N-gram Language Models (Part-I)

Uses of Language Modelling:

1. Predicting is difficult—especially about the future What word, for example, is likely to follow

Please turn your homework ...

Hopefully, most of you concluded that a very likely word is in, or possibly over, but probably not refrigerator or the.

We will formalize this intuition by introducing models that assign a probability to each possible next word. The same models will also serve to assign a probability to an entire sentence. Such a model, for example, could predict that the following sequence has a much higher probability of appearing in a text:

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

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

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

2. Why would you want to predict upcoming words, or assign probabilities to sentences? Probabilities are essential in any task in which we have to identify words in noisy, ambiguous input, like **speech recognition**. For a speech recognizer to realize that you said

I will be back soonish and not **I will be bassoon dish**, it helps to know that back soonish is a much more probable sequence than bassoon dish.

3. For writing tools like **spelling correction or grammatical error correction**, we need to find and correct errors in writing like **Their are two midterms**, in which **There was mistyped as Their**, or **Everything has improve**, in which improve should have been improved. The phrase **There are** will be much more probable than **Their are**, and has improved than has improve, allowing us to help users by detecting and correcting these errors.

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

他 向 记 者 介 绍 了 主要 内容

He to reporters introduced main content

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

he introduced reporters to the main contents of the statement

he briefed to reporters the main contents of the statement

he briefed reporters on the main contents of the statement

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

Language Models: Models that assign probabilities to sequences of words are called language models or LMs. The simplest model that assigns probabilities to sentences and sequences of words are the **n-gram**. An n-gram is a sequence of n words: a **2-gram** (which we'll call bigram) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a **3-gram** (a trigram) is a three-word sequence of words like "please turn your", or "turn your homework".

We'll see how to use n-gram models to estimate the probability of the last word of an n-gram given the previous words, and also to assign probabilities to entire sequences. The n-gram models are much simpler than state-of-the art neural language models based on the RNNs and transformers.

N-Grams

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

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

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

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

With a large enough corpus, such as the web, we can compute these counts and estimate the

probability. While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions of the example sentence may have counts of zero on the web (such as "Walden Pond's water is so transparent that the"; well, used to have counts of zero). Similarly, if we wanted to know the joint probability of an entire sequence of words like its water is so transparent, we could do it by asking "out of all possible sequences of five words, how many of them are its water is so transparent?" We would have to get the count of its water is so transparent and divide by the sum of the counts of all possible five word sequences. That seems rather a lot to estimate!

For this reason, we'll need to introduce more clever ways of estimating the probability of a word w given a history h, or the probability of an entire word sequence W. Now, how can we compute probabilities of entire sequences like $P(w_1;w_2;\ldots;w_n)$? One thing we can do is decompose this probability using the chain rule of probability:

Applying the chain rule to words, we get

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

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. 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, $P(w_n|w_{1:n-1})$.

The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words. The bigram model, 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(wn|wn-1). In other words, instead of computing the probability

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

we approximate it with the probability $P(\text{the}|\text{that})$

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

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

The assumption that the probability of a word depends only on the previous word is Markov called a **Markov assumption.** Markov models are the class of probabilistic models that

assume

we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the n-gram (which looks n-1 words into the past).

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

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

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence

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

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

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

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

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>

<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.

$$P(\text{I}|\text{<s>}) = \tfrac{2}{3} = .67 \qquad P(\text{Sam}|\text{<s>}) = \tfrac{1}{3} = .33 \qquad P(\text{am}|\text{I}) = \tfrac{2}{3} = .67$$
$$P(\text{</s>}|\text{Sam}) = \tfrac{1}{2} = 0.5 \qquad P(\text{Sam}|\text{am}) = \tfrac{1}{2} = .5 \qquad P(\text{do}|\text{I}) = \tfrac{1}{3} = .33$$

Maximum Likelihood Estimate:

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

The above equation 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 frequencies as a way to estimate probabilities is an example of maximum likelihood estimation or MLE. In MLE, the resulting parameter set maximizes the likelihood of the training set T given the model M (i.e., P(T|M)). For example, suppose the word Chinese occurs 400 times in a corpus of a million words like the Brown corpus. What is the probability that a random word selected from some other text of, say, a million words will be the word Chinese? The MLE of its probability is 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.

