Chapter 5

Language Modeling

5.1 Introduction

A *language model* is simply a model of what strings (of words) are more or less likely to be generated by a speaker of English (or some other language). More specifically, it's a model that predicts what the next word will be, given all the words so far.

5.1.1 Applications

Predicting the next word has some direct applications – for example, your phone tries to predict what you will type, in order to make your typing easier.

Most importantly, however, language models are used in many applications whose output is English text: for example, in automatic speech recognition or machine translation, we need a language model in order to make the system prefer to output sentences that are well-formed English. We'll see how this is done in the next chapter.

Finally, we can use a language model as a fancier replacement for the bag-of-words model in text classification: if we build a different language model for each class, then we can classify documents using

$$P(k \mid d) \propto P(k)P(d \mid k)$$

where $P(d \mid k)$ is the class-specific language model.

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

$$\text{perplexity} = 2^{\text{cross-entropy}} \tag{5.1}$$

$$\text{cross-entropy} = -\frac{1}{N}\log_2 \text{likelihood} \tag{5.2}$$

$$\text{likelihood} = P(w_1 \cdots w_N) \tag{5.3}$$

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.

5.2 *n*-gram Language Models

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

$$P(w_1 \cdots w_N) = \prod_{i=2}^{N} p(w_i). \tag{5.4}$$

A bigram (2-gram) language model is:

$$P(w_1 \cdots w_N) = p(w_1 \mid \texttt{<s>}) \times \prod_{i=2}^{N} p(w_i \mid w_{i-1}) \times p(\texttt{</s>} \mid w_N). \tag{5.5}$$

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

$$P(w_1 \cdots w_N) = \prod_{i=1}^{N+1} p(w_i \mid w_{i-m+1} \cdots w_{i-1}), \tag{5.6}$$

where we pretend that $w_i = \texttt{<s>}$ for $i \le 0$, 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.

5.3 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>`.

5.3.1 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.

5.3.2 Additive smoothing

As before, we can do add-one or add-δ smoothing:

$$p(w) = \frac{c(w)+\delta}{N+n\delta}. \tag{5.7}$$

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

$$p(w) = \lambda \frac{c(w)}{N} + (1-\lambda)\frac{1}{n}, \tag{5.8}$$

where

$$\lambda = \frac{N}{N+n\delta}. \tag{5.9}$$

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
- $\delta = \frac{1}{2}$ is called expected likelihood estimation or Jeffreys-Perks 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.

5.3.3 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. For example, we looked at a sample of 56M words of English data, in which the word 'the' appears 3,579,493 times. Add-one-smoothing gives $p(\text{the}) = 0.0641$, which means that in some other equal-sized sample, the model would expect 'the' to occur 3,570,938 times. In other words, 8,555 of the occurrences of 'the' in our training data were somehow a fluke.

Intuitively, smoothing shouldn't decrease the expected count of a word (relative to its empirical count) 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]

$$p(w) = \frac{\max(0, c(w) - d)}{N} + \frac{n_{1+}d}{N}\frac{1}{n}. \tag{5.10}$$

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

$$d = \frac{n_1}{n_1 + 2n_2}. \tag{5.11}$$

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.

5.4 Experiment

We took the same 56M word sample as before 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 5.1. 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 5.2, upper-left corner. 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.

We get a similar picture when we look at bigrams (Figure 5.3), again smoothing with uniform probabilities. The held-out counts are again lower than the MLEs, by a bit more this time, except for the noticeable bump for zero-count words. (If we had used even more data to compute the vocabulary, this bump would be reduced.) Absolute discounting is nearly indistinguishable from the held-out counts. But Witten-Bell looks very different: here, you can see the effect of multiplying by λ.

[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.

