Kneser-Ney Trigram Language Model

Xinyi He
xih108@ucsd.edu

Abstract

In this project we implement the Kneser-Ney trigram language model and the measure its performance in BLEU score, memory usage and speed.

1 Introduction

A language model is a probability distribution over sequences of words which is a widely used concept in Natural Language Processing. We'll explore into the N-gram language models, especially the trigram with Kneser-Ney Smoothing.

2 Related Work

Firstly, we use $P(w_1...w_n)$ to represent a language model. N-gram models are built from local conditional probabilities

$$P(w_1...w_n) = \prod_i P(w_i|w_{i-k}...w_{i-1})$$

Then the probability of the sentence under trigram language model $(k = 3)$ is

$$P(w_1...w_n) = \prod_i P(w_i|w_{i-2}w_{i-1})$$

where we define $w_0 = w_1 =$<s>, $w_n =$<\s>, which are start and end tokens that we add to the sentence. For example, for the sentence "the car runs", we have $P(\text{the car runs}) = P(\text{the}|\text{<s><s>})P(\text{car}|\text{<s>the})P(\text{runs}|\text{the car})$ $P(\text{<\\s>}|\text{car runs})$.
For Kneser-Ney smoothing, it uses the concept of absolute-discounting interpolation and the idea that word probabilities are proportional to context fertility in the backoff model.

$$P(w) \propto |w' : c(w', w) > 0|$$

Based on these, we can construct the formula for Kneser-Ney N-gram:

$$P_k(w_i|w_{i-n+1}^{i-1}) = \frac{\max(c'(w_{i-n+1}^{i}) - d, 0)}{\sum_v c'(w_{i-n+1}^{i-1}v)}$$

$$+\alpha(w_{i-n+1}^{i-1})P_{k-1}(w_i|w_{i-n+2}^{i-1})$$

where d is a discounting factor, α is computed to normalize the probability and

$$c'(\cdot) = \begin{cases} \text{count}(\cdot) & \text{for the highest order} \\ \text{context fertility}(\cdot) & \text{for lower orders} \end{cases}$$

For Kneser-Ney trigram model, we conventionally choose $d = 0.75$ as a discouting factor for the trigram and bigram.

3 Implementation Process

The naive approach to build the KN-smoothing model would be using 3 Java built-in hashmaps. Each takes in (String,Integer) as (key,value) pair for value trigram count, bigram and unigram context fertility. However, such data structure would easily lead to run out of memory error and super long build time. Under such condition, we try to optimize both memory usage and speed in the following ways.

3.1 Bit packing

As we know String is object in Java which takes extra memory so we want to store string in a more efficient way. Since for each word we have a unique corresponding integer, then we can make use of this to encode each word as an integer. Based on the fact that there should be no more than $2^{20} = 1,048,576$ words in the vocabulary, it is sufficient to use 20 bits for each word encoding. Then for strings consisted of more than one word, we can represent them by packing integers together using bit operation.In this way, for a trigram, we pack three 20 bits integer together,

which in total needs 60 bits and perfectly fits in a primitive 64-bit long. We do the similar operation for bigram which needs 40-bit and also fits into primitive long. For unigram, as there is only one word, we just need the encoding integer itself which is 32-bit.
After bit packing, we can see that both memory usage and speed improved a lot since using long as key is more storage efficient and time efficient.

3.2 Open Address Hashmap

For a hashmap, collision is an inevitable problem and resolving collisions takes extra time and space . Under such condition, a good hash function which generates an acceptable number of collisions is preferable, so we use the hash function

$$hash(key) = (key^\wedge(key >>> 32)) * 3875239)$$

In addition, Java Hashmap is closed address hashing which uses separate chaining to resolve collisions. It takes much more space to store the linked list for hash buckets. In order to improve the memory usage, we switch to open address mapping with linear probing. In open addressing hashmap, every elements are stored in the hash table and we would keep probing until an empty slot is found to store. In this way, we can get rid of the chaining and use the hash table more efficiently in terms of memory.
Moreover, open addressing is more sensitive to the load factor, so we want to set an appropriate initial capacity and load factor. During our experiment, we found that the number of trigrams would be around $5 * 10^7$, however, the number of bigrams and unigrams would be much less. If we set a very large capacity for the hashmap at the beginning, then it is a waste of memory so that we use a not so large initial capacity. Also we choose load factor of 0.8 after multiple trials, since a smaller load factor would waste memory and a larger one would lead to a slow-down.

