FLANN - Fast Library for Approximate Nearest Neighbors

User Manual

Marius Muja, mariusm@cs.ubc.ca
David Lowe, lowe@cs.ubc.ca

November 26, 2009

1 Introduction

We can define the *nearest neighbor search (NSS)* problem in the following way: given a set of points $P = p_1, p_2, \ldots, p_n$ in a metric space X, these points must be preprocessed in such a way that given a new query point $q \in X$, finding the point in P that is nearest to q can be done quickly.

The problem of nearest neighbor search is one of major importance in a variety of applications such as image recognition, data compression, pattern recognition and classification, machine learning, document retrieval systems, statistics and data analysis. However, solving this problem in high dimensional spaces seems to be a very difficult task and there is no algorithm that performs significantly better than the standard brute-force search. This has lead to an increasing interest in a class of algorithms that perform approximate nearest neighbor searches, which have proven to be a good-enough approximation in most practical applications and in most cases, orders of magnitude faster that the algorithms performing the exact searches.

FLANN (Fast Library for Approximate Nearest Neighbors) is a library for performing fast approximate nearest neighbor searches. FLANN is written in the C++ programming language. FLANN can be easily used in many contexts through the C, MATLAB and Python bindings provided with the library.

1.1 Quick Start

This section contains small examples of how to use the FLANN library from different programming languages (C++, C, MATLAB and Python).

- **C++**

```
// file flann_example.cpp

#include <flann/flann.hpp>
#include <flann/io/hdf5.h>

#include <stdio.h>

int main(int argc, char** argv)
{
    int nn = 3;

    flann::Matrix<float> dataset;
    flann::Matrix<float> query;
    flann::load_from_file(dataset, "dataset.hdf5","dataset");
    flann::load_from_file(query, "dataset.hdf5","query");

    flann::Matrix<int> indices(new int[query.rows*nn], query.rows, nn);
    flann::Matrix<float> dists(new float[query.rows*nn], query.rows, nn);

    // construct an randomized kd-tree index using 4 kd-trees
    flann::Index<flann::L2<float> > index(dataset, flann::KDTreeIndexParams(4));
    index.buildIndex();

    // do a knn search, using 128 checks
    index.knnSearch(query, indices, dists, nn, flann::SearchParams(128));

    flann::save_to_file(indices,"result.hdf5","result");
```

```
    dataset.free();
    query.free();
    indices.free();
    dists.free();

    return 0;
}
```

- **C**

```
/* file flann_example.c */

#include "flann.h"
#include <stdio.h>
#include <assert.h>

/* Function that reads a dataset */
float* read_points(char* filename, int *rows, int *cols);

int main(int argc, char** argv)
{
   int rows,cols;
   int t_rows, t_cols;
   float speedup;

   /* read dataset points from file dataset.dat */
   float* dataset = read_points("dataset.dat", &rows, &cols);
   float* testset = read_points("testset.dat", &t_rows, &t_cols);

   /* points in dataset and testset should have the same dimensionality */
   assert(cols==t_cols);

   /* number of nearest neighbors to search */
   int nn = 3;
   /* allocate memory for the nearest-neighbors indices */
   int* result = (int*) malloc(t_rows*nn*sizeof(int));
   /* allocate memory for the distances */
   float* dists = (float*) malloc(t_rows*nn*sizeof(float));

   /* index parameters are stored here */
   struct FLANNParameters p = DEFAULT_FLANN_PARAMETERS;
   p.algorithm = AUTOTUNED;  /* or KDTREE, KMEANS, ... /*
   p.target_precision = 0.9;  /* want 90% target precision */

   /* compute the 3 nearest-neighbors of each point in the testset */
   flann_find_nearest_neighbors(dataset, rows, cols, testset, t_rows,
result, dists, nn, &p);

   ...
   free(dataset);
   free(testset);
   free(result);
   free(dists);

   return 0;
}
```

- **MATLAB**

```
% create random dataset and test set
dataset = single(rand(128,10000));
testset = single(rand(128,1000));

% define index and search parameters
```

```
params.algorithm = 'kdtree';
params.trees = 8;
params.checks = 64;

% perform the nearest-neighbor search
[result, dists] = flann_search(dataset,testset,5,params);
```

- **Python**

```
from pyflann import *
from numpy import *
from numpy.random import *

dataset = rand(10000, 128)
testset = rand(1000, 128)

flann = FLANN()
result,dists = flann.nn(dataset,testset,5,algorithm="kmeans",
               branching=32, iterations=7, checks=16);
```

2 Downloading and compiling FLANN

FLANN can be downloaded from the following address:

`http://www.cs.ubc.ca/~mariusm/flann`

After downloading and unpacking, the following files and directories should be present:

- `bin`: directory various for scripts and binary files
- `doc`: directory containg this documentation
- `examples`: directory containg examples of using FLANN
- `src`: directory containg the source files
- `test`: directory containg unit tests for FLANN