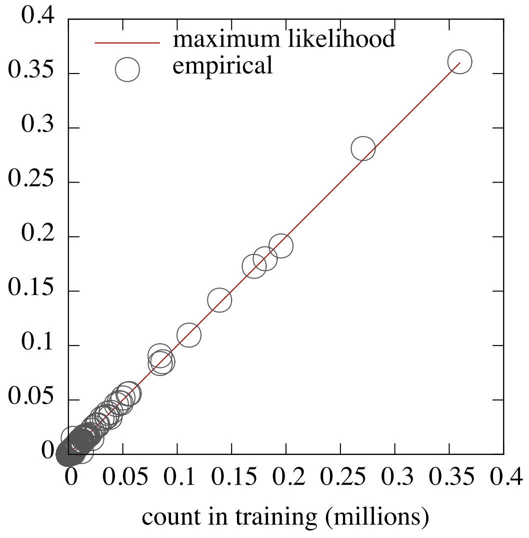


Figure 5.1: Counts in held-out data versus in training data. The held-out counts have been multiplied by a constant factor so as to be on the same scale as the training counts. The red line is what maximum-likelihood estimation would predict.

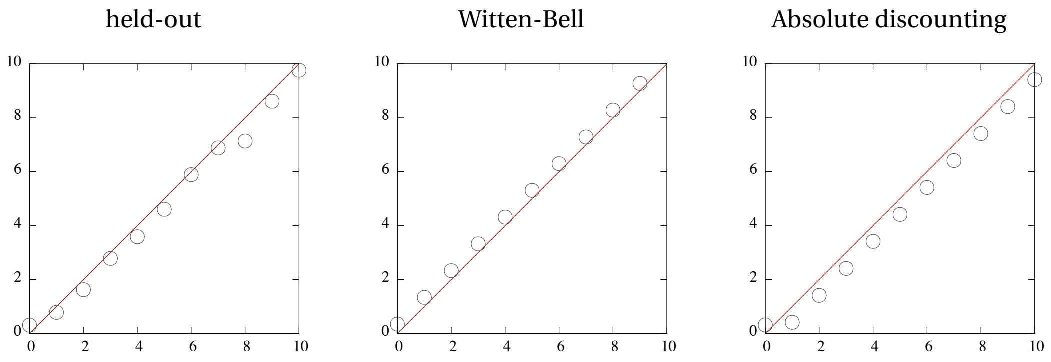


Figure 5.2: Comparison of different estimation methods, for a unigram language model. In all graphs, the x-axis is the count in the training data; the y-axis is the expected count in new data, adjusted for the size of the training data. The red line is what maximum-likelihood estimation would predict. The graph labeled "held-out" comes from actual counts on held-out data; all the other methods are trying to match this one.

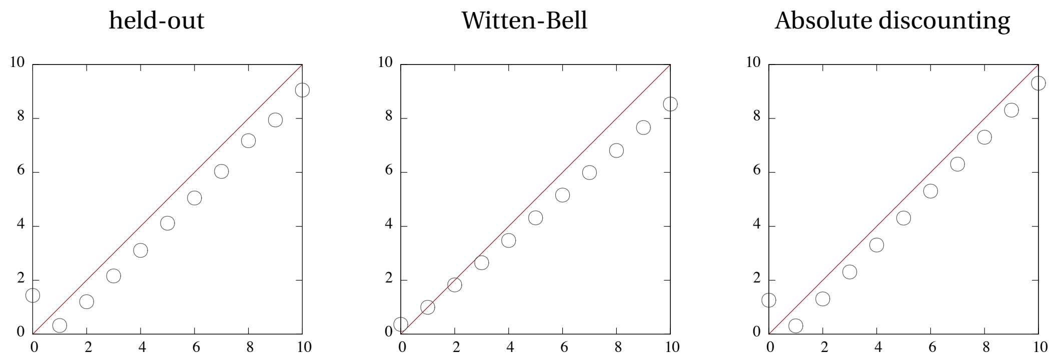


Figure 5.3: Comparison of different estimation methods, for a bigram language model. In all graphs, the x-axis is the count in the training data; the y-axis is the expected count in new data, adjusted for the size of the training data. The red line is what maximum-likelihood estimation would predict. The graph labeled “held-out” comes from actual counts on held-out data; all the other methods are trying to match this one.

