Compressing Trigram Language Models With Golomb Coding

Ken Church
Microsoft
One Microsoft Way
Redmond, WA, USA

Ted Hart
Microsoft
One Microsoft Way
Redmond, WA, USA

Jianfeng Gao
Microsoft
One Microsoft Way
Redmond, WA, USA

{church,tedhar,jfgao}@microsoft.com

Abstract

Trigram language models are compressed using a Golomb coding method inspired by the original Unix spell program. Compression methods trade off space, time and accuracy (loss). The proposed HashTBO method optimizes space at the expense of time and accuracy. Trigram language models are normally considered memory hogs, but with HashTBO, it is possible to squeeze a trigram language model into a few megabytes or less. HashTBO made it possible to ship a trigram contextual speller in Microsoft Office 2007.

1 Introduction

This paper will describe two methods of compressing trigram language models: HashTBO and ZipTBO. ZipTBO is a baseline compression method that is commonly used in many applications such as the Microsoft IME (Input Method Editor) systems that convert Pinyin to Chinese and Kana to Japanese.

Trigram language models have been so successful that they are beginning to be rolled out to applications with millions and millions of users: speech recognition, handwriting recognition, spelling correction, IME, machine translation and more. The EMNLP community should be excited to see their technology having so much influence and visibility with so many people. Walter Mossberg of the Wall Street Journal called out the contextual speller (the blue squiggles) as one of the most notable features in Office 2007:

There are other nice additions. In Word, Outlook and PowerPoint, there is now contextual spell checking, which points to a wrong word, even if the spelling is in the dictionary. For example, if you type "their" instead of "they're," Office catches the mistake. It really works. [1]

The use of contextual language models in spelling correction has been discussed elsewhere: (Church and Gale, 1991), (Mays *et al*, 1991), (Kukich, 1992) and (Golding and Schabes, 1996). This paper will focus on how to deploy such methods to millions and millions of users. Depending on the particular application and requirements, we need to make different tradeoffs among:

1. Space (for compressed language model),
2. Runtime (for *n*-gram lookup), and
3. Accuracy (losses for *n*-gram estimates).

HashTBO optimizes space at the expense of the other two. We recommend HashTBO when space concerns dominate the other concerns; otherwise, use ZipTBO.

There are many applications where space is extremely tight, especially on cell phones. HashTBO was developed for contextual spelling in Microsoft Office 2007, where space was the key challenge. The contextual speller probably would not have shipped without HashTBO compression.

We normally think of trigram language models as memory hogs, but with HashTBO, a few megabytes are more than enough to do interesting things with trigrams. Of course, more memory is always better, but it is surprising how much can be done with so little.

For English, the Office contextual speller started with a predefined vocabulary of 311k word types and a corpus of 6 billion word tokens. (About a

[1] http://online.wsj.com/public/article/SB116786111022966326-T8UUTIl2b10DaW11usf4NasZTYI_20080103.html?mod=tff_main_tff_top

Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pp. 199–207, Prague, June 2007.

third of the words in the vocabulary do not appear in the corpus.) The vocabularies for other languages tend to be larger, and the corpora tend to be smaller. Initially, the trigram language model is very large. We prune out small counts (8 or less) to produce a starting point of 51 million trigrams, 14 million bigrams and 311k unigrams (for English). With extreme Stolcke, we cut the 51+14+0.3 million *n*-grams down to a couple million. Using a Golomb code, each *n*-gram consumes about 3 bytes on average.

With so much Stolcke pruning and lossy compression, there will be losses in precision and recall. Our evaluation finds, not surprisingly, that compression matters most when space is tight. Although HashTBO outperforms ZipTBO on the spelling task over a wide range of memory sizes, the difference in recall (at 80% precision) is most noticeable at the low end (under 10MBs), and least noticeable at the high end (over 100 MBs). When there is plenty of memory (100+ MBs), the difference vanishes, as both methods asymptote to the upper bound (the performance of an uncompressed trigram language model with unlimited memory).

