# Non-Metric Space Library (NMSLIB) 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@cs.cmu.edu

**Maintainer**: Leonid Boytsov

Version 1.5

Friday 13th May, 2016

**Abstract.** This document describes a library for similarity searching. Even though the library contains a variety of metric-space access methods, our main focus is on search methods for non-metric spaces. Because there are fewer exact solutions for non-metric spaces, many of our methods give only approximate answers. Thus, the methods are evaluated in terms of efficiency-effectiveness trade-offs rather than merely in terms of their efficiency. 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. We concentrate on technical details, i.e., how to compile the code, run the benchmarks, evaluate results, and use our code in other applications. Additionally, we explain how to extend the code by adding new search methods and spaces.

## 1 Introduction

### 1.1 History, Objectives, and Principles

Non-Metric Space Library (NMSLIB) is an **efficient** cross-platform similarity search library and a toolkit for evaluation of similarity search methods. The goal of the project is to create an effective and **comprehensive** toolkit for searching in **generic non-metric** spaces. Being comprehensive is important, because no single method is likely to be sufficient in all cases. Because exact solutions are hardly efficient in high dimensions and/or non-metric spaces, the main focus is on **approximate** methods.

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, see § 2.6). In addition, it is also possible to build a query server (see § 2.5), 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.

Even though our methods are generic, they often outperform specialized methods for the Euclidean and/or angular distance (i.e., for the cosine similarity). Tables 1 and 2 contain results (as of May 2016) of NMSLIB compared to the best implementations participated in a public evaluation code-named `ann-benchmarks`. Our main competitors are:

- A popular library Annoy, which uses a forest of random-projection KD-trees [48].
- A new library FALCONN, which is a highly-optimized implementation of the multiprobe LSH [3]. It uses a novel type of random projections based on the fast Hadamard transform.

The benchmarks were run on a c4.2xlarge instance on EC2 (using a previously unseen subset of 5K queries). The benchmarks employ two data sets:

- GloVe: 1.2M 100-dimensional word embeddings trained on Tweets;
- 1M of 128-dimensional SIFT features.

Most search methods were implemented by Bileg(saikhan) Naidan and Leo(nid) Boytsov.[3] Additional contributors are listed on the GitHub page. Bileg and Leo gratefully acknowledge support by the iAd Center [4] and the Open Advancement of Question Answering Systems (OAQA) group [5].

The code written by Bileg and Leo is distributed under the business-friendly Apache License. However, some third-party contributions are licensed differently. For more information regarding licensing and acknowledging the use of the library resource, please refer to § 10.

The design of the library was influenced by and superficially resembles the design of the Metric Spaces Library [25]. Yet our approach is different in many ways:

- We focus on approximate[6] search methods and non-metric spaces.
- We simplify experimentation, in particular, through automatically measuring and aggregating important parameters related to speed, accuracy, index size, and index creation time. In addition, we provide capabilities for testing in both single- and multi-threaded modes to ensure that implemented solutions scale well with the number of available CPUs.
- We care about overall efficiency and aim to implement methods that have runtime comparable to an optimized production system.

[3] Leo(nid) Boytsov is a maintainer.

[4] `http://www.iad-center.com/`

[5] `http://oaqa.github.io/`

[6] An approximate method may not return a true nearest-neighbor or all the points within a given query ball.

Fig. 1: 1.19M vectors from GloVe (100 dimensions, trained from tweets), cosine similarity.

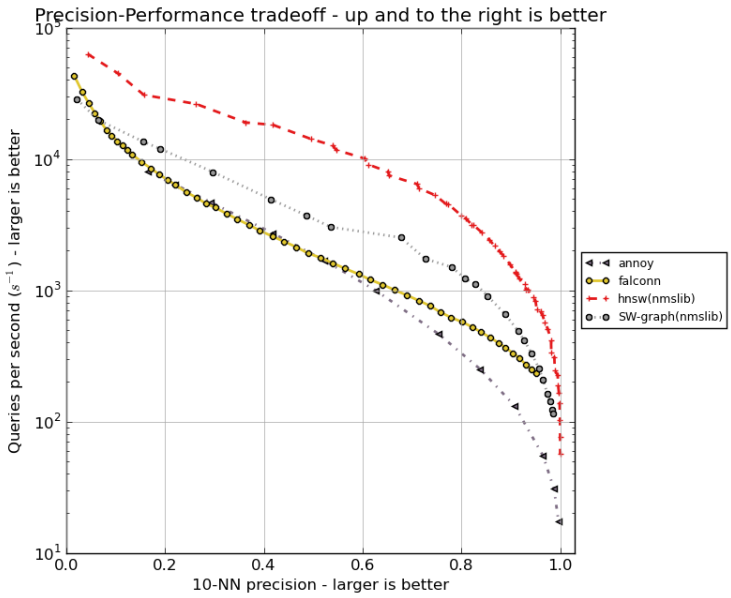
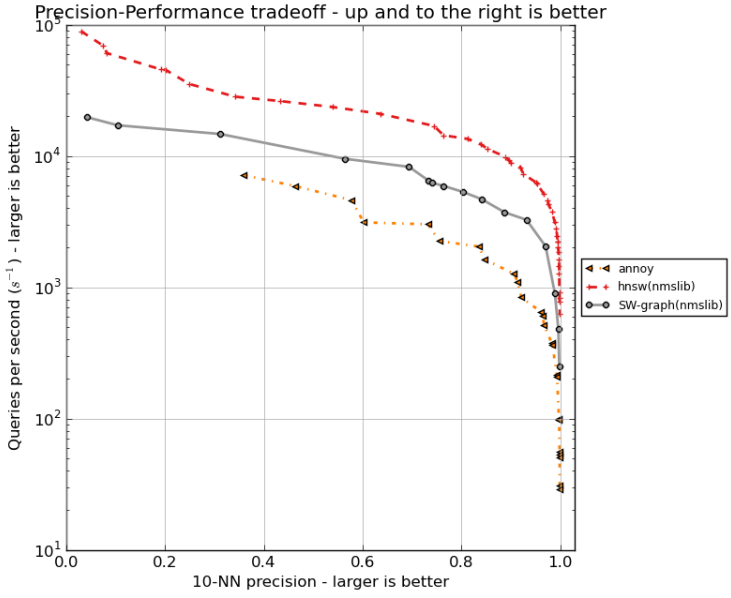



Fig. 2: 1M SIFT features (128 dimensions), Euclidean distance.



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 [49]). In many cases it is not possible to search exactly and instead one has to resort to approximate search procedures.

This is why methods are evaluated in terms of efficiency-effectiveness trade-offs rather than merely in terms of their efficiency. As mentioned previously, we believe that there is no "one-size-fits-all" search method. Therefore, it is important to provide a variety of methods each of which may work best for some specific classes of data.

Our commitment to efficiency affected several design decisions:

- The library is implemented in C++;
- We focus on in-memory indices and, thus, do not require all methods to materialize a disk-based version of an index (this reduces programming effort).
- We provide efficient implementations of many distance functions, which rely on Single Instruction Multiple Data (SIMD) CPU commands and/or approximation of computationally intensive mathematical operations (see § 8).

It is often possible to demonstrate a substantial reduction in the number of distance computations compared to sequential searching. However, such reductions may entail additional computations (i.e., extra book-keeping) and do not always lead to improved overall performance [7]. To eliminate situations where book-keeping costs are "masked" by inefficiencies of the distance function, we pay special attention to distance function efficiency.

### 1.2 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*. 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 $k$ 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 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. If ground-truth judgements (produced by humans) are available, it is also possible to compute an accuracy of a $k$-NN based classification (see § 3.5.2).

# 2 Getting Started

## 2.1 What's new in version 1.5 (major changes)

- We have adopted a new method: a hierarchical (navigable) small-world graph (HNSW), contributed by Yury Malkov [37], see § 5.5.2.
- We have improved performance of two core methods SW-graph (§ 5.5.1) and NAPP (5.4.5).
- We have written basic tuning guidelines for SW-graph, HNSW, and NAPP 6.
- We have modified the workflow of our benchmarking utility `experiment` and improved handling of the gold standard data, see § 3.4.6;
- We have updated the API so that methods can save and restore indices, see § 7.3.
- We have implemented a server, which can have clients in C++, Java, Python, and other languages supported by Apache Thrift, see § 2.5.
- We have implemented generic Python bindings that work for non-vector spaces, see § 2.6.
- Last, we retired older methods `permutation`, `permutation_incsort`, and `permutation_vptree`. The latter two methods are superseded by `proj_incsort` and `proj_vptree`, respectively.

## 2.2 Prerequisites

NMSLIB was developed and tested on 64-bit Linux. Yet, almost all the code can be built and run on 64-bit Windows (two notable exceptions are: LSHKIT and NN-Descent). It should also be possible to build the library on MAC OS. Building the code requires a modern C++ compiler that supports C++11. Currently, we support GNU C++ ($\geq$ 4.7), Intel compiler ($\geq$ 14), Clang ($\geq$ 3.4), and Visual Studio ($\geq$ 12)[7]. Under Linux, the build process relies on CMake. Under

[7] One can use the free express version.

Windows, one could use Visual Studio projects stored in the repository. These projects are for Visual Studio 14 (2015). However, they can be downgraded to work with Visual Studio 12 (see § 3.2).

More specifically, for Linux we require:

1. A **64-bit** distributive (Ubuntu **LTS** is recommended)
2. GNU C++ ($\geq$ 4.7), Intel Compiler ($\geq$ 14), Clang ($\geq$ 3.4)
3. CMake (GNU make is also required)
4. Boost (dev version $\geq$ 48, Ubuntu package `libboost1.48-all-dev` or newer `libboost1.54-all-dev`)
5. GNU scientific library (dev version, Ubuntu package `libgsl0-dev`)
6. Eigen (dev version, Ubuntu package `libeigen3-dev`)

For Windows, we require:

1. A **64-bit** distributive (we tested on Windows 8);
2. Visual Studio Express (or Professional) version 12 or later;
3. Boost is not required to build the core library and test utilities, but it is necessary to build some applications, including the main testing binary `experiment.exe` (see § 3.2).

Efficient implementations of many distance functions (see § 8) rely on SIMD instructions, which operate on small vectors of integer or floating point numbers. These instructions are available on most modern processors, but we support only SIMD instructions available on recent Intel and AMD processors. Each distance function has a pure C++ implementation, which can be less efficient than an optimized SIMD-based implementation.

On Linux, SIMD-based implementations are activated automatically for all sufficiently recent CPUs. On Windows, only SSE2 is enabled by default[8]. Yet, it is necessary to manually update project settings to enable more recent SIMD extensions (see §3.2).

Scripts to generate and process data sets are written in Python. We also provide a sample Python script to plot performance graphs: genplot_configurable.py (see § 3.7). In addition to Python, this plotting script requires Latex and PGF.

### 2.3 Installing C++11 Compilers

Installing C++11 compilers can be tricky, because they are not always provided as a standard package. This is why we briefly review the installation process here. In addition, installing compilers does not necessarily make them default compilers. One way to fix this is on Linux is to set environment variables `CXX` and `CC`. For the GNU 4.7 compiler:

```
export CXX=g++-4.7 CC=gcc-4.7
```

For the Clang compiler:

[8] Because SSE2 is available on all 64-bit computers.

```
export CXX=clang++-3.4 CC=clang-3.4
```

For the Intel compiler:

```
export CXX=icc CC=icc
```

It is, perhaps, the easiest to obtain Visual Studio by simply downloading it from the Microsoft web-site. We were able to build and run the 64-bit code using the free distributive of Visual Studio Express 14 (also called `Express 2015`) and Visual Studio 12 (Express 2013). The professional (and expensive) version of Visual Studio is not required.

To install GNU C++ version 4.7 on newer Linux distributions (in particular, on Ubuntu 14) with the Debian package management system, one can simply type:

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

At the time of this writing, GNU C++ 4.8 was available as a standard package as well.

