Noiseless Language Model Using Trigrams

Jinghao Yan

Abstract

Language models based on n-grams improve with more training data. However, increasing the size of the training set introduces new problems in storage and speed. We assess the various tradeoffs in designing a trigram language model, and propose a solution that provides a good balance of speed, size and accuracy. We discuss the combinations of design decisions and their impact on the three dimensions of performance, then demonstrate the efficacy of the data structure using a test set of 2000 sentences and the Kneser-Ney (KN) smoothing scheme.

1 Introduction

We implement a data structure for a trigram-based language model that is optimized for size, and demonstrate its performance on a test set of 2000 sentences using Kneser-Ney smoothing, and an extension involving Bloom filters that trades a small probability of false positives (involving zero-count trigrams) for a large amount of space. We give a rigorous analysis of the various data structure and algorithmic designs, including the likely efficacy of various noisy language models. We then conclude with data demonstrating the efficacy of the approach, and the benefits of the Bloom filter based approximation scheme. The concepts investigated here are applicable in general but are specially optimized for the test corpus with approximately 500,000 unique words, 8 million bigrams and 42 million trigrams.

2 Data Structure Design Decisions

The language model improves with more data. In order to support a large language model, the data structure must be both compact and fast. Our data structure addresses both needs through the following design decisions. We represent all of the token and type counts using (1) a single multi-level table that maps each word *w* to a `long[]` array containing all the information about the bi- and trigrams that begin with *w* (henceforth referred to as the *suffix table*), and (2) a set of auxiliary arrays that map each word *w* to other features of the training data like $c(w), t(\cdot,w,\cdot), t(\cdot,w), t(w,\cdot)$, which are respectively (a) the number of times word *w* occurs, (b) the number of unique trigrams that have *w* in the middle, (c) the number of unique bigrams that end with *w*, and (d) the number of unique bigrams that begin with *w*. In general, t represents the unique counts of the parameters and c represents the total counts of the parameters, with $\cdot$ being the wildcard.

The multi-level table (**Figure 1**) is flexible enough to handle queries about bigram token counts and type counts using special "words" like SUFFIX and * so that $(w_1, w_2, SUFFIX)$ and $(w_1, w_2, *)$ represent $t(w_1, w_2, \cdot)$ and $c(w_1, w_2, \cdot)$. Therefore, all queries and counts involving two or more words can be considered queries with three words, eliminating the need for maintaining different data structures. Henceforth, *trigrams* will refer to our more flexible definition of trigrams.

2.1 Multi-Level Table

At the top level is an array of pointers to `long` arrays (**Figure 1**), where the pointer at position w_1 points to the array of all bigrams that follow word w_1 (All words are referred by their integer value given by the indexer). Each element in the suffix array packs all necessary information about that trigram, including w_2, w_3, the words that follow w_1, and the counts of (w_1, w_2, w_3).

There are three key benefits to this multi-level approach. The first is that while the space of all trigrams is large, the space of all trigrams that begin with a specific word w_1 is relatively small. Therefore, this structure allows for faster searching using binary search (which is useful in the decoding stage) and sorting (the use of which will become clear soon).

The second benefit is that it saves space. In a naïve one-level implementation, the first word in a trigram is repeated for every trigram sharing that word. The multi-level approach represents it only once. Therefore, for the cost of 500,000 pointers (4-8 bytes each), we save 19 bits per trigram, for a total memory savings of 114-116 MB of RAM, assuming that otherwise the data is packed as much as possible. If all three words were represented together, it would require $19 * 3 = 57$ bits per trigram, leaving no space for the counts. Therefore, all of the counts required for each trigram would require more than a single long to represent in the naïve implementation, so the savings are much more significant (perhaps at the cost of another integer and extra memory accesses).

