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). They 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 can also use a language model as a fancier replacement for the bag-of-words model in text classification.

5.2 n-gram Language Models

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

$$P(w_1 \cdots w_N) = \prod_{i=1}^{N} p(w_i). \tag{5.1}$$

A bigram (2-gram) language model is:

$$P(w_1 \cdots w_N) = p(w_1 \mid \texttt{<s>}) \times \prod_{i=1}^{N} p(w_i \mid w_{i-1}) \times p(\texttt{</s>} \mid w_N). \tag{5.2}$$

A general n-gram language model is:

$$P(w_1 \cdots w_N) = \prod_{i=1}^{N+1} p(w_i \mid w_{i-n+1} \cdots w_{i-1}), \tag{5.3}$$

where we pretend that $w_0 =$ <s> and $w_{N+1} =$ </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
n	number of word types ($= \lvert\Sigma\rvert$)
n_r	number of word types seen r times
n_{r+}	number of word types seen at least r times

A bullet ($\bullet$) means "sum over this." Thus, for example, $c(u\bullet) = \sum_w c(uw)$.

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 Evaluation

Language models are best evaluated extrinsically, that is, how much they help the application (e.g., ASR or MT) in which they are embedded. But for intrinsic evaluation, the standard evaluation metric is (per-word) *perplexity*:

$$\text{perplexity} = 2^{\text{cross-entropy}} \tag{5.4}$$

$$\text{cross-entropy} = \frac{1}{N}\log_2 \text{likelihood} \tag{5.5}$$

$$\text{likelihood} = P(w_1 \cdots w_N) \tag{5.6}$$

A lower perplexity is better.

Calculating the perplexity of unknown words is a nuisance. If we're comparing two language models using the same training data and the same test data, it's fine either to skip the unknown words or to replace them all with a single symbol <unk>.

5.4 Smoothing unigrams

What do we do about unknown words? As with bag of words models, we need to apply smoothing. Let's first think about how to smooth unigram models. Everything in this section could have been applied to naïve Bayes as well.

5.4.1 Additive smoothing

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

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

We could also write this as interpolation between the MLE 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, of course, because the probabilities 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.

Witten-Bell smoothing (Witten and Bell, 1991) will be described in more detail below because it's properly a method for smoothing bigrams and beyond, but it says that $\delta = n_{1+}/n$.

5.4.2 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 from every seen word, and then distributes it equally among all the unseen words:

$$p(w) = \begin{cases} \dfrac{c(w)}{N} - \dfrac{d}{N} & \text{if } c(w) > 0 \\ \dfrac{n_{1+}d}{n_0 N} & \text{otherwise.} \end{cases} \tag{5.10}$$

5.4.3 Leaving one out

Whenever we're faced with a problem like choosing d or δ – let's say d – where we know that we don't want the maximum likelihood estimate, the right thing to do is to make use of *held-out data*. Divide up the corpus into k pieces, estimate parameters from $(k-1)$ pieces, and compute the log-likelihood of the kth piece. The likelihood depends on d, and we can optimize d to maximize it. There isn't any danger of overfitting if the kth piece is independent of the other $(k-1)$ pieces.

If you don't like the idea of sacrificing $1/k$th of your data, then you can do *k-fold cross-validation*, which repeats this process k times ("folds"). The quantity maximized is the sum of the log-likelihoods over all k "folds".

If we let $k = N$, this is called *leaving one out*. The cool thing about leaving one out is that you often don't have to actually train your model N times; there's often a closed-form solution or at least a usable approximation.

Good-Turing discounting The oldest example of leaving one out wasn't just for a single parameter like d or δ, but estimating *all* the probabilities. Called *Good-Turing estimation* (Good, 1953), it was invented by Turing and published by Good; they developed it while breaking the German Enigma cipher during World War II.

