Chapter 7
Language Models

One essential component of any statistical machine translation system is the **language model**, which measures how likely it is that a sequence of words would be uttered by an English speaker. It is easy to see the benefits of such a model. Obviously, we want a machine translation system not only to produce output words that are true to the original in meaning, but also to string them together in fluent English sentences.

language model

In fact, the language model typically does much more than just enable fluent output. It supports difficult decisions about word order and word translation. For instance, a probabilistic language model p_{LM} should prefer correct word order to incorrect word order:

$$p_{\text{LM}}(\text{the house is small}) > p_{\text{LM}}(\text{small the is house}) \tag{7.1}$$

Formally, a language model is a function that takes an English sentence and returns the probability that it was produced by an English speaker. According to the example above, it is more likely that an English speaker would utter the sentence *the house is small* than the sentence *small the is house*. Hence, a good language model p_{LM} assigns a higher probability to the first sentence.

This preference of the language model helps a statistical machine translation system to find the right word order. Another area where the language model aids translation is word choice. If a foreign word (such as the German *Haus*) has multiple translations (*house, home, ...*), lexical translation probabilities already give preference to the more common translation (*house*). But in specific contexts, other translations

may be preferred. Again, the language model steps in. It gives higher probability to the more natural word choice in context, for instance:

$$p_{\text{LM}}(\text{I am going home}) > p_{\text{LM}}(\text{I am going house}) \tag{7.2}$$

This chapter presents the dominant language modeling methodology, n-gram language models, along with smoothing and back-off methods that address issues of data sparseness, the problems that arise from having to build models with limited training data.

7.1 N-Gram Language Models

n-gram

The leading method for language models is **n-gram** language modeling. N-gram language models are based on statistics of how likely words are to follow each other. Recall the last example. If we analyze a large amount of text, we will observe that the word *home* follows the word *going* more often than the word *house* does. We will be exploiting such statistics.

Formally, in language modeling, we want to compute the probability of a string $W = w_1, w_2, ..., w_n$. Intuitively, $p(W)$ is the probability that if we pick a sequence of English words at random (by opening a random book or magazine at a random place, or by listening to a conversation) it turns out to be W.

How can we compute $p(W)$? The typical approach to statistical estimation calls for first collecting a large amount of text and counting how often W occurs in it. However, most long sequences of words will not occur in the text at all. So we have to break down the computation of $p(W)$ into smaller steps, for which we can collect sufficient statistics and estimate probability distributions.

sparse data

Dealing with **sparse data** that limits us to collecting sufficient statistics to estimate reliable probability distributions is the fundamental problem in language modeling. In this chapter, we will examine methods that address this issue.

7.1.1 Markov Chain

predicting one word

In n-gram language modeling, we break up the process of predicting a word sequence W into **predicting one word** at a time. We first decompose the probability using the chain rule:

$$p(w_1, w_2, ..., w_n) = p(w_1)\, p\,(w_2|w_1)\, ...\, p(w_n|w_1, w_2, ..., w_{n-1}) \tag{7.3}$$

history

The language model probability $p(w_1, w_2, ..., w_n)$ is a product of word probabilities given a **history** of preceding words. To be able to estimate these word probability distributions, we limit the history to m words:

$$p(w_n|w_1, w_2, ..., w_{n-1}) \simeq p(w_n|w_{n-m}, ..., w_{n-2}, w_{n-1}) \tag{7.4}$$

This type of model, where we step through a sequence (here: a sequence of words), and consider for the transitions only a limited history, is called a **Markov chain**. The number of previous states (here: words) is the **order** of the model.

Markov chain

order

The **Markov assumption** states that only a limited number of previous words affect the probability of the next word. It is technically wrong, and it is not too hard to come up with counterexamples that demonstrate that a longer history is needed. However, limited data restrict the collection of reliable statistics to short histories.

Markov assumption

Typically, we choose the actually number of words in the history based on how much training data we have. More training data allows for longer histories. Most commonly, **trigram** language models are used. They consider a two-word history to predict the third word. This requires the collection of statistics over sequences of three words, so-called 3-grams (trigrams). Language models may also be estimated over 2-grams (**bigrams**), single words (**unigrams**), or any other order of n-grams.

trigram

bigram

unigram

7.1.2 Estimation

In its simplest form, the estimation of trigram word prediction probabilities $p(w_3|w_1, w_2)$ is straightforward. We count how often in our training corpus the sequence w_1, w_2 is followed by the word w_3, as opposed to other words. According to maximum likelihood estimation, we compute:

$$p(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\sum_w \text{count}(w_1, w_2, w)} \tag{7.5}$$

Figure 7.1 gives some examples of how this estimation works on real data, in this case the European parliament corpus. We consider three different histories: *the green*, *the red*, and *the blue*. The words that most frequently follow are quite different for the different histories. For instance *the red cross* is a frequent trigram in the Europarl corpus, which also mentions *the green party* a lot, a political organization.

the green (total: 1748)

Word	Count	Prob.
paper	801	0.458
group	640	0.367
light	110	0.063
party	27	0.015
ecu	21	0.012

the red (total: 225)

Word	Count	Prob.
cross	123	0.547
tape	31	0.138
army	9	0.040
card	7	0.031
,	5	0.022

the blue (total: 54)

Word	Count	Prob.
box	16	0.296
.	6	0.111
flag	6	0.111
,	3	0.056
angel	3	0.056

Figure 7.1 Counts for trigrams and estimated word probabilities based on a two-word history: 123 out of 225 trigrams in the Europarl corpus that start with *the red* end with *cross,* so the maximum likelihood probability is 0.547.

Let us look at one example of the maximum likelihood estimation of word probabilities given a two-word history. There are 225 occurrences of trigrams that start with *the red* in the Europarl corpus. In 123 of them, the third word is *cross*, hence the maximum likelihood estimation for $p_{\text{LM}}(\text{cross}|\text{the red})$ is $\frac{123}{225} \simeq 0.547$.

7.1.3 Perplexity

Language models differ in many ways. How much text are they trained on? Are they using bigrams, trigrams, or even 4-grams? What kind of smoothing techniques are used (more on that below starting with Section 7.2)?

To guide decisions when building a language model, we need a measure of a model's quality. Our premise for language modeling is that a model should give good English text a high probability and bad text a low probability.

If we have some held-out sample text, we can compute the probability that our language model assigns to it. A better language model should assign a higher probability to this text.

perplexity
cross-entropy

This is the idea behind the common evaluation metric for language models, **perplexity**. It is based on the **cross-entropy**, which is defined as

$$\begin{aligned} H(p_{\text{LM}}) &= -\frac{1}{n} \log p_{\text{LM}}(w_1, w_2, ..., w_n) \\ &= -\frac{1}{n} \sum_{i=1}^{n} \log p_{\text{LM}}(w_i|w_1, ..., w_{i-1}) \end{aligned} \tag{7.6}$$

Perplexity is a simple transformation of cross-entropy:

$$PP = 2^{H(p_{\text{LM}})} \tag{7.7}$$

More on the mathematical derivation of cross-entropy and perplexity later. Let us first develop some intuition about what these metrics mean. Given a language model trained on the Europarl corpus, we want to assess its perplexity on a sentence from a section of the corpus that is not included in training. Our example sentence is:

I would like to commend the rapporteur on his work.

Using a trigram language model,[1] the probability of the trigrams for this sentence are given in Figure 7.2. We frame the sentence with markers for sentence start <s> and sentence end </s>. This allows us to properly model the probability of *i* being the first word of the sentence, by

[1] We report numbers using language models trained on the Europarl corpus using Kneser–Ney smoothing, computed with the SRILM toolkit.

Prediction	p_{LM}	$-\log_2 p_{\text{LM}}$
$p_{\text{LM}}(i\|\text{</s><s>})$	0.109	3.197
$p_{\text{LM}}(\text{would}\|\text{<s>}i)$	0.144	2.791
$p_{\text{LM}}(\text{like}\|\text{i would})$	0.489	1.031
$p_{\text{LM}}(\text{to}\|\text{would like})$	0.905	0.144
$p_{\text{LM}}(\text{commend}\|\text{like to})$	0.002	8.794
$p_{\text{LM}}(\text{the}\|\text{to commend})$	0.472	1.084
$p_{\text{LM}}(\text{rapporteur}\|\text{commend the})$	0.147	2.763
$p_{\text{LM}}(\text{on}\|\text{the rapporteur})$	0.056	4.150
$p_{\text{LM}}(\text{his}\|\text{rapporteur on})$	0.194	2.367
$p_{\text{LM}}(\text{work}\|\text{on his})$	0.089	3.498
$p_{\text{LM}}(.\|\text{his work})$	0.290	1.785
$p_{\text{LM}}(\text{</s>}\|\text{work .})$	0.99999	0.000014
	Average	2.634

Figure 7.2 Computation of perplexity for the sentence *I would like to commend the rapporteur on his work:* For each word, the language model provides a probability. The average of the negative log of these word prediction probabilities is the cross-entropy H, here 2.634. Perplexity is 2^H.

looking at $p_{\text{LM}}(i|\text{</s><s>})$. We also include the probability for the end of the sentence: $p_{\text{LM}}(\text{</s>}|\text{work.})$

The language model provides for each word a probability, which indicates how likely it is that the word would occur next in the sentence, given the previous two-word history. According to the language model, the probability that a sentence starts with the personal pronoun *i* is 0.109043, a fairly probable sentence start for the Europarl corpus from which the language model is built.

The next words are also fairly common: *would* with probability 0.144, *like* with probability 0.489, and the almost certain *to* after that (probability 0.904). Now many words may follow, most likely verbs, but *commend* is a rather unusual verb, even for this corpus. Its language model probability in this context is only 0.002.