To compile the flann library the *CMake*[1] build system is required. Below is an example of how the FLANN library can be compiled on Linux (replace x.y with the corresponding version number).

```
$ cd flann-x.y-src
$ BUILD_TYPE=release INSTALL_PREFIX=<some directory> make install
```

3 Using FLANN

3.1 Using FLANN from C++

The core of the FLANN library is written in C++. To make use of the full power and flexibility of the templated code one should use the C++ bindings

[1] http://www.cmake.org/

if possible. To use the C++ bindings, the library header file `flann.hpp` must be included and the library `libflann_cpp.so` (for linking dynamically) or the `libflann_cpp_s.a` (for linking statically) must be linked in. An example of the compile command that must be used will look something like this:

```
g++ flann_example.cpp -I $FLANN_ROOT/include -L $FLANN_ROOT/lib -o flann_example_cpp
 -lflann_cpp
```

where `$FLANN_ROOT` is the library main directory.

The following sections describe the public C++ API.

3.1.1 flann::Index

The FLANN nearest neighbor index class. This class is used to abstract different types of nearest neighbor search indexes.

```
namespace flann
{
    template<typename Distance>
    class Index
    {
        typedef typename Distance::ElementType ElementType;
        typedef typename Distance::ResultType DistanceType;
    public:
        Index(const Matrix<ElementType>& features, const IndexParams& params);

        ~Index();

        void buildIndex();

        void knnSearch(const Matrix<ElementType>& queries,
                       Matrix<int>& indices,
                       Matrix<DistanceType>& dists,
                       int knn,
                       const SearchParams& params);

        int radiusSearch(const Matrix<ElementType>& query,
                         Matrix<int>& indices,
                         Matrix<DistanceType>& dists,
                         float radius,
                         const SearchParams& params);

        void save(std::string filename);

        int veclen() const;

        int size() const;

        const IndexParams* getIndexParameters();
    };
}
```

flann::Index::Index Constructs a nearest neighbor search index for a given dataset.

```
Index(const Matrix<ElementType>& features, const IndexParams& params);
```

features Matrix containing the features(points) to index. The size of the matrix is $num\_features \times dimensionality$.

params Structure containing the index parameters. The type of index that will be constructed depends on the type of this parameter. The possible parameter types are:

LinearIndexParams When passing an object of this type, the index will perform a linear, brute-force search.

```
struct LinearIndexParams : public IndexParams
{
};
```

KDTreeIndexParams When passing an object of this type the index constructed will consist of a set of randomized kd-trees which will be searched in parallel.

```
struct KDTreeIndexParams : public IndexParams
{
      KDTreeIndexParams( int trees = 4 );
};
```

trees The number of parallel kd-trees to use. Good values are in the range [1..16]

KMeansIndexParams When passing an object of this type the index constructed will be a hierarchical k-means tree.

```
struct KMeansIndexParams : public IndexParams
{
        KMeansIndexParams( int branching = 32,
                        int iterations = 11,
                        flann_centers_init_t centers_init = CENTERS_RANDOM,
                        float cb_index = 0.2 );
};
```

branching The branching factor to use for the hierarchical k-means tree

iterations The maximum number of iterations to use in the k-means clustering stage when building the k-means tree. If a value of -1 is used here, it means that the k-means clustering should be iterated until convergence

centers_init The algorithm to use for selecting the initial centers when performing a k-means clustering step. The possible values are CENTERS_RANDOM (picks the initial cluster centers randomly), CENTERS_GONZALES (picks the initial centers using Gonzales' algorithm) and CENTERS_KMEANSPP (picks the initial centers using the algorithm suggested in [AV07])

cb_index This parameter (cluster boundary index) influences the way exploration is performed in the hierarchical kmeans tree. When `cb_index` is zero the next kmeans domain to be explored is choosen to be the one with the closest center. A value greater then zero also takes into account the size of the domain.

CompositeIndexParams When using a parameters object of this type the index created combines the randomized kd-trees and the hierarchical k-means tree.

```
struct CompositeIndexParams : public IndexParams
{
        CompositeIndexParams( int trees = 4,
                              int branching = 32,
                              int iterations = 11,
                              flann_centers_init_t centers_init = CENTERS_RANDOM,
                              float cb_index = 0.2 );
};
```

AutotunedIndexParams When passing an object of this type the index created is automatically tuned to offer the best performance, by choosing the optimal index type (randomized kd-trees, hierarchical kmeans, linear) and parameters for the dataset provided.

```
struct AutotunedIndexParams : public IndexParams
{
        autotunedindexparams( float target_precision = 0.9,
                              float build_weight = 0.01,
                              float memory_weight = 0,
                              float sample_fraction = 0.1 );
};
```

target_precision Is a number between 0 and 1 specifying the percentage of the approximate nearest-neighbor searches that return the exact nearest-neighbor. Using a higher value for this parameter gives more accurate results, but the search takes longer. The optimum value usually depends on the application.

build_weight Specifies the importance of the index build time raported to the nearest-neighbor search time. In some applications it's acceptable for the index build step to take a long time if the subsequent searches in the index can be performed very fast. In other applications it's required that the index be build as fast as possible even if that leads to slightly longer search times.

memory_weight Is used to specify the tradeoff between time (index build time and search time) and memory used by the index. A value less than 1 gives more importance to the time spent and a value greater than 1 gives more importance to the memory usage.

sample_fraction Is a number between 0 and 1 indicating what fraction of the dataset to use in the automatic parameter configuration algorithm. Running the algorithm on the full dataset gives the most

accurate results, but for very large datasets can take longer than desired. In such case using just a fraction of the data helps speeding up this algorithm while still giving good approximations of the optimum parameters.

SavedIndexParams This object type is used for loading a previously saved index from the disk.

```
struct SavedIndexParams : public IndexParams
{
      SavedIndexParams( std::string filename );
};
```

filename The filename in which the index was saved.

3.1.2 flann::Index::buildIndex

Constructs the nearest neighbor search index using the parameters provided to the constructor (with the exception of saved index type).

```
      void buildIndex();
```

3.1.3 flann::Index::knnSearch

Performs a K-nearest neighbor search for a given query point using the index.

```
void Index::knnSearch(const Matrix<ElementType>& queries,
                Matrix<int>& indices,
                Matrix<DistanceType>& dists,
                int knn,
                const SearchParams& params);
```

query Matrix containing the query points. Size of matrix is ($num\_queries \times dimentionality$)

indices Matrix that will contain the indices of the K-nearest neighbors found (size should be at least $num\_queries \times knn$)

dists Matrix that will contain the distances to the K-nearest neighbors found. (size should be at least $num\_queries \times knn$). The distance values are computed by the distance function used (see `flann::set_distance_type` below), for example in the case of euclidean distance function, this will contain the squared euclidean distances.

knn Number of nearest neighbors to search for.

params Search parameters. Structure currently contains a single field, `checks` specifying the number of times the tree(s) in the index should be recursively traversed. A higher value for this parameter would give better search precision, but also take more time. If automatic configuration was

used when the index was created, the number of checks required to achieve the specified precision was also computed, in which case this parameter is ignored.

3.1.4 flann::Index::radiusSearch

Performs a radius nearest neighbor search for a given query point.

```
int Index::radiusSearch(const Matrix<ElementType>& query,
                  Matrix<int>& indices,
                  Matrix<DistanceType>& dists,
                  float radius,
                  const SearchParams& params);
```

query The query point

indices Vector that will contain the indices of the points found within the search radius in decreasing order of the distance to the query point. If the number of neighbors in the search radius is bigger than the size of this vector, the ones that don't fit in the vector are ignored.

dists Vector that will contain the distances to the points found within the search radius

radius The search radius

params Search parameters

The method returns the number of nearest neighbors found.

3.1.5 flann::Index::save

Saves the index to a file.

```
  void Index::save(std::string filename);
```

filename The file to save the index to

3.1.6 flann::hierarchicalClustering

Clusters the given points by constructing a hierarchical k-means tree and choosing a cut in the tree that minimizes the clusters' variance.

```
template <typename Distance>
int hierarchicalClustering(const Matrix<typename Distance::ElementType>& features,
                        Matrix<typename Distance::ResultType>& centers,
                        const KMeansIndexParams& params,
                        Distance d = Distance())
```

features The points to be clustered

centers The centers of the clusters obtained. The number of rows in this matrix represents the number of clusters desired. However, because of the way the cut in the hierarchical tree is choosen, the number of clusters computed will be the highest number of the form $(branching - 1) * k + 1$ that's lower than the number of clusters desired, where *branching* is the tree's branching factor (see description of the KMeansIndexParams).

params Parameters used in the construction of the hierarchical k-means tree

The function returns the number of clusters computed.

3.2 Using FLANN from C

FLANN can be used in C programs through the C bindings provided with the library. Because there is not template support in C, there are bindings provided for the following data types: `unsigned char`, `int`, `float` and `double`. For each of the functions below there is a corresponding version for each of the for data types, for example for the function:

```
flan_index_t flann_build_index(float* dataset, int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
```

there are also the following versions:

```
flan_index_t flann_build_index_byte(unsigned char* dataset,
        int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
flan_index_t flann_build_index_int(int* dataset,
        int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
flan_index_t flann_build_index_float(float* dataset,
        int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
flan_index_t flann_build_index_double(double* dataset,
        int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
```

