Memory-Efficient Modified Kneser-Ney Language Model Estimation

Kunlin Yang
Computer Sciense and Engineering Department
University of California, San Diego
k6yang@ucsd.edu

Abstract

In this project, I present and describe my implementations for the modified Kneser-Ney Language Model. My model could be generalized to N-gram model with any N. I focus on the accuracy and memory efficiency in this project. I would elaborate on the Kneser-Ney Language Model that I choose to get better accuracy and the data structure I design to take less memory while maintain time efficiency.

1 Disclaimer

The word count for lower orders, i.e. orders smaller than 3, is inaccurate because I use a different count method. But it is accurate for the highest order, i.e. 3-gram.

2 Modified Kneser-Ney Smoothing

The original Kneser-Ney Somoothing (KN) method (Kneser and Ney, 1995) use the raw count of appearance for N-gram and a fixed discount for Smoothing. Based on that, I refer to the modified version (ModKN) (Chen and Goodman, 1999) to use adapted discount and word counts. I would introduce the modified part here. For the intuition and prove of superiority, please refer to the original papers(Chen and Goodman, 1999) (Heafield et al., 2013).

2.1 Contex Based Word Counts

In KN, I set the count of N-gram to $c(w_n\cdot)$ to the times of appearance of the whole N-gram in the train dataset. For N-gram with $n = N$, I take this count method. However, for $n < N$, I count the number unique words before it, i.e. $|v : c(vw_{n-1})|$. The word v is called the context word and this count is called context based word counts.

The final adjusted counts used is:

$$a(w_n) = \begin{cases} c(w_n\cdot), \text{if } n = N \\ |v : c(vw_{n-1})|, \text{if } n < N \end{cases} \quad (1)$$

2.2 Adapted Discount

In KN, the discount is set to a fixed number D. Here I utilize the statistics of counting. For each length n, I collects the number $t_{n,k}$ of N-gram with adjusted count $k \in [1, 4]$.

I derive $t_{n,k}$ from:

$$t_{n,k} = |\{w_n : a(w_n) = k\}| \quad (2)$$

These are used to compute adapted discount $D_n(k)$.

$$Y = \frac{t_{n,1}}{t_{n,1} + 2t_{n,2}} \quad (3)$$

$$D_n(k) = k - \frac{(k + 1)t_{m,k+1}}{t_{n,k}} \quad (4)$$

for $k \in [1, 3]$. Other cases are $D_n(0) = 0$ and $D_n(k) = D_n(3)$ for $k \leq 3$.

3 Data structures

Here I design two innovative data structure. One is called BitPackVector, an alternative for *int[]* in Java. The ohter is NGramVector, the data structure for N-gram.

3.1 BitPackVector

This data structure is to save memory for *int[]* as long as the biggest number in this array use less than 32 bits. It is similar to *BitSet* regarding data representation. However, it supports auto bit increasing and trimming. For example, to store the index vector for uni-gram (less than 500,000), it only needs $500, 000 * 19/8 = 1, 187, 500$ bytes comparing to $500, 000 * 4 = 2, 000, 000$ bytes, saving around 41% of memory (it use 19 bits to represent each number here since the biggest number is less than $500, 000 \leq 2^{19} = 524, 288$).

3.1.1 Overview

I use *long[]* as for the internal storage of data. Each *long* is 64 bits. I use *UnitSize* as the parameter for the number of bits per element. This structure mainly supprts two functions, *Get(int index)* and *Set(int value, int index)* to get and set data.

3.1.2 *Get* and *Set* Implementation

The value to be stored is the lowest *UnitSize* bits of *int*. To get and set the data, the main challenge is to calculate the offset of bit in the storage array. I declare another variable *UnitMask*, which is of length *UnitSize* and each bit is set to 1.

For Get, if all the bits of the value is within the same *long*, simply move it to the lowest bit and use *UnitMask* to clear the higher bits. If the bits to get are in the two consecutive *long*, first move both of them and *UnitMask* to the lower bit and then concat the two parts. For Set, it uses the similar method to find the location to be inserted. But it first applies complement operator on *UnitMask* to reset the location and then uses bitwise or operator to set the value.

3.1.3 Memory Management

Before initiated, it accepts the desired *UnitSize* and capacity, or it would set it in the default value. It should be noted that does not allocate all the memory when initiated. It would allocate a small partition of memory at first, and when the index to be set is larger than the current storage, it would automatically grow. However, for Get operation, it would not trigger memory growing.

If the index to be set is larger than current capacity, it would simply double the storage array. If the value to be set is larger than $2^{UnitSize}$, it would increase 1 bit per element.

After building the language model, I trigger vector trim process, to remove the tailing blank storage, and fix the size of BitPackVector.

3.2 NGramVector

This data structure is to save the N-gram and its index. Given N-gram in the form of *int[]*, it would return the index of it. This index is used to query the count, probability and backoff. Each N-gram is represented by three int regardless the value of N (since it is stored in the form of BitPackVector, the actual bits needed are less than that of Java *int*). This structure mainly supprts two functions, *Get(int history, int word)* and *Set(int history, int word))* to get and set data.