The cross-entropy is the average of the negative logarithm of the word probabilities. In Figure 7.2, next to each probability you can find its negative $\log_2$. Highly expected words, such as the end of sentence marker </s> following the period (0.000014), are balanced against less expected words, such as *commend* (8.794). In this example, the cross-entropy is 2.634, and hence the perplexity is $2^{2.634} = 6.206$.

Note that we do not simply count how many words the language model would have guessed right (e.g., the most likely word following *the green* is *paper*, etc.). The measure of perplexity is based on how much probability mass is given to the actually occurring words. A good model of English would not waste any probability mass on impossible sequences of English words, so more of it is available for possible sequences, and most of it concentrated on the most likely sequences.

Figure 7.3 compares language models of different order, from unigram to 4-gram. It gives negative log-probabilities and the resulting perplexity for our example sentence. Not surprisingly, the higher order n-gram models are better at predicting words, resulting in a lower perplexity.

Figure 7.3 Negative $\log_2$ probabilities for word prediction for unigram to 4-gram language models: A longer history allows more accurate prediction of words, resulting in lower perplexity.

Word	Unigram	Bigram	Trigram	4-gram
i	6.684	3.197	3.197	3.197
would	8.342	2.884	2.791	2.791
like	9.129	2.026	1.031	1.290
to	5.081	0.402	0.144	0.113
commend	15.487	12.335	8.794	8.633
the	3.885	1.402	1.084	0.880
rapporteur	10.840	7.319	2.763	2.350
on	6.765	4.140	4.150	1.862
his	10.678	7.316	2.367	1.978
work	9.993	4.816	3.498	2.394
.	4.896	3.020	1.785	1.510
</s>	4.828	0.005	0.000	0.000
Average	8.051	4.072	2.634	2.251
Perplexity	265.136	16.817	6.206	4.758

Mathematical derivation of perplexity

entropy

Perplexity is based on the the concept of **entropy**, which measures uncertainty in a probability distribution. Mathematically, it is defined as

$$H(p) = -\sum_x p(x) \log_2 p(x) \tag{7.8}$$

If a distribution has very certain outcomes, then entropy is low. In the extreme case, if the distribution has only one event x with probability $p(x) = 1$, the entropy is 0. The more events that are possible with significant probability, the higher the entropy. If there are two equally likely events with $p(x) = 0.5$ each, then $H(p) = 1$; if there are four equally likely events, then $H(p) = 2$; and so on.

Entropy is a common notion in the sciences and also in computer science, since it is related to how many bits are necessary to encode messages from a source. More frequent messages (think: more probable events) are encoded with shorter bit sequences, to save overall on the length over several messages.

If we consider the event of observing a sequence of words $W_1^n = w_1, ..., w_n$ from a language L, we can compute the entropy for this probability distribution in the same way:

$$H(p(W_1^n)) = -\sum_{W_1^n \in L} p(W_1^n) \log p(W_1^n) \tag{7.9}$$

This computation of entropy will strongly depend on the length n of the sequences. Longer sequences are generally less likely than shorter sequences. To have a more meaningful measure, we want to compute

entropy rate

entropy per word, also called the **entropy rate**:

$$\frac{1}{n} H(p(W_1^n)) = -\frac{1}{n} \sum_{W_1^n \in L} p(W_1^n) \log p(W_1^n) \tag{7.10}$$

We can bring this argument to its logical conclusion by attempting to measure the **true entropy of a language** L given its true probability distribution p. Instead of limiting the calculation to a fixed sequence of length n, we need to consider the observation of infinitely long sequences:

true entropy of a language

$$\begin{aligned} H(L) &= \lim_{n\to\text{inf}} -\frac{1}{n} H(p(W_1^n)) \\ &= \lim_{n\to\text{inf}} -\frac{1}{n} \sum_{W_1^n \in L} p(W_1^n) \log p(W_1^n) \end{aligned} \tag{7.11}$$

Instead of computing the probability of all sequences of infinite length, we can simplify this equation to:

$$H(L) = \lim_{n\to\text{inf}} -\frac{1}{n} \log p(W_1^n) \tag{7.12}$$

For a mathematical justification of this last step, refer to the Shannon–McMillan–Breiman theorem. This is explained, for instance, by Cover and Thomas [1991, Chapter 15].

There have been attempts to measure the true entropy of a language such as English. See the box on page 188 for a historical account. In language modeling we are trying to produce a model that predicts the next word, given a history of preceding words. An instructive game is to measure human performance at this task.

In practice, we do not have access to the true probability distribution p, only a model p_{LM} for it. When we replace p with p_{LM} in Equation (7.12), we obtain the formula for the **cross-entropy**:

cross-entropy

$$H(L, p_{\text{LM}}) = \lim_{n\to\text{inf}} -\frac{1}{n} \log p_{\text{LM}}(W_1^n) \tag{7.13}$$

The true entropy of a language $H(L)$ is a lower bound for the cross-entropy of a model p_{LM}:

$$H(L) \leq H(L, p_{\text{LM}}) \tag{7.14}$$

A good language model achieves a cross-entropy close to the true entropy of the language, so this becomes a convenient measure for the quality of language models.

In practice, we do not have infinite sequences of words, but only a limited test set. However, we can still use this formula to approximate the cross-entropy of a model p_{LM}. If you look back at Figure 7.3, this is exactly what we did for our example, with the additional step of converting cross-entropy into perplexity. For more realistic estimates of perplexity, we need of course a larger sample of thousands of words or even more.

Estimating the entropy of English

Take a newspaper article and try to predict the next word given what has already been said in the article. You will often find it very easy to guess. Obviously language is very redundant, and we can make do with many fewer letters and words to communicate the same meaning.

compression

Compression – Anybody who has ever used a file compression program such as `bzip2` will know that we can compress text considerably. Usual ASCII encoding uses 8 bits per letter. When we compress the English Europarl corpus, we can reduce its size from 15 MB to 3 MB. So, its 27 million characters require on average only 1.7 bits each.

Shannon's guessing game

Shannon's guessing game – What is the minimum number of bits required to store text? Shannon [1951] introduced the guessing game to estimate the entropy of English. He worked on the character level, ignoring case and punctuation, leaving 26 characters and the space symbol. A human subject is asked to predict the next letter in the sequence given the preceding text, and we record how many guesses were necessary for each letter.

We can convert the sequence of letters into a sequence of numbers of guesses. This sequence can also be interpreted as follows: A perfect model of English will have a ranking of the predictions for the next letter, based on their probabilities. By telling this model that the next letter is, say, the 5th guess, it is able to reconstruct the text. Given this sequence, we can estimate the probability p_1 that a letter is the first choice, p_2 that it is the second choice, and so. With this distribution p, we are able to compute the entropy using Equation (7.8):

$$H(p) = -\sum_{i=1}^{27} p_i(x) \log p_i(x)$$

gambling estimate

Gambling estimate – Cover and King [1978] propose a gambling estimate of the entropy of English. To be able to have probabilities directly associated with letter predictions, participants in this game are asked to bet a certain amount for each next letter. The size of the bets directly correlate to probabilities.

Both methods estimate entropy at about 1.3 bits per character.

7.2 Count Smoothing

We have so far ignored the following problem: What if an n-gram has not been seen in training, but then shows up in the test sentence we want to score? Using the formula for maximum likelihood

estimation in Equation (7.5), we would assign **unseen n-grams** a probability of 0.

unseen n-gram

Such a low probability seems a bit harsh, and it is also not very useful in practice. If we want to compare different possible translations for a sentence, and all of them contain unseen n-grams, then each of these sentences receives a language model estimate of 0, and we have nothing interesting to say about their relative quality. Since we do not want to give any string of English words zero probability, we need to assign some probability to unseen n-grams.

Also consider n-grams that *do* occur in the training corpus. If an n-gram occurs once in the training data, how often do we expect it to occur again in a corpus of equal size? Exactly once, or more, or less?

When we move to higher order n-gram models, the problem of unseen n-grams becomes more severe. While these models enable us to capture more context, it is less likely that a higher order n-gram has been observed before in a training corpus.

This section will discuss various methods for adjusting the **empirical counts** that we observe in the training corpus to the **expected counts** of n-grams in previously unseen text.

empirical count

expected count

7.2.1 Add-One Smoothing

The simplest form of adapting empirical counts is to add a fixed number (say, 1) to every count. This will eliminate counts of zero. Since probabilities still have to add up to one, we need to consider these additional imaginary occurrences of n-grams when estimating probabilities.

Instead of simply using the count c and the total number of n-grams n for the maximum likelihood estimation of the probability p by

$$p = \frac{c}{n} \tag{7.15}$$

we now adjust this to

$$p = \frac{c+1}{n+v} \tag{7.16}$$

where v is the total number of possible n-grams.

If we have a vocabulary size of, say, 86,700 distinct tokens, the number of possible bigrams is $86{,}700^2 = 7{,}516{,}890{,}000$, just over 7.5 billion. Contrast this number with the size of the Europarl corpus, which has 30 million words, and hence a total number of 30 million bigrams. We quickly notice that **add-one smoothing** gives undue credence to counts that we do not observe.

add-one smoothing

We can remedy this by **add-α smoothing**, adding a smaller number $\alpha < 1$ instead of one:

add-α smoothing

$$p = \frac{c+\alpha}{n+\alpha v} \tag{7.17}$$

Figure 7.4 Add-one smoothing: Counts are adjusted by adding 1 (or a lower value $\alpha = 0.005$), given a corpus size of $n = 29{,}564{,}160$ tokens and a vocabulary size of $v = 86{,}700$ tokens. The table displays raw count, adjusted count after adding a value (1, $\alpha = 0.00017$) and renormalization, and actual test count.

