Chapter 1

Language modeling and probability

1.1 Introduction

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

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

This chapter introduces language models and applies them to a somewhat simpler task, namely *language identification*, i.e., identifying the language in which a text is written. This might be useful in order to determine how to translate a random page found on the Web, for example. Any solution must rely on the fact that fluent phrases and sentences of one language are usually not also fluent in a second language as well. Thus we can identify languages by feeding the text to language models for each of the languages it could belong to, and using their output to guess the text's language.

Virtually all of the methods for language modeling are probabilistic in nature. That is, instead of making a categorical (i.e., yes-or-no) judgement about the fluency of sequence of words, they return (an estimate of) the probability of the sequence. The probability of a sequence is a real number

between 0 and 1, and high probability sequences are more likely to occur than a low probability ones. Thus a model that estimates the probability that a sequence of words is a phrase or sentence permits us to rank different sequences in terms of their fluency; it can answer the question with which we started this chapter.

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

1.2 A brief introduction to probability

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

1.2.1 Outcomes, events and probabilities

To make this still more precise, suppose the urn contains 1,000 strips of paper. These 1,000 strips constitute the *sample space* or the set of all possible outcomes, and is usually denoted by Ω (the Greek letter Omega). In discrete sample spaces such as this, each sample $x \in \Omega$ is assigned a *probability* $\mathrm{P}(x)$, which is a real number that indicates how likely each sample is to occur. If we assume that we are equally likely to choose any of the strips in the urn, then $\mathrm{P}(x) = 0.001$ for all $x \in \Omega$. In general, we require that the probabilities of the samples satisfy the following constraints:

1. $\mathrm{P}(x) \in [0, 1]$ for all $x \in \Omega$, i.e., probabilities are real numbers between 0 and 1, and

2. $\sum_{x \in \Omega} \mathrm{P}(x) = 1$, i.e., the probabilities of all samples sum to one.

Of course, many of these strips of paper have the same word. For example, if the pages you cut up were written in English the word '*the*' is likely to appear on more than 50 of your strips of paper. The 1,000 strips of

paper each correspond to a different word *token* or occurrence of a word in the urn, so the urn contains 1,000 word tokens. But because many of these strips contain the same word, the number of word *types* (i.e., distinct words) labeling these strips is much smaller; our urn might contain only 250 word types.

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

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

We can treat each word type as an event. For example, suppose $E_{\text{`}\mathit{the}\text{'}}$ is the event of drawing a strip labeled '*the*', that $|E_{\text{`}\mathit{the}\text{'}}| = 60$ (i.e., there are 60 strips labeled '*the*') and $\mathrm{P}(x) = 0.001$ for all events $x \in \Omega$, so $\mathrm{P}(E_{\text{`}\mathit{the}\text{'}}) = 60 \times 0.0001 = 0.06$.

1.2.2 Random variables and joint probabilities

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

Given a random variable V and a value v (it's standard to capitalize random variables and use lower-cased variables as their values), $\mathrm{P}(V{=}v)$ is the probability of the event that V takes the value v, i.e.:

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

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

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

which the first letter is 't', and $\mathrm{P}(S{=}\text{'}h\text{'})$ is the probability of the event in which the second letter is 'h'.

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

1.2.3 Conditional and marginal probabilities

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

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

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

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

Example 1.1: Suppose we have an urn containing 10 strips of paper (i.e., our sample space Ω has 10 elements) that are labeled with 5 word types, and the

frequency of each word is as follows:

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

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

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

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

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

Finally, let's compute the *conditional probabilities* of F and S. There are two ways in which this can be done. If we condition on F we obtain the following conditional probabilities:

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

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

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

Newcomers to probability sometimes have trouble distinguishing between the conditional probabilities $\mathrm{P}(A \mid B)$ and $\mathrm{P}(B \mid A)$. They sort of see both as expressing a correlation between A and B. However in general there is no reason to think that they will be the same. In Example 1.1

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

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

1.2.4 Independence

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

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

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

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

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

which is captures the informal definition of independence above.

