then defined as

$$\lambda_1 \times q_{ML}(\text{S} \rightarrow_2 \text{NP VP}|\text{S, examined}) + (1 - \lambda_1) \times q_{ML}(\text{S} \rightarrow_2 \text{NP VP}|\text{S})$$

where λ_1 dictates the relative weights of the two estimates (we have $0 \leq \lambda_1 \leq 1$). The value for λ_1 can be estimated using the methods described in the notes on language modeling for this class.

Next, consider our estimate of the expression in Eq. 4. We can define the following two maximum-likelihood estimates:

$$q_{ML}(\text{lawyer}|\text{S} \rightarrow_2 \text{NP VP, examined}) = \frac{\text{count}(M = \text{lawyer}, R = \text{S} \rightarrow_2 \text{NP VP}, H = \text{examined})}{\text{count}(R = \text{S} \rightarrow_2 \text{NP VP}, H = \text{examined})}$$

$$q_{ML}(\text{lawyer}|\text{S} \rightarrow_2 \text{NP VP}) = \frac{\text{count}(M = \text{lawyer}, R = \text{S} \rightarrow_2 \text{NP VP})}{\text{count}(R = \text{S} \rightarrow_2 \text{NP VP})}$$

The estimate of

$$P(M = \text{lawyer}|R = \text{S} \rightarrow_2 \text{NP VP}, X = \text{S}, H = \text{examined})$$

is then

$$\lambda_2 \times q_{ML}(\text{lawyer}|\text{S} \rightarrow_2 \text{NP VP, examined}) + (1 - \lambda_2) \times q_{ML}(\text{lawyer}|\text{S} \rightarrow_2 \text{NP VP})$$

where $0 \leq \lambda_2 \leq 1$ is a parameter specifying the relative weights of the two terms.

Putting these estimates together, our final estimate of the rule parameter is as follows:

$$\begin{aligned} & q(\text{S(examined)} \rightarrow_2 \text{NP(lawyer) VP(examined)}) \\ = \; & (\lambda_1 \times q_{ML}(\text{S} \rightarrow_2 \text{NP VP}|\text{S, examined}) + (1 - \lambda_1) \times q_{ML}(\text{S} \rightarrow_2 \text{NP VP}|\text{S})) \\ & \times (\lambda_2 \times q_{ML}(\text{lawyer}|\text{S} \rightarrow_2 \text{NP VP, examined}) + (1 - \lambda_2) \times q_{ML}(\text{lawyer}|\text{S} \rightarrow_2 \text{NP VP})) \end{aligned}$$

It can be seen that this estimate combines very lexically-specific information, for example the estimates

$$q_{ML}(\text{S} \rightarrow_2 \text{NP VP} | \text{S, examined})$$

$$q_{ML}(\text{lawyer} | \text{S} \rightarrow_2 \text{NP VP, examined})$$

with estimates that rely less on lexical information, for example

$$q_{ML}(\text{S} \rightarrow_2 \text{NP VP} | \text{S})$$

$$q_{ML}(\text{lawyer} | \text{S} \rightarrow_2 \text{NP VP})$$

The end result is a model that is sensitive to lexical information, but which is nevertheless robust, because we have used smoothed estimates of the very large number of parameters in the model.

6 Parsing with Lexicalized PCFGs

The parsing algorithm for lexicalized PCFGs is very similar to the parsing algorithm for regular PCFGs, as described in the previous lecture notes. Recall that for a regular PCFG the dynamic programming algorithm for parsing makes use of a dynamic programming table $\pi(i, j, X)$. Each entry $\pi(i, j, X)$ stores the highest probability for any parse tree rooted in non-terminal X, spanning words $i \ldots j$ inclusive in the input sentence. The π values can be completed using a recursive definition, as follows. Assume that the input sentence to the algorithm is $x_1 \ldots x_n$. The base case of the recursion is for $i = 1 \ldots n$, for all $X \in N$,

$$\pi(i, i, X) = q(X \rightarrow x_i)$$

where we define $q(X \rightarrow x_i) = 0$ if the rule $X \rightarrow x_i$ is not in the grammar.

The recursive definition is as follows: for any non-terminal X, for any i, j such that $1 \leq i < j \leq n$,

$$\pi(i, j, X) = \max_{X \rightarrow Y\ Z, s \in \{i \ldots (j-1)\}} q(X \rightarrow Y\ Z) \times \pi(i, s, Y) \times \pi(s + 1, j, Z)$$

Thus we have a $\max$ over all rules $X \rightarrow Y\ Z$, and all split-points $s \in \{i \ldots (j - 1)\}$. This recursion is justified because any parse tree rooted in X, spanning words $i \ldots j$, must be composed of the following choices:

- A rule $X \rightarrow Y\ Z$ at the root of the tree.
- A split point $s \in \{i \ldots (j - 1)\}$.

- A sub-tree rooted in Y, spanning words $\{i \dots s\}$.
- A sub-tree rooted in Z, spanning words $\{(s+1) \dots j\}$.

Now consider the case of lexicalized PCFGs. A key difference is that each non-terminal in the grammar includes a lexical item. A key observation is that for a given input sentence $x_1 \dots x_n$, parse trees for that sentence can only include non-terminals with lexical items that are one of $x_1 \dots x_n$.

Following this observation, we will define a dynamic programming table with entries $\pi(i, j, h, X)$ for $1 \leq i \leq j \leq n$, $h \in \{i \dots j\}$, $X \in N$, where N is the set of unlexicalized non-terminals in the grammar. We give the following definition:

Definition 2 (Dynamic programming table for lexicalized PCFGs.) *$\pi(i, j, h, X)$ is the highest probability for any parse tree with non-terminal X and lexical item h at its root, spanning words $i \dots j$ in the input.*

As an example, consider the following sentence from earlier in this note:

workers dumped the sacks into a bin

In this case we have $n = 7$ (there are seven words in the sentence). As one example entry,

$$\pi(2, 7, 2, VP)$$

will be the highest probability for any subtree rooted in $VP(dumped)$, spanning words $2 \dots 7$ in the sentence.

The π values can again be completed using a recursive definition. The base case is as follows:

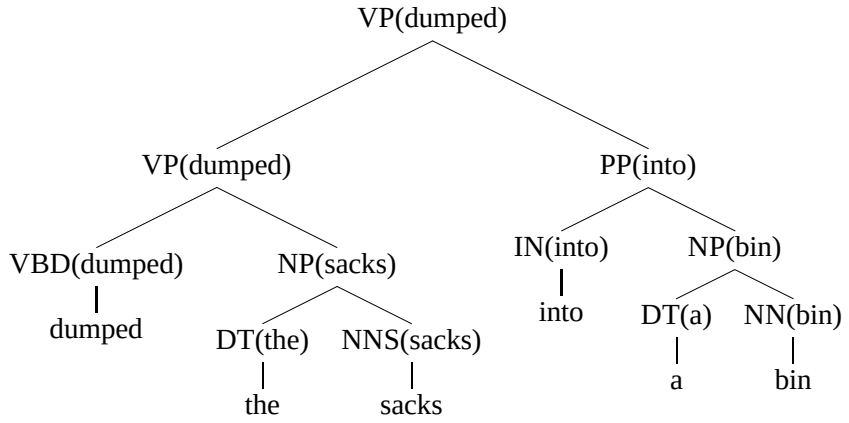
$$\pi(i, i, i, X) = q(X(x_i) \rightarrow x_i)$$

where we define $q(X(x_i) \rightarrow x_i) = 0$ if the rule $q(X(x_i) \rightarrow x_i)$ is not in the lexicalized PCFG. Note that this is very similar to the base case for regular PCFGs. As one example, we would set

$$\pi(1, 1, 1, \text{NNS}) = q(\text{NNS(workers)} \rightarrow \text{workers})$$

for the above example sentence.

We now consider the recursive definition. As an example, consider completing the value of $\pi(2, 7, 2, VP)$ for our example sentence. Consider the following sub-tree that spans words $2 \dots 7$, has lexical item $x_2 =$ dumped and label VP at its root:



We can see that this subtree has the following sub-parts:

- A choice of split-point $s \in \{1 \ldots 6\}$. In this case we choose $s = 4$ (the split between the two subtrees under the rule VP(dumped) $\rightarrow_1$ VBD(dumped) PP(into) is after $x_4 =$ sacks).
- A choice of modifier word m. In this case we choose $m = 5$, corresponding to $x_5 =$ into, because x_5 is the head word of the second child of the rule VP(dumped) $\rightarrow_1$ VBD(dumped) PP(into).
- A choice of rule at the root of the tree: in this case the rule is VP(dumped) $\rightarrow_1$ VBD(dumped) PP(into).

More generally, to find the value for any $\pi(i, j, h, X)$, we need to search over all possible choices for s, m, and all rules of the form $X(x_h) \rightarrow_1 Y_1(x_h)\ Y_2(x_m)$ or $X(x_h) \rightarrow_2 Y_1(x_m)\ Y_2(x_h)$. Figure 8 shows pseudo-code for this step. Note that some care is needed when enumerating the possible values for s and m. If s is in the range $h \ldots (j-1)$ then the head word h must come from the left sub-tree; it follows that m must come from the right sub-tree, and hence must be in the range $(s+1) \ldots j$. Conversely, if s is in the range $i \ldots (h-1)$ then m must be in the left sub-tree, i.e., in the range $i \ldots s$. The pseudo-code in figure 8 treats these two cases separately.

Figure 9 gives the full algorithm for parsing with lexicalized PCFGs. The algorithm first completes the base case of the recursive definition for the π values. It then fills in the rest of the π values, starting with the case where $j = i + 1$, then the case $j = i + 2$, and so on. Finally, the step

$$(X^*, h^*) = \arg \max_{X \in N, h \in \{1 \ldots n\}} \gamma(X, h) \times \pi(1, n, h, X)$$

finds the pair $X^*(h^*)$ which is at the root of the most probable tree for the input sentence: note that the γ term is taken into account at this step. The highest probability tree can then be recovered by following backpointers starting at $bp(1, n, h^*, X^*)$.

1. $\pi(i, j, h, X) = 0$
2. For $s = h \ldots (j-1)$, for $m = (s+1) \ldots j$, for $X(x_h) \rightarrow_1 Y(x_h)Z(x_m) \in R$,
   (a) $p = q(X(x_h) \rightarrow_1 Y(x_h)Z(x_m)) \times \pi(i, s, h, Y) \times \pi(s+1, j, m, Z)$
   (b) If $p > \pi(i, j, h, X)$,
   $$\pi(i, j, h, X) = p$$
   $$bp(i, j, h, X) = \langle s, m, Y, Z \rangle$$
3. For $s = i \ldots (h-1)$, for $m = i \ldots s$, for $X(x_h) \rightarrow_2 Y(x_m)Z(x_h) \in R$,
   (a) $p = q(X(x_h) \rightarrow_2 Y(x_m)Z(x_h)) \times \pi(i, s, m, Y) \times \pi(s+1, j, h, Z)$
   (b) If $p > \pi(i, j, h, X)$,
   $$\pi(i, j, h, X) = p$$
   $$bp(i, j, h, X) = \langle s, m, Y, Z \rangle$$

Figure 8: The method for calculating an entry $\pi(i, j, h, X)$ in the dynamic programming table. The pseudo-code searches over all split-points s, over all modifier positions m, and over all rules of the form $X(x_h) \rightarrow_1 Y(x_h)\ Z(x_m)$ or $X(x_h) \rightarrow_2 Y(x_m)\ Z(x_h)$. The algorithm stores backpointer values $bp(i, j, h, X)$.

Input: a sentence $s = x_1 \dots x_n$, a lexicalized PCFG $G = (N, \Sigma, S, R, q, \gamma)$.
Initialization:
For all $i \in \{1 \dots n\}$, for all $X \in N$,

$$\pi(i, i, i, X) = \begin{cases} q(X(x_i) \rightarrow x_i) & \text{if } X(x_i) \rightarrow x_i \in R \\ 0 & \text{otherwise} \end{cases}$$

Algorithm:

- For $l = 1 \dots (n-1)$
  - For $i = 1 \dots (n-l)$
    - Set $j = i + l$
    - For all $X \in N$, $h \in \{i \dots j\}$, calculate $\pi(i, j, h, X)$ using the algorithm in figure 8.

Output:

$$(X^*, h^*) = \arg \max_{S \in N, h \in \{1 \dots n\}} \gamma(X, h) \times \pi(1, n, h, X)$$

Use backpointers starting at $bp(1, n, h^*, X^*)$ to obtain the highest probability tree.

Figure 9: The CKY parsing algorithm for lexicalized PCFGs.

Statistical Machine Translation: IBM Models 1 and 2

Michael Collins

1 Introduction

The next few lectures of the course will be focused on machine translation, and in particular on *statistical machine translation* (SMT) systems. In this note we will focus on the *IBM translation models*, which go back to the late 1980s/early 1990s. These models were seminal, and form the basis for many SMT models now used today.

Following convention, we'll assume throughout this note that the task is to translate from *French* (the "source" language) into *English* (the "target" language). In general we will use f to refer to a sentence in French: f is a sequence of words $f_1, f_2 \dots f_m$ where m is the length of the sentence, and f_j for $j \in \{1 \dots m\}$ is the j'th word in the sentence. We will use e to refer to an English sentence: e is equal to $e_1, e_2 \dots e_l$ where l is the length of the English sentence.

In SMT systems, we assume that we have a source of example translations, $(f^{(k)}, e^{(k)})$ for $k = 1 \dots n$, where $f^{(k)}$ is the k'th French sentence in the training examples, $e^{(k)}$ is the k'th English sentence, and $e^{(k)}$ is a translation of $f^{(k)}$. Each $f^{(k)}$ is equal to $f_1^{(k)} \dots f_{m_k}^{(k)}$ where m_k is the length of the k'th French sentence. Each $e^{(k)}$ is equal to $e_1^{(k)} \dots e_{l_k}^{(k)}$ where l_k is the length of the k'th English sentence. We will estimate the parameters of our model from these training examples.

So where do we get the training examples from? It turns out that quite large corpora of translation examples are available for many language pairs. The original IBM work, which was in fact focused on translation from French to English, made use of the Canadian parliamentary proceedings (the *Hansards*): the corpus they used consisted of several million translated sentences. The *Europarl data* consists of proceedings from the European parliament, and consists of translations between several European languages. Other substantial corpora exist for Arabic-English, Chinese-English, and so on.

2 The Noisy-Channel Approach

A few lectures ago we introduced generative models, and in particular the noisy-channel approach. The IBM models are an instance of a noisy-channel model, and as such they have two components:

1. A *language model* that assigns a probability $p(e)$ for any sentence $e = e_1 \dots e_l$ in English. We will, for example, use a trigram language model for this part of the model. The parameters of the language model can potentially be estimated from very large quantities of English data.

2. A *translation model* that assigns a conditional probability $p(f|e)$ to any French/English pair of sentences. The parameters of this model will be estimated from the translation examples. The model involves two choices, both of which are conditioned on the English sentence $e_1 \dots e_l$: first, a choice of the length, m, of the French sentence; second, a choice of the m words $f_1 \dots f_m$.

Given these two components of the model, following the usual method in the noisy-channel approach, the output of the translation model on a new French sentence f is:

$$e^* = \arg\max_{e \in E} \quad p(e) \times p(f|e)$$

where E is the set of all sentences in English. Thus the score for a potential translation e is the product of two scores: first, the language-model score $p(e)$, which gives a prior distribution over which sentences are likely in English; second, the translation-model score $p(f|e)$, which indicates how likely we are to see the French sentence f as a translation of e.

Note that, as is usual in noisy-channel models, the model $p(f|e)$ appears to be "backwards": even though we are building a model for translation from French into English, we have a model of $p(f|e)$. The noisy-channel approach has used Bayes rule:

$$p(e|f) = \frac{p(e)p(f|e)}{\sum_e p(e)p(f|e)}$$

hence

$$\begin{aligned} \arg\max_{e \in E} p(e|f) &= \arg\max_{e \in E} \frac{p(e)p(f|e)}{\sum_e p(e)p(f|e)} \\ &= \arg\max_{e \in E} p(e)p(f|e) \end{aligned}$$

A major benefit of the noisy-channel approach is that it allows us to use a language model $p(e)$. This can be very useful in improving the fluency or grammaticality of the translation model's output.

The remainder of this note will be focused on the following questions:

- How can we define the translation model $p(f|e)$?
- How can we estimate the parameters of the translation model from the training examples $(f^{(k)}, e^{(k)})$ for $k = 1 \dots n$?

We will describe the IBM models—specifically, IBM models 1 and 2—for this problem. The IBM models were an early approach to SMT, and are now not widely used for translation: improved models (which we will cover in the next lecture) have been derived in recent work. However, they will be very useful to us, for the following reasons:

1. The models make direct use of the idea of *alignments*, and as a consequence allow us to recover alignments between French and English words in the training data. The resulting alignment models are of central importance in modern SMT systems.
2. The parameters of the IBM models will be estimated using the expectation maximization (EM) algorithm. The EM algorithm is widely used in statistical models for NLP and other problem domains. We will study it extensively later in the course: we use the IBM models, described here, as our first example of the algorithm.

3 The IBM Models

3.1 Alignments

We now turn to the problem of modeling the conditional probability $p(f|e)$ for any French sentence $f = f_1 \dots f_m$ paired with an English sentence $e = e_1 \dots e_l$.

Recall that $p(f|e)$ involves two choices: first, a choice of the length m of the French sentence; second, a choice of the words $f_1 \dots f_m$. We will assume that there is some distribution $p(m|l)$ that models the conditional distribution of French sentence length conditioned on the English sentence length. From now on, we take the length m to be fixed, and our focus will be on modeling the distribution

$$p(f_1 \dots f_m | e_1 \dots e_l, m)$$

i.e., the conditional probability of the words $f_1 \dots f_m$, conditioned on the English string $e_1 \dots e_l$, and the French length m.

It is very difficult to model $p(f_1 \dots f_m | e_1 \dots e_l, m)$ directly. A central idea in the IBM models was to introduce additional alignment variables to the problem. We will have alignment variables $a_1 \dots a_m$—that is, one alignment variable for each French word in the sentence—where each alignment variable can take any value in $\{0, 1, \dots, l\}$. The alignment variables will specify an alignment for each French word to some word in the English sentence.