	Adjusted count		
Count c	$(c+1)\frac{n}{n+v^2}$	$(c+\alpha)\frac{n}{n+\alpha v^2}$	Test count t_c
0	0.00378	0.00016	0.00016
1	0.00755	0.95725	0.46235
2	0.01133	1.91433	1.39946
3	0.01511	2.87141	2.34307
4	0.01888	3.82850	3.35202
5	0.02266	4.78558	4.35234
6	0.02644	5.74266	5.33762
8	0.03399	7.65683	7.15074
10	0.04155	9.57100	9.11927
20	0.07931	19.14183	18.95948

But what is a good value for α? We may determine this value by experimentation, e.g., by trying out different values and settling on the value that optimizes perplexity on a held-out test set.

Figure 7.4 shows how this works out in practice, given the Europarl corpus. Counts are adjusted either by adding one or by adding $\alpha = 0.00017$. The latter number is optimized for a match between the adjusted counts and test counts of unseen bigrams.

Add-one smoothing clearly gives too much weight to unseen n-grams. Their adjusted count is 0.00378 vs. the test count of only 0.00016. As a consequence, not much probability mass is left for observed n-grams. All n-grams in the table, even the ones occurring 20 times in training, are adjusted to a count below 1.

Add-α smoothing assigns the correct adjusted count to unseen n-grams since this is how we set α. But the other adjusted counts are still off. Singletons are adjusted to a count of 0.95725, while the test count is less than half of that (0.46235).

7.2.2 Deleted Estimation

Adjusting counts has to address the question: If we observe an n-gram c times in the training corpus, what does that really mean? How often do we expect it to see in the future?

To be able to answer this question, we need a second corpus. Pooling together all n-grams that appear some fixed number of times in the training corpus, how often do they occur in the second corpus? How many new, previously unseen n-grams do we encounter in the second corpus? How many that have we already seen once, twice, etc.?

This is clearly a function of the size of the original training corpus and the expansiveness of the text domain. Figure 7.5 gives some

Count r	Count of counts N_r	Count in held-out T_r	Exp. count $E[r] = T_r/N_r$
0	7,515,623,434	938,504	0.00012
1	753,777	353,383	0.46900
2	170,913	239,736	1.40322
3	78,614	189,686	2.41381
4	46,769	157,485	3.36860
5	31,413	134,653	4.28820
6	22,520	122,079	5.42301
8	13,586	99,668	7.33892
10	9,106	85,666	9.41129
20	2,797	53,262	19.04992

Figure 7.5 Deleted estimation: In half of the Europarl corpus 753,777 distinct bigrams occur exactly once. These n-grams occur 353,383 times in the held-out set. For each of these n-grams, the expected count is $\frac{353,383}{753,777} = 0.46900$.

numbers for such a study on the example of the Europarl corpus. We split it into one half for training (i.e., collecting counts for n-grams) and the other half for validating these counts.

The distribution of n-gram counts has a **long tail**, meaning a long list of rare n-grams – of the 1,266,566 bigrams in this corpus, more than half, 753,777, occur only once. Such a skewed distribution is typical for natural language. It is often called a **Zipfian distribution**. On the other hand, we have a few events that are very frequent. The most frequent word bigram *of the* occurs 283,846 times.

long tail

Zipfian distribution

Let us come back to our discussion of empirical and expected counts. According to the table, 753,777 bigrams occur exactly once in the training corpus (half of the entire Europarl corpus). These bigrams occur 353,383 times in the held-out corpus (the other half of the Europarl corpus). Thus, on average, each of these singleton bigrams occurred $\frac{353,383}{753,777} = 0.469$ times. This answers our original question: If an n-gram occurs once in the training corpus, how often do we expect it to occur in a corpus of equal size? It turns out that the answer is 0.469 times in this case.

Bigrams not previously observed in the training corpus occur 938,504 times in the held-out corpus. Since there are 7,515,623,434 distinct unseen n-grams, each of them occurs on average $\frac{938,504}{7515,623,434} = 0.00012$ times.

Adjusting real counts to these expected counts will give us better estimates both for seen and unseen events, and also eliminate the problem of zeros in computing language model probabilities.

By using half of the corpus for validation, we lose it as training data for count collection. But this is easy to remedy. By alternating between training and held-out corpus – meaning we first use one half as training corpus and the other as held-out corpus, and then switch around – we can make use of all the data. We then average the count-of-counts and

counts in held-out data that we collect in one direction (N_r^a and T_r^a) with the ones we collect in the other direction (N_r^b and T_r^b):

$$r_{\text{del}} = \frac{T_r^a + T_r^b}{N_r^a + N_r^b} \text{ where } r = \text{count}(w_1, ..., w_n)$$

The expected counts are not entirely accurate. They are estimated with only half of the training data. With more training data, for instance, we would expect to encounter fewer unseen n-grams in new data. We can alleviate this by taking 90% of the corpus for training and 10% for validation, or even 99% for training and 1% for validation.

deleted estimation

The name **deleted estimation** for the technique we just described derives from the fact that we leave one part of the training corpus out for validation. How well does deleted estimation work? If you compare the expected counts as computed in Figure 7.5 with the test counts in Figure 7.4, you see that our estimates come very close.

7.2.3 Good–Turing Smoothing

The idea behind all the smoothing methods we have discussed so far is to get a better assessment of what it means that an n-gram occurs c times in a corpus. Based on this evidence, we would like to know how often we expect to find it in the future.

In other words, we need to adjust the actual count to the expected count. This relates to the probability p, the probability we are ultimately trying to estimate from the counts.

Some mathematical analysis (see the detailed derivation below) leads us to a straightforward formula to calculate the expected count r^* from the actual count r and using count-of-counts statistics:

$$r^* = (r+1)\frac{N_{r+1}}{N_r} \tag{7.18}$$

To apply this formula, we need to collect statistics on how many n-grams occur once in the training corpus (N_1), how many occur twice (N_2), etc., as well as how many n-grams do not occur in the corpus at all (N_0).

See Figure 7.6 for these numbers in the same Europarl corpus we have used so far. Out of a possible $86{,}700^2 = 7{,}516{,}890{,}000$ n-grams, most never occur in the corpus ($N_0 = 7{,}514{,}941{,}065$, to be exact). $N_1 = 1{,}132{,}844$ n-grams occur once in the corpus.

According to Equation (7.18), this leads to an adjusted count for unseen n-grams of $(0+1)\frac{N_1}{N_0} = \frac{1{,}132{,}844}{7{,}514{,}941{,}065} = 0.00015$. This is very close to the actual expected count as validated on the test set, which is 0.00016. The other adjusted counts are very adequate as well.

Count r	Count of counts N_r	Adjusted count r^*	Test count t
0	7,514,941,065	0.00015	0.00016
1	1,132,844	0.46539	0.46235
2	263,611	1.40679	1.39946
3	123,615	2.38767	2.34307
4	73,788	3.33753	3.35202
5	49,254	4.36967	4.35234
6	35,869	5.32928	5.33762
8	21,693	7.43798	7.15074
10	14,880	9.31304	9.11927
20	4,546	19.54487	18.95948

Figure 7.6 Good–Turing smoothing: Based on the count-of-counts N_r, the empirical counts are adjusted to $r^* = (r+1)\frac{N_{r+1}}{N_r}$, a fairly accurate count when compared against the test count.

Good–Turing smoothing

Good–Turing smoothing provides a principled way to adjust counts. It is also fairly simple computationally – all you need to do is collect counts-of-counts on the training corpus.

One caveat: The method fails for large r for which frequently $N_r = 0$, or counts-of-counts are simply unreliable. This can be addressed either by curve-fitting the formula or by simply not adjusting the counts for frequent n-grams.

Derivation of Good–Turing smoothing

Since the derivation of Good–Turing smoothing is a bit tricky, we will first introduce a few concepts that will come in handy later.

Each n-gram α occurs with a probability p at any point in the text. In other words, if we pick an n-gram at random out of any text, it is the n-gram α with the probability p. Unfortunately, we do not have access to the true probabilities of n-grams, otherwise things would be much easier (and we would not have to deal with estimating these probabilities).

If we assume that all occurrences of an n-gram α are independent of each other – if it occurs once, it does not affect the fixed probability p of it occurring again – then the number of times it will occur in a corpus of size N follows the binomial distribution

$$p(c(\alpha) = r) = b(r; N, p_i) = \binom{N}{r} p^r (1-p)^{N-r} \tag{7.19}$$

The goal of Good–Turing smoothing is to compute the *expected count* c^* for n-grams, versus the *empirical count* c that we observe in a fixed training corpus. The expected count of an n-gram α follows from (7.19) pretty straightforwardly:

$$\begin{aligned} E(c^*(\alpha)) &= \sum_{r=0}^{N} r\, p(c(\alpha) = r) \\ &= \sum_{r=0}^{N} r \binom{N}{r} p^r (1-p)^{N-r} \end{aligned} \tag{7.20}$$

As noted before, we do not have access to the true probability p of the n-gram α, so we cannot use this simple formula. So, we have to find another way.

Before we get to this, let us introduce another useful concept, the expected number of n-grams that occur with a certain frequency r, which we denote by $E_N(N_r)$. If we have a finite number s of n-grams that we denote by $\alpha_1, ..., \alpha_s$ that occur with respective probabilities $p_1, ..., p_s$, then each of them may occur with frequency r, so the overall expected number of such n-grams is

$$\begin{aligned} E_N(N_r) &= \sum_{i=1}^{s} p(c(\alpha_i) = r) \\ &= \sum_{i=1}^{s} \binom{N}{r} p_i^r (1 - p_i)^{N-r} \end{aligned} \tag{7.21}$$

Again, we cannot compute this number because we do not have access to the n-gram probabilities p_i, but we actually have a pretty good idea of how many n-grams occur r times in a corpus of N n-grams. This is exactly what we counted previously (see for instance Figure 7.5 on page 191). While of course these counts collected over a large corpus N_r are not the same as true expected counts $E_N(N_r)$, we can still claim that $E_N(N_r) \simeq N_r$, at least for small r.