The third benefit is that smaller individual array sizes facilitate resizing. During the model-building stage,

each array is dynamically grown as needed to store all of the counts. One major problem the multi-level table avoids is the issue of running out of heap space. A single table/array containing all trigram data may require over half a gigabyte of heap space during model-building. Therefore, to double the size of the table, over 1 GB of heap space must be allocated. This level of memory use volatility renders any sparse data representations like hash tables (even open addressing hash tables) impossible. The multi-level data structure avoids this pitfall because each resize is relatively small—the average word has only about 100 unique bigrams that follow it, with the worst case being 7 million for the most common word, which is still significantly better than 50 million.

2.2 Right Structure for the Right Time

During model-building, the suffix tables are represented in an open-addressing hash table using linear probing in order to facilitate fast ($O(1)$) insertion and update. When the model is fully constructed, each suffix table is converted into a sorted array (with the same elements) in order to eliminate empty space. This saves memory. During the decoding stage, lookup uses binary search, which is relatively fast because most of the tables are very small, with the largest having only 7 million elements. In order to have a similar performance using the same hash table, the load factor will have to be approximately ½, wasting half of the space.

2.3 "Perfect" Hashing

In the model-building stage (and in the decoding stage due to caching) having a good hash function is critical for good performance. We tried various implementations: (1) bitshifting and combining (using *xor*s) the integer representations of words with very bad performance (the hash table access times increased *linearly* with the number of elements in the table); (2) caching and using the hash codes from the string representations of each word (`indexer.get(word).hashCode()`), for much better performance; and finally (3) randomly generating a hash code for each word and storing that, for very good performance.

The final implementation is nearly perfect in the sense that it distributes hash codes uniformly between $[1,2^{31})$, and any combination (xor) of it is also therefore uniform. Collisions are as low as theoretically possible for uniform insert/update/lookup operations, speeding up model-building and rehashing, and also making the cache much more effective. The uniformity of the hash function allows us to make rigorous probabilistic assessments of various randomized solutions (such as hash tables and Bloom filters) without the specificity of our training corpus.

2.4 Ordered Rehashing

The "perfect" hash function optimized for uniform distribution of inputs. However, some words (or combinations of words) clearly appear more frequently. In a hash table, average lookup/update would be significantly faster if, in the event of collisions (either due to direct collision or linear probing) the more frequently-accessed values are located before less frequently-accessed values. Therefore, in the rehash stage during model building, the rows are reinserted into the new table in decreasing order of frequency, thus making future update and lookup much faster.

2.5 Direct-Mapped Count Cache

The final data structure design is to build a direct-mapped cache for counts during decoding. This optimizes for repeat queries for the counts of the same combination of words because it allows the decoder to bypass having to do (relatively) expensive binary-search on a suffix table. We tested various cache table sizes and found that, due to the locality of the repeated requests, 10,000 offers nearly the same hit rate as 1 million for a small fraction of the cost (**Figure 2**), but 100,000 performs slightly better (234s compared to 244s during decoding) than 10,000 and 1 million. The cache saves over 50% time during decoding.

Another cache that was investigated was on the actual probability estimates. However, given that decoding time was below 6 minutes, and that would save only some minor recursion/if-else overhead over the existing count cache (because with high probability an entry that would appear in the probability cache would also have counts that appear in the count cache), it was not valuable enough to address relative to other, more important, features.

3 Applying Kneser-Ney Smoothing

The probabilities calculated for each given n-gram is smoothed using the KN smoothing technique with absolute discounting, using a constant discount value $d = 0.75$. In particular:

$$\Pr(w_3|w_2,w_1) = \frac{\max(0, c(w_1,w_2,w_3) - d) + d \cdot t(w_1,w_2,\cdot) \cdot \Pr(w_3|w_2)}{c(w_1,w_2)}$$

with a fallback to $\Pr(w_3|w_2,w_1) = \Pr(w_3|w_2)$ if the context w_1, w_2 was never seen before. $\Pr(w_3|w_2)$ and $\Pr(w_3)$ are defined similarly, with the base case in the recursion at $\Pr(w_3)$ being the uniform distribution, so $\Pr(w_3) = (\max(0, c(w_3) - d) + d \cdot t(\cdot) \cdot 1/t(\cdot))/c(\cdot) = (\max(0, c(w_3) - d) + d)/c(\cdot) = (\max(d, c(w_3))/c(\cdot)$ where $t(\cdot)$ is the number of unique words.