Rather than attempting to define $p(f_1 \dots f_m | e_1 \dots e_l, m)$ directly, we will instead define a conditional distribution

$$p(f_1 \dots f_m, a_l \dots a_m | e_1 \dots e_l, m)$$

over French sequences $f_1 \dots f_m$ together with alignment variables $a_1 \dots a_m$. Having defined this model, we can then calculate $p(f_1 \dots f_m | e_1 \dots e_l, m)$ by summing over the alignment variables ("marginalizing out" the alignment variables):

$$p(f_1 \dots f_m | e_1 \dots e_l) = \sum_{a_1=0}^{l} \sum_{a_2=0}^{l} \sum_{a_3=0}^{l} \dots \sum_{a_m=0}^{l} p(f_1 \dots f_m, a_1 \dots a_m | e_1 \dots e_l)$$

We now describe the alignment variables in detail. Each alignment variable a_j specifies that the French word f_j is aligned to the English word e_{a_j}: we will see soon that intuitively, in the probabilistic model, word f_j will be generated from English word e_{a_j}. We define e_0 to be a special `NULL` word; so $a_j = 0$ specifies that word f_j is generated from the `NULL` word. We will see the role that the `NULL` symbol plays when we describe the probabilistic model.

As one example, consider a case where $l = 6$, $m = 7$, and

$e =$ *And the programme has been implemented*

$f =$ *Le programme a ete mis en application*

In this case the length of the French sentence, m, is equal to 7; hence we have alignment variables $a_1, a_2, \dots a_7$. As one alignment (which is quite plausible), we could have

$$a_1, a_2, \dots, a_7 = \langle 2, 3, 4, 5, 6, 6, 6 \rangle$$

specifying the following alignment:

Le	$\Rightarrow$	the
Programme	$\Rightarrow$	program
a	$\Rightarrow$	has
ete	$\Rightarrow$	been
mis	$\Rightarrow$	implemented
en	$\Rightarrow$	implemented
application	$\Rightarrow$	implemented

Note that each French word is aligned to exactly one English word. The alignment is many-to-one: more than one French word can be aligned to a single English word (e.g., *mis*, *en*, and *application* are all aligned to *implemented*). Some English words may be aligned to zero French words: for example the word *And* is not aligned to any French word in this example.

Note also that the model is asymmetric, in that there is no constraint that each English word is aligned to exactly one French word: each English word can be aligned to any number (zero or more) French words. We will return to this point later.

As another example alignment, we could have

$$a_1, a_2, \ldots a_7 = \langle 1, 1, 1, 1, 1, 1, 1 \rangle$$

specifying the following alignment:

Le	$\Rightarrow$	And
Programme	$\Rightarrow$	And
a	$\Rightarrow$	And
ete	$\Rightarrow$	And
mis	$\Rightarrow$	And
en	$\Rightarrow$	And
application	$\Rightarrow$	And

This is clearly not a good alignment for this example.

3.2 Alignment Models: IBM Model 2

We now describe a model for the conditional probability

$$p(f_1 \ldots f_m, a_1 \ldots a_m | e_1 \ldots e_l, m)$$

The model we describe is usually referred to as IBM model 2: we will use *IBM-M2* as shorthand for this model. Later we will describe how IBM model 1 is a special case of IBM model 2. The definition is as follows:

Definition 1 (IBM Model 2) *An IBM-M2 model consists of a finite set $\mathcal{E}$ of English words, a set $\mathcal{F}$ of French words, and integers M and L specifying the maximum length of French and English sentences respectively. The parameters of the model are as follows:*

- $t(f|e)$ *for any* $f \in \mathcal{F}$, $e \in \mathcal{E} \cup \{NULL\}$. *The parameter* $t(f|e)$ *can be interpreted as the conditional probability of generating French word* f *from English word* e.
- $q(j|i,l,m)$ *for any* $l \in \{1 \ldots L\}$, $m \in \{1 \ldots M\}$, $i \in \{1 \ldots m\}$, $j \in \{0 \ldots l\}$. *The parameter* $q(j|i,l,m)$ *can be interpreted as the probability of alignment variable* a_i *taking the value* j, *conditioned on the lengths* l *and* m *of the English and French sentences.*

Given these definitions, for any English sentence $e_1 \ldots e_l$ *where each* $e_j \in \mathcal{E}$, *for each length* m, *we define the conditional distribution over French sentences* $f_1 \ldots f_m$ *and alignments* $a_1 \ldots a_m$ *as*

$$p(f_1 \ldots f_m, a_1 \ldots a_m | e_1 \ldots e_l, m) = \prod_{i=1}^{m} q(a_i|i,l,m) t(f_i|e_{a_i})$$

Here we define e_0 *to be the* NULL *word.* □

To illustrate this definition, consider the previous example where $l = 6$, $m = 7$,

$e =$ *And the programme has been implemented*

$f =$ *Le programme a ete mis en application*

and the alignment variables are

$$a_1, a_2, \ldots a_7 = \langle 2, 3, 4, 5, 6, 6, 6 \rangle$$

specifying the following alignment:

Le	$\Rightarrow$	the
Programme	$\Rightarrow$	program
a	$\Rightarrow$	has
ete	$\Rightarrow$	been
mis	$\Rightarrow$	implemented
en	$\Rightarrow$	implemented
application	$\Rightarrow$	implemented

In this case we have

$$\begin{aligned} & p(f_1 \ldots f_m, a_1 \ldots a_m | e_1 \ldots e_l, m) \\ = \; & q(2|1,6,7) \times t(\textit{Le}|\textit{the}) \\ & \times q(3|2,6,7) \times t(\textit{Programme}|\textit{program}) \\ & \times q(4|3,6,7) \times t(\textit{a}|\textit{has}) \\ & \times q(5|4,6,7) \times t(\textit{ete}|\textit{been}) \\ & \times q(6|5,6,7) \times t(\textit{mis}|\textit{implemented}) \\ & \times q(6|6,6,7) \times t(\textit{en}|\textit{implemented}) \\ & \times q(6|7,6,7) \times t(\textit{application}|\textit{implemented}) \end{aligned}$$

Thus each French word has two associated terms: first, a choice of alignment variable, specifying which English word the word is aligned to; and second, a choice of the French word itself, based on the English word that was chosen in step 1. For example, for $f_5 =$ *mis* we first choose $a_5 = 6$, with probability $q(6|5,6,7)$, and then choose the word *mis*, based on the English word $e_6 =$ *implemented*, with probability $t(\textit{mis}|\textit{implemented})$.

Note that the alignment parameters, $q(j|i,l,m)$ specify a different distribution $\langle q(0|i,l,m), q(1|i,l,m), \ldots, q(l|i,l,m)\rangle$ for each possible value of the tuple i,l,m, where i is the position in the French sentence, l is the length of the English sentence, and m is the length of the French sentence. This will allow us, for example, to capture the tendency for words close to the beginning of the French sentence to be translations of words close to the beginning of the English sentence.

The model is certainly rather simple and naive. However, it captures some important aspects of the data.

3.3 Independence Assumptions in IBM Model 2

We now consider the independence assumptions underlying IBM Model 2. Take L to be a random variable corresponding to the length of the English sentence; $E_1 \ldots E_l$ to be a sequence of random variables corresponding to the words in the English sentence; M to be a random variable corresponding to the length of the French sentence; and $F_1 \ldots F_m$ and $A_1 \ldots A_m$ to be sequences of French words, and alignment variables. Our goal is to build a model of

$$P(F_1 = f_1 \ldots F_m = f_m, A_1 = a_1 \ldots A_m = a_m | E_1 = e_1 \ldots E_l = e_l, L = l, M = m)$$

As a first step, we can use the chain rule of probabilities to decompose this into two terms:

$$P(F_1 = f_1 \ldots F_m = f_m, A_1 = a_1 \ldots A_m = a_m | E_1 = e_1 \ldots E_l = e_l, L = l, M = m)$$

$$
\begin{aligned}
= \quad & P(A_1 = a_1 \ldots A_m = a_m | E_1 = e_1 \ldots E_l = e_l, L = l, M = m) \\
& \times P(F_1 = f_1 \ldots F_m = f_m | A_1 = a_1 \ldots A_m = a_m, E_1 = e_1 \ldots E_l = e_l, L = l, M = m)
\end{aligned}
$$

We'll now consider these two terms separately.

First, we make the following independence assumptions:

$$
\begin{aligned}
& P(A_1 = a_1 \ldots A_m = a_m | E_1 = e_1 \ldots E_l = e_l, L = l, M = m) \\
= \quad & \prod_{i=1}^{m} P(A_i = a_i | A_1 = a_1 \ldots A_{i-1} = a_{i-1}, E_1 = e_1 \ldots E_l = e_l, L = l, M = m) \\
= \quad & \prod_{i=1}^{m} P(A_i = a_i | L = l, M = m)
\end{aligned}
$$

The first equality is exact, by the chain rule of probabilities. The second equality corresponds to a very strong independence assumption: namely, that the distribution of the random variable A_i depends only on the values for the random variables L and M (it is independent of the English words $E_1 \ldots E_l$, and of the other alignment variables). Finally, we make the assumption that

$$P(A_i = a_i | L = l, M = m) = q(a_i | i, l, m)$$

where each $q(a_i|i, l, m)$ is a parameter of our model.

Next, we make the following assumption:

$$
\begin{aligned}
& P(F_1 = f_1 \ldots F_m = f_m | A_1 = a_1 \ldots A_m = a_m, E_1 = e_1 \ldots E_l = e_l, L = l, M = m) \\
= \quad & \prod_{i=1}^{m} P(F_i = f_i | F_1 = f_1 \ldots F_{i-1} = f_{i-1}, A_1 = a_1 \ldots A_m = a_m, E_1 = e_1 \ldots E_l = e_l, L = l, M = m) \\
= \quad & \prod_{i=1}^{m} P(F_i = f_i | E_{a_i} = e_{a_i})
\end{aligned}
$$

The first step is again exact, by the chain rule. In the second step, we assume that the value for F_i depends only on E_{a_i}: i.e., on the identity of the English word to which F_i is aligned. Finally, we make the assumption that for all i,

$$P(F_i = f_i | E_{a_i} = e_{a_i}) = t(f_i | e_{a_i})$$

where each $t(f_i|e_{a_i})$ is a parameter of our model.

4 Applying IBM Model 2

The next section describes a parameter estimation algorithm for IBM Model 2. Before getting to this, we first consider an important question: what is IBM Model 2 useful for?

The original motivation was the full machine translation problem. Once we have estimated parameters $q(j|i,l,m)$ and $t(f|e)$ from data, we have a distribution

$$p(f,a|e)$$

for any French sentence f, alignment sequence a, and English sentence e; from this we can derive a distribution

$$p(f|e) = \sum_a p(f,a|e)$$

Finally, assuming we have a language model $p(e)$, we can define the translation of any French sentence f to be

$$\arg\max_e p(e)p(f|e) \tag{1}$$

where the $\arg\max$ is taken over all possible English sentences. The problem of finding the $\arg\max$ in Eq. 1 is often referred to as the *decoding problem*. Solving the decoding problem is a computationally very hard problem, but various approximate methods have been derived.

In reality, however, IBM Model 2 is not a particularly good translation model. In later lectures we'll see alternative, state-of-the-art, models that are far more effective.

The IBM models are, however, still crucial in modern translation systems, for two reasons:

1. The lexical probabilities $t(f|e)$ are directly used in various translation systems.

2. Most importantly, the alignments derived using IBM models are of direct use in building modern translation systems.

Let's consider the second point in more detail. Assume that we have estimated our parameters $t(f|e)$ and $q(j|i,l,m)$ from a training corpus (using the parameter estimation algorithm described in the next section). Given any training example consisting of an English sentence e paired with a French sentence f, we can then find the most probable alignment under the model:

$$\arg\max_{a_1 \ldots a_m} p(a_1 \ldots a_m | f_1 \ldots f_m, e_1 \ldots e_l, m) \tag{2}$$

Because the model takes such as simple form, finding the solution to Eq. 2 is straightforward. In fact, a simple derivation shows that we simply define

$$a_i = \arg\max_{j \in \{0 \ldots l\}} \left(q(j|i,l,m) \times t(f_i|e_j) \right)$$

for $i = 1 \ldots m$. So for each French word i, we simply align it to the English position j which maximizes the product of two terms: first, the alignment probability $q(j|i, l, m)$; and second, the translation probability $t(f_i|e_j)$.

5 Parameter Estimation

This section describes methods for estimating the $t(f|e)$ parameters and $q(j|i, l, m)$ parameters from translation data. We consider two scenarios: first, estimation with *fully observed data*; and second, estimation with *partially observed data*. The first scenario is unrealistic, but will be a useful warm-up before we get to the second, more realistic case.

5.1 Parameter Estimation with Fully Observed Data

We now turn to the following problem: how do we estimate the parameters $t(f|e)$ and $q(j|i, l, m)$ of the model? We will assume that we have a training corpus $\{f^{(k)}, e^{(k)}\}_{k=1}^{n}$ of translations. Note however, that a crucial piece of information is missing in this data: *we do not know the underlying alignment for each training example*. In this sense we will refer to the data being only *partially observed*, because some information—i.e., the alignment for each sentence—is missing. Because of this, we will often refer to the alignment variables as being *hidden variables*. In spite of the presence of hidden variables, we will see that we can in fact estimate the parameters of the model.

Note that we could presumably employ humans to annotate data with underlying alignments (in a similar way to employing humans to annotate underlying parse trees, to form a treebank resource). However, we wish to avoid this because manual annotation of alignments would be an expensive task, taking a great deal of time for reasonable size translation corpora—moreover, each time we collect a new corpus, we would have to annotate it in this way.

In this section, as a warm-up for the case of partially-observed data, we will consider the case of *fully-observed data*, where each training example does in fact consist of a triple $(f^{(k)}, e^{(k)}, a^{(k)})$ where $f^{(k)} = f_1^{(k)} \ldots f_{m_k}^{(k)}$ is a French sentence, $e^{(k)} = e_1^{(k)} \ldots e_{l_k}^{(k)}$ is an English sentence, and $a^{(k)} = a_1^{(k)} \ldots a_{m_k}^{(k)}$ is a sequence of alignment variables. Solving this case will be be useful in developing the algorithm for partially-observed data.

The estimates for fully-observed data are simple to derive. Define $c(e, f)$ to be the number of times word e is aligned to word f in the training data, and $c(e)$ to be the number of times that e is aligned to *any* French word. In addition, define $c(j|i, l, m)$ to be the number of times we see an English sentence of length

l, and a French sentence of length m, where word i in French is aligned to word j in English. Finally, define $c(i, l, m)$ to be the number of times we see an English sentence of length l together with a French sentence of length m. Then the maximum-likelihood estimates are

$$t_{ML}(f|e) = \frac{c(e, f)}{c(e)}$$

$$q_{ML}(j|i, l, m) = \frac{c(j|i, l, m)}{c(i, l, m)}$$

Thus to estimate parameters we simply compile counts from the training corpus, then take ratios of these counts.

Figure 1 shows an algorithm for parameter estimation with fully observed data. The algorithm for partially-observed data will be a direct modification of this algorithm. The algorithm considers all possible French/English word pairs in the corpus, which could be aligned: i.e., all possible (k, i, j) tuples where $k \in \{1 \dots n\}$, $i \in \{1 \dots m_k\}$, and $j \in \{0 \dots l_k\}$. For each such pair of words, we have $a_i^{(k)} = j$ if the two words are aligned. In this case we increment the relevant $c(e, f)$, $c(e)$, $c(j|i, l, m)$ and $c(i, l, m)$ counts. If $a_i^{(k)} \neq j$ then the two words are not aligned, and no counts are incremented.

5.2 Parameter Estimation with Partially Observed Data

We now consider the case of partially-observed data, where the alignment variables $a^{(k)}$ are not observed in the training corpus. The algorithm for this case is shown in figure 2. There are two important differences for this algorithm from the algorithm in figure 1:

- The algorithm is iterative. We begin with some initial value for the t and q parameters: for example, we might initialize them to random values. At each iteration we first compile some "counts" $c(e)$, $c(e, f)$, $c(j|i, l, m)$ and $c(i, l, m)$ based on the data together with our current estimates of the parameters. We then re-estimate the parameters using these counts, and iterate.

- The counts are calculated using a similar definition to that in figure 1, but with one crucial difference: rather than defining

  $$\delta(k, i, j) = 1 \text{ if } a_i^{(k)} = j, 0 \text{ otherwise}$$

  we use the definition

  $$\delta(k, i, j) = \frac{q(j|i, l_k, m_k)t(f_i^{(k)}|e_j^{(k)})}{\sum_{j=0}^{l_k} q(j|i, l_k, m_k)t(f_i^{(k)}|e_j^{(k)})}$$

Input: A training corpus $(f^{(k)}, e^{(k)}, a^{(k)})$ for $k = 1 \ldots n$, where $f^{(k)} = f_1^{(k)} \ldots f_{m_k}^{(k)}$, $e^{(k)} = e_1^{(k)} \ldots e_{l_k}^{(k)}$, $a^{(k)} = a_1^{(k)} \ldots a_{m_k}^{(k)}$.

Algorithm:

- Set all counts $c(\ldots) = 0$
- For $k = 1 \ldots n$
  - For $i = 1 \ldots m_k$
    - For $j = 0 \ldots l_k$

$$
\begin{aligned}
c(e_j^{(k)}, f_i^{(k)}) &\leftarrow c(e_j^{(k)}, f_i^{(k)}) + \delta(k, i, j) \\
c(e_j^{(k)}) &\leftarrow c(e_j^{(k)}) + \delta(k, i, j) \\
c(j|i, l_k, m_k) &\leftarrow c(j|i, l_k, m_k) + \delta(k, i, j) \\
c(i, l_k, m_k) &\leftarrow c(i, l_k, m_k) + \delta(k, i, j)
\end{aligned}
$$

where $\delta(k, i, j) = 1$ if $a_i^{(k)} = j$, 0 otherwise.