Let us now move to the derivation of Good–Turing smoothing. Recall that our goal is to compute expected counts c^* for n-grams, based on the actual counts c. Or, to put this into mathematical terms:

$$E(c^*(\alpha)|c(\alpha) = r) \tag{7.22}$$

Let us say for the sake of this mathematical argument that we do not know much about the n-gram α, except that it occurred r times in the training corpus of N n-grams. It may be any of the s n-grams $\alpha_1, ..., \alpha_s$; we do not even know which one.

So we adapt the formula in Equation (7.20) to:

$$E(c^*(\alpha)|c(\alpha) = r) = \sum_{i=1}^{s} N\, p_i\, p(\alpha = \alpha_i|c(\alpha) = r) \tag{7.23}$$

Each n-gram α_i occurs with probability p_i, but we are only interested in it to the degree with which we believe it is our n-gram α, of which we only know that it occurs r times in the training corpus.

Any of the n-grams α_i may occur r times in a corpus with N n-grams, but some more likely than others, depending on its occurrence probability p_i. Devoid of any additional knowledge, we can say that

$$p(\alpha = \alpha_i|c(\alpha) = r) = \frac{p(c(\alpha_i) = r)}{\sum_{j=1}^{s} p(c(\alpha_j) = r)} \tag{7.24}$$

Let us put Equations (7.23) and (7.24) together, and check what we have so far:

$$\begin{aligned} E(c^*(\alpha)|c(\alpha) = r) &= \sum_{i=1}^{s} N\, p_i\, \frac{p(c(\alpha_i) = r)}{\sum_{j=1}^{s} p(c(\alpha_j) = r)} \\ &= \frac{\sum_{i=1}^{s} N\, p_i\, p(c(\alpha_i) = r)}{\sum_{j=1}^{s} p(c(\alpha_j) = r)} \end{aligned} \tag{7.25}$$

Eyeballing this formula, we quickly recognize that the denominator is the formula for $E_N(N_r)$ (see Equation 7.21), for which we claim to have reliable estimates. The numerator is not too different from that either, so we can massage it a bit:

$$\begin{aligned} \sum_{i=1}^{s} N\, p_i\, p(c(\alpha_i) = r) &= \sum_{i=1}^{s} N\, p_i \binom{N}{r} p_i^r (1 - p_i)^{N-r} \\ &= N\, \frac{N!}{N - r!\, r!} p_i^{r+1} (1 - p_i)^{N-r} \\ &= N\, \frac{(r+1)}{N+1}\, \frac{N+1!}{N - r!\, r+1!} p_i^{r+1} (1 - p_i)^{N-r} \\ &= (r+1)\, \frac{N}{N+1} E_{N+1}(N_{r+1}) \\ &\simeq (r+1)\, E_{N+1}(N_{r+1}) \end{aligned} \tag{7.26}$$

Putting this back into Equation (7.25) gives us the formula for Good–Turing smoothing (recall Equation 7.18 on page 192)

$$\begin{aligned} r^* &= E(c^*(\alpha)|c(\alpha) = r) \\ &= \frac{(r+1)\, E_{N+1}(N_{r+1})}{E_N(N_r)} \\ &\simeq (r+1) \frac{N_{r+1}}{N_r} \end{aligned} \tag{7.27}$$

7.2.4 Evaluation

To conclude this section on count smoothing, we use all of the methods we have just described to build bigram language models and check how well they work. We adjust the counts according to the different methods, and then use these counts to estimate bigram probabilities

$$p(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\sum_i \text{count}(w_1, w_i)} \tag{7.28}$$

Since we smoothed out all zero-counts, all bigram probabilities $p(w_2|w_1)$ are positive. Given these probabilities, we can measure the quality of the resulting language models by computing their perplexity on the held-out test set. This is the same test set we used to compute test counts.

Smoothing method	Perplexity
Add-one	383.2
Add-α ($\alpha = 0.00017$)	113.2*
Deleted estimation	113.4
Good–Turing	112.9

Figure 7.7 Evaluation of count smoothing methods: Perplexity for language models trained on the Europarl corpus. Add-α smoothing is optimized on the test set.

Results are given in Figure 7.7. Besides add-one smoothing, which gives too much probability mass to unseen events, all smoothing methods have similar performance, with Good–Turing smoothing coming out slightly ahead. The results for add-α smoothing are unrealistically good, since we optimized the value $\alpha = 0.00017$ on the set we are are testing on.

The small difference between deleted estimation and Good–Turing is partly due to the way we applied deleted estimation. Estimating and validating counts on half the corpus is less reliable than taking the whole corpus into account.

7.3 Interpolation and Back-off

We have already argued that higher order n-grams allow the consideration of a larger context and lead to better language models (see Figure 7.3 on page 186). However, given limited training data, many fluent n-grams of higher order will not be observed.

For instance, we may never observe the two alternative n-grams

- *Scottish beer drinkers*
- *Scottish beer eaters*

if our training corpus hardly mentions *Scottish beer*.

Note that these alternatives may arise if we translate from a foreign language which has a word that equally likely translates to *drinkers* and *eaters*. Our hope is that the language model will aid us in the decision that *drinkers* is the preferred translation in this context.

In the previous section we discuss methods that assign positive probabilities to n-grams with zero count. However, these methods treat all n-grams with the same count the same, and hence make no distinction between the two alternative n-grams above.

In our example, the additional context of *Scottish* adds little valuable information; it just causes a sparse data problem. Hence in this case, we would prefer to only use a bigram language model that draws the distinction between the alternatives

- *beer drinkers*
- *beer eaters*

for which we more likely have evidence in the corpus.

7.3.1 Interpolation

Higher order n-grams may provide valuable additional context, but lower order n-grams are more robust. The idea of **interpolation** is to combine lower order and higher order n-gram language models.

interpolation

We first build n-gram language models p_n for several orders (say $n = 1, 2, 3$), as described in the previous section. This may include Good–Turing smoothing or other count adjusting smoothing techniques, but we may also simply rely on the interpolation with lower order n-grams to remedy zero counts.

Then, we build an interpolated language model p_I by linear combination of the language models p_n:

$$\begin{aligned} p_I(w_3|w_1, w_2) = \; & \lambda_1 \, p_1(w_3) \\ & \times \lambda_2 \, p_2(w_3|w_2) \\ & \times \lambda_3 \, p_3(w_3|w_1, w_2) \end{aligned} \tag{7.29}$$

Each n-gram language model p_n contributes its assessment, which is weighted by a factor λ_n. With a lot of training data, we can trust the higher order language models more and assign them higher weights.

To ensure that the interpolated language model p_I is a proper probability distribution, we require that

$$\begin{aligned} & \forall \lambda_n : 0 \leq \lambda_n \leq 1 \\ & \sum_n \lambda_n = 1 \end{aligned} \tag{7.30}$$

We are left with the task of deciding how much weight is given to the unigram language model (λ_1), and how much is given to the bigram (λ_2) and trigram (λ_3). This may be done by optimizing these parameters on a held-out set.

7.3.2 Recursive Interpolation

Recall that the idea behind interpolating different order n-gram language models is that we would like to use higher order n-gram language models if we have sufficient evidence, but otherwise we rely on the lower order n-gram models.

It is useful to change Equation (7.29) into a definition of **recursive interpolation**:

recursive interpolation

$$\begin{aligned} p_n^I(w_i|w_{i-n+1}, ..., w_{i-1}) = \; & \lambda_{w_{i-n+1},...,w_{i-1}} p_n(w_i|w_{i-n+1}, ..., w_{i-1}) \\ & + \; (1 - \lambda_{w_{i-n+1},...,w_{i-1}}) p_{n-1}^I(w_i|w_{i-n+2}, ..., w_{i-1}) \end{aligned} \tag{7.31}$$

The λ parameters set the degree of how much we trust the n-gram language model p_n, or if we would rather back off to a lower order

model. We may want to make this depend on a particular history $w_{i-n+1}, ..., w_{i-1}$. For instance, if we have seen this history many times before, we are more inclined to trust its prediction.

It is possible to optimize the λ parameters on a held-out training set using EM training. In practice, we do not want to learn for each history a different λ parameter, since this would require large amounts of held-out data. Hence, we may group histories into buckets, for instance based on their frequency in the corpus.

7.3.3 Back-off

We initially motivated smoothing by the need to address the problem of unseen n-grams. If we have seen an n-gram, we can estimate probabilities for word predictions. Otherwise, it seems to be a good idea to back off to lower order n-grams with richer statistics.

back-off

This idea leads us to a recursive definition of **back-off**:

$$p_n^{\text{BO}}(w_i|w_{i-n+1}, ..., w_{i-1}) = \begin{cases} d_n(w_{i-n+1}, ..., w_{i-1})\, p_n(w_i|w_{i-n+1}, ..., w_{i-1}) \\ \qquad \text{if count}_n(w_{i-n+1}, ..., w_i) > 0 \\ \alpha_n(w_{i-n+1}, ..., w_{i-1}))\, p_{n-1}^{\text{BO}}(w_i|w_{i-n+2}, ..., w_{i-1}) \\ \qquad \text{else} \end{cases} \tag{7.32}$$

We build a back-off language model p_n^{BO} from the raw language models p_n. If we have seen an n-gram (count(...) > 0), we use the raw language model probability, otherwise we back off to the lower order back-off p_{n-1}^{BO}.

