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 word someone is going to say? What word, for example, is likely to follow

```
The water of Walden Pond is so beautifully ...
```

language model

LM

You might conclude that a likely word is `blue`, or `green`, or `clear`, but probably not `refrigerator` nor `this`. In this chapter we formalize this intuition by introducing **language models** or **LMs**, models that assign a **probability** to each possible next word. Language models can also assign a probability to an entire sentence, telling us 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
```

AAC

Why would we want to predict upcoming words, or know the probability of a sentence? One reason is for generation: choosing contextually better words. For example we can correct grammar or spelling errors 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` is more probable than `Their are`, and `has improved` than `has improve`, so a language model can help users select the more grammatical variant. Or for a speech system to recognize that you said `I will be back soonish` and not `I will be bassoon dish`, it helps to know that `back soonish` is a more probable sequence. Language models can also help in **augmentative and alternative communication** (Trnka et al. 2007, Kane et al. 2017). People can use **AAC** systems if they are physically unable to speak or sign but can instead use eye gaze or other movements to select words from a menu. Word prediction can be used to suggest likely words for the menu.

Word prediction is also central to NLP for another reason: **large language models** are built just by training them to predict words!! As we'll see in chapters 7-9, large language models learn an enormous amount about language solely from being trained to predict upcoming words from neighboring words.

n-gram

In this chapter we introduce the simplest kind of language model: the **n-gram** language model. 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 `The water`, or `water of`, and a 3-gram (a **trigram**) is a three-word sequence of words like `The water of`, or `water` `of Walden`. But we also (in a bit of terminological ambiguity) use the word 'n-gram' to mean a probabilistic model that can estimate the probability of a word given the n-1 previous words, and thereby also to assign probabilities to entire sequences.

In later chapters we will introduce the much more powerful neural **large language models**, based on the **transformer** architecture of Chapter 9. But because n-grams have a remarkably simple and clear formalization, we use them to introduce some major concepts of large language modeling, including **training and test sets**, **perplexity**, **sampling**, and **interpolation**.

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 "`The water of Walden Pond is so beautifully` " and we want to know the probability that the next word is `blue`:

$$P(\texttt{blue}|\texttt{The water of Walden Pond is so beautifully}) \tag{3.1}$$

One way to estimate this probability is directly from relative frequency counts: take a very large corpus, count the number of times we see `The water of Walden Pond is so beautifully`, and count the number of times this is followed by `blue`. 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:

$$P(\texttt{blue}|\texttt{The water of Walden Pond is so beautifully}) = \frac{C(\texttt{The water of Walden Pond is so beautifully blue})}{C(\texttt{The water of Walden Pond is so beautifully})} \tag{3.2}$$

If we had a large enough corpus, we could compute these two counts and estimate the probability from Eq. 3.2. But even the entire web isn't big enough to give us good estimates for counts of entire sentences. This is because language is **creative**; new sentences are invented all the time, and we can't expect to get accurate counts for such large objects as entire sentences. For this reason, we'll need more clever ways to estimate the probability of a word w given a history h, or the probability of an entire word sequence W.

Let's start with some notation. First, throughout this chapter we'll continue to refer to **words**, although in practice we usually compute language models over **tokens** like the BPE tokens of page **??**. 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 $w_{1:n}$. Thus the expression $w_{1:n-1}$ means the string $w_1, w_2, ..., w_{n-1}$, but we'll also be using the equivalent notation $w_{<n}$, which can be read as "all the elements of w from w_1 up to and including 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

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

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 in some corpus, because language is creative and any particular context might have never occurred before!

3.1.1 The Markov assumption

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

$$P(\texttt{blue}|\texttt{The water of Walden Pond is so beautifully}) \tag{3.5}$$

we approximate it with the probability

$$P(\texttt{blue}|\texttt{beautifully}) \tag{3.6}$$

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

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

Markov

n-gram

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).