Output:

$$t_{ML}(f|e) = \frac{c(e, f)}{c(e)} \qquad q_{ML}(j|i, l, m) = \frac{c(j|i, l, m)}{c(i, l, m)}$$

Figure 1: The parameter estimation algorithm for IBM model 2, for the case of fully-observed data.

Input: A training corpus $(f^{(k)}, e^{(k)})$ for $k = 1 \ldots n$, where $f^{(k)} = f_1^{(k)} \ldots f_{m_k}^{(k)}$, $e^{(k)} = e_1^{(k)} \ldots e_{l_k}^{(k)}$. An integer S specifying the number of iterations of training.
Initialization: Initialize $t(f|e)$ and $q(j|i,l,m)$ parameters (e.g., to random values).

Algorithm:

- For $s = 1 \ldots S$
  - Set all counts $c(\ldots) = 0$
  - For $k = 1 \ldots n$
    - For $i = 1 \ldots m_k$
      - For $j = 0 \ldots l_k$

$$
\begin{aligned}
c(e_j^{(k)}, f_i^{(k)}) &\leftarrow c(e_j^{(k)}, f_i^{(k)}) + \delta(k,i,j) \\
c(e_j^{(k)}) &\leftarrow c(e_j^{(k)}) + \delta(k,i,j) \\
c(j|i,l_k,m_k) &\leftarrow c(j|i,l_k,m_k) + \delta(k,i,j) \\
c(i,l_k,m_k) &\leftarrow c(i,l_k,m_k) + \delta(k,i,j)
\end{aligned}
$$

where

$$\delta(k,i,j) = \frac{q(j|i,l_k,m_k)t(f_i^{(k)}|e_j^{(k)})}{\sum_{j=0}^{l_k} q(j|i,l_k,m_k)t(f_i^{(k)}|e_j^{(k)})}$$

  - Set

$$t(f|e) = \frac{c(e,f)}{c(e)} \qquad q(j|i,l,m) = \frac{c(j|i,l,m)}{c(i,l,m)}$$

Output: parameters $t(f|e)$ and $q(j|i,l,m)$

Figure 2: The parameter estimation algorithm for IBM model 2, for the case of partially-observed data.

where the q and t values are our current parameter estimates.

Let's consider this last definition in more detail. We can in fact show the following identity:

$$P(A_i = j|e_1 \dots e_l, f_1 \dots f_m, m) = \frac{q(j|i,l,m)t(f_i|e_j)}{\sum_{j=0}^{l_k} q(j|i,l,m)t(f_i|e_j)}$$

where $P(A_i = j|e_1 \dots e_l, f_1 \dots f_m, m)$ is the conditional probability of the alignment variable a_i taking the value j, under the current model parameters. Thus we have effectively filled in the alignment variables probabilistically, using our current parameter estimates. This in contrast to the fully observed case, where we could simply define $\delta(k, i, j) = 1$ if $a_i^{(k)} = j$, and 0 otherwise.

As an example, consider our previous example where $l = 6$, $m = 7$, and

$e^{(k)}$ = *And the programme has been implemented*

$f^{(k)}$ = *Le programme a ete mis en application*

The value for $\delta(k, 5, 6)$ for this example would be the current model's estimate of the probability of word f_5 being aligned to word e_6 in the data. It would be calculated as

$$\delta(k, 5, 6) = \frac{q(6|5,6,7) \times t(\textit{mis}|\textit{implemented})}{\sum_{j=0}^{6} q(j|5,6,7) \times t(\textit{mis}|e_j)}$$

Thus the numerator takes into account the translation parameter $t(\textit{mis}|\textit{implemented})$ together with the alignment parameter $q(6|5, 6, 7)$; the denominator involves a sum over terms, where we consider each English word in turn.

The algorithm in figure 2 is an instance of the *expectation-maximization* (EM) algorithm. The EM algorithm is very widely used for parameter estimation in the case of partially-observed data. The counts $c(e)$, $c(e, f)$ and so on are referred to as *expected counts*, because they are effectively expected counts under the distribution

$$p(a_1 \dots a_m|f_1 \dots f_m, e_1 \dots e_l, m)$$

defined by the model. In the first step of each iteration we calculate the expected counts under the model. In the second step we use these expected counts to re-estimate the t and q parmeters. We iterate this two-step procedure until the parameters converge (this often happens in just a few iterations).

6 More on the EM Algorithm: Maximum-likelihood Estimation

Soon we'll trace an example run of the EM algorithm, on some simple data. But first, we'll consider the following question: how can we justify the algorithm? What are its formal guarantees, and what function is it optimizing?

In this section we'll describe how the EM algorithm is attempting to find the maximum-likelihood estimates for the data. For this we'll need to introduce some notation, and in particular, we'll need to carefully specify what exactly is meant by maximum-likelihood estimates for IBM model 2.

First, consider the parameters of the model. There are two types of parameters: the translation parameters $t(f|e)$, and the alignment parameters $q(j|i,l,m)$. We will use t to refer to the vector of translation parameters,

$$t = \{t(f|e) : f \in F, e \in E \cup \{\texttt{NULL}\}\}$$

and q to refer to the vector of alignment parameters,

$$q = \{q(j|i,l,m) : l \in \{1 \ldots L\}, m \in \{1 \ldots M\}, j \in \{0 \ldots l\}, i \in \{1 \ldots m\}\}$$

We will use $\mathcal{T}$ to refer to the *parameter space* for the translation parameters—that is, the set of valid settings for the translation parameters, defined as follows:

$$\mathcal{T} = \{t : \forall e, f, t(f|e) \geq 0; \quad \forall e \in E \cup \{\texttt{NULL}\}, \sum_{f \in F} t(f|e) = 1\}$$

and we will use $\mathcal{Q}$ to refer to the parameter space for the alignment parameters,

$$\mathcal{Q} = \{q : \forall j, i, l, m,\ q(j|i,l,m) \geq 0; \quad \forall i, l, m,\ \sum_{j=0}^{l} q(j|i,l,m) = 1\}$$

Next, consider the probability distribution under the model. This depends on the parameter settings t and q. We will introduce notation that makes this dependence explicit. We write

$$p(f, a|e, m; t, q) = \prod_{i=1}^{m} q(a_i|i,l,m)t(f_i|e_{a_i})$$

as the conditional probability of a French sentence $f_1 \ldots f_m$, with alignment variables $a_1 \ldots a_m$, conditioned on an English sentence $e_1 \ldots e_l$, and the French sentence length m. The function $p(f, a|e, m; t, q)$ varies as the parameter vectors t

and q vary, and we make this dependence explicit by including t and q after the ";" in this expression.

As we described before, we also have the following distribution:

$$p(f|e,m;t,q) = \sum_{a \in \mathcal{A}(l,m)} p(f,a|e,m;t,q)$$

where $\mathcal{A}(l,m)$ is the set of all possible settings for the alignment variables, given that the English sentence has length l, and the French sentence has length m:

$$\mathcal{A}(l,m) = \{(a_1 \dots a_m) : a_j \in \{0 \dots l\} \text{ for } j = 1 \dots m\}$$

So $p(f|e,m;t,q)$ is the conditional probability of French sentence f, conditioned on e and m, under parameter settings t and q.

Now consider the parameter estimation problem. We have the following set-up:

- The input to the parameter estimation algorithm is a set of training examples, $(f^{(k)}, e^{(k)})$, for $k = 1 \dots n$.
- The output of the parameter estimation algorithm is a pair of parameter vectors $t \in \mathcal{T}, q \in \mathcal{Q}$.

So how should we choose the parameters t and q? We first consider a single training example, $(f^{(k)}, e^{(k)})$, for some $k \in \{1 \dots n\}$. For any parameter settings t and q, we can consider the probability

$$p(f^{(k)}|e^{(k)}, m_k; t, q)$$

under the model. As we vary the parameters t and q, this probability will vary. Intuitively, a good model would make this probability as high as possible.

Now consider the entire set of training examples. For any parameter settings t and q, we can evaluate the probability of the entire training sample, as follows:

$$\prod_{k=1}^{n} p(f^{(k)}|e^{(k)}, m_k; t, q)$$

Again, this probability varies as the paramters t and q vary; intuitively, we would like to choose parameter settings t and q which make this probability as high as possible. This leads to the following definition:

Definition 2 (Maximum-likelihood (ML) estimates for IBM model 2) *The ML estimates for IBM model 2 are*

$$(t_{ML}, q_{ML}) = \arg \max_{t \in \mathcal{T}, q \in \mathcal{Q}} L(t,q)$$

where

$$\begin{aligned} L(t,q) &= \log\left(\prod_{k=1}^{n} p(f^{(k)}|e^{(k)}, m_k; t, q)\right) \\ &= \sum_{k=1}^{n} \log p(f^{(k)}|e^{(k)}, m_k; t, q) \\ &= \sum_{k=1}^{n} \log \sum_{a \in \mathcal{A}(l_k, m_k)} p(f^{(k)}, a|e^{(k)}, m_k; t, q) \end{aligned}$$

We will refer to the function $L(t,q)$ as the log-likelihood function. □

Under this definition, the ML estimates are defined as maximizing the function

$$\log\left(\prod_{k=1}^{n} p(f^{(k)}|e^{(k)}, m_k; t, q)\right)$$

It is important to realise that this is equivalent to maximizing

$$\prod_{k=1}^{n} p(f^{(k)}|e^{(k)}, m_k; t, q)$$

because $\log$ is a monotonically increasing function, hence maximizing a function $\log f(t,q)$ is equivalent to maximizing $f(t,q)$. The $\log$ is often used because it makes some mathematical derivations more convenient.

We now consider the function $L(t,q)$ which is being optimized. This is actually a difficult function to deal with: for one thing, there is no analytical solution to the optimization problem

$$(t,q) = \arg \max_{t \in \mathcal{T}, q \in \mathcal{Q}} L(t,q) \tag{3}$$

By an "analytical" solution, we mean a simple, closed-form solution. As one example of an analytical solution, in language modeling, we found that the maximum-likelihood estimates of trigram parameters were

$$q_{ML}(w|u,v) = \frac{count(u,v,w)}{count(u,v)}$$

Unfortunately there is no similar simple expression for parameter settings that maximize the expression in Eq. 3.

A second difficulty is that $L(t,q)$ is *not* a convex function. Figure 3 shows examples of convex and non-convex functions for the simple case of a function $f(x)$ where x is a scalar value (as opposed to a vector). A convex function has a single

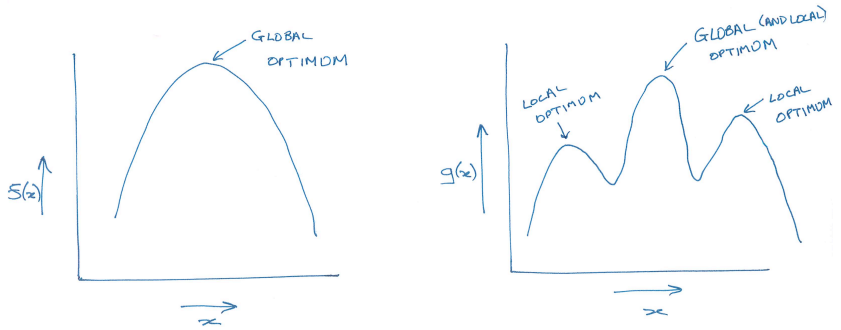


Figure 3: Examples of convex and non-convex functions in a single dimension. On the left, $f(x)$ is convex. On the right, $g(x)$ is non-convex.

global optimum, and intuitively, a simple hill-climbing algorithm will climb to this point. In contrast, the second function in figure 3 has multiple "local" optima, and intuitively a hill-climbing procedure may get stuck in a local optimum which is not the global optimum.

The formal definitions of convex and non-convex functions are beyond the scope of this note. However, in brief, there are many results showing that convex functions are "easy" to optimize (i.e., we can design efficient algorithms that find the $\arg\max$), whereas non-convex functions are generally much harder to deal with (i.e., we can often show that finding the $\arg\max$ is computationally hard, for example it is often NP-hard). In many cases, the best we can hope for is that the optimization method finds a *local* optimum of a non-convex function.

In fact, this is precisely the case for the EM algorithm for model 2. It has the following guarantees:

Theorem 1 (Convergence of the EM algorithm for IBM model 2) *We use $t^{(s)}$ and $q^{(s)}$ to refer to the parameter estimates after s iterations of the EM algorithm, and $t^{(0)}$ and $q^{(0)}$ to refer to the initial parameter estimates. Then for any $s \geq 1$, we have*

$$L(t^{(s)}, q^{(s)}) \geq L(t^{(s-1)}, q^{(s-1)}) \tag{4}$$

Furthermore, under mild conditions, in the limit as $s \rightarrow \infty$, the parameter estimates $(t^{(s)}, q^{(s)})$ converge to a local optimum of the log-likelihood function.

Later in the class we will consider the EM algorithm in much more detail:

we will show that it can be applied to a quite broad range of models in NLP, and we will describe it's theoretical properties in more detail. For now though, this convergence theorem is the most important property of the algorithm.

Eq. 4 states that the log-likelihood is strictly non-decreasing: at each iteration of the EM algorithm, it cannot decrease. However this does not rule out rather uninteresting cases, such as

$$L(t^{(s)}, q^{(s)}) = L(t^{(s-1)}, q^{(s-1)})$$

for all s. The second condition states that the method does in fact converge to a local optimum of the log-likelihood function.

One important consequence of this result is the following: *the EM algorithm for IBM model 2 may converge to different parameter estimates, depending on the initial parameter values $t^{(0)}$ and $q^{(0)}$*. This is because the algorithm may converge to a different local optimum, depending on its starting point. In practice, this means that some care is often required in initialization (i.e., choice of the initial parameter values).

7 Initialization using IBM Model 1

As described in the previous section, the EM algorithm for IBM model 2 may be sensitive to initialization: depending on the inital values, it may converge to different local optima of the log-likelihood function.

Because of this, in practice the choice of a good heuristic for parameter initialization is important. A very common method is to use IBM Model 1 for this purpose. We describe IBM Model 1, and the initialization method based on IBM Model 1, in this section.

Recall that in IBM model 2, we had parameters

$$q(j|i, l, m)$$

which are interpreted as the conditional probability of French word f_i being aligned to English word e_j, given the French length m and the English length l. In IBM Model 1, we simply assume that for all i, j, l, m,

$$q(j|i, l, m) = \frac{1}{l+1}$$

Thus there is a uniform probability distribution over all $l + 1$ possible English words (recall that the English sentence is $e_1 \ldots e_l$, and there is also the possibility that $j = 0$, indicating that the French word is aligned to $e_0 = \texttt{NULL}$.). This leads to the following definition:

Definition 3 (IBM Model 1) *An IBM-M1 model consists of a finite set $\mathcal{E}$ of English words, a set $\mathcal{F}$ of French words, and integers M and L specifying the maximum length of French and English sentences respectively. The parameters of the model are as follows:*

- *$t(f|e)$ for any $f \in \mathcal{F}$, $e \in \mathcal{E} \cup \{NULL\}$. The parameter $t(f|e)$ can be interpreted as the conditional probability of generating French word f from English word e.*

Given these definitions, for any English sentence $e_1 \dots e_l$ where each $e_j \in \mathcal{E}$, for each length m, we define the conditional distribution over French sentences $f_1 \dots f_m$ and alignments $a_1 \dots a_m$ as

$$p(f_1 \dots f_m, a_1 \dots a_m | e_1 \dots e_l, m) = \prod_{i=1}^{m} \frac{1}{(l+1)} \times t(f_i | e_{a_i}) = \frac{1}{(l+1)^m} \prod_{i=1}^{m} t(f_i | e_{a_i})$$

Here we define e_0 to be the NULL *word.* □

The parameters of IBM Model 1 can be estimated using the EM algorithm, which is very similar to the algorithm for IBM Model 2. The algorithm is shown in figure 4. The only change from the algorithm for IBM Model 2 comes from replacing

$$\delta(k,i,j) = \frac{q(j|i,l_k,m_k)t(f_i^{(k)}|e_j^{(k)})}{\sum_{j=0}^{l_k} q(j|i,l_k,m_k)t(f_i^{(k)}|e_j^{(k)})}$$

with

$$\delta(k,i,j) = \frac{\frac{1}{(l^{(k)}+1)}t(f_i^{(k)}|e_j^{(k)})}{\sum_{j=0}^{l_k} \frac{1}{(l^{(k)}+1)}t(f_i^{(k)}|e_j^{(k)})} = \frac{t(f_i^{(k)}|e_j^{(k)})}{\sum_{j=0}^{l_k} t(f_i^{(k)}|e_j^{(k)})}$$

reflecting the fact that in Model 1 we have

$$q(j|i,l_k,m_k) = \frac{1}{(l^{(k)}+1)}$$

A key property of IBM Model 1 is the following:

Proposition 1 *Under mild conditions, the EM algorithm in figure 4 converges to the global optimum of the log-likelihood function under IBM Model 1.*

Thus for IBM Model 1, we have a guarantee of convergence to the *global* optimum of the log-likelihood function. Because of this, the EM algorithm will converge to the same value, regardless of initialization. This suggests the following procedure for training the parameters of IBM Model 2:

Input: A training corpus $(f^{(k)}, e^{(k)})$ for $k = 1 \ldots n$, where $f^{(k)} = f_1^{(k)} \ldots f_{m_k}^{(k)}$, $e^{(k)} = e_1^{(k)} \ldots e_{l_k}^{(k)}$. An integer S specifying the number of iterations of training.
Initialization: Initialize $t(f|e)$ parameters (e.g., to random values).