2 Preliminaries

Both methods start with a TBO (trigrams with backoff) LM (language model) in the standard ARPA format. The ARPA format is used by many toolkits such as the CMU-Cambridge Statistical Language Modeling Toolkit.[2]

2.1 Katz Backoff

No matter how much data we have, we never have enough. Nothing has zero probability. We will see *n*-grams in the test set that did not appear in the training set. To deal with this reality, Katz (1987) proposed backing off from trigrams to bigrams (and from bigrams to unigrams) when we don't have enough training data.

Backoff doesn't have to do much for trigrams that were observed during training. In that case, the backoff estimate of $P(w_i|w_{i-2}w_{i-1})$ is simply a discounted probability $P_d(w_i|w_{i-2}w_{i-1})$.

The discounted probabilities steal from the rich and give to the poor. They take some probability mass from the rich *n*-grams that have been seen in training and give it to poor unseen *n*-grams that might appear in test. There are many ways to discount probabilities. Katz used Good-Turing smoothing, but other smoothing methods such as Kneser-Ney are more popular today.

Backoff is more interesting for unseen trigrams. In that case, the backoff estimate is:

$$\alpha(w_{i-2}w_{i-1})P_d(w_i|w_{i-1})$$

The backoff alphas (α) are a normalization factor that accounts for the discounted mass. That is,

$$\alpha(w_{i-2}w_{i-1}) = \frac{1-\sum_{w_i:C(w_{i-2}w_{i-1}w_i)} P(w_i|w_{i-2}w_{i-1})}{1-\sum_{w_i:C(w_{i-2}w_{i-1}w_i)} P(w_i|w_{i-1})}$$

where $C(w_{i-2}w_{i-1}w_i) > 0$ simply says that the trigram was seen in training data.

3 Stolcke Pruning

Both ZipTBO and HashTBO start with Stolcke pruning (1998).[3] We will refer to the trigram language model after backoff and pruning as a *pruned TBO LM*.

Stolcke pruning looks for *n*-grams that would receive nearly the same estimates via Katz backoff if they were removed. In a practical system, there will never be enough memory to explicitly materialize all *n*-grams that we encounter during training. In this work, we need to compress a large set of *n*-grams (that appear in a large corpus of 6 billion words) down to a relatively small language model of just a couple of megabytes. We prune as much as necessary to make the model fit into the memory allocation (after subsequent HashTBO/ZipTBO compression).

Pruning saves space by removing *n*-grams subject to a loss consideration:

1. Select a threshold θ.
2. Compute the performance loss due to pruning each trigram and bigram individually using the pruning criterion.
3. Remove all trigrams with performance loss less than θ
4. Remove all bigrams with no child nodes (trigram nodes) and with performance loss less than θ
5. Re-compute backoff weights.

[2] http://www.speech.cs.cmu.edu/SLM

[3] http://www.nist.gov/speech/publications/darpa98/html/lm20/lm20.htm

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.

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

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

The ceiling operator $\lceil\,\rceil$ is introduced because Huffman codes use an integer number of bits to encode each symbol.

We could get rid of the ceiling operation if we replaced the Huffman code with an Arithmetic code, but it is probably not worth the effort.

Lookup time is relatively slow. Technically, lookup time is $O(N)$, because one has to start at the beginning and add up the interarrivals to reconstruct the hash codes. McIlroy actually introduced a small table on the side with hash codes and offsets so one could seek to these offsets and avoid starting at the beginning every time. Even so, our experiments will show that HashTBO is an order of magnitude slower than ZipTBO.

Accuracy is also an issue. Fortunately, we don't have a problem with dropouts. If a word is in the dictionary, we aren't going to misplace it. But two words in the dictionary could hash to the same value. In addition, a word that is not in the dictionary could hash to the same value as a word that is in the dictionary. For McIlroy's application (detecting spelling errors), the only concern is the last possibility. McIlroy did what he could to mitigate false positive errors by increasing P as much as he could, subject to the memory constraint (the PDP-11 address space of 64k bytes).