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

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

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

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

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

Figure below shows the bigram probabilities after normalization (dividing each cell above Figure by the appropriate unigram for its row, taken from the following set of unigram

probabilities):

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

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

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

Here are a few other useful probabilities:

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

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

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

compute the probability of i want chinese food.

Some practical issues: Although for pedagogical purposes we have only described trigram bigram models, in practice it's more common to use trigram models, which condition on the previous two words rather than the previous word, or 4-gram or even 5-gram models, when there is sufficient training data. Note that for these larger ngrams, we'll need to assume extra contexts to the left and right of the sentence end.

For example, to compute trigram probabilities at the very beginning of the sentence, we use two pseudo-words for the first trigram (i.e., P(I|<s><s>).

We always represent and compute language model probabilities in log format as log probabilities. Since probabilities are (by definition) less than or equal to1, the more probabilities we multiply together, the smaller the product becomes. Multiplying enough n-grams together would result in numerical underflow. By using log probabilities instead of raw probabilities, we get numbers that are not as small.

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

Evaluating Language Models

The best way to evaluate the performance of a language model is to embed it in an application and measure how much the application improves. Such end-to-end evaluation is called extrinsic evaluation. Extrinsic evaluation is the only way to know if a particular improvement in a component is really going to help the task at hand. Thus, for speech recognition, we can compare the performance of two language models by running the speech recognizer twice, once with each language model, and seeing which gives the more accurate transcription. Unfortunately, running big NLP systems end-to-end is often very expensive. Instead, it would be nice to have a metric that can be used to quickly evaluate potential improvements in a language model. An intrinsic evaluation metric is one that measures the quality of a model independent of any application.

For an intrinsic evaluation of a language model we need a test set. As with many of the statistical models in our field, the probabilities of an n-gram model come from the corpus it is trained on, the training set or training corpus. We can then measure the quality of an n-gram model by its performance on some unseen data called the test set or test corpus. So if we are given a corpus of text and want to compare two different n-gram models, we divide the data into training and test sets, train the parameters of both models on the training set, and then compare how well the two trained models fit the test set. But what does it mean to "fit the test set"? The answer is simple: whichever model assigns a higher probability to the test set—meaning it more accurately predicts the test set, is a better model.

Perplexity

In practice we don't use raw probability as our metric for evaluating language models, but a variant called perplexity. The perplexity (sometimes called PPL for short) of a language model on a test set is the inverse probability of the test set, normalized by the number of words. For a test set W = w1w2…wN,:

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

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

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

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

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

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

Minimizing perplexity is equivalent to maximizing the test set probability according to the language model.

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

	Unigram	Bigram	Trigram
Perplexity	962	170	109

As we see above, the more information the n-gram gives us about the word sequence, the higher the probability the n-gram will assign to the string.

Sampling sentences from a language model

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

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

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

Generalization and Zeros

The n-gram model, like many statistical models, is dependent on the training corpus. One implication of this is that the probabilities often encode specific facts about a given training corpus. Another implication is that n-grams do a better and better job of modelling the training corpus as we increase the value of N. We can use the sampling method from the prior section to visualize both of these facts! To give an intuition for the increasing power of higher-order n-grams, Figure below shows random sentences generated from unigram, bigram, trigram, and 4-grm models trained on Shakespeare's works.

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

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

The longer the context on which we train the model, the more coherent the sentences. In the unigram sentences, there is no coherent relation between words or any sentence-final punctuation. The bigram sentences have some local word-to-word coherence (especially if we consider that punctuation counts as a word). The trigram and 4-gram sentences are beginning to look a lot like Shakespeare. Indeed, a careful investigation of the 4-gram sentences shows that they look a little too much like Shakespeare. The words ***It cannot be but so*** are directly from ***King John***. From Shakespeare (N =884,647, V=29,066), our n-gram probability matrices are ridiculously sparse. There are V2 =844,000,000 possible bigrams alone, and the number of possible 4-grams is V4 = $7X10^{17}$. Thus, once the generator has chosen the first 4-gram (It cannot be but), there are only five possible continuations (that, I, he, thou, and so); indeed, for many 4-grams, there is only one continuation.

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

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

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

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

How should we deal with this problem when we build n-gram models? One step is to be sure to use a training corpus that has a similar genre to whatever task we are trying to accomplish. To build a language model for translating legal documents, we need a training corpus of legal documents. To build a language model for a question-answering system, we need a training corpus of questions. It is equally important to get training data in the appropriate dialect or variety, especially when processing social media posts or spoken transcripts.

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

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

But suppose our test set has phrases like:

denied the offer
denied the loan

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

These zeros—things that don't ever occur in the training set but do occur in the test set—are a problem for two reasons. First, their presence means we are underestimating the probability of all sorts of words that might occur, which will hurt the performance of any application we want to run on this data. Second, if the probability of any word in the test set is 0, the entire probability of the test set is 0. By definition, perplexity is based on the inverse probability of the test set. Thus, if some words have zero probability, we can't compute perplexity at all, since we can't divide by 0! There are two solutions, depending on the kind of zero. For words whose n-gram probability is zero because they occur in a novel test set context, like the example of **denied the** offer above, we'll introduce algorithms called smoothing or discounting. Smoothing algorithms shave off a bit of probability mass from some more frequent events and give it to these unseen events. But first, let's talk about an even more insidious form of zero: words that the model has never seen below at all (in any context): **unknown words!**

Unknown Words

What do we do about words we have never seen before? Perhaps the word **Jurafsky** simply did not occur in our training set, but pops up in the test set! We can choose to disallow this situation from occurring, by stipulating that we already know all the words that can occur. In such a closed vocabulary system the test set can only contain words from this known lexicon, and there will be no unknown words.

In most real situations, however, we have to deal with words we haven't seen before, which we'll call unknown words, or out of vocabulary (OOV) words. The percentage of OOV words that appear in the test set is called the OOV rate. One way to create an open vocabulary system

is to model these potential unknown words in the test set by adding a pseudo-word called <UNK>.

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

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

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

Smoothing

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

Laplace Smoothing

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

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

Laplace smoothing merely adds one to each count (hence its alternate name **add one smoothing**). Since there are V words in the vocabulary and each one was incremented, we also need to adjust the denominator to take into account the extra V observations.

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

Let's smooth our Berkeley Restaurant Project bigrams. Figure below shows the add-one smoothed counts for the bigrams in Berkeley Restaurant Project.

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

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

Recall that normal bigram probabilities are computed by normalizing each row of counts by the unigram count:

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

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

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

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 Figure below.

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

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

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

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

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

Add-k smoothing

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

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

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

Backoff and Interpolation

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on. If we are trying to compute $P(w_n|w_{n-2}w_{n-1})$ but we have no examples of a particular trigram $w_{n-2}w_{n-1}w_n$, we can instead estimate its probability by using the bigram probability $P(w_n|w_{n-1})$. Similarly, if we don't have counts to compute $P(w_n|w_{n-1})$, we can look to the unigram $P(w_n)$. In other words, sometimes using less context is a good thing, helping to generalize more for contexts that the model hasn't learned much about. There are two ways to use this n-gram "hierarchy". In backoff, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order n-gram.

By contrast, in interpolation, we always mix the probability estimates from all the n-gram estimators, weighting and combining the trigram, bigram, and unigram counts. In simple linear interpolation, we combine different order n-grams by linearly interpolating them. Thus, we

estimate the trigram probability $P(w_n|w_{n-2}w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a

λ:

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

The λs must sum to 1, making Equation equivalent to a weighted average:

$$\sum_i \lambda_i = 1$$