5.5 Smoothing bigrams and beyond

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:

$$p(w \mid u) = \lambda(u)\frac{c(uw)}{c(u\bullet)} + (1-\lambda(u))p(w), \tag{5.12}$$

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:

$$p(w \mid u) = \frac{\max(0, c(uw)-d)}{c(u\bullet)} + \frac{n_{1+}(u\bullet)d}{c(u\bullet)}p(w) \tag{5.13}$$

where $p(w)$ is the (smoothed) unigram model. These are the same as (5.8) and (5.10), but with $p(w)$ replacing $1/n$.

Recursive smoothing 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 an $(m-1)$-gram, define $\bar{u}$ to be the $(m-2)$-gram suffix of u (for example, if u = the cat sat, then $\bar{u}$ = cat sat). Then additive smoothing looks like:

$$p(w \mid u) = \lambda(u)\frac{c(uw)}{c(u\bullet)} + (1-\lambda(u))p(w \mid \bar{u}), \tag{5.14}$$

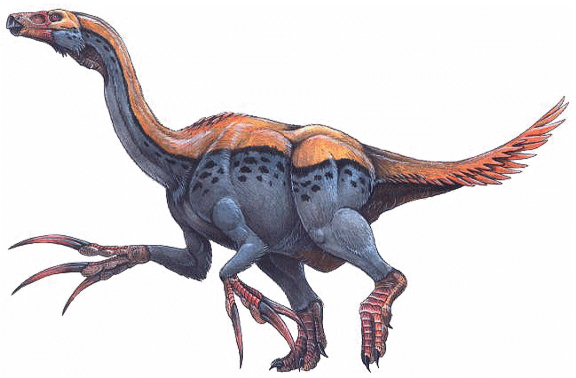


Figure 5.4: Therizinosaur.

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

$$p(w \mid u) = \frac{\max(0, c(uw) - d)}{c(u\bullet)} + \frac{n_{1+}(u\bullet)d}{c(u\bullet)} p(w \mid \bar{u}). \tag{5.15}$$

Kneser-Ney smoothing In absolute discounting for bigrams (5.13), we said that $p(w)$ was the smoothed unigram model. But there's a way to estimate $p(w)$ that makes this work better. The enhancement comes from *Kneser-Ney smoothing* (Ney, Essen, and Kneser, 1995), a form of absolute discounting. Consider the bigram "Notre Dame." In some data (say, web pages of people at Notre Dame), this bigram will be quite frequent. So that means that the unigram probabilities p(Notre) and p(Dame) are reasonably high too. But now think about P(Dame | therizinosaur) – should we estimate that using p(Dame)? No, that would be too high, because we know that "Dame" occurs practically only after "Notre."

The basic idea behind the fix is that we should train the unigram model only on the words that we took away from the bigram model. Every time we see a new bigram uw, feed it with a count of $(1-d)$ to the bigram model, and feed w with a count of 1 to the unigram model.[2] Every subsequent time we see uw, we feed it entirely to the bigram model and not at all to the unigram model. Thus, the unigram model only sees word w when it occurs in a new bigram type. This means that the unsmoothed unigram estimates are:

$$p(w) = \frac{n_{1+}(\bullet w)}{n_{1+}(\bullet\bullet)}. \tag{5.16}$$

Since "Dame" occurs only after "Notre," we have $n_{1+}(\bullet\text{Dame}) = 1$, so the unigram probability p(Dame) will also be low.

But wait! The unigram model should itself be smoothed, again by Kneser-Ney smoothing. This is something that is easier to code than to write in equations, but:

$$p(w) = \frac{\max(0, n_{1+}(\bullet w) - d^{(1)})}{n_{1+}(\bullet\bullet)} + \frac{n_{1+}(\bullet)d^{(1)}}{n_{1+}(\bullet\bullet)} \frac{1}{n}, \tag{5.17}$$

where we've written $d^{(1)}$ with a superscript to distinguish it from the d used in the bigram model.

