Summary of the mathematics and formulas for language models

The *maximum likelihood estimation* (MLE) refers to a model that gives a higher probability to the training data than any other model does. This can be done using a model based on relative frequency. That is, if the count for a word w is $C(w)$, then the probability that MLE (or RF) gives to the word is

$$P_{MLE}(w) = \frac{C(w)}{N}$$

. . . where N is the number of tokens in the training data.

The most commonly used intrinsic evaluation of a language model is *perplexity*: we take a test set and compute the total probability that the language model assigns to that test set, take the reciprocal, and normalize by the size of the test set. You can think of this "normalization" as taking the geometric mean of (the reciprocals of) the probabilities assigned to the individual words. If the test set has K tokens, then

$$\begin{aligned} perplexity &= \left(\prod_{i=1}^{K} P(w_i \mid h)\right)^{\frac{-1}{K}} \\ &= \sqrt[K]{\frac{1}{\prod_{i=1}^{K} P(w_i \mid h)}} \end{aligned}$$

Now, we shouldn't compute perplexity directly because multiplying all those probability values will cause underflow—the product keeps getting smaller the more things we multiply. Instead we should convert this to logarithms. Summing logs (and then computing the exponentiation of the result) is the same as multiplying original values. Using base B, we can compute perplexity as

$$\left(B^{\sum_{i=1}^{K} \log_B P(w_i \mid h)}\right)^{\frac{-1}{K}}$$

The information theory topics we'll be studying in the next couple of weeks should increase your intuition about perplexity, but for now you can think of perplexity as a measure of how surprised the model is by the test data. Low perplexity is good; it means the model did well predicting the test data. If the model assigned zero probability to any word, then perplexity will be infinite, which is bad. This, of course, will happen in a maximum likelihood estimation for any word never seen in the training data.

The simplest way to avoid the zero probability that MLE gives to unseen words Laplace ("Add-One") smoothing: simply add one to each word count, and increase the divisor appropriately.

$$P_{Laplace}(w) = \frac{C(w) + 1}{N + V}$$

Laplace smoothing tends not to get good results. After all, 1 is a completely arbitrary value to increase all the counts by and, as we have seen in class, adding one is inconsistent with Zipf's law. Instead, the Good-Turing estimation provides a better way, not only of avoiding the (infinite) underestimation for unseen words but avoiding the overestimation for seen words. The Good-Turing estimation considers the probability of seeing a word of a certain frequency class. If r is a word count (for example, $C(w) = r$ for some type w), the n_r is the *count of counts*, the number of types with that count. The probability of seeing a word from that class generally is $\frac{(r+1)n_{r+1}}{N}$. The probability of seeing a specific word w of that class is

$$P_{GT}(w) = \frac{(r+1)n_{r+1}}{Nn_r}$$

But Good-Turing, in this basic form, won't work at all for high-frequency words, since n_{r+1} could be zero. We have a few options for making Good-Turing practical. The easiest way is to use Good-Turing on unseen words, use MLE for common words (say, seen more than 5 times, or, more generally, k times), and for the rare words use Good-Turing but scaled so that all the probability mass "donated" to the unseen words comes from the rare words. How to do that scaling is pretty difficult; fortunately we can look the formula up. This is what we're calling Katz's k-cut-off.

$$P_{GT-Katz}(w) = \begin{cases} \frac{n_1}{Nn_0} & \text{if } C(w) = 0 \\ \frac{(r+1)\frac{n_{r+1}}{n_r} - r\frac{(k+1)\cdot n_{k+1}}{n_1}}{N(1-\frac{(k+1)\cdot n_{k+1}}{n_1}} & \text{if } 1 \leq r = C(w) \leq k \\ \frac{C(w)}{N} & \text{otherwise} \end{cases}$$

The other option for making Good-Turing practical is to smooth the frequencies. We know by Zipf's law what the graph of counts (r) vs counts of counts (n_r) will look like. Moreover, graphed on a log-log scale (so, $\log r$ vs $\log n_r$) it will be a straight line. . . -ish. We can use *regression analysis* to fit a line.

In general, regression analysis works this way. If you have k points with x-coordinates $x_1, x_2, \ldots x_k$ and y-coordinates $y_1, y_2, \ldots y_k$, then compute the "average" x and y:

$$\begin{aligned} \bar{x} &= \frac{\sum x_i}{k} \\ \bar{y} &= \frac{\sum y_i}{k} \end{aligned}$$

Then the slope of the line can be found by

$$m = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$$

. . . and the y-intercept by

$$b = \bar{y} - m\bar{x}$$

Thus we have the line

$$y = mx + b$$

This line relates r and n_r. Well, actually it relates $\log r$ and $\log n_r$, so if we want an estimate for n_r given r, we need

$$n_r \approx B^{m \log_B r + b}$$

Again, B is (in theory) any base (don't confuse it with b). However, the choice of base might affect whether values underflow or overflow. I had more success with a larger base (say 40). Putting this together, we have (letting $r = C(w)$),

$$P_{SGT}(w) = \frac{(r+1)B^{m \log_B (r+1) + b}}{NB^{m \log_B r + b}}$$

Also, since it is only for high frequency words that regular Good-Turing is bad for, we could use this regression-analysis version only for high frequencies, reverting to normal Good-Turing for low frequency words.

Finally, we can interpolate k language models:

$$P_{LI}(w) = \sum_{j=1}^{k} \lambda_j P_j(w)$$

The λs must sum to 1. The trick is finding the best λs. We can do this by maximizing the model's performance on some *held-out set*, ie data similar to the training set but reserved for tuning the model. If the held-out set has M tokens, then

- Initialize the λs, for example, to $\lambda_j = \frac{1}{k}$.
- Repeat
  - Compute z_{ij} for each token w_i and model P_j, which measures how much contribution P_j makes towards the overall model's probability for token w_i.

$$z_{ij} = \frac{\lambda_j P_j(w_i \mid h_i)}{\sum_{jj=1}^{k} \lambda_{jj} P_{jj}(w_i \mid h_i)}$$

  - Compute new λs using the average of these z_{ij}s

$$\lambda_j = \frac{1}{M}\sum_{i=1}^{M} z_{ij}$$

- Until convergence, that is, until two successive sets of λs show similar performance.

$$\frac{\ell(\lambda^{new}) - \ell(\lambda^{old})}{|\ell(\lambda^{new})|} \leq \varepsilon$$

  where

$$\ell(\lambda) = \frac{1}{M}\sum_{i=1}^{M} \log \sum_{j=1}^{k} \lambda_j P_j(w_i \mid h_i)$$

That last formula is called the *average log likelihood*, and it is similar to perplexity. The algorithm sketched above is a simplified (or "degenerate") form of the more general EM (expectation-maximization) algorithm for tuning the parameters to a language model.

Throughout this document we have assumed unigram-based models (except where we've used $P(w \mid h)$, where we imply that a history of some size may be taken into account). But any of these can be used with models of higher N-grams.