Chapter 4
N-grams

But it must be recognized that the notion "probability of a sentence" is an entirely useless one, under any known interpretation of this term.
Noam Chomsky (1969, p. 57)

Anytime a linguist leaves the group the recognition rate goes up.
Fred Jelinek (then of the IBM speech group) (1988)[1]

Being able to predict the future is not always a good thing. Cassandra of Troy had the gift of foreseeing, but was cursed by Apollo that her predictions would never be believed. Her warnings of the destruction of Troy were ignored and to simplify, let's just say that things just didn't go well for her later.

Predicting words seems somewhat less fraught, and in this chapter we take up this idea of word prediction. 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 *the*. We formalize this idea of **word prediction** with probabilistic models called *N*-**gram models**, which predict the next word from the previous $N-1$ words. Such statistical models of word sequences are also called **language models** or **LMs**. Computing the probability of the next word will turn out to be closely related to computing the probability of a sequence of words. The following sequence, for example, has a non-zero probability of appearing in a text:

N-gram
Language model
LM

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

while this same set of words in a different order has a much much lower probability:

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

As we will see, estimators like *N*-grams that assign a conditional probability to possible next words can be used to assign a joint probability to an entire sentence. Whether estimating probabilities of next words or of whole sequences, the *N*-gram model is one of the most important tools in speech and language processing.

N-grams are essential in any task in which we have to identify words in noisy, ambiguous input. In **speech recognition**, for example, the input speech sounds are very

[1] This wording from his address is as recalled by Jelinek himself; the quote didn't appear in the proceedings (Palmer and Finin, 1990). Some remember a more snappy version: *Every time I fire a linguist the performance of the recognizer improves*.

confusable and many words sound extremely similar. Russell and Norvig (2002) give an intuition from **handwriting recognition** for how probabilities of word sequences can help. In the movie *Take the Money and Run*, Woody Allen tries to rob a bank with a sloppily written hold-up note that the teller incorrectly reads as “I have a gub”. Any speech and language processing system could avoid making this mistake by using the knowledge that the sequence “I have a gun” is far more probable than the non-word “I have a gub” or even “I have a gull”.

N-gram models are also essential in statistical **machine translation**. Suppose we are translating a Chinese source sentence 他向记者介绍了该声明的主要内容 and as part of the process we have a set of potential rough English translations:

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

An *N*-gram grammar might tell us that, even after controlling for length, *briefed reporters* is more likely than *briefed to reporters*, and *main contents* is more likely than **chief contents**. This lets us select the bold-faced sentence above as the most fluent translation sentence, i.e. the one that has the highest probability.

In **spelling correction**, we need to find and correct spelling errors like the following (from Kukich (1992)) that accidentally result in real English words:

They are leaving in about fifteen *minuets* to go to her house.
The design *an* construction of the system will take more than a year.

Since these errors have real words, we can’t find them by just flagging words that aren’t in the dictionary. But note that *in about fifteen minuets* is a much less probable sequence than *in about fifteen minutes*. A spellchecker can use a probability estimator both to detect these errors and to suggest higher-probability corrections.

augmentative communication

Word prediction is also important for **augmentative communication** (Newell et al., 1998) systems that help the disabled. People who are unable to use speech or sign-language to communicate, like the physicist Steven Hawking, can communicate by using simple body movements to select words from a menu that are spoken by the system. Word prediction can be used to suggest likely words for the menu.

Besides these sample areas, *N*-grams are also crucial in NLP tasks like **part-of-speech tagging**, **natural language generation**, and **word similarity**, as well as in applications from **authorship identification** and **sentiment extraction** to **predictive text input** systems for cell phones.

4.1 Counting Words in Corpora

[upon being asked if there weren’t enough words in the English language for him]:
“Yes, there are enough, but they aren’t the right ones.”
James Joyce, reported in Bates (1997)

Probabilities are based on counting things. Before we talk about probabilities, we need to decide what we are going to count. Counting of things in natural language is based on a **corpus** (plural **corpora**), an on-line collection of text or speech. Let's look at two popular corpora, Brown and Switchboard. The Brown corpus is a 1 million word collection of samples from 500 written texts from different genres (newspaper, novels, non-fiction, academic, etc.), assembled at Brown University in 1963-64 (Kučera and Francis, 1967; Francis, 1979; Francis and Kučera, 1982). How many words are in the following Brown sentence?

Corpus

Corpora

(4.1) He stepped out into the hall, was delighted to encounter a water brother.

Example (4.1) has 13 words if we don't count punctuation marks as words, 15 if we count punctuation. Whether we treat period ("."), comma (","), and so on as words depends on the task. Punctuation is critical for finding boundaries of things (commas, periods, colons), and for identifying some aspects of meaning (question marks, exclamation marks, quotation marks). For some tasks, like part-of-speech tagging or parsing or speech synthesis, we sometimes treat punctuation marks as if they were separate words.

The Switchboard corpus of telephone conversations between strangers was collected in the early 1990s and contains 2430 conversations averaging 6 minutes each, totaling 240 hours of speech and about 3 million words (Godfrey et al., 1992). Such corpora of spoken language don't have punctuation, but do introduce other complications with regard to defining words. Let's look at one utterance from Switchboard; an **utterance** is the spoken correlate of a sentence:

Utterance

(4.2) I do uh main- mainly business data processing

This utterance has two kinds of **disfluencies**. The broken-off word *main-* is called a **fragment**. Words like *uh* and *um* are called **fillers** or **filled pauses**. Should we consider these to be words? Again, it depends on the application. If we are building an automatic dictation system based on automatic speech recognition, we might want to eventually strip out the disfluencies.

Disfluency

Fragment

Filled pause

But we also sometimes keep disfluencies around. How disfluent a person is can be used to identify them, or to detect whether they are stressed or confused. Disfluencies also often occur with particular syntactic structures, so they may help in parsing and word prediction. Stolcke and Shriberg (1996) found for example that treating *uh* as a word improves next-word prediction (why might this be?), and so most speech recognition systems treat *uh* and *um* as words.[2]

Are capitalized tokens like *They* and uncapitalized tokens like *they* the same word? These are lumped together in speech recognition, while for part-of-speech-tagging capitalization is retained as a separate feature. For the rest of this chapter we will assume our models are not case-sensitive.

How about inflected forms like *cats* versus *cat*? These two words have the same **lemma** *cat* but are different wordforms. Recall from Ch. 3 that a lemma is a set of lexical forms having the same stem, the same major part-of-speech, and the same word-sense. The **wordform** is the full inflected or derived form of the word. For morphologically complex languages like Arabic we often need to deal with lemmati-

Wordform

2 Clark and Fox Tree (2002) showed that *uh* and *um* have different meanings. What do you think they are?

zation. N-grams for speech recognition in English, however, and all the examples in this chapter, are based on wordforms.

As we can see, N-gram models, and counting words in general, requires that we do the kind of tokenization or text normalization that we introduced in the previous chapter: separating out punctuation, dealing with abbreviations like *m.p.h.*, normalizing spelling, and so on.

Word type

Word token

How many words are there in English? To answer this question we need to distinguish **types**, the number of distinct words in a corpus or vocabulary size V, from **tokens**, the total number N of running words. The following Brown sentence has 16 tokens and 14 types (not counting punctuation):

(4.3) They picnicked by the pool, then lay back on the grass and looked at the stars.

The Switchboard corpus has about 20,000 wordform types (from about 3 million wordform tokens). Shakespeare's complete works have 29,066 wordform types (from 884,647 wordform tokens) (Kučera, 1992). The Brown corpus has 61,805 wordform types from 37,851 lemma types (from 1 million wordform tokens). Looking at a very large corpus of 583 million wordform tokens, Brown et al. (1992) found that it included 293,181 different wordform types. Dictionaries can help in giving lemma counts; dictionary entries, or **boldface forms** are a very rough upper bound on the number of lemmas (since some lemmas have multiple boldface forms). The American Heritage Dictionary lists 200,000 boldface forms. It seems like the larger corpora we look at, the more word types we find. In general Gale and Church (1990) suggest that the vocabulary size (the number of types) grows with at least the square root of the number of tokens (i.e. $V > O(\sqrt{N})$.

In the rest of this chapter we will continue to distinguish between types and tokens, using "types" to mean wordform types.

4.2 Simple (Unsmoothed) N-grams

Let's start with some intuitive motivations for N-grams. We assume that the reader has acquired some very basic background in probability theory. Our goal is to compute the probability of a word w given some history h, or $P(w|h)$. Suppose the history h is "*its water is so transparent that*" and we want to know the probability that the next word is *the*:

$$P(the|its\ water\ is\ so\ transparent\ that). \tag{4.4}$$

How can we compute this probability? One way is to estimate it from relative frequency counts. For example, we could take a very large corpus, count the number of times we see *the 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:

$$P(the|its\ water\ is\ so\ transparent\ that) =$$

$$\frac{C(\textit{its water is so transparent that the})}{C(\textit{its water is so transparent that})} \tag{4.5}$$

With a large enough corpus, such as the web, we can compute these counts, and estimate the probability from Eq. 4.5. 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*").

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 5 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 5 word sequences. That seems rather a lot to estimate!