4 Noisy Models

We considered three ideas before deciding on a hybrid of Bloom filters and the exact model. The goal was to minimize the error rate such that (consistently) less than 1% of counts were wrong.

4.1 Hash and Hope: Balls and Bins Model

In this model, we randomly map trigrams into bins, and storing the associated counts. If multiple trigrams map into the same bin, that will likely have the wrong value for at least one of those trigrams. Therefore, we need extra allocated (but unused) space. The benefit of this approach is that lookup, insertion and update are extremely fast, but it comes at a significant reliability and storage cost.

Given our uniformly distributed hash function, this is equivalent to the balls and bins model with $balls = t(\cdot,\cdot,\cdot)$ (number of unique trigrams) and $bins = c \cdot balls$ (the number of array elements to allocate in total, including free space), for some constant c. Given our goal, we want to minimize c such that $\frac{\boldsymbol{E}[X_1+\cdots+X_{bins}]}{bins} = \boldsymbol{E}[X_i] = \Pr(X_i) \leq 0.01$ where X_i is a dummy variable for whether bin i has 2 or more trigrams mapped into it. Since there are lots of balls and bins, we can approximate this with the Poisson distribution. So $\Pr(X_i) \approx 1 - \Pr(Y_i = 0) - \Pr(Y_i = 1) = 1 - \left(e^{-\frac{1}{c}} + \left(\frac{1}{c}\right)e^{-\frac{1}{c}}\right)$ (where Y_i is the number of balls/trigrams mapped to bin i). Solving that probability, we need that c be at least 7, which is impractical for 50 million trigrams. Even if each trigram (plus counts) required only 4 bytes of memory, we would need $7 \cdot 4 \cdot 50 = 1400$ MB of memory, which is significantly more than the requirements of the exact model. In effect, it would trade away precision **and** space efficiency for a small performance, and this strategy is dominated by others we will now investigate. In addition, in practice the collision rates are somewhat higher than the theoretical estimate. In our tests, with a $c = 4$, we detected collision rates around 4%, which is higher than the 2.65% that the balls-and-bins model would predict.

4.2 Perfect Hash Functions

The randomized language model described by Talbot et al is promising because it can give arbitrarily good one-sided error rates. However, building the model is non-trivial and memory intensive. In particular, any efficient implementation would require us to keep track of all the (directed) edges from the LHS (n-grams) and RHS (locations) and vice versa. With k hash functions, that will require $2k \cdot t(\cdot,\cdot,\cdot)$ edges. In addition, the paper recommended that the actual array be of size at least $1.23t(\cdot,\cdot,\cdot)$, for a total of $(2k + 1.23)t(\cdot,\cdot,\cdot)$ entries. If each edge is compactly represented as an integer (assuming no overhead for objects/arrays and scratch space), that would be nearly 1.5GB, *in addition to* having the trigram data in-memory while building the perfect hash functions. Therefore, there's a high risk that this may not fit under the 2GB memory requirements.

4.3 Count Quantization and Bloom Filters

Another suggestion by Talbot et al is to quantize the counts and use Bloom filters, making the observation that language models exhibit Zipf's law: Nearly 2/3rds of trigrams appear only once (**Figure 3**). Although this may be a decent trade-off for improved speed or size (with cost measured in lost BLEU), it would fail our criterion regarding count correctness. In particular, quantizing counts will imply that a large fraction (significantly greater than 1%) of the counts will be different from the actual counts, even if the effect on decoding is minimal.

4.4 Better Measure Using BLEU

Clearly, measuring the number of "wrong" counts is not the best metric for assessing noisy models. For example, if counts in the range of 50,000 were "wrong" by 1 or 2%, the effect on decoding should be minimal. Therefore, a better metric would have been to measure just the effect on the BLEU score or the percentage deviation of individual counts.