We recommend these heroics when space dominates other concerns (time and accuracy).

5 Golomb Coding

Golomb coding takes advantage of the sparseness in the interarrivals between hash codes. Let's start with a simple recipe. Let t be an interarrival. We will decompose t into a pair of a quotient (t_q) and a remainder (t_r). That is, let $t = t_q m + t_r$ where $t_q = \lfloor t/m \rfloor$ and $t_r = t \bmod m$. We choose m to be a power of two near $m \approx \left\lceil \frac{E[t]}{2} \right\rceil = \left\lceil \frac{P}{2N} \right\rceil$, where $E[t]$ is the expected value of the interarrivals, defined below. Store t_q in unary and t_r in binary.

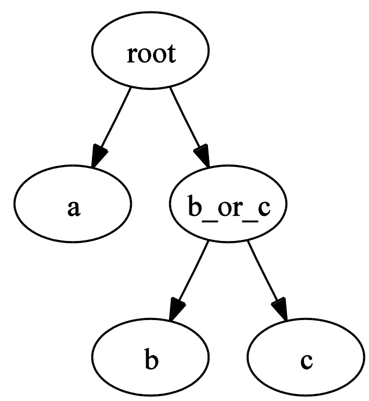
Binary codes are standard, but unary is not. To encode a number z in unary, simply write out a sequence of z-1 zeros followed by a 1. Thus, it takes z bits to encode the number z in unary, as opposed to $\log_2 z$ bits in binary.

This recipe consumes $t_q + \log_2 m$ bits. The first term is for the unary piece and the second term is for the binary piece.

Why does this recipe make sense? As mentioned above, a Golomb code is a Huffman code for an infinite alphabet with exponential probabilities. We illustrate Huffman codes for infinite alphabets by starting with a simple example of a small (very finite) alphabet with just three symbols: $\{a,\ b,\ c\}$. Assume that half of the time, we see a, and the rest of the time we see b or c, with equal probabilities:

Symbol	Code	Length	Pr
A	0	1	50%
B	10	2	25%
C	11	2	25%

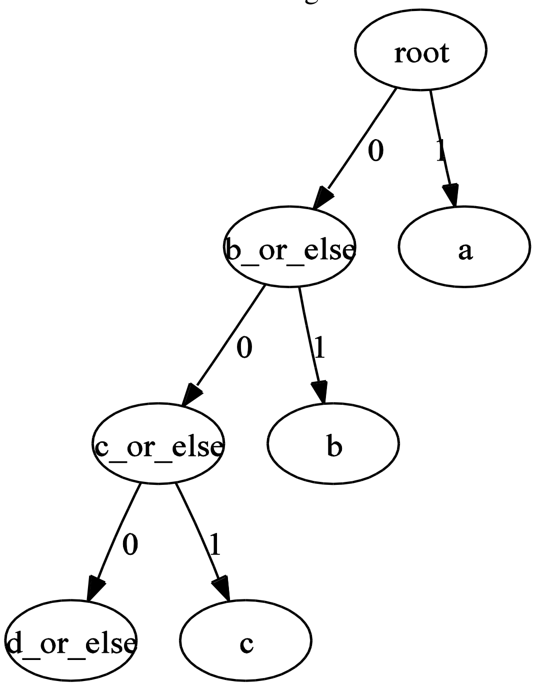
The Huffman code in the table above can be read off the binary tree below. We write out a 0 whenever we take a left branch and a 1 whenever we take a right branch. The Huffman tree is constructed so that the two branches are equally likely (or at least as close as possible to equally likely).



