Knesser Ney language model : Implementation description and insights

Harsh Jhamtani
Language Technology Institue
Carnegie Mellon University
jharsh@cs.cmu.edu

Abstract

This report discusses implemention of a Knesser Ney language model in Java alongwith the impact of various optimizations on reducing memory footprint of the language model and the time taken in decoding phase of Machine Translation task using the language model. The KN model was trained on about 9 million sentences in training data. Testing using MT decoder was done on 2000 sentences. An exact KN model achieved a memory footprint of **680 MB**, with Machine Translation decoding time of **189 seconds**[1], which is about **22.6x** as compared to baseline unigram model. BLEU score of machine translation was **24.93**. The model builds in about 16 minutes.[2] Another model using different data structures was able to achieve decoding time of **120 seconds**, though with a memory footprint of 1.1GB.

1 Introduction

Various configurations and techniques were tried to improve performance of Knesser Ney language model. The best configurations will be summarized in conclusion (Section 4).

1.1 Overview of Knesser Ney Language Model

Knesser Ney smoothing is a technique to calculate probability distribution of n-grams of words. It builds on concepts of backoff to lower ngram models as well as fertility counts.

Knesser Ney model of order 3 to predict probability of seeing w_3 given previous two words as $w_1 w_2$ is given by following equations ((Chen and Goodman, 1999)):

$$P_3(w_3|w_2w_1) = \frac{\max\left(c\left(w_1w_2w_3\right) - D_3, 0\right)}{c\left(w_1, w_2\right)} + \lambda(w_1w_2)P_2\left(w_3|w_2\right)$$

$$P_2(w_3|w_2) = \frac{\max\left(N_1^+\left(\cdot w_2w_3\right) - D_2, 0\right)}{\sum_w N_1^+\left(\cdot w_2, w\right)} + \lambda(w_2)P_1\left(w_3\right)$$

$$P_1\left(w_3\right) = \frac{N_1^+\left(\cdot w_3\right)}{N_1^+\left(..\right)}$$

$N_1^+(x\cdot)$ is number of distinct words w such that count of $xw > 0$

$$\lambda\left(w_1w_2\right) = \frac{D_3N_1^+\left(w_1w_2\cdot\right)}{C\left(w_1w_2\right)}$$

$$\lambda\left(\omega_2\right) = \frac{D_2N_1^+\left(w_2\cdot\right)}{\sum_w N_1^+\left(w_2w\right)}$$

2 Implementation description

In this section, I will describe data structures and algorithms used for the implementation.

2.1 Count tables

It is important to get an understanding of the size of data at hand. Number of distinct ngrams in the provided data set are as follows:

- *Unigrams:* 495,172
- *Bigrams:* 8,374,230
- *Trigrams:* 41,627,672

Ngram encoding scheme: Considering the total number of unigrams, every unigram can be represented uniquely using 20 bits. A simple way to represent higher order ngrams is to successively append bits of constituent words. e.g. a trigram can be represented by 3*20=60 bits.

Types of values to be stored: Various counts and values needed to be stored can be observed using equations in section 1.

For every *unigram* w, we need $c(w)$, $N_1^+(\cdot w)$,

[1] average over three runs

[2] Code was run on a Ubuntu 16.04 machine with 6th Generation Intel Core i5-6200U Processor and 8GB DDR

$N_1^+(\cdot w\cdot)$ and $\lambda(w)$.
For every *bigram* w_1w_2, we need $c(w_1w_2)$, $N_1^+(\cdot w_1w_2)$, and $\lambda(w_1w_2)$.
Note that $N_1^+(w_1w_2\cdot)$ and $N_1^+(w\cdot)$ are used only to calculate lambda's - so there is no point storing them in the final language model.

Following steps are followed to populate these values:
- Training data is scanned to get counts of unigrams, bigrams, trigrams
- Using set of bigrams, calculate $N_1^+(\cdot w)$
- Using set of trigrams, calculate $N_1^+(\cdot w_1w_2)$
- Using set of bigrams and their fertility counts, calculate $N_1^+(\cdot w\cdot)$
- Calculate lambda's for unigrams and bigrams
Various optimizations are carried out to reduce time and memory footprint. These are described later in the section.

2.1.1 Open Address Hashing

Linear probe open address hashing is used for storing various counts for bigrams and trigrams. Load factor γ controls the memory requirement and collisions. We always need to allocate space for much more than the number of elements (for faster access in case of collisions).

2.1.2 Sorted keys

In this case, keys are finally sorted so that any retrieval can be done through binary search on keys. However maintaining sorted list of elements while constructing a language model is not trivial. So to achieve this model, I first do preprocessing using hash table described earlier, and then finally convert from hash table representation to sorted keys representation.