4.5 Hybrid Solution

Given our goal, the best solution involves a hybrid of count quantization/Bloom filters and the existing exact model. In our corpus, there were 38 million trigrams that appeared exactly once. Therefore, in the consolidation stage (after the entire exact model has been built), the trigrams that appear only once n can be removed from the trigram table, and inserted into a Bloom filter using k (basically random) hash functions that map some set of objects to $\{0, \ldots, m-1\}$ (for some size m), which are easy to generate using the same scheme as described above. Mitzenmacher et al derives an equation that describes the optimal relationship between k and m: $k = \ln 2 \cdot \frac{m}{n}$ or equivalently $m = kn/\ln 2$. The Bloom filter will require m bits, which is determined by counting the number of singletons and computing the optimal size given the number of hash functions (which can be adjusted depending on error rate requirements).

During decoding, to find the counts of a particular trigram, we first look in the table as before. If it is not found there, we know that the trigram appeared either once or never appeared in the training set. Therefore, we hash it with the k hash functions, and test for them in

the Bloom filter. For sufficiently large m, we can produce tight bounds on the probability of getting a false positive f: $\left(\frac{1}{2}\right)^k$ (Note that if the trigram did appear before, the Bloom filter will give the correct answer because of the one-sidedness of the errors). Since trigrams that did not appear in training are unlikely to show up frequently during testing, the impact of the error is limited. In addition, in the small fraction of the time when the count is incorrect, it differs by only $1 - d = 0.25$ (because of the $\max(0, c(\cdot) - d)$). Therefore, for all trigrams that have been seen, the solution will give the correct counts. In addition, for all other trigrams, finding their entry in the table is faster because there are much fewer elements in the table.

In general, this idea can be extended to the trigrams that appear at most N times, but in practice the marginal benefit for any $N > 5$ is incredibly small because of the distribution of trigram counts. But even with $N = 1$, this can save $\frac{2}{3}$ of the memory currently used by the trigram model for the cost of an additional m bits (**Figure 5**). For example, for $N = 1$, the entire model will theoretically require 224 MB of RAM.

5 Experiments

Our experiments were conducted on servers with 6 GB of RAM running the Java Virtual Machine with 2 GB of heap space.

For the exact model, model construction takes approximately 205 seconds, including the time to compact the data structure (from hash table into sorted array). The memory usage before decoding is 777M-790M. The decoding takes 680-720s without caching, and 230-240s with caching (no significant difference between different cache sizes, due to the hit rates—**Figure 2**), and produces a BLEU score of **24.690**.

The noisy model builds the entire exact model first, then puts the trigrams that have been seen just once in a Bloom filter and discards them from the table containing exact counts. This additional step takes approximately 2 seconds. The BLEU scores (**Figure 4**), memory savings (**Figure 5**) and decoding time (**Figure 6**) can be seen in the appendix. Decoding became slower because of the additional membership checks for singletons (which are 2/3rds of queries) despite the multi-level table becoming smaller (thus making binary search faster). Because the test set is the same, the cache hit rates are equal (Bloom filter results are also cached). With 8 hash functions and a 60 MB bloom filter, the false positive rate (the rate that novel trigrams are labeled as singletons) is approximately 2.05%.

6 Extensions

There are only about 12,000 unique counts. Currently, counts require 25 bits but could be rank-encoded in 14 bits. In addition, if space were a concern each bigram could be indexed like words and thus be given unique integers. Since there are only 8 million bigrams, each bigram would be represented in 23 bits (instead of the current $19 + 19 = 38$).

However, for any of those bits to be recovered, the values must be packed more tightly together (while "compacting"), and thus an entire array will have to be packed into a separate data structure for "bitfields" so that operations can be done in increments of 37 bits $(23 + 14)$ instead of 32 or 64. This trades significant time for space, although caching should mitigate the drag on speed. A simpler alternative is to use a parallel short to represent the rank-encoded counts, saving 25% space.