Now, let's consider an infinite alphabet where $\Pr(a) = \frac{1}{2}$, $\Pr(b) = \frac{1}{4}$ and the probability of the t+1^{st} symbol is $\Pr(t) = (1-\beta)\beta^t$ where $\beta = \frac{1}{2}$. In this case, we have the following code, which is simply t in unary. That is, we write out $t-1$ zeros followed by a 1.

Symbol	Code	Length	Pr
A	1	1	2^{-1}
B	01	2	2^{-2}
C	001	3	2^{-3}

The Huffman code reduces to unary when the Huffman tree is left branching:



In general, β need not be ½. Without loss of generality, assume $\Pr(t) = (1-\beta)\beta^t$ where $\frac{1}{2} \le \beta < 1$ and $t \ge 0$. β depends on $E[t]$, the expected value of the interarrivals:

$$E[t] = \frac{P}{N} = \frac{\beta}{1-\beta} \Rightarrow \beta = \frac{E[t]}{1+E[t]}$$

Recall that the recipe above calls for expressing t as $m \cdot t_q + t_r$ where $t_q = \lfloor \frac{t}{m} \rfloor$ and $t_r = t \bmod m$. We encode t_q in unary and t_r in binary. (The binary piece consumes $\log_2 m$ bits, since t_r ranges from 0 to m.)

How do we pick m? For convenience, let m be a power of 2. The unary encoding makes sense as a Huffman code if $\beta^m \approx \frac{1}{2}$.

Thus, a reasonable choice[4] is $m \approx \left\lceil \frac{E[t]}{2} \right\rceil$. If $\beta = \frac{E[t]}{1+E[t]}$, then $\beta^m = \frac{E[t]^m}{(1+E[t])^m} \approx 1 - \frac{m}{E[t]}$. Setting $\beta^m \approx \frac{1}{2}$, means $m \approx \frac{E[t]}{2}$.

6 HashTBO Format

The HashTBO format is basically the same as McIlroy's format, except that McIlroy was storing words and we are storing n-grams. One could store all of the n-grams in a single table, though we actually store unigrams in a separate table. An n-gram is represented as a key of n integers (offsets into the vocabulary) and two values, a log likelihood and, if appropriate, an alpha for backing off. We'll address the keys first.

6.1 HashTBO Keys

Trigrams consist of three integers (offsets into the Vocabulary): $w_1 w_2 w_3$. These three integers are mapped into a single hash between 0 and $P-1$ in the obvious way:

$$hash = (w_3 V^0 + w_2 V^1 + w_1 V^2) \bmod P$$

where V is vocabulary size. Bigrams are hashed the same way, except that the vocabulary is padded with an extra symbol for NA (not applicable). In the bigram case, w_3 is NA.

We then follow a simple recipe for bigrams and trigrams:

1. Stolcke prune appropriately
2. Let N be the number of n-grams
3. Choose an appropriate P (hash range)
4. Hash the N n-grams
5. Sort the hash codes
6. Take the first differences (which are modeled as interarrivals of a Poisson process)
7. Golomb code the first differences

We did not use this method for unigrams, since we assumed (perhaps incorrectly) that we will have explicit likelihoods for most of them and therefore there is little opportunity to take advantage of sparseness.

Most of the recipe can be fully automated with a turnkey process, but two steps require appropriate hand intervention to meet the memory allocation for a particular application:

1. Stolcke prune appropriately, and
2. Choose an appropriate P

[4] This discussion follows slide 29 of http://www.stanford.edu/class/ee398a/handouts/lectures/01-EntropyLosslessCoding.pdf. See (Witten *et al*, 1999) and http://en.wikipedia.org/wiki/Golomb_coding, for similar discussion, though with slightly different notation. The primary reference is (Golomb, 1966).

