Chapter 1

Language modeling and probability

1.1 Introduction

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

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

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

In fact, probabilistic methods are pervasive in modern computational linguistics, and all the major topics discussed in this book involve proba-

[1] ©Eugene Charniak, Mark Johnson 2013

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

1.2 A Brief Introduction to Probability

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

1.2.1 Outcomes, Events and Probabilities

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

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

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

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

We can formalize the distinction between types and tokens by using the notion of random *events*. 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.

Example 1.1: 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 that $\mathrm{P}(x) = 0.001$ for all samples $x \in \Omega$. Then $\mathrm{P}(E_{\text{‘}\mathit{the}\text{’}}) = 60 \times 0.0001 = 0.06$.

1.2.2 Random Variables and Joint Probabilities

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

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

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

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

Returning to our type-token example, $\mathrm{P}(W{=}\text{‘}\mathit{the}\text{’}) = 0.06$. If the random variable intended is clear from the context, sometimes we elide it and just write its value, e.g., $\mathrm{P}(\text{‘}\mathit{the}\text{’})$ 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 in it and S is the random variable mapping samples to their second letters (or the space character if there is no second letter). Then $\mathrm{P}(F{=}\text{‘}t\text{’})$ is the probability of the event in which the first letter is ‘t’ and $\mathrm{P}(S{=}\text{‘}h\text{’})$ is the probability of the event in which the second letter is ‘h’.

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

1.2.3 Conditional and marginal probabilities

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

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

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

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

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

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

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

$$\begin{aligned}
\mathrm{P}(F\text{=`}n\text{'}) &= 3/10 \\
\mathrm{P}(F\text{=`}t\text{'}) &= 7/10 \\[1em]
\mathrm{P}(S\text{=`}a\text{'}) &= 4/10 \\
\mathrm{P}(S\text{=`}o\text{'}) &= 6/10
\end{aligned}$$

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

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

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

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

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

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

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

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

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

1.2.4 Independence

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

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

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

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

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

which captures the informal definition of independence above.

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

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

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

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

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

1.2.5 Expectations of random variable

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

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

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

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

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

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

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

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

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

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

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

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

1.3 Modeling documents with unigrams

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

In cases like this, where we are going to use corpora (one corpus, two corpora) as examples to help us solve a problem, we call them the *training data* (or training corpus). The unknown-language document will be (part of) the *testing data.* If we want our results to be indicative of what we would see in a real life application, it is important that we collect the testing data separately from the training data. That is, if we are using a several months of newspaper articles for training, we would pick several different months of testing. We often talk about testing on "unseen" data, by which we mean that our program was not built arround the exact examples we use to test it.

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

1.3.1 Documents as sequences of words

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

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

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

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

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

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

1.3.2 Language models as models of possible documents

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

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

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

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

1.3.3 Unigram language models

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

A *unigram language model* assumes that each word W_i is generated independently of the other words. (The name "unigram" comes from the fact that the model's basic units are single words; in section 1.4.1 on page 24 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 $\mathrm{P}(W_i)$. A model that can be understood as generating their data in this way is called a *generative model*. The "story" of how we generate a document by first picking the length, etc., is called a *generative story* about how the document could have been created.

A unigram model assumes that the probability of a word $\mathrm{P}(W_i)$ does not depend on its position i in the document, i.e., $\mathrm{P}(W_i{=}w) = \mathrm{P}(W_j{=}w)$

for all i, j in $1, \ldots, N$. This means that all words are generated by the same distribution over words, so we only have one distribution to learn. This assumption is clearly false (why?), but it does make the unigram model simple enough to be estimated from a modest amount of data.

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

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

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

1.3.4 Maximum likelihood estimates of unigram parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.3.5 Sparse-data problems and smoothing

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

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

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

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

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

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

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

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

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

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

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

Suppose we set $\alpha = 1$ and we have $|\mathcal{W}| = 100,000$ and $n_\circ(\boldsymbol{d}) = 10^7$. As in Example 1.5, the two words '*the*' and '*equilateral*' have counts $2{\cdot}10^5$ and 2, respectively.

Their maximum likelihood estimates again are 0.02 and $2{\cdot}10^{-7}$. After smoothing, the estimate for '*the*' hardly changes

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

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

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

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

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

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

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

1.3.6 Estimating the smoothing parameters

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

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

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

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

These are then substituted into our normal likelihood equation.

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

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

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

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

1.4 Contextual dependencies and *n*-grams

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

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

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

Methods have been proposed to capture all these dependencies (and many more), and identifying the important contextual dependencies and coming up with language models that capture them well is still one of the

◁ students eat bananas ◁

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

central topics in computational linguistics. In this section we describe how to capture one of simplest yet most important kinds of contextual dependencies: those that hold between a word and its neighbors. The idea is that we slide a window of width n words over the text, and the overlapping sequences of length n that we see through this window are called *n-grams*. When n=2 the sequences are called *bigrams* and when n=3 the sequences are called *trigrams*. The basic idea is illustrated for n=2 in Figure 1.1.

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

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

1.4.1 Bigram language models

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

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

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

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

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

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

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

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

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

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

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

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

where

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.4.3 Implementing n-gram language models

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

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

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

1.5 Kneser-Ney Smoothing

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

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

In KN we change this to

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1. $n_{w_{i-1},w_i} += 1$

2. $n_{w_i} += 1$

3. $n_{\circ} += 1$

Here is the function for KN smoothing:

1. If $n_{w_{i-1},w_i} = 0$

   (a) $k_{w_i} += 1$

   (b) $k_{\circ} += 1$

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

3. $n_{w_i} += 1$

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

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

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

1.6 The Noisy Channel Model

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

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

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

We now make the following transformations on this term:

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

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

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

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

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

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

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

1.7 Exercises

Exercise 1.1: In what follows w and w' are word types and $\boldsymbol{d}$ is a end-of-sentence padded corpus. Assume that there is a stop character at the beginning and end of the corpus as well as between each sentence. True or false:

1. If $n_w(\boldsymbol{d}) = 1$ then there exists exactly one w' such that $n_{w,w'} = 1$.

2. If there is exactly one w, w' pair such that $n_{w,w'} = 1$, then $n_w(\boldsymbol{d}) = 1$.

3. Under an unsmoothed unigram model with parameters $\boldsymbol{\theta}$ trained from a corpus $\boldsymbol{c}$, if $L_{\boldsymbol{d}}(\boldsymbol{\theta}) = 0$ then there must be a w such that $n_w(\boldsymbol{d}) > 0$ and $n_w(\boldsymbol{c}) = 0$.

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

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

Assuming each sentence is padded with the stop symbol on both sides and ignoring capitalization, compute the smoothed bigram probability of "The mouse ate the potato" using the smoothing defined in equation 1.11 (additive smoothing) and the KN smoothing. Use $\alpha = 1$ and $\beta = 10$. Show how you compute the smoothed bigram probabilities for all five words.

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

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

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

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

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

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

1. Show that if $1 \geq \lambda \geq 0$, $\sum_{w'} \widetilde{\Theta}_{w,w'} = 1$ for all w.

2. As with α, we can set λ by maximizing the likelihood of held-out data. As the amount of training data goes up, would we expect λ to increase or decrease?

1.8 Programming problem

Problem 1.1: Distinguishing good from bad English

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

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

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

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

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

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

5. Set the bigram smoothing parameter β using the alpha value as described in step 2, and repeat both evaluations. What values of β

maximize the likelihood of the held-out data?

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

1.9 Further Reading

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

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

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

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

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

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

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