Since we have to ensure that overall probabilities add up to 1 for a history $w_{i-n+1}, ..., w_{i-1}$, we introduce a discounting function d. Probabilities from the raw language model p_n are discounted by a factor $0 \leq d \leq 1$, and the remaining probability mass is given to the back-off model.

The discounting function d is conditioned on the history of the n-gram. One idea is to group histories based on their frequency in the corpus. If we have seen the history very frequently, we would trust predictions based on this history more, and therefore set a fairly high value for $d_n(w_1, ..., w_{n-1})$. On the other hand, for rare histories, we assume that we have seen only a small fraction of possible predicted words w_n, and therefore give more weight to the back-off, resulting in a lower value for d.

Another way to compute the discounting parameters d is to employ Good–Turing smoothing. This reduces the counts and the remaining probability mass is given to the α parameters.

7.3.4 Diversity of Predicted Words

Witten and Bell [1991] turn our attention to the diversity of words that follow a history. Consider the bigram histories *spite* and *constant*. Both words occur 993 times in the Europarl corpus. Any smoothing method that is based on this count alone would treat these histories the same.

The word *spite* occurs 993 times in the corpus, but we observe only nine different words that follow it. In fact, it is almost always followed by *of* (979 times), due to the common expression *in spite of*. Words such as *yours* (three times) or *towards* (two times) also follow *spite*, but this is rather exceptional.

Contrast this with the word *constant*, which also occurs 993 times in the corpus. However, we find 415 different words that follow it. Some stand out – *and* (42 times), *concern* (27 times), *pressure* (26 times) – but there is a huge tail of words that are seen only once after *constant*, to be precise 268 different words.

How likely is it to encounter a previously unseen bigram that starts with *spite* versus one that starts with *constant*? In the case of *spite* this would be highly unusual. In the case of *constant*, we would not be very surprised. **Witten–Bell smoothing** takes the diversity of possible extensions of a history into account.

Witten–Bell smoothing

When defining Witten–Bell smoothing we follow the template of recursive interpolation (recall Equation 7.31). First, let us define the number of possible extensions of a history $w_1, ..., w_{i-1}$ seen in the training data as

$$N_{1+}(w_1, ..., w_{n-1}, \bullet) = |\{w_n : c(w_1, ..., w_{n-1}, w_n) > 0\}| \tag{7.33}$$

Based on this, we define the lambda parameters in Equation (7.31) as

$$1 - \lambda_{w_1,...,w_{n-1}} = \frac{N_{1+}(w_1, ..., w_{n-1}, \bullet)}{N_{1+}(w_1, ..., w_{n-1}, \bullet) + \sum_{w_n} c(w_1, ..., w_{n-1}, w_n)} \tag{7.34}$$

Let us apply this to our two examples:

$$\begin{aligned} 1 - \lambda_{\text{spite}} &= \frac{N_{1+}(\text{spite}, \bullet)}{N_{1+}(\text{spite}, \bullet) + \sum_{w_n} c(\text{spite}, w_n)} \\ &= \frac{9}{9 + 993} = 0.00898 \\ 1 - \lambda_{\text{constant}} &= \frac{N_{1+}(\text{constant}, \bullet)}{N_{1+}(\text{constant}, \bullet) + \sum_{w_n} c(\text{constant}, w_n)} \\ &= \frac{415}{415 + 993} = 0.29474 \end{aligned} \tag{7.35}$$

The lambda values follow our intuition about how much weight should be given to the back-off model. For *spite*, we give it little weight

(0.00898), indicating that we have a good idea of what follows this word. For *constant*, we reserve much more probability mass for unseen events (0.29474).

Another informative example is a history that occurs only once. Here we set the lambda parameter to $\frac{1}{1+1} = 0.5$, also allowing for the high likelihood of unseen events.

7.3.5 Diversity of Histories

To conclude our review of smoothing methods, we add one more consideration, which will lead us to the most commonly used smoothing method today, **Kneser–Ney smoothing** [Kneser and Ney, 1995].

Kneser–Ney smoothing

Consider the role of the lower order n-gram models in the back-off models we have discussed so far. We rely on them if the history of higher order language models is rare or otherwise inconclusive. So, we may want to build the lower order n-gram models that fit this specific role.

The word *york* is a fairly frequent word in the Europarl corpus, it occurs there 477 times (as frequent as *foods*, *indicates* and *providers*). So, in a unigram language model, it would be given a respectable probability. However, when it occurs, it almost always directly follows *new* (473 times), forming the name of the American city New York. So, we expect *york* to be very likely after we have seen *new*, but otherwise it is a very rare word – it occurs otherwise only in rare mentions of the English city York: *in york* (once), *to york* (once) and *of york* (twice).

Recall that the unigram model is only used if the bigram model is inconclusive. We do not expect *york* much as the second word in unseen bigrams. So, in a back-off unigram model, we would want to give the word *york* much less probability than its raw count suggests.

diversity of histories

In other words, we want to take the **diversity of histories** into account. To this end, we define the count of histories for a word as

$$N_{1+}(\bullet w) = |\{w_i : c(w_i, w) > 0\}| \tag{7.36}$$

Recall that the usual maximum likelihood estimation of a unigram language model is

$$p_{\text{ML}}(w) = \frac{c(w)}{\sum_i c(w_i)} \tag{7.37}$$

In Kneser–Ney smoothing, we replace the raw counts with the count of histories for a word:

$$p_{\text{KN}}(w) = \frac{N_{1+}(\bullet w)}{\sum_{w_i} N_{1+}(w_i w)} \tag{7.38}$$

To come back to our example, instead of using the raw count of 477 for the word *york*, we now use the much lower count of four different

one-word histories. Contrast this with the equally frequent *indicates*, which occurs with 172 different one-word histories. Basing probabilities on the counts of histories results in a much higher unigram back-off probability for *indicates* than for *york*.

7.3.6 Modified Kneser–Ney Smoothing

We have argued for the interpolation of higher order n-gram models with lower order n-gram models. We have discussed methods that assign proper probability mass to each, and how the component n-gram models should be built. Now we have to put everything together.

First of all, let us combine the ideas behind interpolation and back-off and redefine our interpolation function as:

$$p_I(w_1 n|w_1, ..., w_{n-1}) = \begin{cases} \alpha(w_n|w_1, ..., w_{n-1}) & \text{if } c(w_1, ..., w_n) > 0 \\ \gamma(w_1, ..., w_{n-1})\, p_I(w_n|w_2, ..., w_{n-1}) & \text{otherwise} \end{cases} \tag{7.39}$$

Two functions are involved. For each n-gram in the corpus, we have a function α that relates to its probability. For each history, we have a function γ that relates to the probability mass reserved for unseen words following this history.

This formulation of interpolated back-off will guide us now in our definition of modified Kneser–Ney smoothing below.

Formula for α for highest order n-gram model

Chen and Goodman [1998] suggest a modified version of Kneser–Ney smoothing, which uses a method called **absolute discounting** to reduce the probability mass for seen events. For $c(w_1, ..., w_n) > 0$, absolute discounting implies we subtract a fixed value D with $0 \leq D \leq 1$ from the raw counts for the highest order n-gram model:

absolute discounting

$$\alpha(w_n|w_1, ..., w_{n-1}) = \frac{c(w_1, ..., w_n) - D}{\sum_w c(w_1, ..., w_{n-1}, w)} \tag{7.40}$$

Chen and Goodman [1998] argue that better results can be achieved by subtracting not a fixed value D from each n-gram count, but the discounting value should depend on the n-gram count itself. They suggest using three different discount values:

$$D(c) = \begin{cases} D_1 & \text{if } c = 1 \\ D_2 & \text{if } c = 2 \\ D_{3+} & \text{if } c \geq 3 \end{cases} \tag{7.41}$$

It turns out that optimal discounting parameters D_1, D_2, D_{3+} can be computed quite easily:

$$\begin{aligned} Y &= \frac{N_1}{N_1 + 2N_2} \\ D_1 &= 1 - 2Y\frac{N_2}{N_1} \\ D_2 &= 2 - 3Y\frac{N_3}{N_2} \\ D_{3+} &= 3 - 4Y\frac{N_4}{N_3} \end{aligned} \tag{7.42}$$

The values N_c are the counts of n-grams with exactly count c. Compare the discounting formulae for D_i to Good–Turing smoothing, which we discussed in Section 7.2.3 on page 192. Alternatively, the D_i parameter can be optimized using held-out data.

Formula for γ for highest order n-gram model

The probability mass we set aside through discounting is available for unseen events, which leads to a very straightforward definition of the γ function:

$$\gamma(w_1, ..., w_{n-1}) = \frac{\sum_{i \in \{1,2,3+\}} D_i N_i(w_1, ..., w_{n-1}\bullet)}{\sum_{w_n} c(w_1, ..., w_n)} \tag{7.43}$$

where the N_i for $i \in \{1, 2, 3+\}$ are computed based on the count of extensions of a history $w_1, ..., w_{n-1}$ with count 1, 2, and 3 or more, respectively.

We subtracted D_1 from each n-gram with count 1, so we have D_1 times the number of n-grams with count 1 available for γ, and so on. Note that the formula for γ resembles the formula we derived for Witten–Bell smoothing (Section 7.3.4 on page 199) where we argued for taking the diversity of predicted words into account, and not just basing the discounted probability mass on the frequency of a history.

Formula for α for lower order n-gram models

We argued that for lower order n-grams it is better to base the estimation of the probability distribution on the count of histories $N_{1+}(\bullet w)$ in which a word may appear, instead of the raw counts.