Ideally, we'd like to do as little pruning as possible and we'd like to use as large a P as possible, subject to the memory allocation. We don't have a principled argument for how to balance Stolcke pruning losses with hashing losses; this can be arrived at empirically on an application-specific basis. For example, to fix the storage per n-gram at around 13 bits:

$$13 = \left\lceil \frac{1}{\log_e 2} + \log_2 \frac{1}{\lambda} \right\rceil$$

If we solve for λ, we obtain $\lambda \approx 1/20{,}0000$. In other words, set P to a prime near $20{,}000N$ and then do as much Stolcke pruning as necessary to meet the memory constraint. Then measure your application's accuracy, and adjust accordingly.

6.2 HashTBO Values and Alphas

There are N log likelihood values, one for each key. These N values are quantized into a small number of distinct bins. They are written out as a sequence of N Huffman codes. If there are Katz backoff alphas, then they are also written out as a sequence of N Huffman codes. (Unigrams and bigrams have alphas, but trigrams don't.)

6.3 HashTBO Lookup

The lookup process is given an n-gram, $w_{i-2}w_{i-1}w_i$, and is asked to estimate a log likelihood, $\log\Pr(w_i \mid w_{i-2}\, w_{i-1})$. Using the standard backoff model, this depends on the likelihoods for the unigrams, bigrams and trigrams, as well as the alphas.

The lookup routine not only determines if the n-gram is in the table, but also determines the offset within that table. Using that offset, we can find the appropriate log likelihood and alpha. Side tables are maintained to speed up random access.

7 ZipTBO Format

ZipTBO is a well-established representation of trigrams. Detailed descriptions can be found in (Clarkson and Rosenfeld 1997; Whittaker and Raj 2001).

ZipTBO consumes 8 bytes per unigram, 5 bytes per bigram and 2.5 bytes per trigram. In practice, this comes to about 4 bytes per n-gram on average.

Note that there are some important interactions between ZipTBO and Stolcke pruning. ZipTBO is relatively efficient for trigrams, compared to bigrams. Unfortunately, aggressive Stolcke pruning generates bigram-heavy models, which don't compress well with ZipTBO.

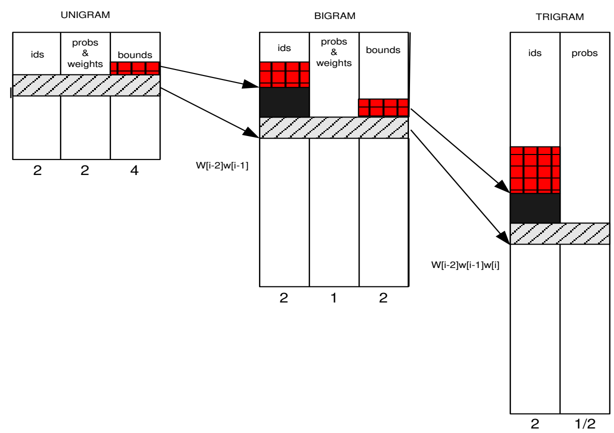


Figure 1. Tree structure of n-grams in ZipTBO format, following Whittaker and Ray (2001)

7.1 ZipTBO Keys

The tree structure of the trigram model is implemented using three arrays. As shown in Figure 1, from left to right, the first array (called *unigram array*) stores unigram nodes, each of which branches out into bigram nodes in the second array (*bigram array*). Each bigram node then branches out into trigram nodes in the third array (*trigram array*).

The length of the unigram array is determined by the vocabulary size (V). The lengths of the other two arrays depend on the number of bigrams and the number of trigrams, which depends on how aggressively they were pruned. (We do not prune unigrams.)

We store a 2-byte word id for each unigram, bigram and trigram.

The unigram nodes point to blocks of bigram nodes, and the bigram nodes point to blocks of trigram nodes. There are boundary symbols between blocks (denoted by the pointers in Figure 1). The boundary symbols consume 4 bytes for each unigram and 2 bytes for each bigram.

In each block, nodes are sorted by their word ids. Blocks are consecutive, so the boundary value of an $n-1$-gram node together with the boundary value of its previous $n-1$-gram node specifies, in the n-gram array, the location of the block containing all its child nodes. To locate a particular child node, a binary search of word ids is performed within the block.