2.2 Caching

As shown in section 1, computing a trigram probability involes a number of computational steps. Since decoder may request probabilities for same trigram multiple times, it might be a good idea to save calculated calues in a limited capacity cache during runtime. Cache is imlemented using a hash table with replacement on collision.

2.2.1 Bigram caching

This is based on hypothesis that a MT model would try out different words consecutively at the end of previous two words. So, various values of last accessed bigram can be cached.

2.3 Compressing counts

Many of the values strictly take integer values. On observing, there are only 10153 distinct values invarious bigram (counts and fertility counts) tables. Similar case is for trigram counts. In both cases, number of distinct values can be represnted in less than 16 bits. So, we can replace count (which occupies 32 bits) with the rank of the count in the set of distinct counts, thereby saving 16 bits per such count. A small overhead would be storing a mapping from ranks to actual values.

3 Experiments and Insights

Table 1 shows the decoding times and memory footprints as additional features are added to the model. The + sign indicates addition of the technique (with all previous techniques still present). Note that D_2 and D_3 were both set to 0.75 for the purpose of these experiments. (The BLEU scores were found to vary slighly with change in D2 and D3.)

Technique	**Memory**	**Decoding time (secs)**
Unigram	128 MB	8.34
Knesser Ney Trigram Counts using hash table ($f2$ function & 0.7 load factor)	1.3 GB	205.3
+ Caching (12 million capacity)	1.3 GB	133.7
+ Compression of trigram counts	1.0 GB	125.2
+ Compression of bigram counts	1.0 GB	124.2
Sorted key tables Sorted key (binary search) trigrams	911 MB	163
+ Sorted key (binary search) bigrams	811 MB	169
Reduce cache to 1 million from 12 million items	**680 MB**	203.1
+ Bigram caching	**680 MB**	**189.0**

Table 1: Performance on addition of various features

Discussion about Table 1:

- Caching provides significant speedup. Im-

[2] 12 million capacity cache is considered subsequently

pact of cache size is discussed later in the section.

- Encoding various counts to ranks reduces memory footprint from 1.3GB to 1.1GB, without any significant increase in model construction time. Impact on running time is insignificant. Also, more reduction comes from trigrams - since number of trigrams are much more.
- Using sorted keys instead of hash tables provides reduction in memory footprint Due to compacter representation, reduction in memory was expected.
- Improvement in performance due to bigram caching supports the hypothesis described earlier.

3.1 Training data size

Figure 1 shows the counts of distinct unigrams, bigrams and trigrams with the increase in training data size. While number of unigrams and bigrams tend to flatten out as the full training data size is reached, number of trigram continue to grow significantly throughout.

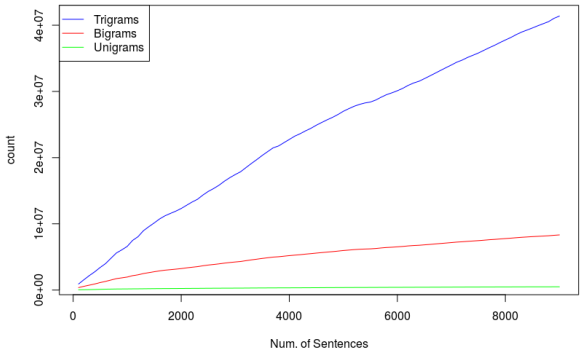

Figure 1: Counts of ngrams

Since a large part of **memory** requirement of the langauge model is due to storing various counts for ngrams, it is expected to increase with increase in number of ngrams. Also, since trigrams continue to grow significantly with training data size, so does the model size. As per Table 2 [3], marginal benefits in **BLEU** score keep on decreasing with increase in data. Model trained on just 500K sentences has about 20% less memory footprint, and takes about 20% less decoding time as compared with the model buil using entire data set. **Decoding time** doesn't change much after training on 40*100K sentences. Initial increase in decoding time can be attributed to more seen ngrams, and thereby slighly more computational steps e.g. subtracting D from count (for unseen ngrams, we switch to lower order model skipping a step of subtracting 'D')

#Sentences (in 100K)	Memory	Decoding time (secs)	BLEU score
5	670 MB	135.0 s	23.3
10	687 MB	140.1 s	23.9
15	703 MB	149.1 s	24.3
20	713 MB	151.6 s	24.4
40	749 MB	160.4 s	24.7
60	777 MB	154.6 s	24.9
80	800 MB	158.9 s	24.8
All	811 MB	163.4 s	24.9

Table 2: Impact of data set size (Counts encoding and cache of 12 million capacity was used with sorted keys table for this experiment)

3.2 Hashing table: Load factor and Hash function

To investigate the impact of choice of hashing function, I experimented with following hash functions:

1. $f_1(key) = key\%tableSize$
2. $\mathbf{f_2(key)} = ((((\mathbf{key} << \mathbf{30}) >>> \mathbf{30}) + (\mathbf{key} >>> \mathbf{30})) * \mathbf{99929})\%\mathbf{tableSize}$
3. $f_3(key) = (key * 99929)\%tableSize$

Some differences in performance metrics are observed for different hash functions. Table 3 shows the time and memory performance with changing load factor for hash functions f_1, f_2 and f_3. Using f_2 at load 0.65 provides a decoding time of **126 seconds**, which is significantly better than best performance using sorted keys table. With increase in load factor, the decoding time inrease for $f2$, as expected. However, decoding time trend for f_3 is not intuitive - it first increases with load factor, then decreases slightly for loads 0.80 and 0.85, but again increases for 0.90.