However, this would not work on older distributives of Linux. One may need 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
```

If the script `add-apt-repository` is missing, it can be installed as follows:

```
sudo apt-get install python-software-properties
```

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

Similarly to the GNU C++ compiler, to install a C++11 version of Clang on newer Debian-based distributives one can simply type:

```
sudo apt-get install clang-3.6
```

However, for older distributives one may need to add a non-standard repository. For Debian and Ubuntu distributions, it is easiest to add repositories from the LLVM web-site. For example, if you have Ubuntu 12 (Precise), you need to add repositories as follows:[9]

```
sudo add-apt-repository \
  "deb http://llvm.org/apt/precise/ llvm-toolchain-precise main"
sudo add-apt-repository \
  "http://llvm.org/apt/precise/ llvm-toolchain-precise main"
sudo add-apt-repository \
  "deb http://llvm.org/apt/precise/ llvm-toolchain-precise-3.4 main"
sudo add-apt-repository \
```

[9] Do not forget to remove `deb-src` for source repositories. See the discussion here for more details.

```
  "http://llvm.org/apt/precise/ llvm-toolchain-precise-3.4 main"
sudo add-apt-repository \
  "deb http://ppa.launchpad.net/ubuntu-toolchain-r/test/ubuntu \
                                                  precise main"
```

Then, Clang 3.4 (and LLDB debugger) can be installed by typing:

```
sudo apt-get install clang-3.4 lldb-3.4
```

The Intel compiler can be freely used for non-commercial purposes by some categories of users (students, academics, and active open-source contributors). It is a part of C++ Composer XE for Linux and can be obtained from the Intel web site. After downloading and running an installation script, one needs to set environment variables. If the compiler is installed to the folder `/opt/intel`, environment variables are set by a script as follows:

```
/opt/intel/bin/compilervars.sh intel64
```

### 2.4 Quick Start on Linux

To build the project, go to the directory similarity_search and type:

```
cmake .
make
```

This creates several binaries in the directory `similarity_search/release`, most importantly, a benchmarking utility `experiment`, which carries out experiments, and testing utilities `bunit`, `test_integer`, and `bench_distfunc`. A more detailed description of the build process on Linux is given in § 3.1.

### 2.5 Query Server (Linux-only)

The query server requires Apache Thrift. We used Apache Thrift 0.9.2, but, perhaps, newer versions will work as well. To install Apache Thrift, you need to build it from source. This may require additional libraries. On Ubuntu they can be installed as follows:

```
sudo apt-get install libboost-dev libboost-test-dev \
     libboost-program-options-dev libboost-system-dev \
     libboost-filesystem-dev libevent-dev \
     automake libtool flex bison pkg-config \
     g++ libssl-dev libboost-thread-dev make
```

To simplify the building process of Apache Thrift, one can disable unnecessary languages:

```
./configure  --without-erlang --without-nodejs --without-lua  \
             --without-php --without-ruby --without-haskell \
             --without-go --without-d
```

After Apache Thrift is installed, you need to build the library itself. Then, change the directory to query_server/cpp_client_server and type `make` (the makefile may need to be modified, if Apache Thrift is installed to a non-standard location). The query server has a similar set of parameters to the benchmarking utility `experiment`. For example, you can start the server as follows:

```
./query_server -i ../../sample_data/final8_10K.txt -s l2 -p 10000 \
       -m sw-graph -c NN=10,efConstruction=200,initIndexAttempts=1
```

There are also three sample clients implemented in C++, Python, and Java. A client reads a string representation of the query object from the standard stream. The format is the same as the format of objects in a data file. Here is an example of searching for ten vectors closest to the first data set vector (stored in row one) of a provided sample data file:

```
export DATA_FILE=../../sample_data/final8_10K.txt
head -1 $DATA_FILE | ./query_client -p 10000 -a localhost  -k 10
```

It is also possible to generate client classes for other languages supported by Thrift from the interface definition file, e.g., for C#. To this end, one should invoke the thrift compiler as follows:

```
thrift --gen csharp  protocol.thrift
```

For instructions on using generated code, please consult the Apache Thrift tutorial.

### 2.6 Python bindings (Linux-only)

We provide basic Python bindings (for Linux and Python 2.7).

To build bindings for dense vector spaces (see § 4 for the description of spaces), build the library first. Then, change the directory to `python_vect_bindings` and type:

```
make
sudo make install
```

For an example of using our library in Python, see the script test_nmslib_vect.py.

Generic vector spaces are supported as well (see python_gen_bindings). However, they work only for spaces that properly define define serialization and de-serialization (see a brief description in § 7.2).

### 2.7 Quick Start on Windows

Building on Windows is straightforward. Download Visual Studio 2015 Express for Desktop. Download and install respective Boost binaries. Please, use the **default** installation directory on disk `c:` (otherwise, it will be necessary to update project files).

Afterwards, one can simply use the provided Visual Studio solution file. The solution file references several project (*.vcxproj) files: NonMetricSpaceLib.vcxproj is the main project file that is used to build the library itself. The output is stored in the folder `similarity_search\x64`. A more detailed description of the build process on Windows is given in § 3.2.

Note that the core library, the test utilities, as well as examples of the standalone applications (projects `sample_standalone_app1` and `sample_standalone_app2`) can be built without installing Boost.

## 3 Building and running the code (in detail)

A build process creates several important binaries, which include:

- NMSLIB library (on Linux `libNonMetricSpaceLib.a`), which can be used in external applications;
- The main benchmarking utility `experiment` (`experiment.exe` on Windows) that carries out experiments and saves evaluation results;
- A tuning utility `tune_vptree` (`tune_vptree.exe` on Windows) that finds optimal VP-tree parameters (see § 5.1.1 and our paper for details [8]);
- A semi unit test utility `bunit` (`bunit.exe` on Windows);
- A utility `bench_distfunc` that carries out integration tests (`bench_distfunc.exe` on Windows);

A build process is different under Linux and Windows. In the following sections, we consider these differences in more detail.

### 3.1 Building under Linux

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. Before creating the makefiles, we need to ensure that a right compiler is used. This is done by setting two environment variables: `CXX` and `CC`. In the case of GNU C++ (version 4.7), you may need to type:

```
export CCX=g++-4.7 CC=gcc-4.7
```

In the case of the Intel compiler, you may need to type:

```
export CXX=icc CC=icc
```

If you do not set variables `CXX` and `CC`, the default C++ compiler is used (which can be fine, if it is the right compiler already).

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
```

If `cmake` complains about the wrong version of the GCC, it is most likely that you forgot to set the environment variables `CXX` and `CC` (as described above). If this is the case, make these variables point to the correction version of the compiler. **Important note:** do not forget to delete the `cmake` cache and make file, before recreating the makefiles. For example, you can do the following (assuming the current directory is `similarity_search`):

```
rm -rf `find . -name CMakeFiles` CMakeCache.txt
```

Also note that, for some reason, `cmake` might sometimes ignore environmental variables `CXX` and `CC`. In this unlikely case, you can specify the compiler directly through `cmake` arguments. For example, in the case of the GNU C++ and the `Release` build, this can be done as follows:

```
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_COMPILER=g++-4.7 \
-DCMAKE_GCC_COMPILER=gcc-4.7 CMAKE_CC_COMPILER=gcc-4.7 .
```

The build process creates several binaries. Most importantly, the main benchmarking utility `experiment`. The directory `similarity_search/release` contains release versions of these binaries. Debug versions are placed into the folder `similarity_search/debug`.

**Important note:** a shortcut command:

```
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!

If the user cannot install necessary libraries to a standard location, it is still possible to build a project. First, download Boost to some local directory. Assume it is `$HOME/boost_download_dir`. Then, set the corresponding environment variable, which will inform `cmake` about the location of the Boost files:

```
export BOOST_ROOT=$HOME/boost_download_dir
```

Second, the user needs to install the additional libraries. Assume that the lib-files are installed to `$HOME/local_lib`, while corresponding include files are installed to `$HOME/local_include`. Then, the user needs to invoke `cmake` with the following arguments (after possibly deleting previously created cache and makefiles):

```
cmake . -DCMAKE_LIBRARY_PATH=$HOME/local_lib \
        -DCMAKE_INCLUDE_PATH=$HOME/local_include \
        -DBoost_NO_SYSTEM_PATHS=true
```

Note the last option. Sometimes, an old version of Boost is installed. Setting the variable `Boost_NO_SYSTEM_PATHS` to true, tells `cmake` to ignore such an installation.

To use the library in external applications, which do not belong to the library repository, one needs to install the library first. Assume that an installation location is the folder `NonMetrLibRelease` in the home directory. Then, the following commands do the trick:

```
cmake \
    -DCMAKE_INSTALL_PREFIX=$HOME/NonMetrLibRelease \
    -DCMAKE_BUILD_TYPE=Release .
make install
```

A directory sample_standalone_app contains two sample programs (see files sample_standalone_app1.cc and sample_standalone_app2.cc) that use NMSLIB binaries installed in the folder `$HOME/NonMetrLibRelease`.

**3.1.1 Developing and Debugging on Linux** There are several debuggers that can be employed. Among them, some of the most popular are: `gdb` (a command line tool) and a `ddd` (a GUI wrapper for gdb). For users who prefer IDEs, one good and free option is Eclipse IDE for C/C++ developers. It is not the same as Eclipse for Java and one needs to download this version of Eclipse separately.. An even better option is, perhaps, CLion. However, it is not free.[10]

After downloading and decompressing, e.g. as follows:

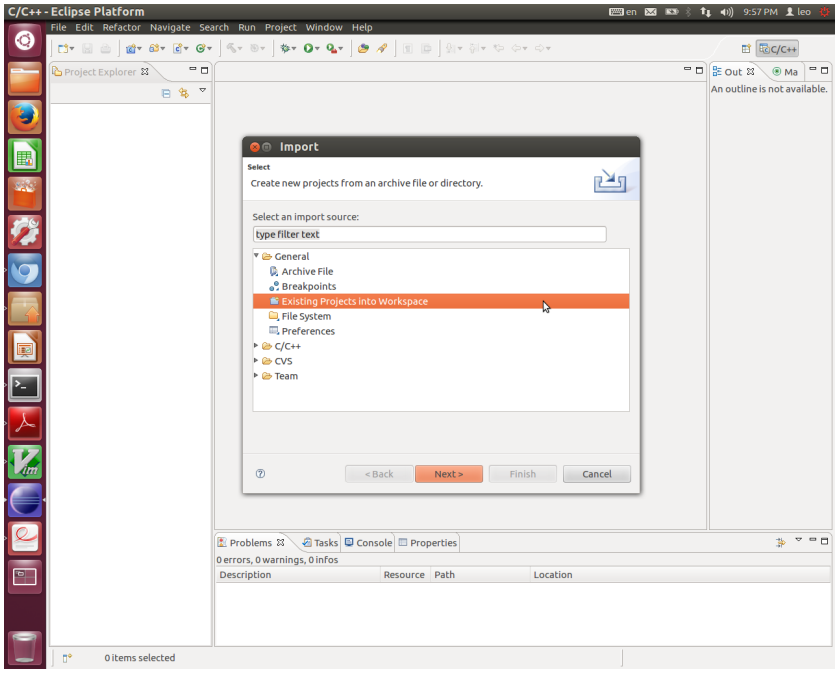
```
tar -zxvf eclipse-cpp-europa-winter-linux-gtk-x86_64.tar.gz
```

one can simply run the binary `eclipse` (in a newly created directory `eclipse`). On the first start, Eclipse will ask you select a repository location. This would be the place to store the project metadata and (optionally) actual project source files. The following description is given for Eclipse Europe. It may be a bit different with newer versions of Eclipse.

After selecting the workspace, the user can import the Eclipse project stored in the GitHub repository. Go to the menu `File`, sub-menu `Import`, category `General` and choose to import an existing project into the workspace as shown in Fig. 3. After that select a root directory. To this end, go to the directory where you checked out the contents of the GitHub repository and enter a sub-directory `similarity_search`. You should now be able to see the project `Non-Metric-Space-Library` as shown in Fig 4. You can now finalize the import by pressing the button `Finish`.

[10] There are a few categories of people, including students, who can ask for a free license, though.

Fig. 3: Selecting an existing project to import



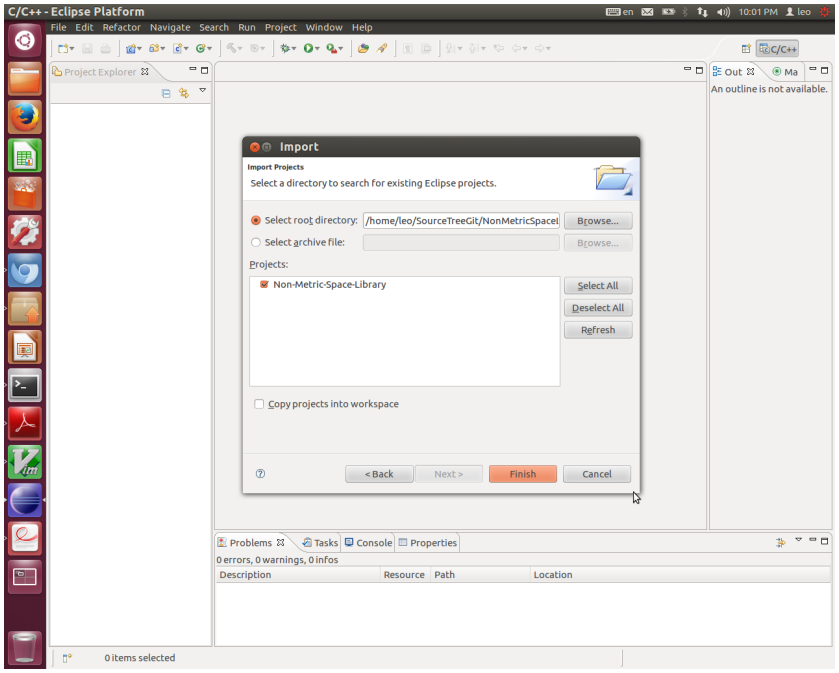

Next, we need to set some useful settings. Most importantly, we need to enable indexing of source files. This would allow us to browse class hierarchies, as well as find declarations of variables or classes. To this end, go to the menu `Window`, sub-menu `Preferences` and select a category `C++/indexing` (see Fig. 5). Then, check the box `Index all files`. Eclipse will start indexing your files with the progress being shown in the status bar (right down corner).

The user can also change the editor settings. We would strongly encourage to disable the use of tabs. Again, go the menu `Window`, sub-menu `Preferences`, and select a category `General/Editors/Text Editors`. Then, check the box `Insert spaces for tabs`. In the same menu, you can also change the fonts (use the category `General/Appearance/Colors and Fonts`).

In a newer Eclipse version, disabling tabs is done differently. To this end, go to the menu `Window`, sub-menu `Preferences`, and select a category `C++/Code Style/Formatter`. Then, you need to create a new profile and make this profile active. In the profile, change the tab policy to `Spaces only`.

It is possible to build the project from Eclipse (see the menu `Project`). However, one first needs to generate makefiles as described in § 3.1. The current limitation is that you can build either release or the debug version at a time. Moreover, to switch from one version to another, you need to recreate the makefiles from the command line.

Fig. 4: Importing an existing project




After building you can debug the project. To do this, you need to create a debug configuration. As an example, one configuration can be found in the project folder `launches`. Right click on the item `sample.launch`, choose the option `Debug as` (in the drop-down menu), and click on `sample` (in the pop-up menu). Do not forget to edit command line arguments before you actually debug the application!

After switching to a debug perspective, the Eclipse may stop the debugger in the file `dl-debug.c` as shown in Fig. 7. If this happened, simply, press the continue icon a couple of times until the debugger enters the code belonging to the library.

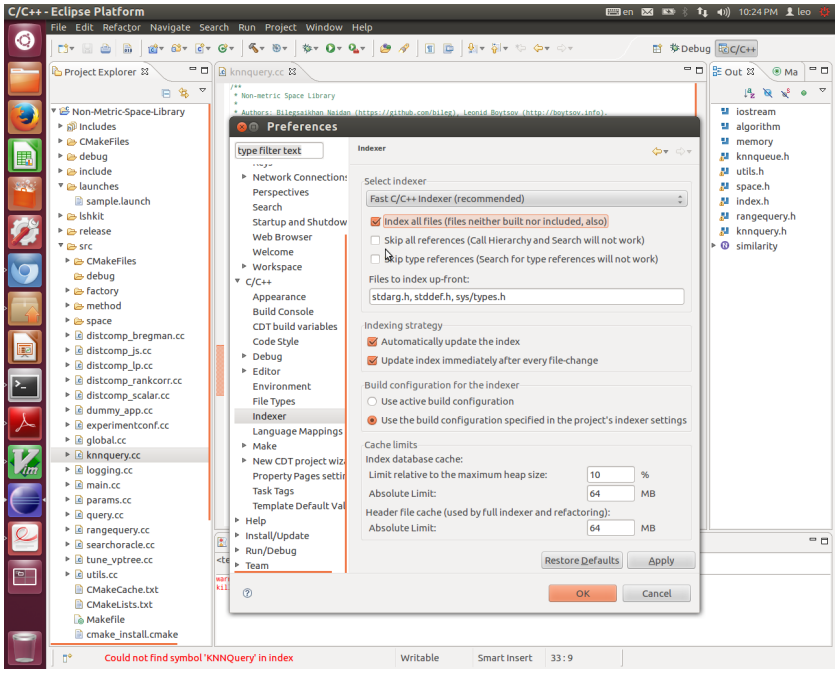
Additional configurations can be created by right clicking on the project name (left pane), selecting `Properties` in the pop-up menu and clicking on `Run/Debug settings`. The respective screenshot is shown in Fig. 6.

Note that this manual contains only a basic introduction to Eclipse. If the user is new to Eclipse, we recommend reading additional documentation available online.

### 3.2 Building under Windows

Download Visual Studio 2015 Express for Desktop. Download and install respective Boost binaries. Please, use the **default** installation directory on disk `c:`. In

Fig. 5: Enabling indexing of the source code




the end of the section, we explain how to select a different location of the Boost files, as well as to how downgrade the project to build it with Visual Studio 2013 (if this is really necessary).

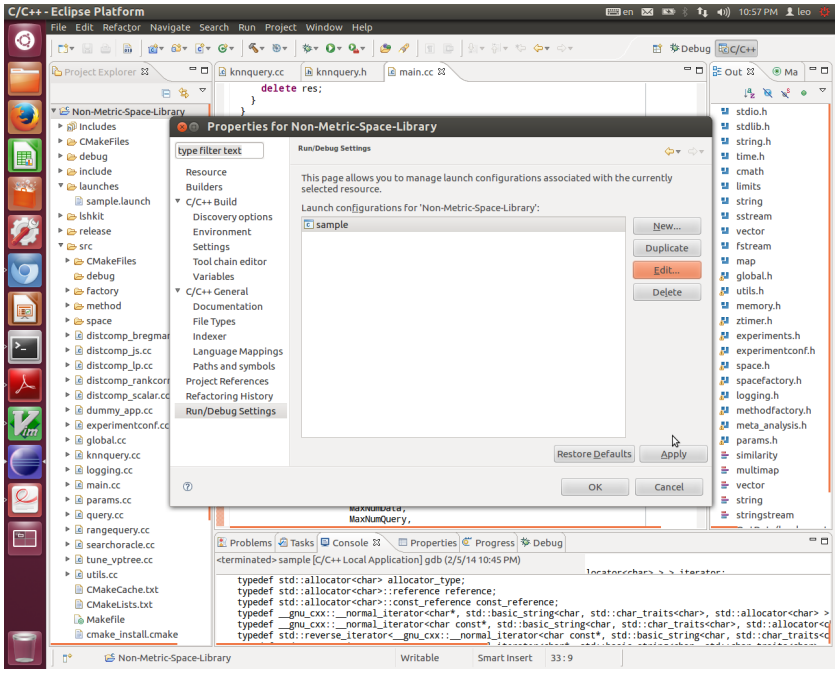
After downloading Visual Studio and installing Boost (version 59, 64-bit binaries, for MSVC-14), it is straightforward to build the project using the provided Visual Studio solution file. The solution file references several (sub)-project (`*.vcxproj`) files, which can be built either separately or all together.

The main sub-project is `NonMetricSpaceLib`, which is built before any other sub-projects. Sub-projects: `sample_standalone_app1`, `sample_standalone_app2` are examples of using the library in a standalone mode. Unlike building under Linux, we provide no installation procedure yet. In a nutshell, the installation consists in copying the library binary as well as the directory with header files.

There are three possible configurations for the binaries: `Release`, `Debug`, and `RelWithDebInfo` (release with debug information). The corresponding output files are placed into the subdirectories:

```
similarity_search\x64\Release,
similarity_search\x64\Debug,
similarity_search\x64\RelWithDebInfo.
```

Fig. 6: Creating a debug/run configuration




Unlike other compilers, there seems to be no way to detect the CPU type in the Visual Studio automatically.[11] And, by default, only SSE2 is enabled (because it is supported by all 64-bit CPUs). Therefore, if the user's CPU supports AVX extensions, it is recommended to modify code generation settings as shown in the screenshot in Fig. 8. This should be done for **all** sub-projects and **all** binary configurations. Note that you can set a property for all projects at once, if you select all the sub-projects, right-click, and then choose `Properties` in the pop-up menu.

The core library, the semi unit test binary as well as examples of the standalone applications can be built without installing Boost. However, Boost libraries are required for the binaries `experiment.exe`, `tune_vptree.exe`, and `test_integr.exe`.

We would re-iterate that one needs 64-bit Boost binaries compiled with the same version of the Visual Studio as the NMSLIB binaries. If you download the installer for Boost 59 and install it to a default location, then you do not have to change project files. Should you install Boost into a different folder, the location of Boost binaries and header file need to be specified in the project settings for all three build configurations (`Release`, `Debug`, `RelWithDebInfo`). An example of specifying the location of Boost libraries (binaries) is given in Fig. 9.

[11] It is not also possible to opt for using only SSE4.

Fig. 7: Starting a debugger

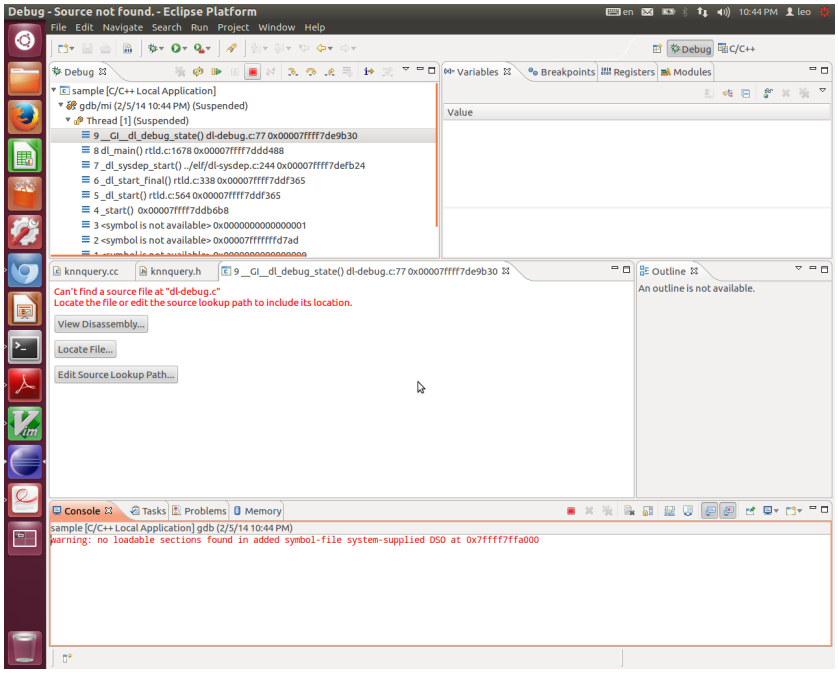



Fig. 8: Enabling advanced SIMD Instructions in the Visual Studio

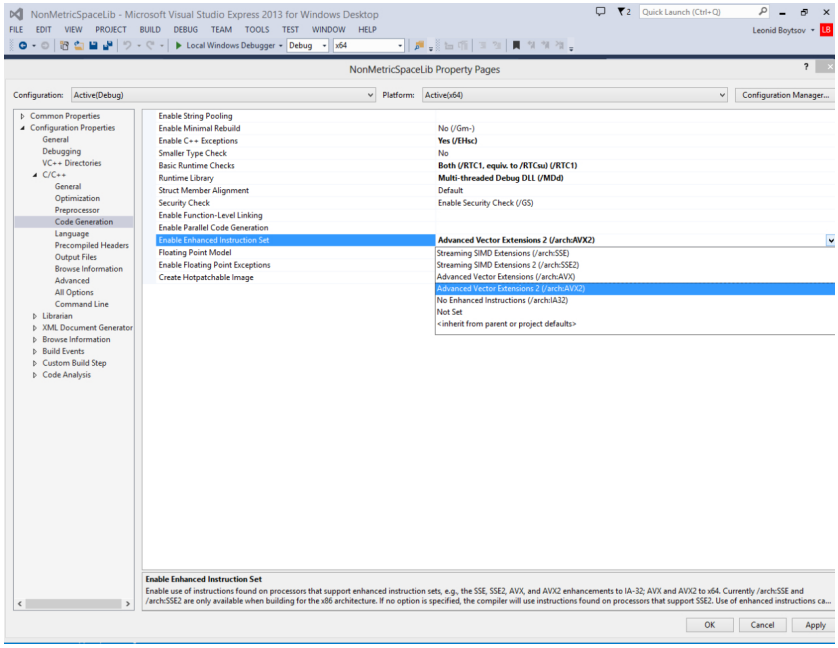
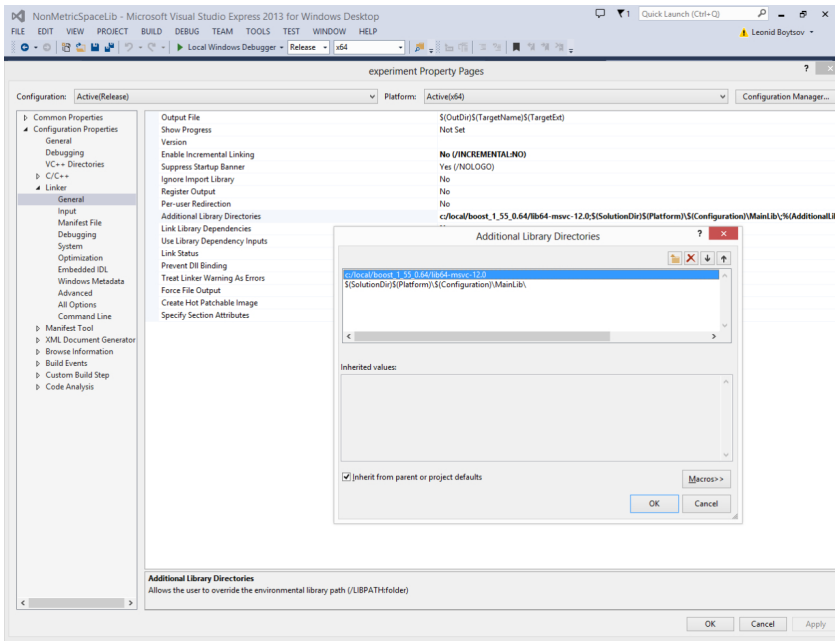


Fig. 9: Specifying Location of Boost libraries




In the unlikely case that the user has to use older Visual Studio 12, the project files are to be downgraded. To do so, one have to manually edit every `*.vcxproj` file by replacing each occurrence of `<PlatformToolset>v140</PlatformToolset>` with `<PlatformToolset>v120</PlatformToolset>`. Additionally, one has to download Boost binaries compatible with the older Visual studio and modify the project files accordingly. In particular, one may need to modify options `Additional Include Directories` and `Additional Library Directories`.

### 3.3 Testing the Correctness of Implementations

We have two main testing utilities `bunit` and `test_integr` (`experiment.exe` and `test_integr.exe` on Windows). Both utilities accept the single optional argument: the name of the log file. If the log file is not specified, a lot of informational messages are printed to the screen.

The `bunit` verifies some basic functitionality akin to unit testing. In particular, it checks that an optimized version of the, e.g., Eucledian, distance returns results that are very similar to the results returned by unoptimized and simpler version. The utility `bunit` is expected to always run without errors.

The utility `test_integr` runs complete implementations of many methods and checks if several effectiveness and efficiency characteristics meet the expectations. The expectations are encoded as an array of instances of the class `MethodTestCase` (see the code here). For example, we expect that the recall (see

§ 3.5.2) fall in a certain pre-recorded range. Because almost all our methods are randomized, there is a great deal of variance in the observed performance characteristics. Thus, some tests may fail infrequently, if e.g., the actual recall value is slightly lower or higher than an expected minimum or maximum. From an error message, it should be clear if the discrepancy is substantial, i.e., something went wrong, or not, i.e., we observe an unlikely outcome due to randomization. The exact search method, however, should always have an almost perfect recall.

Variance is partly due to using low-dimensional test sets. In the future, we plan to change this. For high-dimensional data sets, the outcomes are much more stable despite the randomized nature of most implementations.

### 3.4 Running Benchmarks

There are no principal differences in benchmarking on Linux and Windows. There is a *single* benchmarking utility `experiment` (`experiment.exe` on Windows) that includes implementation of all methods. It has multiple options, which specify, among others, a space, a data set, a type of search, a method to test, and method parameters. These options and their use cases are described in the following subsections. Note that unlike older versions it is not possible to test multiple methods in a single run. However, it is possible to test the single method with different values of query-time parameters.

**3.4.1 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 (parameters may not contain whitespaces). In this case, there is colon after the space mnemonic name 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 3 in § 4.

For real-valued distance functions, one can use either single- or double-precision type. Single-precision is a recommended default. One example of integer-valued distance function the Levenshtein distance.

**3.4.2 Input Data/Test Set** There are two options that define the data to be indexed:

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

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.

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

```
-q [ --queryFile ] arg         query file
-Q [ --maxNumQuery ] arg (=0) if non-zero, use maxNumQuery query
                               elements(required in the case
                               of bootstrapping)
```

If a separate query 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
```

**3.4.3 Query Type** Our framework supports the $k$-NN and the range search. The user can request to run both types of queries:

```
-k [ --knn ] arg          comma-separated values of k
                          for the k-NN search
-r [ --range ] arg        comma-separated radii 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.

**3.4.4 Method Specification** Unlike older versions it is possible to test only a single method at a time. To specify a method's name, use the following option:

```
-m [ --method ] arg            method/index name
```

A method can have a single set of index-time parameters, which is specified via:

```
-c [ --createIndex ] arg         index-time method(s) parameters
```

In addition to the set of index-time parameters, the method can have multiple sets of query-time parameters, which are specified using the following (possibly repeating) option:

```
-t [ --queryTimeParams ] arg    query-time method(s) parameters
```

For each set of query-time parameters, i.e., for each occurrence of the option `--queryTimeParams`, the benchmarking utility `experiment`, carries out an evaluation using the specified set of queries and a query type (e.g., a 10-NN search with queries from a specified file). If the user does not specify any query-time parameters, there is only one evaluation to be carried out. This evaluation uses default query-time parameters. In general, we ensure that *whenever a query-time parameter is missed, the default value is used.*

Similar to parameters of the spaces, a set of method's parameters is a comma-separated list (no-spaces) of parameter-value pairs in the format: `<parameter name>=<parameter value>`. For a detailed list of methods and their parameters, please, refer to § 5.

Note that a few methods can save/restore (meta) indices. To save and load indices one should use the following options:

```
-L [ --loadIndex ] arg        a location to load the index from
-S [ --saveIndex ] arg        a location to save the index to
```

When the user defines the location of the index using the option `--loadIndex`, the index-time parameters may be ignored. Specifically, if the specified index does not exist, the index is created from scratch. Otherwise, the index is loaded from disk. Also note that the benchmarking utility *does not override an already existing index* (when the option `--saveIndex` is present).

If the tests are run the bootstrapping mode, i.e., when queries are randomly sampled (without replacement) from the data set, several indices may need to be created. Specifically, for each split we create a separate index file. The identifier of the split is indicated using a special suffix. Also note that we need to memorize which data points in the split were used as queries. This information is saved in a gold standard cache file (see § 8). Thus, saving and loading of indices in the bootstrapping mode is possible only if gold standard caching is used.

**3.4.5 Saving and Processing Benchmark Results** The benchmarking utility may produce output of three types:

- Benchmarking results (a human readable report and a tab-separated data file);
- Log output (which can be redirected to a file);
- Progress bars to indicate the progress in index creation for some methods (cannot be currently suppressed);

To save benchmarking results to a file, on needs to specify a parameter:

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

As noted above, 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: If a previously created report exists, it is erased. The following option can be used to append results to the previously created report:

```
-a [ --appendToResFile ]    do not override information in results
```

For information on processing and interpreting results see § 3.5. A description of the plotting utility is given in § 3.7.

By default, all log messages are printed to the standard error stream. However, they can also be redirected to a log-file:

```
-l [ --logFile ] arg          log file
```

**3.4.6 Efficiency of Testing** Except for measuring methods' performance, the following are the most expensive operations:

- computing ground truth answers (also known as *gold standard* data);
- loading the data set;
- indexing.

To make testing faster, the following methods can be used:

- Caching of gold standard data;
- Creating gold standard data using multiple threads;
- Reusing previously created indices (when loading and saving is supported by a method);
- Carrying out multiple tests using different sets of query-time parameters.

By default, we recompute gold standard data every time we run benchmarks, which may take long time. However, it is possible to save gold standard data and re-use it later by specifying an additional argument:

```
-g [ --cachePrefixGS ] arg  a prefix of gold standard cache files
```

The benchmarks can be run in a multi-threaded mode by specifying a parameter:

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

In this case, the gold standard data is also created in a multi-threaded mode (which can also be much faster). Note that NMSLIB directly supports only an inter-query parallelism, i.e., multiple queries are executed in parallel, rather than the intra-query parallelism, where a single query can be processed by multiple CPU cores.

Gold standard data is stored in two files. One is a textual meta file that memorizes important input parameters such as the name of the data and/or query file, the number of test queries, etc. For each query, the binary cache files contains ids of answers (as well as distances and class labels). When queries are created by random sampling from the main data set, we memorize which objects belong to each query set.

When the gold standard data is reused later, the benchmarking code verifies if input parameters match the content of the cache file. Thus, we can prevent an accidental use of gold standard data created for one data set while testing with a different data set.

Another sanity check involves verifying that data points obtained via an approximate search are not closer to the query than data points obtained by an exact search. This check has turned out to be quite useful. It has helped detecting a problem in at least the following two cases:

- The user creates a gold standard cache. Then, the user modifies a distance function and runs the tests again;
- Due to a bug, the search method reports distances to the query that are smaller than the actual distances (computed using a distance function). This may occur, e.g., due to a memory corruption.

When the benchmarking utility detects a situation when an approximate method returns points closer than points returned by an exact method, the testing procedure is terminated and the user sees a diagnostic message (see Table 1 for an example).

