CHAPTER

3 N-gram Language Models

"You are uniformly charming!" cried he, with a smile of associating and now and then I bowed and they perceived a chaise and four to wish for.

Random sentence generated from a Jane Austen trigram model

Predicting is difficult—especially about the future, as the old quip goes. But how about predicting something that seems much easier, like the next few words someone is going to say? What word, for example, is likely to follow

```
Please turn your homework ...
```

Hopefully, most of you concluded that a very likely word is *in*, or possibly *over*, but probably not *refrigerator* or *the*. In the following sections we will formalize this intuition by introducing models that assign a **probability** to each possible next word. The same models will also serve to assign a probability to an entire sentence. Such a model, for example, could predict that the following sequence has a much higher probability of appearing in a text:

```
all of a sudden I notice three guys standing on the sidewalk
```

than does this same set of words in a different order:

```
on guys all I of notice sidewalk three a sudden standing the
```

Why would you want to predict upcoming words, or assign probabilities to sentences? Probabilities are essential in any task in which we have to identify words in noisy, ambiguous input, like **speech recognition**. For a speech recognizer to realize that you said *I will be back soonish* and not *I will be bassoon dish*, it helps to know that *back soonish* is a much more probable sequence than *bassoon dish*. For writing tools like **spelling correction** or **grammatical error correction**, we need to find and correct errors in writing like *Their are two midterms*, in which *There* was mistyped as *Their*, or *Everything has improve*, in which *improve* should have been *improved*. The phrase *There are* will be much more probable than *Their are*, and *has improved* than *has improve*, allowing us to help users by detecting and correcting these errors.

Assigning probabilities to sequences of words is also essential in **machine translation**. Suppose we are translating a Chinese source sentence:

他	向	记者	介绍了	主要	内容
He	to	reporters	introduced	main	content

As part of the process we might have built the following set of potential rough English translations:

he introduced reporters to the main contents of the statement
he briefed to reporters the main contents of the statement
he briefed reporters on the main contents of the statement

A probabilistic model of word sequences could suggest that *briefed reporters on* is a more probable English phrase than *briefed to reporters* (which has an awkward *to* after *briefed*) or *introduced reporters to* (which uses a verb that is less fluent English in this context), allowing us to correctly select the boldfaced sentence above.

AAC

Probabilities are also important for **augmentative and alternative communication** systems (Trnka et al. 2007, Kane et al. 2017). People often use such **AAC** devices if they are physically unable to speak or sign but can instead use eye gaze or other specific movements to select words from a menu to be spoken by the system. Word prediction can be used to suggest likely words for the menu.

language model
LM
n-gram

Models that assign probabilities to sequences of words are called **language models** or **LMs**. In this chapter we introduce the simplest model that assigns probabilities to sentences and sequences of words, the **n-gram**. An n-gram is a sequence of n words: a 2-gram (which we'll call **bigram**) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (a **trigram**) is a three-word sequence of words like "please turn your", or "turn your homework". We'll see how to use n-gram models to estimate the probability of the last word of an n-gram given the previous words, and also to assign probabilities to entire sequences. In a bit of terminological ambiguity, we usually drop the word "model", and use the term **n-gram** (and *bigram*, etc.) to mean either the word sequence itself or the predictive model that assigns it a probability. While n-gram models are much simpler than state-of-the art neural language models based on the RNNs and transformers we will introduce in Chapter 9, they are an important foundational tool for understanding the fundamental concepts of language modeling.

3.1 N-Grams

Let's begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is "*its water is so transparent that*" and we want to know the probability that the next word is *the*:

$$P(the|its\ water\ is\ so\ transparent\ that). \tag{3.1}$$

One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see *its water is so transparent that*, and count the number of times this is followed by *the*. This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(the|its\ water\ is\ so\ transparent\ that) = \frac{C(its\ water\ is\ so\ transparent\ that\ the)}{C(its\ water\ is\ so\ transparent\ that)} \tag{3.2}$$

With a large enough corpus, such as the web, we can compute these counts and estimate the probability from Eq. 3.2. You should pause now, go to the web, and compute this estimate for yourself.

While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions of the example sentence may have counts of zero on the web (such as "*Walden Pond's water is so transparent that the*"; well, *used to* have counts of zero).

Similarly, if we wanted to know the joint probability of an entire sequence of words like *its water is so transparent*, we could do it by asking "out of all possible sequences of five words, how many of them are *its water is so transparent*?" We would have to get the count of *its water is so transparent* and divide by the sum of the counts of all possible five word sequences. That seems rather a lot to estimate!

For this reason, we'll need to introduce more clever ways of estimating the probability of a word w given a history h, or the probability of an entire word sequence W. Let's start with a little formalizing of notation. To represent the probability of a particular random variable X_i taking on the value "the", or $P(X_i = \text{"the"})$, we will use the simplification $P(the)$. We'll represent a sequence of n words either as $w_1 \dots w_n$ or $w_{1:n}$ (so the expression $w_{1:n-1}$ means the string $w_1, w_2, ..., w_{n-1}$). For the joint probability of each word in a sequence having a particular value $P(X_1 = w_1, X_2 = w_2, X_3 = w_3, ..., X_n = w_n)$ we'll use $P(w_1, w_2, ..., w_n)$.

Now, how can we compute probabilities of entire sequences like $P(w_1, w_2, ..., w_n)$? One thing we can do is decompose this probability using the **chain rule of probability**:

$$
\begin{aligned}
P(X_1 ... X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_{1:2}) \dots P(X_n|X_{1:n-1}) \\
&= \prod_{k=1}^{n} P(X_k|X_{1:k-1})
\end{aligned}
\tag{3.3}
$$

Applying the chain rule to words, we get

$$
\begin{aligned}
P(w_{1:n}) &= P(w_1)P(w_2|w_1)P(w_3|w_{1:2}) \dots P(w_n|w_{1:n-1}) \\
&= \prod_{k=1}^{n} P(w_k|w_{1:k-1})
\end{aligned}
\tag{3.4}
$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. Equation 3.4 suggests that we could estimate the joint probability of an entire sequence of words by multiplying together a number of conditional probabilities. But using the chain rule doesn't really seem to help us! We don't know any way to compute the exact probability of a word given a long sequence of preceding words, $P(w_n|w_{1:n-1})$. As we said above, we can't just estimate by counting the number of times every word occurs following every long string, because language is creative and any particular context might have never occurred before!

The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can **approximate** the history by just the last few words.

bigram

The **bigram** model, for example, approximates the probability of a word given all the previous words $P(w_n|w_{1:n-1})$ by using only the conditional probability of the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

$$P(\text{the}|\text{Walden Pond's water is so transparent that}) \tag{3.5}$$

we approximate it with the probability

$$P(\text{the}|\text{that}) \tag{3.6}$$

When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1}) \tag{3.7}$$

The assumption that the probability of a word depends only on the previous word is called a **Markov** assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the **n-gram** (which looks $n-1$ words into the past).

Markov

n-gram

Let's see a general equation for this n-gram approximation to the conditional probability of the next word in a sequence. We'll use N here to mean the n-gram size, so $N = 2$ means bigrams and $N = 3$ means trigrams. Then we approximate the probability of a word given its entire context as follows:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) \tag{3.8}$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by substituting Eq. 3.7 into Eq. 3.4:

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{3.9}$$

How do we estimate these bigram or n-gram probabilities? An intuitive way to estimate probabilities is called **maximum likelihood estimation** or **MLE**. We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and **normalizing** the counts so that they lie between 0 and 1.[1]

maximum likelihood estimation

normalize

For example, to compute a particular bigram probability of a word w_n given a previous word w_{n-1}, we'll compute the count of the bigram $C(w_{n-1}w_n)$ and normalize by the sum of all the bigrams that share the same first word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \tag{3.10}$$

We can simplify this equation, since the sum of all bigram counts that start with a given word w_{n-1} must be equal to the unigram count for that word w_{n-1} (the reader should take a moment to be convinced of this):

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.11}$$

Let's work through an example using a mini-corpus of three sentences. We'll first need to augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word. We'll also need a special end-symbol. </s>[2]

```
 <s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>
```

1 For probabilistic models, normalizing means dividing by some total count so that the resulting probabilities fall between 0 and 1.

2 We need the end-symbol to make the bigram grammar a true probability distribution. Without an end-symbol, instead of the sentence probabilities of all sentences summing to one, the sentence probabilities for all sentences *of a given length* would sum to one. This model would define an infinite set of probability distributions, with one distribution per sentence length. See Exercise 3.5.

Here are the calculations for some of the bigram probabilities from this corpus

$P(\texttt{I}|\texttt{<s>}) = \frac{2}{3} = .67 \qquad P(\texttt{Sam}|\texttt{<s>}) = \frac{1}{3} = .33 \qquad P(\texttt{am}|\texttt{I}) = \frac{2}{3} = .67$

$P(\texttt{</s>}|\texttt{Sam}) = \frac{1}{2} = 0.5 \qquad P(\texttt{Sam}|\texttt{am}) = \frac{1}{2} = .5 \qquad P(\texttt{do}|\texttt{I}) = \frac{1}{3} = .33$

For the general case of MLE n-gram parameter estimation:

$$P(w_n|w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1}\, w_n)}{C(w_{n-N+1:n-1})} \tag{3.12}$$

relative frequency

Equation 3.12 (like Eq. 3.11) estimates the n-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a **relative frequency**. We said above that this use of relative frequencies as a way to estimate probabilities is an example of maximum likelihood estimation or MLE. In MLE, the resulting parameter set maximizes the likelihood of the training set T given the model M (i.e., $P(T|M)$). For example, suppose the word *Chinese* occurs 400 times in a corpus of a million words like the Brown corpus. What is the probability that a random word selected from some other text of, say, a million words will be the word *Chinese*? The MLE of its probability is $\frac{400}{1000000}$ or .0004. Now .0004 is not the best possible estimate of the probability of *Chinese* occurring in all situations; it might turn out that in some other corpus or context *Chinese* is a very unlikely word. But it is the probability that makes it *most likely* that Chinese will occur 400 times in a million-word corpus. We present ways to modify the MLE estimates slightly to get better probability estimates in Section 3.5.

Let's move on to some examples from a slightly larger corpus than our 14-word example above. We'll use data from the now-defunct Berkeley Restaurant Project, a dialogue system from the last century that answered questions about a database of restaurants in Berkeley, California (Jurafsky et al., 1994). Here are some text-normalized sample user queries (a sample of 9332 sentences is on the website):

can you tell me about any good cantonese restaurants close by
mid priced thai food is what i'm looking for
tell me about chez panisse
can you give me a listing of the kinds of food that are available
i'm looking for a good place to eat breakfast
when is caffe venezia open during the day

Figure 3.1 shows the bigram counts from a piece of a bigram grammar from the Berkeley Restaurant Project. Note that the majority of the values are zero. In fact, we have chosen the sample words to cohere with each other; a matrix selected from a random set of eight words would be even more sparse.

	i	**want**	**to**	**eat**	**chinese**	**food**	**lunch**	**spend**
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Figure 3.1 Bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Zero counts are in gray.

Figure 3.2 shows the bigram probabilities after normalization (dividing each cell in Fig. 3.1 by the appropriate unigram for its row, taken from the following set of unigram probabilities):

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Figure 3.2 Bigram probabilities for eight words in the Berkeley Restaurant Project corpus of 9332 sentences. Zero probabilities are in gray.

Here are a few other useful probabilities:

$P(\texttt{i}|\texttt{<s>}) = 0.25$ $\quad$ $P(\texttt{english}|\texttt{want}) = 0.0011$
$P(\texttt{food}|\texttt{english}) = 0.5$ $\quad$ $P(\texttt{</s>}|\texttt{food}) = 0.68$

Now we can compute the probability of sentences like *I want English food* or *I want Chinese food* by simply multiplying the appropriate bigram probabilities together, as follows:

$$\begin{aligned} & P(\texttt{<s> i want english food </s>}) \\ &= P(\texttt{i}|\texttt{<s>})P(\texttt{want}|\texttt{i})P(\texttt{english}|\texttt{want}) \\ &\qquad P(\texttt{food}|\texttt{english})P(\texttt{</s>}|\texttt{food}) \\ &= .25 \times .33 \times .0011 \times 0.5 \times 0.68 \\ &= .000031 \end{aligned}$$

We leave it as Exercise 3.2 to compute the probability of *i want chinese food*.

What kinds of linguistic phenomena are captured in these bigram statistics? Some of the bigram probabilities above encode some facts that we think of as strictly **syntactic** in nature, like the fact that what comes after *eat* is usually a noun or an adjective, or that what comes after *to* is usually a verb. Others might be a fact about the personal assistant task, like the high probability of sentences beginning with the words *I*. And some might even be cultural rather than linguistic, like the higher probability that people are looking for Chinese versus English food.

trigram
4-gram
5-gram

Some practical issues: Although for pedagogical purposes we have only described bigram models, in practice it's more common to use **trigram** models, which condition on the previous two words rather than the previous word, or **4-gram** or even **5-gram** models, when there is sufficient training data. Note that for these larger n-grams, we'll need to assume extra contexts to the left and right of the sentence end. For example, to compute trigram probabilities at the very beginning of the sentence, we use two pseudo-words for the first trigram (i.e., $P(\texttt{I}|\texttt{<s><s>})$).

log probabilities

We always represent and compute language model probabilities in log format as **log probabilities**. Since probabilities are (by definition) less than or equal to

1, the more probabilities we multiply together, the smaller the product becomes. Multiplying enough n-grams together would result in numerical underflow. By using log probabilities instead of raw probabilities, we get numbers that are not as small. Adding in log space is equivalent to multiplying in linear space, so we combine log probabilities by adding them. The result of doing all computation and storage in log space is that we only need to convert back into probabilities if we need to report them at the end; then we can just take the exp of the logprob:

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4) \tag{3.13}$$

3.2 Evaluating Language Models

The best way to evaluate the performance of a language model is to embed it in an application and measure how much the application improves. Such end-to-end evaluation is called **extrinsic evaluation**. Extrinsic evaluation is the only way to know if a particular improvement in a component is really going to help the task at hand. Thus, for speech recognition, we can compare the performance of two language models by running the speech recognizer twice, once with each language model, and seeing which gives the more accurate transcription.

extrinsic evaluation

Unfortunately, running big NLP systems end-to-end is often very expensive. Instead, it would be nice to have a metric that can be used to quickly evaluate potential improvements in a language model. An **intrinsic evaluation** metric is one that measures the quality of a model independent of any application.

intrinsic evaluation

For an intrinsic evaluation of a language model we need a **test set**. As with many of the statistical models in our field, the probabilities of an n-gram model come from the corpus it is trained on, the **training set** or **training corpus**. We can then measure the quality of an n-gram model by its performance on some unseen data called the **test set** or test corpus.

training set

test set

So if we are given a corpus of text and want to compare two different n-gram models, we divide the data into training and test sets, train the parameters of both models on the training set, and then compare how well the two trained models fit the test set.

But what does it mean to "fit the test set"? The answer is simple: whichever model assigns a **higher probability** to the test set—meaning it more accurately predicts the test set—is a better model. Given two probabilistic models, the better model is the one that has a tighter fit to the test data or that better predicts the details of the test data, and hence will assign a higher probability to the test data.

Since our evaluation metric is based on test set probability, it's important not to let the test sentences into the training set. Suppose we are trying to compute the probability of a particular "test" sentence. If our test sentence is part of the training corpus, we will mistakenly assign it an artificially high probability when it occurs in the test set. We call this situation **training on the test set**. Training on the test set introduces a bias that makes the probabilities all look too high, and causes huge inaccuracies in **perplexity**, the probability-based metric we introduce below.

Sometimes we use a particular test set so often that we implicitly tune to its characteristics. We then need a fresh test set that is truly unseen. In such cases, we call the initial test set the **development** test set or, **devset**. How do we divide our data into training, development, and test sets? We want our test set to be as large as possible, since a small test set may be accidentally unrepresentative, but we also

development test

want as much training data as possible. At the minimum, we would want to pick the smallest test set that gives us enough statistical power to measure a statistically significant difference between two potential models. In practice, we often just divide our data into 80% training, 10% development, and 10% test. Given a large corpus that we want to divide into training and test, test data can either be taken from some continuous sequence of text inside the corpus, or we can remove smaller "stripes" of text from randomly selected parts of our corpus and combine them into a test set.

3.2.1 Perplexity

perplexity

In practice we don't use raw probability as our metric for evaluating language models, but a variant called **perplexity**. The **perplexity** (sometimes called *PPL* for short) of a language model on a test set is the inverse probability of the test set, normalized by the number of words. For a test set $W = w_1 w_2 \ldots w_N$,:

$$\begin{aligned} \text{perplexity}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}} \end{aligned} \tag{3.14}$$

We can use the chain rule to expand the probability of W:

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}} \tag{3.15}$$

The perplexity of a test set W depends on which language model we use. Here's the perplexity of W with a unigram language model (just the geometric mean of the unigram probabilities):

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i)}} \tag{3.16}$$

The perplexity of W computed with a bigram language model is still a geometric mean, but now of the bigram probabilities:

$$\text{perplexity}(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}} \tag{3.17}$$

Note that because of the inverse in Eq. 3.15, the higher the conditional probability of the word sequence, the lower the perplexity. Thus, minimizing perplexity is equivalent to maximizing the test set probability according to the language model. What we generally use for word sequence in Eq. 3.15 or Eq. 3.17 is the entire sequence of words in some test set. Since this sequence will cross many sentence boundaries, we need to include the begin- and end-sentence markers <s> and </s> in the probability computation. We also need to include the end-of-sentence marker </s> (but not the beginning-of-sentence marker <s>) in the total count of word tokens N.

There is another way to think about perplexity: as the **weighted average branching factor** of a language. The branching factor of a language is the number of possible next words that can follow any word. Consider the task of recognizing the digits in English (zero, one, two,..., nine), given that (both in some training set and in some test set) each of the 10 digits occurs with equal probability $P = \frac{1}{10}$. The perplexity of this mini-language is in fact 10. To see that, imagine a test string of digits of length N, and assume that in the training set all the digits occurred with equal probability. By Eq. 3.15, the perplexity will be

$$\begin{aligned} \text{perplexity}(W) &= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\ &= (\frac{1}{10}^N)^{-\frac{1}{N}} \\ &= \frac{1}{10}^{-1} \\ &= 10 \end{aligned} \tag{3.18}$$

But suppose that the number zero is really frequent and occurs far more often than other numbers. Let's say that 0 occur 91 times in the training set, and each of the other digits occurred 1 time each. Now we see the following test set: 0 0 0 0 0 3 0 0 0 0. We should expect the perplexity of this test set to be lower since most of the time the next number will be zero, which is very predictable, i.e. has a high probability. Thus, although the branching factor is still 10, the perplexity or *weighted* branching factor is smaller. We leave this exact calculation as exercise 3.12.

We see in Section 3.8 that perplexity is also closely related to the information-theoretic notion of entropy.

We mentioned above that perplexity is a function of both the text and the language model: given a text W, different language models will have different perplexities. Because of this, perplexity can be used to compare different n-gram models. Let's look at an example, in which we trained unigram, bigram, and trigram grammars on 38 million words (including start-of-sentence tokens) from the *Wall Street Journal*, using a 19,979 word vocabulary. We then computed the perplexity of each of these models on a test set of 1.5 million words, using Eq. 3.16 for unigrams, Eq. 3.17 for bigrams, and the corresponding equation for trigrams. The table below shows the perplexity of a 1.5 million word WSJ test set according to each of these grammars.

	Unigram	**Bigram**	**Trigram**
Perplexity	962	170	109

As we see above, the more information the n-gram gives us about the word sequence, the higher the probability the n-gram will assign to the string. A trigram model is less surprised than a unigram model because it has a better idea of what words might come next, and so it assigns them a higher probability. And the higher the probability, the lower the perplexity (since as Eq. 3.15 showed, perplexity is related inversely to the likelihood of the test sequence according to the model). So a lower perplexity can tell us that a language model is a better predictor of the words in the test set.

Note that in computing perplexities, the n-gram model P must be constructed without any knowledge of the test set or any prior knowledge of the vocabulary of the test set. Any kind of knowledge of the test set can cause the perplexity to be artificially low. The perplexity of two language models is only comparable if they use identical vocabularies.

An (intrinsic) improvement in perplexity does not guarantee an (extrinsic) improvement in the performance of a language processing task like speech recognition or machine translation. Nonetheless, because perplexity often correlates with such improvements, it is commonly used as a quick check on an algorithm. But a model's improvement in perplexity should always be confirmed by an end-to-end evaluation of a real task before concluding the evaluation of the model.

3.3 Sampling sentences from a language model

sampling

One important way to visualize what kind of knowledge a language model embodies is to sample from it. **Sampling** from a distribution means to choose random points according to their likelihood. Thus sampling from a language model—which represents a distribution over sentences—means to generate some sentences, choosing each sentence according to its likelihood as defined by the model. Thus we are more likely to generate sentences that the model thinks have a high probability and less likely to generate sentences that the model thinks have a low probability.

This technique of visualizing a language model by sampling was first suggested very early on by Shannon (1951) and Miller and Selfridge (1950). It's simplest to visualize how this works for the unigram case. Imagine all the words of the English language covering the probability space between 0 and 1, each word covering an interval proportional to its frequency. Fig. 3.3 shows a visualization, using a unigram LM computed from the text of this book. We choose a random value between 0 and 1, find that point on the probability line, and print the word whose interval includes this chosen value. We continue choosing random numbers and generating words until we randomly generate the sentence-final token `</s>`.

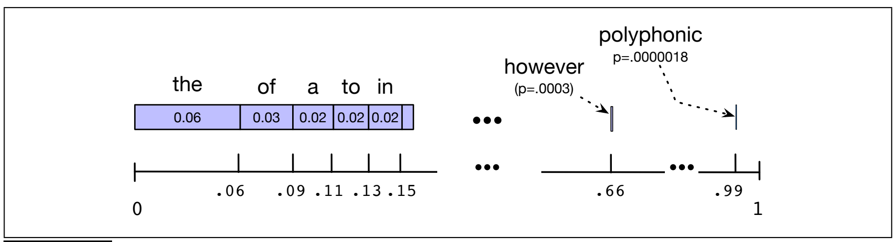

Figure 3.3 A visualization of the sampling distribution for sampling sentences by repeatedly sampling unigrams. The blue bar represents the relative frequency of each word (we've ordered them from most frequent to least frequent, but the choice of order is arbitrary). The number line shows the cumulative probabilities. If we choose a random number between 0 and 1, it will fall in an interval corresponding to some word. The expectation for the random number to fall in the larger intervals of one of the frequent words (*the*, *of*, *a*) is much higher than in the smaller interval of one of the rare words (*polyphonic*).

We can use the same technique to generate bigrams by first generating a random bigram that starts with `<s>` (according to its bigram probability). Let's say the second word of that bigram is w. We next choose a random bigram starting with w (again, drawn according to its bigram probability), and so on.

3.4 Generalization and Zeros

The n-gram model, like many statistical models, is dependent on the training corpus. One implication of this is that the probabilities often encode specific facts about a given training corpus. Another implication is that n-grams do a better and better job of modeling the training corpus as we increase the value of N.

We can use the sampling method from the prior section to visualize both of these facts! To give an intuition for the increasing power of higher-order n-grams, Fig. 3.4 shows random sentences generated from unigram, bigram, trigram, and 4-gram models trained on Shakespeare's works.

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Figure 3.4 Eight sentences randomly generated from four n-grams computed from Shakespeare's works. All characters were mapped to lower-case and punctuation marks were treated as words. Output is hand-corrected for capitalization to improve readability.

The longer the context on which we train the model, the more coherent the sentences. In the unigram sentences, there is no coherent relation between words or any sentence-final punctuation. The bigram sentences have some local word-to-word coherence (especially if we consider that punctuation counts as a word). The trigram and 4-gram sentences are beginning to look a lot like Shakespeare. Indeed, a careful investigation of the 4-gram sentences shows that they look a little too much like Shakespeare. The words *It cannot be but so* are directly from *King John*. This is because, not to put the knock on Shakespeare, his oeuvre is not very large as corpora go ($N = 884,647, V = 29,066$), and our n-gram probability matrices are ridiculously sparse. There are $V^2 = 844,000,000$ possible bigrams alone, and the number of possible 4-grams is $V^4 = 7 \times 10^{17}$. Thus, once the generator has chosen the first 4-gram (*It cannot be but*), there are only five possible continuations (*that*, *I*, *he*, *thou*, and *so*); indeed, for many 4-grams, there is only one continuation.

To get an idea of the dependence of a grammar on its training set, let's look at an n-gram grammar trained on a completely different corpus: the *Wall Street Journal* (WSJ) newspaper. Shakespeare and the *Wall Street Journal* are both English, so we might expect some overlap between our n-grams for the two genres. Fig. 3.5 shows sentences generated by unigram, bigram, and trigram grammars trained on 40 million words from WSJ.

Compare these examples to the pseudo-Shakespeare in Fig. 3.4. While they both model "English-like sentences", there is clearly no overlap in generated sentences, and little overlap even in small phrases. Statistical models are likely to be pretty useless as predictors if the training sets and the test sets are as different as Shakespeare and WSJ.

How should we deal with this problem when we build n-gram models? One step is to be sure to use a training corpus that has a similar **genre** to whatever task we are trying to accomplish. To build a language model for translating legal documents,

1 gram	Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives
2 gram	Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her
3 gram	They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

Figure 3.5 Three sentences randomly generated from three n-gram models computed from 40 million words of the *Wall Street Journal*, lower-casing all characters and treating punctuation as words. Output was then hand-corrected for capitalization to improve readability.

we need a training corpus of legal documents. To build a language model for a question-answering system, we need a training corpus of questions.

It is equally important to get training data in the appropriate **dialect** or **variety**, especially when processing social media posts or spoken transcripts. For example some tweets will use features of African American Language (AAL)— the name for the many variations of language used in African American communities (King, 2020). Such features include words like *finna*—an auxiliary verb that marks immediate future tense —that don't occur in other varieties, or spellings like *den* for *then*, in tweets like this one (Blodgett and O'Connor, 2017):

(3.19) Bored af den my phone finna die!!!

while tweets from varieties like Nigerian English have markedly different vocabulary and n-gram patterns from American English (Jurgens et al., 2017):

(3.20) @username R u a wizard or wat gan sef: in d mornin - u tweet, afternoon - u tweet, nyt gan u dey tweet. beta get ur IT placement wiv twitter

Matching genres and dialects is still not sufficient. Our models may still be subject to the problem of **sparsity**. For any n-gram that occurred a sufficient number of times, we might have a good estimate of its probability. But because any corpus is limited, some perfectly acceptable English word sequences are bound to be missing from it. That is, we'll have many cases of putative "zero probability n-grams" that should really have some non-zero probability. Consider the words that follow the bigram *denied the* in the WSJ Treebank3 corpus, together with their counts:

denied the allegations:	5
denied the speculation:	2
denied the rumors:	1
denied the report:	1

But suppose our test set has phrases like:

denied the offer
denied the loan

Our model will incorrectly estimate that the $P(\text{offer}|\text{denied the})$ is 0!

zeros

These **zeros**—things that don't ever occur in the training set but do occur in the test set—are a problem for two reasons. First, their presence means we are underestimating the probability of all sorts of words that might occur, which will hurt the performance of any application we want to run on this data.

Second, if the probability of any word in the test set is 0, the entire probability of the test set is 0. By definition, perplexity is based on the inverse probability of the test set. Thus if some words have zero probability, we can't compute perplexity at all, since we can't divide by 0!

What do we do about zeros? There are two solutions, depending on the kind of zero. For words whose n-gram probability is zero because they occur in a novel test set context, like the example of *denied the offer* above, we'll introduce in Section 3.5 algorithms called **smoothing** or discounting. Smoothing algorithms shave off a bit of probability mass from some more frequent events and give it to these unseen events. But first, let's talk about an even more insidious form of zero: words that the model has never seen below at all (in any context): unknown words!

3.4.1 Unknown Words

What do we do about words we have never seen before? Perhaps the word *Jurafsky* simply did not occur in our training set, but pops up in the test set!

closed vocabulary

We can choose to disallow this situation from occurring, by stipulating that we already know all the words that can occur. In such a **closed vocabulary** system the test set can only contain words from this known lexicon, and there will be no unknown words.

OOV

open vocabulary

In most real situations, however, we have to deal with words we haven't seen before, which we'll call **unknown** words, or **out of vocabulary** (**OOV**) words. The percentage of OOV words that appear in the test set is called the **OOV rate**. One way to create an **open vocabulary** system is to model these potential unknown words in the test set by adding a pseudo-word called <UNK>.

There are two common ways to train the probabilities of the unknown word model <UNK>. The first one is to turn the problem back into a closed vocabulary one by choosing a fixed vocabulary in advance:

1. **Choose a vocabulary** (word list) that is fixed in advance.
2. **Convert** in the training set any word that is not in this set (any OOV word) to the unknown word token <UNK> in a text normalization step.
3. **Estimate** the probabilities for <UNK> from its counts just like any other regular word in the training set.

The second alternative, in situations where we don't have a prior vocabulary in advance, is to create such a vocabulary implicitly, replacing words in the training data by <UNK> based on their frequency. For example we can replace by <UNK> all words that occur fewer than n times in the training set, where n is some small number, or equivalently select a vocabulary size V in advance (say 50,000) and choose the top V words by frequency and replace the rest by <UNK>. In either case we then proceed to train the language model as before, treating <UNK> like a regular word.

The exact choice of <UNK> has an effect on metrics like perplexity. A language model can achieve low perplexity by choosing a small vocabulary and assigning the unknown word a high probability. Thus perplexities can only be compared across language models with the same vocabularies (Buck et al., 2014).

3.5 Smoothing

What do we do with words that are in our vocabulary (they are not unknown words) but appear in a test set in an unseen context (for example they appear after a word they never appeared after in training)? To keep a language model from assigning zero probability to these unseen events, we'll have to shave off a bit of probability mass from some more frequent events and give it to the events we've never seen. This modification is called **smoothing** or **discounting**. In this section and the following ones we'll introduce a variety of ways to do smoothing: **Laplace (add-one) smoothing**, **add-k smoothing**, **stupid backoff**, and **Kneser-Ney smoothing**.

smoothing

discounting

3.5.1 Laplace Smoothing

The simplest way to do smoothing is to add one to all the n-gram counts, before we normalize them into probabilities. All the counts that used to be zero will now have a count of 1, the counts of 1 will be 2, and so on. This algorithm is called **Laplace smoothing**. Laplace smoothing does not perform well enough to be used in modern n-gram models, but it usefully introduces many of the concepts that we see in other smoothing algorithms, gives a useful baseline, and is also a practical smoothing algorithm for other tasks like **text classification** (Chapter 4).

Laplace smoothing

Let's start with the application of Laplace smoothing to unigram probabilities. Recall that the unsmoothed maximum likelihood estimate of the unigram probability of the word w_i is its count c_i normalized by the total number of word tokens N:

$$P(w_i) = \frac{c_i}{N}$$

Laplace smoothing merely adds one to each count (hence its alternate name **add-one** smoothing). Since there are V words in the vocabulary and each one was incremented, we also need to adjust the denominator to take into account the extra V observations. (What happens to our P values if we don't increase the denominator?)

add-one

$$P_{\text{Laplace}}(w_i) = \frac{c_i + 1}{N + V} \tag{3.21}$$

Instead of changing both the numerator and denominator, it is convenient to describe how a smoothing algorithm affects the numerator, by defining an **adjusted count** c^*. This adjusted count is easier to compare directly with the MLE counts and can be turned into a probability like an MLE count by normalizing by N. To define this count, since we are only changing the numerator in addition to adding 1 we'll also need to multiply by a normalization factor $\frac{N}{N+V}$:

$$c_i^* = (c_i + 1)\frac{N}{N + V} \tag{3.22}$$

We can now turn c_i^* into a probability P_i^* by normalizing by N.

A related way to view smoothing is as **discounting** (lowering) some non-zero counts in order to get the probability mass that will be assigned to the zero counts. Thus, instead of referring to the discounted counts c^*, we might describe a smoothing algorithm in terms of a relative **discount** d_c, the ratio of the discounted counts to the original counts:

discounting

discount

$$d_c = \frac{c^*}{c}$$

Now that we have the intuition for the unigram case, let's smooth our Berkeley Restaurant Project bigrams. Figure 3.6 shows the add-one smoothed counts for the bigrams in Fig. 3.1.

	i	want	to	eat	chinese	food	lunch	spend
i	6	828	1	10	1	1	1	3
want	3	1	609	2	7	7	6	2
to	3	1	5	687	3	1	7	212
eat	1	1	3	1	17	3	43	1
chinese	2	1	1	1	1	83	2	1
food	16	1	16	1	2	5	1	1
lunch	3	1	1	1	1	2	1	1
spend	2	1	2	1	1	1	1	1

Figure 3.6 Add-one smoothed bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences. Previously-zero counts are in gray.

Figure 3.7 shows the add-one smoothed probabilities for the bigrams in Fig. 3.2. Recall that normal bigram probabilities are computed by normalizing each row of counts by the unigram count:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.23}$$

For add-one smoothed bigram counts, we need to augment the unigram count by the number of total word types in the vocabulary V:

$$P_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{\sum_w (C(w_{n-1}w)+1)} = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V} \tag{3.24}$$

Thus, each of the unigram counts given in the previous section will need to be augmented by $V = 1446$. The result is the smoothed bigram probabilities in Fig. 3.7.

	i	want	to	eat	chinese	food	lunch	spend
i	0.0015	0.21	0.00025	0.0025	0.00025	0.00025	0.00025	0.00075
want	0.0013	0.00042	0.26	0.00084	0.0029	0.0029	0.0025	0.00084
to	0.00078	0.00026	0.0013	0.18	0.00078	0.00026	0.0018	0.055
eat	0.00046	0.00046	0.0014	0.00046	0.0078	0.0014	0.02	0.00046
chinese	0.0012	0.00062	0.00062	0.00062	0.00062	0.052	0.0012	0.00062
food	0.0063	0.00039	0.0063	0.00039	0.00079	0.002	0.00039	0.00039
lunch	0.0017	0.00056	0.00056	0.00056	0.00056	0.0011	0.00056	0.00056
spend	0.0012	0.00058	0.0012	0.00058	0.00058	0.00058	0.00058	0.00058

Figure 3.7 Add-one smoothed bigram probabilities for eight of the words (out of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero probabilities are in gray.

It is often convenient to reconstruct the count matrix so we can see how much a smoothing algorithm has changed the original counts. These adjusted counts can be computed by Eq. 3.25. Figure 3.8 shows the reconstructed counts.

$$c^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n)+1] \times C(w_{n-1})}{C(w_{n-1})+V} \tag{3.25}$$

Note that add-one smoothing has made a very big change to the counts. Comparing Fig. 3.8 to the original counts in Fig. 3.1, we can see that $C(\textit{want to})$ changed from 608 to 238! We can see this in probability space as well: $P(\textit{to}|\textit{want})$ decreases from .66 in the unsmoothed case to .26 in the smoothed case. Looking at the discount d (the ratio between new and old counts) shows us how strikingly the counts

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

Figure 3.8 Add-one reconstituted counts for eight words (of $V = 1446$) in the BeRP corpus of 9332 sentences. Previously-zero counts are in gray.

for each prefix word have been reduced; the discount for the bigram *want to* is .39, while the discount for *Chinese food* is .10, a factor of 10!

The sharp change in counts and probabilities occurs because too much probability mass is moved to all the zeros.

3.5.2 Add-k smoothing

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events. Instead of adding 1 to each count, we add a fractional count k (.5? .05? .01?). This algorithm is therefore called **add-k smoothing**.

add-k

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+k}{C(w_{n-1})+kV} \tag{3.26}$$

Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a **devset**. Although add-k is useful for some tasks (including text classification), it turns out that it still doesn't work well for language modeling, generating counts with poor variances and often inappropriate discounts (Gale and Church, 1994).

3.5.3 Backoff and Interpolation

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on. If we are trying to compute $P(w_n|w_{n-2}w_{n-1})$ but we have no examples of a particular trigram $w_{n-2}w_{n-1}w_n$, we can instead estimate its probability by using the bigram probability $P(w_n|w_{n-1})$. Similarly, if we don't have counts to compute $P(w_n|w_{n-1})$, we can look to the unigram $P(w_n)$.

In other words, sometimes using **less context** is a good thing, helping to generalize more for contexts that the model hasn't learned much about. There are two ways to use this n-gram "hierarchy". In **backoff**, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order n-gram. By contrast, in **interpolation**, we always mix the probability estimates from all the n-gram estimators, weighting and combining the trigram, bigram, and unigram counts.

backoff

interpolation

In simple linear interpolation, we combine different order n-grams by linearly interpolating them. Thus, we estimate the trigram probability $P(w_n|w_{n-2}w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a

In a slightly more sophisticated version of linear interpolation, each λ weight is computed by conditioning on the context. This way, if we have particularly accurate counts for a particular bigram, we assume that the counts of the trigrams based on this bigram will be more trustworthy, so we can make the λs for those trigrams higher and thus give that trigram more weight in the interpolation. Equation 3.29 shows the equation for interpolation with context-conditioned weights.

How are these λ values set? Both the simple interpolation and conditional interpolation λs are learned from a **held-out** corpus. A held-out corpus is an additional training corpus, so-called because we hold it out from the training data, that we use to set hyperparameters like these λ values. We do so by choosing the λ values that maximize the likelihood of the held-out corpus. That is, we fix the n-gram probabilities and then search for the λ values that—when plugged into Eq. 3.27—give us the highest probability of the held-out set. There are various ways to find this optimal set of λs. One way is to use the **EM** algorithm, an iterative learning algorithm that converges on locally optimal λs (Jelinek and Mercer, 1980).

held-out

In a **backoff** n-gram model, if the n-gram we need has zero counts, we approximate it by backing off to the (n-1)-gram. We continue backing off until we reach a history that has some counts.

In order for a backoff model to give a correct probability distribution, we have to **discount** the higher-order n-grams to save some probability mass for the lower order n-grams. Just as with add-one smoothing, if the higher-order n-grams aren't discounted and we just used the undiscounted MLE probability, then as soon as we replaced an n-gram which has zero probability with a lower-order n-gram, we would be adding probability mass, and the total probability assigned to all possible strings by the language model would be greater than 1! In addition to this explicit discount factor, we'll need a function α to distribute this probability mass to the lower order n-grams.

discount

This kind of backoff with discounting is also called **Katz backoff**. In Katz backoff we rely on a discounted probability P^* if we've seen this n-gram before (i.e., if we have non-zero counts). Otherwise, we recursively back off to the Katz probability for the shorter-history (n-1)-gram. The probability for a backoff n-gram P_{BO} is thus computed as follows.

Katz backoff

Good-Turing

Katz backoff is often combined with a smoothing method called **Good-Turing**. The combined **Good-Turing backoff** algorithm involves quite detailed computation for estimating the Good-Turing smoothing and the P^* and α values.

3.6 Huge Language Models and Stupid Backoff

By using text from the web or other enormous collections, it is possible to build extremely large language models. The Web 1 Trillion 5-gram corpus released by Google includes various large sets of n-grams, including 1-grams through 5-grams from all the five-word sequences that appear in at least 40 distinct books from 1,024,908,267,229 words of text from publicly accessible Web pages in English (Franz and Brants, 2006). Google has also released Google Books Ngrams corpora with n-grams drawn from their book collections, including another 800 billion tokens of n-grams from Chinese, English, French, German, Hebrew, Italian, Russian, and Spanish (Lin et al., 2012). Smaller but more carefully curated n-gram corpora for English include the million most frequent n-grams drawn from the COCA (Corpus of Contemporary American English) 1 billion word corpus of American English (Davies, 2020). COCA is a balanced corpus, meaning that it has roughly equal numbers of words from different genres: web, newspapers, spoken conversation transcripts, fiction, and so on, drawn from the period 1990-2019, and has the context of each n-gram as well as labels for genre and provenance.

Some example 4-grams from the Google Web corpus:

4-gram	Count
serve as the incoming	92
serve as the incubator	99
serve as the independent	794
serve as the index	223
serve as the indication	72
serve as the indicator	120
serve as the indicators	45

Efficiency considerations are important when building language models that use such large sets of n-grams. Rather than store each word as a string, it is generally represented in memory as a 64-bit hash number, with the words themselves stored on disk. Probabilities are generally quantized using only 4-8 bits (instead of 8-byte floats), and n-grams are stored in reverse tries.

An n-gram language model can also be shrunk by pruning, for example only storing n-grams with counts greater than some threshold (such as the count threshold of 40 used for the Google n-gram release) or using entropy to prune less-important n-grams (Stolcke, 1998). Another option is to build approximate language models using techniques like **Bloom filters** (Talbot and Osborne 2007, Church et al. 2007). Finally, efficient language model toolkits like KenLM (Heafield 2011, Heafield et al. 2013) use sorted arrays, efficiently combine probabilities and backoffs in a single value, and use merge sorts to efficiently build the probability tables in a minimal number of passes through a large corpus.

Bloom filters

Although with these toolkits it is possible to build web-scale language models using advanced smoothing algorithms like the Kneser-Ney algorithm we will see in Section 3.7, Brants et al. (2007) show that with very large language models a much simpler algorithm may be sufficient. The algorithm is called **stupid backoff**. Stupid backoff gives up the idea of trying to make the language model a true probability dis-

stupid backoff

tribution. There is no discounting of the higher-order probabilities. If a higher-order n-gram has a zero count, we simply backoff to a lower order n-gram, weighed by a fixed (context-independent) weight. This algorithm does not produce a probability distribution, so we'll follow Brants et al. (2007) in referring to it as S:

$$S(w_i|w_{i-N+1:i-1}) = \begin{cases} \dfrac{\text{count}(w_{i-N+1:i})}{\text{count}(w_{i-N+1:i-1})} & \text{if count}(w_{i-N+1:i}) > 0 \\ \lambda S(w_i|w_{i-N+2:i-1}) & \text{otherwise} \end{cases} \tag{3.31}$$

The backoff terminates in the unigram, which has score $S(w) = \frac{count(w)}{N}$. Brants et al. (2007) find that a value of 0.4 worked well for λ.

3.7 Advanced: Kneser-Ney Smoothing

Kneser-Ney

A popular advanced n-gram smoothing method is the interpolated **Kneser-Ney** algorithm (Kneser and Ney 1995, Chen and Goodman 1998).

3.7.1 Absolute Discounting

Kneser-Ney has its roots in a method called **absolute discounting**. Recall that **discounting** of the counts for frequent n-grams is necessary to save some probability mass for the smoothing algorithm to distribute to the unseen n-grams.

To see this, we can use a clever idea from Church and Gale (1991). Consider an n-gram that has count 4. We need to discount this count by some amount. But how much should we discount it? Church and Gale's clever idea was to look at a held-out corpus and just see what the count is for all those bigrams that had count 4 in the training set. They computed a bigram grammar from 22 million words of AP newswire and then checked the counts of each of these bigrams in another 22 million words. On average, a bigram that occurred 4 times in the first 22 million words occurred 3.23 times in the next 22 million words. Fig. 3.9 from Church and Gale (1991) shows these counts for bigrams with c from 0 to 9.

Bigram count in training set	Bigram count in heldout set
0	0.0000270
1	0.448
2	1.25
3	2.24
4	3.23
5	4.21
6	5.23
7	6.21
8	7.21
9	8.26

Figure 3.9 For all bigrams in 22 million words of AP newswire of count 0, 1, 2,...,9, the counts of these bigrams in a held-out corpus also of 22 million words.

Notice in Fig. 3.9 that except for the held-out counts for 0 and 1, all the other bigram counts in the held-out set could be estimated pretty well by just subtracting absolute discounting

0.75 from the count in the training set! **Absolute discounting** formalizes this intuition by subtracting a fixed (absolute) discount d from each count. The intuition is that since we have good estimates already for the very high counts, a small discount d won't affect them much. It will mainly modify the smaller counts, for which we don't necessarily trust the estimate anyway, and Fig. 3.9 suggests that in practice this discount is actually a good one for bigrams with counts 2 through 9. The equation for interpolated absolute discounting applied to bigrams:

$$P_{\text{AbsoluteDiscounting}}(w_i|w_{i-1}) = \frac{C(w_{i-1}w_i) - d}{\sum_v C(w_{i-1}\, v)} + \lambda(w_{i-1})P(w_i) \tag{3.32}$$

The first term is the discounted bigram, with $0 \leq d \leq 1$, and the second term is the unigram with an interpolation weight λ. By inspection of Fig. 3.9, it looks like just setting all the d values to .75 would work very well, or perhaps keeping a separate second discount value of 0.5 for the bigrams with counts of 1. There are principled methods for setting d; for example, Ney et al. (1994) set d as a function of n_1 and n_2, the number of unigrams that have a count of 1 and a count of 2, respectively:

$$d = \frac{n_1}{n_1 + 2n_2} \tag{3.33}$$

3.7.2 Kneser-Ney Discounting

Kneser-Ney discounting (Kneser and Ney, 1995) augments absolute discounting with a more sophisticated way to handle the lower-order unigram distribution. Consider the job of predicting the next word in this sentence, assuming we are interpolating a bigram and a unigram model.

I can't see without my reading ________________ .

The word *glasses* seems much more likely to follow here than, say, the word *Kong*, so we'd like our unigram model to prefer *glasses*. But in fact it's *Kong* that is more common, since *Hong Kong* is a very frequent word. A standard unigram model will assign *Kong* a higher probability than *glasses*. We would like to capture the intuition that although *Kong* is frequent, it is mainly only frequent in the phrase *Hong Kong*, that is, after the word *Hong*. The word *glasses* has a much wider distribution.

In other words, instead of $P(w)$, which answers the question "How likely is w?", we'd like to create a unigram model that we might call $P_{\text{CONTINUATION}}$, which answers the question "How likely is w to appear as a novel continuation?". How can we estimate this probability of seeing the word w as a novel continuation, in a new unseen context? The Kneser-Ney intuition is to base our estimate of $P_{\text{CONTINUATION}}$ on the *number of different contexts word w has appeared in*, that is, the number of bigram types it completes. Every bigram type was a novel continuation the first time it was seen. We hypothesize that words that have appeared in more contexts in the past are more likely to appear in some new context as well. The number of times a word w appears as a novel continuation can be expressed as:

$$P_{\text{CONTINUATION}}(w) \propto |\{v : C(vw) > 0\}| \tag{3.34}$$

To turn this count into a probability, we normalize by the total number of word bigram types. In summary:

$$P_{\text{CONTINUATION}}(w) = \frac{|\{v : C(vw) > 0\}|}{|\{(u',w') : C(u'w') > 0\}|} \tag{3.35}$$

An equivalent formulation based on a different metaphor is to use the number of word types seen to precede w (Eq. 3.34 repeated):

$$P_{\text{CONTINUATION}}(w) \propto |\{v : C(vw) > 0\}| \tag{3.36}$$

normalized by the number of words preceding all words, as follows:

$$P_{\text{CONTINUATION}}(w) = \frac{|\{v : C(vw) > 0\}|}{\sum_{w'} |\{v : C(vw') > 0\}|} \tag{3.37}$$

A frequent word (Kong) occurring in only one context (Hong) will have a low continuation probability.

Interpolated Kneser-Ney

The final equation for **Interpolated Kneser-Ney** smoothing for bigrams is then:

$$P_{\text{KN}}(w_i|w_{i-1}) = \frac{\max(C(w_{i-1}w_i) - d, 0)}{C(w_{i-1})} + \lambda(w_{i-1})P_{\text{CONTINUATION}}(w_i) \tag{3.38}$$

The λ is a normalizing constant that is used to distribute the probability mass we've discounted:

$$\lambda(w_{i-1}) = \frac{d}{\sum_v C(w_{i-1}v)} |\{w : C(w_{i-1}w) > 0\}| \tag{3.39}$$

The first term, $\frac{d}{\sum_v C(w_{i-1}v)}$, is the normalized discount (the discount d, $0 \le d \le 1$, was introduced in the absolute discounting section above). The second term, $|\{w : C(w_{i-1}w) > 0\}|$, is the number of word types that can follow w_{i-1} or, equivalently, the number of word types that we discounted; in other words, the number of times we applied the normalized discount.

The general recursive formulation is as follows:

$$P_{\text{KN}}(w_i|w_{i-n+1:i-1}) = \frac{\max(c_{KN}(w_{i-n+1:i}) - d, 0)}{\sum_v c_{KN}(w_{i-n+1:i-1}\ v)} + \lambda(w_{i-n+1:i-1})P_{KN}(w_i|w_{i-n+2:i-1}) \tag{3.40}$$

where the definition of the count c_{KN} depends on whether we are counting the highest-order n-gram being interpolated (for example trigram if we are interpolating trigram, bigram, and unigram) or one of the lower-order n-grams (bigram or unigram if we are interpolating trigram, bigram, and unigram):

$$c_{KN}(\cdot) = \begin{cases} \text{count}(\cdot) & \text{for the highest order} \\ \text{continuationcount}(\cdot) & \text{for lower orders} \end{cases} \tag{3.41}$$

The continuation count of a string $\cdot$ is the number of unique single word contexts for that string $\cdot$.

At the termination of the recursion, unigrams are interpolated with the uniform distribution, where the parameter ϵ is the empty string:

$$P_{\text{KN}}(w) = \frac{\max(c_{KN}(w) - d, 0)}{\sum_{w'} c_{KN}(w')} + \lambda(\epsilon)\frac{1}{V} \tag{3.42}$$

If we want to include an unknown word <UNK>, it's just included as a regular vocabulary entry with count zero, and hence its probability will be a lambda-weighted uniform distribution $\frac{\lambda(\epsilon)}{V}$.

modified Kneser-Ney

The best performing version of Kneser-Ney smoothing is called **modified Kneser-Ney** smoothing, and is due to Chen and Goodman (1998). Rather than use a single fixed discount d, modified Kneser-Ney uses three different discounts d_1, d_2, and d_{3+} for n-grams with counts of 1, 2 and three or more, respectively. See Chen and Goodman (1998, p. 19) or Heafield et al. (2013) for the details.

3.8 Advanced: Perplexity's Relation to Entropy

We introduced perplexity in Section 3.2.1 as a way to evaluate n-gram models on a test set. A better n-gram model is one that assigns a higher probability to the test data, and perplexity is a normalized version of the probability of the test set. The perplexity measure actually arises from the information-theoretic concept of cross-entropy, which explains otherwise mysterious properties of perplexity (why the inverse probability, for example?) and its relationship to entropy. **Entropy** is a measure of information. Given a random variable X ranging over whatever we are predicting (words, letters, parts of speech, the set of which we'll call χ) and with a particular probability function, call it $p(x)$, the entropy of the random variable X is:

Entropy

$$H(X) = -\sum_{x \in \chi} p(x) \log_2 p(x) \tag{3.43}$$

The log can, in principle, be computed in any base. If we use log base 2, the resulting value of entropy will be measured in **bits**.

One intuitive way to think about entropy is as a lower bound on the number of bits it would take to encode a certain decision or piece of information in the optimal coding scheme.

Consider an example from the standard information theory textbook Cover and Thomas (1991). Imagine that we want to place a bet on a horse race but it is too far to go all the way to Yonkers Racetrack, so we'd like to send a short message to the bookie to tell him which of the eight horses to bet on. One way to encode this message is just to use the binary representation of the horse's number as the code; thus, horse 1 would be `001`, horse 2 `010`, horse 3 `011`, and so on, with horse 8 coded as `000`. If we spend the whole day betting and each horse is coded with 3 bits, on average we would be sending 3 bits per race.

Can we do better? Suppose that the spread is the actual distribution of the bets placed and that we represent it as the prior probability of each horse as follows:

Horse 1	$\frac{1}{2}$	Horse 5	$\frac{1}{64}$
Horse 2	$\frac{1}{4}$	Horse 6	$\frac{1}{64}$
Horse 3	$\frac{1}{8}$	Horse 7	$\frac{1}{64}$
Horse 4	$\frac{1}{16}$	Horse 8	$\frac{1}{64}$

The entropy of the random variable X that ranges over horses gives us a lower bound on the number of bits and is

$$\begin{aligned} H(X) &= -\sum_{i=1}^{i=8} p(i) \log p(i) \\ &= -\tfrac{1}{2}\log\tfrac{1}{2} - \tfrac{1}{4}\log\tfrac{1}{4} - \tfrac{1}{8}\log\tfrac{1}{8} - \tfrac{1}{16}\log\tfrac{1}{16} - 4\left(\tfrac{1}{64}\log\tfrac{1}{64}\right) \\ &= 2 \text{ bits} \end{aligned} \tag{3.44}$$

A code that averages 2 bits per race can be built with short encodings for more probable horses, and longer encodings for less probable horses. For example, we could encode the most likely horse with the code `0`, and the remaining horses as `10`, then `110`, `1110`, `111100`, `111101`, `111110`, and `111111`.

What if the horses are equally likely? We saw above that if we used an equal-length binary code for the horse numbers, each horse took 3 bits to code, so the average was 3. Is the entropy the same? In this case each horse would have a probability of $\frac{1}{8}$. The entropy of the choice of horses is then

$$H(X) = -\sum_{i=1}^{i=8} \frac{1}{8} \log \frac{1}{8} = -\log \frac{1}{8} = 3 \text{ bits} \tag{3.45}$$

Until now we have been computing the entropy of a single variable. But most of what we will use entropy for involves *sequences*. For a grammar, for example, we will be computing the entropy of some sequence of words $W = \{w_1, w_2, \ldots, w_n\}$. One way to do this is to have a variable that ranges over sequences of words. For example we can compute the entropy of a random variable that ranges over all finite sequences of words of length n in some language L as follows:

$$H(w_1, w_2, \ldots, w_n) = -\sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n}) \tag{3.46}$$

entropy rate

We could define the **entropy rate** (we could also think of this as the **per-word entropy**) as the entropy of this sequence divided by the number of words:

$$\frac{1}{n} H(w_{1:n}) = -\frac{1}{n} \sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n}) \tag{3.47}$$

But to measure the true entropy of a language, we need to consider sequences of infinite length. If we think of a language as a stochastic process L that produces a sequence of words, and allow W to represent the sequence of words $w_1, \ldots, w_n$, then L's entropy rate $H(L)$ is defined as

$$\begin{aligned} H(L) &= \lim_{n\to\infty} \frac{1}{n} H(w_1, w_2, \ldots, w_n) \\ &= -\lim_{n\to\infty} \frac{1}{n} \sum_{W \in L} p(w_1, \ldots, w_n) \log p(w_1, \ldots, w_n) \end{aligned} \tag{3.48}$$

The Shannon-McMillan-Breiman theorem (Algoet and Cover 1988, Cover and Thomas 1991) states that if the language is regular in certain ways (to be exact, if it is both stationary and ergodic),

$$H(L) = \lim_{n\to\infty} -\frac{1}{n} \log p(w_1 w_2 \ldots w_n) \tag{3.49}$$

That is, we can take a single sequence that is long enough instead of summing over all possible sequences. The intuition of the Shannon-McMillan-Breiman theorem is that a long-enough sequence of words will contain in it many other shorter sequences and that each of these shorter sequences will reoccur in the longer sequence according to their probabilities.

Stationary

A stochastic process is said to be **stationary** if the probabilities it assigns to a sequence are invariant with respect to shifts in the time index. In other words, the probability distribution for words at time t is the same as the probability distribution at time $t+1$. Markov models, and hence n-grams, are stationary. For example, in a bigram, P_i is dependent only on P_{i-1}. So if we shift our time index by x, P_{i+x} is still dependent on P_{i+x-1}. But natural language is not stationary, since as we show in Appendix D, the probability of upcoming words can be dependent on events that were arbitrarily distant and time dependent. Thus, our statistical models only give an approximation to the correct distributions and entropies of natural language.

To summarize, by making some incorrect but convenient simplifying assumptions, we can compute the entropy of some stochastic process by taking a very long sample of the output and computing its average log probability.

cross-entropy

Now we are ready to introduce **cross-entropy**. The cross-entropy is useful when we don't know the actual probability distribution p that generated some data. It allows us to use some m, which is a model of p (i.e., an approximation to p). The cross-entropy of m on p is defined by

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n}\sum_{W\in L} p(w_1,\ldots,w_n)\log m(w_1,\ldots,w_n) \tag{3.50}$$

That is, we draw sequences according to the probability distribution p, but sum the log of their probabilities according to m.

Again, following the Shannon-McMillan-Breiman theorem, for a stationary ergodic process:

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n}\log m(w_1 w_2 \ldots w_n) \tag{3.51}$$

This means that, as for entropy, we can estimate the cross-entropy of a model m on some distribution p by taking a single sequence that is long enough instead of summing over all possible sequences.

What makes the cross-entropy useful is that the cross-entropy $H(p,m)$ is an upper bound on the entropy $H(p)$. For any model m:

$$H(p) \le H(p,m) \tag{3.52}$$

This means that we can use some simplified model m to help estimate the true entropy of a sequence of symbols drawn according to probability p. The more accurate m is, the closer the cross-entropy $H(p,m)$ will be to the true entropy $H(p)$. Thus, the difference between $H(p,m)$ and $H(p)$ is a measure of how accurate a model is. Between two models m_1 and m_2, the more accurate model will be the one with the lower cross-entropy. (The cross-entropy can never be lower than the true entropy, so a model cannot err by underestimating the true entropy.)

We are finally ready to see the relation between perplexity and cross-entropy as we saw it in Eq. 3.51. Cross-entropy is defined in the limit as the length of the observed word sequence goes to infinity. We will need an approximation to cross-entropy, relying on a (sufficiently long) sequence of fixed length. This approximation to the cross-entropy of a model $M = P(w_i|w_{i-N+1:i-1})$ on a sequence of words W is

$$H(W) = -\frac{1}{N}\log P(w_1 w_2 \ldots w_N) \tag{3.53}$$

perplexity

The **perplexity** of a model P on a sequence of words W is now formally defined as 2 raised to the power of this cross-entropy:

$$
\begin{aligned}
\text{Perplexity}(W) &= 2^{H(W)} \\
&= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\
&= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}} \\
&= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1 \ldots w_{i-1})}}
\end{aligned} \tag{3.54}
$$

3.9 Summary

This chapter introduced language modeling and the n-gram, one of the most widely used tools in language processing.

- Language models offer a way to assign a probability to a sentence or other sequence of words, and to predict a word from preceding words.
- n-grams are Markov models that estimate words from a fixed window of previous words. n-gram probabilities can be estimated by counting in a corpus and normalizing (the **maximum likelihood estimate**).
- n-gram **language models** are evaluated extrinsically in some task, or intrinsically using **perplexity.**
- The **perplexity** of a test set according to a language model is the geometric mean of the inverse test set probability computed by the model.
- **Smoothing** algorithms provide a more sophisticated way to estimate the probability of n-grams. Commonly used smoothing algorithms for n-grams rely on lower-order n-gram counts through **backoff** or **interpolation**.
- Both backoff and interpolation require **discounting** to create a probability distribution.
- **Kneser-Ney** smoothing makes use of the probability of a word being a novel **continuation**. The interpolated **Kneser-Ney** smoothing algorithm mixes a discounted probability with a lower-order continuation probability.

Bibliographical and Historical Notes

The underlying mathematics of the n-gram was first proposed by Markov (1913), who used what are now called **Markov chains** (bigrams and trigrams) to predict whether an upcoming letter in Pushkin's *Eugene Onegin* would be a vowel or a consonant. Markov classified 20,000 letters as V or C and computed the bigram and trigram probability that a given letter would be a vowel given the previous one or two letters. Shannon (1948) applied n-grams to compute approximations to English word sequences. Based on Shannon's work, Markov models were commonly used in engineering, linguistic, and psychological work on modeling word sequences by the 1950s. In a series of extremely influential papers starting with Chomsky (1956) and including Chomsky (1957) and Miller and Chomsky (1963), Noam Chomsky argued that "finite-state Markov processes", while a possibly useful engineering heuristic, were incapable of being a complete cognitive model of human grammatical knowledge. These arguments led many linguists and computational linguists to ignore work in statistical modeling for decades.

The resurgence of n-gram models came from Jelinek and colleagues at the IBM Thomas J. Watson Research Center, who were influenced by Shannon, and Baker at CMU, who was influenced by the work of Baum and colleagues. Independently these two labs successfully used n-grams in their speech recognition systems (Baker 1975b, Jelinek 1976, Baker 1975a, Bahl et al. 1983, Jelinek 1990).

Add-one smoothing derives from Laplace's 1812 law of succession and was first applied as an engineering solution to the zero frequency problem by Jeffreys (1948) based on an earlier Add-K suggestion by Johnson (1932). Problems with the add-one algorithm are summarized in Gale and Church (1994).

A wide variety of different language modeling and smoothing techniques were proposed in the 80s and 90s, including Good-Turing discounting—first applied to the n-gram smoothing at IBM by Katz (Nádas 1984, Church and Gale 1991)—Witten-Bell discounting (Witten and Bell, 1991), and varieties of **class-based n-gram** models that used information about word classes.

class-based n-gram

Starting in the late 1990s, Chen and Goodman performed a number of carefully controlled experiments comparing different discounting algorithms, cache models, class-based models, and other language model parameters (Chen and Goodman 1999, Goodman 2006, inter alia). They showed the advantages of **Modified Interpolated Kneser-Ney**, which became the standard baseline for n-gram language modeling, especially because they showed that caches and class-based models provided only minor additional improvement. These papers are recommended for any reader with further interest in n-gram language modeling. SRILM (Stolcke, 2002) and KenLM (Heafield 2011, Heafield et al. 2013) are publicly available toolkits for building n-gram language models.

Modern language modeling is more commonly done with **neural network** language models, which solve the major problems with n-grams: the number of parameters increases exponentially as the n-gram order increases, and n-grams have no way to generalize from training to test set. Neural language models instead project words into a **continuous** space in which words with similar contexts have similar representations. We'll introduce both **feedforward** language models (Bengio et al. 2006, Schwenk 2007) in Chapter 7, and **recurrent** language models (Mikolov, 2012) in Chapter 9.

Exercises

3.1 Write out the equation for trigram probability estimation (modifying Eq. 3.11). Now write out all the non-zero trigram probabilities for the `I am Sam` corpus on page 4.

3.2 Calculate the probability of the sentence `i want chinese food`. Give two probabilities, one using Fig. 3.2 and the 'useful probabilities' just below it on page 6, and another using the add-1 smoothed table in Fig. 3.7. Assume the additional add-1 smoothed probabilities $P(\texttt{i}|\texttt{<s>}) = 0.19$ and $P(\texttt{</s>}|\texttt{food}) = 0.40$.

3.3 Which of the two probabilities you computed in the previous exercise is higher, unsmoothed or smoothed? Explain why.

3.4 We are given the following corpus, modified from the one in the chapter:

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I am Sam </s>
<s> I do not like green eggs and Sam </s>
```

Using a bigram language model with add-one smoothing, what is P(Sam | am)? Include `<s>` and `</s>` in your counts just like any other token.

3.5 Suppose we didn't use the end-symbol `</s>`. Train an unsmoothed bigram grammar on the following training corpus without using the end-symbol `</s>`:

```
<s> a b
<s> b b
<s> b a
<s> a a
```

Demonstrate that your bigram model does not assign a single probability distribution across all sentence lengths by showing that the sum of the probability of the four possible 2 word sentences over the alphabet {a,b} is 1.0, and the sum of the probability of all possible 3 word sentences over the alphabet {a,b} is also 1.0.

3.6 Suppose we train a trigram language model with add-one smoothing on a given corpus. The corpus contains V word types. Express a formula for estimating P(w3|w1,w2), where w3 is a word which follows the bigram (w1,w2), in terms of various n-gram counts and V. Use the notation c(w1,w2,w3) to denote the number of times that trigram (w1,w2,w3) occurs in the corpus, and so on for bigrams and unigrams.

3.7 We are given the following corpus, modified from the one in the chapter:

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I am Sam </s>
<s> I do not like green eggs and Sam </s>
```

If we use linear interpolation smoothing between a maximum-likelihood bigram model and a maximum-likelihood unigram model with $\lambda_1 = \frac{1}{2}$ and $\lambda_2 = \frac{1}{2}$, what is P(Sam|am)? Include `<s>` and `</s>` in your counts just like any other token.

3.8 Write a program to compute unsmoothed unigrams and bigrams.

3.9 Run your n-gram program on two different small corpora of your choice (you might use email text or newsgroups). Now compare the statistics of the two corpora. What are the differences in the most common unigrams between the two? How about interesting differences in bigrams?

3.10 Add an option to your program to generate random sentences.

3.11 Add an option to your program to compute the perplexity of a test set.

3.12 You are given a training set of 100 numbers that consists of 91 zeros and 1 each of the other digits 1-9. Now we see the following test set: 0 0 0 0 0 3 0 0 0 0. What is the unigram perplexity?