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

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

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

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

Example 1.2: The random variables F and S in Example 1.1 on page 4 are dependent because $\mathrm{P}(F \mid S) \neq \mathrm{P}(F)$. But if the urn only contained four strips of paper containing '*nab*', '*no*', '*tap*' and '*tot*' respectively then F and S would be independent because $\mathrm{P}(F \mid S) = \mathrm{P}(F) = 1/2$, no matter what values F and S take.

1.2.5 Expectations of random variable

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

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

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

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

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

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

Example 1.3: Suppose X is a random variable generated by throwing a fair 6-sided die. Then the expected value of X is

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

Now suppose that X_1 and X_2 are random variables generated by throwing two fair 6-sided dice. Then the expected value of their sum is:

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

1.3 Modeling documents with unigram language models

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

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

1.3.1 Documents as sequences of words

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

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

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

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

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

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

1.3.2 Language models as models of possible documents

Now we set about building our language models. Formally a *language model* is a probability distribution over possible documents that specifies the probability that the document is in the language.

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

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

1.3.3 Unigram language models

Finding the best way to define a language model $\mathrm{P}(\boldsymbol{W})$ and to estimate its parameters $\boldsymbol{\theta}$ is still a major research topic in computational linguistics. In

this section we introduce an extremely simple class called unigram language models. These are not especially good models of documents, but they can often be good enough to perform simple tasks such as language identification. (Indeed, one of lessons of computational linguistics is that a model doesn't always need to model everything in order to be able to solve many tasks quite well. Figuring out which features are crucial to a task and which can be ignored is a large part of computational linguistics research).

A *unigram language model* assumes that each word W_i is generated independently from the other words. (The name "unigram" comes from the fact that the model's basic units are single words; in section 1.4.1 on page 18 we see how to define bigram and trigram language models, whose basic units are pairs and triples of words respectively). More precisely, a unigram language model defines a distribution over documents as follows:

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

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

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

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

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

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

1.3.4 Maximum likelihood estimates of unigram parameters

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

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

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

where $n_w(\boldsymbol{d})$ is the number of times that word w occurs in $\boldsymbol{d}$ and $n_\circ(\boldsymbol{d}) = \sum_{w \in \mathcal{W}} n_w(\boldsymbol{d})$ is the total number of words in $\boldsymbol{d}$.

Example 1.4: Suppose we have a corpus size $n_\circ(\boldsymbol{d}) = 10^7$ Consider two words, "the" and "equilateral" with counts $2{\cdot}10^5$ and 2, respectively. Their maximum likelihood estimates are 0.02 and $2{\cdot}10^{-7}$.

In fact, we shall see there are good reasons for *not* using this estimator for a unigram language model, but first we look more closely at its name — the *maximum likelihood* estimator for $\boldsymbol{\theta}$.

The maximum likelihood principle provides a general method for deriving estimators for a wide class of models. The principle says: to estimate the value of a parameter θ from data x, select the value $\hat{\theta}$ of θ that makes x as likely as possible. In more detail, suppose we wish to estimate the parameter θ of a probability distribution $\mathrm{P}_\theta(X)$ given data x (i.e., x is the observed value for X). Then the maximum likelihood estimate $\hat{\theta}$ of θ is value of θ that maximizes the *likelihood function* $L_x(\theta) = \mathrm{P}_\theta(x)$. (The value of the likelihood function is equal to the probability of the data, but in a probability distribution the parameter θ is held fixed while the data x varies, while in the likelihood function the data is fixed while the parameter is varied. To take a more concrete example, imagine two computer programs. One, F(date) takes a date, and returns the lead article in the New York Times for that date. The second, G(newspaper) returns the lead article for June second, 1946 for whatever newspaper is requested. These are very different programs, but they have the property that F(6.2.1964)=G(NYT).)

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

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

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

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

Example 1.5: Consider the "document" (we name it ♡) consisting of the phrase "I love you" one hundred times in succession.

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

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

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

1.3.5 Sparse data problems and smoothing

Returning to our goal of using language models for language identification, recall that it's highly likely that our unknown document contains words that don't occur in either of our English or French corpora. It's easy to see that if w is a word that does not appear in our corpus $\boldsymbol{d}$ then the maximum likelihood estimate $\hat{\theta}_w = 0$, as $n_w(\boldsymbol{d}) = 0$. And this together with Equation 1.2 on page 10 means that $\mathrm{P}(\boldsymbol{w}) = 0$ if the document $\boldsymbol{w}$ contains an unknown word.

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

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

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

A popular method for smoothing the maximum likelihood estimator in (1.3) is to add a positive number α_w called a *pseudo-count* to each word w's empirical frequency $n_w(\boldsymbol{d})$ in (1.3), readjusting the denominator so that the revised estimates of $\boldsymbol{\theta}$ still sum to 1. (You can view the pseudo-counts $\boldsymbol{\alpha}$

as counts coming from hypothetical or pseudo-data that are added to the counts $\boldsymbol{n}(\boldsymbol{d})$ that we get from the real data). This means that our smoothed maximum likelihood estimator $\widetilde{\boldsymbol{\theta}}$ is:

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

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

Example 1.6: Let us assume that all w get the same smoothing constant. In this case the equation simplifies to

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

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

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

while the estimate for “equilateral” goes up by 50%

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

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

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

One standard way to address this is to collect an additional corpus of training documents $\boldsymbol{h}$ known as the *heldout corpus*, and use them to set the

smoothing pseudo-count parameters $\boldsymbol{\alpha}$. This makes sense because one of the primary goals of smoothing is to correct, or at least ameliorate, problems caused by sparse data, and a second set of training data $\boldsymbol{h}$ gives us a way of telling just how bad $\boldsymbol{d}$ is as a representative sample. (The same reasoning tells us that $\boldsymbol{h}$ should be “fresh data”, disjoint from the primary training data $\boldsymbol{d}$).

In fact, it's standard at the onset of a research project to split the available data into a primary training corpus $\boldsymbol{d}$ and a secondary heldout training corpus $\boldsymbol{h}$, and perhaps a third *test set* $\boldsymbol{t}$ as well, to be used to evaluate different models trained on $\boldsymbol{d}$ and $\boldsymbol{h}$. (For example, we would want to evaluate a language identification system on its ability to discriminate novel documents, and a test set $\boldsymbol{t}$ disjoint from our training data $\boldsymbol{d}$ and $\boldsymbol{h}$ gives us a way to do this, as we see in section 1.5 on page 25). While one can't give a hard-and-fast rule, it's common to make the heldout corpus $\boldsymbol{h}$ somewhere around 5–10% of the size of $\boldsymbol{d}$.

Finally, it's standard to *bin* or group words into equivalence classes, and assign the same pseudo-count to all words in the same equivalence class. This way we have fewer pseudo-count parameters to estimate. For example, if we group all the words into one single equivalence class, then there is a single pseudo-count value $\alpha = \alpha_w$ that is used for all $w \in \mathcal{W}$. The advantage of doing this is that there is only a single parameter to estimate from our heldout data. Indeed, for simple applications one can sometimes get adequate performance by using one bin and setting $\alpha = 1$. (This is called *add-one smoothing*, or alternatively, *Laplace smoothing*.)

But other more elaborate binning can produce better language models, and this can be important in more demanding applications. Ideally, a binning method should group together words that are likely to be distributed in similar ways in novel documents, because all of the words in the same bin are assigned the same pseudo-count. We can assign words to bins based on their linguistic properties (e.g., we can assign nouns and verbs to different bins), but it is also common to group together words based on their frequency in the training data $\boldsymbol{d}$.

1.3.6 Estimating the smoothing parameters

We now describe how to use a heldout corpus $\boldsymbol{h}$ to estimate the pseudo-counts $\boldsymbol{\alpha}$ of the smoothed maximum likelihood estimator described on page 14. We treat $\boldsymbol{h}$, like $\boldsymbol{d}$, as a long vector of words obtained by concatenating all of the documents in the heldout corpus together. For simplicity we assume that all of the words are grouped into a single bin, so there is a single pseudo-

Figure 1.1: Graph of the function $f(x) = 1 - x^2$.

count $\alpha = \alpha_w$ for all words w. This means that our smoothed maximum likelihood estimate (1.4) for $\boldsymbol{\theta}$ simplifies to:

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

Example 1.7: Suppose our training data $\boldsymbol{d}$ is $\heartsuit$ from Example 1.5 and the held out data $\boldsymbol{h}$ is $\heartsuit'$, which consists of eight copies of '*I love you*' plus one copy each of '*I can love you*' and '*I will love you*'. When we preprocess the held out data both '*can*' and '*will*' become ∗U∗, so $\mathcal{W} = \{$ i love you ∗U∗$\}$. We will let $\alpha = 1$.

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

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

These are then substituted into our normal likelihood equation.

We seek the value $\hat{\alpha}$ of α that maximizes the likelihood $L_{\boldsymbol{h}}$ of the *heldout* corpus $\boldsymbol{h}$ for reasons explained in section 1.3.5 on page 13.

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

Before going on, note that we have introduced a new piece of notation here arg max. It is pronounced just as it is spelled, and it is an abbreviation of "argument maximum". The idea is that $\arg\max_x f(x)$ has as its value the value of x that makes $f(x)$ as large as possible.

Example 1.8: Consider the function $f(x) = 1 - x^2$ as shown in Figure 1.8. The maximum of $f(x)$ is 1, and it occurs when $x = 0$. so $\arg\max_x 1 - x^2 = 0$

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

Example 1.9: As soon as our students do the experiment we will have a plot of negative log probability vs. α for some data.

1.4 Contextual dependencies and n-gram models

The previous section described unigram language models, in which the words (or whatever the basic units of the model are) are each generated as independent entities. This means that unigram language models have no way of capturing contextual dependencies between words in the same sentence or document.

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

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

Methods have been proposed to capture all these dependencies (and

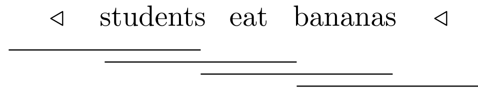

Figure 1.2: The four bigrams extracted by a bigram language from the sentence '*students eat bananas*', padded with '◁' symbols at its beginning and end.

many more), and identifying the important contextual dependencies and coming up with language models that capture them well is still one of the central topics in computational linguistics.

In this section we describe how we can capture one of simplest yet most important kinds of contextual dependency; those that hold between a word and its neighbors. The idea is that we slide a window of width n words over the text, and the overlapping sequences of length n that we see through this window are called *n-grams*. When n=2 the sequences are called *bigrams* and when n=3 the sequences are called *trigrams*. The basic idea is illustrated for n=2 in Figure 1.2.

It turns out that a surprising number of contextual dependencies are visible in an n-word window even for quite small values of n. For example, a bigram language model can distinguish '*students eat bananas*' from '*bananas eat students*'. This is because most linguistic dependencies tend to be local in terms of the word string, even though the actual dependencies may reflect syntactic or other linguistic structure.

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

1.4.1 Bigram language models

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

We can derive the bigram model via a sequence of equations and simplifying approximations. Ignoring the length of $\boldsymbol{W}$ for the moment, we can

decompose the joint probability of the sequence $\boldsymbol{W}$ into a product of conditional probabilities. (This operation is called the *chain rule* and will be used many times in this text.)

$$\mathrm{P}(\boldsymbol{W}) = \mathrm{P}(W_1, \ldots, W_n) \tag{1.5}$$
$$= \mathrm{P}(W_1)\mathrm{P}(W_2|W_1)\ldots\mathrm{P}(W_n|W_{n-1}, \ldots, W_1) \tag{1.6}$$

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

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

for all positions $i \in 1, \ldots, N$. Putting (1.6) and (1.7) together, we have:

$$\mathrm{P}(\boldsymbol{W}) = \mathrm{P}(W_1)\prod_{i=2}^{n}\mathrm{P}(W_i \mid W_{i-1}) \tag{1.7}$$

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

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

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

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

where:

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

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

Example 1.10: A bigram model assigns the following probability to the string '*students eat bananas*'.

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

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

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

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

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

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

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

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

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

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

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

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

Example 1.11: Looking at the $\heartsuit'$ corpus again, we find that

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

The sparse data problems we noticed with the unigram model in section 1.3.5 on page 13 become more serious when we move to the bigram model. In general sparse data problems get worse as we work with n-grams of larger size; if we think of an n-gram model as predicting a word conditioned on the $n-1$ word sequence that precedes it, it becomes increasingly common that the conditioning $n-1$ word sequence only occurs infrequently in the training data $\boldsymbol{d}$, if at all.

For example, consider a word w that only occurs once in our training corpus $\boldsymbol{d}$ (such words are extremely common). Then $n_{w',w}(\boldsymbol{d}) = 1$ for exactly

one word w' and is 0 for all other words. This means that the maximum likelihood estimator is $\widehat{\Theta}_{w',w} = 1$, which corresponds to predicting that w can only be followed by w'. The problem is that we are effectively estimating the distribution over words that follow w from the occurrence of w in $\boldsymbol{d}$, but if there are only very few such occurrences then these estimates are based on very sparse data indeed.

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

$$\widetilde{\Theta}_{w',w} = \frac{n_{w,w'}(\boldsymbol{d}) + \beta_{w,w'}}{n_{w,\circ}(\boldsymbol{d}) + \beta_{w,\circ}} \tag{1.9}$$

where $\beta_{w,\circ} = \sum_{w' \in \mathcal{W}} \beta_{w,w'}$. While it's possible to follow what we did for the unigram model and set $\beta_{w,w'}$ to the same value for all $w, w' \in \mathcal{W}$, it's usually better to make $\beta_{w,w'}$ proportional to the smoothed unigram estimate $\widetilde{\theta}_{w'}$; this corresponds to the assumption that all else being equal, we're more likely to see a high frequency word w' following w than a low frequency one. That is, we set $\beta_{w,w'}$ in (1.9) as follows:

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

where β is a single adjustable constant. Plugging this back into (1.9) we have:

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

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

Example 1.12: Suppose w is "redistribute" and we consider two possible next words w', "food" and "pears" with (assumed) smoothed unigram probabilities 10^{-4} and 10^{-6} respectively. Let β be 1.

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

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

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

$$\begin{aligned}\widetilde{\Theta}_{\text{food,redistribute}} &= \frac{1+10^{-3}}{10+1}\\ \widetilde{\Theta}_{\text{pears,redistribute}} &= \frac{0+10^{-5}}{10+1}\end{aligned}$$

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

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

It's easy to show that the likelihood of the heldout corpus $\boldsymbol{h}$ is:

$$L_{\boldsymbol{h}}(\beta) = \prod_{w',w\in\mathcal{W}} \widetilde{\Theta}_{w',w}^{n_{w,w'}(\boldsymbol{h})} \tag{1.10}$$

where $\widetilde{\Theta}_{w,w'}$ is given by (1.9), and the product in (1.10) need only range over the bigrams (w, w') that actually occur in $\boldsymbol{h}$. (Do you understand why?) Just as in the unigram case, a simple line search can be used to find the value $\hat{\beta}$ of β that optimizes the likelihood (1.10).

1.4.3 Multiple βs

In pseudo-count smoothing the pseudo-count is, in effect, a dial where we move the dial higher if we have little confidence in the representativeness of our training data, and lower when we have more. But particularly when we are dealing with ngrams for $n > 1$ it is usually the case that we have very low confidence in our statistics for some ngrams, and much higher confidence in others. Contrast the situation for "the", where we may have 10^6 occurrences, and "redistribute" where we may only have 10. If after 10^6 samples we have not seen, say, "is" next, we should probably give very low probability to it, even though it is a common word.

For bigrams we can take account of these variations in confidence with different β_ws where w is the conditioning word in the bigram. Again, having one β parameter for each word is overdoing the specificity, but we can bin the w according to how often we have seen w. One β for words we have only seen once, one for 2-3 times, one for 4-7 times, 8-16, 16-32, etc. With, say 20 different βs we can represent a wide range of frequencies.

But setting even 20 different parameters can be difficult. Furthermore, these 20 parameters are not independent of how we set the *alpha* for unigrams. If we move on to trigrams we would be adding another 20 and the interactions get complicated. Thus in most realistic situations we rely on an automatic method, the *expectation maximization algorithm* (known as EM) for their estimation. For pedagogic reasons we introduce EM in the next chapter. However, because of the wide use of EM for the estimation of smoothing parameters, we include a discussion of this in an appendix to this chapter.

1.4.4 Implementing n-gram language models

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

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

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

1.4.5 Speech Recognition and the Noisy Channel Model

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

In fact, we can recast the speech recognition problem in a way that makes language models not just convenient, but (almost) inevitable. From a probabilistic point of view the task confronting a speech recognition is

to find the most like string S in a language given the acoustic signal A. Formally we write this as follows:

$$\arg\max_S \mathrm{P}(S|A) \tag{1.11}$$

We now make the following transformations on this term.

$$\begin{aligned}\arg\max_S \mathrm{P}(S|A) &= \arg\max_S \frac{\mathrm{P}(S)\mathrm{P}(A|S)}{\mathrm{P}(A)} \\ &\propto \arg\max_S \mathrm{P}(S)\mathrm{P}(A|S) \qquad (1.12)\end{aligned}$$

Here we first used Bayes law, and then dropped the $\mathrm{P}(A)$ in the denominator. We can do this because as we vary the S to find the maximum probability the denominator stays constant. (It is, remember, just the sound signal we took in.)

Note the two terms on the right-hand side of Equation 1.12. The first is our language model, the second is called the *acoustic model*. That a language model term arises so directly from the definition of the speech recognition problem is what we meant when we said that language modeling is almost inevitable. It could be avoided, but all serious speech recognition systems have one.

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

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

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

1.5 Programming problems

Problem 1.1: **Language identification using language models**

Suppose we have language models P_e and P_f for two different languages e and f. One simple way to use them to classify an unknown document $\boldsymbol{w}$ is

to compare $\mathrm{P}_e(\boldsymbol{w})$ and $\mathrm{P}_f(\boldsymbol{w})$, and predict the language that makes $\boldsymbol{w}$ most likely. This problem will guide you through the steps involved in doing this.

1. The `/data/Hansards` directory in the materials distributed with this book contains the data we will use to train and test the language models. (This data comes from the Canadian Hansards, which are parlimentary proceedings and appear in both English and French). These files have one sentence per line, and have been tokenized, i.e., split into words. To start with, we will train language models from the Senate Hansards, i.e., train the English language model using `english-senate-0.txt` as the main training data, `english-senate-1.txt` as heldout training data, and `english-senate-2.txt` as test data; and train the French language model from the corresponding French data.

2. Use the training sets from each language to produce a smoothed unigram model for each language with the smoothing parameter $\alpha = 1$. Compute the log probability of each of your language-specific test sets using the language model for that language. Then use your two models to compute the probability of each sentence in the combined test data, and evaluate the accuracy of your classifier

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

4. Now construct smoothed bigram models as described in the text, setting $\beta = 1$, and repeat the evaluation. Do these models do better than the unigram models you constructed in the previous step?

5. Finally, set the bigram smoothing parameter β as described in the text, and repeat the evaluation. What values of β maximize the likelihood of the heldout data? How do these models compare to the other models you have constructed.

Appendix: Using EM to estimate the smoothing parameters

Section 1.3.6 on page 15 described how to use maximum likelihood to estimate the pseudo-count vector $\boldsymbol{\alpha}$ of the smoothed maximum likelihood model

(1.4) from page 14. Here we'll show how to do this using a very general estimation technique known as *Expectation-Maximization* or EM, which we will use many times in this book. Although we won't describe it here, the EM method for estimating α generalizes easily to the situation in which we have grouped words into multiple bins.

