Implementation of Trigram Kneser-Ney Language Modeling

Qing Wei
A53309131
University of California San Diego
Computer Science and Engineering
qiwei@eng.ucsd.edu

Abstract

In this project, a Trigram Kneser-Ney Language Model was implemented. Trained on 9073252 English sentences and tested using BLEU score combining with a standard machine translation decoder, the model reaches 24.956 BLEU Score with a memory usage of 1.1 GB.

1 Language Model

1.1 N-Gram

A language model is a distribution over sequences of words in a sentence. It builds a model based on a training corpus, and based on the probability calculated with the model, predict the next word.

$$\begin{aligned} P(w) &= P(w_1 \cdots w_n) \\ &= \prod_i P(w_i | w_1 \cdots w_{i-1}) \\ w^* &= \arg\max_w P(w|\ \text{prev}_{n-1}) \end{aligned}$$

If the conditional probability table of the language model is pre-computed on a given train set, then the time complexity to predict the probability $P(w)$ for a sentence of length n is $O(n^2)$. In order to simplify the model and reduce the cost, the N-Gram language model make an assumption that only considering the language model as a Variable-order Markov chain where the probability of each event depends only on the state attained in the previous event. For k-order Markov model, we could get the following equation, and the time cost would be $O(nk)$.

$$\begin{aligned} P(w) &= \prod_i P(w_i | w_1 \cdots w_{i-1}) \\ &= \prod_i P(w_i | w_{i-k+1} \cdots w_{i-1}) \end{aligned}$$

1.1.1 Probability Estimation

In N-Grams Model, to compute the probability $P(w_i|w_{i-k+1}$, we still have to acquire the conditional probability table first. We use maximum-likelihood estimation to learn the probabilities on train set.

$$\begin{aligned} \mathcal{L} &= \log P(w) \\ &= \log \prod_i P(w_i | w_{i-k+1} \cdots w_{i-1}) \\ &= \sum_i \log P(w_i | w_{i-k+1} \cdots w_{i-1}) \\ &= \sum_i \sum_w \sum_{\text{prev}_{k-1}} c(\text{prev}_{k-1}, w) \log P(w|\text{prev}_{k-1}) \end{aligned}$$

Note that the $c(x)$ used above denotes the number of x appearing in the train data set.

Let C_α denotes count(prev$_{k-1}, \alpha$), P_α denotes $P(\alpha|\text{prev}_{k-1})$, then the problem becomes:

$$\begin{aligned} \textbf{maximize} \quad & \sum_\alpha C_\alpha \log P_\alpha \\ \textbf{such that} \quad & P_\alpha \geq 0 \\ & \sum_\alpha P_\alpha = 1 \end{aligned}$$

The solution to this problem is:

$$\begin{aligned} P_k(w|\text{prev}_{k-1}) &= \frac{C_\alpha}{\sum_\beta C_\beta} \\ &= \frac{c(\text{prev}_{k-1}, w)}{\sum_v c(\text{prev}_{k-1}, v)} \\ &= \frac{c(\text{prev}_{k-1}, w}{c(\text{prev}_{k-1})} \end{aligned}$$

1.2 Kneser-Ney Smoothing

In the n-gram model, whenever the model encounter a unseen word or n-gram in the test set,

it outputs a probability of 0. However, because of the sparsity of the input corpus, new words or old words in new contexts happen all the time. In addition, Based on an observation that N-grams occur more in training data than they will later (Church and Gale, 1991), discounting is often applied by reducing the counts by a constant d.

Kneser-Ney smoothing (Ney et al., 1994) is therefore commonly used to calculate the probability distribution of n-grams adjusted by discounting and their context fertility.

