Chapter 3

Language Models

3.1 Introduction

3.1.1 Noise and the noisy-channel model

All the forms of language we might encounter in real life are susceptible to *noise*:

- Speech suffers from both speech errors and physical background noise.
- Handwriting can be messy, and the paper can have dirt on it or be damaged.
- Similarly, older printing can have defects (damaged type), and the paper can have dirt on it or be damaged.
- Typing can have typographical errors, especially on mobile devices.

We can think of the process of typing as a two step process: First, think of something to type, w. (The notation w comes from formal language theory and stands for "word," even though most of the time in this class, w is a sentence.) Second, type out w, possibly making some mistakes in the process. Mathematically:

$$P(w, s) = P(w)P(s \mid w).$$

Then, if we are given a sequence of characters s, we can reconstruct w by finding:

$$\hat{w} = \arg\max_{s} P(w)P(s \mid w).$$

The probability distribution $P(w)$ is called a *language model*. It says what kinds of sentences are more likely than others. It is of central importance to all the tasks that we will talk about in this first unit, so it is the topic of this chapter.

3.1.2 What should language models model?

If all the noise is removed, what is language in its pure form? If we take the view (as I do, and I believe most linguists would) that "pure" language is some mental representation, then when we produce any of the above forms of language, there are additional kinds of "noise":

- Speech undergoes some kind of transformation on its way from mental representation to the act of speaking: for example, we think of the /p/ sound in *pat* and *spat* as the same sound, but they are in fact pronounced differently.
- Spelling can be thought of as a non-trivial transformation from mental representations of words to sequences of letters. It is especially non-trivial in English, which is why there are spelling bees in English but not in other languages.

But we don't know what such a mental representation looks like, so for the purpose of building language technologies, it's far more convenient to pretend that pure language is digital text. So our language models are models of text, represented as sequences of words from a finite vocabulary.

3.2 *n*-grams

3.2.1 Model and training

The simplest kind of language model is the n-gram language model. A unigram (1-gram) language model is a bag-of-words model:

$$P(w_1 \cdots w_N) = \prod_{i=1}^{N} p(w_i) \times p(\texttt{</s>} \mid w_N). \tag{3.1}$$

A bigram (2-gram) language model is:

$$P(w_1 \cdots w_N) = p(w_1 \mid \texttt{<s>}) \times \prod_{i=2}^{N} p(w_i \mid w_{i-1}) \times p(\texttt{</s>} \mid w_N). \tag{3.2}$$

A general m-gram language model (we're going to use m instead of n, because n will be used for something else) is:

$$P(w_1 \cdots w_N) = \prod_{i=1}^{N+1} p(w_i \mid w_{i-m+1} \cdots w_{i-1}), \tag{3.3}$$

where we pretend that $w_i = \texttt{<s>}$ for $i \leq 0$, and $w_{N+1} = \texttt{</s>}$.

Training is easy. Let's use the following notation (Chen and Goodman, 1998):

N	number of word tokens (not including <s>)
$c(w)$	count of word w
$c(uw)$	count of bigram uw
$c(u\bullet)$	count of bigrams starting with u
n	number of word types ($= \lvert\Sigma\rvert$), including <unk>
n_r	number of word types seen exactly r times
$n_r(u\bullet)$	number of word types seen exactly r times after u
n_{r+}	number of word types seen at least r times
$n_{r+}(u\bullet)$	number of word types seen at least r times after u

For a bigram language model, we estimate

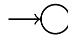
$$p(w \mid u) = \frac{c(uw)}{c(u\bullet)},$$

and similarly for m-grams in general.

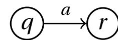
3.2.2 As weighted finite automata

We can also think of an n-gram model as a weighted finite automaton – this will become useful later on when we start combining language models with other models.

A *finite automaton (FA)* is typically represented by a directed graph. We draw nodes to represent the various *states* that the machine can be in. The node can be drawn with or without the state's name inside. The machine starts in the *initial state*, which we draw as:



The edges of the graph represent *transitions*, for example:



which means that if the machine is in state q and the next input symbol is a, then it can read in a and move to state r. The machine also has zero or more *final states*, which we draw as:

If the machine reaches the end of the string and is in a final state, then it accepts the string.

We say that a FA is *deterministic* if every state has the property that, for each label, there is exactly one exiting transition with that label. We will define nondeterministic FAs later.