Assume that if two words have the same count r, they should have the same probability, which we write p_r. Normally, the log-likelihood would be (assuming one long training sentence $w_1 \cdots w_N$ for sim-

plicity):

$$L = \sum_i \log p_{c(w_i)} \tag{5.11}$$

$$= \sum_w c(w) \log p_{c(w)} \tag{5.12}$$

$$= \sum_r r\, n_r \log p_r. \tag{5.13}$$

If we leave out a single word (token) w, the log-likelihood of that one word would be $\log p_{c(w)-1}$, because we would have observed w in the training data one fewer time. If we sum over all word (tokens), we get

$$L = \sum_{r \geq 1} r\, n_r \log p_{r-1} \tag{5.14}$$

which we want to maximize subject to the constraint

$$\sum_{r \geq 0} n_r\, p_r = 1. \tag{5.15}$$

The derivation is in Section 5.6, and the answer turns out to be

$$p_r = \frac{(r+1)\, n_{r+1}}{N n_r}. \tag{5.16}$$

The really interesting implication of this is that we get an estimate of the total probability of all *unseen* words: $n_0 p_0 = n_1 / N$.

Absolute discounting (Ney, Essen, and Kneser, 1994) is similar to Good-Turing, but assumes that the p_r for $r \geq 1$ have the form

$$p_r = \frac{r - d}{N}. \tag{5.17}$$

Unfortunately, there isn't a closed form solution for d like there is with Good-Turing estimation. But there is a commonly used upper bound:

$$d \leq \frac{n_1}{n_1 + 2n_2}, \tag{5.18}$$

whose derivation can be found in Section 5.6.

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.

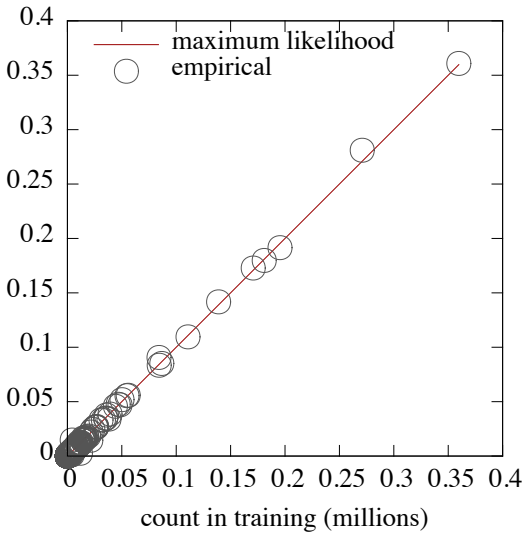


Figure 5.1: Counts in held-out data versus in training data. The held-out counts have been multiplied by a constant factor so as to be on the same scale as the training counts. The red line is what maximum-likelihood estimation would predict.

We also compared against three smoothing methods: Good-Turing, Witten-Bell, and absolute discounting. Although all the methods do a good job of estimating the zero-count words, Witten-Bell stands out in overestimating all the other words. Good-Turing is a little bit noisy, with a bump at 8. Absolute discounting looks like the winner here.

We get a similar picture when we look at bigram probabilities (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.) Good-Turing and absolute discounting are nearly indistinguishable from the held-out counts. But Witten-Bell looks very different: here, you can see the effect of multiplying by λ.

5.5 Smoothing bigrams

To smooth bigrams, we smooth each conditional distribution $p(w \mid u)$ using the same principles that we saw with unigrams. The "taxation" part remains exactly the same. But the "giving back" part is different because, instead of giving back to everyone equally, we give weight back to bigrams uw in proportion to the unigram probability $p(w)$.

