Kneser-Ney Ngram Language Model For Translation

Nitesh Gundavarapu,
A53317353,
University of California, San Diego
nbgundav@ucsd.edu

Abstract

In this project, different technical and mathematical aspects of Kneser-Ney Ngram Language Model are analyzed in the context of French-English translation task. First, a standard implementation is provided that meets benchmarks (Memory 950MB, Execution Speed 25x Unigram, BLEU 24.936). Next, a joint hashing mechanism is introduced to reduce the memory footprint **from 1GB to 770MB.**. Further, caching and its effect on decoding time is studied. Finally, different ideas unified by Kneyser-Ney are empirically validated and then a grid-search on discounting is used to improve the BLEU score to **24.957**.

1 Introduction

Kneyser-Ney language model combines different ideas such as a) interpolation of language models of different orders will improve generalization while keeping sparsity under control, b) ngram counts need to be discounted in test scenarios to correspond with empirical observations, c) context fertility can be used to assign probabilities to unknown contexts etc. into a single unified language model. Mathematically, Kneser-Ney conditional probability equation takes the recursive form

$$P_k(w|prev_{k-1}) = \frac{max(c'(prev_{k-1}, w) - d, 0)}{\sum_v c'(prev_{k-1}, v)}$$

$$+\alpha(prev_{k-1})P_k(w|prev_{k-2})$$

where k is the order of the Ngram, c' is the empirical count for highest order Ngram and context fertility for other values of k, d is the discounting, α is used to normalize the probabilities post-discounting.

$$c'(x) = |\{u : c(u, x) > 0\}|$$

In particular for a Trigram Language Model, the equations turn out to be,

$$P_3(w_3|w_1, w_2) = \frac{max(c(w_1, w_2, w_3) - d, 0)}{\sum_v c(w_1, w_2, v)}$$

$$+\alpha(w_1, w_2)P_2(w_3|w_2)$$

$$. \tag{1}$$

After some elementary algebra, the equation for α turns out to be,

$$\alpha(w_1, w_2) = d\frac{|\{v : c(w_1, w_2, v) > d\}|}{\sum_v c(w_1, w_2, v)}$$

This the backoff bigram probability is calculated as

(2)

$$P_2(w_3|w_2) = \frac{max(c'(w_2, w_3) - d, 0)}{\sum_v c'(w_2, v)}$$

$$+\alpha(w_2)P_1(w_3)$$

(3)

where α becomes

$$\alpha(w_2) = d\frac{|\{v : c'(w_2, v) > d\}|}{\sum_v c(w_2, v)}$$

(4)

and finally the unigram backoff model is just the ratio of unigram fertility count to the total unigram fertility count.

$$P_1(w_3) = \frac{c'(w_3)}{\sum_v c'(v)}$$

(5)

2 Implementation Details

Upon inspection of the dataset, the total number of unique unigrams are found to be 495,172, unqiue bigrams to be 8,374,230 and unique trigrams to be 19,880,264. We need to keep track of various counts and fertilities for each of these ngrams. Strict memory requirements suggest that we cannot use default Java HashMaps to store the various counts required in Equations 1-5.

2.1 OpenAddressing With Linear Probing:

Open addressing scheme hashes each key to a permutation of the hashtable and the key is placed in the nearest empty location in the order of permutation. Due to the size of the hash table that we want to store, it's difficult to maintain random permutations for each input function. So, linear probing is one way to balance speed and randomization, if sufficient load factor is used. Keeping this in mind, an open addressing table is implemented with the keys being the primitive type *long* and values being primitive type *int*

2.2 Bit Packing:

The strings in each sentence are mapped to integer values. So an ngram is composed of multiple integers. Hence a naive trigram hashmap will require 3 bytes for the key and their wrapper objects would contain additional bytes. However we notice that the number of unique words is just 500k and hence we can map each word to 20 bits and the whole trigram to 64 bits.

$$bit\_pack(i_1, i_2, i_3) = (i_1 << 40)+(i_2 << 20)+i_3$$

2.3 Memory vs Decoding speed:

$c(w_1, w_2, w_3)$, $c(w_1, w_2)$, $|\{v : c(w_1, w_2, v) > d\}|$, $c'(w_2, w_3)$, $|\{c : c'(w_2, v) > d\}|$, $|\sum_v(w_2, v)|c'(w_3)$ are the terms required in the recursive third order Kneser-Ney expansion. If we pre-compute these terms, decoding speed would decrease while memory foot-print increases and vice-versa. However, because of above mentioned storage optimizations, we can choose to pre-compute all these terms and improve the decoding speed.

