Kneser-Ney Trigram Language Model

Ziqian Luo
Carnegie Mellon University
ziqianlu@andrew.cmu.edu

Abstract

The project implemented a Kneser-Ney language model to find relationships between BLEU performance,speeding of building language model, decoding test and memory usage in running the model. The final model need 1.8G memory space, 437s to decode the test under a best pre-defined bigram and trigram table size and the final BLEU score is 20.642. This short paper will talk about what difficulties I met in running the experiments and methods to solve them and will discuss some other factors that will influence the model performance like cache size, load factor, discount factor, type of hash function and pre-defined table size. The memory and BLEU score did not meet course requirements due to I cannot find a suitable hash function so that conflicts happen a lot.

1 Implementation details

Kneser-Ney Smoothing is widely considered as one of the most effective and classic methods of smoothing.(Hermann Ney and Kneser, 1994) It is popular due to its use of absolute discounting by subtracting a fixed value from the probability's lower order terms to omit n-grams with lower frequencies. The recursive formula about the model are shown as follows:

$$P(w_3|w_1w_2) = \frac{\max(c(w_1w_2w_3) - d, 0)}{\sum_{v\in V} c(w_1w_2v)} + \alpha(w_1w_2)P(w_3|w_2) = \quad (5)$$
$$= \frac{\max(c(w_1w_2w_3) - d, 0)}{c(w_1w_2)} + \alpha(w_1w_2)P(w_3|w_2) \qquad \alpha(w_1w_2) = d \cdot \frac{N_{1+}(w_1w_2\bullet)}{c(w_1w_2)}$$
$$P(w_3|w_2) = \frac{\max(N_{1+}(\bullet w_2w_3) - d, 0)}{\sum_{v\in V} N_{1+}(\bullet w_2v)} + \alpha(w_2)P(w_3) = \quad (6)$$
$$= \frac{\max(N_{1+}(\bullet w_2w_3) - d, 0)}{N_{1+}(\bullet w_2\bullet)} + \alpha(w_2)P(w_3) \qquad \alpha(w_2) = d \cdot \frac{N_{1+}(w_2\bullet)}{N_{1+}(\bullet w_2\bullet)}$$
$$P(w_3) = \frac{N_{1+}(\bullet w_3)}{\sum_{v\in V} N_{1+}(\bullet v)} = \frac{N_{1+}(\bullet w_3)}{N_{1+}(\bullet\bullet)} \quad (7)$$

Figure 1: Recursive formula

As the formula shows, we need to store count of unigram, bigram, trigram words, and also the fertility of those words. Accordingly, at the first step I need to build up three tables containing counts and fertility of unigram, bigram and trigram. In order to save memory, I used bits shift trick to pack two or three word index into one long value. For example. if we want to store a trigram entry, we need to shift word1 left by 40bits, shift word2 left by 20 bits so that space is fully used in a 64 bit machine. The next step is to store those combined index. However, word index returned by dataset is too large to be directly stored in the array and that's why we need a hash map to convert those shifted index numbers. I choose to use the simplest % method, which used mod operation to get a new position from the large combined index. It's not the best choice since there are many conflicts and we have to waste a lot time going to next position to find the entry we want. I have tried other hash functions but they behave worse compared to mod operation, which will be talk in part 3. In order to decrease map conflicts, I did a lot of experiments to find the best bigram and trigram table size to store their number. If the size of the table is too small, program will run very slowly and it takes a lot of time to build the model and if the table is too large, it will be faster and may be able to run out of memory of 2G. More details of their relationships will be discussed in part 3. In order to count the fertility, I used a twenty bit mask to get low, middle and high parts of the packed words and store them in an array just like building gram tables. I didn't use a recursive method to compute the probability since it ran very slow and I split it into three methods to deal with unigram, bigram and trigram respectively. In order to speed up the process, I used the cache mechanism to store visited entry. I tried to use LRU policy to make the cache memory more efficient but it would be slow so I initialized a very large cache size to store them. More details about this will also be discussed in part 3. There are also some special

cases like the probability of unseen words are 1/V and sometimes trigram model will go back to bigram model and bigram go back to unigram. In my implementation I considered to process these situations first so that I would do less computation and program ran faster.

2 Performance

The performance varies a lot with different parameters. Among oceans of experiment, the best parameters are as follows:

- bigram table size = 12000000
- trigram table size = 62000000
- loadfactor = 0.8
- discountfactor = 0.75
- bigram cache = 20000000
- trigram cache = 35000000

The model under this setting takes around 3 minutes to count 9000000 sentences and 436s to do the decoding test show as below.

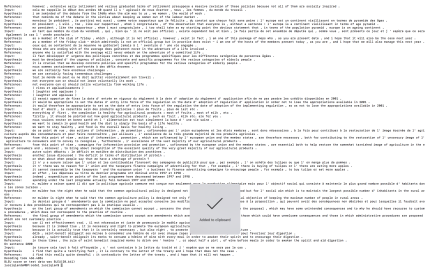

Figure 2: Best decoding time screen shot

The total run time is below 30 minutes but the decoding time is at the boundary of 50 x UnigramRuntime(8.3-9.3s) The total memory usage is 1.8 G in this setting, which exceeds the 1.3G limit. The BLEU score by a machine translation test is 20.641.

3 Experiments

This part will explain how some hyper parameters will influence my model like cache size, predefined n-gram table size, loadfactor. Certainly, how to define the unseed words probability and how to select the discount factor will also affect the BLEU value. However, the hash function, seems to be the key to success and a perfect hash map from long to int will absolutely save me even all the previous parameters are not at their best!

