Implementing Trigram Language Model with Kneser-Ney Smoothing Algorithms for NLP 11-711: Assignment 1

Kinjal Jain[1]

***Abstract*— This paper addresses various strategies employed and experiments conducted to efficiently implement a Trigram Language Model with Kneser-Ney Smoothing for monolingual English text. The language model is evaluated extrinsically by incorporating it into a Machine Translation system and measuring its translation quality. The Language Model implementation is constrained by memory usage, build time of the language model and decoding speed for the translations.**

I. INTRODUCTION

A Language Model is a probability distribution over sequence of words. It estimates the relative likelihood of phrases and serves as a mainstay for a variety of Natural Language Processing applications such as Information Retrieval, Part-of-Speech Tagging, Machine Translation, etc.

There are a variety of language models, but this paper specifically discusses Trigram based Statistical Language Model. The Trigram Language Model assigns a probability to every possible sentence by estimating the probability of each word w given past 2 words. For instance, if you have a sequence W = $\{w_1, w_2, .., w_n\}$,

$$P(w_1, w_2, .., w_n) = \prod_{i=1}^{n} (P(w_i|w_{i-2}, w_{i-1})) \quad (1)$$

$$P(w_i|w_{i-2}, w_{i-1}) = \frac{Count(w_{i-2}, w_{i-1}, w_i)}{Count(w_{i-2}, w_{i-1})} \quad (2)$$

Several challenges are involved in building this language model. With a finite vocabulary size to implement it, out of vocabulary words need to be handled. Moreover, most higher order n-grams are very few in number and thus, sparsity has to be dealt with. Smoothing techniques are applied to the model to deal with the problem of sparsity and unknown words. In this paper, Kneser-Ney Smoothing technique is implemented over Trigram Language Model. Kneser-Ney Smoothing combines the idea of absolute discounting and fertility by reallocating some fixed probability mass ie. $discount$ from higher order n-gram to fertility counts of lower order n-grams. The Recursive formula for the same is given as:

$$P(w_i|w_{i-n+1}^{i-1}) = \frac{max(c^{'}(w_{i-n+1}^{i}) - d, 0)}{\sum_{v \in V} c^{'}(w_{i-n+1}^{i-1}, v)} + \alpha(w_{i-n+1}^{i-1}) P(w_i|w_{i-n+2}^{i-1}) \quad (3)$$

where, α is interpolation weight, d is discount, $c^{'}$ is count for highest order n-gram (3 in our case) and fertility for lower order n-grams, v being the words in Vocabulary.

[1] Email:kinjalj@andrew.cmu.edu

II. IMPLEMENTATION DETAILS

Two Kneser-Ney Trigram Language Model architectures are implemented in this paper.

Model 1: The first model (submit.jar) satisfies the basic constraints of the assignment. As part of the $KneserNeyTrigramLanguageModel$ class, the various counts for unigrams, bigrams and trigrams are stored along with the fertility counts for unigrams and bigrams. For unigrams, $EnglishWordIndexer$ is used for storing the index, and an int array is used to store the counts. For three fertility types based on unigrams, parallel int arrays are used to store the fertility counts. For bigrams and trigrams, a custom designed Open Address Hash Map with Linear probing is used. They are encoded with bit packing method so that memory is efficiently used. Several experiments were conducted to identify the optimal hash function which are discussed in later sections. The classes $BigramIndexer$ and $TrigramIndexer$ extend the class $OpenAddressHashMap$ which uses a primitive $long$ array to store the indexes and int array to store counts. For two fertility types based on bigrams, parallel int arrays are used to store the fertility counts. The approach is to first store the unigrams and bigrams while reading the sentences, and then store the trigrams in second pass over sentences so that the trigram based map can be initialised at run time. This also helps in reducing build time of the model by saving some time spent on resizing and collisions if the initial map size is small.

Model 2: The second model (best.jar) satisfies the basic constraints of the assignment, but saves the memory substantially as compared to Model 1. The saving comes due to a different approach used for storing trigrams. Instead of using $OpenAddressHashMap$, a custom $TrigramArray$ class is implemented which stores both the index and count in just one primitive $long$ array. This was possible only due to the observations made about the unique properties of the collection used for building the language model which are discussed in $Collection\ Statistics$ section later. The Array stores the trigram key in first 44 bits as a combination of bigram index which uses 24 bits and unigram index which uses 20 bits. The remaining 20 bits are used to store the trigram counts. The bigram index fits into 24 bits because the size of the array used to store bigram counts didn't exceed 2^{24} and the array index where bigram was stored in bigram counter is used as bigram index in trigram array. Similarly, the unigram index fits into 20 bits because the size of the array used to store unigram counts didn't exceed 2^{20} and the

array index where unigram was stored in unigram counter is used as unigram index in trigram array.

