An Efficient Indexer for Large N-Gram Corpora

Hakan Ceylan
Department of Computer Science
University of North Texas
Denton, TX 76203
`hakan@unt.edu`

Rada Mihalcea
Department of Computer Science
University of North Texas
Denton, TX 76203
`rada@cs.unt.edu`

Abstract

We introduce a new publicly available tool that implements efficient indexing and retrieval of large N-gram datasets, such as the Web1T 5-gram corpus. Our tool indexes the entire Web1T dataset with an index size of only 100 MB and performs a retrieval of any N-gram with a single disk access. With an increased index size of 420 MB and duplicate data, it also allows users to issue wild card queries provided that the wild cards in the query are contiguous. Furthermore, we also implement some of the smoothing algorithms that are designed specifically for large datasets and are shown to yield better language models than the traditional ones on the Web1T 5-gram corpus (Yuret, 2008). We demonstrate the effectiveness of our tool and the smoothing algorithms on the English Lexical Substitution task by a simple implementation that gives considerable improvement over a basic language model.

1 Introduction

The goal of statistical language modeling is to capture the properties of a language through a probability distribution so that the probabilities of word sequences can be estimated. Since the probability distribution is built from a corpus of the language by computing the frequencies of the N-grams found in the corpus, the data sparsity is always an issue with the language models. Hence, as it is the case with many statistical models used in Natural Language Processing (NLP), the models give a much better performance with larger data sets.

However the large data sets, such as the Web1T 5-Gram corpus of (Brants and Franz, 2006), present a major challenge. The language models built from these sets cannot fit in memory, hence efficient accessing of the N-gram frequencies becomes an issue. Trivial methods such as linear or binary search over the entire dataset in order to access a single N-gram prove inefficient, as even a binary search over a single file of 10,000,000 records, which is the case of the Web1T corpus, requires in the worst case $\lceil log_2(10,000,000) \rceil = 24$ accesses to the disk drive.

Since the access to N-grams is costly for these large data sets, the implementation of further improvements such as smoothing algorithms becomes impractical. In this paper, we overcome this problem by implementing a novel, publicly available tool[1] that employs an indexing strategy that reduces the access time to any N-gram in the Web1T corpus to a single disk access. We also make a second contribution by implementing some of the smoothing models that take into account the size of the dataset, and are shown to yield up to 31% perplexity reduction on the Brown corpus (Yuret, 2008). Our implementation is space efficient, and provides a fast access to both the N-gram frequencies, as well as their smoothed probabilities.

2 Related Work

Language modeling toolkits are used extensively for speech processing, machine translation, and many other NLP applications. The two of the most popular toolkits that are also freely available are the *CMU Statistical Language Modeling (SLM) Toolkit* (Clarkson and Rosenfeld, 1997), and the *SRI Language Modeling Toolkit* (Stolcke, 2002). However,

[1] Our tool can be freely downloaded from the download section under http://lit.csci.unt.edu

even though these tools represent a great resource for building language models and applying them to various problems, they are not designed for very large corpora, such as the Web1T 5-gram corpus (Brants and Franz, 2006), hence they do not provide efficient implementations to access these data sets.

Furthermore, (Yuret, 2008) has recently shown that the widely popular smoothing algorithms for language models such as *Kneser-Ney* (Kneser and Ney, 1995), *Witten-Bell* (Witten and Bell, 1991), or *Absolute Discounting* do not realize the full potentials of very large corpora, which often come with missing counts. The reason for the missing counts is due to the omission of low frequency N-grams in the corpus. (Yuret, 2008) shows that with a modified version of Kneser-Ney smoothing algorithm, named as the Dirichlet-Kneser-Ney, a 31% reduction in perplexity can be obtained on the Brown corpus.

A tool similar to ours that uses a hashing technique in order to provide a fast access to the Web1T corpus is presented in detail in (Hawker et al., 2007). The tool provides access to queries with wild card symbols, and the performance of the tool on 10^6 queries on a 2.66 GHz processor with 1.5 GBytes of memory is given approximately as one hour. Another tool, *Web1T5-Easy*, described in (Evert, 2010), provides indexing of the Web1T corpus via relational database tables implemented in an SQLite engine. It allows interactive searches on the corpus as well as collocation discovery. The indexing time of this tool is reported to be two weeks, while the non-cached retrieval time is given to be in order of a few seconds. Other tools that implement a binary search algorithm as a simpler, yet less efficient method are also given in (Giuliano et al., 2007; Yuret, 2007).

3 The Web1T 5-gram Corpus

The Web1T 5-gram corpus (Brants and Franz, 2006) consists of sequences of words (N-grams) and their associated counts extracted from a Web corpus of approximately one trillion words. The length of each sequence, N, ranges from 1 to 5, and the size of the entire corpus is approximately 88GB (25GB in compressed form). The unigrams form the vocabulary of the corpus and are stored in a single file which includes around 13 million tokens and their associated counts. The remaining N-grams are stored separately across multiple files in lexicographic order. For example, there are 977,069,902 distinct trigrams in the dataset, and they are stored consecutively in 98 files in lexicographic order. Furthermore, each N-gram file contains 10,000,000 N-grams except the last one, which contains less. It is also important to note that N-grams with counts less than 40 are excluded from the dataset for $N = 2, 3, 4, 5$, and the tokens with less than 200 are excluded from the unigrams.

4 The Indexer

4.1 B$^+$-trees

We used a B$^+$-tree structure for indexing. A B$^+$-tree is essentially a balanced search tree where each node has several children. Indexing large files using B$^+$ trees is a popular technique implemented by most database systems today as the underlying structure for efficient range queries. Although many variations of B$^+$-trees exist, we use the definition for primary indexing given in (Salzberg, 1988). Therefore we assume that the data, which is composed of records, is only stored in the leaves of the tree and the internal nodes store only the keys.

The data in the leaves of a B$^+$-tree is grouped into *buckets*, where the size of a bucket is determined by a bucket factor parameter, *bkfr*. Therefore at any given time, each bucket can hold a number of records in the range $[1, bkfr]$. Similarly, the number of keys that each internal node can hold is determined by the *order* parameter, *v*. By definition, each internal node except the root can have any number of keys in the range $[v, 2v]$, and the root must have at least one key. Finally, an internal node with k keys has $k + 1$ children.

4.2 Mapping Unigrams to Integer Keys

A key in a B$^+$-tree is a lookup value for a record, and a record in our case is an N-gram together with its count. Therefore each line of an N-gram file in the Web1T dataset makes up a record. Since each N-gram is distinct, it is possible to use the N-gram itself as a key. However in order to reduce the storage requirements and make the comparisons faster during a lookup, we map each unigram to an integer, and form the keys of the records using the integer values instead of the tokens themselves.[2]

To map unigrams to integers, we use the unigrams sorted in lexicographic order and assign an integer value to each unigram starting from 1. In other words, if we let the m-tuple $U = (t_1, t_2, ..., t_m)$ represent all the unigrams sorted in lexicographic order,

[2]This method does not give optimal storage, for which one should implement a compression Huffman coding scheme.

then for a unigram t_i, i gives its key value. The key of trigram "$t_i\ t_j\ t_k$" is simply given as "$i\ j\ k$." Thus, the comparison of two keys can be done in a similar fashion to the comparison of two N-grams; we first compare the first integer of each key, and in case of equality, we compare the second integers, and so on. We stop the comparison as soon as an inequality is found. If all the comparisons result in equality then the two keys (N-grams) are equal.

4.3 Searching for a Record

We construct a B$^+$-tree for each N-gram file in the dataset for $N = 2, 3, 4, 5$, and keep the key of the first N-gram for each file in memory. When a query q is issued, we first find the file that contains q by comparing the key of q to the keys in memory. Since this is an in-memory operation, it can be simply done by performing a binary search. Once the correct file is found, we then search the B$^+$-tree constructed for that file for the N-gram q by using its key.

As is the case with any binary search tree, a search in a B$^+$-tree starts at the root level and ends in the leaves. If we let r_i and p_j represent a key and a pointer to the child of an internal node respectively, for $i = 1, 2, ..., k$ and $j = 1, 2, ..., k + 1$, then to search an internal node, including the root, for a key q, we first find the key r_m that satisfies one of the following:

- $(q < r_m) \wedge (m = 1)$
- $(r_{m-1} \leq q) \wedge (r_m > q)$ for $1 < m \leq k$
- $(q > r_m) \wedge (m = k)$

If one of the first two cases is satisfied, the search continues on the child node found by following p_m, whereas if the last condition is satisfied, the pointer p_{m+1} is followed. Since the keys in an internal node are sorted, a binary search can be performed to find r_m. Finally, when a leaf node is reached, the entire bucket is read into memory first, then a record with a key value of q is searched.

4.4 Constructing a B$^+$-tree

The construction of a B$^+$-tree is performed through successive record insertions.[3] Given a record, we first compute its key, find the leaf node it is supposed to be in, and insert it if the bucket is not full. Otherwise, the leaf node is split into two nodes, each containing $\lceil bkfr/2 \rceil$, and $\lfloor bkfr/2 \rfloor + 1$ records, and the first key of the node containing the larger key values is placed into the parent internal node together with the node's pointer. The insertion of a key to an internal node is similar, only this time both split nodes contain v values, and the middle key value is sent up to the parent node.

Note that not all the internal nodes of a B$^+$-tree have to be kept on the disk, and read from there each time we do a search. In practice, all but the last two levels of a B$^+$-tree are placed in memory. The reason for this is the high branching factor of the B$^+$-trees together with their effective storage utilization. It has been shown in (Yao, 1978) that the nodes of a high-order B$^+$-tree are $ln2 \approx 69\%$ full on average.

However, note that the tree will be fixed in our case, i.e., once it is constructed we will not be inserting any other N-gram records. Therefore we do not need to worry about the 69% space utilization, but instead try to make each bucket, and each internal node full. Thus, with a $bkfr = 1250$, and $v = 100$, an N-gram file with 10,000,000 records would have 8,000 leaf nodes on level 3, 40 internal nodes on level 2, and the root node on level 1. Furthermore, let us assume that integers, disk and memory pointers all hold 8 bytes of space. Therefore a 5-gram key would require 40 bytes, and a full internal node in level 2 would require $(200x40) + (201x8) = 9,608$ bytes. Thus the level 2 would require $9,608x40 \approx 384$ Kbytes, and level 1 would require $(40*40) + (41*8) = 1,928$ bytes. Hence, a Web1T 5-gram file, which has an average size of 286 MB can be indexed with approximately 386 Kbytes. There are 118 5-gram files in the Web1T dataset, so we would need $386x118 \approx 46$ MBytes of memory space in order to index all of them. A similar calculation for 4-grams, trigrams, and bigrams for which the bucket factor values are selected as 1600, 2000, and 2500 respectively, shows that the entire Web1T corpus, except unigrams, can be indexed with approximately 100 MBytes, all of which can be kept in memory, thereby reducing the disk access to only one. As a final note, in order to compute a key for a given N-gram quickly, we keep the unigrams in memory, and use a hashing scheme for mapping tokens to integers, which additionally require 178 Mbytes of memory space.

The choice of the bucket factor and the inter-

[3] Note that this may cause efficiency issues for very large files as memory might become full during the construction process, hence in practice, the file is usually sorted prior to indexing.

nal node order parameters depend on the hard-disk speed, and the available memory.[4]. Recall that even to fetch a single N-gram record from the disk, the entire bucket needs to be read. Therefore as the bucket factor parameter is reduced, the size of the index will grow, but the access time would be faster as long as the index could be entirely fit in memory. On the other hand, with a too large bucket factor, although the index can be made smaller, thereby reducing the memory requirements, the access time may be unacceptable for the application. Note that a random reading of a bucket of records from the hard-disk requires the disk head to first go to the location of the first record, and then do a sequential read.[5] Assuming a hard-disk having an average transfer rate of 100 MBytes, once the disk head finds the correct location, a 40 bytes N-gram record can be read in $4x10^{-7}$ seconds. Thus, assuming a seek time around 8-10 ms, even with a bucket factor of 1,000, it can be seen that the seek time is still the dominating factor. Therefore, as the bucket size gets smaller than 1,000, even though the index size will grow, there would be almost no speed up in the access time, which justifies our parameter choices.

4.5 Handling Wild Card Queries

Having described the indexing scheme, and how to search for a single N-gram record, we now turn our attention to queries including one or more wild card symbols, which in our case is the underscore character "_", as it does not exist among the unigram tokens of the Web1T dataset. We manually add the wild card symbol to our mapping of tokens to integers, and map it to the integer *0*, so that a search for a query with a wild card symbol would be unsuccessful but would point to the first record in the file that replaces the wild card symbol with a real token as the key for the wild card symbol is guaranteed to be the smallest. Having found the first record we perform a sequential read until the last read record does not match the query. The reason this strategy works is because the N-grams are sorted in lexicographic order in the data set, and also when we map unigram tokens to integers, we preserve their order, i.e., the first token in the lexicographically sorted unigram list is assigned the value 1, the second is assigned 2, and so forth. For example, for a given query *Our Honorable _*, the record that would be pointed at the end of search in the trigram file *3gm-0041* is the N-gram *Our Honorable Court 186*, which is the first N-gram in the data set that starts with the bigram *Our Honorable*.

Note however that the methodology that is described to handle the queries with wild card symbols will only work if the wild card symbols are the last tokens of the query and they are contiguous. For example a query such as *Our _ Court* will not work as N-grams satisfying this query are not stored contiguously in the data set. Therefore in order to handle such queries, we need to store additional copies of the N-grams sorted in different orders. When the last occurrence of the contiguous wild card symbols is in position p of a query N-gram for $p = 0, 1, ..., N - 1$, then the N-grams sorted lexicographically starting from position $(p + 1) mod N$ needs to be searched. A lexicographical sort for a position p, for $0 \leq p \leq (N - 1)$ is performed by moving all the tokens in positions $0...(p - 1)$ to the end for each N-gram in the data set. Thus, for all the bigrams in the data set, we need one extra copy sorted in position 1, for all the trigrams, we need two extra copies; one sorted in position 1, and another sorted in position 2, and so forth. Hence, in order to handle the contiguous wild card queries in any position, in addition to the 88 GBytes of original Web1T data, we need an extra disk space of 265 GBytes. Furthermore, the indexing cost of the duplicate data is an additional 320 MBytes. Thus, the total disk cost of the system will be approximately 353 GBytes plus the index size of 420 MBytes, and since we keep the entire index in memory, the final memory cost of the system will be 420 MBytes + 178 MBytes = 598 MBytes.

4.6 Performance

Given that today's commodity hardware comes with at least 4 GBytes of memory and 1 TBytes of hard-disk space, the requirements of our tool are reasonable. Furthermore, our tool is implemented in a client-server architecture, and it allows multiple clients to submit multiple queries to the server over a network. The server can be queried with an N-gram query either for its count in the corpus, or its smoothed probability with a given smoothing method. The queries with wild cards can ask for the retrieval of all the N-grams satisfying a query, or only for the total count so the network overhead can

[4]We used a 7200 RPM disk-drive with an average read seek time of 8.5 ms, write seek time of 10.0 ms, and a data transfer time up to 3 GBytes per second.

[5]A rotational latency should also be taken into account before the sequential reading can be done.

be avoided depending on the application needs.

Our program requires about one day of offline processing due to resorting the entire data a few times. Note that some of the files in the corpus need to be sorted as many as four times. For the sorting process, the files are first individually sorted, and then a k-way merge is performed. In our implementation, we used a min heap structure for this purpose, and k is always chosen as the number of files for a given N. The index construction however is relatively fast. It takes about an hour to construct the index for the 5-grams. Once the offline processing is done, it only takes a few minutes to start the server, and from that point the online performance of our tool is very fast. It takes about 1-2 seconds to process 1000 randomly picked 5-gram queries (with no wild card symbols), which may or may not exist in the corpus. For the queries asking for the frequencies only, our tool implements a small caching mechanism that takes the temporal locality into account. The mechanism is very useful for wild card queries involving stop words, such as *"the _"*, and *"of the _"* which occur frequently, and take a long time to process due to the sequential read of a large number of records from the data set.

5 Lexical Substitution

In this section we demonstrate the effectiveness of our tool by using it on the the English Lexical Substitution task, which was first introduced in SemEval 2007 (McCarthy and Navigli, 2007). The task requires both the human annotators and the participating systems to replace a target word in a given sentence with the most appropriate alternatives. The description of the tasks, the data sets, the performance of the participating systems as well as a post analysis of the results is given in (McCarthy and Navigli, 2009).

Although the task includes three subtasks, in this evaluation we are only concerned with one of them, namely the *best* subtask. The best subtask asks the systems and the annotators to provide only one substitute for the target words – the most appropriate one. Two separate datasets were provided with this task: a trial dataset was first provided in order for the participants to get familiar with the task and train their systems. The trial data used a lexical sample of 30 words with 10 instances each. The systems were then tested on a larger test data, which used a lexical sample of 171 words each again having 10 instances.

Our methodology for this task is very simple; we replace the target word with an alternative from a list of candidates, and find the probability of the context with the new word using a language model. The candidate that gives the highest probability is provided as the system's best guess. The list of candidates is obtained from two different lexical sources, WordNet (Fellbaum, 1998) and Roget's Thesaurus (Thesaurus.com, 2007). We retrieve all the synonyms for all the different senses of the word from both resources and combine them. We did not consider any lexical relations other than synonymy, and similarly we did not consider any words at a further semantic distance.

Model	Precision	Mod Precision
No Smoothing	10.13	14.78
Absolute Discounting	11.05	16.75
KN with Missing Counts	11.19	16.75
Dirichlet KN	10.98	15.76

Table 1: Results on the trial data

Model	Precision	Mod Precision
No Smoothing	9.01	14.15
Absolute Discounting	11.64	18.62
KN with Missing Counts	11.61	18.54
Dirichlet KN	11.03	17.48
Best Baseline	9.95	15.28
Best SEMEVAL System	12.90	20.65

Table 2: Results on the test data

We start with a simple language model that calculates the probability of the context of a word, and then continue with three smoothing algorithms discussed in (Yuret, 2008), namely *Absolute Discounting*, *Kneser-Ney with Missing Counts*, and the *Dirichlet-Kneser-Ney Discounting*. Note that all three are interpolated models, i.e., they do not just back-off to a lower order probability when an N-gram is not found, but rather use the higher and lower order probabilities all the time in a weighted fashion.

The results on the trial dataset are shown in Table 1, and the results on the test dataset are shown in Table 2. In all the experiments we use the trigram models, i.e., we keep N fixed to 3. Since our system makes a guess for all the target words in the set, our precision and recall scores, as well as the mod precision and the mod recall scores are the same, so only one from each is shown in the table. Note that the highest achievable score for this task is not 100%, but is restricted by the frequency of the best substitute, and it is given as 46.15%. The highest scoring participating system achieved 12.9%, which gave a 2.95% improvement over the baseline (Yuret, 2008; McCarthy and Navigli, 2009); the scores obtained by the best SemEval system as well as the best baseline calculated using the synonyms for the first synset in WordNet are also shown in Table 2.

On both the trial and the test data, we see that the interpolated smoothing algorithms consistently improve over the naive language modeling, which is an encouraging result. Perhaps a surprising result for us was the performance of the Dirichlet-Kneser-Ney Smoothing Algorithm, which is shown to give minimum perplexity on the Brown corpus out of the given models. This might suggest that the parameters of the smoothing algorithms need adjustments for each task.

It is important to note that this evaluation is meant as a simple proof of concept to demonstrate the usefulness of our indexing tool. We thus used a very simple approach for lexical substitution, and did not attempt to integrate several lexical resources and more sophisticated algorithms, as some of the best scoring systems did. Despite this, the performance of our system exceeds the best baseline, and is better than five out of the eight participating systems (see (McCarthy and Navigli, 2007)).

6 Conclusions

In this paper we described a new publicly available tool that provides fast access to large N-gram datasets with modest hardware requirements. In addition to providing access to individual N-gram records, our tool also handles queries with wild card symbols, provided that the wild cards in the query are contiguous. Furthermore, the tool also implements smoothing algorithms that try to overcome the missing counts that are typical to N-gram corpora due to the omission of low frequencies. We tested our tool on the English Lexical Substitution task, and showed that the smoothing algorithms give an improvement over simple language modeling.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation CAREER award #0747340 and IIS awards #0917170 and #1018613. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

T. Brants and A. Franz. 2006. Web 1T 5-gram corpus version 1. *Linguistic Data Consortium*.

P. Clarkson and R. Rosenfeld. 1997. Statistical language modeling using the cmu-cambridge toolkit. In *Proceedings of ESCA Eurospeech*, pages 2707–2710.

S. Evert. 2010. Google web 1t 5-grams made easy (but not for the computer). In *Proceedings of the NAACL HLT 2010 Sixth Web as Corpus Workshop*, WAC-6 '10, pages 32–40.

C. Fellbaum, editor. 1998. *WordNet: An Electronic Lexical Database*. MIT Press, Cambridge, MA.

C. Giuliano, A. Gliozzo, and C. Strapparava. 2007. Fbk-irst: lexical substitution task exploiting domain and syntagmatic coherence. In *SemEval '07: Proceedings of the 4th International Workshop on Semantic Evaluations*, pages 145–148.

T. Hawker, M. Gardiner, and A. Bennetts. 2007. Practical queries of a massive n-gram database. In *Proceedings of the Australasian Language Technology Workshop 2007*, pages 40–48, Melbourne, Australia.

R. Kneser and H. Ney. 1995. Improved backing-off for n-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184 vol.1.

D. McCarthy and R. Navigli. 2007. Semeval-2007 task 10: English lexical substitution task. In *SemEval '07: Proceedings of the 4th International Workshop on Semantic Evaluations*, pages 48–53.

D. McCarthy and R. Navigli. 2009. The english lexical substitution task. *Language Resources and Evaluation*, 43:139–159.

B. Salzberg. 1988. *File structures: an analytic approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

A. Stolcke. 2002. SRILM – an extensible language modeling toolkit. In *Proceedings of ICSLP*, volume 2, pages 901–904, Denver, USA.

Thesaurus.com. 2007. Rogets new millennium thesaurus, first edition (v1.3.1).

I. H. Witten and T. C. Bell. 1991. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094.

A. Chi-Chih Yao. 1978. On random 2-3 trees. *Acta Inf.*, 9:159–170.

D. Yuret. 2007. Ku: word sense disambiguation by substitution. In *SemEval '07: Proceedings of the 4th International Workshop on Semantic Evaluations*, pages 207–213.

D. Yuret. 2008. Smoothing a tera-word language model. In *HLT '08: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies*, pages 141–144.