```
... [INFO] >>>> Computing effectiveness metrics for sw-graph
... [INFO] Ex: -2.097 id = 140154 -> Apr: -2.111 id = 140154 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.974 id = 113850 -> Apr: -2.005 id = 113850 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.883 id = 102001 -> Apr: -1.898 id = 102001 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.6667 id = 58445 -> Apr: -1.6782 id = 58445 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.6547 id = 76888 -> Apr: -1.6688 id = 76888 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.5805 id = 47669 -> Apr: -1.5947 id = 47669 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.5201 id = 65783 -> Apr: -1.4998 id = 14954 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.4688 id = 14954 -> Apr: -1.3946 id = 25564 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.454 id = 90204 -> Apr: -1.3785 id = 120613 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.3804 id = 25564 -> Apr: -1.3190 id = 22051 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.367 id = 120613 -> Apr: -1.205 id = 101722 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.318 id = 71704 -> Apr: -1.1661 id = 136738 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.3103 id = 22051 -> Apr: -1.1039 id = 52950 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.191 id = 101722 -> Apr: -1.0926 id = 16190 1 - ratio: -0.0202 diff: -0.0221
... [INFO] Ex: -1.157 id = 136738 -> Apr: -1.0348 id = 13878 1 - ratio: -0.0202 diff: -0.0221
... [FATAL] bug: the approximate query should not return objects that are closer to the query
                than object returned by (exact) sequential searching!
                Approx: -1.09269 id = 16190 Exact: -1.09247 id = 52950
```

Table 1: A diagnostic message indicating that there is some mismatch in the current experimental setup and the setup used to create the gold standard cache file.

To compute recall, it is enough to memorize only query answers. For example, in the case of a 10-NN search, it is enough to memorize only 10 data points closest to the query as well as respective distances (to the query). However, this is not sufficient for computation of a rank approximation metric (see § 3.5.2). To control the accuracy of computation, we permit the user to change the number of entries memorized. This number is defined by the parameter:

```
--maxCacheGSRelativeQty arg (=10)    a maximum number of gold
                                     standard entries
```

Note that this parameter is a coefficient: the actual number of entries is defined relative to the result set size. For example, if a range search returns 30 entries

and the value of `--maxCacheGSRelativeQty` is 10, then $30 \times 10 = 300$ entries are saved in the gold standard cache file.

### 3.5 Measuring Performance and Interpreting Results

**3.5.1 Efficiency.** We measure several efficiency metrics: query runtime, the number of distance computations, the amount of memory used by the index *and* the data, and the time to create the index. We also measure the improvement in runtime (improvement in efficiency) with respect to a sequential search (i.e., brute-force) approach as well as an improvement in the number of distance computations. If the user runs benchmarks in a multi-threaded mode (by specifying the option `--threadTestQty`), we compare against the multi-threaded version of the brute-force search as well.

A good method should carry out fewer distance computations and be faster than the brute-force search, which compares *all the objects* directly with the query. However, great reduction in the number of distance computations does not always entail good improvement in efficiency: while we do not spend CPU time on computing directly, we may be spending CPU time on, e.g., computing the value of the distance in the projected space (as in the case of projection-based methods, see § 5.3).

Note that the improvement in efficiency is adjusted for the number of threads. Therefore, it does not increase as more threads are added: in contrast, it typically decreases as there is more competition for computing resources (e.g., memory) in a multi-threaded mode.

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. On Linux, we query a special file `/dev/<process id>/status`, which might not work for all Linux distributives. Under Windows, we retrieve the working set size using the function `GetProcessMemoryInfo`. Note that we do not have a truly portable code to measure memory consumption of a process.

**3.5.2 Effectiveness** In the following description, we assume that a method returns a set of points/objects $\{o_i\}$. The value of $\text{pos}(o_i)$ represents a positional distance from $o_i$ to the query, i.e., the number of database objects closer to the query than $o_i$ plus one. Among objects with identical distances to the query, the object with the smallest index is considered to be the closest. Note that $\text{pos}(o_i) \geq i$.

Several effectiveness metrics are computed by the benchmarking utility:

- A *number of points closer* to the query than the nearest returned point. This metric is equal $\text{pos}(o_1)$ minus one. If $o_1$ is always the true nearest object, its positional distance is one and, thus, the *number of points closer* is always equal to zero.
- A *relative position* error for point $o_i$ is equal to $\text{pos}(o_i)/i$, an aggregate value is obtained by computing the geometric mean over all returned $o_i$;

– *Recall*, which is is equal to the fraction of all correct answers retrieved.
– *Classification accuracy*, which is equal to the fraction of labels correctly predicted by a $k$-NN based classification procedure.

The first two metrics represent a so-called rank (approximation) error. The closer the returned objects are to the query object, the better is the quality of the search response and the lower is the rank approximation error.

Recall is a classic metric. It was argued, however, that recall does not account for positional information of returned objects and is, therefore, inferior to rank approximation error metrics [1,11]. Consider the case of 10-NN search and imagine that there are two methods that, on average, find only half of true 10-NN objects. Also, assume that the first method always finds neighbors from one to five, but misses neighbors from six to ten. The second method always finds neighbors from six to ten, but misses the first five ones. Clearly, the second method produces substantially inferior results, but it has the same recall as the first one.

If we specify ground-truth object classes (see § 9 for the description of data set formats), it is possible to compute an accuracy of a $k$-NN based classification procedure. The label of an element is selected as the most frequent class label among $k$ closest objects returned by the method (in the case of ties the class label with the smallest id is chosen).

Table 2: An example of a human-readable report

```
====================================
vptree: triangle inequality
alphaLeft=2.0,alphaRight=2.0
====================================
# of points: 9900
# of queries: 100
------------------------------------
Recall:          0.954 -> [0.95 0.96]
ClassAccuracy:   0     -> [0 0]
RelPosError:     1.05  -> [1.05 1.06]
NumCloser:       0.11  -> [0.09 0.12]
------------------------------------
QueryTime:       0.2   -> [0.19 0.21]
DistComp:        2991  -> [2827 3155]
------------------------------------
ImprEfficiency: 2.37  -> [2.32 2.42]
ImprDistComp:   3.32  -> [3.32 3.39]
------------------------------------
Memory Usage:   5.8 MB
------------------------------------
```

**Note:** *confidence intervals* are in brackets

If we had ground-truth queries and relevance judgements from human assessors, we could in principle compute other realistic effectiveness metrics such as the mean average precision, or the normalized discounted cumulative gain. This remains for the future work.

Note that it is pointless to compute the mean average precision when human judgments are not available, as the mean average precision is identical to the recall in this case.

### 3.6 Interpreting and Processing Benchmark Results

If the user specifies the option `--outFilePrefix`, the benchmarking results are stored to the file system. A prefix of result files is defined by the parameter `--outFilePrefix` while the suffix is defined by a type of the search procedure (the $k$-NN or the range search) as well as by search parameters (e.g., the range search radius). For each type of search, two files are generated: a report 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 as described in § 3.7.

An example of human readable report (*confidence intervals* are in square brackets) is given in Table 2. In addition to averages, the human-readable report provides 95% confidence intervals. In the case of bootstrapping, statistics collected for several splits of the data set are aggregated. For the retrieval time and the number of distance computations, this is done via a classic fixed-effect model adopted in meta analysis [27]. When dealing with other performance metrics, we employ a simplistic approach of averaging split-specific values and computing the sample variance over spit-specific averages.[12] Note for all metrics, except relative position error, an average is computed using an arithmetic mean. For the relative error, however, we use the geometric mean [29].

### 3.7 Plotting results (Linux-Only)

We provide the Python script to generate nice performance graphs from tab-separated data file produced by the benchmarking utility `experiment`. The plotting script is genplot_configurable.py. In addition to Python, it requires Latex and PGF. This script is supposed to run only on Linux.

Consider the following example of using `genplot_configurable.py`:

```
../scripts/genplot_configurable.py  \
          -n MethodName \
          -i result_K\=1.dat -o plot_1nn \
          -x 1~norm~Recall \
          -y 1~log~ImprEfficiency \
          -a  axis_desc.txt \
          -m  meth_desc.txt \
          -l "2~(0.96,-.2)" \
          -t "ImprEfficiency  vs Recall" \
          --xmin 0.01 --xmax 1.2 --ymin -2 --ymax 10
```

Here the goal is to process the tab-separated data file `result_K=1.dat`, which was generated by 1-NN search, and save the plot to an output file `plot_1nn.pdf`. Note that one should not explicitly specify the extension of the output file (as

[12] The distribution of many metric values is not normal. There are approaches to resolve this issue (e.g., apply a transformation), but an additional investigation is needed to understand which approaches work best.

`.pdf` is always implied). Also note that, in addition to the PDF-file, the script generates the source Latex file. The source Latex file can be post-edited and/or embedded directly into a Latex source (see PGF documentation for details). This can be useful for scientific publishing.

The parameter `-n` specifies the name of the field that stores method/indices mnemonic names. In the case of the benchmarking utility `experiment` this field is named `MethodName`. Parameters `-x` and `-y` define X and Y axis, i.e., which metrics are associated with each axis and what is the display format. Arguments `-x` and `-y` have the same format. Specifically, these option arguments include three tilda-separated values each. The first value should be zero or one. Specify zero not to print the axis label. The second value is either `norm` or `log`, which stands for a normal or logarithmic scale, respectively.

The last value defines a metric that we want to visualize: The metric should be a field name, i.e., it is one of the names that appear in the header row of the output data file. However, a display value of the metric can be different. To specify metric display values, one has to provide an axis description file (option `-a`).

The axis description file has two tab-separated columns. The first column is a name of the metric used in the data file (in our example it is `result_K=1.dat`). The second column is a display name. Here is an example of the axis description file (tabs are not shown):

```
Recall        recall
QueryTime     time (ms)
```

Similarly, the user has to specify a method description file (option `-m`). It is a three-column tab-separated file. The first column is the name of the method used in the data file `result_K=1.data` (*exactly* as it is specified there). The second column is the method display name (it can have Latex-parsable expressions). The third column, defines the style of the plot (e.g., line thickness and a line mark). This should be a comma-separated lists of PGF-specifiers (see PGF documentation for details). Here is an example of the method description file (tabs again are not shown):

```
"bbtree" bbtree mark=triangle*
"vptree" vptree mark=star,blue,/tikz/densely dashed,mark size=1.5pt
```

The parameter `-l` defines a plot legend. To hide the legend, use the string `none`. Otheriwse, two tilda-separated values should be specified. The first value gives the number of columns in the legend, while the second value defines a position of the legend. The position can be either absolute or relative. An absolute position is defined by a pair of coordinates (in round brackets). A relative position is defined by one of the following descriptors (quotes are for clarity only): “north west”, “north east”, “south west”, “south east”. If the relative position is specified, the legend is printed inside the main plotting area, e.g.:

```
../scripts/genplot_configurable.py  \
          -n MethodName \
          -i result_K\=1.dat -o plot_1nn \
          -x 1~norm~Recall \
          -y 1~log~ImprEfficiency \
          -a  axis_desc.txt \
          -m  meth_desc.txt \
          -l "2~north west" \
          -t "ImprEfficiency  vs Recall"
```

The title of the plot is defind by `-t` (specify `-t ""` if you do not want to print the title). Finally, note that the user can specify the bounding rectangle for the plot via the options `--xmin`, `--xmax`, `--ymin`, and `--ymax`.

## 4 Spaces

Currently we provide implementations mostly for vector spaces. Vector-space input files can come in either regular, i.e., dense, or sparse variant (see § 9). A detailed list of spaces, their parameters, and performance characteristics is given in Table 3.

The mnemonic name of the space is passed to the benchmarking utility (see § 3.4). 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 § 4.2-4.5.

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 [11]. If you know a better approach, feel free, to tell us.

### 4.1 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.[13] The length of each string was sampled from a normal distribution $\mathcal{N}(32, 4)$.

The Signature Quadratic Form Distance (SQFD) [5,4] was tested using signatures extracted from LSVRC-2014 data set [44], which contains 1.2 million high resolution images. We implemented our own code to extract signatures following the method of Beecks [4]. 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 $10^4$).

### 4.2 $L_p$-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:

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

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

Table 3: Description of implemented spaces

| **Space** | **Mnemonic Name & Formula** | **Efficiency** (million op/sec) |
|---|---|---|
| | **Metric Spaces** | |
| Hamming | `bit_hamming`<br>$\sum_{i=1}^{n} \lvert x_i - y_i \rvert$ | 240 |
| $L_1$ | `l1`, `l1_sparse`<br>$\sum_{i=1}^{n} \lvert x_i - y_i \rvert$ | 35, 1.6 |
| $L_2$ | `l2`, `l2_sparse`<br>$\sqrt{\sum_{i=1}^{n} \lvert x_i - y_i \rvert^2}$ | 30, 1.6 |
| $L_\infty$ | `linf`, `linf_sparse`<br>$\max_{i=1}^{n} \lvert x_i - y_i \rvert$ | 34 , 1.6 |
| $L_p$ (generic $p \geq 1$) | `lp:p=...`, `lp_sparse:p=...`<br>$\left(\sum_{i=1}^{n} \lvert x_i - y_i \rvert^p\right)^{1/p}$ | 0.1-3, 0.1-1.2 |
| Angular distance | `angulardist`, `angulardist_sparse`, `angulardist_sparse_fast`<br>$\arccos\left(\frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}\right)$ | 13, 1.4, 3.5 |
| Jensen-Shan. metr. | `jsmetrslow, jsmetrfast, jsmetrfastapprox`<br>$\sqrt{\frac{1}{2}\sum_{i=1}^{n}\left[x_i \log x_i + y_i \log y_i - (x_i + y_i)\log\frac{x_i+y_i}{2}\right]}$ | 0.3, 1.9, 4.8 |
| Levenshtein | `leven` (see § 4.6 for details) | 0.2 |
| SQFD | `sqfd_minus_func`, `sqfd_heuristic_func:alpha=...`,<br>`sqfd_gaussian_func:alpha=...` (see § 4.7 for details) | 0.05, 0.05, 0.03 |
| | **Non-metric spaces (symmetric distance)** | |
| $L_p$ (generic $p < 1$) | `lp:p=..., lp_sparse:p=...`<br>$\left(\sum_{i=1}^{n} \lvert x_i - y_i \rvert^p\right)^{1/p}$ | 0.1-3, 0.1-1 |
| Jensen-Shan. div. | `jsdivslow, jsdivfast, jsdivfastapprox`<br>$\frac{1}{2}\sum_{i=1}^{n}\left[x_i \log x_i + y_i \log y_i - (x_i + y_i)\log\frac{x_i+y_i}{2}\right]$ | 0.3, 1.9, 4.8 |
| Cosine distance | `cosinesimil`, `cosinesimil_sparse`, `cosinesimil_sparse_fast`<br>$1 - \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$ | 13, 1.4, 3.5 |
| Norm. Levenshtein | `normleven`, see § 4.6 for details | 0.2 |
| | **Non-metric spaces (non-symmetric distance)** | |
| Regular KL-div. | left queries: `kldivfast`<br>right queries: `kldivfastrq`<br>$\sum_{i=1}^{n} x_i \log\frac{x_i}{y_i}$ | 0.5, 27 |
| Generalized KL-div. | left queries: `kldivgenslow`, `kldivgenfast`<br>right queries: `kldivgenfastrq`<br>$\sum_{i=1}^{n}\left[x_i \log\frac{x_i}{y_i} - x_i + y_i\right]$ | 0.5, 27<br>27 |
| Itakura-Saito | left queries: `itakurasaitoslow, itakurasaitofast`<br>right queries: `itakurasaitofastrq`<br>$\sum_{i=1}^{n}\left[\frac{x_i}{y_i} - \log\frac{x_i}{y_i} - 1\right]$ | 0.2, 3, 14<br>14 |

$L_\infty$ 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 $L_\infty$ (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 3). 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 $L_\infty$ 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`.

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).

### 4.3 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. [47]) is about 2.5-3 times faster than a pure C++ implementation based on the merge-sort approach.

### 4.4 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 [20], 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.

### 4.5 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 [10,11]. 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 3).

### 4.6 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 [31]. 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 § 4.1, 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 [54,53,45] and computational biology [40] (see Sankoff [46] for a historical perspective). Despite the early discovery, the algorithm was generally unknown before a publication by Wagner and Fischer [55] in a computer science journal.

### 4.7 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 $\alpha$. For further details, please, see the thesis of Beecks [4].

# 5 Search Methods

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

- Space partitioning methods (including a specialized method bbtree for Bregman divergences) § 5.1;
- Locality Sensitive Hashing (LSH) methods § 5.2;
- Filter-and-refine methods based on projection to a lower-dimensional space § 5.3;
- Filtering methods based on permutations § 5.4;
- Methods that construct a proximity graph § 5.5;
- Miscellaneous methods § 5.6.

In the following subsections (§ 5.1-5.6), we describe implemented methods, explain their parameters, and provide examples of their use via the benchmarking utility `experiment` (`experiment.exe` on Windows). Note that a few parameters are query-time parameters, which means that they can be changed without rebuilding the index see § 3.4.6. For the description of the utility `experiment` see § 3.4. For several methods we provide *basic tuning guidelines*, see § 6.

## 5.1 Space Partitioning Methods

Parameters of space partitioning methods are summarized in Table 4. 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 [15].

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. [14]

[14] 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 $L_\infty$ distance. That is, the $L_\infty$ 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 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 $\pi_i$ 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. [15]

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.

**5.1.1 VP-tree** A VP-tree [52,57] (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.

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 $\pi$ is the pivot in the VP-tree, $q$ is the query with the radius $r$, and $R$ is the median distance from $\pi$ 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

[15] 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.

$r > |R - d(\pi, q)|$, 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 $\alpha$: 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 $\alpha$ possibly smaller than one [8,38]. 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 $\alpha_{left}$ and $\alpha_{right}$ 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 § 5.3.2).
- 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
```

**5.1.2 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 $L_\infty$ 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.

**5.1.3 GH-Tree** A GH-tree [52] 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.

Table 4: 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`) [52,57] | |
| | Common parameters: `bucketSize`, `chunkBucket`, and `maxLeavesToVisit` |
| `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`) [9] | |
| | 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`) [52] | |
| | Common parameters: `bucketSize`, `chunkBucket`, and `maxLeavesToVisit` |
| **List of clusters** (`list_clusters`) [14] | |
| | 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`) [39] | |
| | No parameters |
| **bbtree** (`bbtree`) [11] | |
| | Common parameters: `bucketSize`, `chunkBucket`, and `maxLeavesToVisit` |

**Note:** mnemonic method names are given in round brackets.

**5.1.4 List of Clusters** The list of clusters [14] 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
```

**5.1.5 SA-tree** The Spatial Approximation tree (SA-tree) [39] 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
```

**5.1.6 bbtree** A Bregman ball tree (bbtree) is an exact search method for Bregman divergences [11]. 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 [11] 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 [12].

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
```

### 5.2 Locality-sensitive Hashing Methods

Locality Sensitive Hashing (LSH) [28,30] 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 5. 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 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 [16], 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 [16]. 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 [56,33].

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

Table 5: Parameters of LSH methods

| | |
|---|---|
| **Common parameters** | |
| `W` | A width of the window [17]. |
| `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`)** [34,19,17] | |
| | 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`)** [13] | |
| | Common parameters: `W`, `M`, `H`, and `L` |
| **LSH Cauchy: only for $L_1$ (`lsh_cauchy`)** [13] | |
| | Common parameters: `W`, `M`, `H`, and `L` |
| **LSH thresholding: only for $L_1$ (`lsh_threshold`)** [56,33] | |
| | Common parameters: `M`, `H`, and `L` (`W` is not used) |

**Note:** mnemonic method names are given in round brackets.

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
```

### 5.3 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.

**5.3.1 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 ([26]). 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
```

**5.3.2 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 § 5.1.1) and the projection parameters as in the case of brute-force searching of projections (see § 5.3.1).

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 $\alpha_{left}$ and

$\alpha_{right}$ 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 § 9.

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
```

**5.3.3 OMEDRANK.** In OMEDRANK [22] 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. [22] 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` × `minFreq`, where `minFreq` is a method's parameter, e.g., 0.5.

The first point whose counter becomes equal to or larger than `numPivot` × `minFreq`, becomes the first candidate entry to be compared directly against the query. The next point whose counter matches the threshold value `numPivot` × `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` × `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
```

### 5.4 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 $\pi$ 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 [8,38].

Table 6: 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** [22] (`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.

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 7-8.

**5.4.1 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 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 5.3.1), 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 5.4.6.

**5.4.2 Permutation Prefix Index (PP-Index).** In a permutation prefix index (PP-index), permutation are stored in a prefix tree of limited depth [21]. 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
```

**5.4.3 VP-tree index over permutations.** We can use a VP-tree to index permutations. This approach is similar to that of Figueroa and Fredriksson [24].

We, however, rely on the approximate version of the VP-tree described in § 5.1.1, 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 § 5.3.2). There is also an additional VP-tree based method that indexes binarized permutations. It is described in § 5.4.6.

**5.4.4 Metric Inverted File** (MI-File) relies on the inverted index over permutations [2]. 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 [2] 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
```

**5.4.5 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 [51].

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 [51] 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 [32].

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
```

**5.4.6 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 3). 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
```

## 5.5 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

Table 7: 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`) [21] | |
| `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`) [2] | |
| | 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`) [51] | |
| | 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 8: Parameters of permutation-based filtering methods (continued)

| | |
|---|---|
| **Brute-force search with incremental sorting for binarized permutations** (`perm_incsort_bin`) [50] | |
| Common parameters: `numPivot`, `dbScanFrac`, `binThreshold`. | |
| **VP-tree index over binarized permutations** (`perm_bin_vptree`) Similar to [50], 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 $\alpha_{left}$ in Eq. (6) |
| `alphaRight` | A stretching coefficient $\alpha_{right}$ in Eq. (6) |

**Note:** mnemonic method names are given in round brackets.

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 carries out a $k'$-NN search for some $k' \geq k$ and terminates when the result set stops changing (i.e., we cannot find a point that is closer to a query 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.

In our library we use several approaches to create proximity graphs, which are described below. Parameters of these methods are summarized in Table 9. 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.

**5.5.1 Small World Graph (SW-graph).** In the (Navigable) Small World graph (SW-graph),[16] 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

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

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 until it cannot find a point that is closer than already found `efConstruction` nearest points (`efConstruction` is an index-time parameter). Similarly, the search procedure executes one or more sub-searches that start from some node and proceed until it cannot find a point closer than already found `efSearch` closest to the query. 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) [35]. A simpler and less efficient variant of this algorithm was presented at ICTA 2011 and SISAP 2012 [42,35]. An improved variant appeared as an Information Systems publication [36]. 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 [37].

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`).

**5.5.2 Hierarchical Navigable Small World Graph (HNSW).** The Hierarchical Navigable Small World Graph (HNSW) [37] 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 [37], 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`.

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 [37] 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
```

**5.5.3 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 [35,36]. 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
```

### 5.6 Miscellaneous Methods

Parameters of miscellaneous methods are summarized in Table 10.

**5.6.1 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:

```
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
```

**5.6.2 Several copies of the same index type.** It is possible to generate several copies of the same index using a meta method `mult_indx`. This makes sense for randomized indexing methods, e.g., for the VP-tree or the PP-index.[17]

