11-711 Assignment 1: Language Modeling

Xin Qian
xinq@cs.cmu.edu

Abstract

Our implementation on Kneser-Ney trigram languae model (Chen and Goodman, 1996) extends the basic backoff model with several optimization techniques. In this report we discuss how each individual parameter contribute to the end performance. Our best model achieves a BLEU score of 25.001 with a memory usage of 1023M and a decoding time of 330s.

1 Introduction

In this project we implemented a Kneser-Ney trigram languae model (Chen and Goodman, 1996) with several optimization techniques. We evaluated the language model as a component in a statistical machine translation system, with different configurations of parameters. In this report we discuss how each individual parameter that is related to the language model affect the end performance of the machine translation task. The effect comes in both efficiency (memory usage, decoding speed, LM build speed, etc.) and effectiveness (BLEU score). Our best model, implemented with multi-value key bi-gram count table and bit-packing achieves a BLEU score of 25.001 with a memory usage of 1023M and a decoding time of 330s. We also attempted caching to speedup decoding and quadratic probing to reduce average collision per access.

2 Implementation Details

In this project, besides the basic requirements, we implemented a multi-value hash table for bigram count table to store three types of fertility count. This reduces the computation cost during decoding phase as the fertility wouldn't need to be collected on the fly. A multi-value hash table is advantageous since we are representing trigrams as long type (64 bit) keys. This consumes most table memory and should avoid being replicated across several table. Technically to achieve this, we add three fertility fields, each as an `int[]` array, to the `longIntOpenHashMapBigram` class.

To this end, it is simple to calculate a rough estimate of memory usage. The statistics of the whole training corpus go as follows, num of unique unigram is 495172, num of unique bigram is 8374230 and num of unique trigram is 41627672. We implemented a 32-bit int to 32-bit int mapping for unigram count table which costs no more than 5M (not including the fertility overhead), a bigram count table of (K,V) 64-bit long to 32-bit int by 4 (including fertility) mapping which costs no more than 200M and a trigram count table of (K,V) 64-bit long to 32-bit int without any overhead, which costs no more than 500M. This adds up to around 700M, which is within the basic requirement.

Further, we attempted to substitute linear probing with the commonly-believed more robust quadratic probing. After obtaining the initial hashed position within a table, we increment the step size in quadratic function, rather than linear incrementation. We also implemented a LinkedHashMap-based LRU cache. It is important to tune the allowed cache size in order to tradeoff memory consumption and speedup. Despite the relative convenience of LinkedHashMap in purging the oldest used (key,value) pair. It is slightly disappointing to see a LRU cache size of 50,000 (12% of trigrams are cached) does not offer speedup as expected. We provide a brief analyze on potential reason in next section. As a final note, our hash function that maps 64-bit key to int range position is based on bit multiplication and left-shift [1]. This transports changes from high bit positions to low bit positions. An alternative solu-

[1] https://gist.github.com/badboy/6267743

tion is first converting the long variable to a string, then call Java's native `hashcode()` function.

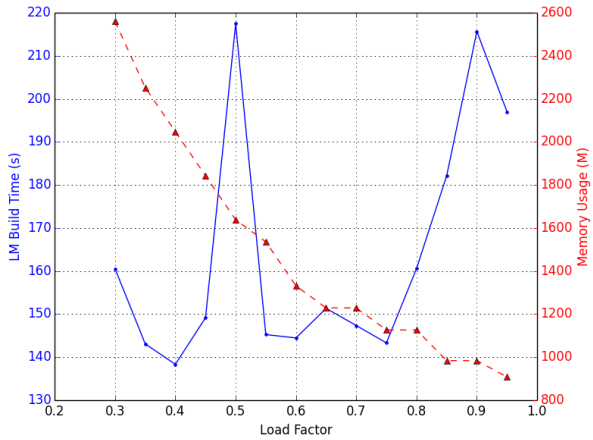
3 Experimental Evaluation

3.1 Final Performance

We submitted an **assign1-submit.jar** with a memory usage of 1023M, decoding time of 333.187s, building time of 156.717s and BLEU score of 25.001. This version utilized the best perfoming set of parameters, including load factor, discount factor, resize expansion coefficient, etc. Below we discuss each of the parameter regarding their role in the end performance.