Algorithm:

- For $s = 1 \ldots S$
  - Set all counts $c(\ldots) = 0$
  - For $k = 1 \ldots n$
    - For $i = 1 \ldots m_k$
      - For $j = 0 \ldots l_k$

$$
\begin{aligned}
c(e_j^{(k)}, f_i^{(k)}) &\leftarrow c(e_j^{(k)}, f_i^{(k)}) + \delta(k, i, j) \\
c(e_j^{(k)}) &\leftarrow c(e_j^{(k)}) + \delta(k, i, j) \\
c(j|i, l_k, m_k) &\leftarrow c(j|i, l_k, m_k) + \delta(k, i, j) \\
c(i, l_k, m_k) &\leftarrow c(i, l_k, m_k) + \delta(k, i, j)
\end{aligned}
$$

where

$$\delta(k, i, j) = \frac{t(f_i^{(k)}|e_j^{(k)})}{\sum_{j=0}^{l_k} t(f_i^{(k)}|e_j^{(k)})}$$

  - Set

$$t(f|e) = \frac{c(e, f)}{c(e)}$$

Output: parameters $t(f|e)$

Figure 4: The parameter estimation algorithm for IBM model 1, for the case of partially-observed data.

1. Estimate the t parameters using the EM algorithm for IBM Model 1, using the algorithm in figure 4.

2. Estimate parameters of IBM Model 2 using the algorithm in figure 2. To initialize this model, use: 1) the $t(f|e)$ parameters estimated under IBM Model 1, in step 1; 2) random values for the $q(j|i, l, m)$ parameters.

Intuitively, if IBM Model 1 leads to reasonable estimates for the t parameters, this method should generally perform better for IBM Model 2. This is often the case in practice.

See the lecture slides for an example of parameter estimation for IBM Model 2, using this heuristic.

Phrase-Based Translation Models

Michael Collins

April 10, 2013

1 Introduction

In previous lectures we've seen IBM translation models 1 and 2. In this note we will describe *phrase-based translation models*. Phrase-based translation models give much improved translations over the IBM models, and give state-of-the-art translations for many pairs of languages.

Crucially, phrase-based translation models allow lexical entries with more than one word on either the source-language or target-language side: for example, we might have a lexical entry

$$(\text{le chien}, \text{the dog})$$

specifying that the string le chien in French can be translated as the dog in English. The option of having multi-word expressions on either the source or target-language side is a significant departure from IBM models 1 and 2, which are essentially word-to-word translation models (i.e., they assume that each French word is generated from a single English word). Multi-word expressions are extremely useful in translation; this is the main reason for the improvements that phrase-based translation models give.

More formally, a phrase-based lexicon is defined as follows:

Definition 1 (Phrase-based lexicon) *A phrase-based lexicon $\mathcal{L}$ is a set of lexical entries, where each lexical entry is a tuple (f, e, g) where:*

- *f is a sequence of one or more foreign words.*
- *e is a sequence of one or more English words.*
- *g is a "score" for the lexical entry. The score could be any value in the reals.*

Note that there is no restriction that the number of foreign words and English words in a lexical entry should be equal. For example, the following entries would be allowed:

$$(\text{au}, \text{to the}, 0.5)$$

$$(\text{au banque}, \text{to the bank}, 0.01)$$

$$(\text{allez au banque}, \text{go to the bank}, -2.5)$$

(similar cases, where there are fewer English words than French words, would also be allowed). This flexibility in the definition of lexical entries is important, because in many cases it is very useful to have a lexical entry where the number of foreign and English words are not equal.

We'll soon describe how a phrasal lexicon $\mathcal{L}$ can be used in translation. First, however, we describe how a phrasal lexicon can be learned from a set of example translations.

2 Learning Phrasal Lexicons from Translation Examples

As before, we'll assume that our training data consists of English sentences $e^{(k)} = e_1^{(k)} \dots e_{l_k}^{(k)}$ paired with French sentences $f^{(k)} = f_1^{(k)} \dots f_{m_k}^{(k)}$, for $k = 1 \dots n$. Here the integer l_k is the length of the k'th English sentence, and $e_j^{(k)}$ is the j'th word in the k'th English sentence. The integer m_k is the length of the k'th French sentence, and $f_i^{(k)}$ is the i'th word in the k'th French sentence.

In addition to the sentences themselves, we will also assume that we have an *alignment matrix* for each training example. The alignment matrix $A^{(k)}$ for the k'th example has $l_k \times m_k$ entries, where

$$A_{i,j}^{(k)} = 1 \quad \text{if French word } i \text{ is aligned to English word } j, \text{ 0 otherwise}$$

Note that this representation is more general than the alignments considered for IBM models 1 and 2. In those models, we had alignment variables a_i for $i \in \{1 \dots m_k\}$, specifying which English word the i'th French word is aligned to. By definition, in IBM models 1 and 2 each French word could only be aligned to a single English word. With an alignment matrix $A_{i,j}^{(k)}$, the alignments can be many-to-many; for example, a given French word could be aligned to more than one English word (i.e., for a given i, we could have $A_{i,j}^{(k)} = 1$ for more than one value of j).

We'll remain agnostic as to how the alignment matrices $A^{(k)}$ are derived. In practice, a common method is something like the following (see the lecture slides, and the slides from Philipp Koehn's tutorial, for more details). First, we train IBM model 2, using the EM algorithm described in the previous lecture. Second, we use various heuristics to extract alignment matrices from the IBM model's output on each training example. To be specific, a very simple method would be as follows (the method is too naive to be used in practice, but will suffice as an example):

- Use the training examples $e^{(k)}, f^{(k)}$ for $k = 1 \dots n$ to train IBM model 2 using the EM algorithm described in the previous lecture. For any English string e, French string f, and French length m, this model gives a conditional probability $p(f, a|e, m)$.
- For each training example, define

  $$a^{(k)} = \arg\max_a p(f^{(k)}, a|e^{(k)}, m_k)$$

  i.e., $a^{(k)}$ is the most likely alignment under the model, for the k'th example (see the notes on IBM models 1 and 2 for how to compute this).

- Define

$$A^{(k)}_{i,j} = 1 \quad \text{if } a^{(k)}_i = j, \text{ 0 otherwise}$$

Assuming that we have derived an alignment matrix for each training example, we can now describe the method for extracting a phrase-based lexicon from a set of translation examples. Figure 1 shows a simple algorithm for this purpose. The input to the algorithm is a set of translation examples, with an alignment matrix for each training example. The algorithm iterates over all training examples ($k = 1 \ldots n$), and over all potential phrase pairs, where a phrase pair is a pair $(s,t),(s',t')$ where (s,t) is a sub-sequence within the source language sentence, and (s',t') is a sub-sequence within the target language sentence. For example, consider the case where the training example consists of the following sentences:

$$f^{(k)} = \text{wir müssen auch diese kritik ernst nehmen}$$

$$e^{(k)} = \text{we must also take these criticisms seriously}$$

then $(s,t) = (1,2)$, $(s',t') = (2,5)$ would correspond to the potential lexical entry

wir müssen, must also take these

For each possible (s,t), (s',t') pair, we test if it is *consistent* with the alignment matrix $A^{(k)}$: the function `consistent`$(A^{(k)},(s,t),(s',t'))$ returns true if the potential lexical entry ((s,t), (s',t')) is consistent with the alignment matrix for the training example. See figure 2 for the definition of the `consistent` function. Intuitively, the function checks whether any alignments from English or foreign words in the proposed lexical entry are to words that are "outside" the proposed entry. If any alignments are to outside words, then the proposed entry is not consistent. In addition, the function checks that there is at least one word in the English phrase aligned to some word in the foreign phrase.

For those phrases that are consistent, we add the lexical entry (f,e) to the lexicon, where $f = f_s \ldots f_t$, and $e = e_{s'} \ldots e_{t'}$. We also increment the counts $c(e,f)$ and $c(e)$, corresponding to the number of times that the lexical entry (f,e) is seen in the data, and the number of times the English string e is seen paired with *any* foreign phrase f. Finally, having extracted all lexical entries for the corpus, we define the score for any phrase (f,e) as

$$\log \frac{c(e,f)}{c(e)}$$

This can be interpreted as an estimate of the log-conditional probability of foreign phrase f, given English phrase e.

It is worth noting that these probabilities are in some sense heuristic—it is not clear what probabilistic model is underlying the overall model. They will, however, be very useful when translating with the model.

3 Translation with Phrase-Based Models

The previous described how a phrase-based lexicon can be derived from a set of training examples. In this section we describe how the phrase-based lexicon can be used to define a set of translations for

Inputs: $e^{(k)}, f^{(k)}, A^{(k)}$ for $k = 1 \ldots n$

Initialization: $\mathcal{L} = \emptyset$

Algorithm:

- For $k = 1 \ldots n$
  - For $s = 1 \ldots m_k$, for $t = s \ldots m_k$
    - For $s' = 1 \ldots l_k$, for $t' = s' \ldots l_k$
      - If $\texttt{consistent}(A^{(k)}, (s,t), (s',t')) = \texttt{True}$
        (1) Define $f = f_s^{(k)} \ldots f_t^{(k)}$, define $e = e_{s'}^{(k)} \ldots e_{t'}^{(k)}$
        (2) Set $\mathcal{L} = \mathcal{L} \cup \{(f, e)\}$
        (3) $c(e, f) = c(e, f) + 1$
        (4) $c(e) = c(e) + 1$
- For each $(f, e) \in \mathcal{L}$ create a lexical entry (f, e, g) where

$$g = \log \frac{c(e, f)}{c(e)}$$

Figure 1: An algorithm for deriving a phrasal lexicon from a set of training examples with alignments. The function $\texttt{consistent}(A^{(k)}, (s,t), (s',t'))$ is defined in figure 2.

Definition of $\texttt{consistent}(A, (s,t), (s',t'))$:
(Recall that A is an alignment matrix with $A_{i,j} = 1$ if French word i is aligned to English word j. (s,t) represents the sequence of French words $f_s \ldots f_t$. (s',t') represents the sequence of English words $e_{s'} \ldots f_{s'}$.)
For a given matrix A, define

$$A(i) = \{j : A_{i,j} = 1\}$$

Similarly, define

$$A'(j) = \{i : A_{i,j} = 1\}$$

Thus $A(i)$ is the set of English words that French word i is aligned to; $A'(j)$ is the set of French words that English word j is aligned to.
Then $\texttt{consistent}(A, (s,t), (s',t'))$ is true if and only if the following conditions are met:

1. For each $i \in \{s \ldots t\}$, $A(i) \subseteq \{s' \ldots t'\}$
2. For each $j \in \{s' \ldots t'\}$, $A'(j) \subseteq \{s \ldots t\}$
3. There is at least one (i, j) pair such that $i \in \{s \ldots t\}$, $j \in \{s' \ldots t'\}$, and $A_{i,j} = 1$

Figure 2: The definition of the `consistent` function.

a given input sentence; how each such translation receives a score under the model; and finally, how we can search for the highest scoring translation for an input sentence, thereby giving a translation algorithm.

3.1 Phrases, and Derivations

The input to a phrase-based translation model is a source-language sentence with n words, $x = x_1 \dots x_n$. The output is a sentence in the target language. The examples in this section will use German as the source language, and English as the target language. We will use the German sentence

wir müssen auch diese kritik ernst nehmen

as a running example.

A key component of a phrase-based translation model is a phrase-based lexicon, which pairs sequences of words in the source language with sequences of words in the target language, as described in the previous sections of this note. For example, lexical entries that are relevent to the German sentence shown above include

(wir müssen, we must)
(wir müssen auch, we must also)
(ernst, seriously)

and so on. Each phrase entry has an associated score, which can take any positive or negative value. As described before, a very simple way to estimate scores for phrases would be to define

$$g(f, e) = \log \frac{c(e, f)}{c(e)} \tag{1}$$

where f is a foreign sequence of words, e is an English sequence of words, and $c(e, f)$ and $c(e)$ are counts taken from some corpus. For example, we would have

$$g(\text{wir müssen, we must}) = \log \frac{c(\text{we must, wir müssen})}{c(\text{we must})}$$

The score for a phrase is then the log of the conditional probability of the foreign string, given the english string.

We introduce the following notation. For a particular input (source-language) sentence $x_1 \dots x_n$, a *phrase* is a tuple (s, t, e), signifying that the subsequence $x_s \dots x_t$ in the source language sentence can be translated as the target-language string e, using an entry from the phrase-based lexicon. For example, the phrase $(1, 2, \text{we must})$ would specify that the sub-string $x_1 \dots x_2$ can be translated as we must. Each phrase $p = (s, t, e)$ receives a score $g(p) \in R$ under the model. For a given phrase p, we will use $s(p)$, $t(p)$ and $e(p)$ to refer to its three components. We will use $\mathcal{P}$ to refer to the set of all possible phrases for the input sentence x.

Note that for a given input sentence $x_1 \dots x_n$, it is simple to compute the set of possible phrases, $\mathcal{P}$. We simply consider each substring of $x_1 \dots x_n$, and include all entries in the phrasal lexicon which

have this substring as their English string. We may end up with more than one phrasal entry for a particular source-language sub-string.

A *derivation* y is then a finite sequence of phrases, $p_1, p_2, \ldots p_L$, where each p_j for $j \in \{1 \ldots L\}$ is a member of $\mathcal{P}$. The length L can be any positive integer value. For any derivation y we use $e(y)$ to refer to the underlying translation defined by y, which is derived by concatenating the strings $e(p_1), e(p_2), \ldots e(p_L)$. For example, if

$$y = \text{(1, 3, we must also), (7, 7, take), (4, 5, this criticism), (6, 6, seriously)} \tag{2}$$

then

$$e(y) = \text{we must also take this criticism seriously}$$

3.2 The Set of Valid Derivations

We will use $\mathcal{Y}(x)$ to denote the set of valid derivations for an input sentence $x = x_1 x_2 \ldots x_n$. The set $\mathcal{Y}(x)$ is the set of finite length sequences of phrases $p_1 p_2 \ldots p_L$ which satisfy the following conditions:

- Each p_k for $k \in \{1 \ldots L\}$ is a member of the set of phrases $\mathcal{P}$ for $x_1 \ldots x_n$. (Recall that each p_k is a triple (s, t, e).)
- Each word is translated exactly once. More formally, if for a derivation $y = p_1 \ldots p_L$ we define

  $$y(i) = \sum_{k=1}^{L} [[s(p_k) \leq i \leq t(p_k)]] \tag{3}$$

  to be the number of times word i is translated (we define $[[\pi]]$ to be 1 if π is true, 0 otherwise), then we must have

  $$y(i) = 1$$

  for $i = 1 \ldots n$.
- For all $k \in \{1 \ldots L-1\}$,

  $$|t(p_k) + 1 - s(p_{k+1})| \leq d$$

  where $d \geq 0$ is a parameter of the model. In addition, we must have

  $$|1 - s(p_1)| \leq d$$

The first two conditions should be clear. The last condition, which depends on the parameter d, deserves more explanation.

The parameter d is a limit on how far consecutive phrases can be from each other, and is often referred to as a *distortion limit*. To illustrate this, consider our previous example derivation:

$$y = \text{(1, 3, we must also), (7, 7, take), (4, 5, this criticism), (6, 6, seriously)}$$

In this case $y = p_1 p_2 p_3 p_4$ (i.e., the number of phrases, L, is equal to 4). For the sake of argument, assume that the distortion parameter d, is equal to 4.

We will now address the following question: does this derivation satisfy the condition

$$|t(p_k) + 1 - s(p_{k+1})| \leq d \tag{4}$$

for $k = 1 \ldots 3$? Consider first the case $k = 1$. In this case we have $t(p_1) = 3$, and $s(p_2) = 7$. Hence

$$|t(p_1) + 1 - s(p_2)| = |3 + 1 - 7| = 3$$

and the constraint in Eq. 4 is satisfied for $k = 1$. It can be seen that the value of $|t(p_1) + 1 - s(p_2)|$ is a measure of how far the phrases p_1 and p_2 are from each other in the sentence. The distortion limit specifies that consecutive phrases must be relatively close to each other.

Now consider the constraint for $k = 2$. In this case we have

$$|t(p_2) + 1 - s(p_3)| = |7 + 1 - 4| = 4$$

so the constraint is satisfied (recall that we have assume that $d = 4$). For $k = 3$ we have

$$|t(p_3) + 1 - s(p_4)| = |5 + 1 - 6| = 0$$

Finally, we need to check the constraint

$$|1 - s(p_1)| \leq d$$

For this example, $s(p_1) = 1$, and the constraint is satisfied. This final constraint ensures that the first phrase in the sequence, p_1, is not too far from the start of the sentence.

As an example of a derivation that is invalid, because it does not satisfy the distortion constraints, consider

$$y = \text{(1, 2, we must), (7, 7, take), (3, 3, also), (4, 5, this criticism), (6, 6, seriously)}$$

In this case it can be verified that

$$|t(p_2) + 1 - s(p_3)| = |7 + 1 - 3| = 5$$

which is greater than the distortion limit, d, which is equal to 4.

The motivation for the distortion limit is two-fold:

1. It reduces the search space in the model, making translation with the model more efficient.
2. Empirically, it is often shown to improve translation performance. For many language pairs, it is preferable to disallow consecutive phrases which are a long distance from each other, as this will lead to poor translations.

However, it should be noted that the distortion limit is really a rather crude method for modeling word order differences between languages. Later in the class we will see systems that attempt to improve upon this method.

3.3 Scoring Derivations

The next question is the following: how do we score derivations? That is, how do we define the function $f(y)$ which assigns a score to each possible derivation for a sentence? The optimal translation under the model for a source-language sentence x will be

$$\arg \max_{y \in \mathcal{Y}(x)} f(y)$$

In phrase-based systems, the score for any derivation y is calculated as follows:

$$f(y) = h(e(y)) + \sum_{k=1}^{L} g(p_k) + \sum_{k=1}^{L-1} \eta \times |t(p_k) + 1 - s(p_{k+1})| \tag{5}$$

The components of this score are as follows:

- As defined before, $e(y)$ is the target-language string for derivation y. $h(e(y))$ is the log-probability for the string $e(y)$ under a trigram language model. Hence if $e(y) = e_1 e_2 \dots e_m$, then
$$h(e(y)) = \log \prod_{i=1}^{m} q(e_i | e_{i-2}, e_{i-1}) = \sum_{i=1}^{m} \log q(e_i | e_{i-2}, e_{i-1})$$
where $q(e_i | e_{i-2}, e_{i-1})$ is the probability of word e_i following the bigram e_{i-2}, e_{i-1} under a trigram language model.
- As defined before, $g(p_k)$ is the score for the phrase p_k (see for example Eq. 1 for one possible way of defining $g(p)$).
- η is a "distortion parameter" of the model. It can in general be any positive or negative value, although in practice it is almost always negative. Each term of the form
$$\eta \times |t(p_k) + 1 - s(p_{k+1})|$$
then corresponds to a penalty (assuming that η is negative) on how far phrases p_k and p_{k+1} are from each other. Thus in addition to having hard constraints on the distance between consecutive phrases, we also have a soft constraint (i.e., a penalty that increases linearly with this distance).