```
release/experiment \
  --distType float --spaceType l2 --testSetQty 5 --maxNumQuery 100 \
  --knn 1  \
  --dataFile ../sample_data/final8_10K.txt --outFilePrefix result \
  --method mult_index \
    --createIndex methodName=vptree,indexQty=5  \
    --queryTimeParams maxLeavesToVisit=2
```

## 6 Tuning Guidelines

In the following subsections, we provide brief tuning guidelines. These guidelines are broad-brush: The user is expected to find optimal parameters through experimentation on some development data set.

### 6.1 NAPP

Generally, increasing the overall number of pivots `numPivot` helps to improve performance. However, using a large number of pivots leads to increased indexing times.

[17] In fact, all of the methods except for the sequential, i.e., brute-force, search are randomized.

Table 9: 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`) [42,35,36] | |
| `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`) [36] | |
| `mult` | A scaling coefficient to determine the depth of a layered structure (see the paper by Malkov and Yashunin [36] 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 [36] |
| **NN-descent** (`nndes`) [18,35,36] | |
| `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.

Table 10: Parameters of miscellaneous methods

| **Several copies of the same index type** (`mult_index`) | |
|---|---|
| `indexQty` | A number of copies |
| `methodName` | A mnemonic method name |
| | Any other parameter that the method accepts. For instance, if we create several copies of the VP-tree, we can specify the parameters `alphaLeft`, `alphaRight`, `maxLeavesToVisit`, and so on. |
| **Brute-force/sequential search** (`seq_search`) | |
| | No parameters. |

**Note:** mnemonic method names are given in round brackets.

A good compromise is to use `numPivot` somewhat larger than $\sqrt{N}$, where $N$ is the overall number of data points. Similarly, the number of pivots to be indexed (`numPivotIndex`) should be somewhat larger than $\sqrt{\texttt{numPivot}}$. Finally, the number of pivots to be searched (`numPivotSearch`) should be in the order of $\sqrt{\texttt{numPivotIndex}}$.

After selecting the values of `numPivot` and `numPivotIndex`, finding an optimal value of `numPivotSearch`—which provides a necessary recall—requires just a single run of the utility `experiment` (you need to specify all potentially good values of `numPivotSearch` multiple times using the option `--QueryTimeParams`).

### 6.2 SW-graph and HNSW

The basic guidelines are similar for both methods. Specifically, increasing the value of `efConstruction` improves the quality of a constructed graph and leads to higher accuracy of search. However this also leads to longer indexing times. Similarly, increasing the value of `efSearch` improves recall at the expense of longer retrieval time. The reasonable range of values for these parameters is 100-2000.

In the case of SW-graph, the user can also specify the number of sub-searches: `initIndexAttempts` and `initSearchAttempts` used during indexing and retrieval, respectively. However, we find that in most cases the number of sub-searches needs to be set to one. Yet, for large values of `efConstruction` and `efSearch` (e.g., larger than 2000) it sometimes makes sense to increase the number of sub-searches rather than further increasing `efConstruction` and/or `efSearch`.

The recall values are also affected by parameters `NN` (for SW-graph) and `M` (HNSW). Increasing the values of these parameters (to a certain degree) leads to better recall and shorter retrieval times (at the expense of longer indexing time). For low and moderate recall values (e.g., 60-80%) increasing these parameters

may lead to longer retrieval times. The reasonable range of values for these parameters is 5-100.

Finally, in the case of HNSW, there is a trade-off between retrieval performance and indexing time related to the choice of the pruning heuristic (controlled by the parameter `delaunay_type`). Specifically, by default `delaunay_type` is equal to 1. Using `delaunay_type`=1 improves performance—especially at high recall values (> 80%) at the expense of longer indexing times. Therefore, for lower recall values, we recommend using `delaunay_type`=0.

## 7 Extending the code

It is possible to add new spaces and search methods. This is done in three steps, which we only outline here. A more detailed description can be found in § 7.2 and § 7.3.

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 containing a method that creates a class or a method. In this helper file, it is necessary to include the method/space header.

Because we tend to give the helper file the same name as the name of header for a specific method/space, we should not include method/space headers using quotes (in other words, use only *angle* brackets). Such code fails to compile under the Visual Studio. Here is an example of a proper include-directive:

```
#include <method/vptree.h>
```

In the third step, the user adds the registration code to either the file init_spaces.h (for spaces) or to the file init_methods.h (for methods). This step has two sub-steps. First, the user includes the previously created helper file into either `init_spaces.h` or `init_methods.h`. Second, the function `initMethods` or `initSpaces` is extended by adding a macro call that actually registers the space or method in a factory class.

Note that no explicit/manual modification of makefiles (or other configuration files) is required. However, you have to re-run `cmake` each time a new source file is created (addition of header files does not require a `cmake` run). This is necessary to automatically update makefiles so that they include new source files.

Is is noteworthy that all implementations of methods and spaces are mostly 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, which are meant to be a part of the testing framework/library. However, adding new applications does require minor editing of the meta-makefile `CMakeLists.txt` (and re-running `cmake` § 3.1) on Linux, or creation of new Visual Studio sub-projects on Windows (see § 3.2). It

is also possible to create standalone applications that use the library. Please, see § 3.1 and § 3.2 for details.

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 zero-functionality examples can also be used as starting points to develop fully functional code.

### 7.1 Test Workflow

The main benchmarking utility `experiment` parses command line parameters. Then, it creates a space and a search method using the space and the method factories. 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 § 7.2 and § 7.3.

When we create a class representing a search method, the constructor of the class does not create an index in the memory. The index is created using either the function `CreateIndex` (from scratch) or the function `LoadIndex` (from a previously created index image). The index can be saved to disk using the function `SaveIndex`. Note, however, that most methods do not support index (de)-serialization.

Depending on parameters passed to the benchmarking utility, 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 gold standard 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 previously unseen data objects (by calling the function `CheckAndAddToResult`). In both cases, the value of the function `Radius` can be 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 to answer 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.

### 7.2 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 may change in the future releases), `Object` is a very basic class that only keeps a blob of data and blob's 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.

For dense vector spaces the easiest way to create a new space, is to create a functor (function object class) that computes a distance. Then, this function should be used to instantiate a template VectorSpaceGen. A sample implementation of this approach can be found in sample_standalone_app1.cc. However, as we explain below, **additional work** is needed if the space should work correctly with all projection methods (see § 5.3) or any other methods that rely on projections (e.g., OMEDRANK § 5.3.3).

To further illustrate the process 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 space.

The sample files include a template class `DummySpace` (see Table 11), which is declared and defined in the namespace `similarity`. It is a direct ancestor of the class `Space`.

It is possible to provide the complete implementation of the `DummySpace` in the header file. However, this would make compilation slower. 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>;
```

Table 11: A sample space class

```
template <typename dist_t>
class SpaceDummy : public Space<dist_t> {
 public:
  ...
  /** Standard functions to read/write/create objects */
  // Create an object from a (possibly binary) string.
  virtual unique_ptr<Object>
  CreateObjFromStr(IdType id, LabelType label, const string& s,
                   DataFileInputState* pInpState) const;

  // Create a string representation of an object.
  // The string representation may include external ID.
  virtual string
  CreateStrFromObj(const Object* pObj, const string& externId) const;

  // Open a file for reading, fetch a header
  // (if there is any) and memorize an input state
  virtual unique_ptr<DataFileInputState>
  OpenReadFileHeader(const string& inputFile) const;

  // Open a file for writing, write a header (if there is any)
  // and memorize an output state
  virtual unique_ptr<DataFileOutputState>
  OpenWriteFileHeader(const ObjectVector& dataset,
                      const string& outputFile) const;
  /*
   * Read a string representation of the next object in a file as well
   * as its label. Return false, on EOF.
   */
  virtual bool
  ReadNextObjStr(DataFileInputState &, string& strObj, LabelType& label,
                 string& externId) const;

  /*
   * Write a string representation of the next object to a file. We totally delegate
   * this to a Space object, because it may package the string representation, by
   * e.g., in the form of an XML fragment.
   */
  virtual void WriteNextObj(const Object& obj, const string& externId,
                            DataFileOutputState &) const;
  /** End of standard functions to read/write/create objects */

  ...

  /*
   * CreateDenseVectFromObj and GetElemQty() are only needed, if
   * one wants to use methods with random projections.
   */
  virtual void CreateDenseVectFromObj(const Object* obj, dist_t* pVect,
                                      size_t nElem) const {
    throw runtime_error("Cannot create vector for the space: " + StrDesc());
  }
  virtual size_t GetElemQty(const Object* object) const {return 0;}
 protected:
  virtual dist_t HiddenDistance(const Object* obj1,
                                const Object* obj2) const;
  // Don't permit copying and/or assigning
  DISABLE_COPY_AND_ASSIGN(SpaceDummy);
};
```

Most importantly, the user needs to implement the function `HiddenDistance`, which computes the distance between objects, and the function `CreateObjFromStr` that creates a data point object from an instance of a C++ class `string`. For simplicity—even though this is not the most efficient approach—all our spaces create objects from textual representations. However, this is not a principal limitation, because a C++ string can hold binary data as well. Perhaps, the next most important function is `ReadNextObjStr`, which reads a string representation of the next object from a file. A file is represented by a reference to a subclass of the class `DataFileInputState`.

Compared to previous releases, the new `Space` API is substantially more complex. This is necessary to standardize reading/writing of generic objects. In turn, this has been essential to implementing a generic query server. The query server accepts data points in the same format as they are stored in a data file. The above mentioned function `CreateObjFromStr` is used for de-serialization of both the data points stored in a file and query data points passed to the query server.

Additional complexity arises from the need to update space parameters after a space object is created. This permits a more complex storage model where, e.g., parameters are stored in a special dedicated header file, while data points are stored elsewhere, e.g., split among several data files. To support such functionality, we have a function that opens a data file (`OpenReadFileHeader`) and creates a state object (sub-classed from `DataFileInputState`), which keeps the current file(s) state as well as all space-related parameters. When we read data points using the function `ReadNextObjStr`, the state object is updated. The function `ReadNextObjStr` may also read an optional external identifier for an object. When it produces a non-empty identifier it is memorized by the query server and is further used for query processing (see § 2.5). After all data points are read, this state object is supposed to be passed to the `Space` object in the following fashion:

```
unique_ptr<DataFileInputState>
inpState(space->ReadDataset(dataSet, externIds, fileName, maxNumRec));
space->UpdateParamsFromFile(*inpState);
```

For a more advanced implementation of the space-related functions, please, see the file space_vector.cc.

Remember that the function `HiddenDistance` should not be directly accessible by classes that are not friends of the `Space`. As explained in § 7.1, 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`. This is by far not a perfect solution and we are contemplating about better ways to proxy distance computations.

Should we implement a vector space that works properly with projection methods and classic random projections, we need to define functions `GetElemQty` and `CreateDenseVectFromObj`. In the case of a *dense* vector space, `GetElemQty`

should return the number of vector elements stored in the object. For *sparse* vector spaces, it should return zero. The function `CreateDenseVectFromObj` extracts elements stored in a vector. For *dense* vector spaces, it merely copies vector elements to a buffer. For *sparse* space vector spaces, it should do some kind of basic dimensionality reduction. Currently, we do it via the hashing trick (see § A).

Importantly, we need to "tell" the library about the space, by registering the space in the space factory. 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);

  pmgr.CheckUnused();

  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`. To verify that no extra parameters are added, it is recommended to call the function `CheckUnused` (it fires an exception if some parameters are unclaimed). This may also help to identify situations where the user misspells a parameter's name.

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 specify 1 (for `true`) or 0 (for `false`). Note that the function `GetParamRequired` raises an exception, 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)
```

This macro should be placed into the function `initSpaces` in the file init_spaces.h. Last, but not least we need to add the include-directive for the helper function, which creates the class, to the file `init_spaces.h` as follows:

```
#include "factory/space/space_dummy.h"
```

To conlcude, we recommend to make a `Space` object is non-copyable. This can be done by using our macro `DISABLE_COPY_AND_ASSIGN`.

### 7.3 Creating a search method

To illustrate 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 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 (see Table 12). Note again that the constructor of the class does not create an index in the memory. The index is created using either the function `CreateIndex` (from scratch) or the function `LoadIndex` (from a previously created index image). The index can be saved to disk using the function `SaveIndex`. It does not have to be a comprehensive index that contains a copy of the data set. Instead, it is sufficient to memorize only the index structure itself (because the data set is always loaded separately). Also note that most methods do not support index (de)-serialization.

The constructor receives a reference to a space object as well as a reference to an array of data objects. In some cases, e.g., when we wrap existing methods such as the multiprobe LSH (see § 5.2), we create a copy of the data set (simply because is was easier to write the wrapper this way). The framework can be informed about such a situation via the virtual function `DuplicateData`. If this function returns true, the framework "knows" that the data was duplicated. Thus, it can correct an estimate for the memory required by the method.

The function `CreateIndex` receives a parameter object. In our example, the parameter object is used to retrieve the single index-time parameter: `doSeqSearch`. When this parameter value is true, our dummy method carries out a sequential search. Otherwise, it does nothing useful. Again, it is recommended to call the function `CheckUnused` to ensure that the user did not enter parameters with incorrect names. It is also recommended to call the function `ResetQueryTimeParams` (`this` pointer needs to be specified explicitly here) to reset query-time parameters after the index is created (or loaded from disk).

Unlike index-time parameters, query-time parameters can be changed without rebuilding the index by invoking the function `SetQueryTimeParams`. The function `SetQueryTimeParams` accepts a constant reference to a parameter object. The programmer, in turn, creates a parameter manager object to extract

Table 12: A sample search method class

```
template <typename dist_t>
class DummyMethod : public Index<dist_t> {
 public:
  DummyMethod(Space<dist_t>& space,
              const ObjectVector& data) : data_(data), space_(space) {}

  void CreateIndex(const AnyParams& IndexParams) override {
    AnyParamManager  pmgr(IndexParams);
    pmgr.GetParamOptional("doSeqSearch",
                          bDoSeqSearch_,
    // One should always specify the default value of an optional parameter!
                          false
                          );
    // Check if a user specified extra parameters,
    // which can be also misspelled variants of existing ones
    pmgr.CheckUnused();
    // It is recommended to call ResetQueryTimeParams()
    // to set query-time parameters to their default values
    this->ResetQueryTimeParams();
  }