III. Collection Statistics

After implementing Model 1, following observations were made about the Sentence Collection provided to build the Language Model:

N-gram	num(Unique N-grams)	max(N-gram Count)
Unigram	495172 $< 2^{19}$	19880264 $< 2^{25}$
Bigram	8374230 $< 2^{23}$	7109704 $< 2^{23}$
Trigram	41627672	1048575 $< 2^{20}$

These indicate that int arrays are sufficient to store counts for unigrams, bigrams and trigrams. Further, following observations were made on the Map created for unigrams and bigrams using optimal values of $loadFactor$ as 0.70, $bigramResizeFactor$ as 2 and $trigramResizeFactor$ as 1.5:

N-gram	N-gram Map size
Unigram	800000 $< 2^{20}$
Bigram	12000000 $< 2^{24}$
Trigram	63591816

Based on the size of the bigram and unigram maps, it was concluded that if we use the index of the bigram where it is located in bigram array of size and index of the unigram where it is located in unigram array of size, we can fit the combined index in 44 bits. Moreover, the value of max(Trigram Count) doesn't exceed 2^{20}, so 20 bits are sufficient to store the trigram counts. This observation is leveraged in Model 2.

IV. Experiments & Analysis

Discount & Load Factor: The optimal $discount$ and $loadFactor$ values were identified to be 0.90 and 0.70 by experimenting between the range {0.70, 0.90} with various $loadFactor$ values ranging from {0.70, 0.80} and picking the one which gives good BLEU score and consumes less memory. Following is the comparison table:

Dis-count	Load factor	Model 1 BLEU	Model 1 Memory	Model 2 BLEU	Model 1 Memory
0.70	0.70	24.921	1.1GB	25.003	902MB
0.70	0.75	24.921	1.1GB	24.921	902MB
0.70	0.80	24.921	1.1GB	24.921	902MB
0.80	0.70	24.956	1.1GB	24.956	902MB
0.80	0.75	24.956	1.1GB	24.956	902MB
0.80	0.80	24.956	1.1GB	24.956	902MB
0.90	0.70	25.003	1.1GB	25.003	902MB
0.90	0.75	25.003	1.1GB	25.003	902MB
0.90	0.80	25.003	1.1GB	25.003	902MB

Hash Functions: 3 Hash functions namely FNV1 hash, Murmur hash, CERN prime hash have shown to give good results in the past to minimise collisions. Once the hash was calculated, the key was calculated to be hash value modulo HashMap size. Following observations were made about the Language Model Building Time, Decoding Time for each of these hash functions with $loadFactor$ value as 0.70 and CERN prime hash was identified to give the best results.

Model 1			
Hash Function	LM Build Time (in Secs)	Decoding Time (in Secs)	Memory Used
FNV1	212.639	240.342	1.1GB
Murmur	214.432	235.805	1.1GB
CERN	203.127	215.064	1.1GB

Model 2			
Hash Function	LM Build Time (in Secs)	Decoding Time (in Secs)	Memory Used
FNV1	191.245	281.3	902MB
Murmur	204.1	262.273	902MB
CERN	188.020	242.01	902MB

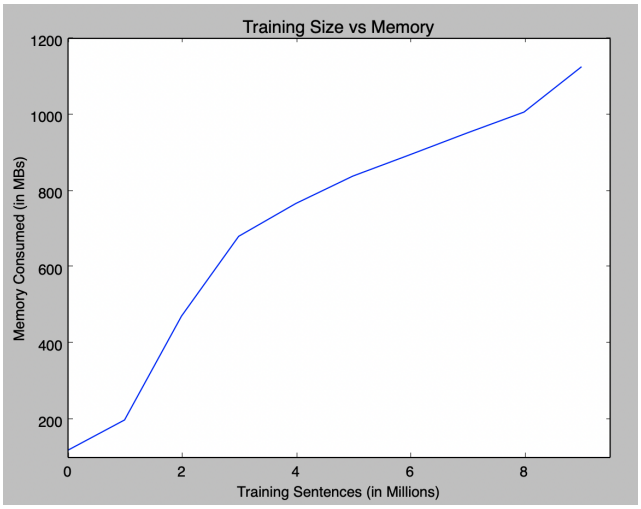
HashMap Resizing Factor: For Model 1, Various resizing factors ranging from 1.2 to 2 were tried. The optimal resizing factors for BigramIndexer and TrigramIndexer were discovered to be 1.5 and 1.5 respectively. For Model 2, in order to fit the unique bigram count under 1,67,77,216 ie. 2^{20}, the resizing factor for BigramIndexer had to fixated to 2 which resulted in an array of size 1,20,00,000 for bigrams. For TrigramIndexer optimal resizing factor was found to be 1.5 by experimentation. There are some previous experiments which present the idea of using a Prime number as a size of array since modulo by a prime number while finding the index can reduce collisions, but for the above two models, not much difference was seen, in fact memory consumption increased in order to always set the size of the array to the next prime after resizing it.

