Non-Metric Space Library Manual

Bilegsaikhan Naidan[1] and Leonid Boytsov[2]

[1] Department of Computer and Information Science,
Norwegian University of Science and Technology,
Trondheim, Norway

[2] Language Technologies Institute,
Carnegie Mellon University,
Pittsburgh, PA, USA
srchvrs@cmu.edu

Maintainer: Leonid Boytsov

Version 1.0

December 2013

Abstract. This document describes a library for similarity searching. Even though it contains a variety of metric-space access methods, our main focus is searching in non-metric spaces. Because there are fewer exact solutions for non-metric spaces, many of our methods give only approximate answers and, thus, are evaluated in terms of efficiency-effectiveness trade-offs rather than merely in terms of their run-time. Our goal is, therefore, to provide not only state-of-the-art approximate search methods for both non-metric and metric spaces, but also the tools to measure search quality. Another important goal is to provide efficient implementations, with run-times comparable to an optimized production system. We concentrate on technical details, i.e., how to compile the code, run the benchmarks, and evaluate results. Additionally, we explain how to extend the code by adding new search methods and spaces.

1 History and Motivation

The Non-Metric Space Library was created in 2013. Most code was written by Bilegsaikhan Naidan and Leonid Boytsov. Leo(nid) Boytsov is a maintainer. The library is hosted in the GitHub repository of the maintainer. We encourage potential users and contributors to use GitHub for bug reports. The code that was written entirely by the authors is distributed under the business-friendly Apache License. The library contains additional contributions, which are sometimes licensed differently. For more information regarding licensing and acknowledging the use of the library resource, please refer to § 10.

The main motivation and methodology is described in our engineering paper [**?**]. The experimental results obtained with the help of the library were published elsewhere [**?**]. This document focuses mostly on low-level implementation

details. Our design was influenced by and superficially resembles the design of the Metric Spaces Library [**?**], but our approach is different in many ways. Most importantly, we focus on approximate search methods (which do not guarantee, e.g., to return a true nearest-neighbor for all queries) and non-metric spaces. We also simplified the procedure to extend the code (see § 8 for details). Another important difference is our focus on efficiency: we aim to implement methods that have run-times comparable to an optimized production system. We, therefore, make an effort to write efficient distance functions and provide capabilities for testing in both single- and multi-threaded modes. Last, but not least, we simplify processing of experimental results through automatically measuring and aggregating important parameters related to speed, accuracy, and memory consumption.

2 Introduction

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 on a finite database of objects $\{o_i\}$, using a search query q and a dissimilarity measure. The dissimilarity measure is typically represented by a distance function $d(o_i, q)$. The ultimate goal is to retrieve a subset of database objects sufficiently similar to the query q. Note that we use the terms `distance` and the `distance function` in a broader sense: We do not assume that the distance is a true metric distance. The distance can be asymmetric and does not necessarily satisfy the triangle inequality.

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 (or the k-NN) search, which looks for the k most closest objects. Given a radius r, the range query retrieves all objects within a query ball (centered at the query object q) with the radius r, or, formally, all the objects $\{o_i\}$ such that $d(o_i, q) \leq r$. 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, or, approximately, e.g., by finding only some nearest neighbors. Search methods for non-metric spaces are especially interesting. This domain does not provide sufficiently generic *exact* search methods. We may know very little about analytical properties of the distance or the analytical representation may not be available at all (e.g., if the distance is computed by a black-box device [**?**]). Hence, employing an approximate approach is virtually unavoidable.

Approximate search methods are typically more efficient than exact ones. Yet, it is harder to evaluate them, because we need to measure retrieval speed at

different levels of rank approximation quality, recall, or any other effectiveness metric. Thus it is crucial to provide capabilities to measure search quality.

As noted in our methodology paper [**?**], we do optimize the code for speed and use efficient implementations of distance functions (see §9 for details). Oftentimes, it is possible to demonstrate a substantial reduction in the number of distance computations. Less frequently, however, such improvements result in improved performance. Note that disk-based indices are typically slower than memory-based ones. Because efficiency is an important goal, we originally provide only in-memory implementations, which can be tested in either single- or multi-threaded mode. This simplification reduces programming effort. It is, nevertheless, possible to benchmark disk-based implementations as well (see 8 for details). In the current release, we focus on vector-space implementations, (i.e., all the distance functions are defined over real-valued vectors). Note that this is not a principal limitation as many methods do not access the objects directly and, instead, rely only on distance values. In the future, we plan to add more complex spaces, in particular, string-based.

3 Prerequisites

The Non-Metric Space Library was created primarily as a research project. This is why we did not mean to create truly portable code. Yet, it is probably possible to build it (with minimum modifications) using any C++ compiler as long as it supports the new C++ 11 standard.

Efficient implementations of many distance functions (see § 9) rely on Single Instruction Multiple Data (SIMD) instructions. These instructions, available on most modern Intel and AMD processors, operate on small vectors. However, each distance function has a pure C++ implementation, which should be automatically selected on other platforms. Note, however, that we do not have a portable code to measure memory consumption: This part will work only for Linux.

More specifically, we require:

1. A GNU C++ compiler, version 4.7 (or higher), or the Intel compiler version 11;
2. cmake (GNU make is also required);
3. Boost (dev version $\geq$ 48, Ubuntu package `libboost1.48-all-dev`);
4. GNU scientific library (dev version, Ubuntu package `libgsl0-dev`).

One needs Python to generate vectors with element values drawn from $U[0, 1]$. To plot the graphs you need Python, Latex, and PGF. Most of the code was tested and run on Linux Ubuntu (Intel CPU), but it should be possible to run the code on other (perhaps, even non-Unix) platforms.