Given these definitions, the optimal translation in the model for a source-language sentence $x = x_1 \dots x_n$ is

$$\arg \max_{y \in \mathcal{Y}(x)} f(y)$$

3.4 Summary: Putting it all Together

Definition 2 (Phrase-based translation models) *A phrase-based translation model is a tuple* $(\mathcal{L}, h, d, \eta)$*, where:*

- *$\mathcal{L}$ is a phrase-based lexicon. Each member of $\mathcal{L}$ is a tuple (f, e, g) where f is a sequence of one or more foreign-language words, e is a sequence of one or more English words, and $g \in R$ is a score for the pair (f, e).*
- *h is a trigram language model: that is, for any English string $e_1 \dots e_m$,*

$$h(e_1 \dots e_m) = \sum_{i=1}^{m} \log q(e_i | e_{i-2}, e_{i-1})$$

  where q are the parameters of the model, and we assume that $e_{-1} = e_0 = *$, where $*$ is a special start symbol in the language model.
- *d is a non-negative integer, specifying the distortion limit under the model.*
- *$\eta \in R$ is the distortion penalty in the model.*

For an input sentence $x_1 \dots x_n$, define $\mathcal{Y}(x)$ to be the set of valid derivations under the model $(\mathcal{L}, h, d, \eta)$. The decoding problem is to find

$$\arg \max_{y \in \mathcal{Y}(x)} f(y)$$

where, assuming $y = p_1 p_2 \dots p_L$,

$$f(y) = h(e(y)) + \sum_{k=1}^{L} g(p_k) + \sum_{k=1}^{L-1} \eta \times |t(p_k) + 1 - s(p_{k+1})|$$

4 Decoding with Phrase-based Models

We now describe a decoding algorithm for phrase-based models: that is, an algorithm that attempts to find

$$\arg \max_{y \in \mathcal{Y}(x)} f(y) \tag{6}$$

where, assuming $y = p_1 p_2 \dots p_L$,

$$f(y) = h(e(y)) + \sum_{k=1}^{L} g(p_k) + \sum_{k=1}^{L-1} \eta \times |t(p_k) + 1 - s(p_{k+1})|$$

The problem in Eq. 6 is in fact NP-hard for this definition of $f(y)$; hence the algorithm we describe is an approximate method, which is not guaranteed to find the optimal solution.

A first critical data structure in the algorithm is a *state*. A state is a tuple

$$(e_1, e_2, b, r, \alpha)$$

where e_1, e_2 are English words, b is a bit-string of length n (recall that n is the length of the source-language sentence), r is an integer specifying the end-point of the last phrase in the state, and α is the score for the state.

Any sequence of phrases can be mapped to a corresponding state. For example, the sequence

$$y = \text{(1, 3, we must also), (7, 7, take), (4, 5, this criticism)}$$

would be mapped to the state

$$(\text{this}, \text{criticism}, 1111101, 5, \alpha)$$

The state records the last two words in the translation underlying this sequence of phrases, namely *this* and *criticism*. The bit-string records which words have been translated: the i'th bit in the bit-string is equal to 1 if the i'th word has been translated, 0 otherwise. In this case, only the 6'th bit is 0, as only the 6'th word has not been translated. The value $r = 5$ indicates that the final phrase in the sequence, $(4, 5, \text{this criticism})$ ends at position 5. Finally, α will be the score of the partial translation, calculated as

$$\alpha = h(e(y)) + \sum_{k=1}^{L} g(p_k) + \sum_{k=1}^{L-1} \eta \times |t(p_k) + 1 - s(p_{k+1})|$$

where $L = 3$, we have

$$e(y) = \text{we must also take this criticism}$$

and

$$p_1 = \text{(1, 3, we must also)}, \quad p_2 = \text{(7, 7, take)}, \quad p_3 = \text{(4, 5, this criticism)}$$

Note that the state only records the last two words in a derivation: as will see shortly, this is because a trigram language model is only sensitive to the last two words in the sequence, so the state only needs to record these last two words.

We define the initial state as

$$q_0 = (*, *, 0^n, 0, 0)$$

where 0^n is bit-string of length n, with n zeroes. We have used * to refer to the special "start" symbol in the language model. The initial state has no words translated (all bits set to 0); the value for r is 0; and the score α is 0.

Next we define a function $\text{ph}(q)$ that maps a state q to the set of phrases which can be appended to q. For a phrase p to be a member of $\text{ph}(q)$, where $q = (e_1, e_2, b, r, \alpha)$, the following conditions must be satisfied:

- p must not overlap with the bit-string b. I.e., we must have $b_i = 0$ for $i \in \{s(p) \dots t(p)\}$.
- The distortion limit must not be violated. More specifically, we must have

  $$|r + 1 - s(p)| \leq d$$

  where d is the distortion limit.

In addition, for any state q, for any phrase $p \in \text{ph}(q)$, we define

$$\text{next}(q, p)$$

to be the state formed by combining state q with phrase p. Formally, if $q = (e_1, e_2, b, r, \alpha)$, and $p = (s, t, \epsilon_1 \dots \epsilon_M)$, then $\text{next}(q, p)$ is the state $q' = (e'_1, e'_2, b', r', \alpha')$ defined as follows:

- First, for convenience, define $\epsilon_{-1} = e_1$, and $\epsilon_0 = e_2$.
- Define $e'_1 = \epsilon_{M-1}$, $e'_2 = \epsilon_M$.
- Define $b'_i = 1$ for $i \in \{s \dots t\}$. Define $b'_i = b_i$ for $i \notin \{s \dots t\}$
- Define $r' = t$
- Define

$$\alpha' = \alpha + g(p) + \sum_{i=1}^{M} \log q(\epsilon_i | \epsilon_{i-2}, \epsilon_{i-1}) + \eta \times |r + 1 - s|$$

Hence e'_1 and e'_2 are updated to record the last two words in the translation formed by appending phrase p to state q; b' is an updated bit-string, which is modified to record the fact that words $s \dots t$ are now translated; r' is simply set to t, i.e., the end point of the phrase p; and α' is calculated by adding the phrase score $g(p)$, the language model scores for the words $\epsilon_1 \dots \epsilon_M$, and the distortion term $\eta \times |r + 1 - s|$.

The final function we need for the decoding algorithm is a very simple function, that tests for equality of two states. This is the function

$$\text{eq}(q, q')$$

which returns true or false. Assuming $q = (e_1, e_2, b, r, \alpha)$, and $q' = (e'_1, e'_2, b', r', \alpha')$, $eq(q, q')$ is true if and only if $e_1 = e'_1$, $e_2 = e'_2$, $b = b'$ and $r = r'$.

Having defined the functions ph, next, and eq, we are now ready to give the full decoding algorithm. Figure 3 gives the basic decoding algorithm. The algorithm manipulates sets Q_i for $i = 0 \dots n$. Each set Q_i contains a set of states corresponding to translations of length i (the length for a state q is simply the number of bits in the bit-string for q whose value is 1—that is, the number of foreign words that are translated in the state). Initially, we set Q_0 to contain a single state, the initial state q_0. We set all other sets Q_i for $i = 1 \dots n$ to be the empty set. We then iterate: for each $i \in \{1, 2, \dots n\}$, we consider each $q \in \text{beam}(Q_i)$ ($\text{beam}(Q_i)$ is a subset of Q_i, containing only the highest scoring elements of Q_i: we will give a formal definition shortly). For each $q \in \text{beam}(Q_i)$ we first calculate the set $\text{ph}(q)$; then for each $p \in \text{ph}(q)$ we calculate the next state $q' = \text{next}(q, p)$. We add this new state to the set Q_j where j is the length of state q'. Note that we always have $j > i$, so we are always adding elements to states that are further advanced than the set Q_i that we are currently considering.

Figure 4 gives a definition of the function $\text{Add}(Q, q', q, p)$. The function first checks whether there is an existing state q'' in Q such that $eq(q'', q')$ is true. If this is the case, then q' replaces q'' if its score is higher than q''; otherwise q' is not added to Q. Hence only one of the states q'' and q' remains in Q. If there is no such state q'', then q' is simply added to Q.

Note that the Add function records backpointers $bp(q')$ for any state q' added to a set Q. These backpointers will allow us to recover the final translation for the highest scoring state. In fact, the final step of the algorithm it to: 1) find the highest scoring state q in the set Q_n; 2) from this state recover the highest scoring translation, by tracing the backpointers for the states.

Finally, we need to define the function $\text{beam}(Q)$; this definition is given in Figure 5. This function first computes α^*, the highest score for any state in Q; it then discards any state with a score that is less than $\alpha^* - \beta$, where $\beta > 0$ is the *beam-width* of the decoding algorithm.

- Inputs: sentence $x_1 \ldots x_n$. Phrase-based model $(\mathcal{L}, h, d, \eta)$. The phrase-based model defines the functions $\text{ph}(q)$ and $\text{next}(q, p)$.
- Initialization: set $Q_0 = \{q_0\}$, $Q_i = \emptyset$ for $i = 1 \ldots n$.
- For $i = 0 \ldots n-1$
  - For each state $q \in \text{beam}(Q_i)$, for each phrase $p \in \text{ph}(q)$:
    (1) $q' = \text{next}(q, p)$
    (2) $\text{Add}(Q_j, q', q, p)$ where $j = \text{len}(q')$
- Return: highest scoring state in Q_n. Backpointers can be used to find the underlying sequence of phrases (and the translation).

Figure 3: The basic decoding algorithm. $\text{len}(q')$ is the number bits equal to 1 in the bit-string for q' (i.e., the number of foreign words translated in the state q').

$\text{Add}(Q, q', q, p)$

- If there is some $q'' \in Q$ such that $eq(q'', q') = \text{True}$:
  - If $\alpha(q') > \alpha(q'')$
    - $Q = \{q'\} \cup Q \setminus \{q''\}$
    - set $bp(q') = (q, p)$
  - Else return
- Else
  - $Q = Q \cup \{q'\}$
  - set $bp(q') = (q, p)$

Figure 4: The $\text{Add}(Q, q', q, p)$ function.

Definition of $\text{beam}(Q)$: define

$$\alpha^* = \arg\max_{q \in Q} \alpha(q)$$

i.e., α^* is the highest score for any state in Q.
Define $\beta \geq 0$ to be the *beam-width* parameter
Then

$$\text{beam}(Q) = \{q \in Q : \alpha(q) \geq \alpha^* - \beta\}$$

Figure 5: Definition of the beam function in the algorithm in figure 3.

Log-Linear Models

Michael Collins

1 Introduction

This note describes *log-linear models*, which are very widely used in natural language processing. A key advantage of log-linear models is their flexibility: as we will see, they allow a very rich set of features to be used in a model, arguably much richer representations than the simple estimation techniques we have seen earlier in the course (e.g., the smoothing methods that we initially introduced for language modeling, and which were later applied to other models such as HMMs for tagging, and PCFGs for parsing). In this note we will give motivation for log-linear models, give basic definitions, and describe how parameters can be estimated in these models. In subsequent classes we will see how these models can be applied to a number of natural language processing problems.

2 Motivation

As a motivating example, consider again the language modeling problem, where the task is to derive an estimate of the conditional probability

$$P(W_i = w_i | W_1 = w_1 \dots W_{i-1} = w_{i-1}) = p(w_i | w_1 \dots w_{i-1})$$

for any sequence of words $w_1 \dots w_i$, where i can be any positive integer. Here w_i is the i'th word in a document: our task is to model the distribution over the word w_i, conditioned on the previous sequence of words $w_1 \dots w_{i-1}$.

In trigram language models, we assumed that

$$p(w_i | w_1 \dots w_{i-1}) = q(w_i | w_{i-2}, w_{i-1})$$

where $q(w|u, v)$ for any trigram (u, v, w) is a parameter of the model. We studied a variety of ways of estimating the q parameters; as one example, we studied linear interpolation, where

$$q(w|u, v) = \lambda_1 q_{ML}(w|u, v) + \lambda_2 q_{ML}(w|v) + \lambda_3 q_{ML}(w) \quad (1)$$

Here each q_{ML} is a maximum-likelihood estimate, and $\lambda_1, \lambda_2, \lambda_3$ are parameters dictating the weight assigned to each estimate (recall that we had the constraints that $\lambda_1 + \lambda_2 + \lambda_3 = 1$, and $\lambda_i \geq 0$ for all i).

Trigram language models are quite effective, but they make relatively narrow use of the context $w_1 \ldots w_{i-1}$. Consider, for example, the case where the context $w_1 \ldots w_{i-1}$ is the following sequence of words:

> Third, the notion "grammatical in English" cannot be identified in any way with the notion "high order of statistical approximation to English". It is fair to assume that neither sentence (1) nor (2) (nor indeed any part of these sentences) has ever occurred in an English discourse. Hence, in any statistical

Assume in addition that we'd like to estimate the probability of the word *model* appearing as word w_i, i.e., we'd like to estimate

$$P(W_i = \text{model} | W_1 = w_1 \ldots W_{i-1} = w_{i-1})$$

In addition to the previous two words in the document (as used in trigram language models), we could imagine conditioning on all kinds of features of the context, which might be useful evidence in estimating the probability of seeing *model* as the next word. For example, we might consider the probability of *model* conditioned on word w_{i-2}, ignoring w_{i-1} completely:

$$P(W_i = \text{model} | W_{i-2} = \text{any})$$

We might condition on the fact that the previous word is an adjective

$$P(W_i = \text{model} | \text{pos}(W_{i-1}) = \text{adjective})$$

here pos is a function that maps a word to its part of speech. (For simplicity we assume that this is a deterministic function, i.e., the mapping from a word to its underlying part-of-speech is unambiguous.) We might condition on the fact that the previous word's suffix is "ical":

$$P(W_i = \text{model} | \text{suff4}(W_{i-1}) = \text{ical})$$

(here suff4 is a function that maps a word to its last four characters). We might condition on the fact that the word *model* does not appear in the context:

$$P(W_i = \text{model} | W_j \neq \text{model} \ \text{ for } j \in \{1 \ldots (i-1)\})$$

or we might condition on the fact that the word *grammatical* does appear in the context:

$$P(W_i = \text{model} | W_j = \text{grammatical} \text{ for some } j \in \{1 \dots (i-1)\})$$

In short, all kinds of information in the context might be useful in estimating the probability of a particular word (e.g., *model*) in that context.

A naive way to use this information would be to simply extend the methods that we saw for trigram language models. Rather than combining three estimates, based on trigram, bigram, and unigram estimates, we would combine a much larger set of estimates. We would again estimate λ parameters reflecting the importance or weight of each estimate. The resulting estimator would take something like the following form (this is intended as a sketch only):

$$\begin{aligned}
&p(\text{model} | w_1, \dots w_{i-1}) = \\
&\lambda_1 \times q_{ML}(\text{model} | w_{i-2} = \text{any}, w_{i-1} = \text{statistical}) + \\
&\lambda_2 \times q_{ML}(\text{model} | w_{i-1} = \text{statistical}) + \\
&\lambda_3 \times q_{ML}(\text{model}) + \\
&\lambda_4 \times q_{ML}(\text{model} | w_{i-2} = \text{any}) + \\
&\lambda_5 \times q_{ML}(\text{model} | w_{i-1} \text{ is an adjective}) + \\
&\lambda_6 \times q_{ML}(\text{model} | w_{i-1} \text{ ends in ``ical''}) + \\
&\lambda_7 \times q_{ML}(\text{model} | \text{``model'' does not occur somewhere in } w_1, \dots w_{i-1}) + \\
&\lambda_8 \times q_{ML}(\text{model} | \text{``grammatical'' occurs somewhere in } w_1, \dots w_{i-1}) + \\
&\dots
\end{aligned}$$

The problem is that the linear interpolation approach becomes extremely unwieldy as we add more and more pieces of conditioning information. In practice, it is very difficult to extend this approach beyond the case where we small number of estimates that fall into a natural hierarchy (e.g., unigram, bigram, trigram estimates). In contrast, we will see that log-linear models offer a much more satisfactory method for incorporating multiple pieces of contextual information.

3 A Second Example: Part-of-speech Tagging

Our second example concerns part-of-speech tagging. Consider the problem where the context is a sequence of words $w_1 \dots w_n$, together with a sequence of tags, $t_1 \dots t_{i-1}$ (here $i < n$), and our task is to model the conditional distribution over the i'th tag in the sequence. That is, we wish to model the conditional distribution

$$P(T_i = t_i | T_1 = t_1 \dots T_{i-1} = t_{i-1}, W_1 = w_1 \dots W_n = w_n)$$

As an example, we might have the following context:

> Hispaniola/NNP quickly/RB became/VB an/DT important/JJ base from which Spain expanded its empire into the rest of the Western Hemisphere .

Here $w_1 \ldots w_n$ is the sentence *Hispaniola quickly ... Hemisphere* ., and the previous sequence of tags is $t_1 \ldots t_5$ = `NNP RB VB DT JJ`. We have $i = 6$, and our task is to model the distribution

$$P(T_6 = t_6 \quad | \quad W_1 \ldots W_n = \textit{Hispaniola quickly} \ldots \textit{Hemisphere} ., \\ T_1 \ldots T_5 = \texttt{NNP RB VB DT JJ})$$

i.e., our task is to model the distribution over tags for the 6th word, *base*, in the sentence.