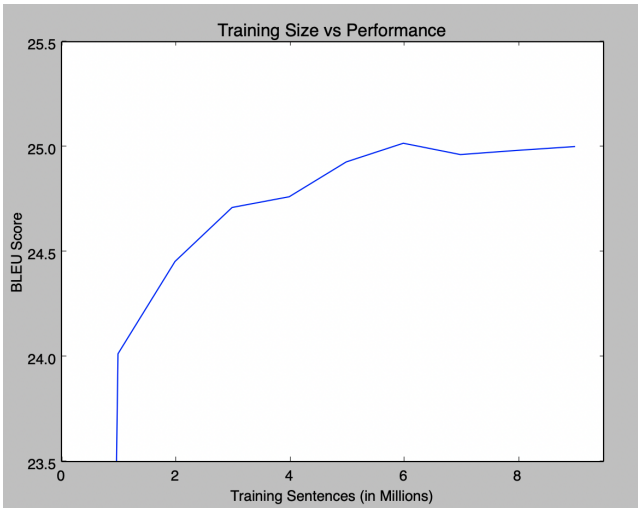
Initial Array Size: The initial size for storing unigram counts, unigram fertility counts and bigram fertility counts was set to be 200000. the initial size for storing bigram indexes and counts was set to be 1500000. The trigrams are initialised at run time, with final bigram map size. These values were chosen so that the total memory consumed at the start doesn't exceed 50M which is the hard limit for sanity checks. Another, crucial deciding factor was that number of collisions and resizing time are reduced as much as possible.

Search for better BLEU score: After successfully implementing Model 1 and Model 2, other approaches for smoothing were also explored. Modified Kneser-Ney Smoothing technique was also tried as part of $ModifiedKNTrigramLanguageModel$ class included in Model 1 submission which discounts differently based on the count or fertility of a n-gram. However, that gave a BLEU score of around 22 which is much lesser than the BLEU score observed by normal Kneser-Ney Smoothing technique. Moreover, it takes more memory to store individual fertility counts ie.$N_1, N_{1+}N_2, N_{3+}$ scores for unigrams and bigrams.

Reduction in Decoding Time: After successfully implementing Model 1 and Model 2, caching was tried to store the n-gram probabilities that are most recently requested, but the cache itself was taking a lot of time to traverse and so, it didn't help in reducing the decoding time.

Modifying the Training Size: BLEU score and Memory consumption was observed over a varying number of training sentences ranging from 1 million to 9 million. The expectation was that BLEU score will improve as the training size increases, however, it was observed that BLEU score was maximum for training size = 6 million sentences which was recorded to be 25.019 which is slightly greater that the BLEU score 25.003 observed for the full training corpus. The explanation for this could be that translation model was able to learn with this limited vocabulary itself and did not require any more training data for language model. Memory consumption increased with size as expected owing to an increase in the number of unigrams, bigrams and trigrams. The trends for the same on Model 1 are:





V. Performance

The following performance metrics are reported for both the models on MacBook Pro 1.4GHz Intel Core i5 Processor with JDK 1.8:

Model	BLEU Score	Decoding Time (in Secs)	Memory Used
Unigram	15.535	5.227	137M
KN Trigram Model 1 (submit.jar)	25.003	215.064 (best)	1.1GB
KN Trigram Model 2 (best.jar)	25.003	242.01 (best)	902MB

Although both the models achieve the same value for optimal BLEU score, and Model 2 takes a little more decoding time than Model 1, Model 2 saves about 220MB of memory compared to Model 1. All the basic constraints posed by the assignment are satisfied.

VI. Conclusions

As part of the assignment, 2 Kneser-Ney Smoothing based Trigram Language Models were successfully implemented. The average Language Model Building Time is 194.2 secs which includes training over 9 million sentences and efficiently storing the n-gram counts. On an average, decoding for Model 1 takes 220 secs and Model 2 takes 250 secs. As the size of training data increase, the performance of the model (measured extrinsically with BLEU score here) improves at the cost of memory. Model 1 uses approximately 1.1GB memory as compared to Model 2 which uses around 900MB.

References

[1] Kneser, R. Ney, H,"Improved backing-off for m-gram language modeling." in Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, Detroit, MI, volume 1, pp. 181184. May 1995

[2] Pauls, Adam and Klein, Dan, "Faster and Smaller N-gram Language Models," in Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1, pp. 258-267, 2011

[3] Stanley F.Chen, Joshua Goodman, "An empirical study of smoothing techniques for language modeling," Centre for Research in Computing Technology, Harvard University, Cambridge Massachusetts.

[4] John Zukowski, "PrimeFinder" in (2001) Colt. In: Java Collections. Apress, Berkeley, CA.

[5] Fowler, Glenn, Landon Curt Noll, Kiem-Phong Vo, Donald Eastlake, and Tony Hansen. "The FNV non-cryptographic hash algorithm." Ietf-draft, 2011

[6] Appleby, Austin. "MurmurHash." URL https://sites. google. com/site/murmurhash , 2008

[7] Jurafsky, Dan. "Speech language processing." Pearson Education India, 2000.