We focused on optimizing the language model for speed, size and reliability, and thus on data structure design. Many promising improvements in decoding (especially the KN smoothing), like variable discounts should be further explored. For example, the d-value should be modified to depend on the counts (for fewer than 4 counts) and level of recursion in the probability estimation.

7 Conclusion

We explored many data structure optimizations that allow for very compact yet efficient representations for a trigram model, and applied the data structure to a test set of 2000 sentences using the Kneser-Ney smoothing technique. In addition, we also analyzed various noisy model techniques and concluded that a hybrid of the exact model and Bloom filter best allows us to exploit the distribution of frequencies in languages. Then we offered various ideas for extending our work to improve the space use. Ultimately we searched for a good balance between the various features like speed, accuracy, and size.

References

David Talbot and Miles Osborne. 2007. Smoothed Bloom filter language models: Tera-scale LMs on the cheap. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, Prague, Czech Republic.

David Talbot and Thorsten Brants. 2008. Randomized language models via perfect hash functions. In Proc. ACL-08: HLT.

Stanley F. Chen and Joshua Goodman. 1998. An empirical study of smoothing techniques for language modeling. Tech. Rep. TR-10-98, Harvard University.

American Psychological Association. 1983. *Publications Manual.* American Psychological Association, Washington, DC.

Yee Whye Teh. 2006. A Bayesian interpretation of interpolated Kneser-Ney. Technical Report TRA2/06, School of Computing, National University of Singapore.

Adam Pauls and Dan Klein. 2010. Faster and Smaller Language Models.

A. Broder and M. Mitzenmacher. 2005. Network applications of Bloom filters: A survey. Internet Mathematics, 1(4):485–509.

Appendix of Tables and Figures

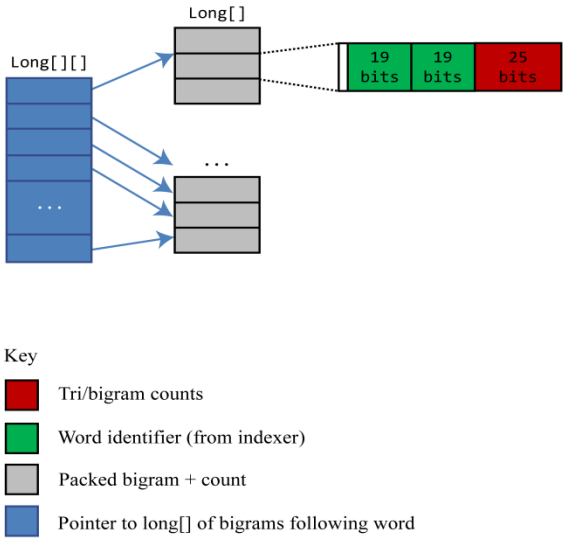

Figure 1. The multi-level table data structure for the exact language model. The many advantages to using the multi-level table become evident: keeping each individual suffix table small (thus making sorting/searching fast, and resizing tractable) and eliminating the redundant representation of the first word of a trigram (thus saving space).

Cache hit rates for various sizes

Cache size	10,000	100,000	1,000,000
Novel n-grams **not** cached	**79.5%**	**81%**	**81.2%**
Novel n-grams cached	**93.9%**	**95.6%**	**96.5%**

Figure 2. The trade-offs in cache effectiveness with cache size and whether novel n-gram (queried n-grams that were never seen in training) counts are cached.

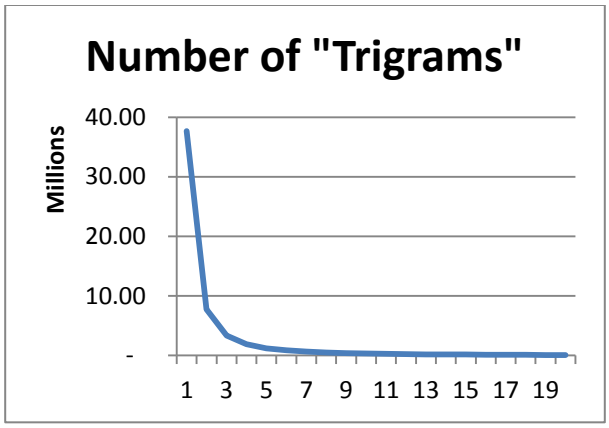

Figure 3. This shows the number of "trigrams" that appear a given number of times, suggesting significant potential space and time savings in separately representing the trigrams that appear only once in the training data.