2.4 Preliminary Results:

The above implementation yields a BLEU score on the test dataset as 24.936, takes up 950MB and runs in 252s on an i7 16Gi laptop which is 25x times the execution time of an empirical unigram model.

Model Order-Backoff	BLEU
3-2	24.519
2-1	20.158
3	22.375
2	20.171
1	15.535
3-2-1	**24.957**

Table 1: Analysis of Kneyser Ney for different orders and different back-offs. x-y-z-w indicates x is the order of the model and it backs off to y, z and w orders successively. BLEU score is reported in each setting.

3 Joint Hashing Mechanism:

From Section.2, it is to be noted that the terms $c(w_1, w_2)$,$|\{v : c(w_1, w_2, v) > d\}|$,$c'(w_2, w_3)$ are all functions of bigrams and there are 8M unique bigrams. Similarly, the terms $|\{c : c'(w_2, v) > d\}|$, $|\sum_v(w_2, v)|c'(w_3)$, $c(w_3)$ are all functions of unigrams. Since each key of unigram is chosen to be a 64bit primite *long*, we can greatly benefit by combining all the open-address hashmaps corresponding to a particular n-gram. This is achieved by a joint open-addressed hash map implemented with an underlying single *long[]* array as key and multiple *int[]* arrays as values. This improves the memory footprint to **770MB**. By presetting the cache sizes to the respective values/0.7, the memory fits in **650MB**.

4 Analysis of Kneser-Ney:

4.1 Interpolation of NGram Models:

A higher order model is sparse while a lower order model wouldn't generalize well. Kneser-Ney model contains both higher order and lower order ngram terms through the recursive formulation. In this section,the implementation above is modified to create Ngram models of different orders with different back-off probabilities.

It can be seen in Tab.1 that the best setting is 3-2-1 where a third order model backs off to second order and then to first order. A fourth order model turns out to be computationally very expensive.

4.2 Discounting

It is empirically observed that the test set follows a distribution where the frequencies of words are

Trigram Discount	Bigram Discount	BLEU
0.95	0.75	24.954
0.85	0.75	**24.957**
0.75	0.75	24.936
0.65	0.75	24.905
0.95	0.65	24.943
0.85	0.65	24.954
0.75	0.65	24.932
0.65	0.65	24.887

Table 2: Analysis of Kneyser Ney for different discount values. BLEU score is reported in each setting.

Model	BLEU
Fertility Backoff	**24.957**
Empirical Backoff	24.827

Table 3: Analysis of Kneser Ney with fertility and empirical backoff. BLEU score is reported in each setting.

slightly discounted from their frequencies in training set. Kneyser Ney takes advantage of this to discount the words and apply the discounted weight to the back-off model for unseen contexts. This is very elegant. The BLEU scores for different discountng values are reported in Tab.2

4.3 Fertility:

Kneyser Ney uses fertility counts to back-off instead of the empirical counts. This allows for unknown contexts to be given a probability reasoned based on the context fertility. To study its impact, models with fertility and empirical probability are studied in Tab.3

It is observed that the model with fertility back-off performs much better than empirical back-off.

5 Caching:

An important element in improving the decoding speed is caching. According to Zipf Law, the frequency of word with which a word appears is inversely proportional to rank. In other words, many ngrams in language are often repeatedly queried. This offers an advantage as to utilize caching to improve decoding speeds. The cache can be placed between hash table and the fetch operation. In this regard, I tried the following two approaches:

5.1 Most Frequently Used Cache (Dynamic):

This is implemented using TreeMap and HashMap datastructures in Java to only store the most frequently used 5000 keys. The map is updated dynamically updated during decoding time whenever a key is queried.

5.2 Most Frequently Used Cache (Static):

This uses the same implementation as the dynamic cache but the cache elements are never updated at decoding. They are simply populated with most frequently used keys in training set.

It is observed that the dynamic cache slows down the decoding speed because of new inserts into the cache during decoding time while the static cache doesn't suffer such drawback. However, since the elements are static, a lot of false negatives ensue and the performance is similar to without using a cache. A future direction to explore is to use Tries to catch negatives first followed by a cache to catch the positives.

6 Conclusion:

In this report, first a standard implementation of Kneser Ney approach is provided. This is followed by a discussion on implementation details and a joint hashing approach is proposed to greatly reduce the memory foot-print. Finally, three different key ideas in the Kneser-Ney approach have been empirically justified.