3.2.1 Overview

Each NGramVector stores all the n-gram under the order n. It consist three BitPackVector, one for the last word of the n-gram, one for the index of $n-1$-gram as the history and one for the index of n-gram. For a n-gram *int[]* $= \{w_1 \ldots w_n\}$, it uses the first word w_1 to query 1-gram to get the index h_1. Then it uses w_1 and h_1 to query 2-gram to get the index h_2. Do this repeatedly until get the final N-gram. It should be noted that, the middle results are not useless, since they can be used to get backoff probability.

3.2.2 *Get* and *Set* Implementation

To get and set one index, it need to provide the history index and the current word. This two value would be hashed into one integer as the position for the index array. If it is a set operation and the position is being taken, it would increase one by one until the next available position is found. Once found, the history and word array are also set to the given value. If it is a get operation, it check the history and word array for the given position to see if it matches the given history and word. The position increases until both the history and word are matched.

3.2.3 Memory Management

The word and history array are managed by BitPackVector. For the index array, it is managed by NGramVector, because the size of it would affect the hash space. Every time it grows, it would do reindex to map the old index into the new one.

4 Pipeline

The estimation has four streaming stages: counting, adjusting counts, normalization and interpolation.

4.1 Counting

For counting, I use two vectorts of NGramVector to store N-gram and reverse N-gram for all orders, and one vector of BitPackVector to store the actual counts. The mapping information between them are stored in NGramVector index. The N-gram vector is to store the normal counting based on number of appearance. Only order of $n = N$ is counted here. The reverse N-gram vector is to store the information for context based word counts. Instead of storing the past $n-1$ words for each n, it stores the future $n-1$ words. All lower orders gram would be added to this vector.

4.2 Adjusting Count

This step is to calculate context based word counts. Instead of countingt the appearance, it counts the distinct words before N-gram for lower orders. As mentioned in NGramVector part, it store the history of the lower $n < N$ order and the nth word. For the reverse NGramVector, it stores the future $n < N$ words and the current word. To get the context based counting, I iterate over the history vector of the reverse one to count the appearance of each future $n < N$ words, which is the distinct count. This count are stored in the count vectors.

Then each order of the count vectors is traversed to get the $t_{n,k}$ value. Thus we could derive the adapted discount. The adapted is around 1 in this project.

After adjusting count, the reverse NGramVector is freed to save memory.

4.3 Normalization

In this part, the target is to calculate the two vectors. One is the pseudo probability u:

$$u(w_n|w_1^{n-1}) = \frac{a(w_1^n) - D_n(a(w_1^n))}{\sum_x a(w_1^{n-1}x)} \qquad (5)$$

where w_i stands for the ith word in N-gram and w_i^n stands for the consecutive n words starting from w_i. The other is backoff b:

$$b(w_1^{n-1}) = \frac{\sum_{i=1}^{3} D_n(i)|\{x : a(w_1^{n-1}x) = i\}|}{\sum_x a(w_1^{n-1}x)} \qquad (6)$$

Based on the design of NGramVector, this two vectors is easily to count. Simply iterate over the NGramVector twice, once to get the sum of counts for the denominator and twice to get the u and b.

4.4 Interpolation

The itnerpolation stage computes the final probability for N-gram based on the recursive equation:

$$p(w_n|w_1^{n-1}) = u(w_n|w_1^n-1) + b(w_1^{b-1})p(w_n|w_2^{n-1}) \qquad (7)$$

The recursion terminates when unigrams are interpolated with the uniform distribution:

$$p(w_n) = u(w_n) + b(\epsilon)\frac{1}{|Vocabulary|} \qquad (8)$$

where ϵ denotes the empty string. The unknown word counts is zero, so the probability of it is also set to $b(\epsilon)/|Vocabulary|$.

In my implementation, this process are intergated with the normalization process, which is calculated in the second iteration.

5 Experiments

In my implementation, I implemented two version, one is to calculate N-gram probability on the fly, with the ability to handle unknown words, it could get a score of **24.802**. But this one is incredibly slow, which costs 2 hours to finish all the test cases. The other one is the one I submit. It pre-calculate all the information for known N-gram. Once there is a unknown word in the N-gram, it would return a default value regardless where the unknown word is. This one is faster and get a lower score of **22.972**

6 Conclusion and Future Work

I have introduced my approach in this paper. However, because of time limit, I could not add more experimetns and comparison. Due to the original limitation of memory, I focus on the memory optimization in this project. This cause the relative low speed of testing. Although I pre-calculate all the information needed, the access of them has to go through hash and collision check, which is very time-consuming. The future work would focus on the trade off between speed and memory efficiency.

References

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

Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H Clark, and Philipp Koehn. 2013. Scalable modified kneser-ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 690–696.

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 181–184. IEEE.