For this reason, we'll need to introduce cleverer 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. In order 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. For the joint probability of each word in a sequence having a particular value $P(X = w_1, Y = w_2, Z = w_3, ..., W = w_n)$ 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 to decompose this probability using the **chain rule of probability**:

$$\begin{aligned} P(X_1 ... X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_1^2) \dots P(X_n|X_1^{n-1}) \\ &= \prod_{k=1}^{n} P(X_k|X_1^{k-1}) \end{aligned} \tag{4.6}$$

Applying the chain rule to words, we get:

$$\begin{aligned} P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2) \dots P(w_n|w_1^{n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_1^{k-1}) \end{aligned} \tag{4.7}$$

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. Eq. 4.7 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 will **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

$$P(\text{the}|\text{Walden Pond's water is so transparent that}) \tag{4.8}$$

we approximate it with the probability

$$P(\text{the}|\text{that}) \tag{4.9}$$

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

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1}) \tag{4.10}$$

This 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 that 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

Thus the general equation for this N-gram approximation to the conditional probability of the next word in a sequence is:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1}) \tag{4.11}$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by substituting Eq. 4.10 into Eq. 4.7:

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{4.12}$$

Maximum Likelihood Estimation MLE

Normalize

How do we estimate these bigram or N-gram probabilities? The simplest and most 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 taking counts from a corpus, and **normalizing** them so they lie between 0 and 1.[3]

For example, to compute a particular bigram probability of a word y given a previous word x, we'll compute the count of the bigram $C(xy)$ and normalize by the sum of all the bigrams that share the same first word x:

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

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \tag{4.13}$$

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):

