Efficient Search in Short Documents

Taras Shevchenko

National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute",
Institute for Applied System Analysis
taras.shechenko.public@gmail.com
http://tshev.github.io

Abstract. This work is about reasonable choice of the component algorithms for the implementation of Okapi BM25. It describes important aspects of building search engines and focuses on the efficient and implementation of Okapi BM25 for k-grams. This extended abstract should be treated as a part of Master's thesis in System Analysis and Control Theory. Since Okapi BM25 requires the usage of several general-purpose algorithms, it is important to choose the best version of the existing algorithms in order to squeeze every bit of the hardware.

Keywords: TF-IDF, Okapi BM25, Inverted index, Information Retrieval, n-grams

1 Introduction

It is impossible to imagine modern website without search engine. High quality search is especially important for E-commerce websites. Since Internet-stores contain millions of documents, it is important to choose appropriate efficient algorithms and use available memory effectively.

Users are making mistakes in search queries and still want to get relevant results. Also, words in a query could be written in different forms, so that bag-of-words model does not work. So, it is important to write robust algorithms for Information Retrieval. In large scale, if the implementation is 3 times slower, that means you have to buy 3 times more server farms, which could cost billions of dollars.

There is a number of probabilistic retrieval methods:

1. Binary independence model.
2. Bayesian network approaches.
3. Okapi BM25 [4].

Okapi BM25 is on of the best choices for probabilistic retrieval. It is used in Modern Search Engines such as Solr and Elastic Search instead of traditional TF-IDF [1].

First of all, would be shown the weaknesses of Okapi BM25 and then would be proposed how to fix them. Then would be explained what component algorithms should be used for the implementation of Okapi BM25 and how to improve state of the art results for frequency counting.

2 Problems of classical Okapi BM25

BM25 is a bag-of-words retrieval function that ranks a set of documents based on the query terms appearing in each document.

$$score(D,Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{avgdl}\right)} \tag{1}$$

$$IDF(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}, \tag{2}$$

$k_1 = 1.5$ and $b = 0.75$ - parameters, $|D|$ - length of the document in words, $avgdll$ - average document length in the text collection, $n(q_i)$ - number of documents which contain term q_i, $f(q_i, D)$ - term frequency in the document D [4].

document_id	Title	Search query	Score
1	Why C++ is better than Rust	Why C++ is better than Rust	0.85
2	Why Rust is better than C++	Why C++ is better than Rust	0.85
3	Programming in C++	Why C++ is better than Rust	0.08
4	Programin in Rust	Why C++ is better than Rust	0.07
5	Samsung 8	Why C++ is better than Rust	0
6	Good life	Why C++ is better than Rust	0

Table 1. Weaknesses of classical usage of Okapi BM 25

Table 2 shows weaknesses of bag-of-words model. The relevance does not depend on the order of words, so documents 1 and 2 have the same score, but, clearly, the first one is more relevant than the second.

3 Improvement of Okapi BM25

3.1 Counting term frequencies

First of all, words from original Okapi BM 25 would replaced by 3-grams and 4-grams. For efficiency they could be encoded and would be encoded as 32 bit integers.

Important algorithms for Okapi BM25:

1. sorting
2. finding frequences
3. linear search

Data structures:

1. inverted index

2. posting list
3. hash table

This change would decrease number of comparisons and speed up sorting and hashing.

It is important to choose an efficient algorithm for term frequency counting and hash table [3] is not the best option. Let's apply the following algorithm on sequence s, which is represented by pair of 2 iterators $[first, last)$ and the target is represented by output iterator *out*:

1. Sort sequence $[first, last)$, go to step 2.
2. If $first$ equals $last$, return *out*. Otherwise go to step 3.
3. Store the data from $first$ at *tmp*.
4. Find the first element which is not equal to *tmp*.
5. Assign offset to the *count*, assign the position of the first element to $first$ which does not equal to *tmp*. Write pair $tmp, count$ to the output iterator, increment *out*, go to step 2.

For more information about iterators, read an article about Generic Programming [6]. The whole point is to choose faster algorithm for short relatively short sequences. Let's compare described algorithm with hash tables on the following configuration: Intel Core i7 7700HQ, RAM Hynix SODIMM 2 x 8GB, 2400MHz, compiler GCC 7.3.1.

Size	Insertion into hash table		Straight sort and counting	
	mean	std	mean	std
50	2.63757	8.22384	0.387384	3.76955
100	6.63765	25.1815	0.903355	6.90381
200	15.129	45.2746	2.51254	22.8
2000	158.216	132.655	101.99	334.018
4000	325.473	282.319	220.108	354.548
32000	2287.54	4274.93	2240.9	935.053
64000	3843.75	5391.52	4799.71	1170.27

Table 2. Timings in milliseconds for frequency counting algorithms.

Implementation details:

1. The list of stopwords is stored as a sorted sequence of 32-bits integers.
2. Stopwords should be removed from the sequence.
3. Posting lists are encoded by varint-G8IU [5].
4. Search query is treated as a sequence of bytes. For example, phrase "Search query" would be treated as a sequence "sea", "ear", "arc", "rch", "ch ", " qu", "que", "uer", "ery".
5. Inverted index is updated by batches.
6. Retrieval of k most relevant documents is done by selecting $n * k$ frequences from posting lists.

4 Conclusion

We've got simple robust efficient search engine. In this article, we've got advan-

Search query	The most relevant document
Samsung Galaxy	Samsung Galaxy Tab 7.7
john ernst bnson	John Ernest Benson
john cole	John Cole (priest)
johny burns	John Burns (audio engineer)

Table 3. Search results on wikipedia_test10k.txt [2]

tages from fast CPU operations on 32-bit integers. It allows us to use binary search, which is faster than hash maps for certain input sizes. Since IDF is never getting less than zero, it is possible to get use it on 3-grams for short texts.

It is important to remember that bag-of-words model, does not depend on the order of terms. Text queries, which contain same terms, have the same scores. Further work:

1. Compare different data structures for posting lists: arrays, b-trees, red-black trees.
2. Compare different top K document retrieval heuristics for different structure of posting lists.
3. Use persistant storage.
4. Combine probabalistic retrieval with topic modeling.
5. Extend search query by applying vector space model.

References

1. Bm25 the next generation of lucene relevance. https://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/.
2. wikipedia_test10k.txt. `https://s3.amazonaws.com/fair-data/starspace/wikipedia_devtst.tgz`.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
4. Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, 1999.
5. Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. Simd-based decoding of posting lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 317–326, New York, NY, USA, 2011. ACM.
6. Alexander A. Stepanov and Daniel E. Rose. *From Mathematics to Generic Programming*. Addison-Wesley Professional, 1st edition, 2014.