Efficient Storage and Retrieval of N-gram Models

Undergraduate Group #5
Daniel Fabian, *dfabian5@kent.edu*
Lauren Greathouse, *lgreath5@kent.edu*

1. Introduction

N-grams are sequences of grams (all sequences are of size n) obtained by using a sliding window technique where grams can be anything from letters in a word, words in a text, DNA or protein sequences, or events [10]. The standard naming convention is to call them unigrams, bigrams, and trigrams when n is equal to one, two, and three respectively. Once n is higher than three 'n' is replaced by the number (e.g., 5-gram) [10]. A collection of n-grams with their associated frequency counts or probability is called a n-gram model [10]. Since n-grams preserve the order in which the grams occur they are useful in making predictions.

N-gram models have many applications and have been applied in many fields such as probability, communication theory, natural language processing, computational biology, and data compression [10]. Some more specific examples of their applications include predictive text, spell checkers, plagiarism and spam detection, machine translation, speech recognition, various compression algorithms, and DNA and protein sequencing [10]. Jurafsky and Martin [11] explain an example of how n-gram models are applied a spell checker. A word is typed and the application does not recognize the word because it is not found in the model. In order to make suggestions of what the word perhaps was supposed to be, the application can look at the surrounding context of the misspelled word (the words coming before and after). Once it has gathered the surrounding words it can use the model to see what words are the most likely to fit the context. These words can be shown to the user as potential corrections.

In Jurafsky and Martin's example the user will anticipate suggestions to be shown quite quickly, perhaps even before they are finished typing that sentence. This is not can uncommon expectation for n-gram model applications, but it is a challenging expectation to meet as n-gram models tend to be extremely large. For example, the Europarl model has a total of 101,428,257 n-grams, YahooV2 has 828,223,677 n-grams, and GoogleV2 has an impressive collection of 11,131,242,087 n-grams [4][5]. The typical large size of n-gram models and desired speed for these applications demands an efficient method of storing and querying data from n-gram models that won't use an excessive amount of storage or slow query responses.

2. Project Description

Since the data structure in which a n-gram model is stored plays an important role in querying and space efficiency, we intend to come up with a novel data structure to manage large n-gram models. We will be focusing on improving query speed while still using a reasonable amount of storage. We will be using two different types of queries which we will refer to as frequency count queries and most likely next queries. A frequency count query is provided a single n-gram and returns the frequency for the given

n-gram. The most likely next query will be given n-1 grams and the number of desired results (x desired results). The query will return the x most likely grams that are to follow. In our implementation, the n-grams will be a sequence of n words occurring in a text.

Often, data structures are able to improve query speed or space efficiency, but at the cost of the other. Possibly our biggest challenge will be to improve search speed without taking up an excessive amount of storage. To address this, we will be using a trie with hashmaps at every node. The structure will apply Elias-Fano to help keep it more compact. This implementation is very similar to Pibiri and Venturini's implementation [4][5]. To make it novel we are adding a sorted array to every node. The sorted array will hold the top-k results for the most likely next query for its respective node. We believe this will speed up the query speed for most likely next queries as the hashmap of all possible next grams will only need to be scanned when more than k results are requested.

The workload has been equally distributed and all members have worked on the same tasks (researching relevant literature, brainstorming ideas, writing, meeting up for in-person discussions, coding, evaluating, and documenting). As we initially planned, we worked closely together on all aspects of the project and communicated quite frequently. Time committed to the project was even for all members.

3. Background

Many different data structures can be used to store n-gram models, but typically some version of a hashmap or tree structure is used. Hashmap implementations will have a table for every order of n (bigrams, trigrams, etc.) and use the sequence of 1 through n-1 grams as a key to access the possible next grams and their frequencies or probabilities. Tries, a particularly well-suited tree for this problem, is a tree designed for storing associative arrays. Each gram becomes a key and n-grams can be recreated by tracing a path down the tree [5].

Hashmaps allow for fast retrieval, but require more storage space. Meanwhile trees can prevent redundancy and reduce the amount of space needed for storage, but produce a slower retrieval [3][5]. Often, data structures are able to improve query speed or space efficiency, but at the cost of the other. As a result, researchers have come up with ways to the combine hashmaps and tries to get the best of both data structures. Pauls and Klein introduced a largely successful structure where a trie is implemented with sorted arrays or hashmaps at every node [1]. This structure was further enhanced by Pibiri & Venturi by representing the words with Elias-Fano [4][5][6][7] before using them as a key.

Our implementation will be in C++ similarly to how Pibiri & Venturi [4][5] have implemented their solution. We also plan on using the same test data. Pibiri & Venturi provide a link to their implementation and data on GitHub [4]. Object oriented programming will be important as we are dealing with data structures.

4. Problem Definition

Because of the sequential nature and large sizes of n-gram models, it can be difficult to store them in an efficient manner. Typically redundancy in data would be removed from a dataset to accomplish compression, but in this case it is very difficult (e.g., "I am going to the store" and "You are going to the store" are very similar, but have different likelihoods of occurring). There is a lot of data to be stored and

often attempts to represent overlap loses necessary context. The trie is one of the most effective structures for representing word order overlap, but it also produces a slower query speed. Hashmaps on the other hand, don't attempt to represent overlap and save space, but still produce a faster query speed [3][5]. Since our main goal is to improve query speed while minimizing storage, we will be using a trie with both a hashmap and sorted array at every node. The words are encoded using Elias-Fano to help use as little memory as possible. In theory this should speed up most likely next queries because the entire hashmap of possible next grams doesn't need to be scanned unless a high number of most likely next words is requested.

5. The Proposed Techniques

5.1 A Trie Base

As briefly described, the base of our implementation is a trie. Typically a trie is constructed by inserting one n-gram at a time. Consider the trigram 'this is text'. When inserting the trigram, the first gram, 'this' is searched for out of the immediate children of the root node. If 'this' is not found, it is added as a child to the root node. If it is found, that child node representing the gram 'this' is traversed to and the second gram 'is' is searched for out of the children nodes. This process is repeated until the entire n-gram is stored. This means that every path from root to a lower node (leaf or overwise) represents a n-gram. The associated frequency count for the n-gram is stored within the last node of the n-gram's path in the trie. Note that n-grams of varying lengths can be stored in one trie. As stated previously, this is one of the main ways that redundancy is prevented.

5.2 Hashmap and Top-k Array at Every Node

As Pauls and Klein [1] first suggested, there will be a hashmap at every node. The hashmap is used to contain the children of possible next grams and make searching for a given child faster. To speed up searching for the most likely next grams there is a sorted array at every node which contains the k most likely next grams to follow. There is no redundancy between the hashmap and the sorted array. A possible next gram is only stored in one or the other.

5.3 Elias-Fano Representation

Elias-Fano representation was first applied to optimizing n-gram model storage by Pibiri and Venturi [4][5]. Elias-Fano representation is used to compress the storage of a sequence of increasing integers. Here, it is used in two ways. The first way is to compress the sequence of n grams which is a search key for the overall structure. Since the grams are words in our case, this was accomplished by first assigning every gram a unique integer ID. Then for every gram in a given n-gram the IDs were summed. This provides a sequence of increasing integers that represent each sub-n-gram of the full n-gram. Consider the trigram 'this is text'. If 'this' is assigned the ID 10, 'is' 2, and 'text' 5, the unigram 'this' can be represented as 10. The bigram 'this is' can be represented as 12 and the full trigram can be represented

as 17. Now the sequence 10, 12, 17 is eligible to be compressed using Elias-Fano representation. The second way it was used was to compress the memory addresses of the children for a given node. This way is more straightforward since memory addresses are already represented as increasing integers.

The Elias-Fano representation is produced by first converting each integer in the sequencing to a binary encoding. This uses $log(m)$ bits. Then each binary encoding is divided into upper and lower bits. The upper bits will use $log(n)$ bits and the lower will now only need $log(m) - log(n)$ bits. The lower bits are concatenated and set aside momentarily. This takes up $n\ log(m)$ bits. Eventually this will be the second half of the final representation. An ordered set of the possible upper bit values is created and the number of times each of the values occurred is saved. To obtain the first half of the final representation the number of times each of the possible upper bit values is occurred is represented with only 1's. A 0 follows the 1's to differentiate between values (e.g., 3, 0, 2 would be 11100110). This requires $2n$ bits. Finally, this bit sequence is concatenated with the lower bit sequence mentioned above [2]. If the example sequence above, 10, 12, 17, is represented using Elias-Fano they will first be converted into binary, 01010, 01100, 10001. The first three digits of every binary will be considered the upper bits and the last two will be considered lower bits, 010 | 10, 011 | 00, 100 | 01. The lower bits are concatenated, 10 00 01. A set of the possible upper bit values is generated, 000, 001, 010, 011, 100, 101, 110, 111, along with the number of times each occurred, 0, 0, 1, 1, 1, 0, 0, 0. The resulting bit sequence created to represent the upper bits is 0 0 10 10 10 0 0 0. The final result is the concatenation of 0101010000 and 100001. Recall that the sequence 10, 12, 17 represented the trigram "this is text". This means that the bit sequence 0101010000100001 is the Elias-Fano representation for "this is text".

5.4 Queries

As stated previously, two types of queries are used in the data structure, frequency count queries and most likely next queries. The frequency count query is given a n-gram and returns the frequency for the n-gram. This is achieved by traversing the trie in a similar fashion to how n-grams are inserted. The Elias-Fano representation (EFR) for the first gram (or equivalent unigram) is searched for in the sorted array of most likely next grams first (the sorted array is always searched first) if it is not found there it will search the hash map. Regardless of where the EFR is found, a pointer will result and direct the traversal to the child node containing the EFRs of all bigrams where the first gram is the same as the first gram in the provided n-gram and the second gram is any of the possible grams that can follow it. The process repeats in this manner until the full n-gram has been found. At this point, the frequency count will be stored with the EFR of the n-gram and will be returned.

The most likely next query will be given n-1 grams and the number of desired results (x desired results). The query will then return the x most likely grams that are to follow. This search starts off the same way as the frequency count search. It will use the same traversal pattern until the last gram provided (n-1) is found. At this point in the traversal, the node being visited represented the n-1 gram that was provided. The sorted array at this node stores the k most likely next grams to occur, so as long as the number of results requested is less than or equal to k, the results can simply be returned from the sorted array. If more than k results have been requested then the grams in the sorted array will still be returned, but the hash map will also need to be scanned to find the k through x most likely next grams to be returned in addition to the grams in the sorted array.

6. Visual Applications

The GUI for our implementation is very simple. First, the data structure is built using the command line. Once the data structure is built, the query and evaluation portion of program will run. This will display how much storage the data structure is using in bytes and prompts for the user to specify which query to run. Follow-up prompts will ask for the n-gram and/or the n-1 gram and the number of results desired depending on which query is chosen. The program will then display the results for the query and how long the query took in nanoseconds. The user can continue to make queries as long as they like. Below is a screenshot of the query and evaluation program being ran.

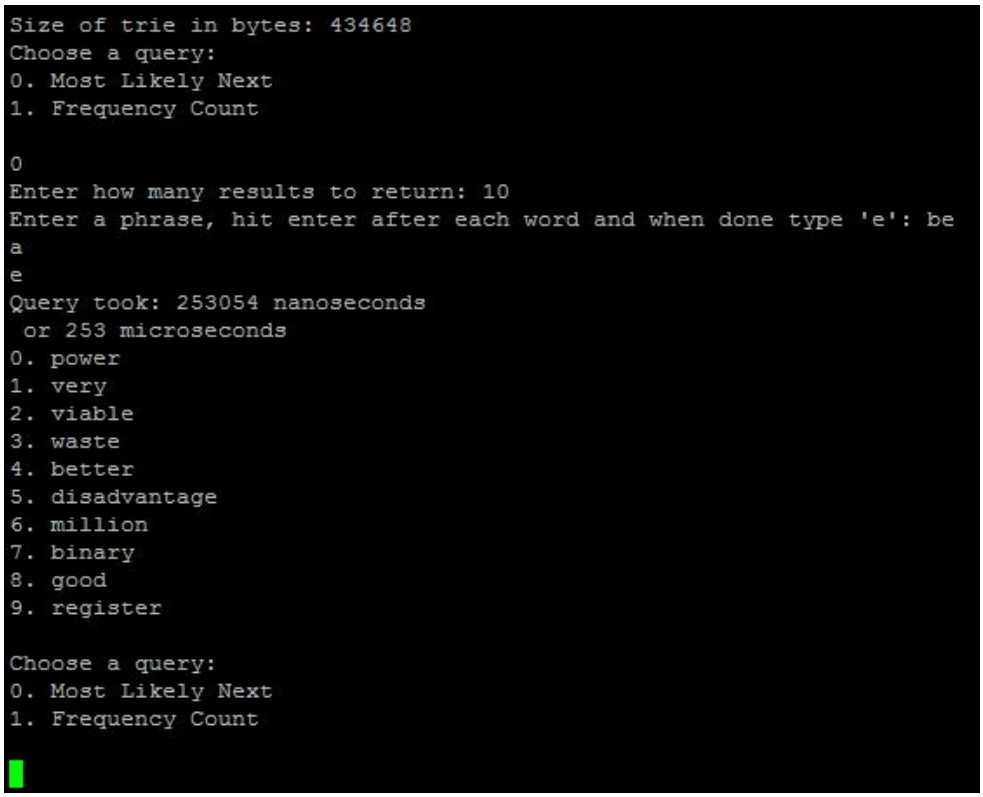

Figure 1: Screenshot of query and evaluation program running a most likely next query with the n-1 gram "be a" given. Ten results were requested. The size of the trie, holding trigrams, is reported along with the amount of time the query took.

7. Experimental Evaluation

In this section evaluation results from Pibiri and Venturini [4] are used for comparison against our own results. We have chosen their results to compare against our own since our data structure was largely inspired by theirs and also because their data structure seems to be the current golden standard. The same data is used in both experiments (we obtained the data straight from Pibiri's GitHub project: https://github.com/jermp/tongrams) which was gathered by randomly sampling from Europarl, YahooV2, and GoogleV2 [4]. However, Pibiri and Venturini report their results specific to Europarl, YahooV2, and

GoogleV2. Since our implementation does not differentiate between the three datasets, we have averaged the reported results from Pibiri and Venturini for comparison to our own results. Both data structures were storing 5-grams.

	Bytes/Gram	Frequency Count Query	Most Likely Next Query (num. of results <= k)	Most Likely Next Query (num. of results > k)
Pibiri & Venturini's Results	1.77	1.66 µsec/query	-	-
Our Results	20.00	22.40 µsec/query	30.53 µsec/query	4130 µsec/query

Figure 2: Table comparing the results of Pibiri and Venturini's results to our own results

The storage efficiency of Pibiri and Venturini's data structure shown in Figure 2 was calculated by averaging the three bytes/gram measurements for the total space occupancy of their three tries (each storing n-grams from one of the three datasets). The total space occupancy measurement takes into account space in use for gram-ID sequences, frequency counts and pointers (the entire structure). The storage efficiency result for our own data structure also takes into account space needed for the entire structure and then the byte count is divided by the number of n-grams being stored (how we imagine Pibiri and Venturini calculated their results). The frequency count query speed for Pibiri and Venturini was similarly calculated by averaging the three reported speeds for their PEF-tries (Partitioned Elias-Fano tries). For our most likely next query speed we made sure to include queries that asked for more than k results as well as less than k results (where k is the size of the sorted array).

The storage efficiency (1.77 bytes/gram) and query speed for frequency count queries (1.66 µsec/query) of Pibiri and Venturini's data structure is much greater than our own (20.00 bytes/gram and 22.40 µsec/query respectively). We believe the sorted array is to blame for the increased storage space and slower query speed. As reflected in Figure 2, Pibiri and Venturini do not run most likely next queries on their data structure. This makes it difficult to compare how our implementation would compare in this category, but it can be argued that our implementation would likely be faster as long as the number of results requested is typically less than k (the size of the sorted array). The support for this argument is that our implementation does not need to scan the entire hashmap at a given node and compare the frequencies of each possible next gram to fulfil this query as Pibiri and Venturini's implementation would need to. This process is very costly, as demonstrated in Figure 2 by comparing the two most likely next query results (num. of results <= k, 30.53 µsec/query; num. of results > k, 4130 µsec/query).

8. Future Work

Adding the sorted array to store the top-k most likely next grams did not increase the efficiency of storage space or query speed for frequency count queries (in fact it made it worse), but it could be argued that it would increase the speed of most likely next queries. This strategy could assist a data structure that is required to handle both types of queries, but should not be applied to data structures that are only required to handle frequency count queries. For those attempting to further improve Pibiri and Venturini's

current data structure [4][5] we recommend experimenting with different tree structures while still applying the hashmaps to every node to eliminate searching for children nodes. Researchers have introduced a path decomposed trie (PDT) where every node represents an entire path of a trie (or an entire n-gram) [8][9]. This may be interesting tree variant to experiment applying hashmaps to. We also recommend considering alternative compression algorithms to apply to the keys and pointer locations in place of the Elias-Fano representation.

9. References

[1] Adam Pauls and Dan Klein. Faster and Smaller N-gram Language Models. In *Association for Computational Linguistics (ACL)*, pages 258–267, 2011.

[2] Antonio Mallia. Sorted integers compression with Elias-Fano encoding. In *Antonio Mallia*. Antonio Mallia, 06 Jan. 2018. Web. 20 Apr. 2019.

[3] Daniel Robenek, Jan Platoš, & Václav Snášel. Efficient in-memory data structures for n-grams indexing. In *DATESO 2013*, pages 48–58, 2013.

[4] Giulio Ermanno Pibiri and Rossano Venturini. Efficient Data Structures for Massive N-Gram Datasets. In *SIGIR '17 Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval,* pages 615-624, 2017.

[5] Giulio Ermanno Pibiri and Rossano Venturini. Handling Massive N-Gram Datasets Efficiently. In *AMC Transactions on Information Systems (TOIS),* Vol. 37, No. 2, pages 1-39, 2019.

[6] Peter Elias. Efficient Storage and Retrieval by Content and Address of Static Files. In *Journal of the ACM (JACM)*, Vol. 21, No. 2, pages 246–260, 1974.

[7] Robert Mario Fano. On the number of bits required to implement an associative memory. In *Memorandum 61, Computer Structures Group, MIT*, 1971.

[8] Roberto Grossi & Giuseppe Ottaviano. Fast Compressed Tries through Path Decompositions. In *ACM Journal of Experimental Algorithmics*, Vol. 19, No. 1, Article 1.8, 2014

[9] Shunsuke Kanda, Kazuhiro Morita, & Masao Fuketa. Practical String Dictionary Compression Using String Dictionary Encoding. In *2017 International Conference on Big Data Innovations and Applications*, 2017.

[10] Wikipedia contributors. N-gram. In *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 27 Feb. 2019. Web. 23 Mar. 2019.

[11] Dan Jurafsky and James H. Martin. Speech and Language Processing Third Edition Draft. 2018.