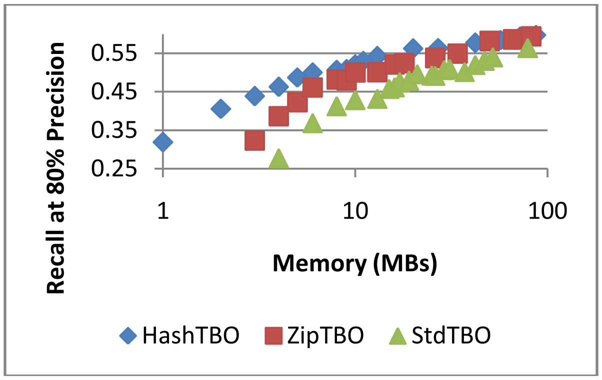


Figure 2. When there is plenty of memory, performance (recall @ 80% precision) asymptotes to the performance of baseline system with no compression (StdTBO). When memory is tight, HashTBO >> ZipTBO >> StdTBO.

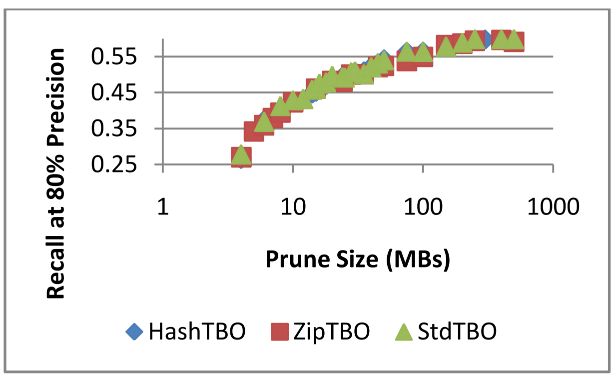


Figure 3. The differences between the methods in Figure 2 vanish if we adjust for prune size.

7.2 ZipTBO Values and Alphas

Like HashTBO, the log likelihood values and backoff alphas are quantized to a small number of quantization levels (256 levels for unigrams and 16 levels for bigrams and trigrams). Unigrams use a full byte for the log likelihoods, plus another full byte for the alphas. Bigrams use a half byte for the log likelihood, plus another half byte for the alphas. Trigrams use a half byte for the log likelihood. (There are no alphas for trigrams.)

7.3 ZipTBO Bottom Line

1. 8 bytes for each unigram:
   a. 2 bytes for a word id +
   b. 4 bytes for two boundary symbols +
   c. 1 byte for a log likelihood +
   d. 1 byte for an alpha
2. 5 bytes for each bigram:
   a. 2 bytes for a word id +
   b. 2 bytes for a boundary symbol +
   c. ½ bytes for a log likelihood +
   d. ½ bytes for an alpha
3. 2.5 bytes for each trigram:
   a. 2 bytes for a word id +
   b. ½ bytes for a log likelihood

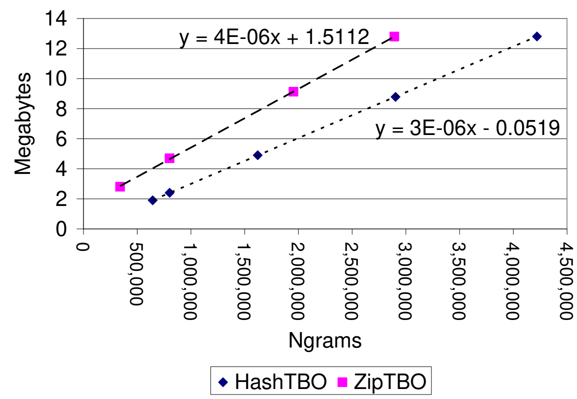


Figure 4. On average, HashTBO consumes about 3 bytes per n-gram, whereas ZipTBO consumes 4.