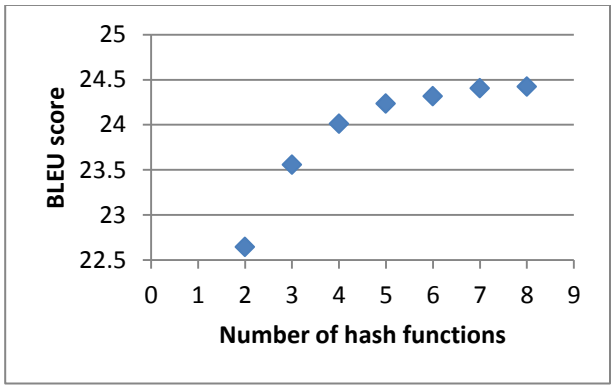

Figure 4. This shows the BLEU score that is achieved by our hybrid hashing scheme for a given number of hash functions (and the Bloom filter optimally sized according to Mitzenmacher). The score rises sharply when the second hash function is introduced, but the rise tapers off after 5 hash functions.

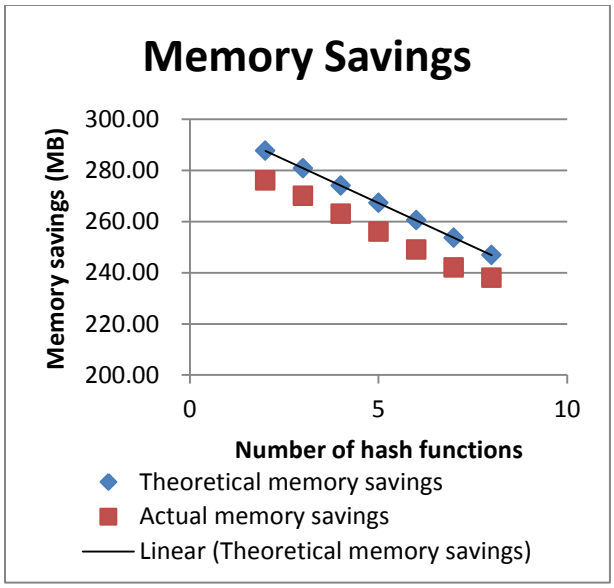

Figure 5. Amount of memory saved by using the hybrid-hashing scheme, comparing the theoretical savings to the actual (realized) savings. Observe that even with 8 hash functions, which guarantee a very low probability of false positive rate, over 200 MB is saved (both in theory and practice). Nearly 280 MB is saved if the threshold for BLEU is 22.

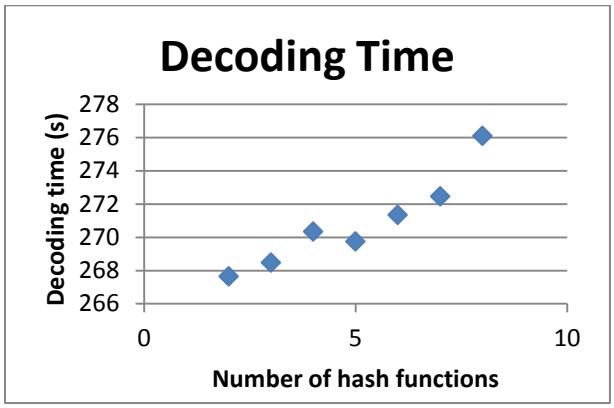

Figure 6. The time to decode n-grams that appear only once (or never) slightly with the number of hash functions because each n-gram must be hashed more times and more values need to be checked in the Bloom filter. The time to decode n-grams that appeared two or more times is not affected.