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{4.14}$$

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>`.[4]

```
<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

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

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

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

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}w_n)}{C(w_{n-N+1}^{n-1})} \tag{4.15}$$

Eq. 4.15 (like Eq. 4.14) 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 Maximum Likelihood Estimation, 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 estimate 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 will see ways to modify the MLE estimates slightly to get better probability estimates in Sec. 4.5.

Relative frequency

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

[4] As Chen and Goodman (1998) point out, we need the end-symbol to make the bigram grammar a true probability distribution. Without an end-symbol, the sentence probabilities for all sentences of a given length would sum to one, and the probability of the whole language would be infinite.

restaurants in Berkeley, California (Jurafsky et al., 1994). Here are some sample user queries, lowercased and with no punctuation (a representative corpus 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 4.1 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 seven 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 4.1 Bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences.

Fig. 4.2 shows the bigram probabilities after normalization (dividing each row by the following unigram counts):

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 4.2 Bigram probabilities for eight words in the Berkeley Restaurant Project corpus of 9332 sentences.

Here are a few other useful probabilities:

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

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:

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

We leave it as an exercise for the reader to compute the probability of *i want chinese food*. But that exercise does suggest that we'll want to think a bit about what kinds of linguistic phenomena are captured in bigrams. 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 more cultural than linguistic, like the low probability of anyone asking for advice on finding English food.

Although we will generally show bigram models in this chapter for pedagogical purposes, note that when there is sufficient training data we are more likely to use *Trigram* **trigram** models, which condition on the previous two words rather than the previous word. To compute trigram probabilities at the very beginning of sentence, we can use two pseudo-words for the first trigram (i.e., $P(\texttt{I|<s><s>})$.

4.3 Training and Test Sets

The N-gram model is a good example of the kind of statistical models that we will be seeing throughout speech and language processing. The probabilities of an N-gram model come from the corpus it is trained on. In general, the parameters of a statistical model are trained on some set of data, and then we apply the models to some new data in some task (such as speech recognition) and see how well they work. Of course this new data or task won't be the exact same data we trained on.

We can formalize this idea of training on some data, and testing on some other data by talking about these two data sets as a **training set** and a **test set** (or a **training corpus** and a **test corpus**). Thus when using a statistical model of language given some corpus of relevant data, we start by dividing the data into training and test sets. We train the statistical parameters of the model on the training set, and then use this trained model to compute probabilities on the test set.

Training set

Test set

Evaluation

This training-and-testing paradigm can also be used to **evaluate** different N-gram architectures. Suppose we want to compare different language models (such as those based on N-grams of different orders N, or using the different **smoothing** algorithms to be introduced in Sec. 4.5). We can do this by taking a corpus and dividing it into a training set and a test set. Then we train the two different N-gram models on the training set and see which one better models the test set. But what does it mean to "model the test set"? There is a useful metric for how well a given statistical model matches a test corpus, called **perplexity**, introduced on page 95. Perplexity is based on computing the probability of each sentence in the test set; intuitively, whichever model assigns a higher probability to the test set (hence more accurately predicts the test set) is a better model.

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.

In addition to training and test sets, other divisions of data are often useful. Sometimes we need an extra source of data to augment the training set. Such extra data is called a **held-out** set, because we hold it out from our training set when we train our N-gram counts. The held-out corpus is then used to set some other parameters; for example we will see the use of held-out data to set interpolation weights in **interpolated** N-gram models in Sec. 4.6. Finally, sometimes we need to have multiple test sets. This happens because we might use a particular test set so often that we implicitly tune to its characteristics. Then we would definitely need a fresh test set which is truly unseen. In such cases, we call the initial test set the **development** test set or, **devset**. We will discuss development test sets again in Ch. 5.

Held-out set

Development test

How do we divide our data into training, dev, and test sets? There is a tradeoff, since we want our test set to be as large as possible and a small test set may be accidentally unrepresentative. On the other hand, we 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.

4.3.1 N-gram Sensitivity to the Training Corpus

The N-gram model, like many statistical models, is very dependent on the training corpus. One implication of this is that the probabilities often encode very 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 visualize both of these facts by borrowing the technique of Shannon (1951) and Miller and Selfridge (1950), of generating random sentences from different N-gram models. It's simplest to visualize how this works for the unigram case. Imagine all the words of English covering the probability space between 0 and 1, each word covering an interval equal to its frequency. We choose a random value between 0 and 1, and print out the word whose interval includes the real value we have chosen. We continue choosing random numbers and generating words until we randomly generate the sentence-final token `</s>`. The same technique can be used to generate bigrams by first generating a random bigram that starts with `<s>` (according to its bigram probability), then choosing a random bigram to follow it (again, according to its conditional probability), and so on.

To give an intuition for the increasing power of higher-order N-grams, Fig. 4.3 shows random sentences generated from unigram, bigram, trigram, and quadrigram models trained on Shakespeare's works.

Unigram

To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have
Every enter now severally so, let
Hill he late speaks; or! a more to leg less first you enter
Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like

Bigram

What means, sir. I confess she? then all sorts, he is trim, captain.
Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.
What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?

Trigram

Sweet prince, Falstaff shall die. Harry of Monmouth's grave.
This shall forbid it should be branded, if renown made it empty.
Indeed the duke; and had a very good friend.
Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.

Quadrigram

King Henry.What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;
Will you not tell me who I am?
It cannot be but so.
Indeed the short and the long. Marry, 'tis a noble Lepidus.

Figure 4.3 Sentences randomly generated from four N-gram models computed from Shakespeare's works. All characters were mapped to lower case and punctuation marks were treated as words. Output was 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, nor any sentence-final punctuation. The bigram sentences have some very local word-to-word coherence (especially if we consider that punctuation counts as a word). The trigram and quadrigram sentences are beginning to look a lot like Shakespeare. Indeed a careful investigation of the quadrigram 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 quadrigrams is $V^4 = 7 \times 10^{17}$. Thus once the generator has chosen the first quadrigram (*It cannot be but*), there are only five possible continuations (*that*, *I*, *he*, *thou*, and *so*); indeed for many quadrigrams 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 Jour-*

nal (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. In order to check whether this is true, Fig. 4.4 shows sentences generated by unigram, bigram, and trigram grammars trained on 40 million words from WSJ.

Unigram

Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives

Bigram

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

Trigram

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 4.4 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. 4.3. While superficially they both seem to model "English-like sentences" there is obviously no overlap whatsoever in possible sentences, and little if any overlap even in small phrases. This stark difference tells us that 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? In general we need to be sure to use a training corpus that looks like our test corpus. We especially wouldn't choose training and tests from different **genres** of text like newspaper text, early English fiction, telephone conversations, and web pages. Sometimes finding appropriate training text for a specific new task can be difficult; to build N-grams for text prediction in SMS (Short Message Service), we need a training corpus of SMS data. To build N-grams on business meetings, we would need to have corpora of transcribed business meetings.

For general research where we know we want written English but don't have a domain in mind, we can use a balanced training corpus that includes cross sections from different genres, such as the 1-million-word Brown corpus of English (Francis and Kučera, 1982) or the 100-million-word British National Corpus (Leech et al., 1994).

Recent research has also studied ways to dynamically **adapt** language models to different genres; see Sec. 4.9.4.

4.3.2 Unknown Words: Open versus closed vocabulary tasks

Closed vocabulary

Sometimes we have a language task in which we know all the words that can occur, and hence we know the vocabulary size V in advance. The **closed vocabulary** assumption is the assumption that we have such a lexicon, and that the test set can only contain words from this lexicon. The closed vocabulary task thus assumes there are no unknown words.

But of course this is a simplification; as we suggested earlier, the number of unseen words grows constantly, so we can't possibly know in advance exactly how many there are, and we'd like our model to do something reasonable with them. We call these unseen events **unknown** words, or **out of vocabulary** (**OOV**) words. The percentage of OOV words that appear in the test set is called the **OOV rate**.

OOV

Open Vocabulary

An **open vocabulary** system is one where we model these potential unknown words in the test set by adding a pseudo-word called `<UNK>`. We can train the probabilities of the unknown word model `<UNK>` as follows:

1. **Choose a vocabulary** (word list) which 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.

4.4 Evaluating N-grams: Perplexity

The best way to evaluate the performance of a language model is to embed it in an application and measure the total performance of the application. Such end-to-end evaluation is called **extrinsic evaluation**, and also sometimes called **in vivo** evaluation (Sparck Jones and Galliers, 1996). Extrinisic 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, end-to-end evaluation is often very expensive; evaluating a large speech recognition test set, for example, takes hours or even days. Thus we would like a metric that can be used to quickly evaluate potential improvements in a language model. An **intrinsitic evaluation** metric is one which measures the quality of a model independent of any application. **Perplexity** is the most common intrinsic evaluation metric for N-gram language models. While an (intrinsic) improvement in perplexity does not guarantee an (extrinsic) improvement in speech recognition performance (or any other end-to-end metric), it often correlates with such improvements. Thus it is commonly used as a quick check on an algorithm and an improvement in perplexity can then be confirmed by an end-to-end evaluation.

Intrinsic evaluation

The intuition of perplexity is that given two probabilistic models, the better model is the one that has a tighter fit to the test data, or predicts the details of the test data better. We can measure better prediction by looking at the probability the model assigns to the test data; the better model will assign a higher probability to the test data.