8 Evaluation

We normally think of trigram language models as memory hogs, but Figure 2 shows that trigrams can be squeezed down to a megabyte in a pinch. Of course, more memory is always better, but it is surprising how much can be done (27% recall at 80% precision) with so little memory.

Given a fixed memory budget, HashTBO outperforms ZipTBO which outperforms StdTBO, a baseline system with no compression. Compression matters more when memory is tight. The gap between methods is more noticeable at the low end (under 10 megabytes) and less noticeable at the high end (over 100 megabytes), where both methods asymptote to the performance of the StdTBO baseline.

All methods start with Stolcke pruning. Figure 3 shows that the losses are largely due to pruning.

All three methods perform about equally well, assuming the same amount of pruning.

The difference is that HashTBO can store more n-grams in the same memory and therefore it doesn't have to do as much pruning. Figure 4 shows that HashTBO consumes 3 bytes per n-gram whereas ZipTBO consumes 4.

Figure 4 combines unigrams, bigrams and trigrams into a single n-gram variable. Figure 5 drills down into this variable, distinguishing bigrams from trigrams. The axes here have been reversed so we can see that HashTBO can store more of both kinds in less space. Note that both HashTBO lines are above both ZipTBO lines.

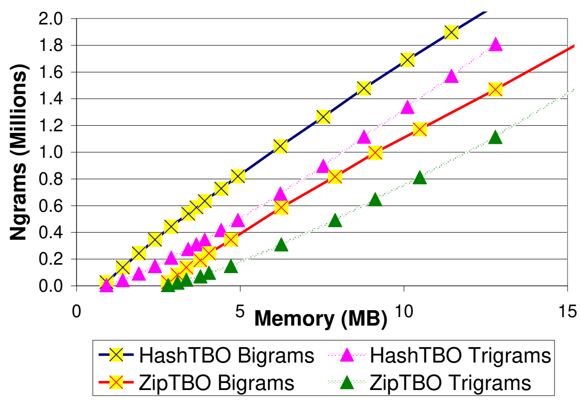


Figure 5. HashTBO stores more bigrams and trigrams than ZipTBO in less space.

In addition, note that both bigram lines are above both trigram lines (triangles). Aggressively pruned models have more bigrams than trigrams!

Linear regression on this data shows that HashTBO is no better than ZipTBO on trigrams (with the particular settings that we used), but there is a big difference on bigrams. The regressions below model M (memory in bytes) as a function of bi and tri, the number of bigrams and trigrams, respectively. (Unigrams are modeled as part of the intercept since all models have the same number of unigrams.)

$$M_{HashTBO} = 0.8 + 3.4bi + 2.6tri$$
$$M_{ZipTBO} = 2.6 + 4.9bi + 2.6tri$$

As a sanity check, it is reassuring that ZipTBO's coefficients of 4.9 and 2.6 are close to the true values of 5 bytes per bigram and 2.5 bytes per trigram, as reported in Section 7.3.

According to the regression, HashTBO is no better than ZipTBO for trigrams. Both models use roughly 2.6 bytes per trigram. When trigram models have relatively few trigrams, the other coefficients matter. HashTBO uses less space for bigrams (3.4 bytes/bigram << 4.9 bytes/bigram) and it has a better intercept (0.8 << 2.6).

We recommend HashTBO if space is so tight that it dominates other concerns. However, if there is plenty of space, or time is an issue, then the tradeoffs work out differently. Figure 6 shows that ZipTBO is an order of magnitude faster than HashTBO. The times are reported in microseconds per n-gram lookup on a dual Xeon PC with a 3.6 ghz clock and plenty of RAM (4GB). These times were averaged over a test set of 4 million lookups. The test process uses a cache. Turning off the cache increases the difference in lookup times.