Here's a more formal definition (for those who like formal definitions).

Definition 1. A *finite automaton* is a tuple $M = \langle Q, \Sigma, \delta, s, F \rangle$, where:

- Q is a finite set of *states*
- Σ is a finite alphabet
- δ is a set of *transitions* of the form $q \xrightarrow{a} r$, where $q, r \in Q$ and $a \in \Sigma$
- $s \in Q$ is the *initial state*
- $F \subseteq Q$ is the set of *final states*

A string $w = w_1 \cdots w_n \in \Sigma^*$ is accepted by M iff there is a sequence of states $q_0, \ldots, q_n \in Q$ such that $q_0 = s$, $q_n \in F$, and for all i, $1 \leq i \leq n$, there is a transition $q_{i-1} \xrightarrow{w_i} q_i$. We write $L(M)$ for the set of strings accepted by M.

A *weighted finite automaton* adds a *weight* to each transition (that is, δ is a mapping $Q \times \Sigma \times Q \to \mathbb{R}$). The weight of an accepting path through a weighted FA is the product of the weights of the transitions along the path. A weighted FA defines a weighted language, or a distribution over strings, in which the weight of a string is the sum of the weights of all accepting paths of the string.

In a *probabilistic FA*, each state has the property that the weights of all of the exiting transitions sum to one. Moreover, there is a special *stop symbol*, which we write as `</s>`, that we assume appears at the end of every string, and only at the end of every string. Then the weighted FA also defines a probability distribution over strings.

So, an n-gram language model is a probabilistic FA with a very simple structure. If we continue to assume a bigram language model, we need a state for every observed context, that is, one for `<s>`, which

we call $q_{\texttt{<s>}}$, and one for each word type a, which we call q_a. And we need a final state $q_{\texttt{</s>}}$. For all a, b, there is a transition

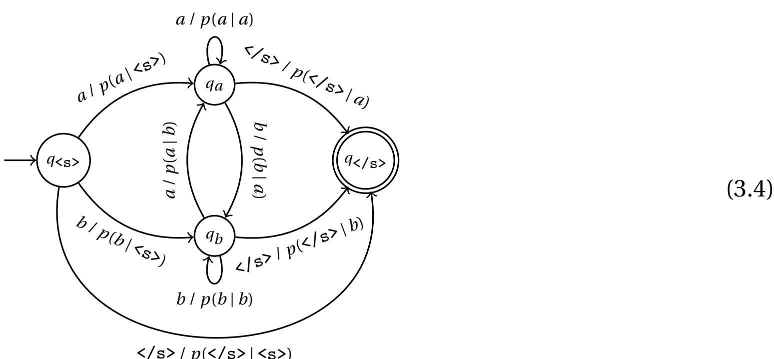
$$q_a \xrightarrow{b / p(b|a)} q_b,$$

and for every state q_a, there is a transition

$$q_a \xrightarrow{\texttt{</s>} / p(\texttt{</s>}|a)} q_{\texttt{</s>}}$$

.

The transition diagram (assuming an alphabet $\Sigma = \{a, b\}$) looks like this:



(3.4)

If we are given a collection of strings $w^1, \ldots, w^N$ and a DFA M, we can learn weights very easily. For each string w^i, run M on w^i and collect, for each state q, counts $c(q)$, $c(q, a)$ for each word a, and $c(q, \texttt{</s>})$, which is the number of times that M stops in state q. Then the weight of transition $q \xrightarrow{a} r$ is $\frac{c(q,a)}{c(q)}$.

If the automaton is not deterministic, the above won't work because for a given string, there might be more than one path that accepts it, and we don't know which path's transitions to count. Training nondeterministic automata is the subject of a later chapter.

3.3 Evaluation

3.3.1 Extrinsic evaluation

Language models are best evaluated extrinsically, that is, how much they help the application (e.g., speech recognition or machine translation) in which they are embedded.

3.3.2 Generating random sentences

One popular way of demonstrating a language model is using it to generate random sentences. While this is entertaining and can give a qualitative sense of what kinds of information a language model does and doesn't capture, but it is *not* a rigorous way to evaluate language models. Why? Imagine a whole-sentence language model

$$P(w) = \frac{c(w)}{N}$$

where N is the number of sentences in the training data. This model would randomly generate perfectly-formed sentences. But if you gave it a sentence w not seen in the training data, it would give w a probability of zero.