Perplexity

More formally, the **perplexity** (sometimes called *PP* for short) of a language model on a test set is a function of the probability that the language model assigns to that test set. For a test set $W = w_1 w_2 \ldots w_N$, the perplexity is the probability of the test set, normalized by the number of words:

$$\text{PP}(W) = P(w_1w_2\ldots w_N)^{-\frac{1}{N}} \tag{4.16}$$
$$= \sqrt[N]{\frac{1}{P(w_1w_2\ldots w_N})}$$

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

$$\text{PP}(W) = \sqrt[N]{\prod_{i=1}^{N}\frac{1}{P(w_i|w_1\ldots w_{i-1})}} \tag{4.17}$$

Thus if we are computing the perplexity of W with a bigram language model, we get:

$$\text{PP}(W) = \sqrt[N]{\prod_{i=1}^{N}\frac{1}{P(w_i|w_{i-1})}} \tag{4.18}$$

Note that because of the inverse in Eq. 4.17, 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. 4.17 or Eq. 4.18 is the entire sequence of words in some test set. Since of course 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 each of the 10 digits occur with equal probability $P = \frac{1}{10}$. The perplexity of this mini-language is in fact 10. To see that, imagine a string of digits of length N. By Eq. 4.17, the perplexity will be:

$$\begin{aligned}\text{PP}(W) &= P(w_1w_2\ldots w_N)^{-\frac{1}{N}}\\ &= (\frac{1}{10}^{N})^{-\frac{1}{N}}\\ &= \frac{1}{10}^{-1}\\ &= 10\end{aligned} \tag{4.19}$$

But now suppose that the number zero is really frequent and occurs 10 times more often than other numbers. Now we should expect the perplexity to be lower, since most of the time the next number will be zero. Thus although the branching factor is still 10, the perplexity or weighted branching factor is smaller. We leave this calculation as an exercise to the reader.

We'll see in Sec. 4.10 that perplexity is also closely related to the information-theoretic notion of entropy.

Finally, let's see an example of how perplexity can be used to compare three N-gram models. 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.[5] We then computed the perplexity of each of these models on a test set of 1.5 million words via Eq. 4.18. 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 lower the perplexity (since as Eq. 4.17 showed, perplexity is related inversely to the likelihood of the test sequence according to the model).

Note that in computing perplexities the N-gram model P must be constructed without any knowledge of the test set. Any kind of knowledge of the test set can cause the perplexity to be artificially low. For example, we defined above the **closed vocabulary** task, in which the vocabulary for the test set is specified in advance. This can greatly reduce the perplexity. As long as this knowledge is provided equally to each of the models we are comparing, the closed vocabulary perplexity can still be useful for comparing models, but care must be taken in interpreting the results. In general, the perplexity of two language models is only comparable if they use the same vocabulary.

4.5 Smoothing

Never do I ever want
to hear another word!
There isn't one,
I haven't heard!
Eliza Doolittle in Alan Jay Lerner's *My Fair Lady*

Sparse data

There is a major problem with the maximum likelihood estimation process we have seen for training the parameters of an N-gram model. This is the problem of **sparse data** caused by the fact that our maximum likelihood estimate was based on a particular set of training data. 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. This

[5] More specifically, Katz-style backoff grammars with Good-Turing discounting trained on 38 million words from the WSJ0 corpus (LDC, 1993), open-vocabulary, using the <UNK> token; see later sections for definitions.

missing data means that the N-gram matrix for any given training corpus is bound to have a very large number of cases of putative "zero probability N-grams" that should really have some non-zero probability. Furthermore, the MLE method also produces poor estimates when the counts are non-zero but still small.

We need a method which can help get better estimates for these zero or low-frequency counts. Zero counts turn out to cause another huge problem. The **perplexity** metric defined above requires that we compute the probability of each test sentence. But if a test sentence has an N-gram that never appeared in the training set, the Maximum Likelihood estimate of the probability for this N-gram, and hence for the whole test sentence, will be zero! This means that in order to evaluate our language models, we need to modify the MLE method to assign some non-zero probability to any N-gram, even one that was never observed in training.

For these reasons, we'll want to modify the maximum likelihood estimates for computing N-gram probabilities, focusing on the N-gram events that we incorrectly assumed had zero probability. We use the term **smoothing** for such modifications that address the poor estimates that are due to variability in small data sets. The name comes from the fact that (looking ahead a bit) we will be shaving a little bit of probability mass from the higher counts, and piling it instead on the zero counts, making the distribution a little less jagged.

Smoothing

In the next few sections we will introduce some smoothing algorithms and show how they modify the Berkeley Restaurant bigram probabilities in Fig. 4.2.

4.5.1 Laplace Smoothing

One simple way to do smoothing might be just to take our matrix of bigram counts, before we normalize them into probabilities, and add one to all the counts. This algorithm is called **Laplace smoothing**, or Laplace's Law (Lidstone, 1920; Johnson, 1932; Jeffreys, 1948). Laplace smoothing does not perform well enough to be used in modern N-gram models, but we begin with it because it introduces many of the concepts that we will see in other smoothing algorithms and also gives us a useful baseline.

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 got 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

$$P_{\text{Laplace}}(w_i) = \frac{c_i + 1}{N + V} \tag{4.20}$$

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 one we'll also need to multiply by a normalization factor $\frac{N}{N+V}$:

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

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

Discount

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

Now that we have the intuition for the unigram case, let's smooth our Berkeley Restaurant Project bigrams. Fig. 4.5 shows the add-one smoothed counts for the bigrams in Fig. 4.1.

	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 4.5 Add-one smoothed bigram counts for eight of the words (out of $V = 1446$) in the Berkeley Restaurant Project corpus of 9332 sentences.

Fig. 4.6 shows the add-one smoothed probabilities for the bigrams in Fig. 4.2. Recall that normal bigram probabilities are computed by normalizing each row of counts by the unigram count:

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

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

(4.23) $$P^*_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V}$$

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. 4.6.

	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 4.6 Add-one smoothed bigram probabilities for eight of the words (out of $V = 1446$) in the BeRP corpus of 9332 sentences.

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. 4.24. Fig. 4.7 shows the reconstructed counts.

$$c^*(w_{n-1}w_n) = \frac{[C(w_{n-1}w_n)+1] \times C(w_{n-1})}{C(w_{n-1})+V} \tag{4.24}$$

	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 4.7 Add-one reconstituted counts for eight words (of $V = 1446$) in the BeRP corpus of 9332 sentences.

Note that add-one smoothing has made a very big change to the counts. $C(\textit{want to})$ 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 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. We could move a bit less mass by adding a fractional count rather than 1 (add-δ smoothing; (Lidstone, 1920; Johnson, 1932; Jeffreys, 1948)), but this method requires a method for choosing δ dynamically, results in an inappropriate discount for many counts, and turns out to give counts with poor variances. For these and other reasons (Gale and Church, 1994), we'll need better smoothing methods for N-grams like the ones we'll see in the next section.

4.5.2 Good-Turing Discounting

Good-Turing

There are a number of much better discounting algorithms that are only slightly more complex than add-one smoothing. In this section we introduce one of them, known as **Good-Turing** smoothing. The Good-Turing algorithm was first described by Good (1953), who credits Turing with the original idea.

The intuition of a number of discounting algorithms (Good-Turing, **Witten-Bell discounting**, and **Kneyser-Ney smoothing**) is to use the count of things you've seen *once* to help estimate the count of things you've *never seen*. A word or N-gram (or any event) that occurs once is called a **singleton**, or a **hapax legomenon**. The Good-Turing intuition is to use the frequency of singletons as a re-estimate of the frequency of zero-count bigrams.

Singleton

Let's formalize the algorithm. The Good-Turing algorithm is based on computing N_c, the number of N-grams that occur c times. We refer to the number of N-grams that occur c times as the **frequency of frequency** c. So applying the idea to smoothing the joint probability of bigrams, N_0 is the number of bigrams with count 0, N_1 the number of bigrams with count 1 (singletons), and so on. We can think of each of the N_c as a bin which stores the number of different N-grams that occur in the training set with that frequency c. More formally:

$$N_c = \sum_{x:count(x)=c} 1 \tag{4.25}$$

The MLE count for N_c is c. The Good-Turing intuition is to estimate the probability of things that occur c times in the training corpus by the MLE probability of things that occur $c+1$ times in the corpus. So the Good-Turing estimate replaces the MLE count c for N_c with a smoothed count c^* that is a function of N_{c+1}:

$$c^* = (c+1)\frac{N_{c+1}}{N_c} \tag{4.26}$$

We can use Eq. 4.26 to replace the MLE counts for all the bins N_1, N_2, and so on. Instead of using this equation directly to re-estimate the smoothed count c^* for N_0, we use the following equation for the probability P^*_{GT} for things that had zero count N_0, or what we might call the **missing mass**:

$$P^*_{GT}(\text{things with frequency zero in training}) = \frac{N_1}{N} \tag{4.27}$$

Here N_1 is the count of items in bin 1, i.e. that were seen once in training, and N is the total number of items we have seen in training. Eq. 4.27 thus gives the probability that the $N+1$st bigram we see will be one that we never saw in training. Showing that Eq. 4.27 follows from Eq. 4.26 is left as Exercise 8 for the reader.

The Good-Turing method was first proposed for estimating the populations of animal species. Let's consider an illustrative example from this domain created by Joshua Goodman and Stanley Chen. Suppose we are fishing in a lake with 8 species (bass, carp, catfish, eel, perch, salmon, trout, whitefish) and we have seen 6 species with the following counts: 10 carp, 3 perch, 2 whitefish, 1 trout, 1 salmon, and 1 eel (so we haven't yet seen the catfish or bass). What is the probability that the next fish we catch will be a new species, i.e., one that had a zero frequency in our training set, i.e., in this case either a catfish or a bass?

The MLE count c of a hitherto-unseen species (bass or catfish) is 0. But Eq. 4.27 tells us that the probability of a new fish being one of these unseen species is $\frac{3}{18}$, since N_1 is 3 and N is 18:

$$P^*_{GT}(\text{things with frequency zero in training}) = \frac{N_1}{N} = \frac{3}{18} \qquad (4.28)$$

What is the probability that the next fish will be another trout? The MLE count for trout is 1, so the MLE estimated probability is $\frac{1}{18}$. But the Good-Turing estimate must be lower, since we just stole $\frac{3}{18}$ of our probability mass to use on unseen events! We'll need to discount the MLE probabilities for trout, perch, carp, etc. In summary, the revised counts c^* and Good-Turing smoothed probabilities p^*_{GT} for species with count 0 (like bass or catfish) or count 1 (like trout, salmon, or eel) are as follows:

	unseen (bass or catfish)	**trout**
c	0	1
MLE p	$p = \frac{0}{18} = 0$	$\frac{1}{18}$
c*		$c^*(\text{trout}) = 2 \times \frac{N_2}{N_1} = 2 \times \frac{1}{3} = .67$
GT $\mathbf{p^*_{GT}}$	$p^*_{\text{GT}}(\text{unseen}) = \frac{N_1}{N} = \frac{3}{18} = .17$	$p^*_{\text{GT}}(\text{trout}) = \frac{.67}{18} = \frac{1}{27} = .037$

Note that the revised count c^* for eel was discounted from $c = 1.0$ to $c^* = .67$, (thus leaving some probability mass $p^*_{\text{GT}}(\text{unseen}) = \frac{3}{18} = .17$ for the catfish and bass). And since we know there were 2 unknown species, the probability of the next fish being specifically a catfish is $p^*_{\text{GT}}(\text{catfish}) = \frac{1}{2} \times \frac{3}{18} = .085$.

Fig. 4.8 gives two examples of the application of Good-Turing discounting to bigram grammars, one on the BeRP corpus of 9332 sentences, and a larger example computed from 22 million words from the Associated Press (AP) newswire by Church and Gale (1991) . For both examples the first column shows the count c, i.e., the number of observed instances of a bigram. The second column shows the number of bigrams that had this count. Thus 449,721 of the AP bigrams have a count of 2. The third column shows c^*, the Good-Turing re-estimation of the count.

4.5.3 Some advanced issues in Good-Turing estimation

Good-Turing estimation assumes that the distribution of each bigram is binomial (Church et al., 1991) and assumes we know N_0, the number of bigrams we haven't seen. We know this because given a vocabulary size of V, the total number of bigrams is V^2, hence N_0 is V^2 minus all the bigrams we have seen.

There are a number of additional complexities in the use of Good-Turing. For example, we don't just use the raw N_c values in Eq. 4.26. This is because the re-estimate c^* for N_c depends on N_{c+1}, hence Eq. 4.26 is undefined when $N_{c+1} = 0$. Such zeros occur quite often. In our sample problem above, for example, since $N_4 = 0$, how can we compute N_3? One solution to this is called **Simple Good-Turing** (Gale and

Simple Good-Turing

AP Newswire			Berkeley Restaurant		
c (MLE)	N_c	c^* (GT)	c (MLE)	N_c	c^* (GT)
0	74,671,100,000	0.0000270	0	2,081,496	0.002553
1	2,018,046	0.446	1	5315	0.533960
2	449,721	1.26	2	1419	1.357294
3	188,933	2.24	3	642	2.373832
4	105,668	3.24	4	381	4.081365
5	68,379	4.22	5	311	3.781350
6	48,190	5.19	6	196	4.500000

Figure 4.8 Bigram "frequencies of frequencies" and Good-Turing re-estimations for the 22 million AP bigrams from Church and Gale (1991) and from the Berkeley Restaurant corpus of 9332 sentences.

Sampson, 1995). In Simple Good-Turing, after we compute the bins N_c, but before we compute Eq. 4.26 from them, we smooth the N_c counts to replace any zeros in the sequence. The simplest thing is just to replace the value N_c with a value computed from a linear regression which is fit to map N_c to c in log space (see Gale and Sampson (1995) for details):

$$\log(N_c) = a + b\,\log(c) \tag{4.29}$$

In addition, in practice, the discounted estimate c^* is not used for all counts c. Large counts (where $c > k$ for some threshold k) are assumed to be reliable. Katz (1987) suggests setting k at 5. Thus we define

$$c^* = c \ \text{ for } c > k \tag{4.30}$$

The correct equation for c^* when some k is introduced (from Katz (1987)) is:

$$c^* = \frac{(c+1)\frac{N_{c+1}}{N_c} - c\frac{(k+1)N_{k+1}}{N_1}}{1 - \frac{(k+1)N_{k+1}}{N_1}}, \ \text{ for } 1 \le c \le k. \tag{4.31}$$

Second, with Good-Turing discounting as with any other, it is usual to treat N-grams with low raw counts (especially counts of 1) as if the count were 0, i.e., to apply Good-Turing discounting to these as if they were unseen.

It turns out that Good-Turing discounting is not used by itself in discounting N-grams; it is only used in combination with the backoff and interpolation algorithms described in the next sections.

4.6 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 $P(w_n)$.

Backoff
Interpolation

There are two ways to use this N-gram "hierarchy", **backoff** and **interpolation**. In backoff, if we have non-zero trigram counts, we rely solely on the trigram counts. 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, i.e., we do a weighted interpolation of trigram, bigram, and unigram counts.

In simple linear interpolation, we combine different order N-grams by linearly interpolating all the models. 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:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1 P(w_n|w_{n-2}w_{n-1}) \\ &\quad +\lambda_2 P(w_n|w_{n-1}) \\ &\quad +\lambda_3 P(w_n)\end{aligned} \tag{4.32}$$

such that the λs sum to 1:

$$\sum_i \lambda_i = 1 \tag{4.33}$$

In a slightly more sophisticated version of linear interpolation, each λ weight is computed in a more sophisticated way, 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. Eq. 4.34 shows the equation for interpolation with context-conditioned weights:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1(w_{n-2}^{n-1})P(w_n|w_{n-2}w_{n-1}) \\ &\quad +\lambda_2(w_{n-2}^{n-1})P(w_n|w_{n-1}) \\ &\quad +\lambda_3(w_{n-2}^{n-1})P(w_n)\end{aligned} \tag{4.34}$$

Held-out

How are these λ values set? Both the simple interpolation and conditional interpolation λs are learned from a **held-out** corpus. Recall from Sec. 4.3 that a held-out corpus is an additional training corpus that we use not to set the N-gram counts, but to set other parameters. In this case we can use such data to set the λ values. We can do this by choosing the λ values which 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. 4.32 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 to be defined in Ch. 6, which is an iterative learning algorithm that converges on locally optimal λs (Baum, 1972; Dempster et al., 1977; Jelinek and Mercer, 1980).

4.7 Backoff

While simple interpolation is indeed simple to understand and implement, it turns out that there are a number of better algorithms. One of these is backoff N-gram modeling. The version of backoff that we describe uses Good-Turing discounting as well. It was introduced by Katz (1987), hence this kind of backoff with discounting is also called *Katz backoff* **Katz backoff**. In a Katz 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:

$$P_{\text{katz}}(w_n|w_{n-N+1}^{n-1}) = \begin{cases} P^*(w_n|w_{n-N+1}^{n-1}), & \text{if } C(w_{n-N+1}^{n}) > 0 \\ \alpha(w_{n-N+1}^{n-1})P_{\text{katz}}(w_n|w_{n-N+2}^{n-1}), & \text{otherwise.} \end{cases} \tag{4.35}$$

Equation 4.35 shows that the Katz backoff probability for an N-gram just relies on the (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. We'll define the discounted probability P^*, the normalizing factor α, and other details about dealing with zero counts in Sec. 4.7.1. Based on these details, the trigram version of backoff might be represented as follows (where for pedagogical clarity, since it's easy to confuse the indices w_i, w_{i-1} and so on, we refer to the three words in a sequence as x, y, z in that order):

$$P_{\text{katz}}(z|x,y) = \begin{cases} P^*(z|x,y), & \text{if } C(x,y,z) > 0 \\ \alpha(x,y)P_{\text{katz}}(z|y), & \text{else if } C(x,y) > 0 \\ P^*(z), & \text{otherwise.} \end{cases} \tag{4.36}$$

$$P_{\text{katz}}(z|y) = \begin{cases} P^*(z|y), & \text{if } C(y,z) > 0 \\ \alpha(y)P^*(z), & \text{otherwise.} \end{cases} \tag{4.37}$$

Katz backoff incorporates discounting as an integral part of the algorithm. Our previous discussions of discounting showed how a method like Good-Turing could be used to assign probability mass to unseen events. For simplicity, we assumed that these unseen events were all equally probable, and so the probability mass got distributed evenly among all unseen events. Katz backoff gives us a better way to distribute the probability mass among unseen trigram events, by relying on information from unigrams and bigrams. We use discounting to tell us how much total probability mass to set aside for all the events we haven't seen and backoff to tell us how to distribute this probability.

Discounting is implemented by using discounted probabilities $P^*(\cdot)$ rather than MLE probabilities $P(\cdot)$ in Eq. 4.35 and Eq. 4.37.

Why do we need discounts and α values in Eq. 4.35 and Eq. 4.37? Why couldn't we just have three sets of MLE probabilities without weights? Because without discounts and α weights, the result of the equation would not be a true probability! The MLE estimates of $P(w_n|w_{n-N+1}^{n-1})$ are true probabilities; if we sum the probability of all w_i over a given N-gram context, we should get 1:

$$\sum_i P(w_i|w_j w_k) = 1 \tag{4.38}$$

But if that is the case, if we use MLE probabilities but back off to a lower order model when the MLE probability is zero, we would be adding extra probability mass into the equation, and the total probability of a word would be greater than 1!

Thus any backoff language model must also be discounted. The P^* is used to discount the MLE probabilities to save some probability mass for the lower order N-grams. The α is used to ensure that the probability mass from all the lower order N-grams sums up to exactly the amount that we saved by discounting the higher-order N-grams. We define P^* as the discounted (c^*) estimate of the conditional probability of an N-gram, (and save P for MLE probabilities):

$$P^*(w_n|w_{n-N+1}^{n-1}) = \frac{c^*(w_{n-N+1}^{n})}{c(w_{n-N+1}^{n-1})} \tag{4.39}$$

Because on average the (discounted) c^* will be less than c, this probability P^* will be slightly less than the MLE estimate, which is

$$\frac{c(w_{n-N+1}^{n})}{c(w_{n-N+1}^{n-1})}$$

This will leave some probability mass for the lower order N-grams which is then distributed by the α weights; details of computing α are in Sec. 4.7.1. Fig. 4.7 shows the Katz backoff bigram probabilities for our 8 sample words, computed from the BeRP corpus using the SRILM toolkit.

	i	**want**	**to**	**eat**	**chinese**	**food**	**lunch**	**spend**
i	0.0014	0.326	0.00248	0.00355	0.000205	0.0017	0.00073	0.000489
want	0.00134	0.00152	0.656	0.000483	0.00455	0.00455	0.00384	0.000483
to	0.000512	0.00152	0.00165	0.284	0.000512	0.0017	0.00175	0.0873
eat	0.00101	0.00152	0.00166	0.00189	0.0214	0.00166	0.0563	0.000585
chinese	0.00283	0.00152	0.00248	0.00189	0.000205	0.519	0.00283	0.000585
food	0.0137	0.00152	0.0137	0.00189	0.000409	0.00366	0.00073	0.000585
lunch	0.00363	0.00152	0.00248	0.00189	0.000205	0.00131	0.00073	0.000585
spend	0.00161	0.00152	0.00161	0.00189	0.000205	0.0017	0.00073	0.000585

Figure 4.9 Good-Turing smoothed bigram probabilities for eight words (of $V = 1446$) in the BeRP corpus of 9332 sentences, computing by using SRILM, with $k = 5$ and counts of 1 replaced by 0.

4.7.1 Advanced: Details of computing Katz backoff α and P^*

In this section we give the remaining details of the computation of the discounted probability P^* and the backoff weights $\alpha(w)$.

We begin with α, which passes the left-over probability mass to the lower order N-grams. Let's represent the total amount of left-over probability mass by the function

β, a function of the (N-1)-gram context. For a given (N-1)-gram context, the total left-over probability mass can be computed by subtracting from 1 the total discounted probability mass for all N-grams starting with that context:

(4.40)
$$\beta(w_{n-N+1}^{n-1}) = 1 - \sum_{w_n:c(w_{n-N+1}^{n})>0} P^*(w_n|w_{n-N+1}^{n-1})$$

This gives us the total probability mass that we are ready to distribute to all (N-1)-gram (e.g., bigrams if our original model was a trigram). Each individual (N-1)-gram (bigram) will only get a fraction of this mass, so we need to normalize β by the total probability of all the (N-1)-grams (bigrams) that begin some N-gram (trigram) which has zero count. The final equation for computing how much probability mass to distribute from an N-gram to an (N-1)-gram is represented by the function α:

(4.41)
$$\begin{aligned}\alpha(w_{n-N+1}^{n-1}) &= \frac{\beta(w_{n-N+1}^{n-1})}{\sum_{w_n:c(w_{n-N+1}^{n})=0} P_{\text{katz}}(w_n|w_{n-N+2}^{n-1})}\\ &= \frac{1-\sum_{w_n:c(w_{n-N+1}^{n})>0} P^*(w_n|w_{n-N+1}^{n-1})}{1-\sum_{w_n:c(w_{n-N+1}^{n})>0} P^*(w_n|w_{n-N+2}^{n-1})}\end{aligned}$$

Note that α is a function of the preceding word string, that is, of w_{n-N+1}^{n-1}; thus the amount by which we discount each trigram (d), and the mass that gets reassigned to lower order N-grams (α) are recomputed for every (N-1)-gram that occurs in any N-gram.

We only need to specify what to do when the counts of an (N-1)-gram context are 0, (i.e., when $c(w_{n-N+1}^{n-1}) = 0$) and our definition is complete:

(4.42)
$$P_{\text{katz}}(w_n|w_{n-N+1}^{n-1}) = P_{\text{katz}}(w_n|w_{n-N+2}^{n-1}) \qquad \text{if } c(w_{n-N+1}^{n-1}) = 0$$

and

(4.43)
$$P^*(w_n|w_{n-N+1}^{n-1}) = 0 \qquad \text{if } c(w_{n-N+1}^{n-1}) = 0$$

and

(4.44)
$$\beta(w_{n-N+1}^{n-1}) = 1 \qquad \text{if } c(w_{n-N+1}^{n-1}) = 0$$

4.8 Practical Issues: Toolkits and Data Formats

Let's now examine how N-gram language models are represented. We represent and compute language model probabilities in log format, in order to avoid underflow and also to speed up computation. Since probabilities are (by definition) less than 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, the numbers are not as small. Since adding in log space is equivalent to multiplying in linear space, we combine log probabilities by adding them. Besides avoiding underflow, addition is faster to compute than multiplication. Since we do all computation and storage in log space, if we ever need to report probabilities we just take the exp of the logprob:

$$p_1 \times p_2 \times p_3 \times p_4 = \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4) \tag{4.45}$$

Backoff N-gram language models are generally stored in **ARPA format**. An N-gram in ARPA format is an ASCII file with a small header followed by a list of all the non-zero N-gram probabilities (all the unigrams, followed by bigrams, followed by trigrams, and so on). Each N-gram entry is stored with its discounted log probability (in $\log_{10}$ format) and its backoff weight α. Backoff weights are only necessary for N-grams which form a prefix of a longer N-gram, so no α is computed for the highest order N-gram (in this case the trigram) or N-grams ending in the end-of-sequence token `<s>`. Thus for a trigram grammar, the format of each N-gram is:

unigram:	$\log p^*(w_i)$	w_i	$\log \alpha(w_i)$
bigram:	$\log p^*(w_i \mid w_{i-1})$	$w_{i-1} w_i$	$\log \alpha(w_{i-1} w_i)$
trigram:	$\log p^*(w_i \mid w_{i-2}, w_{i-1})$	$w_{i-2} w_{i-1} w_i$	