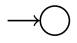
[2] A count of d might have made more sense, but the fractional count would be problematic. This is because the unigram model also needs to be smoothed, and equation (5.11) needs integral counts.

5.6 Finite Automata

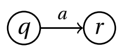
N-gram language models are just one example of a more general class of models that go beyond bag-of-word models by capturing dependencies between a word and its previous words. These models are based on weighted finite-state automata.

5.6.1 Definition

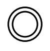
A *finite automaton (FA)* is typically represented by a directed graph. We draw nodes to represent the various *states* that the machine can be in. The node can be drawn with or without the state's name inside. The machine starts in the *initial state*, which we draw as:



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.

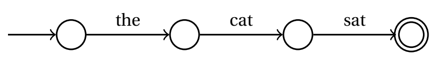
Here's a more formal definition.

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, $1 \le i \le n$, there is a transition $q_{i-1} \xrightarrow{w_i} q_i$. We write $L(M)$ for the set of strings accepted by M.

We will frequently make use of the following very simple construction. Given a string w, let the *singleton DFA* for w be the minimal DFA accepting $\{w\}$. For example, the singleton DFA for "the cat sat" is:



5.6.2 Intersection

You learned about intersection of finite automata in Theory, but here's a quick review. Given two finite automata $M_1 = (\Sigma, Q_1, s_1, F_1, \delta_1)$ and $M_2 = (\Sigma, Q_2, s_2, F_2, \delta_2)$, we can recognize the language $L(M_1) \cap L(M_2)$, intuitively, by feeding the input string simultaneously to both M_1 and M_2. When we reach the end of the input, if both machines are in an accept state, then we accept the string. Otherwise, reject.

We can build a finite automaton M that does exactly this. Namely, $M = (\Sigma, Q_1 \times Q_2, s_1 s_2, F_1 \times F_2, \delta)$, where $\delta(q_1 q_2, \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$. (We use $q_1 q_2$ as a shorthand for (q_1, q_2).)

Note that if M_1 and M_2 are deterministic, so is M.

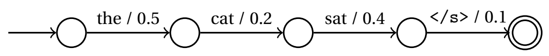
5.6.3 Weighted finite automata

A *weighted finite automaton* adds a *weight* to each transition (that is, δ is a mapping $Q \times \Sigma \times Q \to \mathbb{R}$) and a stop weight to each state. The weight of an accepting path through a weighted FA is the product of the weights of the transitions along the path, times the stop weight of the final state. 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 and the stop weight sum to one. Then the weighted FA also defines a probability distribution over strings.

Question 7. Prove the above statement.

In a transition diagram, there isn't really a nice way to write the stop weight. Some people write it inside the state, but we indicate stop weights by creating a pseudo-final state and a transition on the pseudo-symbol </s>:

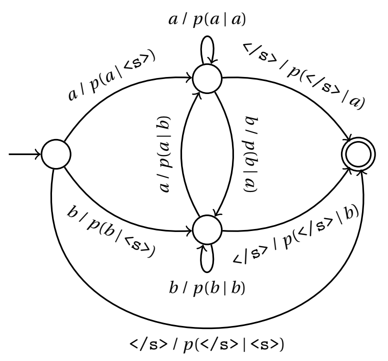


So, an n-gram language model is a probabilistic FA with a very simple structure. If we continue to assume a bigram language model, we need a state for every observed context, that is, one for <s>, which we call $q_{\texttt{<s>}}$ and one for each word type a, which we call q_a. Every state is a final state. For all a, b, there is a transition

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

and for every state q_a, the stop weight is $p(\texttt{</s>} \mid a)$.

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



(5.18)

5.6.4 Training

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 $\frac{c(q,a)}{c(q)}$, and the stop weight of q is $\frac{c(q,\texttt{</s>})}{c(q)}$. This is the weighted DFA that maximizes the likelihood of the training data $w^1, \ldots w^N$.

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.

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.

— (1995). “On the Estimation of ‘Small’ Probabilities by Leaving-One-Out”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 17.12, pp. 1202–1212.

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.