3.3.3 Perplexity

But for intrinsic evaluation, the standard evaluation metric is (per-word) *perplexity*:

$$\text{perplexity} = 2^{\text{cross-entropy}} \tag{3.5}$$

$$\text{cross-entropy} = \frac{1}{N} \log_2 \text{likelihood} \tag{3.6}$$

$$\text{likelihood} = P(w_1 \cdots w_N) \tag{3.7}$$

A lower perplexity is better.

Perplexity should be computed on held-out data, that is, data that is different from the training data. But held-out data is always going to have unknown words (words not seen in the training data), which require some special care. For if a language model assigns zero probability to unknown words, then it will have a perplexity of infinity. But if it assigns a nonzero probability to unknown words, how can it sum to one if it doesn't know how many unknown word types there are?

If we compare two language models, we should ensure that they have exactly the same vocabulary. Then, when calculating perplexity, we can either skip unknown words, or we can merge them all into a single unknown word type, usually written `<unk>`. But if two language models have different vocabularies, there isn't an easy way to make a fair comparison between them.

3.4 Smoothing unigrams

How can a model assign nonzero probability to unknown words? As with bag of words models, we need to apply smoothing. Smoothing is a large and complicated subject; we'll try to cover the main ideas here, but for a full treatment, the authoritative reference is the technical report by Chen and Goodman (1998).

Let's first think about how to smooth unigram models. Smoothing always involves taking counts away from some events and giving those counts back to some (other) events. We'll look at three schemes below. We assume that there's just one unknown word type, `<unk>`.

3.4.1 Limiting the vocabulary

Possibly the simplest scheme is simply to pretend that some word (types) seen in the training data are unknown. For example, we might limit the vocabulary to 10,000 word types, and all other word types are changed to `<unk>`. Or, we might limit the vocabulary just to those seen five or more times, and all other word types are changed to `<unk>`.

This method is used only in situations where it's inconvenient to use a better smoothing method. It's common, for instance, in neural language models.

3.4.2 Additive smoothing

Another very simple scheme is called add-one smoothing: pretend that we've seen every word type one more time than we actually have. More generally, we can add δ instead of adding one:

$$p(w) = \frac{c(w) + \delta}{N + n\delta}. \tag{3.8}$$

Another way to write this is as a mixture of the maximum-likelihood estimate and the uniform distribution:

$$p(w) = \lambda \frac{c(w)}{N} + (1 - \lambda)\frac{1}{n}, \tag{3.9}$$

where

$$\lambda = \frac{N}{N + n\delta}. \tag{3.10}$$

This way of writing it also makes it clear that although we're adding to the count of every word type, we're not adding to the probability of every word type (because the probabilities still have to sum to one). Rather, we "tax" each word type's probability at a flat rate λ, and redistribute it equally to all word types.

How are δ or λ determined? They can be optimized on held-out data (why does it have to be held-out?), but there are theoretical justifications for various magic values:

- $\delta = 1$ is called add-one or Laplace smoothing
- $\delta = \frac{n_{1+}}{n}$ or, equivalently, $\lambda = \frac{N}{N + n_{1+}}$, is called Witten-Bell smoothing (Witten and Bell, 1991)

Ultimately, you just have to try different values and see what works best.

3.4.3 Absolute discounting

As a taxation system, additive smoothing might have some merit, but as a smoothing method, it doesn't make a lot of sense. We know that there are words besides the ones we saw in the data, and therefore it makes sense to believe that the counts we get from the data are too high. But additive smoothing can decrease the count of a frequent word like 'the' by a lot, as though we had observed it by mistake thousands of times. Intuitively, smoothing shouldn't decrease the count of a word by more than about one. This is the idea behind *absolute discounting*. It takes an equal amount d ($0 < d < 1$) from every seen word type and redistributes it equally to all word types:[1]

$$p(w) = \frac{\max(0, c(w) - d)}{N} + \frac{n_{1+}d}{N}\frac{1}{n}. \tag{3.11}$$

How is d determined? Most commonly, the following formula (Ney, Essen, and Kneser, 1994) is used,

$$d = \frac{n_1}{n_1 + 2n_2}. \tag{3.12}$$

The interesting historical background to this method is that it is based on Good-Turing smoothing, which was invented by Alan Turing while trying to break the German Enigma cipher during World War II.

[1] In the original definition, the subtracted counts were all given to `<unk>`. But Chen and Goodman introduced the variant shown here, called interpolated Kneser-Ney, and showed that it works better.

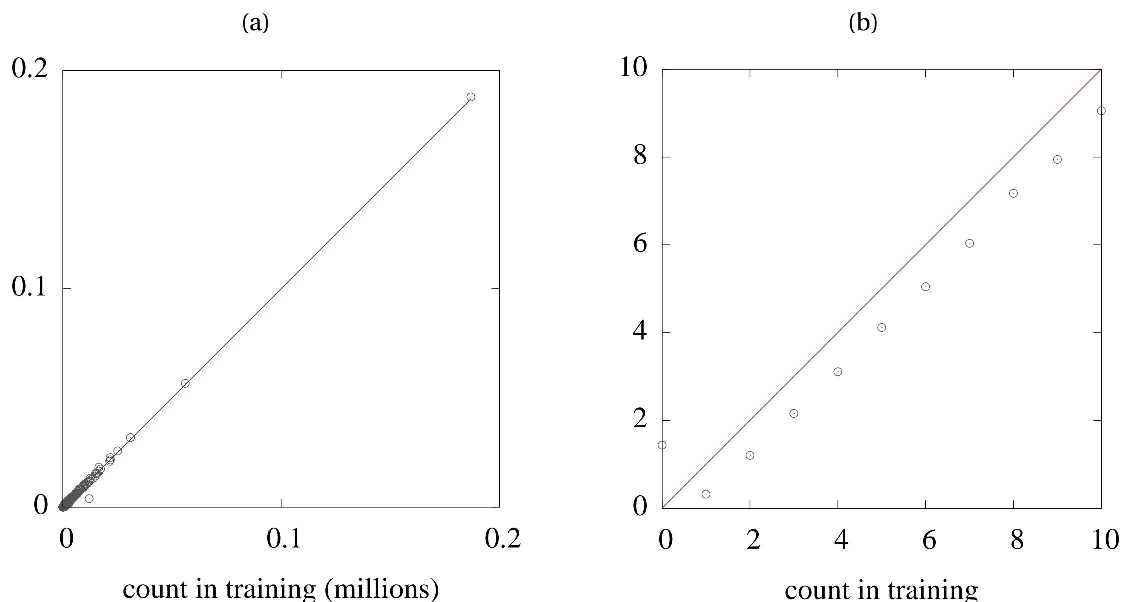


Figure 3.1: When we estimate word frequencies from training data (red line) and compare with actual word frequencies (circles), the estimates are generally good (a), but when we zoom in to rare words (b), we see that their frequencies are all overestimated.

3.4.4 Experiment

We took a 56M word of English text and used 10% of it as training data, 10% as heldout data, and the other 80% just to set the vocabulary (something we do not ordinarily have the luxury of).

We plotted the count of word types in the held-out data versus their count in the training data (multiplying the held-out counts by a constant factor so that they are on the same scale as the training counts) in Figure 3.1a. If a dot is above the red line, it means that it appeared in the heldout data more than the maximum-likelihood estimator predicted; if below, less. You can see that for large counts, the counts more or less agree, and maximum-likelihood is doing a fine job.

What about for low counts? We did the same thing but zooming into counts of ten or less, shown in Figure 3.1b. You can see that the held-out counts tend to be a bit lower than the maximum-likelihood estimate, with the exception of the zero-count words, which obviously couldn't be lower than the MLE, which is zero.

We also compared against Witten-Bell and absolute discounting. Although both methods do a good job of estimating the zero-count words, Witten-Bell stands out in overestimating all the other words. Absolute discounting does much better.