```
\data\
ngram 1=1447
ngram 2=9420
ngram 3=5201

\1-grams:\
-0.8679678     </s>
-99            <s>                          -1.068532
-4.743076      chow-fun                     -0.1943932
-4.266155      fries                        -0.5432462
-3.175167      thursday                     -0.7510199
-1.776296      want                         -1.04292
...

\2-grams:\
-0.6077676     <s>    i                     -0.6257131
-0.4861297     i      want                  0.0425899
-2.832415      to     drink                 -0.06423882
-0.5469525     to     eat                   -0.008193135
-0.09403705    today  </s>
...

\3-grams:\
-2.579416      <s>    i          prefer
-1.148009      <s>    about      fifteen
-0.4120701     to     go         to
-0.3735807     me     a          list
-0.260361      at     jupiter    </s>
-0.260361      a      malaysian  restaurant
...
\end\
```

Figure 4.10 ARPA format for N-grams, showing some sample N-grams. Each is represented by a *logprob*, the word sequence, $w_1 \dots w_n$, followed by the log backoff weight α. Note that no α is computed for the highest-order N-gram or for N-grams ending in `<s>`.

Fig. 4.8 shows an ARPA formatted LM file with selected N-grams from the BeRP corpus. Given one of these trigrams, the probability $P(z|x,y)$ for the word sequence x, y, z can be computed as follows (repeated from (4.37)):

$$P_{\text{katz}}(z|x,y) = \begin{cases} P^*(z|x,y), & \text{if } C(x,y,z) > 0 \\ \alpha(x,y)P_{\text{katz}}(z|y), & \text{else if } C(x,y) > 0 \\ P^*(z), & \text{otherwise.} \end{cases} \tag{4.46}$$

$$P_{\text{katz}}(z|y) = \begin{cases} P^*(z|y), & \text{if } C(y,z) > 0 \\ \alpha(y)P^*(z), & \text{otherwise.} \end{cases} \tag{4.47}$$

Toolkits: There are two commonly used available toolkits for building language models, the SRILM toolkit (Stolcke, 2002) and the Cambridge-CMU toolkit (Clarkson and Rosenfeld, 1997). Both are publicly available, and have similar functionality. In training mode, each toolkit takes a raw text file, one sentence per line with words separated by white-space, and various parameters such as the order N, the type of discounting (Good-Turing or Kneser-Ney, discussed in Sec. 4.9.1), and various thresholds. The output is a language model in ARPA format. In perplexity or decoding mode, the toolkits take a language model in ARPA format, and a sentence or corpus, and produce the probability and perplexity of the sentence or corpus. Both also implement many advanced features to be discussed later in this chapter and in following chapters, including skip N-grams, word lattices, confusion networks, and N-gram pruning.

4.9 Advanced Issues in Language Modeling

4.9.1 Advanced Smoothing Methods: Kneser-Ney Smoothing

In this section we give a brief introduction to the most commonly used modern N-gram smoothing method, the interpolated **Kneser-Ney** algorithm.

Kneser-Ney

Kneser-Ney has its roots in a discounting method called **absolute discounting**. Absolute discounting is a much better method of computing a revised count $c*$ than the Good-Turing discount formula we saw in Eq. 4.26, based on frequencies-of-frequencies. To get the intuition, let's revisit the Good-Turing estimates of the bigram c^* extended from Fig. 4.8 and reformatted below:

c (MLE)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
c^* (GT)	0.0000270	0.446	1.26	2.24	3.24	4.22	5.19	6.21	7.24	8.25

Absolute discounting

The astute reader may have noticed that except for the re-estimated counts for 0 and 1, all the other re-estimated counts c^* could be estimated pretty well by just subtracting 0.75 from the MLE count c! **Absolute discounting** formalizes this intuition, by subtracting a fixed (absolute) discount d from each count. The intuition is that we have good estimates already for the high counts, and 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. The equation for absolute discounting applied to bigrams (assuming a proper coefficient α on the backoff to make everything sum to one) is:

$$P_{\text{absolute}}(w_i|w_{i-1}) = \begin{cases} \frac{C(w_{i-1}w_i)-\mathbf{D}}{C(w_{i-1})}, & \text{if } C(w_{i-1}w_i) > 0 \\ \alpha(w_i)P(w_i), & \text{otherwise.} \end{cases} \tag{4.48}$$

In practice, we might also want to keep distinct discount values D for the 0 and 1 counts.

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

I can't see without my reading ________________ .

The word *glasses* seems much more likely to follow here than the word *Francisco*. But *Francisco* is in fact more common, so a unigram model will prefer it to *glasses*. We would like to capture the intuition that although *Francisco* is frequent, it is only frequent after the word *San*, i.e. in the phrase *San Francisco*. The word *glasses* has a much wider distribution.

Thus instead of backing off to the unigram MLE count (the number of times the word w has been seen), we want to use a completely different backoff distribution! We want a heuristic that more accurately estimates the number of times we might expect to see word w in a new unseen context. The Kneser-Ney intuition is to base our estimate on the *number of different contexts word w has appeared in*. Words that have appeared in more contexts are more likely to appear in some new context as well. We can express this new backoff probability, the "continuation probability", as follows:

$$P_{\text{CONTINUATION}}(w_i) = \frac{|\{w_{i-1} : C(w_{i-1}w_i) > 0\}|}{\sum_{w_i} |\{w_{i-1} : C(w_{i-1}w_i) > 0\}|} \tag{4.49}$$

The Kneser-Ney backoff intuition can be formalized as follows (again assuming a proper coefficient α on the backoff to make everything sum to one):

$$P_{\text{KN}}(w_i|w_{i-1}) = \begin{cases} \frac{C(w_{i-1}w_i)-\mathbf{D}}{C(w_{i-1})}, & \text{if } C(w_{i-1}w_i) > 0 \\ \alpha(w_i)\frac{|\{w_{i-1}:C(w_{i-1}w_i)>0\}|}{\sum_{w_i}|\{w_{i-1}:C(w_{i-1}w_i)>0\}|} & \text{otherwise.} \end{cases} \tag{4.50}$$

Finally, it turns out to be better to use an **interpolated** rather than **backoff** form of Kneser-Ney. While simple *linear* interpolation is generally not as successful as Katz backoff, it turns out that more powerful interpolated models, such as interpolated Kneser-Ney, work better than their backoff version. **Interpolated Kneser-Ney** discounting can be computed with an equation like the following (omitting the computation of β):

Interpolated Kneser-Ney

$$P_{\text{KN}}(w_i|w_{i-1}) = \frac{C(w_{i-1}w_i)-\mathbf{D}}{C(w_{i-1})} + \beta(w_i)\frac{|\{w_{i-1} : C(w_{i-1}w_i) > 0\}|}{\sum_{w_i}|\{w_{i-1} : C(w_{i-1}w_i) > 0\}|} \tag{4.51}$$

A final practical note: it turns out that any interpolation model can be represented as a backoff model, hence stored in ARPA backoff format. We simply do the interpolation when we build the model, so the 'bigram' probability stored in the backoff format is really 'bigram already interpolated with unigram'.

4.9.2 Class-based N-grams

Class-based N-gram
Cluster N-gram

The **class-based N-gram** or **cluster N-gram** is a variant of the N-gram that uses information about word classes or clusters. Class-based N-grams can be useful for dealing with sparsity in the training data. Suppose for a flight reservation system we want to compute the probability of the bigram *to Shanghai*, but this bigram never occurs in the training set. Instead, our training data has *to London*, *to Beijing*, and *to Denver*. If we knew that these were all cities, and assuming *Shanghai* does appear in the training set in other contexts, we could predict the likelihood of a city following *from*.

IBM clustering

There are many variants of cluster N-grams. The simplest one is sometimes known as **IBM clustering**, after its originators (Brown et al., 1992). IBM clustering is a kind of **hard clustering**, in which each word can belong to only one class. The model estimates the conditional probability of a word w_i by multiplying two factors: the probability of the word's class c_i given the preceding classes (based on an N-gram of classes), and the probability of w_i given c_i. Here is the IBM model in bigram form:

$$P(w_i|w_{i-1}) \approx P(c_i|c_{i-1}) \times P(w_i|c_i)$$

If we had a training corpus in which we knew the class for each word, the maximum likelihood estimate (MLE) of the probability of the word given the class and the probability of the class given the previous class could be computed as follows:

$$\begin{aligned} P(w|c) &= \frac{C(w)}{C(c)} \\ P(c_i|c_{i-1}) &= \frac{C(c_{i-1}c_i)}{\sum_c C(c_{i-1}c)} \end{aligned}$$

Cluster N-grams are generally used in two ways. In dialog systems (Ch. 24), we often hand-design domain-specific word classes. Thus for an airline information system, we might use classes like CITYNAME, AIRLINE, DAYOFWEEK, or MONTH. In other cases, we can automatically induce the classes by clustering words in a corpus (Brown et al., 1992). Syntactic categories like part-of-speech tags don't seem to work well as classes (Niesler et al., 1998).

Whether automatically induced or hand-designed, cluster N-grams are generally mixed with regular word-based N-grams.

4.9.3 Language Model Adaptation and Using the Web

Adaptation

One of the most exciting recent developments in language modeling is language model **adaptation**. This is relevant when we have only a small amount of in-domain training data, but a large amount of data from some other domain. We can train on the larger out-of-domain dataset and adapt our models to the small in-domain set. (Iyer and Ostendorf, 1997, 1999a, 1999b; Bacchiani and Roark, 2003; Bacchiani et al., 2004).

An obvious large data source for this type of adaptation is the web. Indeed, use of the web does seem to be helpful in language modeling. The simplest way to apply the web to improve, say, trigram language models is to use search engines to get counts for $w_1w_2w_3$ and $w_1w_2w_3$, and then compute:

$$\hat{p}_{web} = \frac{c_{web}(w_1w_2w_3)}{c_{web}(w_1w_2)} \tag{4.52}$$

We can then mix $\hat{p}_{web}$ with a conventional N-gram (Berger and Miller, 1998; Zhu and Rosenfeld, 2001). We can also use more sophisticated combination methods that make use of topic or class dependencies, to find domain-relevant data on the web data (Bulyko et al., 2003).

In practice it is difficult or impossible to download every page from the web in order to compute N-grams. For this reason most uses of web data rely on page counts from search engines. Page counts are only an approximation to actual counts for many reasons: a page may contain an N-gram multiple times, most search engines round off their counts, punctuation is deleted, and the counts themselves may be adjusted due to link and other information. It seems that this kind of noise does not hugely affect the results of using the web as a corpus (Keller and Lapata, 2003; Nakov and Hearst, 2005), although it is possible to perform specific adjustments, such as fitting a regression to predict actual word counts from page counts (Zhu and Rosenfeld, 2001).

4.9.4 Using Longer Distance Information: A Brief Summary