  // SaveIndex is not necessarily implemented
  virtual void SaveIndex(const string& location) override {
    throw runtime_error(
            "SaveIndex is not implemented for method: " + StrDesc());
  }
  // LoadIndex is not necessarily implemented
  virtual void LoadIndex(const string& location) override {
    throw runtime_error(
             "LoadIndex is not implemented for method: " + StrDesc());
  }
  void SetQueryTimeParams(const AnyParams& QueryTimeParams) override;

  // Description of the method, consider printing crucial parameter values
  const std::string StrDesc() const override {
    stringstream str;
    str << "Dummy method: "
        << (bDoSeqSearch_ ? " does seq. search " :
                            " does nothing (really dummy)");
    return str.str();
  }

  //  One needs to implement two search functions.
  void Search(RangeQuery<dist_t>* query, IdType) const override;
  void Search(KNNQuery<dist_t>* query, IdType) const override;

  // If we duplicate data, let the framework know it
  virtual bool DuplicateData() const override { return false; }
 private:
  ...
  // Don't permit copying and/or assigning
  DISABLE_COPY_AND_ASSIGN(DummyMethod);
};
```

actual parameter values. To this end, two functions are used: `GetParamRequired` and `GetParamOptional`. Note that the latter function must be supplied with a mandatory *default* value for the parameter. Thus, the parameter value is properly reset to its default value when the user does not specify the parameter value explicitly (e.g., the parameter specification is omitted when the user invokes the benchmarking utility `experiment`)!

There are two search functions each of which receives two parameters. The first parameter is a pointer to a query (either a range or a $k$-NN query). The second parameter is currently unused. Note again that during the search phase, a search method can compute a distance only by accessing functions `Distance`, `DistanceObjLeft`, or `DistanceObjRight`, which are member functions of a query object. The function `IndexTimeDistance` **should not be used** in a function `Search`, but it can be used in the function `CreateIndex`. If the user attempts to invoke `IndexTimeDistance` during the test phase, **the program will terminate**. [18]

Finally, we need to "tell" the library about the method, by registering the method in the method factory, similarly to registering a space. At runtime, the method is created through a helper function, which accepts several parameters. One parameter is a reference to an object of the type `AllParams`. In our case, the function name is `CreateDummy`:

```
#include <method/dummy.h>

namespace similarity {
template <typename dist_t>
Index<dist_t>* CreateDummy(bool PrintProgress,
                           const string& SpaceType,
                           Space<dist_t>& space,
                           const ObjectVector& DataObjects) {
    return new DummyMethod<dist_t>(space, DataObjects);
}
```

There is an include-directive preceding the creation function, which uses angle brackets. As explained previously, if you opt to using quotes (in the include-directive), the code may not compile under the Visual Studio.

Again, similarly to the case of the space, the method-creating function `CreateDummy` needs to be registered in the method factory in two steps. First, we need to include `dummy.h` into the file init_methods.h as follows:

```
#include "factory/method/dummy.h"
```

Then, this file is further modified by adding the following lines to the function `initMethods`:

[18] 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.

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

If we want our method to work only with integer-valued distances, we only need the following line:

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

When adding the method, please, consider expanding the test utility `test_integr`. This is especially important if for some combination of parameters the method is expected to return all answers (and will have a perfect recall). Then, if we break the code in the future, this will be detected by `test_integr`.

To create a test case, the user needs to add one or more test cases to the file test_integr.cc. A test case is an instance of the class `MethodTestCase`. It encodes the range of plausible values for the following performance parameters: the recall, the number of points closer to the query than the nearest returned point, and the improvement in the number of distance computations.

### 7.4 Creating an application on Linux (inside the framework)

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 § 3.1.

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})
  ```

### 7.5 Creating an application on Windows (inside the framework)

The following description was created for Visual Studio Express 2015. It may be a bit different for newer releases of the Visual Studio. Creating a new sub-project in the Visual Studio is rather straightforward.

In addition, one can use a provided sample project file dummy_app.vcxproj as a template. To this end, one needs to to create a copy of this sample project file and subsequently edit it. One needs to do the following:

- Obtain a new value of the project GUI and put it between the tags `<ProjectGUID>...</ProjectGUID>`;
- Add/delete new files;
- Add/delete/change references to the boost directories (both header files and libraries);
- If the CPU has AVX extension, it may be necessary to enable them as explained in § 3.2.
- Finally, one may manually add an entry to the main project file NonMetricSpaceLib.sln.

## 8 Notes on Efficiency

### 8.1 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 § 4.4 and § 4.5 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!
```

Because we do not know a good way to create/modify Visual Studio project files that enable advanced SSE extensions automatically (depending on whether hardware supports them), the user has to enable these extensions manually. For the instructions, the user is referred to § 3.2.

### 8.2 Cache-friendly Data Layout

In our previous report [7], 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.

## 9 Data Sets

Currently we provide mostly vector space data sets, which come in either dense or sparse format. For simplicity, these are textual formats where each row of the file contains a single vector. If a row starts with a prefix in the form: `label:<non-negative integer value> <white-space>`, the integer value is interpreted as the identifier of a class. These identifiers can be used to compute the accuracy of $k$-NN based classification procedure.

Aside from the prefix, the sparse and dense vectors are stored in a different format. In the 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 element 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
```

The vectors are sparse and most values are not specified. It is up to a designer of the space to decide on the default value for an unspecified vector element. All existing implementations use *zero* as the default value. Again, elements can be separated by spaces or commas/columns instead of spaces.

In addition, the directory `previous_releases_scripts` contains the full set of scripts that can be used to re-produce our NIPS'13, SISAP'13, DA'14, and VLDB'15 results [7,8,41,38]. However, one would need to use older software version (1.0 for NIPS'13 and 1.1 for VLDB'15). Additionally, to reproduce our previous results, one needs to obtain data sets using scripts data/get_data_nips2013.sh and data/get_data_vldb2015.sh. Note that for all evaluations except VLDB'15, you need the previous version of software (1.0), which can be download from here.

If we use any of the provided data sets, 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` or `bunzip2` to uncompress all the data except the Wikipedia (sparse and dense) vectors. The Wikipedia data is compressed using `7z`, which provides superior compression ratios.

## 10 Licensing and Acknowledging the Use of Library Resources

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[19] and to cite our engineering paper [7]:

```
@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},
  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}
}
```

If you re-use a specific implementation, please, consider citing the original authors (if they are known). If you re-use one of the downloaded data sets, please, consider citing one (or more) of the following people:

- authors of software that was used to generate the data;
- authors of the source data set;
- authors who generated data set (if it was not done by us).

In particular, note the following:

- Most of the data sets used in our SISAP'13 and NIPS'13 evaluation were created by by Lawrence Cayton [11]. Our implementation of the bbtree, an exact search method for Bregman divergences, is also based on the code of Cayton.
- The Colors data set originally belongs to the Metric Spaces Library [25].
- All Wikipedia-based data sets were created with a help of the `gensim` library [43].
- The VLDB'15 data set for the Signature Quadratic Form Distance (SQFD) [5,4] was tested using signatures extracted from LSVRC-2014 data set [44].
- Our library incorporates the efficient LSHKIT library as well as the implementation of NN-Descent: an iterative algorithm to construct an approximate $k$-NN-graph [18].
- Note that LSHKIT is distributed under a different license: GNU General Public License version 3 or later. The NN-Descent is distributed under a free license similar to that of the Apache license (see the text of the license here). There is also a newer open-source implementation of the NN-Descent that includes both the graph construction and the search algorithm, but we have not incorporated it yet.

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

## 11 Acknowledgements

Bileg and Leo gratefully acknowledge support by the iAd Center [20] and the Open Advancement of Question Answering Systems (OAQA) group [21]. We also thank Lawrence Cayton for providing data sets (and allowing us to make them public); Nikita Avrelin for implementing the first version of the SW-graph; Yury Malkov for contributing a hierarchical modification of the SW-graph, as well as for guideliness for tuning the original SW-graph; David Novak for the suggestion to use external pivots in permutation algorithms; Daniel Lemire for contributing the implementation of the original Schlegel et al. [47] intersection algorithm. We also thank Andrey Savchenko, Alexander Ponomarenko, and Yury Malkov for suggestions to improve the library and the documentation.

## 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. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt. Practical and optimal lsh for angular distance. In *Advances in Neural Information Processing Systems*, pages 1225–1233, 2015.
4. C. Beecks. *Distance based similarity models for content based multimedia retrieval.* PhD thesis, 2013.
5. 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.
6. 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.
7. 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.
8. L. Boytsov and B. Naidan. Learning to prune in metric and non-metric spaces. In *Advances in Neural Information Processing Systems*, 2013.
9. T. Bozkaya and M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS)*, 24(3):361–404, 1999.
10. 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.

[20] `http://www.iad-center.com/`
[21] `http://oaqa.github.io/`

11. 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.
12. L. Cayton. Efficient bregman range search. In *Advances in Neural Information Processing Systems*, pages 243–251, 2009.
13. 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.
14. E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.
15. E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
16. 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.
17. W. Dong. *High-Dimensional Similarity Search for Large Datasets*. PhD thesis, Princeton University, 2011.
18. 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.
19. 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.
20. D. M. Endres and J. E. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, 2003.
21. A. Esuli. Use of permutation prefixes for efficient and scalable approximate similarity search. *Inf. Process. Manage.*, 48(5):889–902, Sept. 2012.
22. 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.
23. 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.
24. 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.
25. K. Figueroa, G. Navarro, and E. Chávez. Metric Spaces Library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.
26. 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.
27. L. V. Hedges and J. L. Vevea. Fixed-and random-effects models in meta-analysis. *Psychological methods*, 3(4):486–504, 1998.
28. 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.
29. G. King. How not to lie with statistics: Avoiding common mistakes in quantitative political science. *American Journal of Political Science*, pages 666–687, 1986.

30. 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.
31. V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR,*, 163(4):845–848, 1966.
32. 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.
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.
38. B. Naidan, L. Boytsov, and E. Nyberg. Permutation search methods are efficient, yet faster search is possible. *PVLDB*, 8(12):1618–1629, 2015.
39. G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46, 2002.
40. 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.
41. A. Ponomarenko, N. Avrelin, B. Naidan, and L. Boytsov. Comparative analysis of data structures for approximate nearest neighbor search. In *DATA ANALYTICS 2014, The Third International Conference on Data Analytics*, pages 125–130, 2014.
42. 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`.
43. R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.
44. 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.
45. 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.
46. D. Sankoff. The early introduction of dynamic programming into computational biology. *Bioinformatics*, 16(1):41–47, 2000.

47. B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. In *ADMS@ VLDB*, pages 1–8, 2011.
48. C. Silpa-Anan and R. I. Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, 2008.
49. T. Skopal. Unified framework for fast exact and approximate search in dissimilarity spaces. *ACM Trans. Database Syst.*, 32(4), Nov. 2007.
50. 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.
51. E. S. Tellez, E. Chávez, and G. Navarro. Succinct nearest neighbor search. *Information Systems*, 38(7):1019–1030, 2013.
52. J. Uhlmann. Satisfying general proximity similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.
53. V. Velichko and N. Zagoruyko. Automatic recognition of 200 words. *International Journal of Man-Machine Studies*, 2(3):223 – 234, 1970.
54. T. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57, 1968.
55. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.
56. 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.
57. 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. [22] 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 § 9), it is safe to use the value `intermDim`=4096.

[22] 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.

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. [6]).

### A.2 FastMap

FastMap introduced by Faloutsos and Lin [23] is also a type of the random-projection method. At indexing time, we randomly select $projDim$ pairs $A_i$ and $B_i$. The $i$-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 $i$-th coordinate of point $x$ in the projected space is computed as simply $d(x, \pi_i)$, where $\pi_i$ is a pivot in the original space, i.e., a randomly selected reference point. Pivots are selected once during indexing time.

### 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 § 5.4.