In this case there are again many pieces of contextual information that might be useful in estimating the distribution over values for t_i. To be concrete, consider estimating the probability that the tag for *base* is `V` (i.e., $T_6 = \texttt{V}$). We might consider the probability conditioned on the identity of the i'th word:

$$P(T_6 = \texttt{V} | W_6 = \textit{base})$$

and we might also consider the probability conditioned on the previous one or two tags:

$$P(T_6 = \texttt{V} | T_5 = \texttt{JJ})$$

$$P(T_6 = \texttt{V} | T_4 = \texttt{DT}, T_5 = \texttt{JJ})$$

We might consider the probability conditioned on the *previous* word in the sentence

$$P(T_6 = \texttt{V} | W_5 = \textit{important})$$

or the probability conditioned on the *next* word in the sentence

$$P(T_6 = \texttt{V} | W_7 = \textit{from})$$

We might also consider the probability based on spelling features of the word w_6, for example the last two letters of w_6:

$$P(T_6 = \texttt{V} | \text{suff2}(W_6) = \textit{se})$$

(here suff2 is a function that maps a word to its last two letters).

In short, we again have a scenario where a whole variety of contextual features might be useful in modeling the distribution over the random variable of interest (in this case the identity of the i'th tag). Again, a naive approach based on an extension of linear interpolation would unfortunately fail badly when faced with this estimation problem.

4 Log-Linear Models

We now describe how log-linear models can be applied to problems of the above form.

4.1 Basic Definitions

The abstract problem is as follows. We have some set of possible *inputs*, $\mathcal{X}$, and a set of possible *labels*, $\mathcal{Y}$. Our task is to model the conditional probability

$$p(y|x)$$

for any pair (x, y) such that $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.

For example, in the language modeling task we have some finite set of possible words in the language, call this set $\mathcal{V}$. The set $\mathcal{Y}$ is simply equal to $\mathcal{V}$. The set $\mathcal{X}$ is the set of possible sequences $w_1 \dots w_{i-1}$ such that $i \geq 1$, and $w_j \in \mathcal{V}$ for $j \in \{1 \dots (i-1)\}$.

In the part-of-speech tagging example, we have some set $\mathcal{V}$ of possible words, and a set $\mathcal{T}$ of possible tags. The set $\mathcal{Y}$ is simply equal to $\mathcal{T}$. The set $\mathcal{X}$ is the set of contexts of the form

$$\langle w_1 w_2 \dots w_n, t_1 t_2 \dots t_{i-1} \rangle$$

where $n \geq 1$ is an integer specifying the length of the input sentence, $w_j \in \mathcal{V}$ for $j \in \{1 \dots n\}$, $i \in \{1 \dots (n-1)\}$, and $t_j \in \mathcal{T}$ for $j \in \{1 \dots (i-1)\}$.

We will assume throughout that $\mathcal{Y}$ is a finite set. The set $\mathcal{X}$ could be finite, countably infinite, or even uncountably infinite.

Log-linear models are then defined as follows:

Definition 1 (Log-linear Models) *A log-linear model consists of the following components:*

- *A set $\mathcal{X}$ of possible inputs.*
- *A set $\mathcal{Y}$ of possible labels. The set $\mathcal{Y}$ is assumed to be finite.*
- *A positive integer d specifying the number of features and parameters in the model.*
- *A function $f : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^d$ that maps any (x, y) pair to a* feature-vector *$f(x, y)$.*
- *A parameter vector $v \in \mathbb{R}^d$.*

For any $x \in \mathcal{X}$, $y \in \mathcal{Y}$, the model defines a condtional probability

$$p(y|x; v) = \frac{\exp(v \cdot f(x, y))}{\sum_{y' \in \mathcal{Y}} \exp(v \cdot f(x, y'))}$$

Here $\exp(x) = e^x$, and $v \cdot f(x, y) = \sum_{k=1}^{d} v_k f_k(x, y)$ is the inner product between v and $f(x, y)$. The term $p(y|x; v)$ is intended to be read as "the probability of y conditioned on x, under parameter values v". □

We now describe the components of the model in more detail, first focusing on the feature-vector definitions $f(x, y)$, then giving intuition behind the model form

$$p(y|x; v) = \frac{\exp(v \cdot f(x, y))}{\sum_{y' \in \mathcal{Y}} \exp(v \cdot f(x, y'))}$$

5 Features

As described in the previous section, for any pair (x, y), $f(x, y) \in \mathbb{R}^d$ is a feature vector representing that pair. Each component $f_k(x, y)$ for $k = 1 \ldots d$ in this vector is referred to as a *feature*. The features allows us to represent different properties of the input x, in conjunction with the label y. Each feature has an associated parameter, v_k, whose value is estimated using a set of training examples. The training set consists of a sequence of examples $(x^{(i)}, y^{(i)})$ for $i = 1 \ldots n$, where each $x^{(i)} \in \mathcal{X}$, and each $y^{(i)} \in \mathcal{Y}$.

In this section we first give an example of how features can be constructed for the language modeling problem, as introduced earlier in this note; we then describe some practical issues in defining features.

5.1 Features for the Language Modeling Example

Consider again the language modeling problem, where the input x is a sequence of words $w_1 w_2 \ldots w_{i-1}$, and the label y is a word. Figure 1 shows a set of example features for this problem. Each feature is an indicator function: that is, each feature is a function that returns either 1 or 0. It is extremely common in NLP applications to have indicator functions as features. Each feature returns the value of 1 if some property of the input x conjoined with the label y is true, and 0 otherwise.

The first three features, f_1, f_2, and f_3, are analogous to unigram, bigram, and trigram features in a regular trigram language model. The first feature returns 1 if the label y is equal to the word *model*, and 0 otherwise. The second feature returns 1 if the bigram $\langle w_{i-1}\ y \rangle$ is equal to $\langle$*statistical model*$\rangle$, and 0 otherwise. The third feature returns 1 if the trigram $\langle w_{i-2}\ w_{i-1}\ y \rangle$ is equal to $\langle$*any statistical model*$\rangle$,

$$
\begin{aligned}
f_1(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model} \\ 0 & \text{otherwise} \end{cases} \\
f_2(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model} \text{ and } w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases} \\
f_3(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-2} = \texttt{any}, w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases} \\
f_4(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-2} = \texttt{any} \\ 0 & \text{otherwise} \end{cases} \\
f_5(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-1} \text{ is an adjective} \\ 0 & \text{otherwise} \end{cases} \\
f_6(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-1} \text{ ends in ``ical''} \\ 0 & \text{otherwise} \end{cases} \\
f_7(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, \text{``model'' is not in } w_1, \ldots w_{i-1} \\ 0 & \text{otherwise} \end{cases} \\
f_8(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, \text{``grammatical'' is in } w_1, \ldots w_{i-1} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 1: Example features for the language modeling problem, where the input x is a sequence of words $w_1 w_2 \ldots w_{i-1}$, and the label y is a word.

and 0 otherwise. Recall that each of these features will have a parameter, v_1, v_2, or v_3; these parameters will play a similar role to the parameters in a regular trigram language model.

The features $f_4 \ldots f_8$ in figure 1 consider properties that go beyond unigram, bigram, and trigram features. The feature f_4 considers word w_{i-2} in conjunction with the label y, ignoring the word w_{i-1}; this type of feature is often referred to as a "skip bigram". Feature f_5 considers the part-of-speech of the previous word (assume again that the part-of-speech for the previous word is available, for example through a deterministic mapping from words to their part-of-speech, or perhaps through a POS tagger's output on words $w_1 \ldots w_{i-1}$). Feature f_6 considers the suffix of the previous word, and features f_7 and f_8 consider various other features of the input $x = w_1 \ldots w_{i-1}$.

From this example we can see that it is possible to incorporate a broad set of contextual information into the language modeling problem, using features which are indicator functions.

5.2 Feature Templates

We now discuss some practical issues in defining features. In practice, a key idea in defining features is that of *feature templates*. We introduce this idea in this section.

Recall that our first three features in the previous example were as follows:

$$f_1(x, y) = \begin{cases} 1 & \text{if } y = \texttt{model} \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(x, y) = \begin{cases} 1 & \text{if } y = \texttt{model} \text{ and } w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases}$$

$$f_3(x, y) = \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-2} = \texttt{any}, w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases}$$

These features track the unigram $\langle$*model*$\rangle$, the bigram $\langle$*statistical model*$\rangle$, and the trigram $\langle$*any statistical model*$\rangle$.

Each of these features is specific to a particular unigram, bigram or trigram. In practice, we would like to define a much larger class of features, which consider all possible unigrams, bigrams or trigrams seen in the training data. To do this, we use *feature templates* to generate large sets of features.

As one example, here is a feature template for trigrams:

Definition 2 (Trigram feature template) *For any trigram* (u, v, w) *seen in train-*

ing data, create a feature

$$f_{N(u,v,w)}(x,y) = \begin{cases} 1 & \textit{if } y = w, \, w_{i-2} = u, \, w_{i-1} = v \\ 0 & \textit{otherwise} \end{cases}$$

where $N(u,v,w)$ is a function that maps each trigram in the training data to a unique integer.

A couple of notes on this definition:

- Note that the template only generates trigram features for those trigrams seen in training data. There are two reasons for this restriction. First, it is not feasible to generate a feature for every possible trigram, even those not seen in training data: this would lead to V^3 features, where V is the number of words in the vocabulary, which is a very large set of features. Second, for any trigram (u,v,w) not seen in training data, we do not have evidence to estimate the associated parameter value, so there is no point including it in any case.[1]

- The function $N(u,v,w)$ maps each trigram to a unique integer: that is, it is a function such that for any trigrams (u,v,w) and (u',v',w') such that $u \neq u'$, $v \neq v'$, or $w \neq w'$, we have

  $$N(u,v,w) \neq N(u',v',w')$$

  In practice, in implementations of feature templates, the function N is implemented through a hash function. For example, we could use a hash table to hash strings such as `trigram=any_statistical_model` to integers. Each distinct string is hashed to a different integer.

Continuing with the example, we can also define bigram and unigram feature templates:

Definition 3 (Bigram feature template) *For any bigram (v,w) seen in training data, create a feature*

$$f_{N(v,w)}(x,y) = \begin{cases} 1 & \textit{if } y = w, \, w_{i-1} = v \\ 0 & \textit{otherwise} \end{cases}$$

where $N(v,w)$ maps each bigram to a unique integer.

[1] This isn't quite accurate: there may in fact be reasons for including features for trigrams (u,v,w) where the bigram (u,v) *is* observed in the training data, but the trigram (u,v,w) is *not* observed in the training data. We defer discussion of this until later.

Definition 4 (Unigram feature template) *For any unigram (w) seen in training data, create a feature*

$$f_{N(w)}(x, y) = \begin{cases} 1 & \text{if } y = w \\ 0 & \text{otherwise} \end{cases}$$

where $N(w)$ maps each unigram to a unique integer.

We actually need to be slightly more careful with these definitions, to avoid overlap between trigram, bigram, and unigram features. Define T, B and U to be the set of trigrams, bigrams, and unigrams seen in the training data. Define

$$N_t = \{i : \exists (u, v, w) \in T \text{ such that } N(u, v, w) = i\}$$

$$N_b = \{i : \exists (v, w) \in B \text{ such that } N(v, w) = i\}$$

$$N_u = \{i : \exists (w) \in U \text{ such that } N(w) = i\}$$

Then we need to make sure that there is no overlap between these sets—otherwise, two different n-grams would be mapped to the same feature. More formally, we need

$$N_t \cap N_b = N_t \cap N_u = N_b \cap N_u = \emptyset \tag{2}$$

In practice, it is easy to ensure this when implementing log-linear models, using a single hash table to hash strings such as `trigram=any_statistical_model`, `bigram=statistical_model`, `unigram=model`, to distinct integers.

We could of course define additional templates. For example, the following is a template which tracks the length-4 suffix of the previous word, in conjunction with the label y:

Definition 5 (Length-4 Suffix Template) *For any pair (v, w) seen in training data, where $v =$ suff4(w_{i-1}), and $w = y$, create a feature*

$$f_{N(\textit{suff4}=v,w)}(x, y) = \begin{cases} 1 & \text{if } y = w \text{ and } \textit{suff4}(x) = v \\ 0 & \text{otherwise} \end{cases}$$

where $N(\textit{suff4} = v, w)$ maps each pair (v, w) to a unique integer, with no overlap with the other feature templates used in the model (where overlap is defined analogously to Eq. 2 above).

5.3 Feature Sparsity

A very important property of the features we have defined above is feature sparsity. The number of features, d, in many NLP applications can be extremely large. For example, with just the trigram template defined above, we would have one feature for each trigram seen in training data. It is not untypical to see models with 100s of thousands or even millions of features.

This raises obvious concerns with efficiency of the resulting models. However, we describe in this section how feature sparsity can lead to efficient models.

The key observation is the following: for any given pair (x, y), the number of values for k in $\{1 \dots d\}$ such that

$$f_k(x, y) = 1$$

is often very small, and is typically much smaller than the total number of features, d. Thus all but a very small subset of the features are 0: the feature vector $f(x, y)$ is a very sparse bit-string, where almost all features $f_k(x, y)$ are equal to 0, and only a few features are equal to 1.

As one example, consider the language modeling example where we use only the trigram, bigram and unigram templates, as described above. The number of features in this model is large (it is equal to the number of distinct trigrams, bigrams and unigrams seen in training data). However, it can be seen immediately that for any pair (x, y), at most three features are non-zero (in the worst case, the pair (x, y) contains trigram, bigram and unigram features which are all seen in the training data, giving three non-zero features in total).

When implementing log-linear models, models with sparse features can be quite efficient, because there is no need to explicitly represent and manipulate d-dimensional feature vectors $f(x, y)$. Instead, it is generally much more efficient to implement a function (typically through hash tables) that for any pair (x, y) computes the indices of the non-zero features: i.e., a function that computes the set

$$Z(x, y) = \{k : f_k(x, y) = 1\}$$

This set is small in sparse feature spaces—for example with unigram/bigram/trigram features alone, it would be of size at most 3. In general, it is straightforward to implement a function that computes $Z(x, y)$ in $O(|Z(x, y)|)$ time, using hash functions. Note that $|Z(x, y)| \ll d$, so this is much more efficient than explicitly computing all d features, which would take $O(d)$ time.

As one example of how efficient computation of $Z(x, y)$ can be very helpful, consider computation of the inner product

$$v \cdot f(x, y) = \sum_{k=1}^{d} v_k f_k(x, y)$$

This computation is central in log-linear models. A naive method would iterate over each of the d features in turn, and would take $O(d)$ time. In contrast, if we make use of the identity

$$\sum_{k=1}^{d} v_k f_k(x, y) = \sum_{k \in Z(x,y)} v_k$$

hence looking at only non-zero features, we can compute the inner product in $O(|Z(x, y)|)$ time.

6 The Model form for Log-Linear Models

We now describe the model form for log-linear models in more detail. Recall that for any pair (x, y) such that $x \in \mathcal{X}$, and $y \in \mathcal{Y}$, the conditional probability under the model is

$$p(y \mid x; v) = \frac{\exp(v \cdot f(x, y))}{\sum_{y' \in \mathcal{Y}} \exp(v \cdot f(x, y'))}$$

The inner products

$$v \cdot f(x, y)$$

play a key role in this expression. Again, for illustration consider our languge-modeling example where the input $x = w_1 \dots w_{i-1}$ is the following sequence of words:

> Third, the notion "grammatical in English" cannot be identified in any way with the notion "high order of statistical approximation to English". It is fair to assume that neither sentence (1) nor (2) (nor indeed any part of these sentences) has ever occurred in an English discourse. Hence, in any statistical

The first step in calculating the probability distribution over the next word in the document, conditioned on x, is to calculate the inner product $v \cdot f(x, y)$ for each possible label y (i.e., for each possible word in the vocabulary). We might, for example, find the following values (we show the values for just a few possible words—in reality we would compute an inner product for each possible word):

$$\begin{aligned} v \cdot f(x, model) &= 5.6 & v \cdot f(x, the) &= -3.2 \\ v \cdot f(x, is) &= 1.5 & v \cdot f(x, of) &= 1.3 \\ v \cdot f(x, models) &= 4.5 & \dots \end{aligned}$$

Note that the inner products can take any value in the reals, positive or negative. Intuitively, if the inner product $v \cdot f(x, y)$ for a given word y is high, this indicates that the word has high probability given the context x. Conversely, if $v \cdot f(x, y)$ is low, it indicates that y has low probability in this context.

The inner products $v \cdot f(x, y)$ can take any value in the reals; our goal, however, is to define a conditional distribution $p(y|x)$. If we take

$$\exp\left(v \cdot f(x, y)\right)$$

for any label y, we now have a value that is greater than 0. If $v \cdot f(x, y)$ is high, this value will be high; if $v \cdot f(x, y)$ is low, for example if it is strongly negative, this value will be low (close to zero).

Next, if we divide the above quantity by

$$\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right)$$

giving

$$p(y|x; v) = \frac{\exp\left(v \cdot f(x, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right)} \tag{3}$$

then it is easy to verify that we have a well-formed distribution: that is,

$$\sum_{y \in \mathcal{Y}} p(y|x; v) = 1$$

Thus the denominator in Eq. 3 is a normalization term, which ensures that we have a distribution that sums to one. In summary, the function

$$\frac{\exp\left(v \cdot f(x, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right)}$$

performs a transformation which takes as input a set of values $\{v \cdot f(x, y) : y \in \mathcal{Y}\}$, where each $v \cdot f(x, y)$ can take any value in the reals, and as output produces a probability distribution over the labels $y \in \mathcal{Y}$.

Finally, we consider where the name log-linear models originates from. It follows from the above definitions that

$$\begin{aligned} \log p(y|x; v) &= v \cdot f(x, y) - \log \sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right) \\ &= v \cdot f(x, y) - g(x) \end{aligned}$$

where

$$g(x) = \log \sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right)$$

The first term, $v \cdot f(x, y)$, is linear in the features $f(x, y)$. The second term, $g(x)$, depends only on x, and does not depend on the label y. Hence the log probability $\log p(y|x; v)$ is a linear function in the features $f(x, y)$, as long as we hold x fixed; this justifies the term "log-linear".

7 Parameter Estimation in Log-Linear Models

7.1 The Log-Likelihood Function, and Regularization

We now consider the problem of parameter estimation in log-linear models. We assume that we have a training set, consisting of examples $(x^{(i)}, y^{(i)})$ for $i \in \{1 \dots n\}$, where each $x^{(i)} \in \mathcal{X}$, and each $y^{(i)} \in \mathcal{Y}$.