There are many methods for incorporating longer-distance context into N-gram modeling. While we have limited our discussion mainly to bigrams and trigrams, state-of-the-art speech recognition systems, for example, are based on longer-distance N-grams, especially 4-grams, but also 5-grams. Goodman (2006) showed that with 284 million words of training data, 5-grams do improve perplexity scores over 4-grams, but not by much. Goodman checked contexts up to 20-grams, and found that after 6-grams, longer contexts weren't useful, at least not with 284 million words of training data.

Many models focus on more sophisticated ways to get longer-distance information. For example people tend to repeat words they have used before. Thus if a word is used once in a text, it will probably be used again. We can capture this fact by a **cache** language model (Kuhn and De Mori, 1990). For example to use a unigram cache model to predict word i of a test corpus, we create a unigram grammar from the preceding part of the test corpus (words 1 to $i-1$) and mix this with our conventional N-gram. We might use only a shorter window from the previous words, rather than the entire set. Cache language models are very powerful in any applications where we have perfect knowledge of the words. Cache models work less well in domains where the previous words are not known exactly. In speech applications, for example, unless there is some way for users to correct errors, cache models tend to "lock in" errors they made on earlier words.

Cache

The fact that words are often repeated in a text is a symptom of a more general fact about texts; texts tend to be **about** things. Documents which are about particular topics tend to use similar words. This suggests that we could train separate language models for different topics. In **topic-based** language models (Chen et al., 1998; Gildea

Topic-based

and Hofmann, 1999), we try to take advantage of the fact that different topics will have different kinds of words. For example we can train different language models for each topic t, and then mix them, weighted by how likely each topic is given the history h:

$$p(w|h) = \sum_t P(w|t)P(t|h) \tag{4.53}$$

A very similar class of models relies on the intuition that upcoming words are semantically similar to preceding words in the text. These models use a measure of semantic word association such as the **latent semantic indexing** described in Ch. 20 (Coccaro and Jurafsky, 1998; Bellegarda, 1999, 2000), or on-line dictionaries or thesauri (Demetriou et al., 1997) to compute a probability based on a word's similarity to preceding words, and then mix it with a conventional N-gram.

Latent semantic indexing

There are also various ways to extend the N-gram model by having the previous (conditioning) word be something other than a fixed window of previous words. For example we can choose as a predictor a word called a **trigger** which is not adjacent but which is very related (has high mutual information with) the word we are trying to predict (Rosenfeld, 1996; Niesler and Woodland, 1999; Zhou and Lua, 1998). Or we can create **skip N-grams**, where the preceding context 'skips over' some intermediate words, for example computing a probability such as $P(w_i|w_{i-1}, w_{i-3})$. We can also use extra previous context just in cases where a longer phrase is particularly frequent or predictive, producing a **variable-length** N**-gram** (Ney et al., 1994; Kneser, 1996; Niesler and Woodland, 1996).

Trigger

Skip N-gram

Variable-length N-gram

In general, using very large and rich contexts can result in very large language models. Thus these models are often pruned by removing low-probability events. Pruning is also essential for using language models on small platforms such as cellphones (Stolcke, 1998; Church et al., 2007).

Finally, there is a wide body of research on integrating sophisticated linguistic structures into language modeling. Language models based on syntactic structure from probabilistic parsers are described in Ch. 14. Language models based on the current speech act in dialogue are described in Ch. 24.

4.10 Advanced: Information Theory Background

I got the horse right here
Frank Loesser, *Guys and Dolls*

We introduced perplexity in Sec. 4.4 as a way to evaluate N-gram models on a test set. A better N-gram model is one which assigns a higher probability to the test data, and perplexity is a normalized version of the probability of the test set. Another way to think about perplexity is based on the information-theoretic concept of cross-entropy. In order to give another intuition into perplexity as a metric, this section gives a quick review of fundamental facts from **information theory** including the concept of cross-entropy that underlies perplexity. The interested reader should consult a good information theory textbook like Cover and Thomas (1991).

Entropy

Perplexity is based on the information-theoretic notion of **cross-entropy**, which we will now work toward defining. **Entropy** is a measure of information, and is invaluable throughout speech and language processing. It can be used as a metric for how much information there is in a particular grammar, for how well a given grammar matches a given language, for how predictive a given N-gram grammar is about what the next word could be. Given two grammars and a corpus, we can use entropy to tell us which grammar better matches the corpus. We can also use entropy to compare how difficult two speech recognition tasks are, and also to measure how well a given probabilistic grammar matches human grammars.

Computing entropy requires that we establish a random variable X that ranges over whatever we are predicting (words, letters, parts of speech, the set of which we'll call χ), and that has a particular probability function, call it $p(x)$. The entropy of this random variable X is then

$$H(X) = -\sum_{x\in\chi} p(x)\log_2 p(x) \tag{4.54}$$

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**.

The most intuitive way to define entropy for computer scientists is to think of the entropy 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.

Cover and Thomas (1991) suggest the following example. 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, and we'd like to send a short message to the bookie to tell him which horse to bet on. Suppose there are eight horses in this particular race.

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:

$$\begin{aligned} H(X) &= -\sum_{i=1}^{i=8} p(i)\log p(i) \\ &= -\tfrac{1}{2}\log\tfrac{1}{2} - \tfrac{1}{4}\log\tfrac{1}{4} - \tfrac{1}{8}\log\tfrac{1}{8} - \tfrac{1}{16}\log\tfrac{1}{16} - 4(\tfrac{1}{64}\log\tfrac{1}{64}) \\ &= 2 \text{ bits} \end{aligned} \tag{4.55}$$

A code that averages 2 bits per race can be built by using 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:

$$H(X) = -\sum_{i=1}^{i=8} \frac{1}{8}\log\frac{1}{8} = -\log\frac{1}{8} = 3 \text{ bits} \tag{4.56}$$

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 $W = \{w_0, w_1, w_2, \ldots, w_n\}$. 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:

$$H(w_1, w_2, \ldots, w_n) = -\sum_{W_1^n \in L} p(W_1^n)\log p(W_1^n) \tag{4.57}$$

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:

$$\frac{1}{n}H(W_1^n) = -\frac{1}{n}\sum_{W_1^n \in L} p(W_1^n)\log p(W_1^n) \tag{4.58}$$

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, its entropy rate $H(L)$ is defined as:

$$\begin{aligned} H(L) &= -\lim_{n\to\infty}\frac{1}{n}H(w_1, w_2, \ldots, w_n) \\ &= -\lim_{n\to\infty}\frac{1}{n}\sum_{W\in L} p(w_1, \ldots, w_n)\log p(w_1, \ldots, w_n) \end{aligned} \tag{4.59}$$

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),

$$H(L) = \lim_{n\to\infty} -\frac{1}{n}\log p(w_1 w_2 \ldots w_n) \tag{4.60}$$

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 will see in Ch. 12, 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. In the next section we talk about the why and how: *why* we would want to do this (i.e., for what kinds of problems would the entropy tell us something useful), and *how* to compute the probability of a very long sequence.

4.10.1 Cross-Entropy for Comparing Models

Cross-entropy

In this section we introduce **cross-entropy**, and discuss its usefulness in comparing different probabilistic models. 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:

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n} \sum_{W\in L} p(w_1,\ldots,w_n)\log m(w_1,\ldots,w_n) \tag{4.61}$$

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:

$$H(p,m) = \lim_{n\to\infty} -\frac{1}{n}\log m(w_1 w_2 \ldots w_n) \tag{4.62}$$

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:

$$H(p) \leq H(p,m) \tag{4.63}$$

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. 4.62. 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 $M = P(w_i|w_{i-N+1}...w_{i-1})$ on a sequence of words W is:

$$H(W) = -\frac{1}{N}\log P(w_1 w_2 \ldots w_N) \tag{4.64}$$

Perplexity

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

$$\begin{aligned}
\text{Perplexity}(W) &= 2^{H(W)} \\
&= P(w_1 w_2 \ldots w_N)^{-\frac{1}{N}} \\
&= \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}} \\
&= \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}} \qquad (4.65)
\end{aligned}$$

4.11 Advanced: The Entropy of English and Entropy Rate Constancy

As we suggested in the previous section, the cross-entropy of some model m can be used as an upper bound on the true entropy of some process. We can use this method to get an estimate of the true entropy of English. Why should we care about the entropy of English?

One reason is that the true entropy of English would give us a solid lower bound for all of our future experiments on probabilistic grammars. Another is that we can use the entropy values for English to help understand what parts of a language provide the most information (for example, is the predictability of English mainly based on word order, on semantics, on morphology, on constituency, or on pragmatic cues?) This can help us immensely in knowing where to focus our language-modeling efforts.

There are two common methods for computing the entropy of English. The first was employed by Shannon (1951), as part of his groundbreaking work in defining the field of information theory. His idea was to use human subjects, and to construct a psychological experiment that requires them to guess strings of letters. By looking at how many guesses it takes them to guess letters correctly we can estimate the probability of the letters, and hence the entropy of the sequence.

The actual experiment is designed as follows: we present a subject with some English text and ask the subject to guess the next letter. The subjects will use their knowledge of the language to guess the most probable letter first, the next most probable next, and so on. We record the number of guesses it takes for the subject to guess correctly. Shannon's insight was that the entropy of the number-of-guesses sequence is the same as the entropy of English. (The intuition is that given the number-of-guesses sequence, we could reconstruct the original text by choosing the "*n*th most probable" letter whenever the subject took *n* guesses). This methodology requires the use of letter guesses rather than word guesses (since the subject sometimes has to do an exhaustive search of all the possible letters!), so Shannon computed the **per-letter entropy** of English rather than the per-word entropy. He reported an entropy of 1.3 bits (for 27 characters (26 letters plus space)). Shannon's estimate is likely to be too low, since it is based on a single text (*Jefferson the Virginian* by Dumas Malone). Shannon notes that his subjects had worse guesses (hence higher entropies) on other texts (newspaper writing, scientific work, and poetry). More recent variations on the Shannon experiments include the use of a gambling paradigm where the subjects get to bet on the next letter (Cover and King, 1978; Cover and Thomas, 1991).

The second method for computing the entropy of English helps avoid the single-text problem that confounds Shannon's results. This method is to take a very good stochastic model, train it on a very large corpus, and use it to assign a log-probability to a very long sequence of English, using the Shannon-McMillan-Breiman theorem:

(4.66)
$$H(\text{English}) \leq \lim_{n\to\infty} -\frac{1}{n}\log m(w_1 w_2 \ldots w_n)$$

For example, Brown et al. (1992) trained a trigram language model on 583 million words of English (293,181 different types) and used it to compute the probability of the entire Brown corpus (1,014,312 tokens). The training data include newspapers, encyclopedias, novels, office correspondence, proceedings of the Canadian parliament, and other miscellaneous sources.

They then computed the character entropy of the Brown corpus by using their word-trigram grammar to assign probabilities to the Brown corpus, considered as a sequence of individual letters. They obtained an entropy of 1.75 bits per character (where the set of characters included all the 95 printable ASCII characters).

The average length of English written words (including space) has been reported at 5.5 letters (Nádas, 1984). If this is correct, it means that the Shannon estimate of 1.3 bits per letter corresponds to a per-word perplexity of 142 for general English. The numbers we report earlier for the WSJ experiments are significantly lower than this, since the training and test set came from the same subsample of English. That is, those experiments underestimate the complexity of English (since the Wall Street Journal looks very little like Shakespeare, for example)

A number of scholars have independently made the intriguing suggestion that entropy rate plays a role in human communication in general (Lindblom, 1990; Van Son et al., 1998; Aylett, 1999; Genzel and Charniak, 2002; Van Son and Pols, 2003). The idea is that people speak so as to keep the rate of information being transmitted per second roughly constant, i.e., transmitting a constant number of bits per second, or maintaining a constant entropy rate. Since the most efficient way of transmitting information through a channel is at a constant rate, language may even have evolved for such communicative efficiency (Plotkin and Nowak, 2000). There is a wide variety of evidence for the constant entropy rate hypothesis. One class of evidence, for speech, shows that speakers shorten predictable words (i.e., they take less time to say predictable words) and lengthen unpredictable words (Aylett, 1999; Jurafsky et al., 2001; Aylett and Turk, 2004). In another line of research, Genzel and Charniak (2002, 2003) show that entropy rate constancy makes predictions about the entropy of individual sentences from a text. In particular, they show that it predicts that local measures of sentence entropy which ignore previous discourse context (for example the *N*-gram probability of sentence), should increase with the sentence number, and they document this increase in corpora. Keller (2004) provides evidence that entropy rate plays a role for the addressee as well, showing a correlation between the entropy of a sentence and the processing effort it causes in comprehension, as measured by reading times in eye-tracking data.

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, Mercer, Bahl, 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, 1990; Jelinek, 1976; Baker, 1975; Bahl et al., 1983; Jelinek, 1990). A trigram model was used in the IBM TANGORA speech recognition system in the 1970s, but the idea was not written up until later.

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). The Good-Turing algorithm was first applied to the smoothing of *N*-gram grammars at IBM by Katz, as cited in Nádas

(1984). Church and Gale (1991) give a good description of the Good-Turing method, as well as the proof. Sampson (1996) also has a useful discussion of Good-Turing. Jelinek (1990) summarizes this and many other early language model innovations used in the IBM language models.

A wide variety of different language modeling and smoothing techniques were tested through the 1980's and 1990's, including Witten-Bell discounting (Witten and Bell, 1991), varieties of class-based models (Jelinek, 1990; Kneser and Ney, 1993; Heeman, 1999; Samuelsson and Reichl, 1999), and others (Gupta et al., 1992). In the late 1990's, Chen and Goodman produced a very influential series of papers with a comparison of different language models (Chen and Goodman, 1996, 1998, 1999; Goodman, 2006). They performed a number of carefully controlled experiments comparing different discounting algorithms, cache models, class-based (cluster) models, and other language model parameters. They showed the advantages of Interpolated Kneser-Ney, which has since become one of the most popular current methods for language modeling. These papers influenced our discussion in this chapter, and are recommended reading if you have further interest in language modeling.

As we suggested earlier in the chapter, recent research in language modeling has focused on adaptation, on the use of sophisticated linguistic structures based on syntactic and dialogue structure, and on very very large N-grams. For example in 2006, Google publicly released a very large set of N-grams that is a useful research resource, consisting of all the five-word sequences that appear at least 40 times from 1,024,908,267,229 words of running text; there are 1,176,470,663 five-word sequences using over 13 million unique words types (Franz and Brants, 2006). Large language models generally need to be pruned to be practical, using techniques such as Stolcke (1998) and Church et al. (2007).

4.12 Summary

This chapter introduced the N-gram, one of the oldest and most broadly useful practical tools in language processing.

- An N-gram probability is the conditional probability of a word given the previous $N-1$ words. N-gram probabilities can be computed by simply counting in a corpus and normalizing (the **Maximum Likelihood Estimate**) or they can be computed by more sophisticated algorithms. The advantage of N-grams is that they take advantage of lots of rich lexical knowledge. A disadvantage for some purposes is that they are very dependent on the corpus they were trained on.
- **Smoothing** algorithms provide a better way of estimating the probability of N-grams than Maximum Likelihood Estimation. Commonly used N-gram smoothing algorithms rely on lower-order N-gram counts via **backoff** or **interpolation**.
- Both backoff and interpolation require discounting such as **Kneser-Ney**, **Witten-Bell** or **Good-Turing** discounting.
- N-gram **language models** are evaluated by separating the corpus into a **training set** and a **test set**, training the model on the training set, and evaluating on the test

set. The **perplexity** 2^H of the language model on a test set is used to compare language models.

Exercises

4.1 Write out the equation for trigram probability estimation (modifying Eq. 4.14).

4.2 Write a program to compute unsmoothed unigrams and bigrams.

4.3 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?

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

4.5 Add an option to your program to do Good-Turing discounting.

4.6 Add an option to your program to implement Katz backoff.

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

4.8 (Adapted from Michael Collins). Prove Eq. 4.27 given Eq. 4.26 and any necessary assumptions. That is, show that given a probability distribution defined by the GT formula in Eq. 4.26 for the N items seen in training, that the probability of the next, (i.e. $N+1$st) item being unseen in training can be estimated by Eq. 4.27. You may make any necessary assumptions for the proof, including assuming that all N_c are non-zero.

bag of words

4.9 (Advanced) Suppose someone took all the words in a sentence and reordered them randomly. Write a program which take as input such a **bag of words** and produces as output a guess at the original order. You will need to an N-gram grammar produced by your N-gram program (on some corpus), and you will need to use the Viterbi algorithm introduced in the next chapter. This task is sometimes called **bag generation**.

Bag generation

Authorship attribution

4.10 The field of **authorship attribution** is concerned with discovering the author of a particular text. Authorship attribution is important in many fields, including history, literature, and forensic linguistics. For example Mosteller and Wallace (1964) applied authorship identification techniques to discover who wrote *The Federalist* papers. The Federalist papers were written in 1787-1788 by Alexander Hamilton, John Jay and James Madison to persuade New York to ratify the United States Constitution. They were published anonymously, and as a result, although some of the 85 essays were clearly attributable to one author or another, the authorship of 12 were in dispute between Hamilton and Madison. Foster

(1989) applied authorship identification techniques to suggest that W.S.'s *Funeral Elegy* for William Peter might have been written by William Shakespeare (he turned out to be wrong on this one), and that the anonymous author of *Primary Colors*, the roman à clef about the Clinton campaign for the American presidency, was journalist Joe Klein (Foster, 1996).

A standard technique for authorship attribution, first used by Mosteller and Wallace, is a Bayesian approach. For example, they trained a probabilistic model of the writing of Hamilton and another model on the writings of Madison, then computed the maximum-likelihood author for each of the disputed essays. There are many complex factors that go into these models, including vocabulary use, word length, syllable structure, rhyme, grammar; see Holmes (1994) for a summary. This approach can also be used for identifying which genre a text comes from.

One factor in many models is the use of rare words. As a simple approximation to this one factor, apply the Bayesian method to the attribution of any particular text. You will need three things: a text to test and two potential authors or genres, with a large on-line text sample of each. One of them should be the correct author. Train a unigram language model on each of the candidate authors. You are only going to use the **singleton** unigrams in each language model. You will compute $P(T|A_1)$, the probability of the text given author or genre A_1, by (1) taking the language model from A_1, (2) by multiplying together the probabilities of all the unigrams that only occur once in the "unknown" text and (3) taking the geometric mean of these (i.e., the *n*th root, where *n* is the number of probabilities you multiplied). Do the same for A_2. Choose whichever is higher. Did it produce the correct candidate?