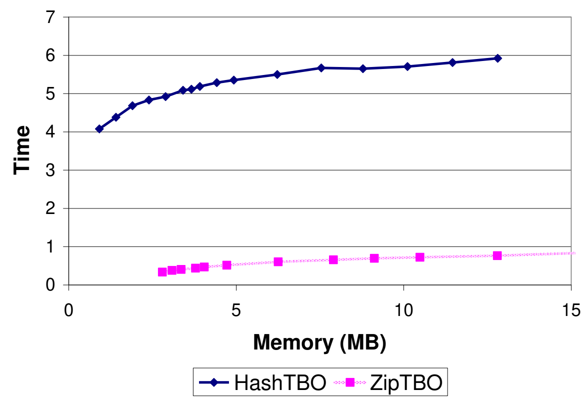


Figure 6. HashTBO is slower than ZipTBO.

9 Conclusion

Trigram language models were compressed using HashTBO, a Golomb coding method inspired by McIlroy's original spell program for Unix. McIlroy used the method to compress a dictionary of 32,000 words into a PDP-11 address space of 64k bytes. That is just 2 bytes per word!

We started with a large corpus of 6 billion words of English. With HashTBO, we could compress the trigram language model into just a couple of megabytes using about 3 bytes per n-gram (compared to 4 bytes per n-gram for the ZipTBO baseline). The proposed HashTBO method is not fast, and it is not accurate (not lossless), but it is hard to beat if space is tight, which was the case for the contextual speller in Microsoft Office 2007.

Acknowledgments

We would like to thank Dong-Hui Zhang for his contributions to ZipTBO.

References

Ashok K. Chandra, Dexter C. Kozen, and Larry J.Stockmeyer. 1981 Alternation. *Journal of the Association for Computing Machinery*, 28(1):114-133.

Church, K., and Gale, W. 1991 Probability Scoring for Spelling Correction, *Statistics and Computing*.

Clarkson, P. and Robinson, T. 2001 Improved language modeling through better language model evaluation measures, *Computer Speech and Language*, 15:39-53, 2001.

Dan Gusfield. 1997 *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, UK

Gao, J. and Zhang, M., 2002 Improving language model size reduction using better pruning criteria. *ACL 2002*: 176-182.

Gao, J., Goodman, J., and Miao, J. 2001 The use of clustering techniques for language modeling – application to Asian languages. *Computational Linguistics and Chinese Language Processing*, 6:1, pp 27-60.

Golding, A. R. and Schabes, Y. 1996 Combining Trigram-based and feature-based methods for context-sensitive spelling correction, *ACL*, pp. 71-78.

Golomb, S.W. 1966 Run-length encodings *IEEE Transactions on Information Theory*, 12:3, pp. 399-40.

Goodman, J. and Gao, J. 2000 Language model size reduction by pruning and clustering, *ICSLP-2000, International Conference on Spoken Language Processing,* Beijing, October 16-20, 2000.

Mays, E., Damerau, F. J., and Mercer, R. L. 1991 Context based spelling correction. *Inf. Process. Manage.* 27, 5 (Sep. 1991), pp. 517-522.

Katz, Slava, 1987 Estimation of probabilities from sparse data for other language component of a speech recognizer. *IEEE transactions on Acoustics, Speech and Signal Processing*, 35:3, pp. 400-401.

Kukich, Karen, 1992 Techniques for automatically correcting words in text, *Computing Surveys*, 24:4, pp. 377-439.

M. D. McIlroy, 1982 Development of a spelling list, *IEEE Trans. on Communications* 30 pp. 91-99.

Seymore, K., and Rosenfeld, R. 1996 Scalable backoff language models. *Proc. ICSLP*, Vol. 1, pp.232-235.

Stolcke, A. 1998 Entropy-based Pruning of Backoff Language Models. Proc. DARPA News Transcription and Understanding Workshop, 1998, pp. 270--274, Lansdowne, VA.

Whittaker, E. and Ray, B. 2001 Quantization-based language model compression. *Proc. Eurospeech*, pp. 33-36.

Witten, I. H., Moffat, A., and Bell, T. C. 1999 *Managing Gigabytes (2nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc.