Off the Beaten Path: Let’s Replace Term-Based Retrieval with k-NN Search

Leonid Boytsov
Carnegie Mellon University
Pittsburgh, PA, USA
srchvrs@cs.cmu.edu

David Novak
Masaryk University
Brno, Czech Republic
david.novak@fi.muni.cz

Yury Malkov
Institute of Applied Physics RAS
Nizhny Novgorod, Russia
yurymalkov@mail.ru

Eric Nyberg
Carnegie Mellon University
Pittsburgh, PA, USA
ehn@cs.cmu.edu

ABSTRACT
Retrieval pipelines commonly rely on a term-based search to obtain candidate records, which are subsequently re-ranked. Some candidates are missed by this approach, e.g., due to a vocabulary mismatch. We address this issue by replacing the term-based search with a generic k-NN retrieval algorithm, where a similarity function can take into account subtle term associations. While an exact brute-force k-NN search using this similarity function is slow, we demonstrate that an approximate algorithm can be nearly two orders of magnitude faster at the expense of only a small loss in accuracy. A retrieval pipeline using an approximate k-NN search can be more effective and efficient than the term-based pipeline. This opens up new possibilities for designing effective retrieval pipelines. Our software (including data-generating code) and derivative data based on the Stack Overflow collection is available online[1]. This revision is a slightly extended version of the respective CIKM’16 paper.

1. INTRODUCTION

Due to advances in computing, a full-text search has become a ubiquitous information technology. However, this technology still largely relies on memorization of document terms and matching them with the query terms provided by a user.

The full-text search is powered by a term-based inverted index: a classic data structure that links document terms—and sometimes phrases—with their locations in a text collection. This way of organizing text data traces back to paper book indices containing alphabetical lists of principal words. In particular, it was used in a 13th century Bible concordance, long before the computer era[2].

Modern retrieval systems answer queries in a pipeline fashion. First, an term-based inverted index is used to generate a list of candidate documents containing some or all query terms. Second, this list is refined and re-ranked. A few highly-ranked documents are then presented to the user.

Re-ranking may be carried out in several steps, where earlier steps employ cheap ranking functions—such as BM25[3] or language models[4]—relying solely on term occurrence statistics. A final, aggregation, step typically combines numerous relevance signals generated by upstream components. The aggregation step is often carried out using statistical learning-to-rank algorithms[5].

This filter-and-refine approach hinges on the assumption that a term-based search generates a reasonably complete list of candidate documents. However, this assumption is not fully accurate, in particular, because of a vocabulary gap, i.e., a mismatch between query and document terms denoting same concepts. The vocabulary gap is a well-known phenomenon. Furnas et al.[6] showed that, given a random concept, there is less than a 20% chance that two randomly selected humans denote this concept using the same term. Zhao and Callan[7] found that a term mismatch ratio—i.e., a rate at which a query term fails to appear in a relevant document—is roughly 50%.

Furthermore, according to Furnas et al.[6], focusing only on a few synonyms is not sufficient to effectively bridge the vocabulary gap. Specifically, it was discovered that, after soliciting 15 synonyms describing a single concept from a panel of subject experts, there was still a 20% chance that a new person coined a previously unseen term. To cope with this problem, Furnas et al. [8] proposed a system of unlimited term aliases, where potential synonyms would be interactively explored and presented to the user in a dialog mode.

An established automatic technique aiming to reduce the vocabulary gap is a query expansion. It consists in expanding (and sometimes rewriting) a source query using related terms and/or phrases. For efficiency reasons, traditional query expansion techniques are limited to dozens of expansion terms[9]. Using hundreds or thousands of expansion terms seems to be infeasible within a framework of the term-based inverted index. In contrast, we demonstrate that a system of unlimited term aliases can be successfully implemented within a more generic framework of a k-nearest neighbor search (k-NN search).

It has been long recognized that the k-NN search shows a promise to make retrieval a conceptually simple optimization procedure[10]. This approach may permit a separation of labor between data scientists, focusing on methods’ accuracy, and software engineers, focusing on development of more efficient and/or scalable search
approaches. However, the $k$-NN search proved to be a challenging problem due to the curse of dimensionality. There is empirical and theoretical evidence that this problem cannot be solved both exactly and efficiently in a high-dimensional setting. For some data sets, e.g., in the case of vectors with randomly generated elements, exact methods degenerate to a brute force search for just a dozen of dimensions. Some data sets only “look” high-dimensional, but possess properties of low-dimensional data sets, i.e., they have a low intrinsic dimensionality. Unfortunately, textual data seems to be intrinsically high-dimensional. For example, using the definition of Chávez et al. [13], we estimate that the intrinsic dimensionality of Wikipedia TF$\times$IDF vectors is about 2500 in the case of the metric angular distance.

The curse of dimensionality can be partially lifted by using Locality Sensitive Hashing (LSH) techniques [6, 27, 35]. There are numerous modifications of LSH, which differ primarily in how they construct families of locality-sensitive functions [69]. Most of the research focuses on hash functions for well-studied similarities, such as the Euclidean distance and the cosine similarity.

In this paper, however, we explore an effective similarity function $BM25+Model 1$, which is neither metric nor symmetric (see § 2.1.3). We demonstrate that it is possible to carry out an efficient and effective $k$-NN search (for $BM25+Model 1$) using pivoting techniques. In that, the approximate $k$-NN search is nearly two orders of magnitude faster than the respective exact brute force search. The $k$-NN search can be $1.5 \times$ faster than Lucene, while being more effective due to bridging the vocabulary gap.

To ease reproducibility, we make our software (including data-generating code) and derivative data based on the Stack Overflow collection available online.

2. APPROACH

We focus on a task of searching a large collection of answers extracted from a community QA website. The questions and answers are submitted by real people, who also select best answers. A question and the respective best answer represent one QA pair. While community QA is an important task on its own, it is used here primarily as a testbed to demonstrate the potential of the $k$-NN search as a substitute for term-based retrieval. Due to the curse of dimensionality, we have to resort to approximate searching. Note that we need a similarity function that outstrips the baseline method $BM25$ by a good margin. Otherwise, gains achieved by employing a more sophisticated similarity would be invalidated by the inaccuracy of the search procedure.

One effective way to build such a similarity function is to learn a generative question-answer translation model, e.g., IBM Model 1 [10]. However, “...the goal of question-answer translation is to learn associations between question terms and synonymous answer terms, rather than the translation of questions into fluent answers.” [57] The idea of using a translation model in retrieval applications was proposed by Berger et al. [4]. It is now widely adopted by the IR and QA communities [15, 82, 57, 74, 63, 23]. Linearly combining $BM25$ and $logarithms$ of IBM Model 1 scores produces a similarity function that is considerably more accurate than $BM25$ alone (by up to 30% on our data, see Table 5).

Learning IBM Model 1 requires a large monolingual parallel corpus. In that, the community QA data sets seem to be the best publicly available source of such corpora. Note that a monolingual corpus can be built from search engine click-through logs [55]. Yet, such data is not readily available for a broad scientific community. Another advantage of community QA data sets is that they permit a large scale automatic evaluation with sizeable training and testing subsets.

Specifically, we extract QA pairs from the following collections:

- L6 - Yahoo! Answers Comprehensive version 1.0 (about 4.4M questions);
- L5 - Yahoo! Answers Manner version 2.0 (about 142K questions), which is a subset of L6 created by Surdeanu al. [63];
- Stack Overflow (about 8.8M answered questions).

Yahoo! Answers collections are available through Yahoo! WebScope and can be requested by researchers from accredited universities.

For each question, there is always an answer (and the best answer is always present). The Stack Overflow collection is freely available for download.

While there are 8.8M answered questions, the best answer is not always selected by an asker. Such questions are discarded leaving us with 6.2M questions.

Each question has a (relatively) short summary of content, which is usually accompanied by a longer description. The question summary concatenated with the description is used as a query with the objective of retrieving the corresponding best answer. The best answer is considered to be the only relevant document for the query.

The accuracy of a retrieval system is measured using standard IR metrics: a Mean Reciprocal Rank (MRR), a precision at rank one (P@1) and an answer recall measured for the set of top-$N$ ranked documents. P@1—one of the main evaluation metrics—is equal to the fraction of queries where the highest ranked document is a true best answer to the question.

We process collections by removing punctuation, extracting tokens and termlemmas using Stanford CoreNLP [52] (instead, one can use any reasonably accurate tokenizer and lemmatizer). All terms and lemmas are lowercased; stopwords are removed. Note that we keep both lemmas and original terms. In Stack Overflow we remove all the code (the content marked by the tag `code`).

Each collection is randomly split into several subsets, which include training, two development (dev1 and dev2), and testing subsets. In the case of Comprehensive and Stack Overflow, there is an additional subset that is used to learn IBM Model 1. The answers from this subset are indexed, but the questions are discarded after learning Model 1 (i.e., they are not used for training and testing). In the case of Manner, IBM Model 1 is trained on a subset of Comprehensive from which we exclude QA pairs that belong to Manner. The split of Manner mimics the setup Surdeanu al. [63] and the test set contains 29K queries. Collection statistics is summarized in Table 4.

<table>
<thead>
<tr>
<th>Collection</th>
<th>QA pairs total</th>
<th>dev1</th>
<th>dev2</th>
<th>tran</th>
<th>Terms in question</th>
<th>Terms in answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manner</td>
<td>142K</td>
<td>86K</td>
<td>7K</td>
<td>21K</td>
<td>29K</td>
<td>4.2M</td>
</tr>
<tr>
<td></td>
<td>Comprehensive</td>
<td>4.4M</td>
<td>212K</td>
<td>11K</td>
<td>42K</td>
<td>2.9M</td>
</tr>
<tr>
<td></td>
<td>Stack Overflow</td>
<td>6.2M</td>
<td>298K</td>
<td>15K</td>
<td>58K</td>
<td>10K</td>
</tr>
</tbody>
</table>

Table 1: Collection statistics. The column tran describes the size of the $BM25+Model 1$ training corpus. Note that for Manner we use an external corpus to train $BM25+Model 1$.

We have implemented multiple retrieval methods and four similarity models (i.e., similarity functions). Retrieval methods, similarity models, and their interactions are summarized in Figure 1. Each

https://github.com/oaqa/knn4qa

https://webscope.sandbox.yahoo.com

We use a dump from https://archive.org/download/stackexchange/ dated March 10th 2016.
retrieval method returns a ranked list of answers, which may be optionally re-ranked. The output is a list of \( N \) scored answers. There are two classes of retrieval methods: term-based retrieval supported by Apache Lucene [5] and the \( k \)-NN search methods implemented in the Non-Metric Space Library (NMSLIB). NMSLIB is an extensible framework for the \( k \)-NN search in generic spaces [7]. Similarity models include:

- **TF×IDF models**: the cosine similarity between TF×IDF vectors (shortly Cosine TF×IDF) and BM25 (§2.1.1).
- The cosine similarity between averaged word embeddings, henceforth, Cosine Embed (§2.1.4).
- The linear combination of BM25 and IBM Model 1 scores, henceforth, BM25+Model 1 (§2.1.3).

In the case of Lucene, we index lemmatized terms and use BM25 as a similarity model [53]. We have found that Lucene’s implementation of BM25 is imperfect (see §2.1.1 for details), which leads to at least a 10% loss in P@1 for both Comprehensive and Stack Overflow. To compensate for this drawback, we obtain 100 top-scored documents using Lucene and re-rank them using our own implementation of BM25.

In the case of NMSLIB, we use two indexing methods and the brute force search. The indexing methods are: the Neighborhood APProXimation index (NAPP) [64] and the Small-World graph (SW-graph) [59]. They are discussed in §2.2. The SW-graph is used only with the Cosine Embed; NAPP is applied to both BM25 and BM25+Model 1. We do not create an index for the Cosine TF×IDF, but use the brute force search instead. The brute force search is slow, but it is applicable to any similarity model.

Because the Cosine TF×IDF and Cosine Embed are not very accurate, the output from these models may be further re-ranked using BM25. To this end, we first retrieve 500 candidate records. Next, we discard all but \( N \) records with highest BM25 scores. To compare effectiveness of Cosine TF×IDF and BM25, we also evaluate a variant where the output of Cosine TF×IDF is not re-ranked.

To re-rank efficiently, we use a forward index. Given a document identifier, this index allows us to quickly retrieve the list of terms and their in-document frequencies. Following a recent trend in high throughput in-memory database systems [31], we load forward indices into memory. The overall re-ranking time is negligibly small.

Note that the depth of a candidate pool represents a reasonable efficiency-effectiveness trade-off. While increasing the depth of the pool improves the answer recall, it also makes it harder to rank results accurately. Beyond a certain point, increasing the depth leads only to a marginal improvement in P@1 at the expense of disproportionately large computational effort.

### 2.1 Similarity Models

#### 2.1.1 Cosine TF×IDF and BM25

Cosine TF×IDF and BM25 are computed for lemmatized text. Cosine TF×IDF is the classic model where the similarity score is equal to the cosine similarity between TF×IDF vectors [59]-[61]. An element \( i \) of such a vector is equal to the product of the unnormalized term frequency \( TF_i \) and the inverse document frequency (IDF). To compute IDF, we use the formula implemented in Lucene:

\[
\ln \left(1 + \frac{(D - d + 0.5)}{(d + 0.5)}\right),
\]

where \( D \) is the number of documents and \( d \) is the number of documents containing the term \( i \).

BM25 scores [53] are computed as the sum of term IDs (Eq. [1]) multiplied by respective normalized term frequencies. The sum includes only terms appearing in both the query and the answer. We also normalize BM25 scores using the sum of query term IDs. Normalized frequencies are as follows:

\[
\frac{TF_i + (k_1 + 1)}{k_1 + 1} 
\]

\[
\text{BM25}_{avg} = \frac{D}{|D|} = \frac{|D|_{avg}}{|D|},
\]

where \( k_1 \) and \( b \) are parameters (\( k_1 = 1.2 \) and \( b = 0.75 \)); \( |D| \) is a document length in words; \( |D|_{avg} \) is the average document length. Lucene’s implementation of BM25 uses a lossy compression for the document length, which results in reduced effectiveness.

#### 2.1.2 IBM Model 1

Computing translation probabilities via IBM Model 1 [10] is one common way to quantify the strength of associations among question and answer terms. The transformed IBM Model 1 scores are used as input to a learning-to-rank algorithm. Specifically, we take the logarithm of the translation probability and divide it by the number of query terms.

Let \( T(q|a) \) denote a probability that a question term \( q \) is a translation of an answer term \( a \). Then, a probability that a question \( Q \) is a translation of an answer \( A \) is equal to:

\[
P(Q|A) = \prod_{q \in Q} P(q|A)
\]

\[
P(q|A) = (1 - \lambda) \sum_{a \in A} T(q|a)P(a|A) + \lambda P(q|C)
\]

\( T(q|a) \) is a translation probability learned by the GIZA++ toolkit [56] via the EM algorithm; \( P(a|A) \) is a probability that a term \( a \) is generated by the answer \( A \); \( P(q|C) \) is a probability that a term \( q \) is generated by the entire collection \( C \); \( \lambda \) is a smoothing parameter. \( P(a|A) \) and \( P(q|C) \) are computed using the maximum likelihood estimator. For an out-of-vocabulary term \( q \), \( P(q|C) \) is set to a small number (10⁻⁹). Similar to BM25 and Cosine TF×IDF, computation is based on the lemmatized text.

A straightforward but slow approach to compute IBM Model 1 scores involves storing \( T(q|a) \) in the form of a sparse hash table. Then, computation of Eq. [3] entails one hash table lookup for each combination of question and answer terms. We can do better by creating an inverted index for each query, which permits retrieving
query-specific entries $T(q|a)$ using the identifier of answer term $a$ as a key. Thus, we need only one lookup per answer term. Identifiers are indexed using an efficient hash table (the class `dense_hash_map` from the package `sparsehash`).

Building such an inverted index is computationally expensive (about 15 ms for each Comprehensive and 90 ms for each Stack Overflow query). Yet, the cost is amortized over multiple comparisons between the query and data points.

We take several measures to maximize the effectiveness of IBM Model 1. First, we compute translation probabilities on a symmetrized corpus as proposed by Jeon et al. [30]. Formally, for every pair of documents $(A, Q)$ in the parallel corpus, we expand the corpus by adding entry $(Q, A)$.

Second, unlike previous work, which seems to use complete translation tables, we discard all translation probabilities $T(q|a)$ below an empirically found threshold of 2.5 · $10^{-3}$. The rationale is that small probabilities are likely to be the result of model overfitting. Pruning of the translation table improves both efficiency and effectiveness. It also reduces memory requirements.

Third, following prior proposals [30, 63], we set $T(w|w')$, a self-translation probability, to an empirically found positive value and rescale probabilities $T(w'|w)$ so that $\sum_w T(w'|w) = 1$.

Fourth, we make an ad hoc decision to use as many QA pairs as possible to train IBM Model 1. A positive impact of this decision has been confirmed by a post hoc assessment.

Finally, we tune parameters on a development set (dev1 or dev2). Rather than evaluating individual performance of IBM Model 1, we aim to maximize performance of the model that linearly combines BM25 and IBM Model 1 scores.

### 2.2 Methods of k-NN Search

We employ a k-NN retrieval framework NMSLIB, which provides several implementations of distance based indexing methods [7]. These indexing methods treat data points as unstructured objects, together with a black-box distance function. In that, the indexing and searching process exploit only values of mutual object distances. NMSLIB can be extended by implementing new black-box “distance” functions. In particular, we add an implementation for the similarity functions BM25, Cosine TF-IDF, Cosine Embed, and BM25+Model 1 (see [21]). None of these similarity functions is a metric distance. In particular, in the case of BM25 + Model 1 the “distance” lacks symmetry.

Because exact k-NN search is too slow to be practical, we resort to an approximate procedure, which does not necessarily find all k nearest neighbors. The accuracy of the k-NN search is measured using a recall denoted as R@k. R@k is equal to the fraction of true k-nearest neighbors found.

NMSLIB reads contents of the forward index (created by a separate indexing pipeline) into memory and builds an additional in-memory index. In this work, we create indices using one of the following methods: the Neighborhood APProximation index (NAPP) due to Tellez et al. [64] or the proximity graph method called a Small-World graph (SW-graph) due to Malkov et al. [59].
NAPP is a pivoting method that arranges points based on their distances to pivots. This is a filtering method: Candidate points share numPivotSearch closest pivots with the query point. The search algorithm employs an inverted index. Unlike term-based indices, however, for each pivot the index keeps references to close data points. More specifically, the pivot should be one of the point’s numPivotIndex closest pivots. Answering a query requires efficient merging of posting lists. Merging of posting lists represents a substantial overhead.

Tellez et al. [65] use pivots randomly sampled from the data set, but we find that for sparse data such as TF×IDF vectors substantially shorter retrieval times—sometimes by orders of magnitude—can be obtained by using a special pivot generation algorithm. Specifically, pivots are generated as pseudo-documents containing $K$ entries sampled from the set of $M$ most frequent words (in our setup $K = 1000$ and $M = 50000$). A more detailed description and analysis of this approach will be presented elsewhere.

During indexing, we have to compute the distances between a data point and every pivot. Because there are thousands of pivots, this operation is quite expensive, especially for BM25+Model 1. To optimize computation of Eq. 3 we organize all pivot-specific $T(q|a)$ entries in the form of the inverted index.

A proximity graph is a data structure, where data points are nodes. Sufficiently close nodes, i.e., neighbors, are connected by edges. Searching starts from some, e.g., random, point/node and traverses the graph until it stops discovering new points sufficiently close to the query or after visiting a given number of nodes. [2][60][26][25][16][70]. Specifically, the SW-graph algorithm (implemented in NMSLIB) keeps a list of efSearch points sorted in the order of increasing distance from the query as well as a candidate queue. Traversal proceeds in the best-first manner, by exploring the neighborhood of the candidate that is closest to the query. If a candidate neighbor is closer to the query than the efSearch-th closest point seen so far, it is added to the candidate queue. Otherwise, the neighbor is discarded. The traversal stops when the candidate queue is exhausted. To improve recall, the algorithm may restart several times. For our data, however, it is more efficient to start from a single point and search using a large-enough value of efSearch.

SW-graph works well for dense vectorial data (i.e., embeddings), where it outstrips NAPP by an order of magnitude. SW-graph was found to be much faster [69] than the multi-probe LSH due to Dong et al. [17]. In a public evaluation in May 2016 [62] SW-graph outperformed two efficient popular libraries: FLANN [48] and Annoy [11]. SW-graph was also mostly faster than a novel LSH algorithm [1]. In contrast, NAPP substantially outperforms SW-graph for sparse TF×IDF data, i.e., for models BM25 and BM25+Model 1.

3. MAIN EXPERIMENTS

Experiments are carried out on Amazon EC2 instance r3.4xlarge, which has 16 virtual cores and 122 GB of memory. The main retrieval pipeline, which is implemented in Java (1.8.0_11), uses 16 search threads. We use a modified version of NMSLIB 1.5 [65] which operates as a server processing queries via TCP/IP. NMSLIB and its extensions are written in C++ and compiled using GCC 4.8.4 with optimization flags -O3 and -march=native. Lucene version is 4.10.3. The retrieval architecture (see §2) is outlined in Figure 1.

The collection processing/indexing system is implemented in Java. It employs the framework Apache UIMA and UIMA components from DKPro Core [19]. Translation probabilities are computed using GIZA++ toolkit [50] via the EM algorithm (five iterations).

For each approximate $k$-NN pipeline, we execute several runs with different parameters. In the case of SW-graph, we vary the parameter efSearch. In the case of NAPP, we vary the number of indexed pivots (parameter numPivotIndex) and the number of pivots that should be shared between the query and an answer (numPivotSearch). Optimal parameters have been found on a dev1 set (using a subset of 5K queries).

Retrieval times are measured by a special client application that submits search requests to either Lucene or NMSLIB. In the case
Table 3: Efficiency-effectiveness trade-offs of retrieval modules for \( N = 100 \) (brute force Cosine TF×IDF runs are omitted). Statistically significant differences (at level 0.01) from BF BM25 are marked with \(*\). P-values are adjusted for multiple testing via the Bonferroni correction.

<table>
<thead>
<tr>
<th>Stack Overflow</th>
<th>Comprehensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query time</td>
<td>Speed-up (over BF)</td>
</tr>
<tr>
<td>BF BM25+Model 1</td>
<td>20.5 s</td>
</tr>
<tr>
<td>NAPP BM25+Model 1</td>
<td>0.84 s</td>
</tr>
<tr>
<td></td>
<td>0.65 s</td>
</tr>
<tr>
<td></td>
<td>0.50 s</td>
</tr>
<tr>
<td></td>
<td>0.47 s</td>
</tr>
<tr>
<td></td>
<td>0.40 s</td>
</tr>
</tbody>
</table>

| BF BM25 | 3.5 s | 0.062 | 0.239 | | | | 1.7 s | 0.067 | 0.239 |
| NAPP BM25 | 0.23 s | 15 | 0.060 | 2.1% | 0.228* | 4.7% | 0.37 s | 5 | 0.063* | 5.1% | 0.227* | 5.0% |
| | 0.14 s | 25 | 0.059* | 5.0% | 0.221* | 7.6% | 0.18 s | 9 | 0.062* | 6.6% | 0.222* | 7.1% |
| | 0.07 s | 50 | 0.057* | 7.6% | 0.208* | 12.9% | 0.15 s | 11 | 0.061* | 8.4% | 0.217* | 9.2% |
| | 0.06 s | 56 | 0.055* | 11.2% | 0.195* | 18.5% | 0.15 s | 11 | 0.060* | 9.3% | 0.212* | 11.2% |
| Lucene BM25 | 0.62 s | 0.062 | 0.229* | | | | 0.08 s | 0.067 | 0.233* |
| BF Cosine Embed | 3.9 s | 0.041* | 0.108* | | | | 2.7 s | 0.055* | 0.174* |
| SW-graph Cosine Embed | 0.78 s | 5 | 0.041* | -0.2% | 0.107* | 0.6% | 0.19 s | 14 | 0.054* | 1.6% | 0.172* | 1.0% |
| | 0.40 s | 10 | 0.041* | 0.5% | 0.107* | 0.8% | 0.09 s | 31 | 0.054* | 3.1% | 0.170* | 1.9% |
| | 0.34 s | 11 | 0.041* | 0.7% | 0.106* | 1.3% | 0.07 s | 37 | 0.053* | 3.6% | 0.170* | 2.4% |
| | 0.13 s | 29 | 0.039* | 4.9% | 0.102* | 5.8% | 0.03 s | 104 | 0.050* | 10.3% | 0.160* | 7.8% |

Note that all BM25-based runs have similar performance. However, BM25+Model 1 has a recall that is 16% higher in the case of Comprehensive and 26% higher in the case of Stack Overflow. BM25+Model 1 is possible to match the recall of BM25+Model 1 by increasing \( N \). However, this may increase a load on a downstream re-ranking module. For example, in the case of Stack Overflow, BM25+Model 1 has a nearly 0.2 answer recall for \( N = 10 \) (Panel 2a in Figure 2). To obtain the same recall level using Lucene, we need to use \( N > 20 \). Next, we compare efficiency of \( k \)-NN search methods against that of Lucene. Note that Lucene is a strong baseline, which fares well against optimized C++ code, especially for disjunctive queries [68]. Lucene’s average retrieval times are equal to 80 ms for Comprehensive and 620 ms for Stack Overflow (see Table 3). There are at least two factors that contribute to the difference in retrieval times between two collections: (1) questions in Stack Overflow have 2.7× as many terms, (2) Stack Overflow has 1.4× as many answers (see Table 1). SW-graph is quite fast for both collections. For example, for Stack Overflow, it can answer queries in 340 ms at the expense of Lucene, we “warm up” the index by executing the whole set of queries twice. Run-times are measured only for the third run. Effectiveness of retrieval runs is measured using an external application, namely, treec_eval 9.0.4 [55]. The main experimental results are presented in Figure 2 and Table 3. For Table 3 we compute statistical significance of results using the t-test with a subsequent Bonferroni adjustment for multiple testing. This adjustment for statistical significance of results using the t-test with a subsequent Bonferroni adjustment for multiple testing consists in multiplying p-values by the total number of runs for \( N = 100 \) (to save space, some of the runs are not shown in the table). The significance level is 0.01.

In Figure 2 we plot the answer recall (measured for a set of top-\( N \) ranked documents) for nine implemented retrieval modules. Note that approximate \( k \)-NN methods are represented by their most accurate runs. Exact brute force \( k \)-NN runs are plotted using thicker lines (with star marks) of the same style/color as corresponding approximate runs. Their mnemonic names start with the word BF (short for brute force).

The cosine-similarity models are the least effective. The recall of the brute force run BF Cosine TF×IDF -rerank is less than half of that for the brute force run BF BM25. We can nearly match the performance of BM25 by adding a BM25-based optional re-ranker (the run BF Cosine TF×IDF +rerank). In contrast, the cosine-similarity between averaged word embeddings (e.g., the run BF Cosine Embed) is much worse than BM25 despite using the re-ranker! Somewhat surprisingly, the cosine similarity between TF×IDF vectors without re-ranker is sometimes more effective than the cosine similarity between word embeddings whose performance is boosted by the re-ranker (see Panel 2a in Figure 2). This is a discouraging finding given that embedding-based retrieval can be quite efficient (see Table 3). It remains to be verified if better results can be obtained with document embeddings that compute vectorial representations of complete sentences or even documents [28, 36].

Note that all BM25-based runs have similar performance. However, BM25+Model 1 has a recall that is 16% higher in the case of Comprehensive and 26% higher in the case of Stack Overflow. For BM25, it is possible to match the recall of BM25+Model 1 by increasing \( N \). However, this may increase a load on a downstream re-ranking module. For example, in the case of Stack Overflow, BM25+Model 1 has a nearly 0.2 answer recall for \( N = 10 \) (Panel 2a in Figure 2). To obtain the same recall level using Lucene, we need to use \( N > 20 \). Next, we compare efficiency of \( k \)-NN search methods against that of Lucene. Note that Lucene is a strong baseline, which fares well against optimized C++ code, especially for disjunctive queries [68]. Lucene’s average retrieval times are equal to 80 ms for Comprehensive and 620 ms for Stack Overflow (see Table 3). There are at least two factors that contribute to the difference in retrieval times between two collections: (1) questions in Stack Overflow have 2.7× as many terms, (2) Stack Overflow has 1.4× as many answers (see Table 1).

SW-graph is quite fast for both collections. For example, for Stack Overflow, it can answer queries in 340 ms at the expense of

10https://github.com/usnistgov/treec_eval
Table 4: Reduction in the number of the distance computation for two similarity models at approximately equal levels of R@1 (larger reduction is better). Using 5K queries from dev1 set.

<table>
<thead>
<tr>
<th></th>
<th>Comprehensive</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BM25+Model 1</td>
<td>BM25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R@1 Reduction in</td>
<td>R@1 Reduction in</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>distance comp.</td>
<td>distance comp.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.982</td>
<td>8.7</td>
<td>0.982</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>0.968</td>
<td>61</td>
<td>0.970</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>0.961</td>
<td>142</td>
<td>0.963</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>0.952</td>
<td>246</td>
<td>0.956</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>0.930</td>
<td>434</td>
<td>0.927</td>
<td>226</td>
<td></td>
</tr>
</tbody>
</table>

|                  |                  |                |                  |                  |
| Stock Overflow   | BM25+Model 1    | BM25           |                  |                  |
| R@1 Reduction in | R@1 Reduction in |                |                  |                  |
| distance comp.   | distance comp.   |                |                  |                  |
| 0.982            | 13.4           | 0.980          | 157              |                  |
| 0.970            | 39             | 0.972          | 208              |                  |
| 0.964            | 64             | 0.964          | 260              |                  |
| 0.957            | 97             | 0.959          | 287              |                  |
| 0.948            | 137            | 0.955          | 315              |                  |

Second, there are differences in filtering effectiveness of the methods. To demonstrate this, we evaluate the reduction in the number of distance computations compared to the brute force search. For example, if an algorithm answers a query by checking only 10% of data points, the reduction in the number of distance computations is 10. Reductions in the number of distance computations are compared for nearly equal values of the k-NN recall R@1, which is equal to the fraction of true nearest neighbors found by the retrieval module. (R@1 should not be confused with the answer recall). The results of this comparison are presented in Table 4. For NAPP, the more pivots are indexed, the fewer distance computations are necessary to achieve a given accuracy level. Thus, to make a fair comparison, we index equal number of pivots for both BM25 and BM25+Model 1.

According to Table 4, in the case of Comprehensive, it takes 2-3\times fewer distance computations for the model BM25+Model 1 than for BM25. In contrast, in the case of Stack Overflow, answering queries for the model BM25 takes significantly fewer distance computations than for BM25+Model 1. Furthermore, the reduction in the number of distance computations for BM25 on Stack Overflow can be two orders of magnitude higher compared to that of Comprehensive. What are the possible explanations for these stark differences?

We think that pivoting methods are effective only if comparing a query and an answer with the same pivot provides a meaningful information regarding their proximity. In the case of a simple BM25 model, this is only possible if the pivot, the query, and the answer have at least one common term. Such an overlap is much more likely in the case of Stack Overflow where questions are nearly 3\times longer compared to Comprehensive. In contrast, for the model BM25+Model 1 information regarding proximity of answers and queries may be obtained if pivots, queries, and answers share only related but not necessarily identical terms. Thus, using BM25+Model 1 is more advantageous in the case of short queries (e.g., in the case of Comprehensive).

To further illustrate importance of using the right function to compute distance to pivots, we evaluate filtering effectiveness in two scenarios: (1) when the distance to pivots is computed using an original distance function and (2) when the distance to pivots is computed using a different, i.e., proxy function. For each scenarios, we use two models: BM25+Model 1 and BM25. In the case of BM25+Model 1, the proxy distance is BM25. In the case of BM25, the proxy distance is Cosine TF\times IDF. The results are presented in Figure 3 where the curves corresponding to the original distance are blue and the curves corresponding to the proxy distance are red.

Panels 3a and 3c show us what happens if the distance to pivots is computed using cheap BM25 instead of expensive BM25+Model 1. We can see that resorting to using the proxy distance makes us check more candidate documents to achieve the same level of recall. In other words relying on the proxy distance has a negative effect on filtering effectiveness. In turn, this can drastically reduce overall search efficiency.

The difference is bigger for Panel 3a which corresponds to the collection Comprehensive. A likely explanation of this difference is the above-described disparity in query lengths between two collections. In the case of Stack Overflow queries are long and there is a bigger overlap between queries and answer documents. This is why the similarity function that relies on a pure lexical match (in this case BM25) allows us to find answers rather effectively. In the case of Comprehensive a lexical overlap between queries and answer documents is less likely, which can be, nevertheless, remedied by enhancing BM25 model with Model 1 scores. However, when Model 1 scores are excluded—by using the proxy distance function to compute distance to pivots—this exclusion has a larger negative effect for Comprehensive than for Stack Overflow.
4. DISCUSSION AND RELATED WORK

The $k$-NN search is an extensively studied area. For a detailed discussion the reader is addressed to the surveys of metric [13] and non-metric [61] access methods, as well to the recent survey of hashing techniques [69].

The $k$-NN search is a popular technique in IR and NLP, where the following two approaches are typically used. The first approach relies on a term-based inverted index in retrieving documents that share common terms with the query. These documents are further re-ranked using some similarity function. Dynamic and static pruning can be used to improve efficiency, sometimes at the expense of decreased recall [67] [11] [15]. This approach supports arbitrary similarity functions, but it suffers from the problem of the vocabulary mismatch [4, 63, 23].

The second approach involves carrying out the $k$-NN search via LSH [53, 65, 46, 37]. It is most appropriate for the cosine similarity. For example, Li et al. [37] propose the following two-stage scheme to the task of finding thematically similar documents. In the first step they retrieve candidates using LSH. Next, these candidate are re-ranked using the Hamming distance between quantized TF×IDF vectors. Li et al. [37] find that their approach is up to $30\times$ faster than the classic term-based index while sometimes being equally accurate.

Petrović et al. [53] applied a hybrid of LSH and the term-based index to the task of the streaming First Story Detection (FSD). The LSH keeps a large chunk of sufficiently recent documents, while the term-based index keeps a small subset of recently added documents. They report their system to be substantially faster than the state-of-

Figure 3: Filtering effectiveness of NAPP for original and a proxy distance function (when computing distances to pivots). The curves for the original distance are blue while proxy distance curves are red. Filtering effectiveness is measured using via reduction in the number of distance computations (larger is better). The left column has data for the distance BM25+Model 1 and the right column has data for BM25. Using 5K queries from dev1 set.

Panels (a) and (c) show us what happens if the distance to pivots is computed using Cosine TF×IDF instead of BM25. In this case, there is a much larger performance deterioration than removal of Model 1 scores. This is not surprising: As we can see from Figure 2 there is a much bigger gap in effectiveness between BM25 and Cosine TF×IDF than between BM25+Model 1 and BM25. Thus, replacing BM25 with Cosine TF×IDF has also a larger negative effective on filtering effectiveness (than replacing BM25+Model 1 with BM25).

The second approach involves carrying out the $k$-NN search via LSH [53, 65, 46, 37]. It is most appropriate for the cosine similarity. For example, Li et al. [37] propose the following two-stage scheme to the task of finding thematically similar documents. In the first step they retrieve candidates using LSH. Next, these candidate are re-ranked using the Hamming distance between quantized TF×IDF vectors. Li et al. [37] find that their approach is up to $30\times$ faster than the classic term-based index while sometimes being equally accurate.

Petrović et al. [53] applied a hybrid of LSH and the term-based index to the task of the streaming First Story Detection (FSD). The LSH keeps a large chunk of sufficiently recent documents, while the term-based index keeps a small subset of recently added documents. They report their system to be substantially faster than the state-of-
the-art system—which relies on the classic term-based index—while
being similarly effective. In a follow up work, Petrović et al. [54]
incorporate term associations into the similarity function. Their
solution relies on an approximation for the kernelized cosine simi-
larly. The associations are obtained from an external paraphrasing
database. Moran et al. [46] use the same method as Petrović et al.
[54], but find synonyms via the k-NN search in the space of word
embeddings (which works better for Twitter data). Moran et al. [46]
as well as Petrović et al. [54] calculate performance using an aggre-
gated metric designed specifically for the FSD task. Unfortunately,
they do not report performance gains using standard IR metrics such
as precision and recall.

Most importantly, as shown in the literature (see [72 and refer-
ences therein), similarity functions based on the cosine similarity
are not especially effective. In particular, compared to BM25, our
implementation of the TF×IDF cosine similarity finds 2× fewer an-
swers for any given rank N (see Figure 2). It is possible to improve
answer recall by increasing N. However, this has effect on search
module’ performance. In particular, if we retrieve top-N entries
using an approximate k-NN algorithm, as N increases, accuracy or
the efficiency of the search decreases. Simply speaking, it is easi-
er to carry out an accurate 1-NN search than an accurate 500-NN
search.

One notable exception is a recent paper by Brokos et al. [9] who,
in contrast to our findings, learned that the cosine-similarity between
averaged word embeddings is an effective model for retrieving Pubmed abstracts. However, they do not compare against standard
IR baselines such as BM25, which makes an interpretation of their
finding difficult.

We argue that instead of relying on the cheap cosine similarity it
may be better to employ an expensive but more accurate similarity
function. The exact brute force search using this function would be
expensive, but the cost could be reduced by applying an approximate
search method for generic—i.e., not necessarily metric—spaces.

A common approach to non-metric space indexing involves pro-
jecting data to a low-dimensional Euclidean space. The goal is to
find a mapping without a large distortion of the original similarity
measure. Jacobs et al. [29] review projection methods and argue that
such a coercion is often against the nature of a similarity measure,
which can be, e.g., intrinsically non-symmetric.

Among other factors, the lack of symmetry prevents us from
using the kernelized LSH [15][17]. The only LSH variant that might
be directly applicable in our case is the Distance-Based Hashing
(DBH) [3], which uses randomly selected pivots to project points
to a one-dimensional space via FastMap [20]. The space is further
binarized so that approximately one half of data points are mapped
to one, and the other half is mapped to zero.

While a detailed comparison of pivoting approaches to DBH
is out the scope of the paper, we hypothesize that performance of
DBH—like performance of NAPP—depends on the choice of pivots.
In the case of NAPP, we have found that composing pivots from
randomly selected terms allows us to achieve substantially better
performance than selecting pivots randomly. Thus, engineering
pivots to support effective searching in a non-metric space seems to
be an important research area. Results obtained from this area will
likely benefit both DBH and NAPP.

Proximity graphs (see §2.2) is another promising class of distance-
based methods, which are shown to be useful in non-metric spaces
[49]. In this work we employ the SW-graph [19], which works quite
well for dense vector spaces. However, it has been less useful for
BM25 and BM25+Model 1. We have not been able to understand
what causes the lack of performance, but this remains an important
research question as well.

Finally, we want to highlight the relationship of our approach
to indexing automatically learned features for QA [76]. Yao et al.
propose to automatically learn associations between a question type
and various linguistic annotations such as named entities and POS
tags [76]. For example, for a question “Who is the president of
the United States” an answer sentence contains a person name (a
named entity). Given a training corpus, we can automatically learn
associations and exploit them to guide the retrieval process. Tech-
nically, this requires indexing linguistic annotations and carrying out
a query expansion by adding expected annotations to the query.

For efficiency reasons, this works well only if we can find few
strong associations for a query. To demonstrate that this is not
true in the case of the vocabulary gap, we compute effectiveness of
BM25+Model 1 for varying sizes of the translation table. Specifically,
we prune all the entries T(q,i) below a threshold. In addition, we
estimate the average number of non-zero translation entries
T(q,i) associated with a single query term. As a reference point
we also include data for BM25. We present results only for Compre-
hensive, because results for Stack Overflow are analogous.

<table>
<thead>
<tr>
<th>Minimum translation probability</th>
<th>P@1</th>
<th>Number of associated terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>BM25</td>
<td>0.065</td>
<td>N/A</td>
</tr>
<tr>
<td>0.1</td>
<td>0.066 (+2.6%)</td>
<td>1700</td>
</tr>
<tr>
<td>0.05</td>
<td>0.070 (+8.6%)</td>
<td>3800</td>
</tr>
<tr>
<td>0.025</td>
<td>0.073 (+12.5%)</td>
<td>6200</td>
</tr>
<tr>
<td>0.005</td>
<td>0.077 (+19.3%)</td>
<td>12000</td>
</tr>
<tr>
<td>0.0025</td>
<td>0.079 (+21.6%)</td>
<td>15000</td>
</tr>
</tbody>
</table>

Table 5: Average number of terms associated with a query term at
various performance levels of BM25+Model 1 (estimated on dev2). The first row represents a BM25 run.

According to Table 5 outperforming BM25 by about 20% re-
quires to keep more than 10K associations per query term (on aver-
age). This number is so high because frequent words, which tend to
appear in queries and text, are associated with many less frequent
words (i.e., respective translation probabilities are non-zero).

If we keep only translation entries with high (≥ 0.1) probabilities,
the improvement over BM25 is merely 2.6%. Yet, we still have to
keep nearly 2K associations per query term! This further corrobo-
rates the finding of Furnas et al. [24] that accurate retrieval requires
using a large number of term aliases, which is hard to implement us-
ing term-based indices. Yet, it is possible to do within a framework
of the k-NN search.

That said, the proposed methods are likely have limitations as
well. For example, for both data sets employed in our experiments,
the queries are quite long. It is not yet clear if k-NN can be applied
to shorter ad hoc queries, which are frequently submitted to Web
search engines.

5. CONCLUSION

In this paper we attempt to replace the classic term-based retrieval
with the k-NN search. To this end, we train a linguistically moti-
vated non-metric and non-symmetric similarity function: a weighted
combination of BM25 scores and IBM Model 1 log-scores. Then,
we demonstrate that it is possible to carry out an efficient and effect-
ive approximate k-NN search using this function.

An exact brute-force k-NN search using this similarity function
is slow. Yet, an approximate algorithm can be nearly two orders
of magnitude faster at the expense of only a small loss in accu-
racity. A retrieval pipeline using an approximate k-NN search can
be sometimes both faster and more accurate compared to the term-based Lucene pipeline (see Table 3). The success of our approach stems from the novel combination of existing methods and new algorithmic tricks to compute IBM Model 1 efficiently.

While the k-NN search has been previously applied to IR and NLP problems, the previous work focuses largely on the cosine similarity and LSH methods (see §4[1] for a discussion). This is the first successful attempt to apply a generic k-NN search algorithm to a similarity function as challenging as a combination of BM25 and IBM Model 1. In that, we find that the cosine similarity alone (in particular, the cosine similarity between averaged word embeddings) lacks a lot in effectiveness (see Figure 2).

The focus of our study is on techniques that bridge the vocabulary gap. Yet, our methods are generic in the sense that they can be used to model various types of semantic and syntactic mismatch [6, 76]. This opens up new possibilities for designing effective retrieval pipelines.

Our software (including data-generating code) and derivative data based on the Stack Overflow collection is available online.

Acknowledgements
Leonid Boytsov is supported by the Open Advancement of Question Answering Systems (OQA) group. David Novak is supported by the Czech Research Foundation project P103/12/G084. Yury Malkov is supported by the Russian Foundation for Basic Research (project No. 16-31-00140 mol_a_dk).

We also thank Di Wang for helping with a Lucene baseline; Chris Dyer for a discussion of IBM Model 1 efficiency; Yoav Goldberg, Manaal Faruqui, Chenyan Xiong, Ruey-Cheng Chen for discussions related to word embeddings; Michael Denkowski for a discussion on approximating alignment scores.

6. REFERENCES