3.2 Building Speed/Memory Usage vs. Load Factor

Figure 1: Build Time/Memory Usage vs. Load Factor

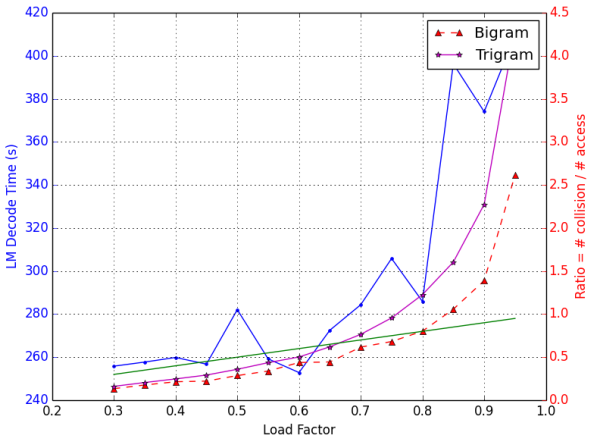


Load factor is an efficiency concern, which does not influence effectiveness. We see a clear negative correlation between load factor and memory usage. It is intuitive as we are wasting more empty cells in the hash table when the load factor is small, thus increases memory usage. Build time here fluctuates along an increasing load factor, it makes sense here as build time is not only influenced by how vacant the table is, but also by resizing frequency and resizing copying burden. Another possible reason is our experiment setup, of CPU throttle, when we run jobs in parallel.

3.3 Avg num of collision and decode time vs. Load Factor

We quantify the ratio here as $ratio = \frac{\#collision}{\#access}$. This plot is consistent with the figure in Wikipedia

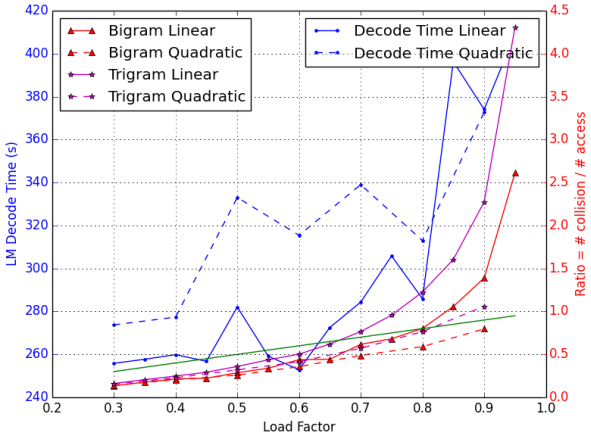
Figure 2: Decode Time/Ratio vs. Load Factor



of Hashtable [2] on the open address hashing section. Our plot is equivalent to the figure that compares the average number of cache misses required to look up elements in tables with chaining and linear probing. We observe an upgoing trend that when load factor increases, i.e., when the table is more packed the average collision per access increases. Ideally the ideal load factor should equal to the ratio. We plot the green curve to show the intersection point. An ideal load factor is likely to be 0.8. It is yet unknown to us why decode time at load factor = 0.5 is the lowest. Since we are running the experiment through a personal computer. this might affect the actual running time.

3.4 Avg num of collision and decode time between linear probing and quadratic probing

Figure 3: Decode Time/Ratio across probing scheme

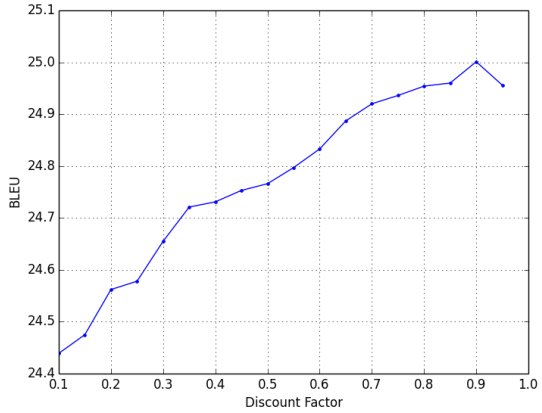


[2] https://en.wikipedia.org/wiki/Hash_table

Linear probing usually suffers from the problem of clustering. Clusters arise when an empty slot only shows up after a sequence of full slots, thus increases probing time and collision probability. Quadratic probing mitigates the issue by strategically producing milder and smaller clustering, which are separated by available slots. However, even for quadratic probing, as long as the initial probe position is the same, two probe sequences are the same. We observe a similar issue here, when using a quadratic function of $i^2 + 2i$. This quadratic probing function can be weak when dealing with large load factor. Take for example, for a pretty-packed table of size 11, and a load factor of 0.95, regardless of i value, some empty slots will never able to be visited, thus cause a dead loop to expand the count table.
Nevertheless, we could still say that quadratic probing is more robust at reducing average collision per access. The upgoing trend of average collision per access vs. an increasing load factor is more linear than linear probing.

3.5 BLEU vs. Discount Factor

Figure 4: BLEU vs. Discount Factor



Discount factor only concerns with effectiveness, which is reflected by BLEU score. We observe an optimal discount factor value of 0.9. Discount factor controls how much backoff we receive from lower-order probability while calculating current order of probability, i.e., how much smoothing we would need. Our explanation here is that the corpus is not large enough to exhaustively include most trigram, thus it relies heavily on backoff.

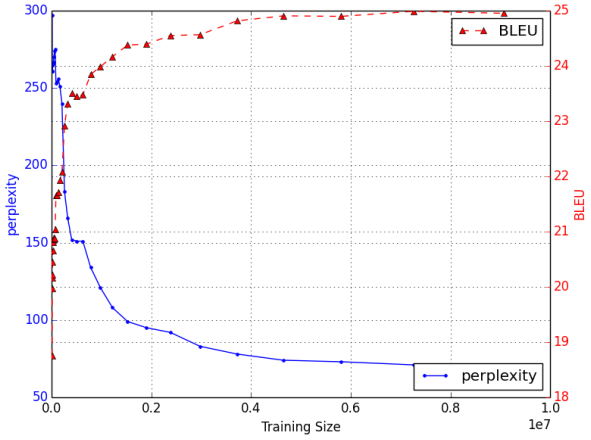
3.6 Memory Usage and Speed vs. Caching Capacity

We implemented a LinkedHashMap-based LRU cache to store most recently used trigram probability. However, experiments show our cache does not speedup over non-cache version, both for insertion-ordered HashMap and for access-ordered HashMap. Possible explanation for this senario is that we are storing too much trigram in the cache. Amortized probing time from the cache takes longer than a start-from-scratch trigram probability calculation. As a next step, we are going to plot the trigram count histogram to understand the distribution. For example, if it follows the power law distribution, then we could set the cache size to 20% of the unique trigram count.

3.7 Perplexity vs. Training Size

Perplexity, defined as $2^{-\sum_{x \in trigram\_set} p(x) log_2 p(x)}$ [3] is the metric to evaluate "branching factor" of a language model. The less branching it has on that decision, the more determinant/certain the LM is on a given prediction. Here we observe an upgoing trend of BLEU in parallel with an downward trend of perplexity. The more training data a LM is trained from, the more accurate the LM is.

Figure 5: LM perplexity/BLEU vs. Training Size



4 Error Analysis and Potential Extension

Currently we are approximating the true probability of a words sequence simply by the conditional probabilities to the proceeding local con-

[3] This equation is integrated from two course notes http://www.cs.columbia.edu/~mcollins/lm-spring2013.pdf and http://phontron.com/class/mtandseq2seq2017/mt-spring2017.chapter3.pdf

Table 1: Caching Implementation on Speedup

Cache Implementation	Memory Usage	Build Time	BLEU	Decoding Time
Insertion-ordered LinkedHashMap	1.0G	169.324s	25.001	405.214s
Access-ordered LinkedHashMap	1.0G	151.036s	25.001	499.637s
Submitted non-Cache baseline	1023M	156.717s	25.001	333.187s

text. However, the actual word choice could also depend on subsequent context or be less dependent on the actual word choice in immediate precedent. Manual inspection of error pattern suggests us to also consider skip-gram that combine several lower order models to tolerate rare words when they appear in the context. For example, *by mrs theato , on behalf of the committee on budgetary control*, the word *budegetary* is hard to be predicted and could be treated as a rare world, it might affect subsequent prediction of the word *control* given current trigram LM scheme.

```
\usepackage[nohyperref]{acl2017}
```

References

Stanley F Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, pages 310–318.