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:

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

3.1.2 How to estimate probabilities

maximum likelihood estimation

normalize

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

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>.[1]

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

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

$$P(\texttt{I|<s>}) = \tfrac{2}{3} = 0.67 \qquad P(\texttt{Sam|<s>}) = \tfrac{1}{3} = 0.33 \qquad P(\texttt{am|I}) = \tfrac{2}{3} = 0.67$$
$$P(\texttt{</s>|Sam}) = \tfrac{1}{2} = 0.5 \qquad P(\texttt{Sam|am}) = \tfrac{1}{2} = 0.5 \qquad P(\texttt{do|I}) = \tfrac{1}{3} = 0.33$$

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. 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 0.0004. Now 0.0004 is not the best possible estimate of the probability of *Chinese* occurring in all situations; it

[1] 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.

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 3.6.

Let's move on to some examples from a real but tiny corpus, drawn 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 sample user queries (text-normalized, by lower casing and with punctuation striped) (a sample of 9332 sentences is on the website):

can you tell me about any good cantonese restaurants close by
tell me about chez panisse
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 part of a bigram grammar from text-normalized Berkeley Restaurant Project sentences. 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. Each cell shows the count of the column label word following the row label word. Thus the cell in row **i** and column **want** means that **want** followed **i** 827 times in the corpus.

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 counts):

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

Here are a few other useful probabilities:

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

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:

$$\begin{aligned}
&P(\texttt{<s> i want english food </s>}) \\
&\quad = P(\texttt{i|<s>})P(\texttt{want|i})P(\texttt{english|want}) \\
&\qquad\qquad P(\texttt{food|english})P(\texttt{</s>|food}) \\
&\quad = 0.25 \times 0.33 \times 0.0011 \times 0.5 \times 0.68 \\
&\quad = 0.000031
\end{aligned}$$

	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.

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.

3.1.3 Dealing with scale in large n-gram models

In practice, language models can be very large, leading to practical issues.

log probabilities

Log probabilities Language model probabilities are always stored and computed in log space as **log probabilities**. This is because probabilities are (by definition) less than or equal to 1, and so the more probabilities we multiply together, the smaller the product becomes. Multiplying enough n-grams together would result in numerical underflow. Adding in log space is equivalent to multiplying in linear space, so we combine log probabilities by adding them. By adding log probabilities instead of multiplying probabilities, we get results that are not as small. We do all computation and storage in log space, and just convert back into probabilities if we need to report probabilities at the end by taking the exp of the logprob:

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4) \qquad (3.13)$$

In practice throughout this book, we'll use log to mean natural log (ln) when the base is not specified.

trigram
4-gram
5-gram

Longer context Although for pedagogical purposes we have only described bigram models, when there is sufficient training data we use **trigram** models, which condition on the previous two words, or **4-gram** or **5-gram** models. 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., $P(\texttt{I|<s><s>})$).

Some large n-gram datasets have been created, like the million most frequent n-grams drawn from the Corpus of Contemporary American English (COCA), a curated 1 billion word corpus of American English (Davies, 2020), Google's Web 5-gram corpus from 1 trillion words of English web text (Franz and Brants, 2006), or the Google Books Ngrams corpora (800 billion tokens from Chinese, English, French, German, Hebrew, Italian, Russian, and Spanish) (Lin et al., 2012)).

It's even possible to use extremely long-range n-gram context. The infini-gram (∞-gram) project (Liu et al., 2024) allows n-grams of any length. Their idea is to avoid the expensive (in space and time) pre-computation of huge n-gram count tables. Instead, n-gram probabilities with arbitrary n are computed quickly at inference time by using an efficient representation called suffix arrays. This allows computing of n-grams of every length for enormous corpora of 5 trillion tokens.

Efficiency considerations are important when building large n-gram language models. It is standard to quantize the probabilities using only 4-8 bits (instead of 8-byte floats), store the word strings on disk and represent them in memory only as a 64-bit hash, and represent n-grams in special data structures like 'reverse tries'. It is also common to prune n-gram language models, for example by only keeping n-grams with counts greater than some threshold or using entropy to prune less-important n-grams (Stolcke, 1998). Efficient language model toolkits like KenLM (Heafield 2011, Heafield et al. 2013) use sorted arrays and use merge sorts to efficiently build the probability tables in a minimal number of passes through a large corpus.