First, note that the smoothed multinomial distribution over words $P_{\widetilde{\boldsymbol{\theta}}}(W)$ can be rewritten as a *mixture model*, specifically, as a mixture of the maximum-likelihood model $P_{\hat{\boldsymbol{\theta}}}(W)$ and the *uniform model* $P_1(W)$ that assigns the same probability to each word $w \in \mathcal{W}$.

$$
\begin{aligned}
P_{\widetilde{\boldsymbol{\theta}}}(W{=}w) &= \widetilde{\theta}_w = \frac{n_w(\boldsymbol{d}) + \alpha}{n_\circ(\boldsymbol{d}) + \alpha|\mathcal{W}|} \\
&= \lambda \frac{n_w(\boldsymbol{d})}{n_\circ(\boldsymbol{d})} + (1-\lambda)\frac{1}{\mathcal{W}} \\
&= \lambda P_{\hat{\boldsymbol{\theta}}}(W) + (1-\lambda) P_1(W)
\end{aligned}
$$

where the uniform distribution is $P_1(w) = 1/|\mathcal{W}|$ for all $w \in \mathcal{W}$, and the *mixing parameter* $\lambda \in [0,1]$ satisfies:

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

Informally, this means we can think of the smoothed distribution $P_{\widetilde{\boldsymbol{\theta}}}(W)$ as being generated as follows. For each word W we want to generate, we flip a biased coin that comes out heads with probability λ. If this coin comes out "heads" we generate W from $P_{\hat{\boldsymbol{\theta}}}$, while if it comes out "tails" we generate W from the uniform distribution P_1.

We can formalize this in terms of random variables as follows. In order to generate a word W we first generate a word V from the maximum likelihood distribution $P_{\hat{\boldsymbol{\theta}}}(V)$ and we generate a word U from the uniform distribution $P_1(U)$. Then we flip a coin Z; if it comes out heads (which we record as $Z = 1$) then we set $W = V$, otherwise the coin is tails (i.e., $Z = 0$) and $W = U$. That is:

$$
\begin{aligned}
P(W{=}w) &= P(Z{=}1, V{=}w) \ + \ P(Z{=}0, U{=}w) \\
&= P(Z{=}1)\, P(V{=}w) \ + \ P(Z{=}0)\, P(U{=}w) \\
&= \lambda \hat{\theta}_w \ + \ (1-\lambda)\frac{1}{|\mathcal{W}|}
\end{aligned}
$$

because the coin flip Z is independent of V and U.

Now imagine we actually generated our heldout corpus $\boldsymbol{h}$ in this way. If we could actually observe the values z_i of the coin flips for each word w_i in the corpus we could compute a maximum likelihood estimate of λ as follows:

$$\hat{\lambda} \quad = \quad \frac{n_1(\boldsymbol{z})}{n_\circ(\boldsymbol{z})} \tag{1.13}$$

where $n_1(\boldsymbol{z})$ is the number of times 1 occurs in $\boldsymbol{z}$ and $n_\circ(\boldsymbol{z})$ is the length of $\boldsymbol{z}$. Unfortunately we can't use this formula because we don't observe (i.e., know) $\boldsymbol{z}$, as Z is just a theoretical artifact of our model. (In statistical terminology Z is a *hidden variable*, i.e., a variable whose value is not observed in the data).

However, suppose we have an approximation to λ. We'll see we can use this to compute an estimate $n_1(\boldsymbol{z})$, which we can use in (1.13) to compute $\hat{\lambda}$, which turn out to be an improved estimate of λ. We repeat the entire process as many times as we please using the last improved estimate as the basis for computing the next. Perhaps surprisingly, the entire process is guaranteed to converge to a local maximum of the likelihood $L_{\boldsymbol{h}}$.