Given parameter values v, for any example i, we can calculate the log conditional probability

$$\log p(y^{(i)}|x^{(i)}; v)$$

under the model. Intuitively, the higher this value, the better the model fits this particular example. The log-likelihood considers the sum of log probabilities of examples in the training data:

$$L(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v) \tag{4}$$

This is a function of the parameters v. For any parameter vector v, the value of $L(v)$ can be interpreted of a measure of how well the parameter vector fits the training examples.

The first estimation method we will consider is maximum-likelihood estimation, where we choose our parameters as

$$v_{ML} = \arg\max_{v \in \mathbb{R}^d} L(v)$$

In the next section we describe how the parameters v_{ML} can be found efficiently. Intuitively, this estimation method finds the parameters which fit the data as well as possible.

The maximum-likelihood estimates can run into problems, in particular in cases where the number of features in the model is very large. To illustrate, consider the language-modeling problem again, and assume that we have trigram, bigram and unigram features. Now assume that we have some trigram (u, v, w) which is seen only once in the training data; to be concrete, assume that the trigram is *any statistical model*, and assume that this trigram is seen on the 100'th

training example alone. More precisely, we assume that

$$f_{N(any,statistical,model)}(x^{(100)}, y^{(100)}) = 1$$

In addition, assume that this is the only trigram (u, v, w) in training data with $u =$ *any*, and $v =$ *statistical*. In this case, it can be shown that the maximum-likelihood parameter estimate for v_{100} is $+\infty$,[2], which gives

$$p(y^{(100)}|x^{(100)}; v) = 1$$

In fact, we have a very similar situation to the case in maximum-likelihood estimates for regular trigram models, where we would have

$$q_{ML}(\textit{model}|\textit{any, statistical}) = 1$$

for this trigram. As discussed earlier in the class, this model is clearly under-smoothed, and it will generalize badly to new test examples. It is unreasonable to assign

$$P(W_i = \textit{model}|W_{i-1}, W_{i-2} = \textit{any, statistical}) = 1$$

based on the evidence that the bigram *any statistical* is seen once, and on that one instance the bigram is followed by the word *model*.

A very common solution for log-linear models is to modify the objective function in Eq. 4 to include a *regularization term*, which prevents parameter values from becoming too large (and in particular, prevents parameter values from diverging to infinity). A common regularization term is the 2-norm of the parameter values, that is,

$$||v||^2 = \sum_k v_k^2$$

(here $||v||$ is simply the length, or Euclidean norm, of a vector v; i.e., $||v|| = \sqrt{\sum_k v_k^2}$). The modified objective function is

$$L'(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v) - \frac{\lambda}{2} \sum_k v_k^2 \tag{5}$$

[2]It is relatively easy to prove that v_{100} can diverge to ∞. To give a sketch: under the above assumptions, the feature $f_{N(any,statistical,model)}(x, y)$ is equal to 1 on only a single pair $x^{(i)}, y$ where $i \in \{1 \ldots n\}$, and $y \in \mathcal{Y}$, namely the pair $(x^{(100)}, y^{(100)})$. Because of this, as $v_{100} \to \infty$, we will have $p(y^{(100)}|x^{(100)}; v)$ tending closer and closer to a value of 1, with all other values $p(y^{(i)}|x^{(i)}; v)$ remaining unchanged. Thus we can use this one parameter to maximize the value for $\log p(y^{(100)}|x^{(100)}; v)$, independently of the probability of all other examples in the training set.

where $\lambda > 0$ is a parameter, which is typically chosen by validation on some held-out dataset. We again choose the parameter values to maximize the objective function: that is, our optimal parameter values are

$$v^* = \arg\max_v L'(v)$$

The key idea behind the modified objective in Eq. 5 is that we now balance two separate terms. The first term is the log-likelihood on the training data, and can be interpreted as a measure of how well the parameters v fit the training examples. The second term is a penalty on large parameter values: it encourages parameter values to be as close to zero as possible. The parameter λ defines the relative weighting of the two terms. In practice, the final parameters v^* will be a compromise between fitting the data as well as is possible, and keeping their values as small as possible.

In practice, this use of regularization is very effective in smoothing of log-linear models.

7.2 Finding the Optimal Parameters

First, consider finding the maximum-likelihood parameter estimates: that is, the problem of finding

$$v_{ML} = \arg\max_{v \in \mathbb{R}^d} L(v)$$

where

$$L(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v)$$

The bad news is that in the general case, there is no closed-form solution for the maximum-likelihood parameters v_{ML}. The good news is that finding $\arg\max_v L(v)$ is a relatively easy problem, because $L(v)$ can be shown to be a *convex* function. This means that simple gradient-ascent-style methods will find the optimal parameters v_{ML} relatively quickly.

Figure 2 gives a sketch of a gradient-based algorithm for optimization of $L(v)$. The parameter vector is initialized to the vector of all zeros. At each iteration we first calculate the gradients δ_k for $k = 1 \ldots d$. We then move in the direction of the gradient: more precisely, we set $v \leftarrow v + \beta^* \times \delta$ where β^* is chosen to give the optimal improvement in the objective function. This is a "hill-climbing" technique where at each point we compute the steepest direction to move in (i.e., the direction of the gradient); we then move the distance in that direction which gives the greatest value for $L(v)$.

Simple gradient ascent, as shown in figure 2, can be rather slow to converge. Fortunately there are many standard packages for gradient-based optimization,

Initialization: $v = 0$

Iterate until convergence:

- Calculate $\delta_k = \frac{dL(v)}{dv_k}$ for $k = 1 \ldots d$
- Calculate $\beta^* = \arg\max_{\beta \in \mathbb{R}} L(v + \beta\delta)$ where δ is the vector with components δ_k for $k = 1 \ldots d$ (this step is performed using some type of line search)
- Set $v \leftarrow v + \beta^* \delta$

Figure 2: A gradient ascent algorithm for optimization of $L(v)$.

which use more sophisticated algorithms, and which give considerably faster convergence. As one example, a commonly used method for parameter estimation in log-linear models is LBFGs. LBFGs is again a gradient method, but it makes a more intelligent choice of search direction at each step. It does however rely on the computation of $L(v)$ and $\frac{dL(v)}{dv_k}$ for $k = 1$ at each step—in fact this is the only information it requires about the function being optimized. In summary, if we can compute $L(v)$ and $\frac{dL(v)}{dv_k}$ efficiently, then it is simple to use an existing gradient-based optimization package (e.g., based on LBFGs) to find the maximum-likelihood estimates.

Optimization of the regularized objective function,

$$L'(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v) - \frac{\lambda}{2} \sum_k v_k^2$$

can be performed in a very similar manner, using gradient-based methods. $L'(v)$ is also a convex function, so a gradient-based method will find the global optimum of the parameter estimates.

The one remaining step is to describe how the gradients

$$\frac{dL(v)}{dv_k}$$

and

$$\frac{dL'(v)}{dv_k}$$

can be calculated. This is the topic of the next section.

7.3 Gradients

We first consider the derivatives

$$\frac{dL(v)}{dv_k}$$

where

$$L(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v)$$

It is relatively easy to show (see the appendix of this note), that for any $k \in \{1 \dots d\}$,

$$\frac{dL(v)}{dv_k} = \sum_{i=1}^{n} f_k(x^{(i)}, y^{(i)}) - \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} p(y|x^{(i)}; v) f_k(x^{(i)}, y) \tag{6}$$

where as before

$$p(y|x^{(i)}; v) = \frac{\exp\left(v \cdot f(x^{(i)}, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)}$$

The expression in Eq. 6 has a quite intuitive form. The first part of the expression,

$$\sum_{i=1}^{n} f_k(x^{(i)}, y^{(i)})$$

is simply the number of times that the feature f_k is equal to 1 on the training examples (assuming that f_k is an indicator function; i.e., assuming that $f_k(x^{(i)}, y^{(i)})$ is either 1 or 0). The second part of the expression,

$$\sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} p(y|x^{(i)}; v) f_k(x^{(i)}, y)$$

can be interpreted as the expected number of times the feature is equal to 1, where the expectation is taken with respect to the distribution

$$p(y|x^{(i)}; v) = \frac{\exp\left(v \cdot f(x^{(i)}, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)}$$

specified by the current parameters. The gradient is then the difference of these terms. It can be seen that the gradient is easily calculated.

The gradients

$$\frac{dL'(v)}{dv_k}$$

where

$$L'(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v) - \frac{\lambda}{2} \sum_k v_k^2$$

are derived in a very similar way. We have

$$\frac{d}{dv_k} \left(\sum_k v_k^2 \right) = 2v_k$$

hence

$$\frac{dL'(v)}{dv_k} = \sum_{i=1}^{n} f_k(x^{(i)}, y^{(i)}) - \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} p(y|x^{(i)}; v) f_k(x^{(i)}, y) - \lambda v_k \qquad (7)$$

Thus the only difference from the gradient in Eq. 6 is the additional term $-\lambda v_k$ in this expression.

A Calculation of the Derivatives

In this appendix we show how to derive the expression for the derivatives, as given in Eq. 6. Our goal is to find an expression for

$$\frac{dL(v)}{dv_k}$$

where

$$L(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v)$$

First, consider a single term $\log p(y^{(i)}|x^{(i)}; v)$. Because

$$p(y^{(i)}|x^{(i)}; v) = \frac{\exp\left(v \cdot f(x^{(i)}, y^{(i)})\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)}$$

we have

$$\log p(y^{(i)}|x^{(i)}; v) = v \cdot f(x^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)$$

The derivative of the first term in this expression is simple:

$$\frac{d}{dv_k} \left(v \cdot f(x^{(i)}, y^{(i)})\right) = \frac{d}{dv_k} \left(\sum_k v_k f_k(x^{(i)}, y^{(i)}) \right) = f_k(x^{(i)}, y^{(i)}) \qquad (8)$$

Now consider the second term. This takes the form

$$\log g(v)$$

where

$$g(v) = \sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)$$

By the usual rules of differentiation,

$$\frac{d}{dv_k} \log g(v) = \frac{\frac{d}{dv_k}(g(v))}{g(v)}$$

In addition, it can be verified that

$$\frac{d}{dv_k} g(v) = \sum_{y' \in \mathcal{Y}} f_k(x^{(i)}, y') \exp\left(v \cdot f(x^{(i)}, y')\right)$$

hence

$$\begin{aligned}
\frac{d}{dv_k} \log g(v) &= \frac{\frac{d}{dv_k}(g(v))}{g(v)} && (9)\\
&= \frac{\sum_{y' \in \mathcal{Y}} f_k(x^{(i)}, y') \exp\left(v \cdot f(x^{(i)}, y')\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)} && (10)\\
&= \sum_{y' \in \mathcal{Y}} \left(f_k(x^{(i)}, y') \times \frac{\exp\left(v \cdot f(x^{(i)}, y')\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)} \right) && (11)\\
&= \sum_{y' \in \mathcal{Y}} f_k(x^{(i)}, y') p(y'|x; v) && (12)
\end{aligned}$$

Combining Eqs 8 and 12 gives

$$\frac{dL(v)}{dv_k} = \sum_{i=1}^{n} f_k(x^{(i)}, y^{(i)}) - \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} p(y|x^{(i)}; v) f_k(x^{(i)}, y)$$

Chapter 8

MEMMs (Log-Linear Tagging Models)

8.1 Introduction

In this chapter we return to the problem of tagging. We previously described hidden Markov models (HMMs) for tagging problems. This chapter describes a powerful alternative to HMMs, *log-linear tagging models*, which build directly on ideas from log-linear models. A key advantage of log-linear tagging models is that they allow highly flexible representations, allowing features to be easily integrated in the model.

Log-linear tagging models are sometimes referred to as "maximum entropy Markov models (MEMMs)".[1] We will use the terms "MEMM" and "log-linear tagging model" interchangeably in this chapter. The name MEMM was first introduced by McCallum et al. (2000).

Log-linear tagging models are conditional tagging models. Recall that a generative tagging model defines a joint distribution $p(x_1 \dots x_n, y_1 \dots y_n)$ over sentences $x_1 \dots x_n$ paired with tag sequences $y_1 \dots y_n$. In contrast, a conditional tagging model defines a conditional distribution

$$p(y_1 \dots y_n | x_1 \dots x_n)$$

corresponding to the probability of the tag sequence $y_1 \dots y_n$ conditioned on the input sentence $x_1 \dots x_n$. We give the following definition:

[1] This name is used because 1) log-linear models are also referred to as maximum entropy models, as it can be shown in the unregularized case that the maximum likelihood estimates maximize an entropic measure subject to certain linear constraints; 2) as we will see shortly, MEMMs make a Markov assumption that is closely related to the Markov assumption used in HMMs.

Definition 1 (Conditional Tagging Models) *A conditional tagging model consists of:*

- *A set of words $\mathcal{V}$ (this set may be finite, countably infinite, or even uncountably infinite).*
- *A finite set of tags $\mathcal{K}$.*
- *A function $p(y_1 \dots y_n | x_1 \dots x_n)$ such that:*

  1. *For any $\langle x_1 \dots x_n, y_1 \dots y_n \rangle \in \mathcal{S}$,*

  $$p(y_1 \dots y_n | x_1 \dots x_n) \geq 0$$

  where $\mathcal{S}$ is the set of all sequence/tag-sequence pairs $\langle x_1 \dots x_n, y_1 \dots y_n \rangle$ such that $n \geq 1$, $x_i \in \mathcal{V}$ for $i = 1 \dots n$, and $y_i \in \mathcal{K}$ for $i = 1 \dots n$.

  2. *For any $x_1 \dots x_n$ such that $n \geq 1$ and $x_i \in \mathcal{V}$ for $i = 1 \dots n$,*

  $$\sum_{y_1 \dots y_n \in \mathcal{Y}(n)} p(y_1 \dots y_n | x_1 \dots x_n) = 1$$

  where $\mathcal{Y}(n)$ is the set of all tag sequences $y_1 \dots y_n$ such that $y_i \in \mathcal{K}$ for $i = 1 \dots n$.

Given a conditional tagging model, the function from sentences $x_1 \dots x_n$ to tag sequences $y_1 \dots y_n$ is defined as

$$f(x_1 \dots x_n) = \arg \max_{y_1 \dots y_n \in \mathcal{Y}(n)} p(y_1 \dots y_n | x_1 \dots x_n)$$

Thus for any input $x_1 \dots x_n$, we take the highest probability tag sequence as the output from the model. □

We are left with the following three questions:

- How we define a conditional tagging model $p(y_1 \dots y_n | x_1 \dots x_n)$?
- How do we estimate the parameters of the model from training examples?
- How do we efficiently find

  $$\arg \max_{y_1 \dots y_n \in \mathcal{Y}(n)} p(y_1 \dots y_n | x_1 \dots x_n)$$

  for any input $x_1 \dots x_n$?

The remainder of this chapter describes how MEMMs give a solution to these questions. In brief, a log-linear model is used to define a conditional tagging model. The parameters of the model can be estimated using standard methods for parameter estimation in log-linear models. MEMMs make a Markov independence assumption that is closely related to the Markov independence assumption in HMMs, and that allows the $\arg\max$ above to be computed efficiently using dynamic programming.

8.2 Trigram MEMMs

This section describes the model form for MEMMs. We focus on *trigram* MEMMs, which make a second order Markov assumption, where each tag depends on the previous two tags. The generalization to other Markov orders, for example first-order (bigram) or third-order (four-gram) MEMMs, is straightforward.

Our task is to model the conditional distribution

$$P(Y_1 = y_1 \dots Y_n = y_n | X_1 = x_1 \dots X_n = x_n)$$

for any input sequence $x_1 \dots x_n$ paired with a tag sequence $y_1 \dots y_n$.

We first use the following decomposition:

$$\begin{aligned} & P(Y_1 = y_1 \dots Y_n = y_n | X_1 = x_1 \dots X_n = x_n) \\ = & \prod_{i=1}^{n} P(Y_i = y_i | X_1 = x_1 \dots X_n = x_n, Y_1 = y_1 \dots Y_{i-1} = y_{i-1}) \\ = & \prod_{i=1}^{n} P(Y_i = y_i | X_1 = x_1 \dots X_n = x_n, Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \end{aligned}$$

The first equality is exact, by the chain rule of probabilities. The second equality makes use of a trigram independence assumption, namely that for any $i \in \{1 \dots n\}$,

$$\begin{aligned} & P(Y_i = y_i | X_1 = x_1 \dots X_n = x_n, Y_1 = y_1 \dots Y_{i-1} = y_{i-1}) \\ = & P(Y_i = y_i | X_1 = x_1 \dots X_n = x_n, Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \end{aligned}$$

Here we assume that $y_{-1} = y_0 = *$, where * is a special symbol in the model denoting the start of a sequence.

Thus we assume that the random variable Y_i is independent of the values for $Y_1 \dots Y_{i-3}$, once we condition on the entire input sequence $X_1 \dots X_n$, and the previous two tags Y_{i-2} and Y_{i-1}. This is a trigram independence assumption, where each tag depends only on the previous two tags. We will see that this independence

assumption allows us to use dynamic programming to efficiently find the highest probability tag sequence for any input sentence $x_1 \dots x_n$.

Note that there is some resemblance to the independence assumption made in trigram HMMs, namely that

$$\begin{aligned} & P(Y_i = y_i | Y_1 = y_1 \dots Y_{i-1} = y_{i-1}) \\ = \quad & P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \end{aligned}$$

The key difference is that we now condition on the entire input sequence $x_1 \dots x_n$, in addition to the previous two tags y_{i-2} and y_{i-1}.