Poor performance of f_1 can be realized by analyzing the key encoding scheme. Let's say if table size is less than 16 million (about 2^{24}). Now all the trigrams beginning with a word w but having same subsequent words (ww_2w_3) will map to

[3] Counts encoding and caching was used with sorted keys table for this experiment

Load factor	Memory	Decoding Time		
		f1	f2	f3
0.55	Heap error	-	-	-
0.65	1.1 GB	576.6 s	126.6 s	141.9 s
0.70	1.0 GB	621.2 s	124.2 s	131.0 s
0.75	980 MB	932.3 s	131.3 s	136.3 s
0.80	936 MB	>1000 s	134.0 s	126.5 s
0.85	898 MB	>1000 s	148.2 s	131.2 s

Table 3: Impact of choice of hashing function and load factor

same to same address, as the first word wouild not play any role, thereby causing collisions. f_2 rectifies this issue by using the bits representation of the first word of the trigram as well.

3.3 Impact of cache size

Figure 2 shows how ratio of number of number of cache hits to total cache requests at various cache sizes (smallest cache size is 4 elements, while biggest is 100K elements). Note that cache of just 300 elements achieves more than 50% cache hits. Figure 3 shows the same measure at large cache sizes. There is little performance improvement once cache hit percentage reaches about 80%.

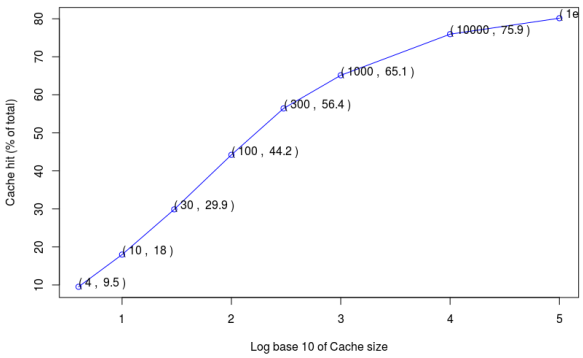

Figure 2: Cache hit percentatage at various cache sizes (at small cache sizes)

3.4 Impact of bigram caching

I applied bigram caching over following configuration: encoded counts, hash tables with load 0.6 and f_2 function, cache of size 13 million. The resulting decoding time is **120 seconds**. With sorted key table, decoding time improved from 203 seconds to **189 seconds**. Bigram caching hits are

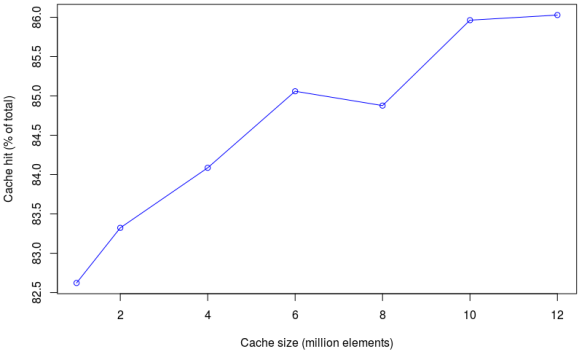

Figure 3: Cache hit percentatage at various cache sizes (at large sizes)

74.7% of the total requests.

4 Conclusion

Configuration of final models:
Based on the experiments, following model configurations had favorable performance. Choice among them would depend on relative importance of decoding time versus memory footprint.

- Smallest model: 1 million capacity cache, counts encoding using ranks, sorted keys count tables. **Decoding time: 189 seconds. Memory: 680 MB. BLEU Score: 24.9**
- Fastest model: 13 million capacity cache, counts encoding, Hash count tables with load factor 0.6 and f_2 hashing function. **Decoding time: 120 seconds. Memory: 1.1 GB. BLEU Score: 24.9**

To conclude, there are decreasing marginal returns of having more training data. Cache plays a significant role - due to nature of queries issued by the decoder. There are apparent trade-offs between model size and BLEU score, and model size and decoding time. Decoding time is motly influenced by cache size and data structure (sorted keys vs hash table) used. Choice of D_2 and D_3 does have an impact on BLEU score.

References

Stanley F Chen and Joshua Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394.