This leads us to change the formula for computing α for lower order n-gram models slightly from Equation (7.38) to

$$\alpha(w_n | w_1, ..., w_{n-1}) = \frac{N_{1+}(\bullet w_1, ..., w_n) - D}{\sum_w N_{1+}(\bullet w_1, ..., w_{n-1}, w)} \tag{7.44}$$

Again, we use three different values for D (D_1, D_2, D_{3+}), estimated as specified in Equation (7.42), based on the count of the history $w_1, ..., w_{n-1}$.

Formula for γ for lower order n-gram models

As for the highest order n-gram model, the probability mass that was set aside by discounting the observed counts is available for the γ function:

$$\gamma(w_1, ..., w_{n-1}) = \frac{\sum_{i \in \{1,2,3+\}} D_i N_i(w_1, ..., w_{n-1}\bullet)}{\sum_{w_n} c(w_1, ..., w_n)} \tag{7.45}$$

Interpolated back-off

The back-off models we defined rely on the highest order n-gram that matches the history and predicted words. If these are sparse, they may be not very reliable. If two different n-grams with the same history occur once in the training data, we assign their predicted words the same probability. However, one may be an outlier, and the other an under-representative sample of a relatively common occurrence. Keep in mind that if a history is not very frequent, then the observed n-grams with that history are sparse counts.

To remedy this, we may always consider the lower order back-off models, even if we have seen the n-gram. We can do this simply by adapting the α function into an interpolated α_I function by adding the back-off probability:

$$\begin{aligned} \alpha_I(w_n|w_1, ..., w_{n-1}) = \alpha(w_n|w_1, ..., w_{n-1}) \\ + \gamma(w_1, ..., w_{n-1})\, p_I(w_n|w_2, ..., w_{n-1}) \end{aligned} \tag{7.46}$$

Note that the values for the γ function need to be reduced accordingly.

7.3.7 Evaluation

Figure 7.8 compares the smoothing methods we described in this section. The figure shows perplexity results for different order n-gram language models, trained on the Europarl corpus. The increasingly sophisticated smoothing methods result in better (lower) perplexity.

Good–Turing smoothing is used here as a back-off method. Probability mass for unseen events is reserved for the lower order language model, which is estimated the same way.

Modified Kneser–Ney smoothing and especially the interpolated variant leads to lower perplexity, by about 5–10%. There is also a great

Smoothing method	Bigram	Trigram	4-gram
Good–Turing	96.2	62.9	59.9
Witten–Bell	97.1	63.8	60.4
Modified Kneser–Ney	95.4	61.6	58.6
Interpolated modified Kneser–Ney	94.5	59.3	54.0

Figure 7.8 Evaluation of smoothing methods: Perplexity for language models trained on the Europarl corpus.

benefit from using larger n-grams. Overall, the 4-gram interpolated modified Kneser–Ney language model has nearly half the perplexity of the bigram Good–Turing language model.

7.4 Managing the Size of the Model

Fortunately, we have access to vast quantities of monolingual text for many languages, especially English, and their use has been shown to be beneficial in statistical machine translation systems, especially if they cover a domain similar to the test data.

Parallel texts often contain tens to hundreds of millions of words of English. A Gigaword corpus of several billion words is available through the Linguistic Data Consortium (LDC). It is also not too hard to write a web crawler that downloads even more text, up to trillions of English words.

On the other hand, using language models trained on such large corpora in a machine translation decoder becomes a challenge. Once the language model no longer fits into the working memory of the machine, we have to resort to clever methods to manage the size of the models. This final section of the chapter will discuss such methods.

7.4.1 Number of Unique N-Grams

First, let us get a sense of the size of language models. How many distinct n-grams can we find in a large corpus? Figure 7.9 has some numbers for the Europarl corpus. This corpus of 29,501,088 words has 86,700 unique English words (including punctuation as well as beginning-of-sentence and end-of-sentence markers). This number is also referred to as the **vocabulary size**.

vocabulary size

Given this vocabulary size, $86{,}700^2 = 7{,}516{,}890{,}000$, about 7.5 billion, bigrams are possible. However, we find only about 2 million distinct bigrams in the corpus. Not every sequence of two words that is possible actually occurs in English text, which is more or less the point of n-gram language models.

Order	Unique n-grams	Singletons
unigram	86,700	33,447 (38.6%)
bigram	1,948,935	1,132,844 (58.1%)
trigram	8,092,798	6,022,286 (74.4%)
4-gram	15,303,847	13,081,621 (85.5%)
5-gram	19,882,175	18,324,577 (92.2%)

Figure 7.9 Number of unique n-grams in the English part of the Europarl corpus, which comprises 29,501,088 tokens (words and punctuation).

For larger n-grams, the count seems to be more constrained by the actual corpus size. We find that about every second 4-gram is unique in the corpus. While the number of possible n-grams is polynomial with vocabulary size, the number is also bound to be linear with the corpus size. Check Figure 7.9 again. The number of unique n-grams grows rapidly from 86,700 unigrams to 1,948,935 bigrams. But for the highest order the growth stalls; there are not many more unique 5-grams than unique 4-grams: 19,882,175 vs. 15,303,847. But then, the corpus size is 29,501,088, which is an upper limit on the number of unique n-grams.

Most of the unique higher order n-grams are **singletons**, i.e., they occur only once. For 5-grams, the ratio of singletons is 92.2%. Since there is diminishing value in such rarely occurring events, higher order singletons are often excluded from language model estimation, typically starting with trigrams.

singletons

7.4.2 Estimation on Disk

The first challenge is to *build* a language model that is too large to fit into the working memory of the computer. However, none of the estimation and smoothing methods require that everything is stored in memory.

Consider the maximum likelihood estimation of a conditional probability distribution. Given a history, we need to sum over all possible word predictions. So, we need to loop twice over the n-grams with a shared history: once to get their counts, and then – given the total count – to estimate their probability. To avoid reading n-gram counts from disk twice, we may want to store all n-grams with a shared history in memory. After writing out the estimated probabilities, we can delete the respective n-grams and their counts from the working memory. The number of n-grams we have to keep in memory at any time is bounded by the vocabulary size.

To enable such **estimation on disk** we need a file of all n-grams with counts, sorted by their histories. This can be done – again, on disk – by first going through the corpus and writing all occurring n-grams into a file, and then sorting it by history and collecting counts. Standard UNIX tools (`sort`, `uniq`) are available to do this very efficiently.

estimation on disk

Smoothing methods require additional statistics. For Good–Turing discounting, we need the counts of counts, i.e. how many n-grams occur once, how many occur twice, etc. These statistics are obtained straightforwardly from the n-gram count file.

Kneser–Ney smoothing requires additional statistics on the number of distinct predicted words for a given history, and the number of distinct histories for a given word. The latter requires an n-gram count file, sorted by predicted word, but again these statistics can be obtained by linearly processing sorted n-gram count files.

7.4.3 Efficient Data Structures

We need to store large language models both memory-efficiently and in a way that allows us to retrieve probabilities in a time-efficient manner.

trie

Typically, this is done using a data structure known as a **trie**.

Consider the following: We want to store the 4-gram probabilities for *the very large majority* and for *the very large number*. They share the history *the very large*, so we would like to store the probabilities for the two 4-grams without storing the histories twice. Even *the very big event* shares the first two words in the history, so we would like to avoid storing the common part of its history separately as well.

Figure 7.10 shows a fragment of a trie that contains language model probabilities. Each word in the history is a node that points to a list of possible extensions, i.e., subsequent words in existing histories. The final node is a table of language model probabilities for all possible predictions given the history of the preceding path.

When we want to look up the probability for an n-gram, we walk through the trie by following the words of the history until we reach the word prediction table, where we look up the language model probability.

In this case, the language model probability is simply the 4-gram probability stored in the language model:

$$\begin{aligned} p_{\text{LM}}(\textit{majority}|\textit{the very large}) &= p_4(\textit{majority}|\textit{the very large}) \\ &= \exp(-1.147) \end{aligned} \tag{7.47}$$

What happens when the 4-gram does not exist in the language model, i.e., when we want to look up the language model probability for *the very large amount*?

Let us state again the back-off formula for interpolated language models (compare with Equation 7.39 on page 201):

$$\begin{aligned} &p_{\text{LM}}(w_n|w_0,...,w_{n-1}) \\ &\quad = \begin{cases} p_n(w_n|w_1,...,w_{n-1}) & \text{if } c(w_1,...,w_n) > 0 \\ \text{backoff}(w_1,...,w_{n-1})\, p_{\text{LM}}(w_n|w_2,...,w_{n-1}) & \text{otherwise} \end{cases} \end{aligned} \tag{7.48}$$

