Stolcke pruning uses a loss function based on relative entropy. Formally, let *P* denote the trigram probabilities assigned by the original unpruned model, and let *P'* denote the probabilities in the pruned model. Then the relative entropy *D*(*P*||*P'*) between the two models is

$$-\sum_{w,h} P(w,h)[\log P'(w|h) - \log P(w,h)]$$

where *h* is the history. For trigrams, the history is the previous two words. Stolcke showed that this reduces to

$$-P(h)\{P(w|h) \\ [\log P(w|h') + \log \alpha'(h) - \log P(w|h)] \\ +[\log \alpha'(h) - \log \alpha(h)] \sum_{w:C(h,w)>0} P(w|h)\}$$

where $\alpha'(h)$ is the revised backoff weight after pruning and *h'* is the revised history after dropping the first word. The summation is over all the trigrams that were seen in training: $C(h,w) > 0$.

Stolcke pruning will remove *n*-grams as necessary, minimizing this loss.

3.1 Compression on Top of Pruning

After Stolcke pruning, we apply additional compression (either ZipTBO or HashTBO). ZipTBO uses a fairly straightforward data structure, which introduces relatively few additional losses on top of the pruned TBO model. A few small losses are introduced by quantizing the log likelihoods and the backoff alphas, but those losses probably don't matter much. More serious losses are introduced by restricting the vocabulary size, *V*, to the 64k most-frequent words. It is convenient to use byte aligned pointers. The actual vocabulary of more than 300,000 words for English (and more for other languages) would require 19-bit pointers (or more) without pruning. Byte operations are faster than bit operations. There are other implementations of ZipTBO that make different tradeoffs, and allow for larger *V* without pruning losses.

HashTBO is more heroic. It uses a method inspired by McIlroy (1982) in the original Unix Spell Program, which squeezed a word list of *N*=32,000 words into a PDP-11 address space (64k bytes). That was just 2 bytes per word!

HashTBO uses similar methods to compress a couple million *n*-grams into half a dozen megabytes, or about 3 bytes per *n*-gram on average (including log likelihoods and alphas for backing off). ZipTBO is faster, but takes more space (about 4 bytes per *n*-gram on average, as opposed to 3 bytes per *n*-gram). Given a fixed memory budget, ZipTBO has to make up the difference with more aggressive Stolcke pruning. More pruning leads to larger losses, as we will see, for the spelling application.

Losses will be reported in terms of performance on the spelling task. It would be nice if losses could be reported in terms of cross entropy, but the values output by the compressed language models cannot be interpreted as probabilities due to quantization losses and other compression losses.

4 McIlroy's Spell Program

McIlroy's spell program started with a hash table. Normally, we store the clear text in the hash table, but he didn't have space for that, so he didn't. Hash collisions introduce losses.

McIlroy then sorted the hash codes and stored just the interarrivals of the hash codes instead of the hash codes themselves. If the hash codes, *h*, are distributed by a Poisson process, then the interarrivals, *t*, are exponentially distributed:

$$\Pr(t) = \lambda e^{-\lambda t},$$

where $\lambda = \frac{N}{P}$. Recall that the dictionary contains *N*=32,000 words. *P* is the one free parameter, the range of the hash function. McIlroy hashed words into a large integer mod *P*, where *P* is a large prime that trades off space and accuracy. Increasing *P* consumes more space, but also reduces losses (hash collisions).

McIlroy used a Golomb (1966) code to store the interarrivals. A Golomb code is an optimal Huffman code for an infinite alphabet of symbols with exponential probabilities.

The space requirement (in bits per lexical entry) is close to the entropy of the exponential.

$$H = -\int_{t=0}^{\infty} \Pr(t) \log_2 \Pr(t)\, dt$$

$$\widehat{H} = \left\lceil \frac{1}{\log_e 2} + \log_2 \frac{1}{\lambda} \right\rceil$$