5.5.1 Additive smoothing

Jelinek-Mercer smoothing Add-δ smoothing was probably already a questionable idea for unigrams, but for bigrams (and beyond), it is probably the worst way of smoothing a language model. At minimum,

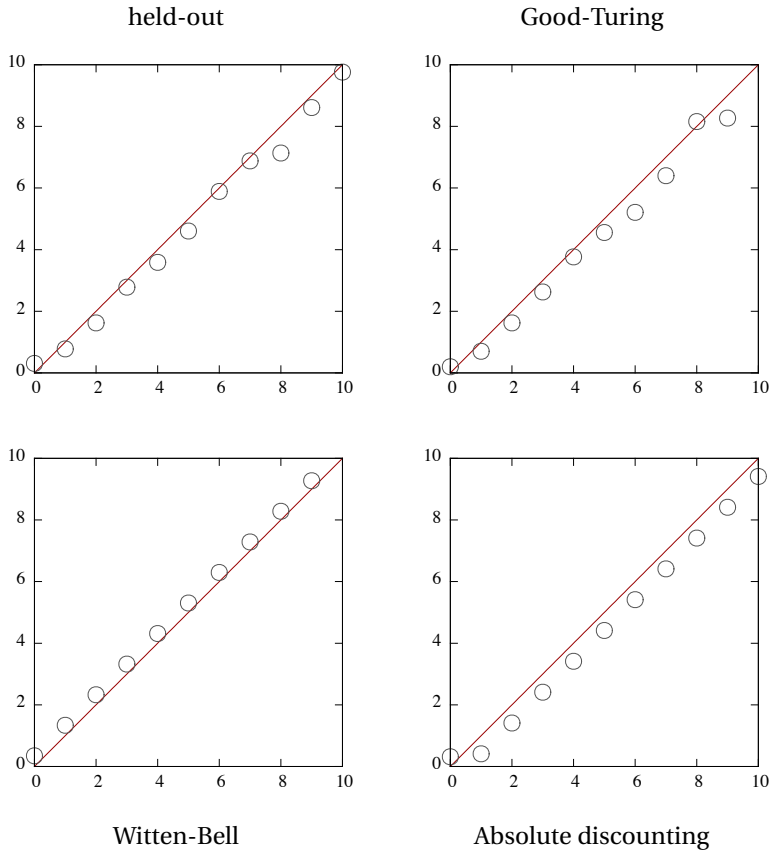


Figure 5.2: Comparison of different estimation methods, for a unigram language model. In all graphs, the x-axis is the count in the training data; the y-axis is the expected count in new data, adjusted for the size of the training data. The red line is what maximum-likelihood estimation would predict.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

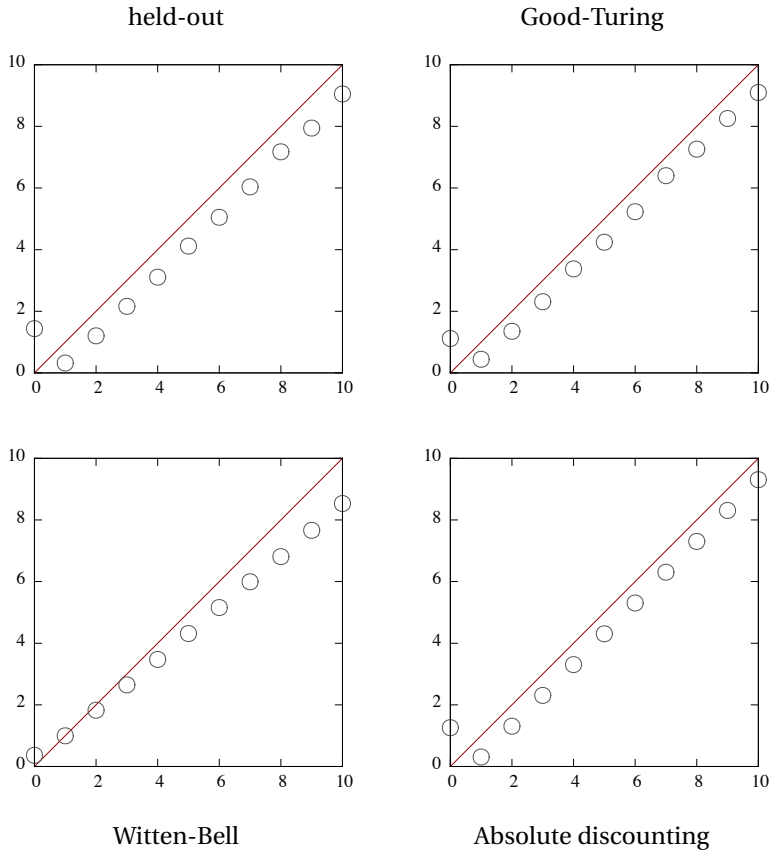


Figure 5.3: Comparison of different estimation methods, for a bigram language model. In all graphs, the x-axis is the count in the training data; the y-axis is the expected count in new data, adjusted for the size of the training data. The red line is what maximum-likelihood estimation would predict.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