3.3 Speed-up tricks

In order to decrease the build time, we try to construct the hashmaps in a more efficient way. Instead of going through the large training data again and again, we can actually build the bigram fertility based on trigram token count. Since the key of trigram token count is unique, then the last two words of each key can be used for the bigram fertility. Similar strategy is also utilized for the unigram fertility. In this way, we can save a large amount of time for the model building.
In addition, we try to fasten the decode process as to meet the time requirement. We found that if we compute everything needed in the formula during the decoding process then it should take a very long time because we need to traverse the n-gram hashmap every time. However, if we precompute the normalization constants $\sum_v c'(w_{i-n+1}^{i-1}v)$ and α during the model building, then it would save a lot of duplicate work so that all we need during encoding is to retrieve and do simple arithmetic operations. We also make use of shared index table to store the bigram stuffs so that we can save memory of storing duplicate keys.

4 Performance Evaluation

After building the Kneser-Ney trigram language model, we measured its memory usage and decoding speed. Also, we evaluate it over test data which is a and get a BLEU score. We experimented multiple times and report the average below.

	Memory Usage	Decode Time	BLEU Score
Trigram	286s	995M	24.979
Unigram	8s	138M	15.535

Table 1: Performance of language models.

Table 1 shows that the Kneser-Ney trigram model have a much higher BLEU score than the unigram model which means it is closer to the natural language. However, it has a larger memory and time consumption due to its complexity.

5 Investigation and Analysis

Based on the Kneser-Ney trigram language model that we've already implemented, we tried to explore the model perplexity more in depth because it is a common method to measure the quality of a language model.
For a test set X which consists of sentences

$$x^{(i)} = (x_1^{(i)}, ..., x_{L_i}^{(i)})$$

we have $|X|$ be the total number of words in X, and $\log P(X)$ be the sum of log probability of each sentence, then perplexity is defined as

$$2^{-\frac{\log_2 P(X)}{|X|}}$$

Perplexity of a model p is actually the reciprocal average probability assigned by the model to each

word in the test set X. Thus, we expect a lower perplexity for a better model as it indicts a higher log likelihood.
We experimented for different training size of our model and evaluate the perplexity over the test set (same as the one used for reporting BLEU score).

Training Size	Perplexity
10000	457.14
100000	319.74
500000	162.11
1000000	123.76
5000000	76.43
9073252	73.07

Table 2: Perplexity on the test set for different training size.

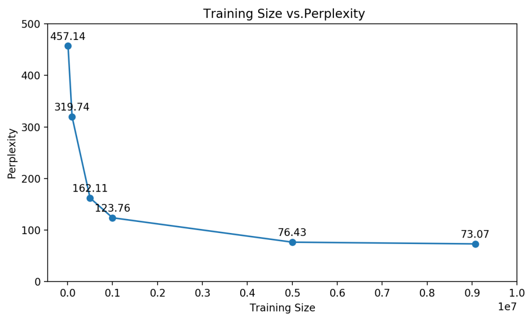

Figure 1: Training Size v.s Perplexity on test data

Table 2 and Figure 1 demonstrate that the perplexity of the test set decreases as the training data size increases which indicts that the language model gets better.
For a language model that is built on a small training data size, it is not generalized enough as the vocabulary size is small and the pattern of word occurrence may not follow the natural language so it has a bad performance on predicting unseen data. As the training size gets larger, the model becomes more generalized as the vocabulary size expands so it performs better on predicting samples. However, for very large training data sizes, the decrease of perplexity may be not that obvious since both models have approximately the same vocabulary size and generalized pattern of words. So they should both work better in predicting and have almost the same low perplexity.

6 Conclusion and Future Work

We have built the Kneser-Ney Trigram laguage model successfully with all the specified requirements satisfied. For future work, we may try to further optimize our term in terms of memory and time usage. For example, we can try ranking table and fast caching as mentioned in lecture.