Engineering Efficient and Effective Non-Metric Space Library

Leonid Boytsov[1] and Bilegsaikhan Naidan[2]

[1] Language Technologies Institute,
Carnegie Mellon University,
Pittsburgh, PA, USA
leo@boytsov.info
[2] Department of Computer and Information Science,
Norwegian University of Science and Technology,
Trondheim, Norway
bileg@idi.ntnu.no

Abstract. We present a new similarity search library and discuss a variety of design and performance issues related to its development. We adopt a position that engineering is equally important to design of the algorithms and pursue a goal of producing realistic benchmarks. To this end, we pay attention to various performance aspects and utilize modern hardware, which provides a high degree of parallelization. Since we focus on realistic measurements, performance of the methods should not be measured using merely the number of distance computations performed, because other costs, such as computation of a cheaper distance function, which approximates the original one, are oftentimes substantial. The paper includes preliminary experimental results, which support this point of view. Rather than looking for the best method, we want to ensure that the library implements competitive baselines, which can be useful for future work.

Keywords: benchmarks, (non)-metric spaces, Bregman divergences

1 Introduction

A lot of domains, including content-based retrieval of multimedia, computational biology, and statistical machine learning, rely on similarity search methods. Given a finite database of objects $\{o_i\}$, a search query q and a dissimilarity measure (which is typically represented by a distance function $d(o_i, q)$), the goal is to find a subset of database objects sufficiently similar to the query q.

Two major retrieval tasks are typically considered: a nearest neighbor and a range search. The nearest neighbor search aims to find the least dissimilar object, i.e., the object at the smallest distance from the query. Its direct generalization is the k-nearest neighbor (or the k-NN) search, which looks for the k most closest objects. Given a radius r, the range query retrieves all objects within a query ball (centered at the query object q) with the radius r, or, formally, all the objects $\{o_i\}$ such that $d(o_i, q) \leq r$.

The queries can be answered either exactly, i.e., by returning a complete result, or, approximately, e.g., by finding only some nearest neighbors. The exact versions of near neighbor and range search received a lot of attention. Yet, in many applications exact searching is not essential, because the notion of similarity, e.g., between two images, is not specified rigorously. Applying an exact retrieval method does not necessarily mean that we will find the image that is most similar to a query from a human perspective. Likewise, a k-NN classifier may perform well even if the search method does not produce a precise and/or a complete result [4,31].

Search methods for non-metric spaces are especially interesting. This domain does not provide sufficiently generic *exact* search methods. We may know very little about analytical properties of the distance or the analytical representation may not be available at all (e.g., if the distance is computed by a black-box device [35]). Hence, employing an approximate approach is virtually unavoidable.

Approximate search methods are typically more efficient than exact ones. Yet, it is harder to evaluate them, because we need to measure retrieval speed at different levels of recall (or any other effectiveness metric). To the best of our knowledge, there is no publicly available software suit that (1) includes state-of-the-art approximate search methods for both non-metric and metric spaces and (2) provides capabilities to measure search quality. Thus, we developed our own test framework and presented it in this paper.

1.1 Related Work

There is large body of literature devoted to exact search methods in metric spaces, which are thoroughly surveyed in the books by Faloutsos [12], Samet [32], and Zezula et al. [40] (see also a survey by Chávez et al. [5]). Exact methods have a limited value in high-dimensional spaces, which exhibit phenomena of the empty space [33] and measure concentration [5]. Experiments show that, as the dimensionality increases, every nearest neighbor search method degrades to sequential searching [39]. This is commonly known as the "curse of dimensionality". In that, methods, which are allowed to return inexact answers, are less affected by the curse [30]. For a discussion of these phenomena, we address the reader to the papers of Indyk [20] and Pestov [30].

To answer the approximate nearest neighbor queries, Indyk and Motwani [21] as well as Kushilevitz et al. [25] proposed to use random projections. The locality sensitivity hashing (LSH) is one of the most well-known implementations of this idea [21,20]. The LSH indexing uses several hash functions, such that a probability of a collision (hashing to the same value) is sufficiently high for close objects, but is small for distant ones.

The LSH works best in L_p spaces where $p \in (0, 2]$. There exists an extension of the LSH for an arbitrary metric space [28] as well as for *symmetric* non-metric distances [27]. Performance of the LSH depends on the choice of parameters, which can be tuned to fit the distribution of a data set [7].

Most exact search methods can be transformed into approximate ones by applying an *early termination strategy*. In particular, Zezula et al. [41] demon-

strated that this approach works well for M-trees. One of the most efficient strategies relies on density estimates for a distribution of distances [41,1]. The density-based approach to space pruning was also discussed by Chávez and Navarro [6] (in the context of pivoting methods), who called it "stretching" of the triangle inequality.

Let us consider a metric space, where we selected a single reference point π, known as *pivot*. The pivot is used to prune space during searching. Imagine that we computed a distance from the pivot π to every other data point. Then, points are sorted in the order of increasing distances from π. The median distance is m and points are divided into two buckets. If the distance from a point to π is smaller than m, the point is put into the first bucket. Points with distances larger than (or equal to) m are placed into the second bucket.

Let q be a query point and r be a radius of the range query. If $r < m - d(\pi, q)$, an answer can be only in the first bucket. If $r \leq d(\pi, q) - m$, an answer can be only in the second bucket. Otherwise, the answer can be in both buckets and no pruning is possible (without risking to miss an answer). In the "stretched" triangle inequality, we choose constants $\alpha_1, \alpha_2 \geq 1$.[3] If $r < \alpha_1(m - d(\pi, q))$, we check only the left bucket. If $r < \alpha_2(d(\pi, q) - m)$, we check only the right bucket. This is an example of an *oracle* procedure that defines a pruning algorithm of a pivoting method. Note that (1) it is possible to learn the oracle in both metric and *non-metric* spaces, (2) we can learn a pivot-specific oracle, instead of the global one, (3) most existing methods designed for metric spaces can be converted into non-metric search methods by simply replacing the triangle-inequality based pruning method with a search oracle. We plan to present these learning approaches in detail elsewhere.

In a recent survey [36], Skopal and Bustos discussed several types of non-metric access methods, which we divide into the following categories: (1) projective and lower/upper bounding approaches, (2) methods that prune the space using properties other than the triangle inequality (e.g., the Ptolemaic inequality [26]), and (3) domain-specific methods. Inverted files are a classic domain-specific algorithm applicable to high-dimensional, but sparse, vector spaces, where the distance function is the cosine similarity (or a similar distance).

Jacobs et al. [22] review various projection methods and argue that a projection is not always feasible, for instance, when the similarity cannot be expressed by a numeric distance function, or the distance function is not symmetric. In the case of symmetric, non-negative, and reflexive distance, one can use the TriGen algorithm [35], which applies a monotonic transformation to the distance function. Consider, e.g., the squared Euclidean distance, which is a (non-metric) Bregman divergence. By taking the square root, we obtain the metric function. Similarly, the TriGen algorithm allows one to convert a distance into a function that satisfies the triangle inequality only approximately. In addition, it provides control over the degree of approximation.

Chávez et al. [18] proposed a projective method, which is applicable to both metric and *non-metric spaces*. The method, called the permutation index, selects

[3] Chávez and Navarro [6] employed only one stretching constant.

k pivots $\{\pi_i\}$ and for every data point o it creates a permutation of pivots: a list where pivots are sorted in the order of increasing distances $d(\pi_i, o)$. Independently, this method was invented by Amato and Savino [2], who additionally proposed to index permutations using an inverted file.

To answer the query, the correlation is computed between the permutation of the vector and the permutation of every data point. Then, all data points are sorted in the order of ascending correlation values and a given fraction of objects are compared directly with the query (by computing the distance in the original space). Performance of the permutation index can be improved by using incremental sorting [17] or by indexing permutations using an inverted file [2], a permutation prefix tree [11], or a metric space index [14].

Bregman divergences is a class of non-metric distance functions. This divergences include the squared Euclidean distance, the KL-divergence:

$$d(x, y) = \sum x_i \log(x_i/y_i) \tag{1}$$

and the Itakura-Saito distance:

$$d(x, y) = \sum x_i/y_i - \log(x_i/y_i) - 1. \tag{2}$$

For the Bregman divergences, there exist two exact search methods. The Bregman ball tree (bbtree) [4], which recursively divides the space using two covering Bregman balls at each recursion step, and a mapping method due to Zhang et al. [42]. Both approaches use properties of Bregman divergences to lower/upper bound distance values.

2 Methodology

2.1 Evaluation Approach

Performance of approximate methods is typically represented by a curve that plots efficiency against effectiveness. Two most common efficiency metrics are retrieval time and a number of distance computations. Additionally, we use the *improvement in efficiency* (with respect to the single-thread sequential search algorithm) and the *improvement in the number of distance computations*.

Recall is a commonly used effectiveness metric. It is equal to the fraction of all correct answers retrieved. The *relative error* [41] is defined for a pair of points o and $\widetilde{o}$, such that o is an exact and $\widetilde{o}$ is an approximate answer. It is simply a ratio of the distances $d(\widetilde{o}, q)$ and $d(o, q)$. The relative error can be misleading, especially in high dimensional spaces. Due to high concentration of measure, an increase in relative error can be very small, but the method can return the 1,000th nearest-neighbor instead of the most closest one. This concern was also expressed by Cayton [4]. Similarly, recall does not account for position information and has the same issue [1].

Let $\text{pos}(o_i)$ represent a positional distance from o_i to the query, i.e., the number of objects closer to the query than o_i plus one. In the case of ties, we

assume that the object with a smaller index is closer to the query. Note that $\text{pos}(o_i) \geq i$. A *relative position* error is equal to $\text{pos}(o_i)/i$ and is more informative than a relative distance error and/or recall. We average relative position errors using the geometric mean [23].

Zezula et al. [41] proposed to use the average value of the inverse relative position error (called the precision of approximation) as a performance metric (m is the number of found objects):

$$\frac{1}{m}\sum_{i=1}^{m}\frac{i}{pos(o_i)} \tag{3}$$

Amato et al. [1] suggested the metric that measures the absolute position error. It is equal to:

$$\frac{1}{m}\sum_{i=1}^{m}\frac{pos(o_i)-i}{\#\text{of indexed points}} \tag{4}$$

Unfortunately, this metric produces results that are not comparable across collections and result sets of different sizes. Consider an example of the result set, where $pos(o_i) = 2i$. The absolute position error is equal to:

$$\frac{1}{m}\sum_{i=1}^{m}\frac{2i-i}{\#\text{of indexed points}} = \frac{0.5(m+1)}{\#\text{of indexed points}}$$

We have no good explanation why the position error should grow with m, while the relative position error and the degree of approximation remain constant (in this case). Even worse, due to the large factor in the denominator of Eq. 4, the computed error is generally very small. It is easy to make a wrong conclusion that the algorithm works almost ideally, whereas, in truth, it provides a poor approximation.

If we have a separate test set, testing is straightforward. Otherwise, we need to randomly divide the original data set into indexable data and testing data. This method is based on the assumption that distributions of test queries and indexed data objects are similar. The random division should be repeated several times (an approach known as bootstrapping), and performance metrics computed for each split should be aggregated.

One may be tempted to select queries among indexed data objects, or, alternatively, to create test vectors by randomly perturbing the indexed data. Both approaches are not ideal and can lead to overly optimistic or pessimistic results, especially, in the case of the nearest neighbor searching. We experimented with the `Colors` data set[13], indexed using the Vantage Point tree (VP-tree) [37]. If we selected queries from the vectors that were already indexed, it took on average only 20 distance computations to find the query's nearest neighbor. Since the query and the found vector were identical, the pruning algorithm was unrealistically efficient. For the randomly selected held-out test data, it took about 6,000 distance computations to answer the nearest neighbor query! If we used a query obtained by random additive (and uniform) perturbations of vector elements, the results depended on the amount of noise. In our experiment, we got

800 distance computations in one case and 10^5 distance computations (i.e., the algorithm degraded to the linear scan) in another.

We speculate that queries obtained by random perturbations can be useful if the model of random perturbations fits data well. This assumption is apparently reasonable for the Euclidean data, but additive transformations may significantly change the histogram of distances in the case of the KL-divergence. In one example, the application of the additive noise led to a 2x decrease in the median distance value between two randomly selected vectors. The *multiplicative log-normal* noise seemed to produce more realistic results, yet, additional experimentation is needed to understand applicability of this approach.

2.2 Choice of Programming Language

C,C++, and Java are the three most popular general-purpose programming languages[24]. [4] The authors are familiar with all three and considered them as implementation languages. According to "The Computer Language Benchmarks Game", C and C++ have comparable performance.[5] Major C/C++ compilers (GNU C++ and Microsoft Visual C) support Single Instruction Multiple Data (SIMD) commands, which allows one to compute distances more efficiently.

Yet, only C++ supports run-time and compile-time polymorphism. The new C++ specifications standardize multi-threading and simplify the use of STL containers (threads are not standardized in the pure C). [6] There is evidence, including anecdotal experience of authors, that C++ allows programmers to be more productive than does C [3]

Even though performance of Java sometimes matches performance of C++ [38,34], Java is generally 2-3 times slower than C or C++ [19,16]. Unlike C/C++, there is no built-in support for the SIMD instructions [29]. Java objects are heavy and programmers have to use parallel arrays as well as manual memory management (e.g., reusing small objects) to work around this problem [10]. Thus, writing "algorithmic-intensive" applications in Java may sometimes be harder than in C++.

Because C++ is largely a superset of C, reusing the code already implemented in the Metric Spaces Library would be straightforward. Yet, it is harder to port C-code to Java. There are tools for seamless integration of C++ and R. In particular, one can call R scripts directly from a C++ program [9]. All in all, using the latest C++ compiler that implements the new standard is the most appealing choice for us.

---

[4] See, also `http://www.langpop.com/` and `http://spectrum.ieee.org/at-work/tech-careers/the-top-10-programming-languages`

[5] According to at least this page: `http://benchmarksgame.alioth.debian.org/u64/benchmark.php`

[6] See `http://www.open-std.org/jtc1/sc22/wg21/`

2.3 Design

Our software was designed in the spirit of the Metric Spaces Library [13], but there are multiple important differences. We have classes that represent an `Object`, a `Space`, and an `Index`. The `Space` abstraction is necessary to encapsulate the computation of the distance. We can have multiple `Space` sub-classes implementing different distance functions. In addition, the `Space` class provides functionality to read objects from a file.

A distance can be integer-valued, real-valued, or represented by an arbitrarily complex object (if, e.g., we compare objects using multiple criteria). Similarly to the Metric Spaces Library, the same implementation can work with different distance types (e.g., the VP-tree can be used with both the integer-valued edit distance and with the real-valued L_1 metric). This is effectively supported by the compile-time polymorphism of C++ (templates). All implementations (including indices for real-valued and integer-valued distances) co-exist in the same binary and there is no need to update makefiles when a new method or a distance is implemented.

The `Object` has an identifier and can store arbitrary data (of any type). When necessary objects are transformed: One may need to reduce the dimensionality or precompute the logarithms to accelerate evaluation of the KL-divergence (see Section 2.4). Unlike the Metric Spaces Library, a distance function accepts pointers to the objects rather than object ids.

A `Query` object proxies distance evaluations during search time, which allows us to get the average number of computations carried out by a search method as well as to compute confidence intervals (even in the *multi-threading* testing mode). It is still possible to access the distance function through a `Space` object, but this should be done only at indexing time. If (due to programmer's error) an instance of the `Index` tried to access distance through the `Space` object, the program would terminate. [7]

There are two types of query classes and both classes have the `Radius` function. For the range queries, this function returns the constant value specified by the user. For the k-NN queries, the value returned by `Radius` changes during the search, because it represents the distance from the query to the k-th closest object found so far. Because of this abstraction, it is often sufficient to implement a single (template) function that handles both the range and the k-NN search.

The distance function can be non-symmetric, thus two types of queries (left and right are possible). Currently, the framework directly supports only the left queries (q is the second argument of the distance function). For some methods, e.g., permutation-based approaches, right queries can be implemented by simply swapping arguments of the distance function. Yet, a different distance function as well as a transformation of original data may be necessary for Bregman divergences [4]. We plan to address this issue in the future.

[7] We actually have two versions of the `Space` distance function. The public, restricted, version "knows" the current phase (indexing or searching). It terminates the program if called during the search phase. The unrestricted, but private, function is accessible by a `Query` object at search time, because the latter is a friend of `Space`.

As explained in Section 1.1, one can use the concept of the search oracle to convert metric access methods into non-metric ones. We implement two oracle classes (one is based on sampling and another on "stretching" of the triangle inequality). Currently, only the VP-tree can use the generic search oracle, but we plan to embed the search oracle into other metric indices. In addition, most tree-based methods in our library implement a simple early termination strategy, where the search stops after visiting a given number of buckets.

There is a special function that governs the test process. It creates a `Space` object, which loads a data set into memory, divides the data into testing and training sets or loads a separate test set (see Section 2.1). Then, the `Factory` creates instances of specific methods. Parsing of method-specific command line parameters, though, is delegated to the `Index`. Search methods explicitly return pointers/ids of found objects. Thus, we can verify methods' correctness, as well compute recall and other effectiveness metrics discussed in Section 2.1.

Because there are no exact search methods for generic non-metric spaces, evaluation involves comparing a query object against every object in the database. This expensive procedure can be optimized: When we test several different methods in a single session, we compute exact answers only once for each query object. This is reasonably fast on our current data sets, but in the future we may memorize answers, so that they can be re-used when we run multiple tests (using the same data set).

The testing module saves evaluation results to a CSV-file and produces a human readable report. Note that the plots in Section 3 are produced by a Python script that read and processed such CSV-files.

We decided to focus on memory-resident indices. On one hand, modern servers have plenty of memory and a typical high-performance search application would keep most of its index in memory. On the other hand, we do not have to implement serialization/de-serialization or, the code that searches data stored on disk. This simplification allows us to be more productive coders. Our implementations create essentially static indices from scratch. In the future, we plan to consider incremental indexing approaches as well.

One purpose of serialization is to estimate space requirements. Yet, it is possible to obtain an approximate size of the index by measuring the amount of memory used by the program before and after the index is created (one should also include memory to store data objects). There is an opinion that better estimates can be achieved, if we compute the size of allocated memory ourselves, by writing the special code that traverses the index and measures the size of atomic index elements (such as vectors). Yet, we believe that this approach is error prone.

2.4 Efficiency Issues

Even though some distance functions are expensive, it can be quite cheap to compare two vectors using an L_p norm. Furthermore, it can be done even faster using special SIMD instructions [15]. Currently, most x86 CPUs support operations with 128-bit registers containing, e.g., 4 single-precision or 2 double-precision

Table 1: The number of computations per second for optimized and unoptimized distance functions.

	128 elements				1024 elements			
Distance	L_1	L_2	Itakura-Saito	KL-div.	L_1	L_2	Itakura-Saito	KL-div.
C++ (no logs)	$9.6 \cdot 10^6$	$9.1 \cdot 10^6$	$1.9 \cdot 10^5$	$5.3 \cdot 10^5$	$1.2 \cdot 10^6$	$1.2 \cdot 10^6$	$2.4 \cdot 10^4$	$6.7 \cdot 10^4$
SIMD (precomp. logs)	$2.7 \cdot 10^7$	$3.3 \cdot 10^7$	$8.3 \cdot 10^6$	$2.8 \cdot 10^7$	$3.4 \cdot 10^6$	$4.5 \cdot 10^6$	$1.04 \cdot 10^6$	$2.4 \cdot 10^6$

Note: vector elements are randomly, uniformly, and independently drawn from $(0, 1]$

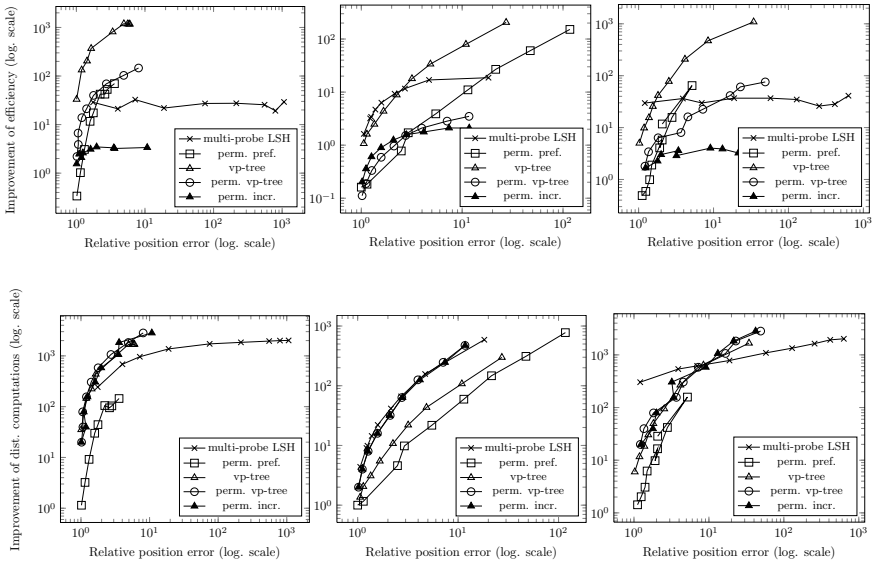
numbers. Some CPUs already support operations with 256-bit registers, which can process 8-element vectors of single-precision numbers.[8] This fact is rather well known, but it appears to be underappreciated. In addition, evaluation of some distance functions can be accelerated at the expense of higher storage requirements (or by dimensionality reduction). In the case of the KL-divergence and the Itakura-Saito distance we can precompute and memorize logarithms of vector elements.

According to Table 1, a single CPU core can carry out more than 30 million computations of the Euclidean distance between two 128-element vectors and more than 4 million distance computations between two 1024-element vectors. In that, the efficient SIMD version spends about one CPU cycle per vector element. [9] The optimized versions of the L_1, L_2 and distances, which use SIMD, are 3 times faster than pure C++ versions. The optimized versions of the KL-divergence and of the Itakura-Saito distance are about 30-50 times as fast as the original ones. In comparison, for a data set of dimensionality 128, the bbtree, which is designed to search using the KL-divergence, is only 5 times faster than sequential scan [4]. It should now be clear that (1) distance computations are not necessarily expensive and (2) optimizing computation of the distance function can be more important than designing data structures.

It has been claimed that a random memory access may take hundreds of CPU cycles [8]. Yet, our experiments showed the cost of a random access on our server to be only 60 cycles. Thus, reducing memory fragmentation may not necessarily lead to substantial improvements in performance. In particular, storing vectors of a VP-tree bucket in adjacent memory regions did not allow us to get more than a 2x speedup. Perhaps, more importantly, the SIMD-based algorithms of distance computations are so fast that communications with RAM can become a major bottleneck in a multi-threading environment. Indeed, to sustain the processing speed of one vector element per CPU cycle (see Table 1) we need to read from memory at the speed of $\approx$ 12 GB/sec (one element is a 4-byte single-precision number). Our server's memory bandwidth of 20 GB/sec can be exhausted with just two threads.

[8] See, e.g., `http://software.intel.com/en-us/avx`

[9] Reading unaligned data does not apparently hurt performance, even for SIMD operations.



(d) `Colors`, L_2 (e) `Unif-64`, L_2 (f) `RCV-128`, L_2

Fig. 1: Improvement in efficiency and in the number of distance computations for 1-NN search in L_2.

3 Experiments

Experiments were carried out on a Linux server equipped with Intel Core i7 2600 (3.40 GHz, 8192 KB of L3 CPU cache) and 16 GB of DDR3 RAM (transfer rate 20GB/sec). The code was compiled using GNU C++ 4.7 (optimization flag -Ofast) and tested in a single-thread (using 1,000 queries). The library can be downloaded from GitHub.[10]

The following collections were used:

1. `Colors`: 112-dimensional data set from the Metric Spaces Library [13];
2. `Unif64`: 64-dimensional vectors with elements generated randomly, independently, and uniformly;
3. `RCV-16` and `RCV-128`: 16- and 128-dimensional topic histograms [4];
4. `SIFT`: the normalized 1111-dimensional SIFT signatures [4].

We extracted the first 10^5 vectors from collections (1)-(3) and used the whole collection (4), which contained only 10^4 vectors.

We carried out two series of experiments (both involving 1-NN search). In the first series (see Fig. 1), we used collections `Colors`, `Unif-64`, and `RCV-128`. The distance was Euclidean. We measured both the improvement in efficiency and in the number of distance computations. The values of efficiency metrics were

[10] `https://github.com/searchivarius/NonMetricSpaceLib`

plotted against the relative position error. In the second experimental series (see Fig. 2), we measured how the improvement in efficiency corresponded to the relative position error. Two Bregman divergences were used: the KL-divergence (see Eq. 1) and the Itakura-Saito distance (see Eq. 2). Implemented methods included domain specific and permutation-based approaches as well the VP-tree.

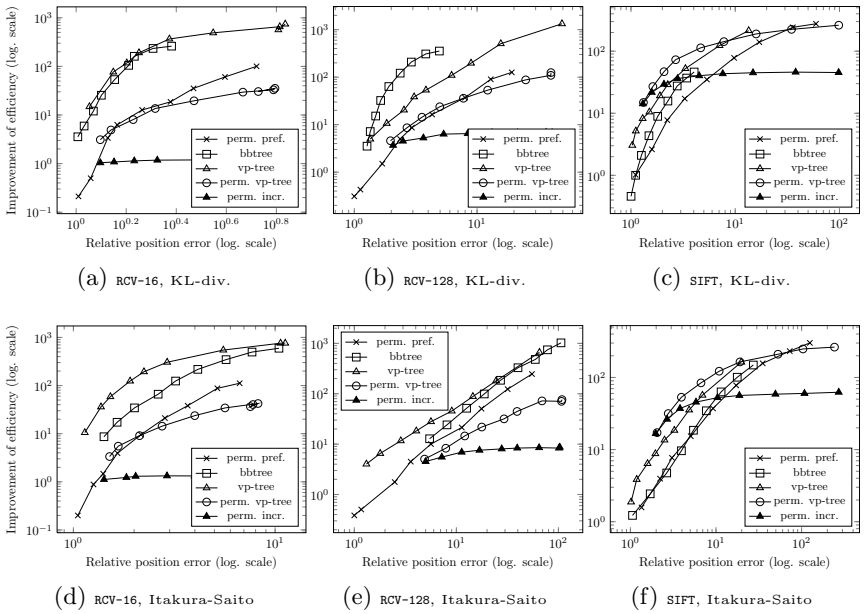
- The VP-tree employed the search oracle that "stretched" the triangle inequality (see Section 1.1). Optimal stretching coefficients were found using a simple grid search. We indexed a small database sample ($\approx$ 1,000 vectors), executed the 1-NN search for various values of stretching coefficients and measured performance. Then, we selected coefficients resulting in the fastest search at given recall values.
- Permutation-based approaches were: an improved permutation index with incremental sorting [17], a permutation prefix tree [11], and the method where permutations were indexed using a metric space index, as proposed by Figueroa and Fredriksson [14]. Unlike Figueroa and Fredriksson, we used an approximate method (the VP-tree that stretched the triangle inequality using $\alpha_1 = \alpha_2 = 2$). In all cases, we used 16 pivots and the prefix length was 4. The maximum fraction of the objects exhaustively compared against the query depended on the data set and varied from 0.01 to 0.05. The minimum fraction of the database objects to be scanned was 0.0002. The number of candidate objects in the permutation prefix index varied from 1 to 24,000.
- The bbtree [4] is the exact indexing method for Bregman divergences. It was extended by the early termination strategy, where the search stopped after visiting a certain number of buckets (the number varied from one to 1,000).
- The multi-probe LSH is designed only for L_2. We used the LSHKit implementation with the following parameters: $H = 1017881$, $T = 10$, $L = 50$.[11]

All methods, including the multi-probe LSH, relied on optimized distance functions. The correlation function (Spearman's rho) was also optimized and implemented using SIMD instructions. The vectors in the buckets of the VP-tree and bbtree were stored in contiguous chunks of memory (the bucket size was 50).

From Fig. 1 we learn that both the classic permutation method (without the index over permutations) and the multi-probe LSH carried out fewer distance computations than most other methods. Yet, they were generally outperformed by the VP-tree and the methods that index permutations (using either the prefix tree or the VP-tree). The reason is that exhaustive comparison of data-object permutations against the permutation of the query vector is costly. In that, the permutation index worked better for high-dimensional data (see Fig. 2f and 2c). Again, we see that the number of distance computations is not necessarily a good predictor of method's performance. Yet, it may give insights into scalability of methods with respect to the size of the data set and data dimensionality.

As can be seen from Fig. 2, we implemented strong baselines that worked well in *non-metric* spaces with *non-symmetric* distance functions. Note that the bbtree, which was tailored to spaces with Bregman divergences, was outperformed

[11] Remaining parameters were automatically computed by the LSHKit [7].



(a) RCV-16, KL-div. (b) RCV-128, KL-div. (c) SIFT, KL-div.

(d) RCV-16, Itakura-Saito (e) RCV-128, Itakura-Saito (f) SIFT, Itakura-Saito

Fig. 2: Improvement in efficiency of 1-NN search for the KL-divergences and Itakura-Saito distance.

by the VP-tree (which is a generic method) in most cases. These are encouraging results, but more work needs to be done. We plan to employ new complex domains and implement additional search methods.

Acknowledgments. We would like to thank Lawrence Cayton for providing the data sets, Vladimir Pestov for the discussion on the curse of dimensionality, and anonymous reviewers for helpful suggestions.

References

1. Amato, G., Rabitti, F., Savino, P., Zezula, P.: Region proximity in metric spaces and its use for approximate similarity search. ACM Trans. Inf. Syst. **21**(2) (April 2003) 192–227
2. Amato, G., Savino, P.: Approximate similarity search in metric spaces using inverted files. In: Proceedings of the 3rd international conference on Scalable information systems. InfoScale '08, ICST, Brussels, Belgium, Belgium, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2008) 28:1–28:10
3. Bhattacharya, P., Neamtiu, I.: Assessing programming language impact on development and maintenance: a study on C and C++. In: Software Engineering (ICSE), 2011 33rd International Conference on. (2011) 171–180

4. Cayton, L.: Fast nearest neighbor retrieval for bregman divergences. In: Proceedings of the 25th international conference on Machine learning. ICML '08, New York, NY, USA, ACM (2008) 112–119
5. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquin, J.L.: Searching in metric spaces. ACM Computing Surveys **33**(3) (2001) 273–321
6. Chávez, E., Navarro, G.: Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. Information Processing Letters **85**(1) (2003) 39–46
7. Dong, W., Wang, Z., Josephson, W., Charikar, M., Li, K.: Modeling lsh for performance tuning. In: Proceedings of the 17th ACM conference on Information and knowledge management. CIKM '08, New York, NY, USA, ACM (2008) 669–678
8. Drepper, U.: What every programmer should know about memory (2007) `http://www.akkadia.org/drepper/cpumemory.pdf` [Last checked August 2012].
9. Eddelbuettel, D., Francois, R.: Rcpp: Seamless R and C++ integration. Journal of Statistical Software **40**(8) (4 2011) 1–18
10. Elizarov, R.: Millions quotes per second in pure Java (2013) `http://blog.devexperts.com/millions-quotes-per-second-in-pure-java/` [Last accessed on May 14th 2013].
11. Esuli, A.: Use of permutation prefixes for efficient and scalable approximate similarity search. Inf. Process. Manage. **48**(5) (September 2012) 889–902
12. Faloutsos, C.: Searching Multimedia Databases by Content. Kluwer Academic Publisher (1996)
13. Figueroa, K., Navarro, G., Chávez, E.: Metric Spaces Library (2007) Available at `http://www.sisap.org/Metric_Space_Library.html`.
14. Figueroa, K., Fredriksson, K.: Speeding up permutation based indexing with indexing. In: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications. SISAP '09, Washington, DC, USA, IEEE Computer Society (2009) 107–114
15. Fredriksson, K.: Engineering efficient metric indexes. Pattern Recognition Letters **28**(1) (2007) 75 – 84
16. Fulgham, B.: The computer language benchmarks game (2013) `http://benchmarksgame.alioth.debian.org/` [Last accessed on May 14th 2013].
17. Gonzalez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. Pattern Analysis and Machine Intelligence, IEEE Transactions on **30**(9) (2008) 1647–1658
18. Gonzalez, E.C., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. Pattern Analysis and Machine Intelligence, IEEE Transactions on **30**(9) (2008) 1647–1658
19. Hundt, R.: Loop recognition in C++/Java/Go/Scala. Proceedings of Scala Days **2011** (2011)
20. Indyk, P.: Nearest neighbors in high-dimensional spaces. In Goodman, J.E., O'Rourke, J., eds.: Handbook of discrete and computational geometry. Chapman and Hall/CRC (2004) 877–892
21. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on Theory of computing. STOC '98, New York, NY, USA, ACM (1998) 604–613
22. Jacobs, D., Weinshall, D., Gdalyahu, Y.: Classification with nonmetric distances: Image retrieval and class representation. Pattern Analysis and Machine Intelligence, IEEE Transactions on **22**(6) (2000) 583–600
23. King, G.: How not to lie with statistics: Avoiding common mistakes in quantitative political science. American Journal of Political Science (1986) 666–687

24. King, R.S.: The top 10 programming languages [the data]. Spectrum, IEEE **48**(10) (2011) 84–84
25. Kushilevitz, E., Ostrovsky, R., Rabani, Y.: Efficient search for approximate nearest neighbor in high dimensional spaces. In: Proceedings of the 30th annual ACM symposium on Theory of computing. STOC '98, New York, NY, USA, ACM (1998) 614–623
26. Lokoč, J., Hetland, M.L., Skopal, T., Beecks, C.: Ptolemaic indexing of the signature quadratic form distance. In: Proceedings of the Fourth International Conference on SImilarity Search and APplications. SISAP '11, New York, NY, USA, ACM (2011) 9–16
27. Mu, Y., Yan, S.: Non-metric locality-sensitive hashing. In: AAAI. (2010)
28. Novak, D., Kyselak, M., Zezula, P.: On locality-sensitive indexing in generic metric spaces. In: Proceedings of the Third International Conference on SImilarity Search and APplications. SISAP '10, New York, NY, USA, ACM (2010) 59–66
29. Parri, J., Shapiro, D., Bolic, M., Groza, V.: Returning control to the programmer: Simd intrinsics for virtual machines. Commun. ACM **54**(4) (April 2011) 38–43
30. Pestov, V.: Indexability, concentration, and {VC} theory. Journal of Discrete Algorithms **13**(0) (2012) 2 – 18 Best Papers from the 3rd International Conference on Similarity Search and Applications (SISAP 2010).
31. Pestov, V.: Is the k-NN classifier in high dimensions affected by the curse of dimensionality? Computers & Mathematics with Applications (2012)
32. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers Inc. (2005)
33. Scott, D.W., Thompson, J.R.: Probability density estimation in higher dimensions (1983) Technical Report, Rice University, Texas Huston.
34. Shafi, A., Carpenter, B., Baker, M., Hussain, A.: A comparative study of java and c performance in two large-scale parallel applications. Concurrency and Computation: Practice and Experience **21**(15) (2009) 1882–1906
35. Skopal, T.: Unified framework for fast exact and approximate search in dissimilarity spaces. ACM Trans. Database Syst. **32**(4) (November 2007)
36. Skopal, T., Bustos, B.: On nonmetric similarity search problems in complex domains. ACM Comput. Surv. **43**(4) (October 2011) 34:1–34:50
37. Uhlmann, J.: Satisfying general proximity similarity queries with metric trees. Information Processing Letters **40** (1991) 175–179
38. Vivanco, R.A., Pizzi, N.J.: Scientific computing with Java and C++: a case study using functional magnetic resonance neuroimages. Software: Practice and Experience **35**(3) (2005) 237–254
39. Weber, R., Schek, H.J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: Proceedings of the 24th International Conference on Very Large Data Bases, Morgan Kaufmann (August 1998) 194–205
40. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach (Advances in Database Systems). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
41. Zezula, P., Savino, P., Amato, G., Rabitti, F.: Approximate similarity retrieval with m-trees. The VLDB Journal **7**(4) (December 1998) 275–293
42. Zhang, Z., Ooi, B.C., Parthasarathy, S., Tung, A.K.H.: Similarity search on bregman divergence: towards non-metric indexing. Proc. VLDB Endow. **2**(1) (August 2009) 13–24