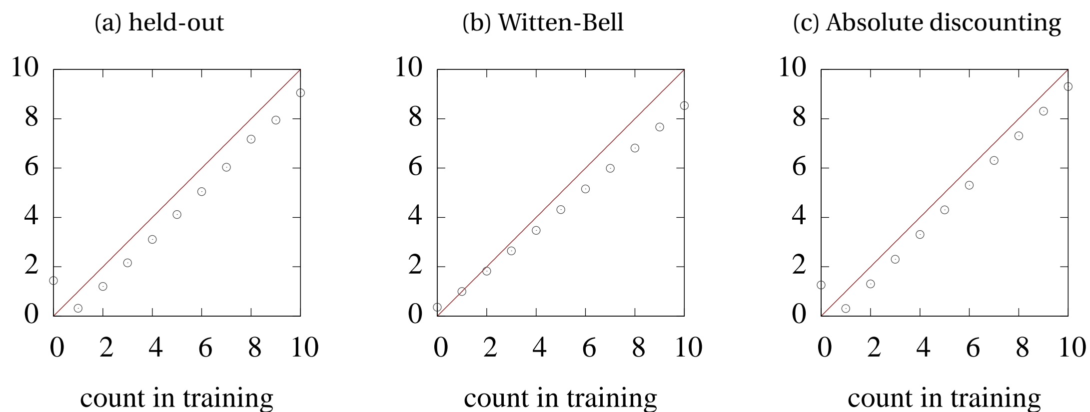


Figure 3.2: Comparison of different estimation methods, for a unigram language model. Graph (a) is the same as Figure 3.1b; graphs (b) and (c) show the estimates produced by other smoothing methods.

3.5 Smoothing bigrams and beyond

3.5.1 Bigrams

To smooth bigrams, we smooth each conditional distribution $p(w \mid u)$ using the same principles that we saw with unigrams. The "taking away" part remains exactly the same. But the "giving back" part is different. Previously, we gave back to each word type equally. But now, a better option is to give weight back to word types in proportion to their unigram probability $p(w)$. For example, suppose we've seen "therizinosaur" in the training data, but never "therizinosaur egg." It's not reasonable to say $p(\text{egg} \mid \text{therizinosaur}) = 0$; instead, it's better to let our estimate of $p(\text{egg} \mid \text{therizinosaur})$ be partly based on $p(\text{egg})$.

So, additive smoothing for bigrams looks like:

$$p(w \mid u) = \lambda(u)\frac{c(uw)}{c(u\bullet)} + (1-\lambda(u))p(w), \tag{3.13}$$

where λ now depends on u. In particular, for Witten-Bell smoothing, $\lambda(u) = \frac{c(u\bullet)}{c(u\bullet)+n_{1+}(u\bullet)}$. Absolute discounting for bigrams looks like:

$$p(w \mid u) = \frac{\max(0, c(uw)-d)}{c(u\bullet)} + \frac{n_{1+}(u\bullet)d}{c(u\bullet)}p(w) \tag{3.14}$$

where $p(w)$ is the (smoothed) unigram model. These are the same as (3.9) and (3.11), but with $p(w)$ replacing $1/n$.

3.5.2 Beyond bigrams

To smooth a general m-gram model, we smooth it with the $(m-1)$-gram model, which is in turn smoothed with the $(m-2)$-gram model, and so on down to the 1-gram model, which is smoothed with the uniform

distribution. If u is any string, define $\bar{u}$ to be u with the first word removed (for example, if u = the cat sat, then $\bar{u}$ = cat sat). Then additive smoothing looks like:

$$p(w \mid u) = \lambda(u) \frac{c(uw)}{c(u\bullet)} + (1 - \lambda(u))\, p(w \mid \bar{u}), \tag{3.15}$$

where, again, Witten-Bell says that $\lambda(u) = \frac{c(u\bullet)}{c(u\bullet)+n_{1+}(u\bullet)}$. Absolute discounting looks like

$$p(w \mid u) = \frac{\max(0, c(uw) - d)}{c(u\bullet)} + \frac{n_{1+}(u\bullet)\, d}{c(u\bullet)}\, p(w \mid \bar{u}). \tag{3.16}$$

Bibliography

Chen, Stanley F. and Joshua Goodman (1998). *An Empirical Study of Smoothing Techniques for Language Modeling*. Tech. rep. TR-10-98. Harvard University Center for Research in Computing Technology.

Ney, Hermann, Ute Essen, and Reinhard Kneser (1994). “On Structuring Probabilistic Dependencies in Stochastic Language Modelling”. In: *Computer Speech and Language* 8, pp. 1–38.

Witten, Ian H. and Timothy C. Bell (1991). “The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression”. In: *IEEE Trans. Information Theory* 37.4, pp. 1085–1094.