3.2 Evaluating Language Models: Training and Test Sets

extrinsic evaluation

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 the language model (or any component) is really going to help the task at hand. Thus for evaluating n-gram language models that are a component of some task like speech recognition or machine translation, we can compare the performance of two candidate language models by running the speech recognizer or machine translator twice, once with each language model, and seeing which gives the more accurate transcription.

intrinsic evaluation

Unfortunately, running big NLP systems end-to-end is often very expensive. Instead, it's helpful 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. In the next section we'll introduce **perplexity**, which is the standard intrinsic metric for measuring language model performance, both for simple n-gram language models and for the more sophisticated neural large language models of Chapter 9.

training set

development set

test set

In order to evaluate any machine learning model, we need to have at least three distinct data sets: the **training set**, the **development set**, and the **test set**.

The **training set** is the data we use to learn the parameters of our model; for simple n-gram language models it's the corpus from which we get the counts that we normalize into the probabilities of the n-gram language model.

The **test set** is a different, held-out set of data, not overlapping with the training set, that we use to evaluate the model. We need a separate test set to give us an unbiased estimate of how well the model we trained can generalize when we apply it to some new unknown dataset. A machine learning model that perfectly captured the training data, but performed terribly on any other data, wouldn't be much use when it comes time to apply it to any new data or problem! We thus measure the quality of an n-gram model by its performance on this unseen test set or test corpus.

How should we choose a training and test set? The test set should reflect the language we want to use the model for. If we're going to use our language model for speech recognition of chemistry lectures, the test set should be text of chemistry lectures. If we're going to use it as part of a system for translating hotel booking requests from Chinese to English, the test set should be text of hotel booking requests. If we want our language model to be general purpose, then the test set should be drawn from a wide variety of texts. In such cases we might collect a lot of texts from different sources, and then divide it up into a training set and a test set. It's important to do the dividing carefully; if we're building a general purpose model, we don't want the test set to consist of only text from one document, or one author, since that wouldn't be a good measure of general performance.

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

But what does it mean to "fit the test set"? The standard answer is simple: whichever language model assigns a **higher probability** to the test set—which means it more accurately predicts the test set—is a better model. Given two probabilistic models, the better model is the one 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.

Even if we don't train on the test set, if we test our language model on the test set many times after making different changes, we might implicitly tune to its characteristics, by noticing which changes seem to make the model better. For this reason, we only want to run our model on the test set once, or a very few number of times, once we are sure our model is ready.

development test

For this reason we normally instead have a third dataset called a **development** test set or, **devset**. We do all our testing on this dataset until the very end, and then we test on the test once to see how good our model is.

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 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. It's important that the devset be drawn from the same kind of text as the test set, since its goal is to measure how we would do on the test set.

3.3 Evaluating Language Models: Perplexity

We said above that we evaluate language models based on which one assigns a higher probability to the test set. A better model is better at predicting upcoming words, and so it will be less surprised by (i.e., assign a higher probability to) each word when it occurs in the test set. Indeed, a perfect language model would correctly guess each next word in a corpus, assigning it a probability of 1, and all the other words a probability of zero. So given a test corpus, a better language model will assign a higher probability to it than a worse language model.

But in fact, we do not use raw probability as our metric for evaluating language models. The reason is that the probability of a test set (or any sequence) depends on the number of words or tokens in it; the probability of a test set gets smaller the longer the text. We'd prefer a metric that is per-word, normalized by length, so we could compare across texts of different lengths. The metric we use is, a function of probability called **perplexity**, is one of the most important metrics in NLP, used for evaluating large language models as well as n-gram models.

perplexity

The **perplexity** (sometimes abbreviated as PP or PPL) of a language model on a test set is the inverse probability of the test set (one over the probability of the test set), normalized by the number of words (or tokens). For this reason it's sometimes called the per-word or per-token perplexity. We normalize by the number of words N by taking the Nth root. For a test set $W = w_1 w_2 \ldots w_N$,:

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

Or we can use the chain rule to expand the probability of W:

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

Note that because of the inverse in Eq. 3.15, the higher the probability of the word sequence, the lower the perplexity. Thus the **the lower the perplexity of a model on the data, the better the model**. Minimizing perplexity is equivalent to maximizing the test set probability according to the language model. Why does perplexity use the inverse probability? It turns out the inverse arises from the original definition of perplexity from cross-entropy rate in information theory; for those interested, the explanation is in the advanced Section 3.7. Meanwhile, we just have to remember that perplexity has an inverse relationship with probability.

The details of computing 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 inverse 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 inverse of the bigram probabilities:

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

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, if our vocabulary includes a between-sentence token `<EOS>` or separate begin- and end-sentence markers `<s>` and `</s>` then we can include them in the probability computation. If we do, then we also include one token per sentence in the total count of word tokens N.[2]

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 language models. For example, here we trained unigram, bigram, and trigram grammars on 38 million words from the *Wall Street Journal* newspaper. We then computed the perplexity of each of these models on a WSJ test set using Eq. 3.16 for unigrams, Eq. 3.17 for bigrams, and the corresponding equation for trigrams. The table below shows the perplexity of the 1.5 million word test set according to each of the language models.

	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 probability of the test sequence according to the model). So a lower perplexity tells us that a language model is a better predictor of the test set.

Note that in computing perplexities, the language model must be constructed without any knowledge of the test set, or else the perplexity will be artificially low. And 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 usually correlates with task improvements, it is commonly used as a convenient evaluation metric. Still, when possible a model's improvement in perplexity should be confirmed by an end-to-end evaluation on a real task.

3.3.1 Perplexity as Weighted Average Branching Factor

It turns out that perplexity can also be thought of 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. For example consider a mini artificial language that is deterministic (no probabilities), any word can follow any word, and whose vocabulary consists of only three colors:

$$L = \{\texttt{red}, \texttt{blue}, \texttt{green}\} \tag{3.18}$$

The branching factor of this language is 3.

Now let's make a probabilistic version of the same LM, let's call it A, where each word follows each other with equal probability $\frac{1}{3}$ (it was trained on a training set with equal counts for the 3 colors), and a test set T = "`red red red red blue`".

Let's first convince ourselves that if we compute the perplexity of this artificial digit language on this test set (or any such test set) we indeed get 3. By Eq. 3.15, the

[2] For example if we use both begin and end tokens, we would include the end-of-sentence marker `</s>` but not the beginning-of-sentence marker `<s>` in our count of N; This is because the end-sentence token is followed directly by the begin-sentence token with probability almost 1, so we don't want the probability of that fake transition to influence our perplexity.

perplexity of A on T is:

$$\begin{aligned}\text{perplexity}_A(T) &= P_A(\texttt{red red red red blue})^{-\frac{1}{5}} \\ &= \left(\left(\frac{1}{3}\right)^5\right)^{-\frac{1}{5}} \\ &= \left(\frac{1}{3}\right)^{-1} = 3 \qquad (3.19)\end{aligned}$$

But now suppose `red` was very likely in the training set a different LM B, and so B has the following probabilities:

$$P(\texttt{red}) = 0.8 \quad P(\texttt{green}) = 0.1 \quad P(\texttt{blue}) = 0.1 \qquad (3.20)$$

We should expect the perplexity of the same test set `red red red red blue` for language model B to be lower since most of the time the next color will be red, which is very predictable, i.e. has a high probability. So the probability of the test set will be higher, and since perplexity is inversely related to probability, the perplexity will be lower. Thus, although the branching factor is still 3, the perplexity or *weighted* branching factor is smaller:

$$\begin{aligned}\text{perplexity}_B(T) &= P_B(\texttt{red red red red blue})^{-1/5} \\ &= 0.04096^{-\frac{1}{5}} \\ &= 0.527^{-1} = 1.89 \qquad (3.21)\end{aligned}$$

3.4 Sampling sentences from a language model

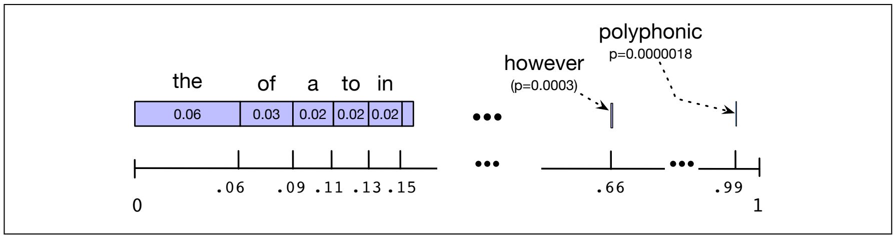

Figure 3.3 A visualization of the sampling distribution for sampling sentences by repeatedly sampling unigrams. The blue bar represents the relative frequency of each word (we've ordered them from most frequent to least frequent, but the choice of order is arbitrary). The number line shows the cumulative 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*).

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 (1948) 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 number line 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>`.

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.

3.5 Generalizing vs. overfitting the training set

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, the more coherent the sentences. The unigram sentences show no coherent relation between words nor any sentence-final punctuation. The bigram sentences have some local word-to-word coherence (especially considering punctuation as words). The trigram sentences are beginning to look a lot like Shakespeare. Indeed, the 4-gram sentences 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 3-gram (*It cannot be*), there are only seven possible next words for the 4th element (*but*, *I*, *that*, *thus*, *this*, and the period).

To get an idea of the dependence on the training set, let's look at LMs trained on a completely different corpus: the *Wall Street Journal* (WSJ) newspaper. Shakespeare and the WSJ are both English, so we might have expected 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.

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.

Compare these examples to the pseudo-Shakespeare in Fig. 3.4. While they both model "English-like sentences", there is no overlap in the generated sentences, and little overlap even in small phrases. Statistical models are pretty useless as predictors if the training sets and the test sets are as different as Shakespeare and the 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, 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 English (AAE)— the name for the many variations of language used in African American communities (King, 2020). Such features can 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):

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

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

(3.23) @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

Is it possible for the testset nonetheless to have a word we have never seen before? What happens if the word *Jurafsky* never occurs in our training set, but pops up in the test set? The answer is that although words might be unseen, we actually run our NLP algorithms not on words but on **subword tokens**. With subword tokenization (like the BPE algorithm of Chapter 2) any word can be modeled as a sequence of known smaller subwords, if necessary by a sequence of individual letters. So although for convenience we've been referring to words in this chapter, the language model vocabulary is actually the set of tokens rather than words, and the test set can never contain unseen tokens.

3.6 Smoothing, Interpolation, and Backoff

There is a problem with using maximum likelihood estimates for probabilities: any finite training corpus will be missing some perfectly acceptable English word sequences. That is, cases where a particular n-gram never occurs in the training data but appears in the test set. Perhaps our training corpus has the words `ruby` and `slippers` in it but just happens not to have the phrase `ruby slippers`.

zeros

These unseen sequences or **zeros**—sequences that don't 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 word sequences that might occur, which hurts 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 probability of the whole test set is 0. Perplexity is defined based on the inverse probability of the test set. Thus if some words in context have zero probability, we can't compute perplexity at all, since we can't divide by 0!

smoothing

discounting

The standard way to deal with putative "zero probability n-grams" that should really have some non-zero probability is called **smoothing** or **discounting**. Smoothing algorithms shave off a bit of probability mass from some more frequent events and give it to unseen events. Here we'll introduce some simple smoothing algorithms: **Laplace (add-one) smoothing**, **stupid backoff**, and n-gram **interpolation**.

3.6.1 Laplace Smoothing

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).

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:

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

add-one

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?)

$$P_{\text{Laplace}}(w_i) = \frac{c_i+1}{N+V} \tag{3.24}$$

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 $\frac{N}{N+V}$:

$$c_i^* = (c_i+1)\frac{N}{N+V} \tag{3.25}$$

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

discounting

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_i, the ratio of the discounted counts to the original counts:

discount

$$d_i = \frac{c_i^*}{c_i}$$

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:

$$P_{\text{MLE}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.26}$$

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

$$P_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{\sum_w (C(w_{n-1}w)+1)} = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V} \tag{3.27}$$

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.

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

	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.

	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.

computed by Eq. 3.28. Figure 3.8 shows the reconstructed counts.

$$c^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n)+1] \times C(w_{n-1})}{C(w_{n-1})+V} \tag{3.28}$$

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 0.66 in the unsmoothed case to 0.26 in the smoothed case. Looking at the discount d (the ratio between new and old counts) shows us how strikingly the counts for each prefix word have been reduced; the discount for the bigram *want to* is 0.39, while the discount for *Chinese food* is 0.10, a factor of 10! The sharp change occurs because too much probability mass is moved to all the zeros.

3.6.2 Add-k smoothing

add-k

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 (0.5? 0.01?). This algorithm is therefore called **add-k smoothing**.

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+k}{C(w_{n-1})+kV} \tag{3.29}$$

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).

3.6.3 Language Model Interpolation

There is an alternative source of knowledge we can draw on to solve the problem of zero frequency n-grams. 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** can help us generalize more for contexts that the model hasn't learned much about.

interpolation

The most common way to use this n-gram hierarchy is called **interpolation**: computing a new probability by interpolating (weighting and combining) the trigram, bigram, and unigram probabilities.[3] 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 λ:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) \;=\; & \lambda_1 P(w_n) \\ & +\lambda_2 P(w_n|w_{n-1}) \\ & +\lambda_3 P(w_n|w_{n-2}w_{n-1})\end{aligned} \tag{3.30}$$

The λs must sum to 1, making Eq. 3.30 equivalent to a weighted average. 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 3.31 shows the equation for interpolation with context-conditioned weights, where each *lambda* takes an argument that is the two prior word context:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) \;=\; & \lambda_1(w_{n-2:n-1})P(w_n) \\ & +\lambda_2(w_{n-2:n-1})P(w_n|w_{n-1}) \\ & +\lambda_3(w_{n-2:n-1})P(w_n|w_{n-2}w_{n-1})\end{aligned} \tag{3.31}$$

held-out

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 these λ values.[4] 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. 3.30—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).

3.6.4 Stupid Backoff

backoff

An alternative to interpolation is **backoff**. In a backoff model, if the n-gram we need

[3] We won't discuss the less-common alternative, called **backoff**, in which we use the trigram if the evidence is sufficient for it, but if not we instead just use the bigram, otherwise the unigram. That is, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order n-gram.

[4] Held-out corpora are generally used to set **hyperparameters**, which are special parameters, unlike regular counts that are learned from the training data; we'll discuss hyperparameters in Chapter 7.

discount

stupid backoff

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. 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. In practice, instead of discounting, it's common to use a much simpler non-discounted backoff algorithm called **stupid backoff** (Brants et al., 2007).

Stupid backoff gives up the idea of trying to make the language model a true probability distribution. 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:

$$S(w_i|w_{i-N+1:i-1}) = \begin{cases} \dfrac{\text{count}(w_{i-N+1:i})}{\text{count}(w_{i-N+1:i-1})} & \text{if } \text{count}(w_{i-N+1:i}) > 0 \\ \lambda S(w_i|w_{i-N+2:i-1}) & \text{otherwise} \end{cases} \tag{3.32}$$

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: Perplexity's Relation to Entropy

Entropy

We introduced perplexity in Section 3.3 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:

$$H(X) = -\sum_{x\in\chi} p(x)\log_2 p(x) \tag{3.33}$$

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

$$\begin{aligned} H(X) &= -\sum_{i=1}^{i=8} p(i)\log_2 p(i) \\ &= -\tfrac{1}{2}\log_2\tfrac{1}{2} - \tfrac{1}{4}\log_2\tfrac{1}{4} - \tfrac{1}{8}\log_2\tfrac{1}{8} - \tfrac{1}{16}\log_2\tfrac{1}{16} - 4(\tfrac{1}{64}\log_2\tfrac{1}{64}) \\ &= 2 \text{ bits} \end{aligned} \tag{3.34}$$

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

$$H(X) = -\sum_{i=1}^{i=8}\frac{1}{8}\log_2\frac{1}{8} = -\log_2\frac{1}{8} = 3 \text{ bits} \tag{3.35}$$

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 sequences of words of length n in some language L as follows:

$$H(w_1, w_2, \ldots, w_n) = -\sum_{w_{1:n}\in L} p(w_{1:n})\log p(w_{1:n}) \tag{3.36}$$

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:

$$\frac{1}{n}H(w_{1:n}) = -\frac{1}{n}\sum_{w_{1:n}\in L} p(w_{1:n})\log p(w_{1:n}) \tag{3.37}$$

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

$$\begin{aligned} H(L) &= \lim_{n\to\infty}\frac{1}{n}H(w_{1:n}) \\ &= -\lim_{n\to\infty}\frac{1}{n}\sum_{W\in L} p(w_{1:n})\log p(w_{1:n}) \end{aligned} \tag{3.38}$$

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),

$$H(L) = \lim_{n\to\infty} -\frac{1}{n} \log p(w_{1:n}) \tag{3.39}$$

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

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n} \sum_{W\in L} p(w_1,\ldots,w_n) \log m(w_1,\ldots,w_n) \tag{3.40}$$

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:

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n} \log m(w_1 w_2 \ldots w_n) \tag{3.41}$$

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:

$$H(p) \le H(p,m) \tag{3.42}$$

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. 3.41. Cross-entropy is defined in the limit as the length of the observed word sequence goes to infinity. We approximate this cross-entropy by 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

$$H(W) = -\frac{1}{N}\log P(w_1 w_2 \ldots w_N) \tag{3.43}$$

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:

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

3.8 Summary

This chapter introduced language modeling via the n-gram model, a classic model that allows us to introduce many of the basic concepts in language modeling.

- Language models offer a way to assign a probability to a sentence or other sequence of words or tokens, and to predict a word or token from preceding words or tokens.
- **N-grams** are perhaps the simplest kind of language model. They are Markov models that estimate words from a fixed window of previous words. N-gram models can be trained by counting in a **training corpus** and normalizing the counts (the **maximum likelihood estimate**).
- N-gram **language models** can be evaluated on a **test set** using **perplexity.**
- The **perplexity** of a test set according to a language model is a function of the probability of the test set: the inverse test set probability according to the model, normalized by the length.
- **Sampling** from a language model means to generate some sentences, choosing each sentence according to its likelihood as defined by the model.
- **Smoothing** algorithms provide a way to estimate probabilities for events that were unseen in training. Commonly used smoothing algorithms for n-grams include add-1 smoothing, or rely on lower-order n-gram counts through **interpolation**.

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 language models came from Fred Jelinek and colleagues at the IBM Thomas J. Watson Research Center, who were influenced by Shannon, and James Baker at CMU, who was influenced by the prior, classified work of Leonard Baum and colleagues on these topics at labs like the US Institute for Defense SAnalyses (IDA) after they were declassified. Independently these two labs successfully used n-grams in their speech recognition systems at the same time (Baker 1975b, Jelinek et al. 1975, Baker 1975a, Bahl et al. 1983, Jelinek 1990). The terms "language model" and "perplexity" were first used for this technology by the IBM group. Jelinek and his colleagues used the term *language model* in a pretty modern way, to mean the entire set of linguistic influences on word sequence probabilities, including grammar, semantics, discourse, and even speaker characteristics, rather than just the particular n-gram model itself.

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).

class-based n-gram

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. Starting in the late 1990s, Chen and Goodman performed a number of carefully controlled experiments comparing different algorithms and 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 around the turn of the century, especially because they showed that caches and class-based models provided only minor additional improvement. SRILM (Stolcke, 2002) and KenLM (Heafield 2011, Heafield et al. 2013) are publicly available toolkits for building n-gram language models.

Large language models are based on **neural networks** rather than n-grams, enabling them to solve the two major problems with n-grams: (1) the number of parameters increases exponentially as the n-gram order increases, and (2) n-grams have no way to generalize from training examples to test set examples unless they use identical words. Neural language models instead project words into a **continuous** space in which words with similar contexts have similar representations. We'll introduce transformer-based **large language models** in Chapter 9, along the way introducing **feedforward** language models (Bengio et al. 2006, Schwenk 2007) in Chapter 7 and **recurrent** language models (Mikolov, 2012) in Chapter 8.

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?