Installing GNU C++ version 4.7 compiler may be tricky, because it is not always provided as a standard package. On some Linux distributions with the Debian package management system, you can simply type:

```
sudo apt-get install gcc-4.7 g++-4.7
```

However, it did not work for us and we needed to use an experimental repository as follows:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt-get update
sudo apt-get install gcc-4.7 g++-4.7
```

More details can be found on the AskUbuntu web-site.

The Intel compiler has a powerful math library, which allows one to efficiently compute several hard distances functions such as the KL-divergence, the Jensen-Shanon divergence/metric, the L_p spaces for non-integer values of p. Unlike our custom implementation, which is used for non-Intel compilers, the Intel math library does not rely on tricks like pre-computation of logarithms at index time and allows, therefore, to store data in less space. The Intel provides the compiler for individual (i.e., non-commercial use for free). The details can be found on the Intel web site.

4 Quick start

To compile, go to the directory similarity_search and type:

```
cmake .
make
```

This creates a binary (a benchmarking utility) `experiment` that runs the actual benchmarks. Examples of using this benchmarking utility can be found in the directory sample_scripts. Please, check the script sample_run.sh. In addition, the directory `sample_scripts` contains the full set of scripts that can be used to re-produce our NIPS'13 and SISAP'13 results [**?**,**?**]. This includes the software to generate plots. Additionally, to reproduce our previous results, one needs to obtain a complete data set using the script data/get_all_data.sh. The complete set contains the following:

- The data set created by Lawrence Cayton. To download, use the script data/-download_cayton.sh;
- The Colors data set, which comes with the Metric Spaces Library[**?**]. To download, use the script data/download_colors.sh;
- The Wikipedia tf-idf vectors in the sparse format. To download, use the script data/download_wikipedia.sh;
- A synthetic, randomly generated, 64-dimensional data set, where each coordinate is a real number sampled independently from $U[0, 1]$:
  `data/genunif.py -d 64 -n 500000 -o unif64.txt`

Note that all data sets, except the Wikipedia data set, are vectors in the dense format (see § 6). If we use any of them, please consider citing the sources (see Section 10) for details. Also note that, the data will be downloaded in the **compressed** form. You would need the standard `gunzip` to uncompress all the data except the Wikipedia sparse vectors. The Wikipedia vectors are compressed using `7z`, which provides superior compression ratios.

5 Building and running the code (in detail)

5.1 Compiling

Implementation of similarity search methods is in the directory `similarity_search`. The code is built using a `cmake`, which works on top of the GNU make. To create makefiles for a release version of the code, type:

```
cmake -DCMAKE_BUILD_TYPE=Release .
```

If you did not create any makefiles before, you can shortcut by typing:

```
cmake .
```

To create makefiles for a debug version of the code, type:

```
cmake -DCMAKE_BUILD_TYPE=Debug .
```

When makefiles are created, just type:

```
make
```

The build process creates the following binaries:

- `bunit` is used to perform unit tests and evaluate the speed of various distance functions;
- `tune_vptree` implements a grid search procedure to find optimal VP-tree parameters (see our paper for details [**?**]);
- `experiment` is a main binary (the benchmarking utility) that runs benchmarks and saves evaluation results.

Note that a shortcut `cmake .` (re)-creates makefiles for the previously created build. When you type `cmake .` for the first time, it creates release makefiles. However, if you create debug makefiles and then type `cmake .`, this will not lead to creation of release makefiles!

5.2 Running Benchmarks

There is a one `single` benchmarking utility `similarity_search/release/experiment` that includes implementation of all methods. It has multiple options, which specify, among others, a space, a data set, a type of search, and a list of methods to test (with parameters). These options and their use cases are described below.

5.2.1 Space and distance value type A distance function can return an integer (`int`), a single-precision (`float`), or a double-precision (`double`) real value. A type of the distance and its value is specified as follows:

```
-s [ --spaceType ] arg       space type, e.g., l1, l2, lp:p=0.25
--distType arg (=float)      distance value type:
                             int, float, double
```

A description of a space may contain parameters. In this case, there is colon after the space benchmark label followed by a comma-separated (not spaces) list of parameters in the format: `<parameter name>=<parameter value>`. Currently, this is used only for L_p spaces. For instance, `lp:0.5` denotes the space $L_{0.5}$. A detailed list of possible spaces and respective distance functions is given in Table 2 in § 6.

For real-valued distance functions, one can use either single- or double-precision type. Single-precision is a recommended default.[3] We do not have a space with integer-valued distance function, but we plan to implement an edit distance in the nearest future.

5.2.2 Input Data/Test Set There are three options that define the data to be indexed:

```
-i [ --dataFile ] arg         input data file
--maxNumData arg (=0)         if non-zero, only the first
                              maxNumData elements are used
-d [ --dimension ] arg (=0)   optional dimensionality
```

The input file can be indexed either completely, or partially. In the latter case, the user can create the index using only the first `--maxNumData` elements. In the case of vector-space data, the dimensionality is determined by the number of columns in the data file. The user may choose to restrict the dimensionality and use only the first `--dimension` columns.

For testing, the user can use a separate test set. It is, again, possible to limit the number of queries:

```
-q [ --queryFile ] arg        query file
--maxNumQuery arg (=1000)     if non-zero, use maxNumQuery query
                              elements(required in the case
                              of bootstrapping)
```

If a separate test set is not available, it can be simulated by bootstrapping. To this, end the `--maxNumData` elements of the original data set are randomly divided into testing and indexable sets. The number of queries in this case is defined by the option `--maxNumQuery`. A number of bootstrap iterations is specified through an option:

```
-b [ --testSetQty ] arg (=0) # of sets created by bootstrapping;
```

Benchmarking can be carried out in either a single- or a multi-threaded mode. The number of test threads are specified as follows:

```
--threadTestQty arg (=1)   # of threads
```

[3] It is not clear yet, if having double-precision distance functions is essential. Yet, we decided to keep them. Thanks to C++ templates, it requires very little additional effort.

5.2.3 Query Type Our framework supports the k-NNand the range search. The user can request to run both types of queries:

```
-k [ --knn ] arg          comma-separated Ks for k-NN search
-r [ --range ] arg        comma-separated values for range search
```

For example, by specifying the options

```
--knn 1,10 --range 0.01,0.1,1
```

the user requests to run queries of five different types: 1-NN, 10-NN, as well three range queries with radii 0.01, 0.1, and 1.

5.2.4 Methods The following is an option to specify search methods:

```
-m [ --method ] arg       list of method(s) with parameters
```

Methods, similar to spaces, accept parameters. In this case, the name of the method is followed by a colon and a comma-separated list (no-spaces) of parameters in the format: `<parameter name>=<parameter value>`. For a detailed list of methods and their parameters, please, refer to § 7.

5.2.5 Saving and Processing Benchmark Results The benchmarking utility outputs a detailed report (including all the log entries) to the screen (we plan to improve logging in the nearest future). To save benchmarking results to a file, on needs to specify a parameter:

```
-o [ --outFilePrefix ] arg    output file prefix
```

In fact, we create two files: a human-readable report (suffix `.rep`) and a tab-separated data file (suffix `.data`). By default, the benchmarking utility creates files from scratch. The following option can be used to make it work in the append mode:

```
--appendToResFile arg (=0)    append mode flag
```

For information on processing and interpreting results see § 5.3.

5.3 Measuring Performance and Interpreting Results

5.3.1 Efficiency Three types of efficiency indicators are used: query runtime, the number of distance computations, and the amount of memory used by the index *and* the data. We also measure the improvement in runtime (improvement in efficiency) and in the number of distance computations compared to a sequential scan method. For each query, this method reads compares data objects against the query. The sequential search baseline processes *all the objects*.

The amount of memory consumed by a search method is measured indirectly: We record the overall memory usage of a benchmarking process before and after creation of the index. Then, we add the amount of memory used by the data. Memory used is computed by querying special file `/dev/<process id>/status`. This works only for Linux as we do not have a portable code to measure memory consumption of a process.

5.3.2 Effectiveness Several effectiveness metrics are computed by the benchmarking utility:

- A *number of points closer* to the query than the nearest returned point. Let $\text{pos}(o_i)$ represent a positional distance from o_i to the query, i.e., the number of objects closer to the query than o_i plus one. In the case of ties, we assume that the object with a smaller index is closer to the query. Note that $\text{pos}(o_i) \geq i$.
- A *relative position* error is equal to $\text{pos}(o_i)/i$;
- *Recall*, which is is equal to the fraction of all correct answers retrieved.

The first two metrics represent a so-called rank (approximation) error: if we sort (i.e., rank) all the data objects with respect to their distances to the query, where do we place objects returned by the search method? The closer the returned objects are to the beginning of the list (i.e., the closer they are to to the query object), the better is the quality of the search response.

Recall is a classic metric. It was argued, however, that recall does not account for position information of returned objects and is, therefore, inferior to rank error metrics [**?**,**?**]. If we had ground-truth queries and relevance judgements from human assessors (e.g., if a vector represents an image, it is similar to the query image?), we could in principle compute even more realistic effectiveness metrics such as the mean average precision, or the normalized discounted cumulative gain. This remains for the future work.

5.4 Interpreting and Processing Benchmark Results

If the user specifies the option `--outFilePrefix`, the benchmarking results are saved to two files: the file in a human-readable format, and a tab-separated data file intended for automatic processing.

The data file contains only the average values, which can be used to, e.g., produce efficiency-effectiveness plots. A sample script for doing this so can be found in the sample_scripts directory. The human-readable report also provides 95% confidence intervals. In the case of bootstrapping, statistics collected from several iterations, are aggregated using a classic fixed-effect model adopted in meta analysis [**?**]. Note for all metrics, except relative error, an average is computed using an arithmetic mean. For the relative error, however, we use the geometric mean [**?**].

An example of human readable report (*confidence intervals* are in square brackets) is given in Table 1.

6 Spaces

Currently we support only vector spaces. The input files can come in either regular, i.e., dense, or sparse variant. In both cases vectors are stored as a plain-text file, one vector per row. Yet, the vector formats are different.

In the regular, dense-vector, format, each row contains the same number of vector elements, one per each dimension. The values can be separated by spaces or commas/-columns. In the sparse format, each vector elements is preceded by a *zero-based* vector element id. The ids can be unsorted, but they should not repeat. For example, the following line describes a vector with three explicitly specified values, which represent vector elements 0, 25, and 257:

```
0 1.234  25 0.03 257 -3.4
```

Most values are missing and are assumed to have a default value. It is up to a designer of the space to decide on the default. Yet, all current implementations choose the *zero* default value. Again, elements can be separated by spaces or commas/-columns instead of spaces.

Table 1: An example of a human-readable report

```
===================================
vptree: triangle inequality
===================================
# of points: 9900
# of queries: 100
-----------------------------------
Recall:         0.83 -> [0.756 0.903]
RelPosError:    1.33 -> [1.14   1.55]
NumCloser:      2.05 -> [-0.11  4.21]
-----------------------------------
QueryTime:      2.04 -> [1.62   2.46]
DistComp:       472 ->  [1963  2981]
-----------------------------------
ImprEfficiency:3.97 -> [3.97   3.97]
ImprDistComp:  4    -> [4   4]
-----------------------------------
Memory Usage:  8.48 MB
-----------------------------------
```

Note: *confidence intervals* are in brackets

For a detailed list of spaces, their parameters, and performance characteristics is given in Table 2. The label of the space is passed to the benchmarking utility (see § 5.2). 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 (for the GNU C++) by precomputing 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 § 6.1-6.3.

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 query object is the first argument of the distance function) and another is for right queries (the query object is the second 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 for left and right queries.

Distance computation efficiency was evaluated on a Core i7 laptop (3.4 Ghz peak frequency) in a single-threaded mode (by the utility bunit). 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 computed using the GNU compiler. Somewhat higher efficiency numbers can be obtained by using the Intel 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). 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 a small sample file in the sample_data directory.

6.1 L_p-norms

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 Leo'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 labels, which are different from dense-space labels in that they contain a suffix `_sparse` (see also Table 2). For instance `l1` and `l1_sparse` are both L_1 spaces, but the former is dense and the latter is sparse. The labels 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 name *lp*, which is used in combination with a parameter. For instance, L_3 is denoted as `lp:p=3`.

Table 2: Description of implemented spaces

Space	**Label&Formula**	**Efficiency** (million op/sec)
	Metric Spaces	
Hamming	`bit_hamming` $\sum_{i=1}^{n} \lvert x_i - y_i \rvert$	240
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` $\arccos\left(1 - \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}\right)$	13, 1.4
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
	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 similarity	`cosinesimil`, `cosinesimil_sparse` $1 - \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$	13, 1.4
	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

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).

6.2 Scalar-product Related Distances

We have two distance function whose formulas include normalized scalar product. One is the cosine similarity, 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 similarity is not a true metric, but it can be converted into one by applying a monotonic transformation (i.e.., taking an inverse cosine). The resulting distance function is a true metric that is called the angular distance. The angular distance is computed using the following formula:

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

6.2.1 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 [**?**], 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. To this end, we rewrite the equation 3 as follows:

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

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

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

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

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 operations to implement this approach. As our tests show, 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 use an efficient approximation for logarithm computation. However, as our tests show, this method is still relatively slow (it takes almost 20 CPU cycles per log), while the relative error is as high as $3 \cdot 10^{-4}$ for logarithm arguments smaller than 2.

6.3 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 [**?**,**?**]. More formally, given the convex and differentiable function f (of many variables), its corresponding Bregman divergence $d_f(x,y)$ is equal to:

$$d_f(x,y) = f(x) - f(y) - \langle \nabla f(y), x-y\rangle$$

where $\langle x,y\rangle$ 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 evaluation 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 labels (see also Table 2).

7 Search Methods

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

- Pivoting and compact partitioning methods for metric spaces [**?**];
- Filtering methods based on permutations;
- Filtering methods based on random projections;
- A specialized method (bbtree) for Bregman divergences [**?**];
- A vantage-point tree (a VP-tree) with pluggable search oracles, which can be used in both metric and non-metric spaces.

Pivoting methods rely on embedding into a vector space with the L_∞ distance. Partitioning is based on how far (or close) the data points are located with respect to pivots. If the original space is a metric spacing such an embedding is contractive: The L_∞ distance in the target vector space is a lower bound for the original distance.

Hierarchical partitions produced by pivoting methods lack locality: a single partition can contain far away data points. Compact partitioning schemes exploit locality. They either divide the data into clusters or use a Voronoi partitioning (or its approximation). In the latter case, for example, we can select several centers/pivots π_i and associate data points with the closest center.

These classic search methods are exact. It is possible to convert most of them to approximate methods using an early termination technique, where we terminate the search after exploring a sufficient number of data points. The data points are often grouped into buckets (also called leaves) and the early termination condition is defined in terms of the maximum number of buckets to visit (before terminating). To implement such a technique, we need to 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 can greedily explore partitions containing the query.

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 $r > |R - d(\pi, q)|$, we cannot be sure that all sought data points are in the same partition as q. Thus, we visit both partitions.

The pruning condition based on the triangle inequality can be overly pessimistic. By selecting some $\alpha > 1$ and pruning when $r \leq \alpha|R - d(\pi, q)|$, we 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, an almost negligible loss in recall (by 1-5%) can lead to an order of magnitude faster retrieval.

Besides being too pessimistic, the triangle inequality cannot be used directly in non-metric spaces: the recall can be too low or retrieval time be too long. Yet, it is often possible to searching using a relaxed version, with α possibly smaller than one. In this version, we would use different α for different partitions. 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 previously described piece-wise linear function is a special case of the search oracle described by the formula:

$$D_{\pi,R}(x) = \begin{cases} \alpha_{left}|x - R|, & \text{if } x \leq R \\ \alpha_{right}|x - R|, & \text{if } x \geq R \end{cases} \tag{5}$$

Optimal α_{left} and α_{right} are determined using a grid search. To this end, we index a small subset of the data points and seek to obtain parameters that give the shortest retrieval time at a specified recall threshold. Another approach to learn $D()$, which did not work out well yet, is to use sampling. It is possible to implement new search oracles and plug them into the implementation of the VP-tree.

Table 3: Description of implemented methods

Tree-based data methods (except permutation methods)			
Method	**Label**	**Parameters**	**Notes**
VP-tree [**?**,**?**]	`vptree`	`alphaLeft`, `alphaRight`: see (5) `bucketSize`, `chunkBucket`, `maxLeavesToVisit`	Employs a piecewise linear oracle [**?**]. Also, see the description of common parameters below.
Multi-Vantage Point Tree [**?**]	`mvptree`	`bucketSize`, `chunkBucket`, `maxLeavesToVisit` `maxPathLen`: the number of preceeding pivot pairs distances to which we remember during searching	A **binary** version of the Multi-Vantage Point Tree
VP-tree over projections (**only** for sparse cosine similarity and angular distance spaces)	`proj_vptree`	`projPivotQty`: a number of random projections `projMaxElem`: a maximum number of vector elements to be used in computation of projections. `dbScanFrac`: a fraction of the candidate records retrieved from the VP-tree `alphaLeft`, `alphaRight`: see (5)	Also, see the description of common parameters below.
GHtree [**?**]	`ghtree`	`bucketSize`, `chunkBucket`, `maxLeavesToVisit`	See the description of common parameters below
List of clusters [**?**]	`list_clusters`	`strategy`: pivot selection strategy (random, closestPrevCenter, farthestPrevCenter, minSumDistPrevCenters, maxSumDistPrevCenters) `useBucketSize`: use the size of the bucket to determine the cluster (0,1) `radius`: the radius of the cluster that is used to determine clusters (if `useBucketSize` is 0). `bucketSize`, `chunkBucket`, `maxLeavesToVisit`	See the description of common parameters below
Spatial approximation tree [**?**]	`satree`	`no parameters`	
bbtree [**?**]	`bbtree`	`bucketSize`, `chunkBucket`, `maxLeavesToVisit`	See the description of common parameters below

Common parameters for tree-based methods	
`bucketSize`:	A maximum number of elements in a bucket/leaf.
`chunkBucket`:	Indicates if bucket elements should be stored contiguously in memory (1 by default). If `chunkBucket` is 1, when the bucket is created we allocate a new chunk of memory that contains a copy of all bucket vectors. This increases memory consumption, but speeds up searching.
`maxLeavesToVisit`:	This parameter, equal to the maximum number of leaves/buckets visited by a search algorithm, controls early termination. By default, `maxLeavesToVisit` is a very large number, i.e., no early termination is employed.