The final step is to use a log-linear model to estimate the probability

$$P(Y_i = y_i | X_1 = x_1 \dots X_n = x_n, Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1})$$

For any pair of sequences $x_1 \dots x_n$ and $y_1 \dots y_n$, we define the i'th "history" h_i to be the four-tuple

$$h_i = \langle y_{i-2}, y_{i-1}, x_1 \dots x_n, i \rangle$$

Thus h_i captures the conditioning information for tag y_i in the sequence, in addition to the position i in the sequence. We assume that we have a feature-vector representation $f(h_i, y) \in \mathbb{R}^d$ for any history h_i paired with any tag $y \in \mathcal{K}$. The feature vector could potentially take into account any information in the history h_i and the tag y. As one example, we might have features

$$f_1(h_i, y) = \begin{cases} 1 & \text{if } x_i = \texttt{the} \text{ and } y = \texttt{DT} \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(h_i, y) = \begin{cases} 1 & \text{if } y_{i-1} = \texttt{V} \text{ and } y = \texttt{DT} \\ 0 & \text{otherwise} \end{cases}$$

Section 8.3 describes a much more complete set of example features.

Finally, we assume a parameter vector $\theta \in \mathbb{R}^d$, and that

$$\begin{aligned} & P(Y_i = y_i | X_1 = x_1 \dots X_n = x_n, Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \\ = \quad & \frac{\exp(\theta \cdot f(h_i, y_i))}{\sum_{y \in \mathcal{K}} \exp(\theta \cdot f(h_i, y))} \end{aligned}$$

Putting this all together gives the following:

Definition 2 (Trigram MEMMs) *A trigram MEMM consists of:*

- *A set of words $\mathcal{V}$ (this set may be finite, countably infinite, or even uncountably infinite).*
- *A finite set of tags $\mathcal{K}$.*
- *Given $\mathcal{V}$ and $\mathcal{K}$, define $\mathcal{H}$ to be the set of all possible histories. The set $\mathcal{H}$ contains all four-tuples of the form $\langle y_{-2}, y_{-1}, x_1 \dots x_n, i \rangle$, where $y_{-2} \in \mathcal{K} \cup \{*\}$, $y_{-1} \in \mathcal{K} \cup \{*\}$, $n \geq 1$, $x_i \in \mathcal{V}$ for $i = 1 \dots n$, $i \in \{1 \dots n\}$. Here * is a special "start" symbol.*
- *An integer d specifying the number of features in the model.*
- *A function $f : \mathcal{H} \times \mathcal{K} \rightarrow \mathbb{R}^d$ specifying the features in the model.*
- *A parameter vector $\theta \in \mathbb{R}^d$.*

Given these components we define the conditional tagging model

$$p(y_1 \dots y_n | x_1 \dots x_n) = \prod_{i=1}^{n} p(y_i | h_i; \theta)$$

where $h_i = \langle y_{i-2}, y_{i-1}, x_1 \dots x_n, i \rangle$, and

$$p(y_i | h_i; \theta) = \frac{\exp(\theta \cdot f(h_i, y_i))}{\sum_{y \in \mathcal{K}} \exp(\theta \cdot f(h_i, y))}$$

□

At this point there are a number of questions. How do we define the feature vectors $f(h_i, y)$? How do we learn the parameters θ from training data? How do we find the highest probability tag sequence

$$\arg \max_{y_1 \dots y_n \in \mathcal{Y}(n)} p(y_1 \dots y_n | x_1 \dots x_n)$$

for an input sequence $x_1 \dots x_n$? The following sections answer these questions.

8.3 Features in Trigram MEMMs

Recall that the feature vector definition in a trigram MEMM is a function $f(h, y) \in \mathbb{R}^d$ where $h = \langle y_{-2}, y_{-1}, x_1 \dots x_n, i \rangle$ is a history, $y \in \mathcal{K}$ is a tag, and d is an integer specifying the number of features in the model. Each feature $f_j(h, y)$ for $j \in \{1 \dots d\}$ can potentially be sensitive to *any* information in the history h in

conjunction with the tag y. This will lead to a great deal of flexibility in the model. This is the primary advantage of trigram MEMMs over trigram HMMs for tagging: a much richer set of features can be employed for the tagging task.

In this section we give an example of how features can be defined for the part-of-speech (POS) tagging problem. The features we describe are taken from Ratnaparkhi (1996); Ratnaparkhi's experiments show that they give competitive performance on the POS tagging problem for English. Throughout this section we assume that the history h is a four-tuple $\langle y_{-2}, y_{-1}, x_1 \ldots x_n, i \rangle$. The features are as follows:

Word/tag features One example word/tag feature is the following:

$$f_{100}(h, y) = \begin{cases} 1 & \text{if } x_i \text{ is } \texttt{base} \text{ and } y = \texttt{VB} \\ 0 & \text{otherwise} \end{cases}$$

This feature is sensitive to the word being tagged, x_i, and the proposed tag for that word, y. In practice we would introduce features of this form for a very large set of word/tag pairs, in addition to the pair base/VB. For example, we could introduce features of this form for all word/tag pairs seen in training data.

This class of feature allows the model to capture the tendency for particular words to take particular parts of speech. In this sense it plays an analogous role to the emission parameters $e(x|y)$ in a trigram HMM. For example, given the definition of f_{100} given above, a large positive value for θ_{100} will indicate that base is very likely to be tagged as a VB; conversely a highly negative value will indicate that this particular word/tag pairing is unlikely.

Prefix and Suffix features An example of a suffix feature is as follows:

$$f_{101}(h, y) = \begin{cases} 1 & \text{if } x_i \text{ ends in } \texttt{ing} \text{ and } y = \texttt{VBG} \\ 0 & \text{otherwise} \end{cases}$$

This feature is sensitive to the suffix of the word being tagged, x_i, and the proposed tag y. In practice we would introduce a large number of features of this form. For example, in Ratnaparkhi's POS tagger, all suffixes seen in training data up to four letters in length were introduced as features (in combination with all possible tags).

An example of a *prefix* feature is as follows:

$$f_{102}(h, y) = \begin{cases} 1 & \text{if } x_i \text{ starts with } \texttt{pre} \text{ and } y = \texttt{NN} \\ 0 & \text{otherwise} \end{cases}$$

Again, a large number of prefix features would be used. Ratnaparkhi's POS tagger employs features for all prefixes up to length four seen in training data.

Prefix and suffix features are very useful for POS tagging and other tasks in English and many other languages. For example, the suffix `ing` is frequently seen with the tag `VBG` in the Penn treebank, which is the tag used for gerunds; there are many other examples.

Crucially, it is very straightforward to introduce prefix and suffix features to the model. This is in contrast with trigram HMMs for tagging, where we used the idea of mapping low-frequency words to "pseudo-words" capturing spelling features. The integration of spelling features—for example prefix and suffix features—in log-linear models is much less ad-hoc than the method we described for HMMs. Spelling features are in practice very useful when tagging words in test data that are infrequent or not seen at all in training data.

Trigram, Bigram and Unigram Tag features An example of a trigram tag feature is as follows:

$$f_{103}(h, y) = \begin{cases} 1 & \text{if } \langle y_{-2}, y_{-1}, y \rangle = \langle \text{DT, JJ, VB} \rangle \\ 0 & \text{otherwise} \end{cases}$$

This feature, in combination with the associated parameter θ_{103}, plays an analogous role to the $q(\text{VB}|\text{DT, JJ})$ parameter in a trigram HMM, allowing the model to learn whether the tag trigram $\langle \text{DT, JJ, VB} \rangle$ is likely or unlikely. Features of this form are introduced for a large number of tag trigrams, for example all tag trigrams seen in training data.

The following two features are examples of bigram and unigram tag features:

$$f_{104}(h, y) = \begin{cases} 1 & \text{if } \langle y_{-1}, y \rangle = \langle \text{JJ, VB} \rangle \\ 0 & \text{otherwise} \end{cases}$$

$$f_{105}(h, y) = \begin{cases} 1 & \text{if } \langle y \rangle = \langle \text{VB} \rangle \\ 0 & \text{otherwise} \end{cases}$$

The first feature allows the model to learn whether the tag bigram `JJ VB` is likely or unlikely; the second feature allows the model to learn whether the tag `VB` is likely or unlikely. Again, a large number of bigram and unigram tag features are typically introduced in the model.

Bigram and unigram features may at first glance seem redundant, given the obvious overlap with trigram features. For example, it might seem that features f_{104} and f_{105} are subsumed by feature f_{103}. Specifically, given parameters $\theta_{103}, \theta_{104}$ and θ_{105} we can redefine θ_{103} to be equal to $\theta_{103}+\theta_{104}+\theta_{105}$, and $\theta_{104} = \theta_{105} = 0$,

giving exactly the same distribution $p(y|x;\theta)$ for the two parameter settings.[2] Thus features f_{104} and f_{105} can apparently be eliminated. However, when used in conjunction with regularized approaches to parameter estimation, the bigram and unigram features play an important role that is analogous to the use of "backed-off" estimates $q_{\text{ML}}(\text{VB}|\text{JJ})$ and $q_{\text{ML}}(\text{VB})$ in the smoothed estimation techniques seen previously in the class. Roughly speaking, with regularization, if the trigram in feature f_{103} is infrequent, then the value for θ_{103} will not grow too large in magnitude, and the parameter values θ_{104} and θ_{105}, which are estimated based on more examples, will play a more important role.

Other Contextual Features Ratnaparkhi also used features which consider the word before or after x_i, in conjunction with the proposed tag. Example features are as follows:

$$f_{106}(h,y) = \begin{cases} 1 & \text{if previous word } x_{i-1} = \textit{the} \text{ and } y = \text{VB} \\ 0 & \text{otherwise} \end{cases}$$

$$f_{107}(h,y) = \begin{cases} 1 & \text{if next word } x_{i+1} = \textit{the} \text{ and } y = \text{VB} \\ 0 & \text{otherwise} \end{cases}$$

Again, many such features would be introduced. These features add additional context to the model, introducing dependencies between x_{i-1} or x_{i+1} and the proposed tag. Note again that it is not at all obvious how to introduce contextual features of this form to trigram HMMs.

Other Features We have described the main features in Ratnaparkhi's model. Additional features that he includes are: 1) spelling features which consider whether the word x_i being tagged contains a number, contains a hyphen, or contains an upper-case letter; 2) contextual features that consider the word at x_{i-2} and x_{i+2} in conjunction with the current tag y.

8.4 Parameter Estimation in Trigram MEMMs

Parameter estimation in trigram MEMMs can be performed using the parameter estimation methods for log-linear models described in the previous chapter. The training data is a set of m examples $(x^{(k)}, y^{(k)})$ for $k = 1 \ldots m$, where each $x^{(k)}$ is a sequence of words $x_1^{(k)} \ldots x_{n_k}^{(k)}$, and each $y^{(k)}$ is a sequence of tags $y_1^{(k)} \ldots y_{n_k}^{(k)}$.

[2] To be precise, this argument is correct in the case where for every unigram and bigram feature there is at least one trigram feature that subsumes it. The reassignment of parameter values would be applied to all trigrams.

Here n_k is the length of the k'th sentence or tag sequence. For any $k \in \{1 \dots m\}$, $i \in \{1 \dots n_k\}$, define the history $h_i^{(k)}$ to be equal to $\langle y_{i-2}^{(k)}, y_{i-1}^{(k)}, x_1^{(k)} \dots x_{n_k}^{(k)}, i \rangle$. We assume that we have a feature vector definition $f(h, y) \in \mathbb{R}^d$ for some integer d, and hence that

$$p(y_i^{(k)} | h_i^{(k)}; \theta) = \frac{\exp\left(\theta \cdot f(h_i^{(k)}, y_i)\right)}{\sum_{y \in \mathcal{K}} \exp\left(\theta \cdot f(h_i^{(k)}, y)\right)}$$

The regularized log-likelihood function is then

$$L(\theta) = \sum_{k=1}^{m} \sum_{i=1}^{n_k} \log p(y_i^{(k)} | h_i^{(k)}; \theta) - \frac{\lambda}{2} \sum_{j=1}^{d} \theta_j^2$$

The first term is the log-likelihood of the data under parameters θ. The second term is a regularization term, which penalizes large parameter values. The positive parameter λ dictates the relative weight of the two terms. An optimization method is used to find the parameters θ^* that maximize this function:

$$\theta^* = \arg\max_{\theta \in \mathbb{R}^d} L(\theta)$$

In summary, the estimation method is a direct application of the method described in the previous chapter, for parameter estimation in log-linear models.

8.5 Decoding with MEMMs: Another Application of the Viterbi Algorithm

We now turn to the problem of finding the most likely tag sequence for an input sequence $x_1 \dots x_n$ under a trigram MEMM; that is, the problem of finding

$$\arg \max_{y_1 \dots y_n \in \mathcal{Y}(n)} p(y_1 \dots y_n | x_1 \dots x_n)$$

where

$$p(y_1 \dots y_n | x_1 \dots x_n) = \prod_{i=1}^{n} p(y_i | h_i; \theta)$$

and $h_i = \langle y_{i-2}, y_{i-1}, x_1 \dots x_n, i \rangle$.

First, note the similarity to the decoding problem for a trigram HMM tagger, which is to find

$$\arg \max_{y_1 \dots y_n \in \mathcal{Y}(n)} q(\text{STOP} | y_{n-1}, y_n) \times \prod_{i=1}^{n} q(y_i | y_{i-2}, y_{i-1}) e(x_i | y_i)$$

Putting aside the $q(\text{STOP}|y_{n-1}, y_n)$ term, we have essentially replaced

$$\prod_{i=1}^{n} q(y_i|y_{i-2}, y_{i-1}) \times e(x_i|y_i)$$

by

$$\prod_{i=1}^{n} p(y_i|h_i; \theta)$$

The similarity of the two decoding problems leads to the decoding algorithm for trigram MEMMs being very similar to the decoding algorithm for trigram HMMs. We again use dynamic programming. The algorithm is shown in figure 8.1. The base case of the dynamic program is identical to the base case for trigram HMMs, namely

$$\pi(*, *, 0) = 1$$

The recursive case is

$$\pi(k, u, v) = \max_{w \in \mathcal{K}_{k-2}} (\pi(k-1, w, u) \times p(v|h; \theta))$$

where $h = \langle w, u, x_1 \dots x_n, k \rangle$. (We again define $\mathcal{K}_{-1} = \mathcal{K}_0 = *$, and $\mathcal{K}_k = \mathcal{K}$ for $k = 1 \dots n$.)

Recall that the recursive case for a trigram HMM tagger is

$$\pi(k, u, v) = \max_{w \in \mathcal{K}_{k-2}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

Hence we have simply replaced $q(v|w, u) \times e(x_k|v)$ by $p(v|h; \theta)$.

The justification for the algorithm is very similar to the justification of the Viterbi algorithm for trigram HMMs. In particular, it can be shown that for all $k \in \{0 \dots n\}$, $u \in \mathcal{K}_{k-1}$, $v \in \mathcal{K}_k$,

$$\pi(k, u, v) = \max_{y_{-1} \dots y_k \in S(k,u,v)} \prod_{i=1}^{k} p(y_i|h_i; \theta)$$

where $S(k, u, v)$ is the set of all sequences $y_{-1} y_0 \dots y_k$ such that $y_i \in \mathcal{K}_i$ for $i = -1 \dots k$, and $y_{k-1} = u$, $y_k = v$.

8.6 Summary

To summarize, the main ideas behind trigram MEMMs are the following:

Input: A sentence $x_1 \dots x_n$. A set of possible tags $\mathcal{K}$. A model (for example a log-linear model) that defines a probability

$$p(y|h;\theta)$$

for any h, y pair where h is a history of the form $\langle y_{-2}, y_{-1}, x_1 \dots x_n, i\rangle$, and $y \in \mathcal{K}$.
Definitions: Define $\mathcal{K}_{-1} = \mathcal{K}_0 = \{*\}$, and $\mathcal{K}_k = \mathcal{K}$ for $k = 1 \dots n$.
Initialization: Set $\pi(0, *, *) = 1$.
Algorithm:

- For $k = 1 \dots n$,
  - For $u \in \mathcal{K}_{k-1}$, $v \in \mathcal{K}_k$,

$$\pi(k, u, v) = \max_{w \in \mathcal{K}_{k-2}} (\pi(k-1, w, u) \times p(v|h;\theta))$$

$$bp(k, u, v) = \arg\max_{w \in \mathcal{K}_{k-2}} (\pi(k-1, w, u) \times p(v|h;\theta))$$

    where $h = \langle w, u, x_1 \dots x_n, k\rangle$.

- Set $(y_{n-1}, y_n) = \arg\max_{u \in \mathcal{K}_{n-1}, v \in \mathcal{K}_n} \pi(n, u, v)$
- For $k = (n-2) \dots 1$,

$$y_k = bp(k+2, y_{k+1}, y_{k+2})$$

- **Return** the tag sequence $y_1 \dots y_n$

Figure 8.1: The Viterbi Algorithm with backpointers.

1. We derive the model by first making the independence assumption

$$\begin{aligned} & P(Y_i = y_i | X_1 = x_1 \ldots X_n = x_n, Y_1 = y_1 \ldots Y_n = y_n) \\ = \quad & P(Y_i = y_i | X_1 = x_1 \ldots X_n = x_n, Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \end{aligned}$$

and then assuming that

$$\begin{aligned} & P(Y_i = y_i | X_1 = x_1 \ldots X_n = x_n, Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \\ = \quad & p(y_i | h_i; \theta) \\ = \quad & \frac{\exp\{\theta \cdot f(h_i, y_i)\}}{\sum_y \exp\{\theta \cdot f(h_i, y)\}} \end{aligned}$$

where $h_i = \langle y_{i-2}, y_{i-1}, x_1 \ldots x_n, i \rangle$, $f(h, y) \in \mathbb{R}^d$ is a feature vector, and $\theta \in \mathbb{R}^d$ is a parameter vector.

2. The parameters θ can be estimated using standard methods for parameter estimation in log-linear models, for example by optimizing a regularized log-likelihood function.

3. The decoding problem is to find

$$\arg \max_{y_1 \ldots y_n \in \mathcal{Y}(n)} \prod_{i=1}^{n} p(y_i | h_i; \theta)$$

This problem can be solved by dynamic programming, using a variant of the Viterbi algorithm. The algorithm is closely related to the Viterbi algorithm for trigram HMMs.

4. The feature vector $f(h, y)$ can be sensitive to a wide range of information in the history h in conjunction with the tag y. This is the primary advantage of MEMMs over HMMs for tagging: it is much more straightforward and direct to introduce features into the model.