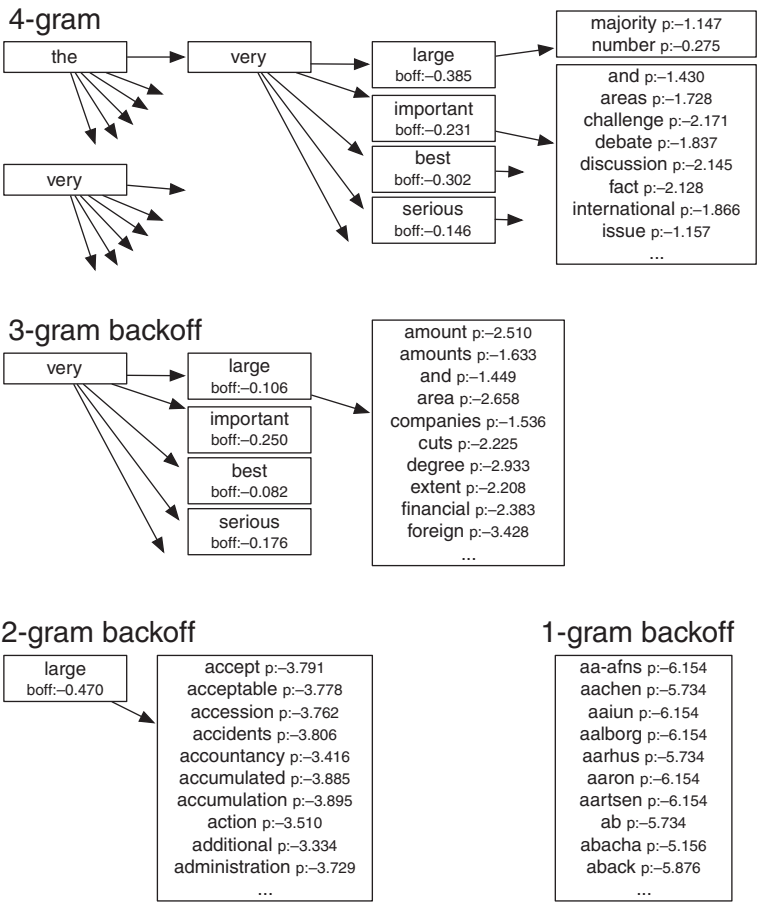

Figure 7.10 Language model probabilities stored in a trie: Starting with the first word of the history, we traverse the trie to the predicted word. If not found, we add a back-off cost and back off to a lower order language model. For instance *the very large amount* does not occur in the 4-gram LM, so we compute its LM probability by adding the back-off cost for *the very large* exp(−0.385) to the trigram language model cost for *very large amount* exp(−2.510).

If we do not find the n-gram in the language model, i.e., $c(w_1, ..., w_n) > 0$, we back off to lower order n-gram statistics. This implies a back-off cost which depends on the history. In the trie data structure, the back-off costs are stored at the penultimate nodes (the end point of the history).

In the case of *the very large amount*, we find probabilities at the trigram order. The language model probability is therefore (see also Figure 7.10):

$$\begin{aligned} p_{\text{LM}}(\textit{amount}|\textit{the very large}) &= \text{backoff}(\textit{the very large}) \\ &\quad + p_3(\textit{amount}|\textit{very large}) \\ &= \exp(-0.385 + -2.510) \end{aligned} \tag{7.49}$$

If we do not find the n-gram at reduced order, we back off further, incurring back-off costs, until we reach a prediction. See the following two examples which are ultimately resolved as bigram and unigram:

$$
\begin{aligned}
p_{\mathrm{LM}}(action|the\ very\ large) &= \mathrm{backoff}(the\ very\ large) \\
&\quad + \mathrm{backoff}(very\ large) \\
&\quad + p_2(action|large) \\
&= \exp(-0.385 + -0.106 + -3.510) \\
p_{\mathrm{LM}}(aaron|the\ very\ large) &= \mathrm{backoff}(the\ very\ large) \\
&\quad + \mathrm{backoff}(very\ large) \\
&\quad + \mathrm{backoff}(large) \\
&\quad + p_1(aaron) \\
&= \exp(-0.385 + -0.106 + -0.470 + -6.154)
\end{aligned} \tag{7.50}
$$

Note that if the history is unknown, we can back off with no cost.

Fewer bits to store probabilities

The trie data structure allows us to store n-grams more compactly by grouping their histories together. We may also make more efficient use of memory when storing words and probabilities.

Words are typically indexed, so that they are represented by integers (two bytes allow a vocabulary of $2^{16} = 65{,}536$ words). We can be even more efficient by employing Huffman coding, a common data compression method, to use fewer bits for frequent words.

Probabilities are typically stored in log format as floats (4 or 8 bytes). By using less memory to store these numbers, we can trade off accuracy against size of the language model.

Quantization of probabilities allows us to use even less memory, maybe just 4–8 bits to store these probabilities [Federico and Bertoldi, 2006]. The occurring probability values are grouped into bins, forming a code table. The bin number is stored in the trie data structure, or more specifically, in the word prediction table for a given history.

7.4.4 Reducing Vocabulary Size

We will now look at various ways to cut down on the number of n-grams that need to be stored in memory. First, let us focus on the vocabulary. It is true that there is a somewhat limited number of words that constitute the English language, maybe around 100,000. But if we process large collections of text, we will quickly find that the count of unique tokens easily rises to a multiple of that number.

Names, numbers, misspellings, and random material in the corpus will continually create new tokens. Numbers are an especially fertile source for new tokens in text, since there are infinitely many of them. Since different numbers in text do not behave that differently from

each other, we could simply group them all together into a special **NUMBER token**. NUMBER token

However, not all numbers occur in text in the same way. For instance, we may still want to be able to make a distinction in the case:

$$p_{\text{LM}}(\text{I pay 950.00 in May 2007}) > p_{\text{LM}}(\text{I pay 2007 in May 950.00}) \quad (7.51)$$

This would not be possible if we reduced this distinction to

$$p_{\text{LM}}(\text{I pay NUM in May NUM}) = p_{\text{LM}}(\text{I pay NUM in May NUM}) \quad (7.52)$$

A trick that is often applied here is to replace all digits with a unique symbol, such as @, or 5. So we are still able to distinguish between four-digit integers like *2007* (replaced by *5555*), which often follow words like *May*, and fractional numbers like *950.00* (replaced by *555.55*):

$$p_{\text{LM}}(\text{I pay 555.55 in May 5555}) > p_{\text{LM}}(\text{I pay 5555 in May 555.55}) \quad (7.53)$$

In our effort to reduce the vocabulary size for the language model, let us consider how the language model will be used. It is part of an application where one module produces words. A speech recognizer only produces words that are in the vocabulary of recognizable words. Similarly, a machine translation system only produces words that are in the translation table.

When using a language model for these applications, we can safely discard specifics about words that cannot be produced by the system. We may simply replace all **un-producible tokens** with a special token UNKNOWN. un-producible token

One has to keep in mind the impact of this on various smoothing methods. For instance, Kneser–Ney smoothing is sensitive to the number of *different* words that may occur in a specific position. If we collapse many different words into the same token, we may not be able to count this number correctly anymore.

7.4.5 Extracting Relevant n-Grams

Our concerns about the size of the language model arise from the limitations of modern computers that (currently) have main memories of a few gigabytes. However, modern disk array storage is measured in terabytes, which would fit even the largest collection of English text, so one idea is to keep language models on disk.

While it is possible to build huge language models from corpora such as the Gigaword corpus or data collected from the web, only a fraction of it will be required when translating a particular segment of

text. The translation model will produce only a small number of English words, given the foreign input words and the translation tables. At most, we need the probabilities for n-grams that contain only these English words.

Hence, we may want to filter the language model down to the n-grams that are needed for decoding a specific segment of a foreign sentence. The language model on disk may be huge, but the language model needed for decoding and kept in the working memory of the machine is much smaller.

bag-of-words

Figure 7.11 gives some idea of the size of the language model needed for decoding one sentence using the **bag-of-words** approach to language model filtering. When translating one sentence, we consult the phrase translation table and obtain the English words that may be produced by the model. We then restrict the language model to this bag of English words, i.e., we filter out all n-grams that contain at least one word that is not in this bag. Such n-grams could never be requested during decoding, so we can exclude them.

To illustrate this on one example, we examine how many 5-grams are left after bag-of-words filtering. We use the Europarl corpus (about 30 million words). The language model includes all 5-grams that occur at least twice. The test set is taken from the WMT 2006 shared task [Koehn and Monz, 2006], which comes from a held-out portion of the Europarl corpus plus additional News Commentary texts.

For most sentences, filtering limits the ratio of required 5-grams to about 3–5% of the total; some really long sentences of over 80 words in length require more than 10% of all 5-grams. This is still a remarkably

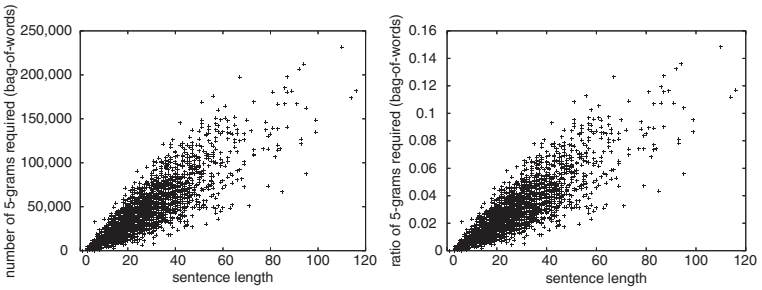

Figure 7.11 Number of 5-grams required to translate one sentence (bag of words approach): This graph plots the sentence length against the number of 5-grams that contain only English words that occur in the translation options, for one sentence each. The right-hand graph displays the ratio of 5-grams required to all 5-grams in the language model (German–English phrase model trained on Europarl, tested on WMT 2006 test set).

high ratio: It means that some 100,000 5-grams include only the few hundred words that are considered during the decoding of one sentence.

7.4.6 Loading N-Grams on Demand

The bag-of-words approach to language model filtering may overestimate the subset of the language model that is needed for decoding. So, it may be preferable to load n-gram statistics only **on demand**. A much larger set of n-grams could be stored on disk or on a cluster of networked machines.

on demand

This raises the question: how many n-gram requests are really made to the language model during decoding? Figure 7.12 answers this question for the example setting we just described.

The number of n-grams grows linearly with the input sentence length, here about 5,000 distinct 5-gram requests are made for each input word that is decoded (the total number of 5-gram requests is about five times this rate). This number depends on the number of translation options (here a maximum of 20 per input phrase), reordering limit (here 6), and stack size of the decoder (here 100).