Define a new function c':

$$c'(x) = \begin{cases} c(x) & \text{For the highest order,} \\ |u : c(u, x) > 0| & \text{Otherwise} \end{cases}$$

Then the probability of the k-th order of a n-grams model is:

$$P_k(w|\text{prev}_{k-1}) = \frac{c'(\text{prev}_{k-1}, w}{\sum_v c'(\text{prev}_{k-1}, v)} + \alpha(\text{prev}_{k-1}) P_{k-1}(w|\text{prev}_{k-2})$$

d in the aboe equation is a constant discount, and $\alpha(\text{prev}_{k-1})$ is a normalization factor to ensure $\sum_w P_k(w|\text{prev}_{k-1}) = 1$. Its value relies on prev_{k-1}.

$$\begin{aligned} \text{LHS} &= \frac{\sum_v \max(c'(\text{prev}_{k-1}, v) - d, 0)}{\sum_v c'(\text{prev}_{k-1}, v)} + \alpha(\text{prev}_{k-1}) \\ \text{RHS} &= 1 \\ \alpha(\text{prev}_{k-1}) &= \frac{\sum_{v, c'(\text{prev}_{k-1}, v) > d} d}{\sum_v c'(\text{prev}_{k-1}, v)} \\ &= \frac{|v : c'(\text{prev}_{k-1}, v) > d| \times d}{\sum_v c'(\text{prev}_{k-1}, v)} \end{aligned}$$

2 Implementation

In this project, a Trigram Kneser-Ney Language Model is implemented with Java. It trains on a large amount of English sentences and is able to return any $c(w)$ and $\log P(w|prev_{k-1})$ where $k = 1, 2, 3$.

2.1 Data Structure

To calculate $P_k(w|\text{prev}_{k-1})$ in approximately $O(1)$ time, we have to store the parameters learn from the training corpus, including $c'(x)$ for all unigrams, bigrams and trigrams, as well as the number of v such that $c'(\text{prev}_{k-1}, v) > d$ for $k = 1, 2, 3$ to compute α.

To compute $c(x)$, whenever encounter a n-gram, we increase the frequency of this n-gram by 1. To compute $c'(x)$ for unigram or bigrams, the program keeps a set for each n-gram, and store the word preceding it. Whenever encountering a unseen preceding word, add the fertility count for the n-gram by 1. To compute $\sum_v c'(\text{prev}_{k-1}, v)$, it is obviously equal to the bigram count of prev_{k-1} when $k = 3$. To track this value for $k = 1$ or 2, for every word, maintain a set and store every combination of the word before and after it, then add 1 to its *context* count by 1 when a new combination is added to the set. For $|\{v : c'(\text{prev}_{k-1}, v) > d\}|$, if we choose d such that $0 < d < 1$, it would be quite easy. It is basically the number of the word appearing after this prev_{k-1}.

In order to reduce the memory usage and speedup the map lookup process, we assigned unique ranks for all word combinations and store the corresponding data. A lookup map is used to record the mapping between n-grams and their corresponding ranks.

2.1.1 N-grams Representation

For any n-grams, we used its rank stored in the rank lookup table as its representation. However, strings or list of strings are too much memory consuming for the language model where the number of n-grams is tens of millions. Therefore, only for unigrams, we maintain a lookup table where the key in the table is **string**, and for bigrams and trigrams, we use bit operations on their elements to get a key in **long** type.

Statistics show that there are 495172 unique words in the corpus. $\log_2 495172 = 18.9$, indicating that we can store a unigram with only 20 bits. Therefore, a bigram consisting w_1 and w_2 can be represented as $\text{key}(w_1, w_2) = (\text{rank}(w_1) << 2) \mid \text{rank}(w_2)$. Similarly, a trigram (prev_2, w) can be represented as $\text{key}(\text{prev}_2, w) = (\text{key}(\text{prev}_2) << 20) \mid \text{rank}(w)$. Both representations can be stored with a 64-bit **long** type value.

2.2 Rank Lookup

The time complexity to look up a key in a table is $O(n)$ if only keeping a unordered list. One way to speedup is whenever insert a new key, sort the table by the key and lookup each key with binary search. The time complexity for model build is $O(n \log n)$

and $O(\log n)$ for each searching process.

Another common approach is to maintain a hash table by storing key at the index hash(x). For good hash function with fewer collisions, the best time complexity for a single insertion and search will be $O(1)$.

2.2.1 Hash Function

A good hash function is essential for the speed and memory consumption of hash tables. It should ideally distribute all the input data across the entire set of possible hash values uniformly and is easy to compute.

The key of the hash value will be in **long** type in Java. A common hash function for **long** types is to compute the exclusive OR of their upper half (32 bits) and lower half (32 bits). Based on this, the hash function I used is:

$$\text{hash(key)} = (\text{key} \wedge (\text{key} >>> 32)) * 3875239$$

2.2.2 Open Hash Address

Hash functions may result in collisions where the function generates the same index for more than one key. Whenever a collision occurs in a hash map, there are two ways to solve the problem: separate chaining and open addressing.

In separate chaining approach, every entry in a hash table is actually a linked list containing all the keys that are hashed to the exact same address. In contrary, in open hash address algorithm, when the hash-to entry is already occupied, the program continues to search in the hash map for the next blank space to insert the key.

The drawback of separate chaining is obvious: too much space will be used to store pointers for the linked list. And the open addressing strategy requires the number of keys not exceeding the size of the map. In other word, the load factor $= \frac{\text{\# of keys}}{\text{\# of entries}} < 1$. For different load factor, the performance varies significantly.

2.2.3 Speedup by Reducing Collisions

It is almost impossible to find a perfect hash function. Any hash function may result in hash collisions which will sometimes largely slow down the lookup process.

To reduce the chance of collisions, we might decrease the load factor of our open hash address algorithm. With respect to the number of unique keys n, we set the size of the lookup hash table to $\frac{n}{\text{load factor}}$.

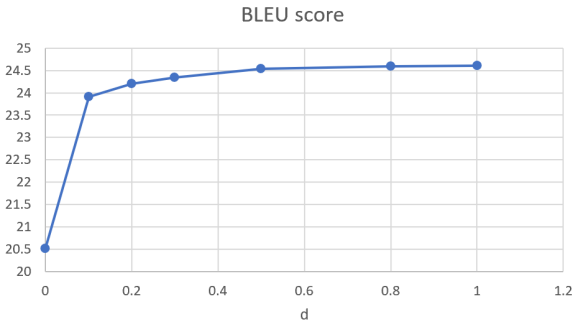

Figure 1: The BLEU scores over different choices of d.

3 Experiment

3.1 Corpus

The corpus used in this project for training language model contains 9073252 sentences in total. Statistics shows that the corpus contains 495172 unique words, 8374230 bigrams, and 41701111 trigrams.

3.2 Metrics

Each language model is tested by providing its scores (log-probability of n-grams) to a standard machine translation decoder, which then takes a French sentence and returns the highest-scoring English sentence, and measuring the quality of the resulting translations by BLEU. BLEU (bilingual evaluation understudy) is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another (Papineni et al., 2002). The BLEU score of a unigram model is 15.535, whereas the trigram Kneser-Ney model implemented in this project reaches 24.956.

3.3 Impact of Choice of Discounting Factor

Smoothing is proved efficient in n-grams language models. Fig.1 shows the impact of the choice of discounting factor d[1]. For the given corpus and the chosen values of $d \in [0, 1]$, the BLEU score increases with d. The trigram model is significantly improved by the Kneser-Ney Smoothing.

[1]To simplify the test and save the time, we only used 3000,000 sentences to train the model to show the result instead of the full corpus. For the whole corpus, I used 0.8 as discounting factor

3.4 Results

The performance, including memory usage, model build time and decoding time, BLEU score, of the implemented trigram Kneser-Ney model, is compared with the unigram and a trigram model without any smoothing (See Table 1). The BLEU score improves largely, and the memory usage does not exceed 1.3 GB. A load factor of approximately 0.8 is used, and the discounting factor d is set to 0.8.

The language model is also used to test the impact of the size of training set. The train corpus consists 9073252 sentences in total. Fig 2 illustrates the change of the number of unique unigrams, bigrams, trigrams and the BLEU scores over different size of training data. We could see that as the size of the training set grows larger, the number of unigrams, bigrams and trigrams increases largely as well, indicating that more and more unseen word combinations appears in the later sentences. However, although the number of n-grams increases fast, the BLEU score improves only slightly after 5000,000 sentences.

4 Conclusion

This project successfully built a trigram language model with Kneser-Ney smoothing. The performance is quite satisfying.

- 1. The memory usage printed before decoding is 1.1 G.
- 2. Decoding time is 234.296s on an Intel i7-8650U Processor[2], which is around 20x slower than when decoding with the unigram model.
- 3. It takes about 20 minutes to build the language model and decode the test set on the above stated processor.
- BLEU score is 24.956.

Experiments were also conducted to verify the performance of model on different training data size and discounting factor.

References

Kenneth W Church and William A Gale. 1991. A comparison of the enhanced good-turing and deleted estimation methods for estimating probabilities of english bigrams. *Computer Speech & Language*, 5(1):19–54.

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

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

[2]Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz, 2112 Mhz, 4 Core(s), 8 Logical Processor(s)

Language Model	Memory Usage	Total Time	Decoding Time	BLEU Score
Trigram Kneser-Ney (d=0.8)	1.1 GB	20 min	234.296 s	24.956
Trigram w/o Discounting	1.1 GB	9 min	210.921 s	22.286
Unigram	139 MB	1.5 min	10.470 s	15.535
STUB	137 MB	50 s	41.784 s	15.803

Table 1: Performance Comparison between Different Language Models

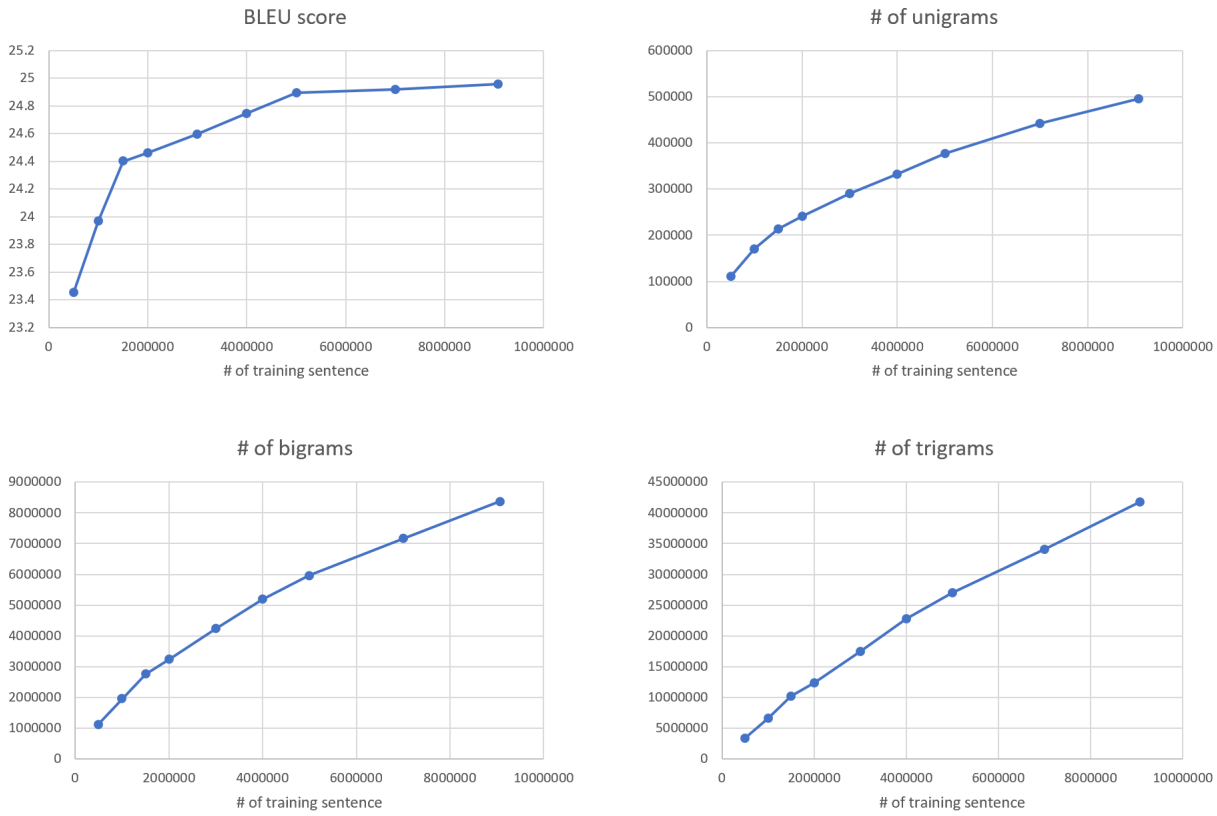

Figure 2: The number of unique unigrams, bigrams, trigrams and the BLEU scores over different size of training data.