we should add different pseudocounts to different bigrams uw proportional to the unigram probability of w:

$$p(w \mid u) = \frac{c(uw) + \alpha p(w)}{c(u) + \alpha} \tag{5.19}$$

$$= \lambda \frac{c(uw)}{c(u)} + (1 - \lambda) p(w), \tag{5.20}$$

where

$$\lambda = \frac{c(u)}{c(u) + \alpha}. \tag{5.21}$$

The second way of writing it shows that we can think of this as linearly interpolating between the MLE for the bigram model and the unigram model (Jelinek and Mercer, 1980).

Witten-Bell smoothing (Witten and Bell, 1991) – falsely so-called, because it was invented by Moffat – sets $\alpha = n_{1+}(u\bullet)$, but larger values (say, a few times bigger) can work better in practice.

5.5.2 Leaving one out

Katz smoothing (Katz, 1987) uses the Good-Turing estimates for seen bigrams, and backs off to the unigram model for unseen bigrams. More precisely, for bigrams:

$$p(w \mid u) = \begin{cases} \frac{(c(uw)+1) n_{c(uw)+1}}{c(u) n_{c(uw)}} & \text{if } c(uw) \geq 1 \\ \alpha(u) p(w) & \text{otherwise,} \end{cases} \tag{5.22}$$

where $\alpha(u)$ is chosen to make the conditional distribution $p(\cdot \mid u)$ sum to one.

Kneser-Ney smoothing (Ney, Essen, and Kneser, 1995) uses the absolute discounting formula () we showed above. Furthermore, it estimates the unigram model in a different way than Katz smoothing does. 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."

So, every time we see a new bigram uw, we count it only $(1-d)$ times; the subtracted d count is given to the unigram model, so that the unigram estimates are:

$$p(w) = \frac{n_{1+}(\bullet w) d}{\sum_{w'} n_{1+}(\bullet w') d} = \frac{n_{1+}(\bullet w)}{\sum_{w'} n_{1+}(\bullet w')}. \tag{5.23}$$

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

Chen and Goodman (1998) modify this so that the form (5.5.2) is enforced only for (typically) $r \geq 3$. The p_r for $r < 3$ are proportional to the Good-Turing estimates; for $r \geq 3$, another approximation is used for d. Modified Kneser-Ney is widely considered to be the best-performing smoothing method, and for m-gram languages models is used almost universally. (But the other smoothing methods are still common in other settings.)

5.5.3 Recursive smoothing

The above smoothing methods combine a bigram model with a unigram model. Note that the unigram model can itself be smoothed. If we want to smooth a trigram model, we can combine it with a bigram model, which can itself can be smoothed by combining with a unigram model, which can itself be smoothed. In general, we can smooth an m-gram model by combining it with a $(m-1)$-gram model, and so on.

5.6 Optional: Derivations of smoothing methods

5.6.1 Good-Turing smoothing

We want to maximize

$$L = \sum_{r \geq 1} r\, n_r \log p_{r-1} \tag{5.24}$$

subject to

$$\sum_{r \geq 0} n_r\, p_r = 1. \tag{5.25}$$

Form the Lagrangian:

$$\mathscr{L} = \sum_{r \geq 1} r\, n_r \log p_{r-1} - \lambda \left(\sum_{r \geq 0} n_r\, p_r - 1 \right) \tag{5.26}$$