Cache Size	Decoding Time
no cache	2800+s
15000000+30000000	563s
15000000+40000000	528s
20000000+33000000	487s
20000000+35000000	436s
20000000+40000000	Out of Memory

Table 1: Decoding time in different cache size

3.1 Cache Size

Different cache sizes are used for bigram and trigram searching in computing probability. Before I compute the probability of an entry, I firstly look up the cache that if there is an entry used last time which saves a lot of time. However, if the cache is too big, it will run out of memory. Some values are recorded in the table 1.

3.2 Pre-defined Table Size

In my implementation, the predefined table size will greatly affect the table building time and final performance. If it is too small, hash conflicts occurred a lot in the early phase and it takes a far longer time to build the table if it is to large, memory is not enough. So I used nearly full memory given to speed up the process. The details are illustrated as below:

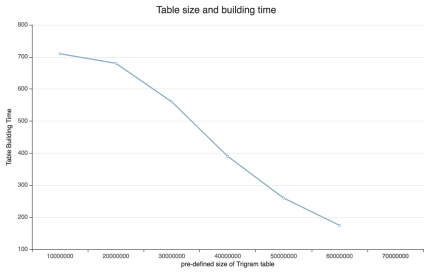

Figure 3: Model building time with Trigram table size

Of course, the loadfactor will also influence this process since it will decide when to enlarge the current table to make less conflicts. In my previous model, I start with a small n-gram table and use loadfactor to enlarge it. After I can estimate the final number of total bigram and trigram, I can fully allocated my 2GB space to them according to their size so that the model will build much faster.

3.3 discount factor

The discount factor will influence the BLEU score. The table below are the BLEU score un-

Discount factor	BLEU
0.55	20.539
0.6	20.542
0.65	20.548
0.7	20.552
0.75	20.563
0.8	20.586
0.85	20.641
0.9	20.642
0.95	20.636

Table 2: Decoding time in different cache size

Performance	Unigram	Trigram
Memory	124M	1.9G
BLEU	15.07	20.642
Decoding time	9.3s	436s
Model Building time	50s	206s
Total Time	60s	761s

Table 3: Comparison With Two Models

der the above settings and adjust discount factor from 0.55 to 0.95. The results are in the table 2.

3.4 Model comparison

The table 3 is the comparison between trigram and unigram model. The table selects the best running results of the Trigram model.

3.5 hashFunction

Given the fact that java hash function is very slow and I should implement a hash function by myself. I tried Long.hashcode() method and it performs very worse. I also tried the Knuth's Multiplicative Method but it performance is not as good as mod hash I currently used. I believe there exists a better hash function to perfectly meet the course required level.

4 Discussion

4.1 Error analysis

There is bug worth mentioning that cost me four days to move on. When I was doing the hash map, I wrote the following code:

int pos = (int) key % keys.length;

trying to convert the long to int value. However, the program is very slow and the result is totally wrong. It will firstly convert the long key into a int type which will lose information contained by high bits. The right conversion is:

int pos = (int) (key % keys.length);

It is also important to select a hash function and this has troubled me for the whole homework1. Maybe I will go to recitation the next time and get more information about those necessary steps. My BLEU score is below 23 and maybe I wrongly count the words fertility and I asked a lot of friends to find out potential problems but still didn't work. This is really struggling and hard to debug. I think I will start earlier in HW2.

4.2 Future Work

To improve the current mode, firstly, to find a good hash function so that I can save memory and run time. Secondly, get a right BLEU. Last but not least, try the ranking method taught in class.

Actually, there exists a modified Kneser-Ney smoothing method.Instead of using a single discount D for all nonzero counts as in Kneser-Ney smoothing, it uses three different parameters, D1, D2, and D3+, that are applied to n-grams with one, two, and three or more counts, respectively. The formula is shown below:

$$p_{\mathrm{KN}}(w_i|w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i}) - D(c(w_{i-n+1}^{i}))}{\sum_{w_i} c(w_{i-n+1}^{i})} + \gamma(w_{i-n+1}^{i-1}) p_{\mathrm{KN}}(w_i|w_{i-n+2}^{i-1})$$

where

$$D(c) = \begin{cases} 0 & \text{if } c = 0 \\ D_1 & \text{if } c = 1 \\ D_2 & \text{if } c = 2 \\ D_{3+} & \text{if } c \geq 3 \end{cases}$$

To make the distribution sum to 1, it takes

$$\gamma(w_{i-n+1}^{i-1}) = \frac{D_1 N_1(w_{i-n+1}^{i-1}\bullet) + D_2 N_2(w_{i-n+1}^{i-1}\bullet) + D_{3+} N_{3+}(w_{i-n+1}^{i-1}\bullet)}{\sum_{w_i} c(w_{i-n+1}^{i})}$$

This modification is motivated by evidence to be presented that the ideal average discount for n-grams with one or two counts is substantially different from the ideal average discount for n-grams with higher counts. There are papers and experiment showing that the modified version outperforms the regular version.

5 Conclusion

Using the bit pack trick mod hash method taught in class, combined with the cache mechanism can obviously improve the performance of the model. It shortens the running time from 4000s to around 800s. However, the model still cannot translate those unique marks in french precisely like ' symbol very well, as the translating results showed, which need to be improved in the future.

References

Ute Essen Hermann Ney and Reinhard Kneser. 1994. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech Language*, 8(1):1–38.