Locality Sensitive Hashing (LSH) methods			
Method	**Label**	**Parameters**	**Notes**
Multiprobe LSH [**?**,**?**] (**only** for L_2)	`lsh_multiprobe`	`M, W, H, L` `T`: a number of probes `desiredRecall`: a desired recall `tuneK`: find optimal parameter for k-NN, search where k is defined by this parameter	This method finds optimal parameters (for `M`,`W`) using a model described by Dong et al. [**?**]. Note that parameters `L` and `T` are specified by the user: The code will not try to find their optimal values. Alos, see the description of common parameters below.
LSH (Cauchy) [**?**] (**only** for L_1)	`lsh_cauchy`	`M, W, H, L`	See the description of common parameters below
LSH (Gaussian) [**?**] **only** for L_2)	`lsh_gaussian`	`M, W, H, L`	See the description of common parameters below
LSH (Thresholding) [**?**] (**only** for L_1)	`lsh_threshold`	`M, H, L`	See the description of common parameters below

Common parameters for LSH methods	
`W`:	A width of the window [**?**].
`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.

Permutation-based filtering methods

Method	Label	Parameters	Notes
Sequential-search [**?**]	`permutation`	`numPivot`, `dbScanFrac`	See the description of common parameters below.
Sequential-search with incremental sorting [**?**]	`perm_incsort`	`numPivot`, `dbScanFrac`	See the description of common parameters below.
Sequential-search with incremental sorting, but using binarized permutations (similar to [**?**])	`perm_incsort_bin`	`numPivot`, `dbScanFrac`, `binThreshold`: binarization threshold.	See the description of common parameters below.
PP-index [**?**]	`perm_prefix`	`minCandidate`: a minimum number of candidates to retrieve `prefixLength`: a maximum length of the tree prefix that is used to retrieve candidate records. `chunkBucket`: the same meaning as in the above tree-based data structures. `numPivot`, `dbScanFrac`	See also the description of common parameters below.
Inverted file-based search [**?**]	`perm_inv_indx`	`numPivot`, `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	See also the description of common parameters below.
VP-tree index for permutations. Similar to [**?**], but uses an approximate search in the VP-tree.	`perm_vptree`	`numPivot`, `dbScanFrac`, `alphaLeft`, `alphaRight` see (5)	See also the description of common parameters below.
Brief permutation index: VP-tree index for binarized permutations. Similar to [**?**], but uses an approximate search in the VP-tree.	`perm_bin_vptree`	`numPivot`, `dbScanFrac` `binThreshold`: binarization threshold. `alphaLeft`, `alphaRight`: see (5)	See also the description of common parameters below.

Common parameters for permutation-based filtering methods

`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.

Auxilliary methods			
Several copies of the same index type	`mult_index`	`indexQty`: a number of copies `methodName`: method label/name	The user can specify any other parameter that the method `methodName` accepts
Exhaustive/sequntial search	`seq_search`		Can be used to determine scalability of exhaustive search with respect to the # of threads

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 essentiall 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 soruce vector space with real coordinates to a target vector space with integer coordinates. A distance in the target space (typically L_1 or L_2) correlates well with the distance in the source space. This property is used in permutation methods.

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. The number of candidate records retrieved by (most) permutation algorithms can be controlled by a parameter `dbScanFrac` or `minCandidate` (see Table 3 for details). 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 [**?**].

Another filtering method maps data from the original space to a target lower dimensional space using random projections. In the target space, the random projections are indexed using a VP-tree [**?**,**?**].

For Bregman divergences (see § 6.3) it is possible to devise an exact search algorithm [**?**], which hierarchically divides the data set into two clusters at index time (there exists an efficient clustering algorithm for Bregman divergence [**?**]). At search time, the algorithm relies on properties of Bregman divergences to compute distances from query points to Bregman balls, which represent clusters constructed at index time [**?**].

A complete list of the methods, including their labels and parameters, is given in Table 3

8 Extending the code

It is possible to add new spaces and search methods. This is done in two steps. In the first step, the user writes the code that implements a functionality of a method or a space. In the second step, the user writes a special helper file that registers the space or method in a factory class. This helper files need to be placed into a special directory. After that ,the user needs to re-run `cmake` as described in § 5. No modification of makefiles (or other configuration files) is required.

Is is noteworthy that all implementations of methods and spaces are template classes parameterized by the distance value type. Recall that the distance function can return an integer (`int`), a single-precision (`float`), or a double-precision (`double`) real value. The user may choose to provide specializations for all possible distance values or decide to focus, e.g., only on integer-valued distances.

The user can also add new applications. However, adding new applications does require minor editing of the meta-makefile `CMakeLists.txt` (and re-running `cmake` § 5).

In the following subsections, we consider extension tasks in more detail. For illustrative purposes, we created a zero-functionality space (`DummySpace`), method (`DummyMethod`), and application (`dummy_app`). These dummy classes can also be used as starting points to develop fully functional code.

8.1 Test Workflow

The main benchmarking utility `experiment` parses command line parameters. Then, it creates a space and all required search methods using the space and the method factories. In our library, we implement only in-memory indices. Thus, when we create a class representing a search method, the constructor of this class has to create an index. Both search method and spaces can have parameters, which are passed to the method/space in an instance of the class `AnyParams`. We consider this in detail in § 8.2 and § 8.3.

Depending on parameters, two test scenarios are possible. In the first scenario, the user specifies separate data and test files. In the second scenario, a test file is created by bootstrapping: The data set is randomly divided into training and a test set. Then, we call the function RunAll and subsequently Execute for all possible test sets.

The function `Execute` is a main workhorse, which creates queries, runs searches, produces ground truth data, and collects execution statistics. There are two types of queries: nearest-neighbor and range queries, which are represented by (template) classes `RangeQuery` and `KNNQuery`. Both classes inherit from the class

`Query`. Similar to spaces, these template classes are parameterized by the type of the distance value.

Both types of queries are similar in that they implement the `Radius` function and the functions `CheckAndAddToResult`. In the case of the range query, the radius of a query is constant. However, in the case of the nearest-neighbor query, the radius typically decreases as we compare the query with new data objects (by calling the function `CheckAndAddToResult`). In both cases, the value of the function `Radius` is used to prune unpromising partitions and data points.

This commonality between the `RangeQuery` and `KNNQuery` allows us in many cases to carry out a nearest-neighbor query using an algorithm designed for range queries. Thus, only a single implementation of a search method–that answers queries of both types–can be used in many cases.

A query object proxies distance computations during the testing phase. Namely, the distance function is accessible through the function `IndexTimeDistance`, which is defined in the class `Space`. During the testing phase, a search method can compute a distance only by accessing functions `Distance`, `DistanceObjLeft` (for left queries) and `DistanceObjRight` for right queries, which are member functions of the class `Query`. The function `Distance` accepts two parameters (i.e., object pointers) and can be used to compare two arbitrary objects. The functions `DistanceObjLeft` and `DistanceObjRight` are used to compare data objects with the query. Note that it is a query object memorizes the number of distance computations. This allows us to compute the variance in the number of distance evaluations and, consequently, a respective confidence interval.

8.2 Creating a space

A space is a collection of data objects. In our library, objects are represented by instances of the class `Object`. The functionality of this class is limited to creating new objects and/or their copies as well providing access to the raw (i.e., unstructured) representation of the data (through functions `data` and `datalength`). We would re-iterate that currently (though this is likely to change in the future), `Object` is a very basic class that only keeps a blob of data and remembers its size. For example, the `Object` can store an array of single-precision floating point numbers, but it has no function to obtain the number of elements. These are the spaces that are responsible for reading objects from files, interpreting the structure of the data blobs (stored in the `Object`), and computing a distance between two objects.

To explain the basics of developing a new space, we created a sample zero-functionality space `DummySpace`. It is represented by the header file space_dummy.h and the source file space_dummy.cc. The user is encouraged to study these files and read the comments. Here we focus only on the main aspects of creating a new method.

The sample files describe a template class `DummySpace`, which is declared and defined in the namespace `similarity`. It is a direct ancestor of the class `Space`:

```
template <typename dist_t>
class SpaceDummy : public Space<dist_t> {
 public:
  ...
  virtual void ReadDataset(ObjectVector& dataset,
                      const ExperimentConfig<dist_t>* config,
                      const char* inputfile,
                      const int MaxNumObjects) const;
 protected:
  virtual dist_t HiddenDistance(const Object* obj1,
                                const Object* obj2) const;
}
```

It is possible to provide the complete implementation of the `DummySpace` in the header file. However, this would make compilation very slow. Instead, we recommend to use the mechanism of explicit template instantiation. To this end, the user should instantiate the template in the source file for all possible combination of parameters. In our case, the *source* file space_dummy.cc contains the following lines:

```
template class SpaceDummy<int>;
template class SpaceDummy<float>;
template class SpaceDummy<double>;
```

Most importantly, the user needs to implement the function `ReadDataset`, which reads objects from a file, and the function `HiddenDistance`, which computes the distance between objects. For a sample implementation of `ReadDataset`, please, see the file space_bit_hamming.cc. Note that `ReadDataset` is supposed to read at most `MaxNumObjects` from the file. If the file has more objects, only the first `maxNumData` should be retrieved. The space is responsible for following this convention, the library does not enforce this behavior.

Remember that the function `HiddenDistance` should not be directly accessible by classes that are not friends of the `Space`. As explained in § 8.1, during the indexing phase, `HiddenDistance` is accessible through the function `Space::IndexTimeDistance`. During the testing phase, a search method can compute a distance only by accessing functions `Distance`, `DistanceObjLeft`, or `DistanceObjRight`, which are member functions of the `Query`.

Finally, we need to "tell" the library about the space, by registering the space in the space factory. An example of registering the space `SpaceDummy` is given in the file space_dummy. **It is very important** to place the registration code in the subdirectory `similarity_search/src/factory/space`. Otherwise, for certain platforms and compilers, this code may not be executed.

At runtime, the space is created through a helper function. In our case, it is called `CreateDummy`. The function, accepts only one parameter, which is a reference to an object of the type `AllParams`:

```
template <typename dist_t>
Space<dist_t>* CreateDummy(const AnyParams& AllParams) {
  AnyParamManager pmgr(AllParams);

  int param1, param2;

  pmgr.GetParamRequired("param1",  param1);
  pmgr.GetParamRequired("param2",  param2);

  return new SpaceDummy<dist_t>(param1, param2);
}
```

To extract parameters, the user needs an instance of the class `AnyParamManager` (see the above example). In most cases, it is sufficient to call two functions: `GetParamOptional` and `GetParamRequired`. Parameter values specified in the commands line are interpreted as strings. The `GetParam*` functions can convert these string values to integer or floating-point numbers if necessary. A conversion occurs, if the type of a receiving variable (passed as a second parameter to the functions `GetParam*`) is different from a string. It is possible to use boolean variables as parameters. In that, in the command line, one has to use 1 (for `true`) or 0 (for `false`). Note that the function `GetParamRequired` raises an error, if the request parameter was not supplied in the command line.

The function `CreateDummy` is registered in the space factory using a special macro. This macro should be used for all possible values of the distance function, for which our space is defined. For example, if the space is defined only for integer-valued distance function, this macro should be used only once. However, in our case the space `CreateDummy` is defined for integers, single- and double-precision floating pointer numbers. Thus, we use this macro three times as follows:

```
REGISTER_SPACE_CREATOR(int,    SPACE_DUMMY,  CreateDummy)
REGISTER_SPACE_CREATOR(float,  SPACE_DUMMY,  CreateDummy)
REGISTER_SPACE_CREATOR(double, SPACE_DUMMY,  CreateDummy)
```

To include new files into the build process, the user needs to re-run `cmake` as described in § 5.

8.3 Creating a method

To explain the basics of developing a new search method, we created a sample zero-functionality method `DummyMethod`. It is represented by the header file dummy.h and the source file dummy.cc. The user is encouraged to study these files and read the comments. Here we would omit certain minor details.

Similar to the space and query classes, a search method is implemented using a template class, which is parameterized by the distance function value:

```
template <typename dist_t>
class DummyMethod : public Index<dist_t> {
 public:
  DummyMethod(const Space<dist_t>* space,
              const ObjectVector& data,
              bool bDoSeqSearch)
              : data_(data), bDoSeqSearch_(bDoSeqSearch) {}
  ~DummyMethod(){};
  const std::string ToString() const;

  void Search(RangeQuery<dist_t>* query);
  void Search(KNNQuery<dist_t>* query);

 private:
  const ObjectVector&     data_;
  // disable copy and assign
  DISABLE_COPY_AND_ASSIGN(DummyMethod);
};
```

Note that it is the constructor that creates a search index (or calls a function to create it)! Here it accepts the pointer to a space, a reference to an array of data objects, and an additional parameter. When this parameter is true, our dummy method will carry out a sequential search. Otherwise, it does nothing useful.

The space object is typically used to compute the distance by calling the function `IndexTimeDistance`. Note again that `IndexTimeDistance` **should not be used** in functions `Search`. If the user attempts to invoke `IndexTimeDistance` during the test phase, **the program will terminate**. As noted previously, we want to compute the number of times the distance was computed for each query. This allows us to estimate the variance. Hence, during the testing phase, the distance function should be invoked only through a query object.

Finally, we need to "tell" the library about the method, by registering the method in the method factory. This done similarly to registering a space. An example of registering the method `MethodDummy` is given in the file dummy.cc. **It is very important** to place the registration code in the subdirectory `similarity_search/src/factory/space`. Otherwise, the registering code may not execute (for certain platforms and compilers).

At runtime, the method is created through a helper function. In our case, it is called `CreateDummy`. This function accepts several parameters, one which is a reference to an object of the type `AllParams`:

```
template <typename dist_t>
Index<dist_t>* CreateDummy(bool PrintProgress,
                           const string& SpaceType,
                           const Space<dist_t>* space,
                           const ObjectVector& DataObjects,
                           const AnyParams& AllParams) {
    AnyParamManager pmgr(AllParams);
    bool bDoSeqSearch = false;
    pmgr.GetParamOptional("doSeqSearch",  bDoSeqSearch);

    return new DummyMethod<dist_t>(space, DataObjects, bDoSeqSearch);
}
```

Note that, similarly to the dummy space example, we use a parameter extraction class `AnyParamManager`. The only difference, is that we retrieve an optional parameter `doSeqSearch`. If this parameter is not present in the command line, the code uses an old value of the variable `bDoSeqSearch` (which it had before we called `GetParamOptional`). Again, similarly to space, the method-creating function `CreateDummy` needs to be registered in the space factory as follows:

```
REGISTER_METHOD_CREATOR(float,  METH_DUMMY, CreateDummy)
REGISTER_METHOD_CREATOR(double, METH_DUMMY, CreateDummy)
REGISTER_METHOD_CREATOR(int,    METH_DUMMY, CreateDummy)
```

Imagine that we want our method to work only with integer-valued distances. Then, we only need the following line:

```
REGISTER_METHOD_CREATOR(int,    METH_DUMMY, CreateDummy)
```

To include new files into the build process, the user needs to re-run `cmake` as described in § 5.

8.4 Creating an application

First, we create a hello-world source file dummy_app.cc:

```
#include <iostream>

using namespace std;
int main(void) {
  cout << "Hello world!" << endl;
}
```

Now we need to modify the meta-makefile similarity_search/src/CMakeLists.txt and re-run `cmake` as described in § 5.

More specifically, we do the following:

- by default, all source files in the similarity_search/src/ directory are included into the library. To prevent `dummy_app.cc` from being included into the library, we use the following command:

```
list(REMOVE_ITEM SRC_FILES ${PROJECT_SOURCE_DIR}/src/dummy_app.cc)
```

- tell `cmake` to build an additional executable:

```
add_executable (dummy_app dummy_app.cc ${SRC_FACTORY_FILES})
```

- specify the necessary libraries:

```
target_link_libraries (dummy_app NonMetricSpaceLib lshkit
                       ${Boost_LIBRARIES} ${GSL_LIBRARIES}
                       ${CMAKE_THREAD_LIBS_INIT})
```

9 Notes on Efficiency

9.1 Efficient 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. 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. Thus, we would encourage potential contributors to implement efficient distance functions whenever it is possible (should they decide to create a new space).

In this library, we optimized several distance functions, especially non-metric functions that involve computations of logarithms. In the case of the Intel compiler, logarithms can be computed reasonably fast. However, most users still rely on the GNU C++ compiler and, consequently, on the GNU math library, where computation of a logarithm takes dozens of CPU cycles. 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 § 6.3 for more details). Yet, this doubles the size of an index.

Efficient implementations of some other distance functions (see § 9) rely on Single Instruction Multiple Data (SIMD) CPU instructions. These instructions, available on most modern Intel and AMD processors, operate on small vectors. Previously, we found that these optimized implementations were always much faster than pure C++ versions. Recently, we realized that with appropriate compiler flags, 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. We are, nevertheless, willing to keep and maintain our customized functions, because we cannot expect that the compiler would be always able to vectorize these distance functions. In particular,

if we port the code to Windows or Mac. For example, the compiler apparently fails to efficiently vectorize computation of the KL-divergence (with precomputed logarithms).

Even though distance computation is efficient for dense vector spaces, it is challenging (but possible) to implement efficient distance functions for sparse vector spaces. Our current implementations are not especially efficient and their optimization remains as future work.

9.2 Cache-friendly Data Layout

In our previous report [**?**], we underestimated a cost of a random memory access. A more careful analysis showed that, on the author's laptop (Core i7, DDR3), a truly random access "costs" about 200 CPU cycles, which may be 2-3 times longer than a single distance computation.

Many implemented methods use some form of bucketing. For example, in the VP-tree we recursively decompose the space until partitions become sufficiently small. The buckets are searched sequentially, which can be done much faster, if bucket objects are stored in contiguous memory regions. Thus, to check elements in a bucket we 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: A bucket 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.

10 Credits

The code that was written entirely by the authors is distributed under the business-friendly Apache License. The best way to acknowledge the use of this code in a scientific publication is to provide the URL of the GitHub repository[4] and to cite our engineering paper:

```
@incollection{Boytsov_and_Bilegsaikhan:sisap2013,
  year={2013},
  isbn={978-3-642-41061-1},
  booktitle={Similarity Search and Applications},
  volume={8199},
  series={Lecture Notes in Computer Science},
  editor={Brisaboa, Nieves and Pedreira, Oscar and Zezula, Pavel},
```

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

```
  doi={10.1007/978-3-642-41062-8_28},
  title={Engineering Efficient and Effective
         \mbox{Non-Metric Space Library}},
  url={http://dx.doi.org/10.1007/978-3-642-41062-8_28},
  publisher={Springer Berlin Heidelberg},
  keywords={benchmarks; (non)-metric spaces; Bregman divergences},
  author={Boytsov, Leonid and Naidan, Bilegsaikhan},
  pages={280-293}
}
```

Most provided data sets are created by Lawrence Cayton. Our implementation of the bbtree, an exact search method for Bregman divergences, is also based on the code of Cayton. If you use any of these, please, consider citing:

```
@inproceedings{cayton:2008,
  title=     {Fast nearest neighbor retrieval for bregman divergences},
  author=    {Cayton, Lawrence},
  booktitle= {Proceedings of the 25th international conference on
             Machine learning},
  pages=     {112--119},
  year=      {2008},
  organization={ACM}
}
```

The Colors data set originally belongs to the Metric Spaces Library:

```
@misc{LibMetricSpace,
  Author =    {K.~Figueroa and G.{}~Navarro and E.~Ch\'avez},
  Keywords =  {Metric Spaces, similarity searching},
  Lastchecked = {August 18, 2012},
  Note = {Available at
         {\url{http://www.sisap.org/Metric\_Space\_Library.html}}},
  Title = {\mbox{Metric Spaces Library}},
  Year =  {2007}
}
```

The Wikipedia data set was created with a help of the `gensim` library:

```
@inproceedings{rehurek_lrec,
  title = {{Software Framework for Topic Modelling
            with Large Corpora}},
  author = {Radim {\v R}eh{\r u}{\v r}ek and Petr Sojka},
  booktitle = {{Proceedings of the LREC 2010 Workshop on New
               Challenges for NLP Frameworks}},
  pages = {45--50},
  year = 2010,
  month = May,
  day = 22,
```

```
  publisher = {ELRA},
  address = {Valletta, Malta},
  note={\url{http://is.muni.cz/publication/884893/en}},
  language={English}
}
```

Last, but not least, our library incorporates the efficient LSHKIT library. Note that is is distributed under a different license: GNU General Public License version 3 or later.

If you (re)-use it, please, consider citing the authors:

```
@inproceedings{Dong_et_al:2008,
  author =      {Dong, Wei and Wang, Zhe and Josephson, William
                 and Charikar, Moses and Li, Kai},
  title =       {Modeling LSH for performance tuning},
  booktitle =   {Proceedings of the 17th ACM conference on Information
                 and knowledge management},
  series =      {CIKM '08},
  year =        {2008},
  isbn =        {978-1-59593-991-3},
  location =    {Napa Valley, California, USA},
  pages =       {669--678},
  numpages =    {10},
  url =         {http://doi.acm.org/10.1145/1458082.1458172},
  doi =         {10.1145/1458082.1458172},
  acmid =       {1458172},
  publisher =   {ACM},
  address =     {New York, NY, USA},
 keywords = {locality sensitive hashing, similarity search},
}
```

References

1. G. Amato, F. Rabitti, P. Savino, and P. Zezula. Region proximity in metric spaces and its use for approximate similarity search. *ACM Trans. Inf. Syst.*, 21(2):192–227, Apr. 2003.
2. 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.
3. A. Banerjee, S. Merugu, I. S. Dhillon, and J. Ghosh. Clustering with bregman divergences. *The Journal of Machine Learning Research*, 6:1705–1749, 2005.
4. 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.

5. L. Boytsov and B. Naidan. Learning to prune in metric and non-metric spaces. In *Advances in Neural Information Processing Systems*, 2013.
6. T. Bozkaya and M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS)*, 24(3):361–404, 1999.
7. 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.
8. 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.
9. 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.
10. E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.
11. E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
12. 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.
13. W. Dong. *High-Dimensional Similarity Search for Large Datasets*. PhD thesis, Princeton University, 2011.
14. 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.
15. D. M. Endres and J. E. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, 2003.
16. A. Esuli. Use of permutation prefixes for efficient and scalable approximate similarity search. *Inf. Process. Manage.*, 48(5):889–902, Sept. 2012.
17. K. Figueroa and K. Frediksson. 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.
18. K. Figueroa, G. Navarro, and E. Chávez. Metric Spaces Library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.
19. 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.
20. L. V. Hedges and J. L. Vevea. Fixed-and random-effects models in meta-analysis. *Psychological methods*, 3(4):486–504, 1998.
21. G. King. How not to lie with statistics: Avoiding common mistakes in quantitative political science. *American Journal of Political Science*, pages 666–687, 1986.
22. 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.
23. G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46, 2002.
24. T. Skopal. Unified framework for fast exact and approximate search in dissimilarity spaces. *ACM Trans. Database Syst.*, 32(4), Nov. 2007.

25. 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.
26. J. Uhlmann. Satisfying general proximity similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.
27. 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.