Set partial derivatives to zero:

$$\frac{\partial \mathscr{L}}{\partial p_r} = \frac{(r+1)\, n_{r+1}}{p_r} - \lambda n_r = 0 \tag{5.27}$$

$$p_r = \frac{(r+1)\, n_{r+1}}{\lambda n_r} \tag{5.28}$$

Substitute into (5.25):

$$\sum_{r \geq 0} n_r\, p_r = \frac{N}{\lambda} = 1 \tag{5.29}$$

$$\lambda = N \tag{5.30}$$

Thus

$$p_r = \frac{(r+1)\, n_{r+1}}{N n_r}. \tag{5.31}$$

5.6.2 Kneser-Ney smoothing

We want to maximize

$$L = \sum_{r \geq 1} r\, n_r \log p_{r-1} \tag{5.32}$$

where

$$p_r = \begin{cases} \frac{r-d}{N} & r \geq 1 \\ \frac{n_{1+}d}{N} & r = 0. \end{cases} \tag{5.33}$$

Take the derivative:

$$\frac{\partial L}{\partial d} = \frac{n_1}{d} - \sum_{r \geq 2} \frac{r\, n_r}{r-1-d} \tag{5.34}$$

$$\leq \frac{n_1}{d} - \frac{2n_2}{1-d}. \tag{5.35}$$

By setting the derivative to zero, we get the upper bound:

$$d \leq \frac{n_1}{n_1 + 2n_2}. \tag{5.36}$$

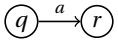
5.7 Finite-State Automata

But 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.

5.7.1 Review

A *finite state automaton (FSA)* 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 an FSA 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 FSAs later.

Here's a more formal definition.

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

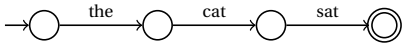
- Q is a finite set of *states*

- 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.

Question 10. Write an algorithm to decide whether a given FSA accepts the empty language or a nonempty language.

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



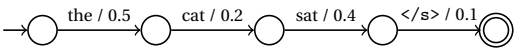
5.7.2 Weighted FSAs

A *weighted FSA* 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 FSA is the product of the weights of the transitions along the path, times the stop weight of the final state. A weighted FSA 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 FSA*, each state has the property that the weights of all of the exiting transitions and the stop weight sum to one. Then the weighted FSA also defines a probability distribution over strings.

Question 11. 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>`:

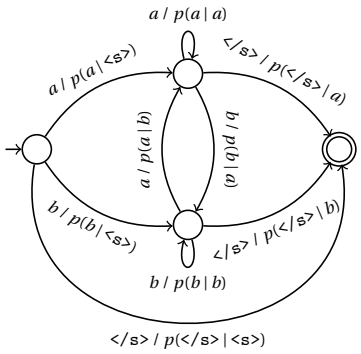


So, an n-gram language model is a probabilistic FSA with a very simple structure. If we continue to assume a bigram language model, we need a state for every possible context, that is, one for `<s>`, which we call $q_{\texttt{<s>}}$ and one for each word type a, which we call q_a. Every state is a final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



(5.37)

5.7.3 Training

If we are given a collection of strings $w^1, \ldots, w^N$ and a deterministic FSA 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 FSA that maximizes the likelihood of the training data $w^1, \ldots w^N$.

The smoothing methods discussed above extend to arbitrary weighted FSAs too. All that we need is a way to group states together. If q is a state, let $\bar{q}$ be the state group that q belongs to. Then, probability $p(a, q' \mid q)$ is smoothed by combining it with $p(a, q' \mid \bar{q})$.

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.

Good, I. J. (1953). "The Population Frequencies of Species and the Estimation of Population Parameters". In: *Biometrika* 40.3–4, pp. 237–264.

Jelinek, F. and R. L. Mercer (1980). "Interpolated Estimation of Markov source parameters from sparse data". In: *Proc. Workshop on Pattern Recognition in Practice*, pp. 381–397.

Katz, Slava M. (1987). "Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer". In: *IEEE Trans. Acoustics, Speech, and Signal Processing* 35.3, pp. 400–401.

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.