When we request an n-gram probability, this n-gram may not exist, and the language model backs off to lower order n-grams. Figure 7.13 shows at what order 5-gram requests to the language model are resolved in this experiment. Only 0.75% of 5-gram requests match an existing 5-gram in the model. This is largely due to the nature of the decoding algorithm that produces a lot of nonsensical English word combinations. Most 5-gram requests are resolved as trigrams (24.4%), bigrams (53.0%), or unigrams (16.5%).

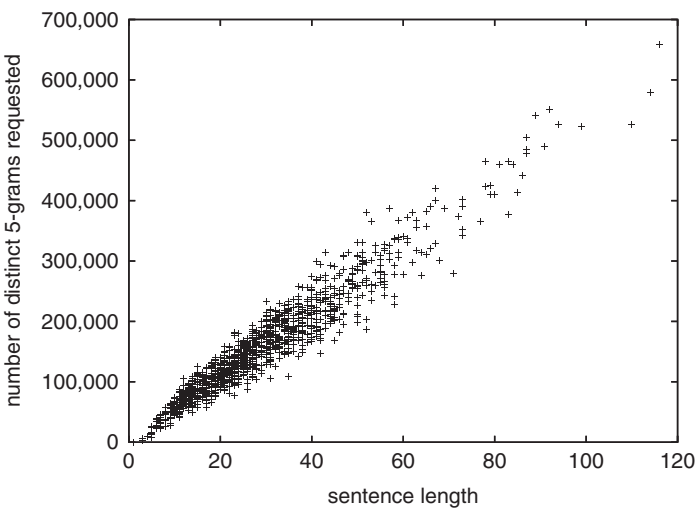

Figure 7.12 Number of distinct 5-gram language model requests: The graph plots the sentence length against the number of distinct 5-gram requests when translating one sentence each.

	5-gram	4-gram	Trigram	Bigram	Unigram
Resolved at order	0.75%	5.3%	24.4%	53.0%	16.5%
Unique resolutions	0.75%	2.1%	4.1%	3.1%	0.1%

Figure 7.13 Resolution of 5-gram decoder requests to the language model. Only a small fraction (0.75%) match a 5-gram in the language model; most often the language model backs off to bigrams (53.0%) or trigrams (24.4%). Hits to lower order n-grams are resolved with a few distinct n-grams; most clearly unigram resolutions end up at only a handful of words. On average, of 1,000 unique 5-gram requests, 165 are resolved as unigrams (16.5%) and hit the same word (0.1%).

Many distinct 5-gram requests that are resolved as, say, bigrams may resolve as the same bigram. For instance, looking up the probabilities for *doll chinese fireworks he says* and *commission fish institution he says* may both be resolved as $p_2(\text{says}|\text{he})$, since the full histories are unknown. This means that even fewer existing n-gram statistics are used than the number of 5,000 distinct n-gram requests per input word suggests. According to the findings, only about 500 distinct n-grams per input word need to be stored (we are ignoring here the storage of back-off histories).

7.5 Summary

7.5.1 Core Concepts

This chapter described the estimation and efficient use of **language models** for statistical machine translation. Language models provide the probability that a given string of English words is uttered in that language, thus helping the machine translation system to produce fluent output.

We decompose the problem into a sequence of word predictions using **n-gram** statistics, i.e., strings of length n. The first $n-1$ words are the **history** of the **predicted word** n. This type of model is called a **Markov chain**, with the **Markov assumption** that only a limited number of $n-1$ previous word states matter (this is an independence assumption). The size n is called the **order** of the language model. N-grams of size 1, 2, and 3 are called **unigrams**, **bigrams**, and **trigrams**, respectively. There are many methods for the **estimation of language models**. The quality of the resulting model is measured by its **perplexity**, which is related to its **cross-entropy** on a test set. **Entropy** measures uncertainty in a probability distribution. The entropy per word is called the **entropy rate**. We discussed methods to estimate upper

bounds for the **true entropy of a language** L, starting with Shannon's guessing game.

Since we will not observe all possible n-grams in a finite training corpus, we have to deal with the problem of **sparse data**. By **smoothing** the **empirical counts** we mean discounting the actual counts and saving some probability mass for unseen events. Ideally, the discounted counts match **expected counts** of a word given a history. The simplest form of smoothing is **add-one smoothing**, where we add one to any event. Due to the very skewed **Zipfian distribution** of words and n-grams in a natural language, this tends to overestimate unseen events. This can be somewhat alleviated by adding not 1, but a small value α. A more principled way to find estimates for the expected counts is **deleted estimation**, where we use part of the data as training data to count n-grams, and the remaining part to validate the counts by checking how often n-grams of a certain count actually occur. Another way of discounting is **Good–Turing smoothing**, which does not require held-out validation data.

Besides discounting counts in an n-gram model of fixed size n, we may also use **interpolation** to combine higher order n-gram models with lower order n-gram models. In **recursive interpolation**, we assign some weight λ to the higher order n-gram model, and the remaining weight to the interpolated lower order n-gram model. In **back-off** models, we use higher order n-gram models if the history exists with a minimum count, otherwise we back off to the lower order n-gram back-off model. There are several methods for defining the weights given to the higher order and to the lower order back-off model. **Witten–Bell smoothing** takes the diversity of predicted words for a given history into account. **Kneser–Ney smoothing** also considers the diversity of histories of a given predicted word for the estimation of lower order back-off models. Modified Kneser–Ney smoothing uses **absolute discounting**, which subtracts a fixed number D from each observed count (with special handling of counts 1 and 2).

Large monolingual corpora allow the estimation of large language models that do not fit into the working memory of modern computers, but we can resort to **estimation on disk**. Also, using such large language models in a statistical machine translation decoder is a problem. Language model size is often reduced by eliminating **singletons**, i.e., n-grams that occur only once. We can also reduce **vocabulary size** by simplifying numbers and removing words that cannot be generated by the translation model. Language models are stored in an efficient data structure called a **trie**. Efficiency can be further improved by using

fewer bits to store word indexes and language model probabilities. When translating a single sentence, the required size of the language model to be stored in memory can be further reduced by **bag-of-words filtering**, or by requesting n-grams **on demand** from disk or a cluster of machines.

7.5.2 Further Reading

Introduction – The discount methods we present in this chapter were proposed by Good [1953] – see also the description by Gale and Sampson [1995] – Witten and Bell [1991], as well as Kneser and Ney [1995]. A good introduction to the topic of language modelling is given by Chen and Goodman [1998]. Instead of training language models, large corpora can also be exploited by checking whether potential translations occur in them as sentences [Soricut *et al.*, 2002].

Targeted language models – Zhao *et al.* [2004a] describe a method for extracting training data for targeted language models based on similarity to the n-best output of a first decoding stage. Hasan and Ney [2005] cluster sentences according to syntactic types (i.e., questions vs. statements), train a separate language model for each cluster and interpolate it with the full language model for coverage.

Use of morphology in language models – Factored language models are also used in statistical machine translation systems [Kirchhoff and Yang, 2005]. For morphologically rich languages, better language models may predict individual morphemes. Sarikaya and Deng [2007] propose a joint morphological-lexical language model that uses maximum entropy classifiers to predict each morpheme.

Using very large language models – Very large language models that require more space than the available working memory may be distributed over a cluster of machines, but may not need sophisticated smoothing methods [Brants *et al.*, 2007]. Alternatively, storing the language model on disk using memory mapping is an option [Federico and Cettolo, 2007]. Methods for quantizing language model probabilities are presented by Federico and Bertoldi [2006], who also examine this for translation model probabilities. Alternatively, lossy data structures such as bloom filters may be used to store very large language models efficiently [Talbot and Osborne, 2007a,b]. Schwenk *et al.* [2006b] introduce continuous space language models that are trained using neural networks. The use of very large language models is often reduced to a re-ranking stage [Olteanu *et al.*, 2006b].

7.5.3 Exercises

1. (⋆) Given the following statistics:

Count of counts statistics:

Count	Count of counts
1	5000
2	1600
3	800
4	500
5	300

The word *beer* occurs as history in three bigrams in the data:

Count	Bigram
4	*beer drinker*
4	*beer lover*
2	*beer glass*

   (a) What are the discounted counts under Good–Turing discounting for the three given bigrams?
   (b) The amounts from discounting counts are given to a back-off unigram model. Using such a back-off model, what are the probabilities for the following bigrams?
      (i) p(drinker|beer)
      (ii) p(glass|beer)
      (iii) p(mug|beer)

   Note: $p(\text{mug}) = 0.01$. State any assumptions that you make.
2. (⋆⋆) Download the SRILM language modeling toolkit[2] and train a language model using the English text from the Europarl corpus[3].
3. (⋆⋆) Some decoding algorithms require the computation of the cheapest context a word or phrase may occur in. Given the ARPA file format used in the SRILM language modeling toolkit, implement a function that computes
   (a) the cost of the cheapest word following a given context;
   (b) the cost of the cheapest context a given word may occur in.
4. (⋆⋆) Implement a program to estimate a language model in a number of steps. You may want to compare the output of your program against the SRILM language modeling toolkit.
   (a) Collect distinct bigrams from a corpus and compute their counts.
   (b) Compute the count of counts.
   (c) Implement Good–Turing discounting for the bigram counts with the unigram model as back-off.

[2] Available at `http://www.speech.sri.com/projects/srilm/`
[3] Available at `http://www.statmt.org/europarl/`

(d) Make this program flexible, so it works with arbitrary n-gram sizes (e.g., trigram with bigram and unigram as back-off).
(e) Implement Kneser–Ney discounting.
(f) Implement interpolated Kneser–Ney discounting.
(g) Use the language model to generate random English sentences.

5. (★★★) Adapt your language model estimation program so that it works efficiently with on-disk data files and minimal use of RAM.