11-711 Project 1

Hengruo Zhang
Electrical & Computer Engineering
Carnegie Mellon University
5000 Forbes Avenue
hengruoz@andrew.cmu.edu

1 Background

Denote $c(x)$ the count of n-gram x in the corpus, $N_{1+}(\bullet w) = |\{u : c(u, w) > 0\}|$, $N_{1+}(w\bullet) = |\{u : c(w, u) > 0\}|$, $N_{1+}(\bullet w \bullet) = |\{(u, v) : c(u, w, v) > 0\}|$, and $N_{1+}(\bullet\bullet)$ the number of all unique bigrams.

Therefore, we got the following probabilities:

$$\begin{aligned}
P(w_3) &= \frac{N_{1+}(\bullet w_3)}{N_{1+}(\bullet\bullet)} \\
P(w_3|w_2) &= \frac{\max(N_{1+}(\bullet w_2 w_3) - d, 0)}{\sum_{v \in V} N_{1+}(\bullet v)} \\
&\quad + \alpha(w_2) P(w_3) \\
P(w_3|w_2 w_1) &= \frac{\max(c(w_1 w_2 w_3) - d, 0)}{c(w_1 w_2)} \\
&\quad + \alpha(w_1 w_2) P(w_3|w_2)
\end{aligned}$$

where

$$\begin{aligned}
\alpha(w_1 w_2) &= d \cdot \frac{N_{1+}(w_1 w_2 \bullet)}{c(w_1 w_2)} \\
\alpha(w_2) &= d \cdot \frac{N_{1+}(w_2 \bullet)}{N_{1+}(\bullet w_2 \bullet)}
\end{aligned}$$

2 Implementation Choice

The most frequent word "the" occurs 19880264 in the dataset, which is less than 2^{31}, so we can use an integer to store the count of a n-gram. We also count the number of unique n-grams in advance and the result is presented in Table 2 Since we

Unigram	**Bigram**	**Trigram**
495172	8374230	41627672

Table 1: Sizes of n-grams

need to compute $N_{1+}(\bullet w)$, $N_{1+}(w\bullet)$, $N_{1+}(\bullet w \bullet)$, $N_{1+}(w_1 w_2 \bullet)$, and $N_{1+}(\bullet w_2 w_3)$ beside the count of n-grams, there are $495172 * 4 + 8374230 * 3 + 41627672 * 1 = 68731050$ entries. If we use 8-byte key and 4-byte value with 0.85 load factor for hash tables, it will take about 1.0G space less than 1.3G. The data structure details are listed in Table 2.

	Data Structure
Unigram Counter	int[]
$N_{1+}(\bullet w \bullet)$	int[]
$N_{1+}(\bullet w)$	int[]
$N_{1+}(w \bullet)$	int[]
Bigram Counter	Hash Table
$N_{1+}(w_1 w_2 \bullet)$	Hash Table
$N_{1+}(\bullet w_2 w_3)$	Hash Table
Trigram Counter	Hash Table

Table 2: Data structures

After careful experiments, we select 0.85 as load factor, 0.75 as reducing count, $1.0e - 5$ as default probability (see 3.4), and hash function $h(x) = (x \text{ XOR } (x >>> OFFSET)) \,\%\, len$. In Section 3, we will introduce the details of experiments.

3 Performance

In this section, we compare some factors in control experiments. The default factors not tested are given as Section 2 goes.

3.1 Reducing Count

Assume $\alpha(w_1 w_2) = d_2 \cdot \frac{N_{1+}(w_1 w_2 \bullet)}{c(w_1 w_2)}$ and $\alpha(w_2) = d_1 \cdot \frac{N_{1+}(w_2 \bullet)}{N_{1+}(\bullet w_2 \bullet)}$. We tested different $d1$ and $d2$. By the results in Table 3, reducing count won't affect the performance much.

D1	D2	BLEU
0.0	0.0	24.422
0.1	0.1	24.444
0.25	0.25	24.482
0.75	0.75	24.663
1.0	1.0	24.621
0.75	1.0	24.651
0.25	1.0	24.646
0.75	0.25	24.577

Table 3: Reducing count d1 and d2

3.2 Hash Function

We compare these hash functions:

$$h_0(x) = x * 1145141919 \,\%\, len$$
$$h_1(x) = x \,\%\, len$$
$$h_2(x) = (x \text{ XOR } (x >>> OFFSET)) \,\%\, len$$
$$h_3(x) = (x \text{ XOR } (x >>> OFFSET)) * 37 \,\%\, len$$

where $OFFSET$ is half of bitsize of n-gram keys. For trigrams, $OFFSET = 19 * 3/2 = 28$, and for bigrams, $OFFSET = 19 * 2/2 = 19$ (The bitsize of unigram is 19). As Table 4, h_2 is the best hash function so select this one. We adopt square probing to prevent collision, i.e., when i-th collision, we add $(7i)^2$ to the hash value to get a new hash.

$h(x)$	Decoding time	Avg. collision #
h_0	1609.825	14.4
h_1	766.231	8.2
h_2	336.723	2.4
h_3	367.122	2.5

Table 4: Hash function comparison

3.3 Load Factor

The results of load factor comparison in Table 5 are a little bit strange. It's reasonable for memory to increases with the decline of the load factor but decoding time is irregular. That may be because of the design of hash functions. We tested many combinations of different hash functions and different load factors and still didn't find any patterns.

3.4 Default Probability

If w_3 in a trigram doesn't exist, the model will return a small probability ϵ. By Table 6, the best ϵ is $1.0e - 5$ where V is the vocabulary.

f	Decoding time	Memory
0.6	194.827	1.6G
0.7	392.512	1.4G
0.8	1126.999s	1.2G
0.85	336.723	1.1G
0.9	487.127	1.1G

Table 5: Load factor comparison

ϵ	BLEU
$1/\|V\|$	24.649
1e-10	24.406
1e-5	24.663
$2/\|V\|$	24.653

Table 6: Default probability comparison

4 Some Acceleration Tricks

4.1 Utilize JVM's optimization

When computing $P(w_3|w_1w_2)$, we often get the value of the same key many times. For example, we need to compute $N_{1+}(w_1w_2\bullet)$ twice and $c(w_1w_2)$ once to get $P(w_3|w_1w_2)$ so the model computes hash 3 times. We first tried computing hash first and accessing the value by the hash directly, however, we realized it could prevent JVM optimizing hot functions. So we choose to access values by keys and JVM's excellent optimization reduces time from above 500s to less than 350s.

But the previous try does speed up the phase of decoding training dataset. That's because in that phase, we add new keys into the table so it has side effect and is considerated by JVM unoptimizable.

4.2 Reduce Floating-point Multiplication

Floating-point number multiplication costs much more time than integer multiplication so we compute all integer multiplication first and then convert them into floating-point number to compute. This optimization gains about 30s.