Non-Metric Space Library (NMSLIB) Manual

Bilegsaikhan Naidan, Leonid Boytsov, and Yury Malkov

Maintainer: Leonid Boytsov `leo@boytsov.info`

Version 1.8

Wednesday 5th June, 2019

Abstract. This document covers a library for fast similarity (k-NN) search. It describes **only** search methods and distances (spaces). Details about building, installing, Python bindings can be found online: `https://github.com/searchivarius/nmslib/tree/v1.8/`. Even though the library contains a variety of exact metric-space access methods, our main focus is on more *generic* and *approximate* search methods, in particular, on methods for non-metric spaces. NMSLIB is possibly the first library with a principled support for non-metric space searching.

1 Objectives and History

Non-Metric Space Library (NMSLIB) is an **efficient** and **extendable** cross-platform similarity search library and a toolkit for evaluation of similarity search methods. The core-library does **not** have any third-party dependencies.

The goal of the project is to create an effective and **comprehensive** toolkit for searching in **generic metric and non-metric** spaces. Even though the library contains a variety of metric-space access methods, our main focus is on *generic* and *approximate* search methods, in particular, on methods for non-metric spaces. NMSLIB is possibly the first library with a principled support for non-metric space searching.

NMSLIB is an extendible library, which means that is possible to add new search methods and distance functions. NMSLIB can be used directly in C++ and Python (via Python bindings). In addition, it is also possible to build a query server, which can be used from Java (or other languages supported by Apache Thrift). Java has a native client, i.e., it works on many platforms without requiring a C++ library to be installed.

NMSLIB started as a personal project of Bilegsaikhan Naidan, who created the initial code base, the Python bindings, and participated in earlier evaluations. The most successful class of methods–neighborhood/proximity graphs–is represented by the Hierarchical Navigable Small World Graph (HNSW) due to Malkov and Yashunin [33]. Other most useful methods, include a modification of the Vantage-Point Tree (VP-tree) [6], a Neighborhood APProximation index (NAPP) [43], which was improved by David Novak, as well as a vanilla uncompressed inverted file.

The current version of the manual focuses solely on the description of:

- search spaces and distances (see § 3);
- search methods (see § 4).

Details about building/installing, benchmarking, extending the code, as well as basic tuning guidelines, can be found online:
`https://github.com/nmslib/nmslib/tree/v1.8/manual/README.md`.

2 Terminology and Problem Formulation

Similarity search is an essential part of many applications, which include, among others, content-based retrieval of multimedia and statistical machine learning. The search is carried out in a finite database of objects $\{o_i\}$, using a search query q and a dissimilarity measure (the term data point or simply a point is often used a synonym to denote either a data object or a query). The dissimilarity measure is typically represented by a distance function $d(o_i, q)$. The ultimate goal is to answer a query by retrieving a subset of database objects sufficiently similar to the query q. These objects will be called *answers*. A combination of data points and the distance function is called a *search space*, or simply a *space*.

Note that we use the terms *distance* and the *distance function* in a broader sense than some of the textbooks: We do not assume that the distance is a true metric distance. The distance function can disobey the triangle inequality and/or be even non-symmetric.

Two 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 search (the k-NN search), which looks for the k 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 $d(o_i, q) \leq r$. In generic spaces, the distance is not necessarily symmetric. Thus, two types of queries can be considered. In a *left* query, the object is the left argument of the distance function, while the query is the right argument. In a *right* query, q is the first argument and the object is the second, i.e., the right, argument.

The queries can be answered either exactly, i.e., by returning a complete result set that does not contain erroneous elements, or, approximately, e.g., by finding only some answers. Thus, the methods are evaluated in terms of efficiency-effectiveness trade-offs rather than merely in terms of their efficiency. One common effectiveness metric is recall. In the case of the nearest neighbor search, it is computed as an average fraction of true neighbors returned by the method with ties broken arbitrarily.

3 Spaces

Currently we provide implementations mostly for vector spaces. Vector-space input files can come in either regular, i.e., dense, or sparse variant. A detailed list of spaces, their parameters, and performance characteristics is given in Table 1.

The mnemonic name of the space is passed to python bindings function as well as to the benchmarking utility (see `https://github.com/nmslib/nmslib/tree/v1.8/manual/benchmarking.md`). There can be more than one version of a distance function, which have different space-performance trade-off. In particular, for distances that require computation of logarithms we can achieve an order of magnitude improvement (e.g., for the GNU C++ and Clang) by pre-computing logarithms at index time. This comes at a price of extra storage. In the case of Jensen-Shannon distance functions, we can pre-compute some of the logarithms and accurately approximate those we cannot pre-compute. The details are explained in § 3.2-3.6.

Straightforward slow implementations of the distance functions may have the substring `slow` in their names, while faster versions contain the substring `fast`. Fast functions that involve approximate computations contain additionally the substring `approx`. For non-symmetric distance function, a space may have two variants: one variant is for left queries (the data object is the first, i.e., left, argument of the distance function while the query object is the second argument) and another is for right queries (the data object is the second argument and the query object is the first argument). In the latter case the name of the space ends on `rq`. Separating spaces by query types, might not be the best approach. Yet, it seems to be unavoidable, because, in many cases, we need separate indices to support left and right queries [10]. If you know a better approach, feel free, to tell us.

3.1 Details of Distance Efficiency Evaluation

Distance computation efficiency was evaluated on a Core i7 laptop (3.4 Ghz peak frequency) in a single-threaded mode (by the utility bench_distfunc). It is measured in millions of computations per second for single-precision floating pointer numbers (double precision computations are, of course, more costly). The code was compiled using the GNU compiler. All data sets were small enough to fit in a CPU cache, which may have resulted in slightly more optimistic performance numbers for cheap distances such as L_2.

Somewhat higher efficiency numbers can be obtained by using the Intel compiler or the Visual Studio (Clang seems to be equally efficient to the GNU compiler). In fact, performance is much better for distances relying on "heavy" math functions: slow versions of KL- and Jensen-Shannon divergences and Jensen-Shannon metrics, as well as for L_p spaces, where $p \notin \{1, 2, \infty\}$.

In the efficiency test, all dense vectors have 128 elements. For all dense-vector distances except the Jensen-Shannon divergence, their elements were generated randomly and uniformly. For the Jensen-Shannon divergence, we first generate elements randomly, and next we randomly select elements that are set to zero (approximately half of all elements). Additionally, for KL-divergences and the JS-divergence, we normalize vector elements so that they correspond a true discrete probability distribution.

Sparse space distances were tested using sparse vectors from two sample files in the sample_data directory. Sparse vectors in the first and the second file on average contain about 100 and 600 non-zero elements, respectively.

String distances were tested using DNA sequences sampled from a human genome.[1] The length of each string was sampled from a normal distribution $\mathcal{N}(32, 4)$.

The Signature Quadratic Form Distance (SQFD) [3,2] was tested using signatures extracted from LSVRC-2014 data set [38], which contains 1.2 million high resolution images. We implemented our own code to extract signatures following the method of Beecks [2]. For each image, we selected 10^4 pixels randomly and mapped them into 7-dimensional feature space: three color, two position, and two texture dimensions. The features were clustered by the standard k-means algorithm with 20 clusters. Then, each cluster was represented by an 8-dimensional vector, which included a 7-dimensional centroid and a cluster weight (the number of cluster points divided by 10^4).

3.2 L_p-norms and the Hamming Distance

The L_p distance between vectors x and y are given by the formula:

$$L_p(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p} \tag{1}$$

In the limit ($p \to \infty$), the L_p distance becomes the Maximum metric, also known as the Chebyshev distance:

$$L_\infty(x, y) = \max_{i=1}^{n} |x_i - y_i| \tag{2}$$

L_∞ and all spaces L_p for $p \geq 1$ are true metrics. They are symmetric, equal to zero only for identical elements, and, most importantly, satisfy *the triangle inequality*. However, the L_p norm is *not* a metric if $p < 1$.

In the case of dense vectors, we have reasonably efficient implementations for L_p distances where p is either integer or infinity. The most efficient implementations are for L_1 (Manhattan), L_2 (Euclidean), and L_∞ (Chebyshev). As explained in the author's blog, we compute exponents through square rooting. This works best when the number of digits (after the binary digit) is small, e.g., if $p = 0.125$.

Any L_p space can have a dense and a sparse variant. Sparse vector spaces have their own mnemonic names, which are different from dense-space mnemonic names in that they contain a suffix `_sparse` (see also Table 1). For instance `l1` and `l1_sparse` are both L_1 spaces, but the former is dense and the latter is sparse. The mnemonic names of L_1, L_2, and L_∞ spaces (passed to the benchmarking utility) are `l1`, `l2`, and `linf`, respectively. Other generic L_p have the

[1] `http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/`

Table 1: Description of implemented spaces

Space	**Mnemonic Name & Formula**		**Efficiency** (million op/sec)
	Metric Spaces		
Hamming	`bit_hamming`	$\sum_{i=1}^{n} \lvert x_i - y_i \rvert$	50-400
Jaccard	`bit_jaccard`, `jaccard_sparse`	$\sum_{i=1}^{n} \min(x_i, y_i) / \sum_{i=1}^{n} \max(x_i, y_i)$	500-200, 2
L_1	`l1`, `l1_sparse`	$\sum_{i=1}^{n} \lvert x_i - y_i \rvert$	35, 1.6
L_2	`l2`, `l2_sparse`	$\sqrt{\sum_{i=1}^{n} \lvert x_i - y_i \rvert^2}$	30, 1.6
L_∞	`linf`, `linf_sparse`	$\max_{i=1}^{n} \lvert x_i - y_i \rvert$	34 , 1.6
L_p (generic $p \geq 1$)	`lp:p=...`, `lp_sparse:p=...`	$\left(\sum_{i=1}^{n} \lvert x_i - y_i \rvert^p\right)^{1/p}$	0.1-3, 0.1-1.2
Angular distance	`angulardist`, `angulardist_sparse`, `angulardist_sparse_fast` $\arccos\left(\frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}\right)$		13, 1.4, 3.5
Jensen-Shan. metr.	`jsmetrslow, jsmetrfast, jsmetrfastapprox` $\sqrt{\frac{1}{2}\sum_{i=1}^{n}\left[x_i \log x_i + y_i \log y_i - (x_i + y_i) \log \frac{x_i + y_i}{2}\right]}$		0.3, 1.9, 4.8
Levenshtein	`leven` (see § 3.8 for details)		0.2
SQFD	`sqfd_minus_func`, `sqfd_heuristic_func:alpha=...`, `sqfd_gaussian_func:alpha=...` (see § 3.9 for details)		0.05, 0.05, 0.03
	Non-metric spaces (symmetric distance)		
L_p (generic $p < 1$)	`lp:p=..., lp_sparse:p=...`	$\left(\sum_{i=1}^{n} \lvert x_i - y_i \rvert^p\right)^{1/p}$	0.1-3, 0.1-1
Jensen-Shan. div.	`jsdivslow, jsdivfast, jsdivfastapprox` $\frac{1}{2}\sum_{i=1}^{n}\left[x_i \log x_i + y_i \log y_i - (x_i + y_i) \log \frac{x_i + y_i}{2}\right]$		0.3, 1.9, 4.8
Cosine distance	`cosinesimil`, `cosinesimil_sparse`, `cosinesimil_sparse_fast` $1 - \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$		13, 1.4, 3.5
Norm. Levenshtein	`normleven`, see § 3.8 for details		0.2
	Non-metric spaces (non-symmetric distance)		
Regular KL-div.	left queries: `kldivfast` right queries: `kldivfastrq` $\sum_{i=1}^{n} x_i \log \frac{x_i}{y_i}$		0.5, 27
Generalized KL-div.	left queries: `kldivgenslow`, `kldivgenfast` right queries: `kldivgenfastrq` $\sum_{i=1}^{n}\left[x_i \log \frac{x_i}{y_i} - x_i + y_i\right]$		0.5, 27 27
Itakura-Saito	left queries: `itakurasaitoslow, itakurasaitofast` right queries: `itakurasaitofastrq` $\sum_{i=1}^{n}\left[\frac{x_i}{y_i} - \log \frac{x_i}{y_i} - 1\right]$		0.2, 3, 14 14
Rényi divergence	`renyidiv_slow`, `renyidiv_fast` $\frac{1}{\alpha - 1} \log\left[\sum_{i=1}^{m} x_i^{\alpha} y_i^{1-\alpha}\right]$		0.4, 0.5-1.5

name `lp`, which is used in combination with a parameter. For instance, L_3 is denoted as `lp:p=3`.

Distance functions for sparse-vector spaces are far less efficient, due to a costly, branch-heavy, operation of matching sparse vector indices (between two sparse vectors).

In the special case of L_1 for binary vectors, the L_1 distance becomes the Hamming distance. This case is represented by the `bit_hamming` space, where data points are stored as compact bit vectors.

3.3 Scalar-product Related Distances

We have two distance function whose formulas include normalized scalar product. One is the cosine distance, which is equal to:

$$d(x,y) = 1 - \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

The cosine distance is not a true metric, but it can be converted into one by applying a monotonic transformation (i.e.., subtracting the cosine distance from one and taking an inverse cosine). The resulting distance function is a true metric, which is called the angular distance. The angular distance is computed using the following formula:

$$d(x,y) = \arccos\left(\frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}\right)$$

In the case of sparse spaces, to compute the scalar product, we need to obtain an intersection of vector element ids corresponding to non-zero elements. A classic text-book intersection algorithm (akin to a merge-sort) is not particularly efficient, apparently, due to frequent branching. For *single-precision* floating point vector elements, we provide a more efficient implementation that relies on the all-against-all comparison SIMD instruction `_mm_cmpistrm`. This implementation (inspired by the set intersection algorithm of Schlegel et al. [41]) is about 2.5-3 times faster than a pure C++ implementation based on the merge-sort approach.

3.4 Jaccard Distance

The Jaccard distance is true metric. Given two binary vectors x and y, the Jaccard distance (also called the index) is computed using the following formula:

$$\frac{\sum_{i=1}^{n} \min(x_i, y_i)}{\sum_{i=1}^{n} \max(x_i, y_i)}$$

For the Jaccard distance, there are two ways to represent data:

- A sparse set of dimensions (space `jaccard_sparse`);
- A dense binary bit vector (space `bit_jaccard`).

3.5 Jensen-Shannon Divergence

Jensen-Shannon divergence is a symmetrized and smoothed KL-divergence:

$$\frac{1}{2}\sum_{i=1}^{n}\left[x_i \log x_i + y_i \log y_i - (x_i + y_i)\log\frac{x_i + y_i}{2}\right] \tag{3}$$

This divergence is symmetric, but it is not a metric function. However, the square root of the Jensen-Shannon divergence is a proper a metric [19], which we call the Jensen-Shannon metric.

A straightforward implementation of Eq. 3 is inefficient for two reasons (at least when one uses the GNU C++ compiler) (1) computation of logarithms is a slow operation (2) the case of zero x_i and/or y_i requires conditional processing, i.e., costly branches.

A better method is to pre-compute logarithms of data at index time. It is also necessary to compute logarithms of a query vector. However, this operation has a little cost since it is carried out once for each nearest neighbor or range query. Pre-computation leads to a 3-10 fold improvement depending on the sparsity of vectors, albeit at the expense of requiring twice as much space. Unfortunately, it is not possible to avoid computation of the third logarithm: it needs to be computed in points that are not known until we see the query vector.

However, it is possible to approximate it with a very good precision, which should be sufficient for the purpose of approximate searching. Let us rewrite Equation 3 as follows:

$$\frac{1}{2}\sum_{i=1}^{n}\left[x_i \log x_i + y_i \log y_i - (x_i + y_i)\log\frac{x_i + y_i}{2}\right] =$$

$$= \frac{1}{2}\sum_{i=1}^{n}\left[x_i \log x_i + y_i \log y_i\right] - \sum_{i=1}^{n}\left[\frac{(x_i + y_i)}{2}\log\frac{x_i + y_i}{2}\right] =$$

$$= \frac{1}{2}\sum_{i=1}^{n} x_i \log x_i + y_i \log y_i -$$

$$\sum_{i=1}^{n}\frac{(x_i + y_i)}{2}\left[\log\frac{1}{2} + \log\max(x_i, y_i) + \log\left(1 + \frac{\min(x_i, y_i)}{\max(x_i, y_i)}\right)\right] \tag{4}$$

We can pre-compute all the logarithms in Eq. 4 except for $\log\left(1 + \frac{\min(x_i,y_i)}{\max(x_i,y_i)}\right)$. However, its argument value is in a small range: from one to two. We can discretize the range, compute logarithms in many intermediate points and save the computed values in a table. Finally, we employ the SIMD instructions to implement this approach. This is a very efficient approach, which results in a very little (around 10^{-6} on average) relative error for the value of the Jensen-Shannon divergence.

Another possible approach is to use an efficient approximation for logarithm computation. As our tests show, this method is about 1.5x times faster (1.5 vs

1.0 billions of logarithms per second), but for the logarithms in the range $[1, 2]$, the relative error is one order magnitude higher (for a single logarithm) than for the table-based discretization approach.

3.6 Bregman Divergences

Bregman divergences are typically non-metric distance functions, which are equal to a difference between some convex differentiable function f and its first-order Taylor expansion [8,10]. More formally, given the convex and differentiable function f (of many variables), its corresponding Bregman divergence $d_f(x, y)$ is equal to:

$$d_f(x, y) = f(x) - f(y) - (f(y) \cdot (x - y))$$

where $x \cdot y$ denotes the scalar product of vectors x and y. In this library, we implement the generalized KL-divergence and the Itakura-Saito divergence, which correspond to functions $f = \sum x_i \log x_i - \sum x_i$ and $f = -\sum \log x_i$. The generalized KL-divergence is equal to:

$$\sum_{i=1}^{n} \left[x_i \log \frac{x_i}{y_i} - x_i + y_i \right],$$

while the Itakura-Saito divergence is equal to:

$$\sum_{i=1}^{n} \left[\frac{x_i}{y_i} - \log \frac{x_i}{y_i} - 1 \right].$$

If vectors x and y are proper probability distributions, $\sum x_i = \sum y_i = 1$. In this case, the generalized KL-divergence becomes a regular KL-divergence:

$$\sum_{i=1}^{n} \left[x_i \log \frac{x_i}{y_i} \right].$$

Computing logarithms is costly: We can considerably improve efficiency of Itakura-Saito divergence and KL-divergence by pre-computing logarithms at index time. The spaces that implement this functionality contain the substring `fast` in their mnemonic names (see also Table 1).

3.7 Rényi Divergence

The Rényi divergence is a family of generally non-symmetric distances computed by the formula:

$$\frac{1}{\alpha - 1} \log \left[\sum_{i=1}^{m} x_i^{\alpha} y_i^{1-\alpha} \right]$$

The value of the parameter α should be greater than zero. For all values except $\alpha = 0.5$ these distances are non-symmetric. There are two variants of each

space: `renyidiv_slow` and `renyidiv_fast`. They have a parameter `alpha`. The slower variant computes the exponents using a standard function `pow`. For the fast variant, as explained in the author's blog, we compute exponents through square rooting. This works best when the number of digits (after the binary digit) is small, e.g., if $p = 0.125$. The slow variant can actually be faster, if the compiler, e.g., MS Visual Studio or the Intel Compiler, implements an efficient approximate variant of the `pow` function.

3.8 String Distances

We currently provide implementations for the Levenshtein distance and its length-normalized variant. The *original* Levenshtein distance is equal to the minimum number of insertions, deletions, and substitutions (but not transpositions) required to obtain one string from another [27]. The distance between strings p and s is computed using the classic $O(m \times n)$ dynamic programming solution, where m and n are lengths of strings p and s, respectively. The *normalized* Levenshtein distance is obtained by dividing the original Levenshtein distance by the maximum of string lengths. If both strings are empty, the distance is equal to zero.

While the original Levenshtein distance is a metric distance, the normalized Levenshtein function is not, because the triangle inequality may not hold. In practice, when there is little variance in string length, the violation of the triangle inequality is infrequent and, thus, the normalized Levenshtein distance is approximately metric for many real data sets.

Technically, the classic Levenshtein distance is equal to $C_{n,m}$, where $C_{i,j}$ is computed via the classic recursion:

$$C_{i,j} = \min \begin{cases} 0, & \text{if } i = j = 0 \\ C_{i-1,j} + 1, & \text{if } i > 0 \\ C_{i,j-1} + 1, & \text{if } j > 0 \\ C_{i-1,j-1} + [p_i \neq s_j], & \text{if } i, j > 0 \end{cases} \tag{5}$$

Because computation time is proportional to both strings' length, this can be a costly operation: for the sample data set described in § 3.1, it is possible to compute only about 200K distances per second.

The classic algorithm to compute the Levenshtein distance was independently discovered by several researchers in various contexts, including speech recognition [47,46,39] and computational biology [36] (see Sankoff [40] for a historical perspective). Despite the early discovery, the algorithm was generally unknown before a publication by Wagner and Fischer [48] in a computer science journal.

3.9 Signature Quadratic Form Distance (SQFD)

Images can be compared using a *family* of metric functions called the Signature Quadratic Form Distance (SQFD). During the preprocessing stage, each image

is converted to a set of n signatures (the number of signatures n is a parameter). To this end, a fixed number of pixels is randomly selected. Then, each pixel is represented by a 7-dimensional vector with the following components: three color, two position, and two texture elements. These 7-dimensional vectors are clustered by the standard k-means algorithm with n centers. Finally, each cluster is represented by an 8-dimensional vector, called *signature*. A signature includes a 7-dimensional centroid and a cluster weight (the number of cluster points divided by the total number of randomly selected pixels). Cluster weights form a *signature histogram*.

The SQFD is computed as a quadratic form applied to a $2n$-dimensional vector constructed by combining images' signature histograms. The combination vector includes n unmodified signature histogram values of the first image followed by n negated signature histogram values of the second image. Unlike the classic quadratic form distance, where the quadratic form matrix is fixed, in the case of the SQFD, the matrix is re-computed for each pair of images. This can be seen as computing the distance between infinite-dimensional vectors each of which has only a finite number of non-zero elements.

To compute the quadratic form matrix, we introduce the new global enumeration of signatures, in which a signature k from the first image has number k, while the signature k from the second image has number $n + k$. To obtain a quadratic form matrix element in row i column j we first compute the Euclidean distance d between the i-th and the j-th signature. Then, the value d is transformed using one of the three functions: negation (the *minus* function $-d$), a *heuristic* function $\frac{1}{\alpha+d}$, and the *Gaussian* function $\exp(-\alpha d^2)$. The larger is the distance, the smaller is the coefficient in the matrix of the quadratic form.

Note that the SQFD is a family of distances parameterized by the choice of the transformation function and α. For further details, please, see the thesis of Beecks [2].

4 Search Methods

Implemented search methods can be broadly divided into the following categories:

- Space partitioning methods (including a specialized method bbtree for Bregman divergences) § 4.1;
- Locality Sensitive Hashing (LSH) methods § 4.2;
- Filter-and-refine methods based on projection to a lower-dimensional space § 4.3;
- Filtering methods based on permutations § 4.4;
- Methods that construct a proximity graph § 4.5;
- Miscellaneous methods § 4.6.

In the following subsections (§ 4.1-4.6), we describe implemented methods, explain their parameters, and provide examples of their use via the benchmarking utility `experiment` (`experiment.exe` on Windows). The details on building the

utility `experiment` can be found online: `https://github.com/nmslib/nmslib/tree/v1.8/manual/README.md`.

4.1 Space Partitioning Methods

Parameters of space partitioning methods are summarized in Table 2. Most of these methods are hierarchical partitioning methods.

Hierarchical space partitioning methods create a hierarchical decomposition of the space (often in a recursive fashion), which is best represented by a tree (or a forest). There are two main partitioning approaches: pivoting and compact partitioning schemes [14].

Pivoting methods rely on embedding into a vector space where vector elements are distances from the object to pivots. Partitioning is based on how far (or close) the data points are located with respect to pivots. [2]

Hierarchical partitions produced by pivoting methods lack locality: a single partition can contain not-so-close data points. In contrast, compact partitioning schemes exploit locality. They either divide the data into clusters or create, possibly approximate, Voronoi partitions. In the latter case, for example, we can select several centers/pivots π_i and associate data points with the closest center.

If the current partition contains fewer than `bucketSize` (a method parameter) elements, we stop partitioning of the space and place all elements belonging to the current partition into a single bucket. If, in addition, the value of the parameter `chunkBucket` is set to one, we allocate a new chunk of memory that contains a copy of all bucket vectors. This method often halves retrieval time at the expense of extra memory consumed by a testing utility (e.g., `experiment`) as it does not deallocate memory occupied by the original vectors. [3]

Classic hierarchical space partitioning methods for metric spaces are exact. It is possible to make them approximate via an early termination technique, where we terminate the search after exploring a pre-specified number of partitions. To implement this strategy, we define an order of visiting partitions. In the case of clustering methods, we first visit partitions that are closer to a query point. In the case of hierarchical space partitioning methods such as the VP-tree, we greedily explore partitions containing the query.

In NMSLIB, the early termination condition is defined in terms of the maximum number of buckets (parameter `maxLeavesToVisit`) to visit before terminating the search procedure. By default, the parameter `maxLeavesToVisit` is set to a large number (2147483647), which means that no early termination is employed. The parameter `maxLeavesToVisit` is supported by many, but not all space partitioning methods.

[2] If the original space is metric, mapping an object to a vector of distances to pivots defines the contractive embedding in the metric spaces with L_∞ distance. That is, the L_∞ distance in the target vector space is a lower bound for the original distance.

[3] Keeping original vectors simplifies the testing workflow. However, this is not necessary for a real production system. Hence, storing bucket vectors at contiguous memory locations does not have to result in a larger memory footprint.

4.1.1 VP-tree A VP-tree [45,50] (also known as a ball-tree) is a pivoting method. During indexing, a (random) pivot is selected and a set of data objects is divided into two parts based on the distance to the pivot. If the distance is smaller than the median distance, the objects are placed into one (inner) partition. If the distance is larger than the median, the objects are placed into the other (outer) partition. If the distance is exactly equal to the median, the placement can be arbitrary.

We attempt to select the pivot multiple times. Each time, we measure the variance of distances to the pivot. Eventually, we use the pivot that corresponds to the maximum variance. The number of attempts to select the pivot is controlled by the index-time parameter `selectPivotAttempts`.

The VP-tree in metric spaces is an exact search method, which relies on the triangle inequality. It can be made approximate by applying the early termination strategy (as described in the previous subsection). Another approximate-search approach, which is currently implemented only for the VP-tree, is based on the relaxed version of the triangle inequality.

Assume that π is the pivot in the VP-tree, q is the query with the radius r, and R is the median distance from π to every other data point. Due to the triangle inequality, pruning is possible only if $r \leq |R - d(\pi, q)|$. If this latter condition is true, we visit only one partition that contains the query point. If $r > |R - d(\pi, q)|$, there is no guarantee that all answers are in the same partition as q. Thus, to guarantee retrieval of all answers, we need to visit both partitions.

The pruning condition based on the triangle inequality can be overly pessimistic. By selecting some $\alpha > 1$ and opting to prune when $r \leq \alpha|R - d(\pi, q)|$, we can improve search performance at the expense of missing some valid answers. The efficiency-effectiveness trade-off is affected by the choice of α: Note that for some (especially low-dimensional) data sets, a modest loss in recall (e.g., by 1-5%) can lead to an order of magnitude faster retrieval. Not only the triangle inequality can be overly pessimistic in metric spaces, but it often fails to capture the geometry of non-metric spaces. As a result, if the metric space method is applied to a non-metric space, the recall can be too low or retrieval time can be too long.

Yet, in non-metric spaces, it is often possible to answer queries, when using α possibly smaller than one [6,34]. More generally, we assume that there exists an unknown decision/pruning function $D(R, d(\pi, q))$ and that pruning is done when $r \leq D(R, d(\pi, q))$. The decision function $D()$, which can be learned from data, is called a search oracle. A pruning algorithm based on the triangle inequality is a special case of the search oracle described by the formula:

$$D_{\pi,R}(x) = \begin{cases} \alpha_{left}|x - R|^{exp_{left}}, & \text{if } x \leq R \\ \alpha_{right}|x - R|^{exp_{right}}, & \text{if } x \geq R \end{cases} \tag{6}$$

There are several ways to obtain/specify optimal parameters for the VP-tree:

- using the auto-tuning procedure fired before creation of the index;
- using the standalone tuning utility `tune_vptree` (`tune_vptree.exe` for Windows);

– fully manually.

It is, perhaps, easiest to initiate the tunning procedure during creation of the index. To this end, one needs to specify parameters `desiredRecall` (the minimum desired recall), `bucketSize` (the size of the bucket), `tuneK` or `tuneR`, and (optionally) parameters `tuneQty`, `minExp` and `maxExp`. Parameters `tuneK` and `tuneR` are used to specify the value of k for k-NN search, or the search radius r for the range search.

The parameter `tuneQty` defines the maximum number of records in a subset that is used for tuning. The tunning procedure will sample `tuneQty` records from the main set to make a (potentially) smaller data test. Additional query sets will be created by further random sampling of points from this smaller data set.

The tuning procedure considers all possible values for exponents between `minExp` and `maxExp` with a restriction that $exp_{left} = exp_{right}$. By default, `minExp` = `maxExp` = 1, which is usually a good setting. For each value of the exponents, the tunning procedure carries out a grid-like search procedure for parameters α_{left} and α_{right} with several random restarts. It creates several indices for the tuning subset and runs a batch of mini-experiments to find parameters yielding the desired recall value at the minimum cost. If it is necessary to produce more accurate estimates, the tunning method may use automatically adjusted values for parameters `tuneQty`, `bucketSize`, and `desiredRecall`. The tunning algorithm cannot adjust the parameter `maxLeavesToVisit`: please, do not use it with the auto-tunning procedure.

The disadvantage of automatic tuning is that it might fail to obtain parameters for a desired recall level. Another limitations is that a tunning procedure cannot run on very small data sets (less than two thousand entries).

The standalone tuning utility `tune_vptree` exploits an almost identical tuning procedure. It differs from index-time auto-tuning in several ways:

– It can be used with other VP-tree based methods, in particular, with the projection VP-tree (see § 4.3.2).
– It allows the user to specify a separate query set, which can be useful when queries cannot be accurately modelled by a bootstrapping approach (sampling queries from the main data set).
– Once the optimal values are computed, they can be further re-used without the need to start the tunning procedure each time the index is created.
– However, the user is fully responsible for specifying the size of the test data set and the value of the parameter `desiredRecall`: the system will not try to change them for optimization purposes.

If automatic tunning fails, the user can restart the procedure with the smaller value of `desiredRecall`. Alternatively, the user can manually specify values of parameters: `alphaLeft`, `alphaRight`, `expLeft`, and `expRight` (by default exponents are one).

The following is an example of testing the VP-tree with the benchmarking utility `experiment` without the auto-tunning (note the separation into index- and query-time parameters):

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method vptree \
    --createIndex  bucketSize=10,chunkBucket=1 \
    --queryTimeParams alphaLeft=2.0,alphaRight=2.0,\
                      expLeft=1,expRight=1,\
                      maxLeavesToVisit=500
```

To initiate auto-tuning, one may use the following command line (note that we do not use the parameter `maxLeavesToVisit` here):

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method vptree \
    --createIndex tuneK=1,desiredRecall=0.9,\
                  bucketSize=10,chunkBucket=1
```

4.1.2 Multi-Vantage Point Tree It is possible to have more than one pivot per tree level. In the binary version of the multi-vantage point tree (MVP-tree), which is implemented in NMSLIB, there are two pivots. Thus, each partition divides the space into four parts, which are similar to partitions created by two levels of the VP-tree. The difference is that the VP-tree employs three pivots to divide the space into four parts, while in the MVP-tree two pivots are used.

In addition, in the MVP-tree we memorize distances between a data object and the first `maxPathLen` (method parameter) pivots on the path connecting the root and the leaf that stores this data object. Because mapping an object to a vector of distances (to `maxPathLen` pivots) defines the contractive embedding in the metric spaces with L_∞ distance, these values can be used to improve the filtering capacity of the MVP-tree and, consequently to reduce the number of distance computations.

The following is an example of testing the MVP-tree with the benchmarking utility `experiment`:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method mvptree \
    --createIndex maxPathLen=4,bucketSize=10,chunkBucket=1 \
    --queryTimeParams maxLeavesToVisit=500
```

Our implementation of the MVP-tree permits to answer queries both exactly and approximately (by specifying the parameter `maxLeavesToVisit`). Yet, this implementation should be used only with metric spaces.

Table 2: Parameters of space partitioning methods

Common parameters	
`bucketSize`	A maximum number of elements in a bucket/leaf.
`chunkBucket`	Indicates if bucket elements should be stored contiguously in memory (1 by default).
`maxLeavesToVisit`	An early termination parameter equal to the maximum number of buckets (tree leaves) visited by a search algorithm (2147483647 by default).
VP-tree (`vptree`) [45,50]	
	Common parameters: `bucketSize`, `chunkBucket`, and `maxLeavesToVisit`
`selectPivotAttempts`	A number of pivot selection attempts (5 by default)
`alphaLeft/alphaRight`	A stretching coefficient $\alpha_{left}/\alpha_{right}$ in Eq. (6)
`expLeft/expRight`	The left/right exponent in Eq. (6)
`tuneK`	The value of k used in the auto-tunning procedure (in the case of k-NN search)
`tuneR`	The value of the radius r used in the auto-tunning procedure (in the case of the range search)
`minExp/maxExp`	The minimum/maximum value of exponent used in the auto-tunning procedure
Multi-Vantage Point Tree (`mvptree`) [7]	
	Common parameters: `bucketSize`, `chunkBucket`, and `maxLeavesToVisit`
`maxPathLen`	the maximum number of top-level pivots for which we memorize distances to data objects in the leaves
GH-tree (`ghtree`) [45]	
	Common parameters: `bucketSize`, `chunkBucket`, and `maxLeavesToVisit`
List of clusters (`list_clusters`) [13]	
	Common parameters: `bucketSize`, `chunkBucket`, and `maxLeavesToVisit`. Note `maxLeavesToVisit` is a **query-time** parameter.
`useBucketSize`	If equal to one, we use the parameter `bucketSize` to determine the number of points in the cluster. Otherwise, the size of the cluster is defined by the parameter `radius`.
`radius`	The maximum radius of a cluster (used when `useBucketSize` is set to zero).
`strategy`	A cluster selection strategy. It is one of the following: `random`, `closestPrevCenter`, `farthestPrevCenter`, `minSumDistPrevCenters`, `maxSumDistPrevCenters`.
SA-tree (`satree`) [35]	
	No parameters
bbtree (`bbtree`) [10]	
	Common parameters: `bucketSize`, `chunkBucket`, and `maxLeavesToVisit`

Note: mnemonic method names are given in round brackets.

4.1.3 GH-Tree A GH-tree [45] is a binary tree. In each node the data set is divided using two randomly selected pivots. Elements closer to one pivot are placed into a left subtree, while elements closer to the second pivot are placed into a right subtree.

The following is an example of testing the GH-tree with the benchmarking utility `experiment`:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method ghtree \
    --createIndex bucketSize=10,chunkBucket=1 \
    --queryTimeParams maxLeavesToVisit=10
```

Our implementation of the GH-tree permits to answer queries both exactly and approximately (by specifying the parameter `maxLeavesToVisit`). Yet, this implementation should be used only with metric spaces.

4.1.4 List of Clusters The list of clusters [13] is an exact search method for metric spaces, which relies on flat (i.e., non-hierarchical) clustering. Clusters are created sequentially starting by randomly selecting the first cluster center. Then, close points are assigned to the cluster and the clustering procedure is applied to the remaining points. Closeness is defined either in terms of the maximum `radius`, or in terms of the maximum number (`bucketSize`) of points closest to the center.

Next we select cluster centers according to one of the policies: random selection, a point closest to the previous center, a point farthest from the previous center, a point that minimizes the sum of distances to the previous center, and a point that maximizes the sum of distances to the previous center. In our experience, a random selection strategy (a default one) works well in most cases.

The search algorithm iterates over the constructed list of clusters and checks if answers can potentially belong to the currently selected cluster (using the triangle inequality). If the cluster can contain an answer, each cluster element is compared directly against the query. Next, we use the triangle inequality to verify if answers can be outside the current cluster. If this is not possible, the search is terminated.

We modified this exact algorithm by introducing an early termination condition. The clusters are visited in the order of increasing distance from the query to a cluster center. The search process stops after vising a `maxLeavesToVisit` clusters. Our version is supposed to work for metric spaces (and symmetric distance functions), but it can also be used with mildly-nonmetric symmetric distances such as the cosine distance.

An example of testing the list of clusters using the `bucketSize` as a parameter to define the size of the cluster:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method list_clusters \
    --createIndex useBucketSize=1,bucketSize=100,strategy=random \
    --queryTimeParams maxLeavesToVisit=5
```

An example of testing the list of clusters using the `radius` as a parameter to define the size of the cluster:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method list_clusters \
    --createIndex useBucketSize=0,radius=0.2,strategy=random \
    --queryTimeParams maxLeavesToVisit=5
```

4.1.5 SA-tree The Spatial Approximation tree (SA-tree) [35] aims to approximate the Voronoi partitioning. A data set is recursively divided by selecting several cluster centers in a greedy fashion. Then, all remaining data points are assigned to the closest cluster center.

A cluster-selection procedure first randomly chooses the main center point and arranges the remaining objects in the order of increasing distances to this center. It then iteratively fills the set of clusters as follows: We start from the empty cluster list. Then, we iterate over the set of data points and check if there is a cluster center that is closer to this point than the main center point. If no such cluster exists (i.e., the point is closer to the main center point than to any of the already selected cluster centers), the point becomes a new cluster center (and is added to the list of clusters). Otherwise, the point is added to the nearest cluster from the list.

After the cluster centers are selected, each of them is indexed recursively using the already described algorithm. Before this, however, we check if there are points that need to be reassigned to a different cluster. Indeed, because the list of clusters keeps growing, we may miss the nearest cluster not yet added to the list. To fix this, we need to compute distances among every cluster point and cluster centers that were not selected at the moment of the point's assignment to the cluster.

Currently, the SA-tree is an exact search method for metric spaces without any parameters. The following is an example of testing the SA-tree with the benchmarking utility `experiment`:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
```

```
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method satree
```

4.1.6 bbtree A Bregman ball tree (bbtree) is an exact search method for Bregman divergences [10]. The bbtree divides data into two clusters (each covered by a Bregman ball) and recursively repeats this procedure for each cluster until the number of data points in a cluster falls below `bucketSize`. Then, such clusters are stored as a single bucket.

At search time, the method relies on properties of Bregman divergences to compute the shortest distance to a covering ball. This is a rather expensive iterative procedure that may require several computations of direct and inverse gradients, as well as of several distances.

Additionally, Cayton [10] employed an early termination method: The algorithm can be told to stop after processing a `maxLeavesToVisit` buckets. The resulting method is an approximate search procedure.

Our implementation of the bbtree uses the same code to carry out the nearest-neighbor and the range searching. Such an implementation of the range searching is somewhat suboptimal and a better approach exists [11].

The following is an example of testing the bbtree with the benchmarking utility `experiment`:

```
release/experiment \
  --distType float --spaceType kldivgenfast \
  --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method bbtree \
    --createIndex bucketSize=10  \
    --queryTimeParams maxLeavesToVisit=20
```

4.2 Locality-sensitive Hashing Methods

Locality Sensitive Hashing (LSH) [25,26] is a class of methods employing hash functions that tend to have the same hash values for close points and different hash values for distant points. It is a probabilistic method in which the probability of having the same hash value is a monotonically decreasing function of the distance between two points (that we compare). A hash function that possesses this property is called *locality sensitive*.

Our library embeds the LSHKIT which provides locality sensitive hash functions in L_1 and L_2. It supports only the nearest-neighbor (but not the range) search. Parameters of LSH methods are summarized in Table 3. The LSH methods are not available under Windows.

Random projections is a common approach to design locality sensitive hash functions. These functions are composed from `M` binary hash functions $h_i(x)$. A concatenation of the binary hash function values, i.e., $h_1(x)h_2(x)\dots h_M(x)$, is interpreted as a binary representation of the hash function value $h(x)$. Pointers

Table 3: Parameters of LSH methods

Common parameters	
`W`	A width of the window [16].
`M`	A number of atomic (binary hash functions), which are concatenated to produce an integer hash value.
`H`	A size of the hash table.
`L`	The number hash tables.
Multiprobe LSH: only for L_2 (`lsh_multiprobe`) [30,18,16]	
	Common parameters: `W`, `M`, `H`, and `L`
`T`	a number of probes
`desiredRecall`	a desired recall
`numSamplePairs`	a number of samples (P in lshkit)
`numSampleQueries`	a number of sample queries (Q in lshkit)
`tuneK`	find optimal parameter for k-NN , search where k is defined by this parameter
LSH Gaussian: only for L_2 (`lsh_gaussian`) [12]	
	Common parameters: `W`, `M`, `H`, and `L`
LSH Cauchy: only for L_1 (`lsh_cauchy`) [12]	
	Common parameters: `W`, `M`, `H`, and `L`
LSH thresholding: only for L_1 (`lsh_threshold`) [49,29]	
	Common parameters: `M`, `H`, and `L` (`W` is not used)

Note: mnemonic method names are given in round brackets.

to objects with equal hash values (modulo `H`) are stored in same cells of the hash table (of the size `H`). If we used only one hash table, the probability of collision for two similar objects would be too low. To increase the probability of finding a similar object multiple hash tables are used. In that, we use a separate (randomly selected) hash function for each hash table.

To generate binary hash functions we first select a parameter `W` (called a *width*). Next, for every binary hash function, we draw a value a_i from a p-stable distribution [15], and a value b_i from the uniform distribution with the support $[0, W]$. Finally, we define $h_i(x)$ as:

$$h_i(x) = \left\lfloor \frac{a_i \cdot x + b_i}{W} \right\rfloor,$$

where $\lfloor x \rfloor$ is the `floor` function and $x \cdot y$ denotes the scalar product of x and y.

For the L_2 a standard Guassian distribution is p-stable, while for L_1 distance one can generate hash functions using a Cauchy distribution [15]. For L_1, the LSHKIT defines another ("thresholding") approach based on sampling. It is supposed to work best for data points enclosed in a cube $[a, b]^d$. We omit the description here and refer the reader to the papers that introduced this method [49,29].

One serious drawback of the LSH is that it is memory-greedy. To reduce the number of hash tables while keeping the collision probability for similar objects sufficiently high, it was proposed to "multi-probe" the same hash table more than once. When we obtain the hash value $h(x)$, we check (i.e., probe) not only the contents of the hash table cell $h(x) \mod H$, but also contents of cells whose binary codes are "close" to $h(x)$ (i.e, they may differ by a small number of bits). The LSHKIT, which is embedded in our library, contains a state-of-the-art implementation of the multi-probe LSH that can automatically select optimal values for parameters `M` and `W` to achieve a desired recall (remaining parameters still need to be chosen manually).

The following is an example of testing the multi-probe LSH with the benchmarking utility `experiment`. We aim to achieve the recall value 0.25 (parameter `desiredRecall`) for the 1-NN search (parameter `tuneK`):

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method lsh_multiprobe \
    --createIndex desiredRecall=0.25,tuneK=1,\
                  T=5,L=25,H=16535
```

The classic version of the LSH for L_2 can be tested as follows:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method lsh_gaussian \
    --createIndex W=2,L=5,M=40,H=16535
```

There are two ways to use LSH for L_1. First, we can invoke the implementation based on the Cauchy distribution:

```
release/experiment \
  --distType float --spaceType l1 --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method lsh_cauchy \
    --createIndex W=2,L=5,M=10,H=16535
```

Second, we can use L_1 implementation based on thresholding. Note that it does not use the width parameter `W`:

```
release/experiment \
  --distType float --spaceType l1 --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method lsh_threshold  \
    --createIndex L=5,M=60,H=16535
```

4.3 Projection-based Filter-and-Refine Methods

Projection-based filter-and-refine methods operate by mapping data and query points to a low(er) dimensional space (a *projection* space) with a simple, easy to compute, distance function. The search procedure consists in generation of candidate entries by searching in a low-dimensional projection space with subsequent refinement, where candidate entries are directly compared against the query using the original distance function.

The number of candidate records is an important method parameter, which can be specified as a fraction of the total number of data base entries (parameter `dbScanFrac`).

Different projection-based methods arise depending on: the type of a projection, the type of the projection space, and on the type of the search algorithm for the projection space. A type of the projection can be specified via a method's parameter `projType`. A dimensionality of the projection space is specified via a method's parameter `projDim`.

We support four well-known types of projections:

- Classic random projections using random orthonormal vectors (mnemonic name `rand`);
- Fastmap (mnemonic name `fastmap`);
- Distances to random reference points/pivots (mnemonic name `randrefpt`);
- Based on permutations `perm`;

All but the classic random projections are distance-based and can be applied to an arbitrary space with the distance function. Random projections can be applied only to vector spaces. A more detailed description of projection approaches is given in § A

We provide two basic implementations to generate candidates. One is based on brute-force searching in the projected space and another builds a VP-tree over objects' projections. In what follows, these methods are described in detail.

4.3.1 Brute-force projection search. In the brute-force approach, we scan the list of projections and compute the distance between the projected query and a projection of every data point. Then, we sort all data points in the order of increasing distance to the projected query. A fraction (defined by `dbScanFrac`) of data points is compared directly against the query. Top candidates (most closest entries) are identified using either the priority queue or incremental sorting ([24]). Incremental sorting is a more efficient approach enabled by default. The mnemonic code of this method is `proj_incsort`.

A choice of the distance in the projected space is governed by the parameter `useCosine`. If it set to 1, the cosine distance is used (this makes most sense if we use the cosine distance in the original space). By default `useCosine` = 0, which forces the use of L_2 in the projected space.

The following is an example of testing the brute-force search of projections with the benchmarking utility `experiment`:

```
release/experiment \
  --distType float --spaceType cosinesimil --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method proj_incsort \
    --createIndex projType=rand,projDim=4 \
    --queryTimeParams useCosine=1,dbScanFrac=0.01
```

4.3.2 Projection VP-tree. To avoid exhaustive search in the space of projections, it is possible to index projected vectors using a VP-tree. The method's mnemonic name is `proj_vptree`. In that, one needs to specify both the parameters of the VP-tree (see § 4.1.1) and the projection parameters as in the case of brute-force searching of projections (see § 4.3.1).

The major difference from the brute-force search over projections is that, instead of choosing between L_2 and cosine distance as the distance in the projected space, one uses a methods' parameter `projSpaceType` to specify an arbitrary one. Similar to the regular VP-tree implementation, optimal α_{left} and α_{right} are determined by the utility `tune_vptree` via a grid search like procedure (`tune_vptree.exe` on Windows).

This method, unfortunately, tends to perform worse than the VP-tree applied to the original space. The only exception are spaces with high intrinsic (and, perhaps, representational) dimensionality where VP-trees (even with an approximate search algorithm) are useless unless dimensionality is reduced substantially. One example is Wikipedia tf-idf vectors (see `https://github.com/nmslib/nmslib/tree/v1.8/manual/datasets.md`).

The following is an example of testing the VP-tree over projections with the benchmarking utility `experiment`:

```
release/experiment \
  --distType float --spaceType cosinesimil --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method proj_vptree  \
    --createIndex projType=rand,projDim=4,projSpaceType=cosinesimil \
    --queryTimeParams alphaLeft=2,alphaRight=2,dbScanFrac=0.01
```

4.3.3 OMEDRANK. In OMEDRANK [21] there is a small set of voting pivots, each of which ranks data points based on a somewhat imperfect notion of the distance from points to the query (computed by a classic random projection or a projection of some different kind). While each individual ranking is

imperfect, a more accurate ranking can be achieved by rank aggregation. When such a consolidating ranking is found, the most highly ranked objects from this *aggregate* ranking can be used as answers to a nearest-neighbor query. Finding the aggregate ranking is an NP-complete problem that Fagin et al. [21] solve only heuristically.

Technically, during the index time, each point in the original space is projected into a (low)er dimensional vector space. The dimensionality of the projection is defined using a method's parameter `numPivot` (note that this is different from other projection methods). Then, for each dimension i in the projected space, we sort data points in the order of increasing value of the i-th element of its projection.

We also divide the index in chunks each accounting for at most `chunkIndexSize` data points. The search algorithm processes one chunk at a time. The idea is to make a chunk sufficiently small so that auxiliary data structures fit into L1 or L2 cache.

The retrieval algorithm uses `numPivot` pointers low_i and `numPivot` pointers $high_i$ ($low_i \leq high_i$), The i-th pair of pointers (low_i, $high_i$) indicate a start and an end position in the i-th list. For each data point, we allocate a zero-initialized counter. We further create a projection of the query and use `numPivot` binary searches to find `numPivot` data points that have the closest i-th projection coordinates. In each of the i list, we make both $high_i$ and low_i point to the found data entries. In addition, for each data point found, we increase its counter. Note that a single data point may appear the closest with respect to more than one projection coordinate!

After that, we run a series of iterations. In each iteration, we increase `numPivot` pointers $high_i$ and decrease `numPivot` pointers low_i (unless we reached the beginning or the end of a list). For each data entry at which the pointer points, we increase the value of the counter. Obviously, when we complete traversal of all `numPivot` lists, each counter will have the value `numPivot` (recall that each data point appears exactly once in each of the lists). Thus, sooner or later the value of a counter becomes equal to or larger than `numPivot` $\times$ `minFreq`, where `minFreq` is a method's parameter, e.g., 0.5.

The first point whose counter becomes equal to or larger than `numPivot` $\times$ `minFreq`, becomes the first candidate entry to be compared directly against the query. The next point whose counter matches the threshold value `numPivot` $\times$ `minFreq`, becomes the second candidate and so on so forth. The total number of candidate entries is defined by the parameter `dbScanFrac`. Instead of all `numPivot` lists, it its possible to use only `numPivotSearch` lists that correspond to the smallest absolute value of query's projection coordinates. In this case, the counter threshold is `numPivotSearch` $\times$ `minFreq`. By default, `numPivot` = `numPivotSearch`.

Note that parameters `numPivotSearch` and `dbScanFrac` were introduced by us, they were not employed in the original version of OMEDRANK.

The following is an example of testing OMEDRANK with the benchmarking utility `experiment`:

```
release/experiment \
  --distType float --spaceType cosinesimil --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
    --method omedrank \
    --createIndex projType=rand,numPivot=8  \
    --queryTimeParams minFreq=0.5,dbScanFrac=0.02
```

4.4 Permutation-based Filtering Methods

Rather than relying on distance values directly, we can assess similarity of objects based on their relative distances to reference points (i.e., pivots). For each data point x, we can arrange pivots π in the order of increasing distances from x (for simplicity we assume that there are no ties). This arrangement is called a *permutation*. The permutation is essentially a pivot ranking. Technically, it is a vector whose i-th element keeps an (ordinal) position of the i-th pivot (in the set of pivots sorted by a distance from x).

Computation of the permutation is a mapping from a source space, which may not have coordinates, to a target vector space with integer coordinates. In our library, the distance between permutations is defined as either L_1 or L_2. Values of the distance in the source space often correlates well with the distance in the target space of permutations. This property is exploited in permutation methods. An advantage of permutation methods is that they are not relying on metric properties of the original distance and can be successfully applied to non-metric spaces [6,34].

Note that there is no simple relationship between the distance in the target space and the distance in the source space. In particular, the distance in the target space is neither a lower nor an upper bound for the distance in the source space. Thus, methods based on indexing permutations are filtering methods that allow us to obtain only approximate solutions. In the first step, we retrieve a certain number of candidate points whose permutations are sufficiently close to the permutation of the query vector. For these candidate data points, we compute an actual distance to the query, using the original distance function. For almost all implemented permutation methods, the number of candidate objects can be controlled by a parameter `dbScanFrac` or `minCandidate`.

Permutation methods differ in how they index and process permutations. In the following subsections, we briefly review implemented variants. Parameters of these methods are summarized in Tables 5-6.

4.4.1 Brute-force permutation search. In the brute-force approach, we scan the list of permutation methods and compute the distance between the permutation of the query and a permutation of every data point. Then, we sort

Table 4: Parameters of projection-based filter-and-refine methods

Common parameters	
`projType`	A type of projection.
`projDim`	Dimensionality of projection vectors.
`intermDim`	An intermediate dimensionality used to reduce dimensionality via the hashing trick (used only for sparse vector spaces).
`dbScanFrac`	A number of candidate records obtained during the filtering step.
Brute-force Projection Search (`proj_incsort`)	
	Common parameters: `projType`, `projDim`, `intermDim`, `dbScanFrac`
`useCosine`	If set to one, we use the cosine distance in the projected space. By default (value zero), L_2 is used.
`useQueue`	If set to one, we use the priority queue instead of incremental sorting. By default is zero.
Projection VP-tree (`proj_vptree`)	
	Common parameters: `projType`, `projDim`, `intermDim`, `dbScanFrac`
`projSpaceType`	Type of the space of projections
`bucketSize`	A maximum number of elements in a bucket/leaf.
`chunkBucket`	Indicates if bucket elements should be stored contiguously in memory (1 by default).
`maxLeavesToVisit`	An early termination parameter equal to the maximum number of buckets (tree leaves) visited by a search algorithm (2147483647 by default).
`alphaLeft/alphaRight`	A stretching coefficient $\alpha_{left}/\alpha_{right}$ in Eq. (6)
`expLeft/expRight`	The left/right exponent in Eq. (6)
OMEDRANK [21] (`omedrank`)	
	Common parameters: `projType`, `intermDim`, `dbScanFrac`
`numPivot`	Projection dimensionality
`numPivotSearch`	Number of data point lists to be used in search
`minFreq`	The threshold for being considered a candidate entry: whenever a point's counter becomes $\geq$ `numPivotSearch` $\times$ `minFreq`, this point is compared directly to the query.
`chunkIndexSize`	A number of documents in one index chunk.

Note: mnemonic method names are given in round brackets.

all data points in the order of increasing distance to the query permutation and a fraction (`dbScanFrac`) of data points is compared directly against the query.

In the current version of the library, the brute-force search over regular permutations is a special case of the brute-force search over projections (see 4.3.1), where the projection type is `perm`. There is also an additional brute-force filtering method, which relies on the so-called binarized permutations. It is described in 4.4.6.

4.4.2 Permutation Prefix Index (PP-Index). In a permutation prefix index (PP-index), permutation are stored in a prefix tree of limited depth [20]. A parameter `prefixLength` defines the depth. The filtering phase aims to find `minCandidate` candidate data points. To this end, it first retrieves the data points whose prefix of the inverse pivot ranking is exactly the same as that of the query. If we do not get enough candidate objects, we shorten the prefix and repeat the procedure until we get a sufficient number of candidate entries. Note that we do not the use the parameter `dbScanFrac` here.

The following is an example of testing the PP-index with the benchmarking utility `experiment`.

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method pp-index \
    --createIndex numPivot=4 \
    --queryTimeParams prefixLength=4,minCandidate=100
```

4.4.3 VP-tree index over permutations. We can use a VP-tree to index permutations. This approach is similar to that of Figueroa and Fredriksson [23]. We, however, rely on the approximate version of the VP-tree described in § 4.1.1, while Figueroa and Fredriksson use an exact one. The "sloppiness" of the VP-tree search is governed by the stretching coefficients `alphaLeft` and `alphaRight` as well as by the exponents in Eq. (6). In NMSLIB, the VP-tree index over permutations is a special case of the projection VP-tree (see § 4.3.2). There is also an additional VP-tree based method that indexes binarized permutations. It is described in § 4.4.6.

4.4.4 Metric Inverted File (MI-File) relies on the inverted index over permutations [1]. We select (a potentially large) subset of pivots (parameter `numPivot`). Using these pivots, we compute a permutation for every data point. Then, `numPivotIndex` most closest pivots are memorized in a data file. If a pivot number i is the pos-th most distant pivot for the object x, we add the pair (pos, x) to the posting list number i. All posting lists are kept sorted in the order of the increasing first element (equal to the ordinal position of the pivot in a permutation).

During searching, we compute the permutation of the query and select posting lists corresponding to `numPivotSearch` most closest pivots. These posting lists are processed as follows: Imagine that we selected posting list i and the position of pivot i in the permutation of the query is pos. Then, using the posting list i, we retrieve all candidate records for which the position of the pivot i in their respective permutations is from $pos - \text{maxPosDiff}$ to $pos + \text{maxPosDiff}$. This allows us to update the estimate for the L_1 distance between retrieved candidate records' permutations and the permutation of the query (see [1] for more details).

Finally, we select at most $\text{dbScanFrac} \cdot N$ objects (N is the total number of indexed objects) with the smallest estimates for the L_1 between their permutations and the permutation of the query. These objects are compared directly against the query. The filtering step of the MI-file is expensive. Therefore, this method is efficient only for computationally-intensive distances.

An example of testing this method using the utility `experiment` is as follows:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method mi-file  \
    --createIndex numPivot=128,numPivotIndex=16 \
    --queryTimeParams numPivotSearch=4,dbScanFrac=0.01
```

4.4.5 Neighborhood APProximation Index (NAPP). Recently it was proposed to index pivot neighborhoods: For each data point, we compute distances to `numPivot` points and select `numPivotIndex` (typically, much smaller than `numPivot`) pivots that are closest to the data point. Then, we associate these `numPivotIndex` closest pivots with the data point via an inverted file [43]. One can hope that for similar points two pivot neighborhoods will have a non-zero intersection.

To exploit this observation, our implementation of the pivot neighborhood indexing method retrieves all points that share at least `numPivotSearch` nearest neighbor pivots (using an inverted file). Then, these candidates points can be compared directly against the query, which works well for cheap distances like L_2.

For computationally expensive distances, one can add an additional filtering step by setting the parameter `useSort` to one. If `useSort` is one, all candidate entries are additionally sorted by the number of shared pivots (in the decreasing order). Afterwards, a subset of candidates are compared directly against the query. The size of the subset is defined by the parameter `dbScanFrac`. When selecting the subset, we give priority to candidates sharing more common pivots with the query. This secondary filtering may eliminate less promising entries, but it incurs additional computational costs, which may outweigh the benefits of additional filtering "power", if the distance is cheap.

In many cases, good performance can be achieved by selecting pivots randomly. However, we find that pivots can also be engineered (more information

on this topic will be published soon). To load external pivots, the user should specify an index-time parameter `pivotFile`. The pivots should be in the same format as the data points.

Note that our implementation is different from that of Tellez [43] in several ways. First, we do not use a succinct inverted index. Second, we use a simple posting merging algorithm based on counting (a *ScanCount* algorithm). Before a query is processed, we zero-initialize an array that keeps one counter for every data point. As we traverse a posting list and encounter an entry corresponding to object i, we increment a counter number i. The ScanCount is known to be quite efficient [28].

We also divide the index in chunks each accounting for at most `chunkIndexSize` data points. The search algorithm processes one chunk at a time. The idea is to make a chunk sufficiently small so that counters fit into L1 or L2 cache.

An example of running NAPP without the additional filtering stage:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --cachePrefixGS napp_gold_standard \
  --method napp \
    --createIndex numPivot=32,numPivotIndex=8,chunkIndexSize=1024 \
    --queryTimeParams numPivotSearch=8 \
    --saveIndex napp_index
```

Note that NAPP is capable of saving/loading the meta index. However, in the bootstrapping mode this is only possible if gold standard data is cached (hence, the option `--cachePrefixGS`).

An example of running NAPP **with** the additional filtering stage:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method napp   \
    --createIndex numPivot=32,numPivotIndex=8,chunkIndexSize=1024 \
    --queryTimeParams useSort=1,dbScanFrac=0.01,numPivotSearch=8
```

4.4.6 Binarized permutation methods. Instead of computing the L_2 distance between two permutations, we can binarize permutations and compute the Hamming distance between binarized permutations. To this end, we select an adhoc binarization threshold `binThreshold` (the number of pivots divided by two is usually a good setting). All integer values smaller than `binThreshold` become zeros, and values larger than or equal to `binThreshold` become ones.

The Hamming distance between binarized permutations can be computed much faster than L_2 or L_1 (see Table 1). This comes at a cost though, as the Hamming distance appears to be a worse proxy for the original distance than L_2 or L_1 (for the same number of pivots). One can compensate in quality by using

more pivots. In our experiments, it was usually sufficient to double the number of pivots.

The binarized permutation can be searched sequentially. An example of testing such a method using the utility `experiment` is as follows:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method perm_incsort_bin \
    --createIndex numPivot=32,binThreshold=16 \
    --queryTimeParams dbScanFrac=0.05
```

Alternatively, binarized permutations can be indexed using the VP-tree. This approach is usually more efficient than searching binarized permutations sequentially, but one needs to tune additional parameters. An example of testing such a method using the utility `experiment` is as follows:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1 --range 0.1 \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method perm_bin_vptree  \
    --createIndex numPivot=32 \
    --queryTimeParams alphaLeft=2,alphaRight=2,dbScanFrac=0.05
```

4.5 Proximity/Neighborhood Graphs

One efficient and effective search approach relies on building a graph, where data points are graph nodes and edges connect sufficiently close points. When edges connect nearest neighbor points, such graph is called a k-NN graph (or a nearest neighbor graph).

In a proximity-graph a search process is a series of greedy sub-searches. A sub-search starts at some, e.g., random node and proceeds to expanding the set of traversed nodes in a best-first fashion by following neighboring links. The algorithm resembles a Dijkstra's shortest-path algorithm in that, in each step, it selects an unvisited point closest to the query.

There have been multiple stopping heuristics proposed. For example, we can stop after visiting a certain number of nodes. In NMSLIB, the sub-search terminates essentially when the candidate queue is exhausted. Specifically, the candidate queue is expanded only with points that are closer to the query than the k'-*th* closest point already discovered by the sub-search (k' is a search parameter). When we stop finding such points, the queue dwindles and eventually becomes empty. We also terminate the sub-search when the queue contains only points farther than the k'-*th* closest point already discovered by the sub-search. Note that the greedy search is only approximate and does not necessarily return all true nearest neighbors.

Table 5: Parameters of permutation-based filtering methods

Common parameters	
`numPivot`	A number of pivots.
`dbScanFrac`	A number of candidate records obtained during the filtering step. It is specified as a *fraction* (not a percentage!) of the total number of data points in the data set.
`binThreshold`	Binarization threshold. If a value of an original permutation vector is below this threshold, it becomes 0 in the binarized permutation. If the value is above, the value is converted to 1.
Permutation Prefix Index (`pp-index`) [20]	
`numPivot`	A number of pivots.
`minCandidate`	a minimum number of candidates to retrieve (note that we do not use `dbScanFrac` here.
`prefixLength`	a maximum length of the tree prefix that is used to retrieve candidate records.
`chunkBucket`	1 if we want to store vectors having the same permutation prefix in the same memory chunk (i.e., contiguously in memory)
Metric Inverted File (`mi-file`) [1]	
	Common parameters: `numPivot` and `dbScanFrac`.
`numPivotIndex`	a number of (closest) pivots to index
`numPivotSearch`	a number of (closest) pivots to use during searching
`maxPosDiff`	the maximum position difference permitted for searching in the inverted file
Neighborhood Approximation Index (`napp`) [43]	
	Common parameter `numPivot`.
`invProcAlg`	An algorithm to merge posting lists. In practice, only `scan` worked well.
`chunkIndexSize`	A number of documents in one index chunk.
`indexThreadQty`	A number of indexhing threads.
`numPivotIndex`	A number of closest pivots to be indexed.
`numPivotSearch`	A candidate entry should share this number of pivots with the query. This is a **query-time** parameter.

Note: mnemonic method names are given in round brackets.

Table 6: Parameters of permutation-based filtering methods (continued)

Brute-force search with incremental sorting for binarized permutations (`perm_incsort_bin`) [42]	
Common parameters: `numPivot`, `dbScanFrac`, `binThreshold`.	
VP-tree index over binarized permutations (`perm_bin_vptree`) Similar to [42], but uses an approximate search in the VP-tree.	
Common parameters: `numPivot`, `dbScanFrac`, `binThreshold`. Note that `dbScanFrac` is a **query-time** parameter.	
`alphaLeft`	A stretching coefficient α_{left} in Eq. (6)
`alphaRight`	A stretching coefficient α_{right} in Eq. (6)

Note: mnemonic method names are given in round brackets.

In our library we use several approaches to create proximity graphs, which are described below. Parameters of these methods are summarized in Table 7. Note that SW-graph and NN-descent have the parameter with the same name, namely, `NN`. However, this parameter has a somewhat different interpretation depending on the method. Also note that our proximity-graph methods support only the nearest-neighbor, but not the range search.

4.5.1 Small World Graph (SW-graph). In the (Navigable) Small World graph (SW-graph),[4] indexing is a bottom-up procedure that relies on the previously described greedy search algorithm. The number of restarts, though, is defined by a different parameter, i.e., `initIndexAttempts`. We insert points one by one. For each data point, we find `NN` closest points using an already constructed index. Then, we create an *undirected* edge between a new graph node (representing a new point) and nodes that represent `NN` closest points found by the greedy search. Each sub-search starts from some, e.g., random node and proceeds expanding the candidate queue with points that are closer than the `efConstruction`-*th* closest point (`efConstruction` is an index-time parameter). Similarly, the search procedure executes one or more sub-searches that start from some node. The queue is expanded only with points that are closer than the `efSearch`-*th* closest point. The number of sub-searches is defined by the parameter `initSearchAttempts`.

Empirically, it was shown that this method often creates a navigable small world graph, where most nodes are separated by only a few edges (roughly logarithmic in terms of the overall number of objects) [31]. A simpler and less efficient variant of this algorithm was presented at ICTA 2011 and SISAP 2012

[4] SW-graph is also known as a Metrized Small-World (MSW) graph and a Navigable Small World (NSW) graph.

[37,31]. An improved variant appeared as an Information Systems publication [32]. In the latter paper, however, the values of `efSearch` and `efConstruction` are set equal to `NN`. The idea of using values of `efSearch` and `efConstruction` potentially (much) larger than `NN` was proposed by Malkov and Yashunin [33].

The indexing algorithm is rather expensive and we accelerate it by running parallel searches in multiple threads. The number of threads is defined by the parameter `indexThreadQty`. By default, this parameter is equal to the number of virtual cores. The graph updates are synchronized: If a thread needs to add edges to a node or obtain the list of node edges, it first locks a node-specific mutex. Because different threads rarely update and/or access the same node simultaneously, such synchronization creates little contention and, consequently, our parallelization approach is efficient. It is also necessary to synchronize updates for the list of graph nodes, but this operation takes little time compared to searching for `NN` neighboring points.

An example of testing this method using the utility `experiment` is as follows:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --cachePrefixGS sw-graph \
  --method sw-graph \
    --createIndex NN=3,initIndexAttempts=5,indexThreadQty=4 \
    --queryTimeParams initSearchAttempts=1,efSearch=10 \
    --saveIndex sw-graph_index
```

Note that SW-graph is capable of saving/loading the meta index. However, in the bootstrapping mode this is only possible if gold standard data is cached (hence, the option `--cachePrefixGS`).

4.5.2 Hierarchical Navigable Small World Graph (HNSW). The Hierarchical Navigable Small World Graph (HNSW) [33] is a new search method, a successor of the SW-graph. HNSW can be much faster (especially during indexing) and is more robust. However, the current implementation is still experimental and we will update it in the near future.

HNSW can be seen as a multi-layer and a multi-resolution variant of a proximity graph. A ground (zero-level) layer includes all data points. The higher is the layer, the fewer points it has. When a data point is added to HNSW, we select the maximum level m randomly. In that, the probability of selecting level m decreases exponentially with m.

Similarly to the SW-graph, the HNSW is constructed by inserting data points, one by one. A new point is added to all layers starting from layer m down to layer zero. This is done using a search-based algorithm similar to that of the basic SW-graph. The quality is controlled by the parameter `efConstruction`.

Specifically, a search starts from the maximum-level layer and proceeds to lower layers by searching one layer at a time. For all layers higher than the ground layer, the search algorithm is a 1-NN search that greedily follows the

closest neighbor (this is equivalent to having `efConstruction`=1). The closest point found at the layer $h+1$ is used as a starting point for the search carried out at the layer h. For the ground layer, we carry an `M`-NN search whose quality is controlled by the parameter `efConstruction`. Note that the ground-layer search relies one the same algorithm as we use for the SW-graph (yet, we use only a single sub-search, which is equivalent to setting `initIndexAttempts` to one).

An outcome of a search in a layer is a set of data points that are close to the new point. Using one of the heuristics described by Malkov and Yashunin [33], we select points from this set to become neighbors of the new point (in the layer's graph). Note that unlike the older SW-graph, the new algorithm has a limit on the maximum number of neighbors. If the limit is exceeded, the heuristics are used to keep only the best neighbors. Specifically, the maximum number of neighbors in all layers but the ground layer is `maxM` (an index-time parameter, which is equal to `M` by default). The maximum number of neighbors for the ground layer is `maxM0` (an index-time parameter, which is equal to 2×`M` by default). The choice of the heuristic is controlled by the parameter `delaunay_type`. Specifically, by default `delaunay_type` is equal to 2. This default is generally quite good. However, it maybe worth trying other viable options values: 0, 1, and 3.

A search algorithm is similar to the indexing algorithm. It starts from the maximum-level layer and proceeds to lower-level layers by searching one layer at a time. For all layers higher than the ground layer, the search algorithm is a 1-NN search that greedily follows the closest neighbor (this is equivalent to having `efSearch`=1). The closest point found at the layer $h+1$ is used as a starting point for the search carried out at the layer h. For the ground layer, we carry an k-NN search whose quality is controlled by the parameter `efSearch` (in the paper by Malkov and Yashunin [33] this parameter is denoted as `ep`). The ground-layer search relies one the same algorithm as we use for the SW-graph, but it does not carry out multiple sub-searches starting from different random data points.

For L_2 and the cosine similarity, HNSW has optimized implementations, which are enabled by default. To enforce the use of the generic algorithm, set the parameter `skip_optimized_index` to one.

Similar to SW-graph, the indexing algorithm can be expensive. It is, therefore, accelerated by running parallel searches in multiple threads. The number of threads is defined by the parameter `indexThreadQty`. By default, this parameter is equal to the number of virtual cores.

A sample command line to test HNSW using the utility `experiment`:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method hnsw \
    --createIndex M=10,efConstruction=20,indexThreadQty=4,searchMethod=0 \
    --queryTimeParams efSearch=10
```

HNSW is capable of saving an index for optimized L_2 and the cosine-similarity implementations. Here is an example for L_2:

```
release/experiment \
  --distType float --spaceType cosinesimil --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --cachePrefixGS hnsw \
  --method hnsw \
    --createIndex M=10,efConstruction=20,indexThreadQty=4,searchMethod=4 \
    --queryTimeParams efSearch=10
    --saveIndex hnsw_index
```

4.5.3 NN-Descent. The NN-descent is an iterative procedure initialized with randomly selected nearest neighbors. In each iteration, a random sample of queries is selected to participate in neighborhood propagation.

This process is governed by parameters `rho` and `delta`. Parameter `rho` defines a fraction of the data set that is randomly sampled for neighborhood propagation. A good value that works in many cases is `rho` = 0.5. As the indexing algorithm iterates, fewer and fewer neighborhoods change (when we attempt to improve the local neighborhood structure via neighborhood propagation). The parameter `delta` defines a stopping condition in terms of a fraction of modified edges in the k-NNgraph (the exact definition can be inferred from code). A good default value is `delta`=0.001. The indexing algorithm is multi-threaded: the method uses all available cores.

When NN-descent was incorporated into NMSLIB, there was no open-source search algorithm released, only the code to construct a k-NNgraph. Therefore, we use the same algorithm as for the SW-graph [31,32]. The new, open-source, version of NN-descent (code-named `kgraph`), which does include the search algorithm, can be found on GitHub.

Here is an example of testing this method using the utility `experiment`:

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method nndes  \
    --createIndex NN=10,rho=0.5,delta=0.001 \
    --queryTimeParams initSearchAttempts=3
```

4.6 Miscellaneous Methods

Currently our major miscellaneous methods do not have parameters.

4.6.1 Brute-force (sequential) searching. To verify how the speed of brute-force searching scales with the number of threads, we provide a reference implementation of the sequential searching. For example, to benchmark sequential searching using two threads, one can type the following command:

Table 7: Parameters of proximity-graph based methods

Common parameters	
`efSearch`	The search depth: specifically, a sub-search is stopped, when it cannot find a point closer than `efSearch` points (seen so far) closest to the query.
SW-graph (`sw-graph`) [37,31,32]	
`NN`	For a newly added point find this number of most closest points that make the initial neighborhood of the point. When more points are added, this neighborhood may be expanded.
`efConstruction`	The depth of the search that is used to find neighbors during indexing. This parameter is analogous to `efSearch`.
`initIndexAttempts`	The number of random search restarts carried out to add one point.
`indexThreadQty`	The number of indexing threads. The default value is equal to the number of (logical) CPU cores.
`initSearchAttempts`	A number of random search restarts.
Hierarchical Navigable SW-graph (`hnsw`) [32]	
`mult`	A scaling coefficient to determine the depth of a layered structure (see the paper by Malkov and Yashunin [32] for details). A default value seems to be good enough.
`skip_optimized_index`	Setting this parameter to one disables the use of the optimized implementations (for L_2 and the cosine similarity).
`maxM`	The maximum number of neighbors in all layers but the ground layer (the default value seems to be good enough).
`maxM0`	The maximum number of neighbors in the *ground* layer (the default value seems to be good enough).
`M`	The size of the initial set of potential neighbors for the indexing phase. The set may be further pruned so that the overall number of neighbors does not exceed `maxM0` (for the ground layer) or `maxM` (for all layers but the ground one).
`efConstruction`	The depth of the search that is used to find neighbors during indexing (this parameter is used only for the search in the ground layer).
`delaunay_type`	A type of the pruning heuristic: 0 indicates that we keep only `maxM` (or `maxM0` for the ground layer) neighbors, 1 activates a heuristic described by Algorithm 4 [32]
NN-descent (`nndes`) [17,31,32]	
`NN`	For each point find this number of most closest points (neighbors).
`rho`	A fraction of the data set that is randomly sampled for neighborhood propagation.
`delta`	A stopping condition in terms of the fraction of updated edges in the k-NNgraph.
`initSearchAttempts`	A number of random search restarts.

Note: mnemonic method names are given in round brackets.

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method seq_search --threadTestQty 2
```

4.6.2 Term-Based Uncompressed Inverted File. For sparse data sets (especially when queries are much sparser than documents), it can be useful to employ the inverted file. The inverted file is a mapping from a set of dimensions to their respective posting lists. The posting list of a dimension is a sorted list of documents/vectors, where this dimension is non-zero. To answer queries, we use a classic document-at-a-time (DAAT) processing algorithm [44] with a priority queue (see Figure 5.3 in [9]), which is a part of NMSLIB. This algorithm is implemented in Lucene 6.0 (and earlier versions). However, our implementation is faster than Lucene (1.5-2× faster on *compr* and *stack*) for several reasons: our implementation does not use compression, it does not explicitly compute BM25, it is written in C++ rather than Java

```
release/experiment \
  --distType float --spaceType negdotprod_sparse_fast --testSetQty 5 \
  --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/sparse_wiki_5K.txt --outFilePrefix result \
  --method simple_invindx --threadTestQty 2
```

5 Notes on Efficiency

5.1 Efficiency of Distance Functions

Note that improvement in efficiency and in the number of distance computations obtained with slow distance functions can be overly optimistic. That is, when a slow distance function is replaced with a more efficient version, the improvements over sequential search may become far less impressive. In some cases, the search method can become even slower than the brute-force comparison against every data point. This is why we believe that optimizing computation of a distance function is equally important (and sometimes even more important) than designing better search methods.

In this library, we optimized several distance functions, especially non-metric functions that involve computations of logarithms. An order of magnitude improvement can be achieved by pre-computing logarithms at index time and by approximating those logarithms that are not possible to pre-compute (see § 3.5 and § 3.6 for more details). Yet, this doubles the size of an index.

The Intel compiler has a powerful math library, which allows one to efficiently compute several hard distance functions such as the KL-divergence, the Jensen-Shanon divergence/metric, and the L_p spaces for non-integer values of p more efficiently than in the case of GNU C++ and Clang. In the Visual Studio's fast

math mode (which is enabled in the provided project files) it is also possible to compute some hard distances several times faster compared to GNU C++ and Clang. Yet, our custom implementations are often much faster. For example, in the case of the Intel compiler, the custom implementation of the KL-divergence is 10 times faster than the standard one while the custom implementation of the JS-divergence is two times faster. In the case of the Visual studio, the custom KL-divergence is 7 times as fast as the standard one, while the custom JS-divergence is 10 times faster. Therefore, doubling the size of the data set by storing pre-computed logarithms seems to be worthwhile.

Efficient implementations of some other distance functions rely on SIMD instructions. These instructions, available on most modern Intel and AMD processors, operate on small vectors. Some C++ implementations can be efficiently vectorized by both the GNU and Intel compilers. That is, instead of the scalar operations the compiler would generate more efficient SIMD instructions. Yet, the code is not always vectorized, e.g., by the Clang. And even the Intel compiler, fails to efficiently vectorize computation of the KL-divergence (with pre-computed logarithms).

There are also situations when efficient automatic vectorization is hardly possible. For instance, we provide an efficient implementation of the scalar product for sparse *single-precision* floating point vectors. It relies on the all-against-all comparison SIMD instruction `_mm_cmpistrm`. However, it requires keeping the data in a special format, which makes automatic vectorization impossible.

Intel SSE extensions that provide SIMD instructions are automatically detected by all compilers but the Visual Studio. If some SSE extensions are not available, the compilation process will produce warnings like the following one:

```
LInfNormSIMD: SSE2 is not available, defaulting to pure C++ implementation!
```

5.2 Cache-friendly Data Layout

In our previous report [5], we underestimated a cost of a random memory access. A more careful analysis showed that, on a recent laptop (Core i7, DDR3), a truly random access "costs" about 200 CPU cycles, which may be 2-3 times longer than a computation of a cheap distance such as L_2.

Many implemented methods use some form of bucketing. For example, in the VP-tree or bbtree we recursively decompose the space until a partition contains at most `bucketSize` elements. The buckets are searched sequentially, which could be done much faster, if bucket objects were stored in contiguous memory regions. Thus, to check elements in a bucket we would need only one random memory access.

A number of methods support this optimized storage model. It is activated by setting a parameter `chunkBucket` to 1. If `chunkBucket` is set to 1, indexing is carried out in two stages. At the first stage, a method creates unoptimized buckets, each of which is an array of pointers to data objects. Thus, objects are not necessarily contiguous in memory. In the second stage, the method iterates over buckets, allocates a contiguous chunk of memory, which is sufficiently large to keep all bucket objects, and copies bucket objects to this new chunk.

Important note: Note that currently we do not delete old objects and do not deallocate the memory they occupy. Thus, if `chunkBucket` is set to 1, the memory usage is overestimated. In the future, we plan to address this issue.

References

1. G. Amato and P. Savino. Approximate similarity search in metric spaces using inverted files. In *Proceedings of the 3rd international conference on Scalable information systems*, page 28. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.
2. C. Beecks. *Distance based similarity models for content based multimedia retrieval*. PhD thesis, 2013.
3. C. Beecks, M. S. Uysal, and T. Seidl. Signature quadratic form distance. In *Proceedings of the ACM International Conference on Image and Video Retrieval*, CIVR '10, pages 438–445, New York, NY, USA, 2010. ACM.
4. E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 245–250. ACM, 2001.
5. L. Boytsov and B. Naidan. Engineering efficient and effective Non-Metric Space Library. In N. Brisaboa, O. Pedreira, and P. Zezula, editors, *Similarity Search and Applications*, volume 8199 of *Lecture Notes in Computer Science*, pages 280–293. Springer Berlin Heidelberg, 2013.
6. L. Boytsov and B. Naidan. Learning to prune in metric and non-metric spaces. In *Advances in Neural Information Processing Systems*, 2013.
7. T. Bozkaya and M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS)*, 24(3):361–404, 1999.
8. L. Bregman. The relaxation method of finding the common point of convex sets and its application to the solution of problems in convex programming. {*USSR*} *Computational Mathematics and Mathematical Physics*, 7(3):200 – 217, 1967.
9. S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval - Implementing and Evaluating Search Engines*. MIT Press, 2010.
10. L. Cayton. Fast nearest neighbor retrieval for bregman divergences. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 112–119, New York, NY, USA, 2008. ACM.
11. L. Cayton. Efficient bregman range search. In *Advances in Neural Information Processing Systems*, pages 243–251, 2009.
12. M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.
13. E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.
14. E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
15. M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.
16. W. Dong. *High-Dimensional Similarity Search for Large Datasets*. PhD thesis, Princeton University, 2011.

17. W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586. ACM, 2011.
18. W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling lsh for performance tuning. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 669–678, New York, NY, USA, 2008. ACM.
19. D. M. Endres and J. E. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, 2003.
20. A. Esuli. Use of permutation prefixes for efficient and scalable approximate similarity search. *Inf. Process. Manage.*, 48(5):889–902, Sept. 2012.
21. R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 301–312, New York, NY, USA, 2003. ACM.
22. C. Faloutsos and K.-I. Lin. *FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets*, volume 24. ACM, 1995.
23. K. Figueroa and K. Fredriksson. Speeding up permutation based indexing with indexing. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, pages 107–114. IEEE Computer Society, 2009.
24. E. Gonzalez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(9):1647–1658, 2008.
25. P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.
26. E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the 30th annual ACM symposium on Theory of computing*, STOC '98, pages 614–623, New York, NY, USA, 1998. ACM.
27. V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR,*, 163(4):845–848, 1966.
28. C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 257–266. IEEE, 2008.
29. Q. Lv, M. Charikar, and K. Li. Image similarity search with compact data structures. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 208–217. ACM, 2004.
30. Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, pages 950–961. VLDB Endowment, 2007.
31. Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces. In *Similarity Search and Applications*, pages 132–147. Springer, 2012.
32. Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Inf. Syst.*, 45:61–68, 2014.

33. Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *ArXiv e-prints*, Mar. 2016.
34. B. Naidan, L. Boytsov, and E. Nyberg. Permutation search methods are efficient, yet faster search is possible. *PVLDB*, 8(12):1618–1629, 2015.
35. G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46, 2002.
36. S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3):443–453, March 1970.
37. A. Ponomarenko, Y. Malkov, A. Logvinov, , and V. Krylov. Approximate nearest neighbor search small world approach, 2011. Available at `http://www.iiis.org/CDs2011/CD2011IDI/ICTA_2011/Abstract.asp?myurl=CT175ON.pdf`.
38. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge, 2014.
39. H. Sakoe and S. Chiba. A dynamic programming approach to continuous speech recognition. In *Proceedings of the Seventh International Congress on Acoustics*, pages 65–68, August 1971. paper 20C13.
40. D. Sankoff. The early introduction of dynamic programming into computational biology. *Bioinformatics*, 16(1):41–47, 2000.
41. B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. In *ADMS@ VLDB*, pages 1–8, 2011.
42. E. S. Téllez, E. Chávez, and A. Camarena-Ibarrola. A brief index for proximity searching. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 529–536. Springer, 2009.
43. E. S. Tellez, E. Chávez, and G. Navarro. Succinct nearest neighbor search. *Information Systems*, 38(7):1019–1030, 2013.
44. H. R. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, 1995.
45. J. Uhlmann. Satisfying general proximity similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.
46. V. Velichko and N. Zagoruyko. Automatic recognition of 200 words. *International Journal of Man-Machine Studies*, 2(3):223 – 234, 1970.
47. T. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968.
48. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.
49. Z. Wang, W. Dong, W. Josephson, Q. Lv, M. Charikar, and K. Li. Sizing sketches: a rank-based analysis for similarity search. *ACM SIGMETRICS Performance Evaluation Review*, 35(1):157–168, 2007.
50. P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

A Description of Projection Types

A.1 The classic random projections

The classic random projections work only for vector spaces (both sparse and dense). At index time, we generate `projDim` vectors by sampling their elements from the standard normal distribution $\mathcal{N}(0,1)$ and orthonormalizing them. [5] Coordinates in the projection spaces are obtained by computing scalar products between a given vector and each of the `projDim` randomly generated vectors.

In the case of sparse vector spaces, the dimensionality is first reduced via the hashing trick: the value of the element i is equal to the sum of values for all elements whose indices are hashed into number i. After hashing, classic random projections are applied. The dimensionality of the intermediate space is defined by a method's parameter `intermDim`.

The hashing trick is used purely for efficiency reasons. However, for large enough values of the intermediate dimensionality, it has virtually no adverse affect on performance. For example, in the case of Wikipedia tf-idf vectors (see `https://github.com/nmslib/nmslib/tree/v1.8/manual/datasets.md`), it is safe to use the value `intermDim`=4096.

Random projections work best if both the source and the target space are Euclidean, whereas the distance is either L_2 or the cosine distance. In this case, there are theoretical guarantees that the projection preserves well distances in the original space (see e.g. [4]).

A.2 FastMap

FastMap introduced by Faloutsos and Lin [22] is also a type of the random-projection method. At indexing time, we randomly select $projDim$ pairs A_i and B_i. The $i\text{-}th$ coordinate of vector x is computed using the formula:

$$\texttt{FastMap}_i(x) = \frac{d(A_i,x)^2 - d(B_i,x_i)^2 + d(A_i,B_i)}{2d(A_i,B_i)^2} \tag{7}$$

Given points A and B in the Euclidean space, Eq. 7 gives the length of the orthogonal projection of x to the line connecting A and B. However, FastMap can be used in non-Euclidean spaces as well.

A.3 Distances to the Random Reference Points

This method is a folklore projection approach, where the $i\text{-}th$ coordinate of point x in the projected space is computed as simply $d(x,\pi_i)$, where π_i is a pivot in the original space, i.e., a randomly selected reference point. Pivots are selected once during indexing time.

[5] If the dimensionality of the projection space is larger than the dimensionality of the original space, only the first `projDim` vectors are orthonormalized. The remaining are simply divided by their norms.

A.4 Permutation-based Projections.

In this approach, we also select `projDim` pivots at index time. However, instead of using raw distances to the pivots, we rely on ordinal positions of pivots sorted by their distance to a point. A more detailed description is given in § 4.4.