We now describe the details of the Expectation Maximization algorithm for estimating λ. Note that we can compute the probability that $Z = 1$ given that we generated a particular word w:

$$\begin{aligned}
\mathrm{P}(Z{=}1 \mid W{=}w) \quad &= \quad \frac{\mathrm{P}(Z{=}1,\, W{=}w)}{\mathrm{P}(W{=}w)} \\
&= \quad \frac{\mathrm{P}(Z{=}1,\, V{=}w)}{\mathrm{P}(W{=}w)} \\
&= \quad \frac{\lambda \hat{\theta}_w}{\lambda \hat{\theta}_w \,+\, (1-\lambda)\dfrac{1}{|\mathcal{W}|}}
\end{aligned}$$

Now imagine we generate the entire heldout set $\boldsymbol{h}$ in this fashion; i.e., we have variables U_i, V_i and Z_i for each word w_i in $\boldsymbol{h}$. Further, because we assume that each word is generated independently, we have:

$$\mathrm{P}(\boldsymbol{Z} \mid \boldsymbol{W}) \quad = \quad \prod_{i=1}^{n_\circ(\boldsymbol{W})} \mathrm{P}(Z_i \mid W_i)$$

The idea behind the Expectation Maximization algorithm is that because we don't know what the "true" value $\boldsymbol{z}$ of the hidden variables $\boldsymbol{Z}$ is, we should treat $\mathrm{P}(\boldsymbol{Z}|\boldsymbol{w})$ as virtual data and estimate λ from it. That is, our

training data consists of $(\boldsymbol{w}, \boldsymbol{z})$ for all possible values $\boldsymbol{z}$ of $\boldsymbol{Z}$ where each datum is weighted by $\mathrm{P}(\boldsymbol{z}|\boldsymbol{w})$.

Fortunately we do not have to enumerate the (exponentially large) virtual training data just described. It turns out it is sufficient to compute the expected value of $n_1(\boldsymbol{Z})$ and use that in (1.13). ($n_1(\boldsymbol{Z})$ is a random quantity because it depends on the random variable $\boldsymbol{Z}$, so it has an expected value). Since $n_1(\boldsymbol{Z}) = \sum_i Z_i$, we can use the linearity of expectations (see section 1.2.5 on page 7) to show that:

$$\begin{aligned} \mathrm{E}[\, n_1 \mid \boldsymbol{w}] &= \sum_{i=1}^{|\boldsymbol{w}|} \mathrm{E}[Z_i \mid \boldsymbol{w}] \\ &= \sum_{i=1}^{|\boldsymbol{w}|} \mathrm{P}(Z_i{=}1 \mid w_i) \\ &= \sum_{i=1}^{|\boldsymbol{w}|} \frac{\lambda \hat{\theta}_{w_i}}{\lambda \hat{\theta}_{w_i} \;+\; (1-\lambda)\dfrac{1}{|\mathcal{W}|}} \end{aligned}$$

Pulling all of these pieces together, Expectation Maximization yields the following iterative algorithm for estimating λ from a heldout corpus $\boldsymbol{h}$:

1. Set λ to an initial guess, say 0.5
2. For $t = 1, 2, \ldots$ until converged:
   (a) Calculate:
   $$\mathrm{E}[\, n_1 \mid \boldsymbol{h}] \;=\; \sum_{i=1}^{|\boldsymbol{h}|} \frac{\lambda \hat{\theta}_{h_i}}{\lambda \hat{\theta}_{h_i} \;+\; (1-\lambda)\dfrac{1}{|\mathcal{W}|}}$$
   (b) Recalculate λ using the new value of $\mathrm{E}[\, n_1|\boldsymbol{h}]$:
   $$\lambda \;=\; \frac{\mathrm{E}[\, n_1 \mid \boldsymbol{h}]}{n_\circ(\boldsymbol{h})}$$

Step (2a) of this algorithm is known as the *E-step* because it involves computing the expected value of the statistics used to estimate the model, while step (2b) is known as the *M-step* because it involves selecting the model parameters that maximize the likelihood with respect to the statistics just computed.

The power behind the EM algorithm comes from its generality; it can often be used to reduce the problem of estimating a model with hidden variables to a sequence of corresponding estimation problems in which all variables are fully observed. It also has a number of important mathematical properties, chief among which is that each iteration is guaranteed not to lower the likelihood of the (visible) training data. If the likelihood is bounded above this means that the EM algorithm will eventually converge, and one can show that when it does converge it converges to a local maximum of the likelihood. (Like most similar algorithms, EM can get "trapped" at a local maximum).