3.2.1 flann_build_index()

```
flan_index_t flann_build_index(float* dataset,
        int rows,
        int cols,
        float* speedup,
        struct FLANNParameters* flann_params);
```

This function builds an index and return a reference to it. The arguments expected by this function are as follows:

dataset, rows and cols - are used to specify the input dataset of points: dataset is a pointer to a rows × cols matrix stored in row-major order.

speedup - is used to return the approximate speedup over linear search achieved when using the automatic index and parameter configuration (see section 3.3.1)

flann_params - is a structure containing the parameters passed to the function. This structure is defined as follows:

```
struct FLANNParameters {
  enum flann_algorithm_t algorithm;         /* the algorithm to use */

  /* search parameters */
  int checks;            /* how many leafs (features) to check in one search */
  float cb_index;        /* cluster boundary index. Used when searching the
                            kmeans tree */

  /*  kdtree index parameters */
  int trees;                /* number of randomized trees to use (for kdtree) */

  /* kmeans index parameters */
  int branching;            /* branching factor (for kmeans tree) */
  int iterations;           /* max iterations to perform in one kmeans cluetering
                               (kmeans tree) */
  enum flann_centers_init_t centers_init;  /* algorithm used for picking the initial
                                  cluster centers for kmeans tree */

  /* autotuned index parameters */
  float target_precision;    /* precision desired (used for autotuning, -1 otherwise) */
  float build_weight;        /* build tree time weighting factor */
  float memory_weight;       /* index memory weigthing factor */
  float sample_fraction;     /* what fraction of the dataset to use for autotuning */

  /* other parameters */
  enum flann_log_level_t log_level;   /* determines the verbosity of each flann function */
  long random_seed;                          /* random seed to use */
};
```

The `algorithm` and `centers_init` fields can take the following values:

```
enum flann_algorithm_t {
        LINEAR = 0,
        KDTREE = 1,
        KMEANS = 2,
        COMPOSITE = 3,
        SAVED = 254,
        AUTOTUNED = 255
};

enum flann_centers_init_t {
        CENTERS_RANDOM = 0,
        CENTERS_GONZALES = 1,
        CENTERS_KMEANSPP = 2
};
```

The `algorithm` field is used to manually select the type of index used. The `centers_init` field specifies how to choose the inital cluster centers when performing the hierarchical k-means clustering (in case the algorithm

used is k-means): CENTERS_RANDOM chooses the initial centers randomly, CENTERS_GONZALES chooses the initial centers to be spaced apart from each other by using Gonzales' algorithm and CENTERS_KMEANSPP chooses the initial centers using the algorithm proposed in [AV07].

The fields: checks, cb_index, trees, branching, iterations, target_precision, build_weight, memory_weight and sample_fraction have the same meaning as described in 3.1.1.

The random_seed field contains the random seed useed to initialize the random number generator.

The field log_level controls the verbosity of the messages generated by the FLANN library functions. It can take the following values:

```
enum flann_log_level_t {
    LOG_NONE = 0,
    LOG_FATAL = 1,
    LOG_ERROR = 2,
    LOG_WARN = 3,
    LOG_INFO = 4
};
```

3.2.2 flann_find_nearest_neighbors_index()

```
int flann_find_nearest_neighbors_index(FLANN_INDEX index_id,
        float* testset,
        int trows,
        int* indices,
        float* dists,
        int nn,
        int checks,
        struct FLANNParameters* flann_params);
```

This function searches for the nearest neighbors of the testset points using an index already build and referenced by index_id. The testset is a matrix stored in row-major format with trows rows and the same number of columns as the dimensionality of the points used to build the index. The function computes nn nearest neighbors for each point in the testset and stores them in the indices matrix (which is a trows $\times$ nn matrix stored in row-major format). The memory for the result matrix must be allocated before the flann_find_nearest_neighbors_index() function is called. The distances to the nearest neighbors found are stored in the dists matrix. The checks parameter specifies how many tree traversals should be performed during the search.

3.2.3 flann_find_nearest_neighbors()

```
int flann_find_nearest_neighbors(float* dataset,
        int rows,
        int cols,
        float* testset,
        int trows,
```

```
        int* indices,
        float* dists,
        int nn,
        struct FLANNParameters* flann_params);
```

This function is similar to the `flann_find_nearest_neighbors_index()` function, but instread of using a previously constructed index, it constructs the index, does the nearest neighbor search and deletes the index in one step.

3.2.4 flann_radius_search()

```
int flann_radius_search(FLANN_INDEX index_ptr,
        float* query,   /* query point */
        int* indices, /* array for storing the indices */
        float* dists, /* similar, but for storing distances */
        int max_nn,  /* size of arrays indices and dists */
        float radius, /* search radius (squared radius for euclidian metric) */
        int checks,  /* number of features to check, sets the level
                        of approximation */
        FLANNParameters* flann_params);
```

This function performs a radius search to single query point. The indices of the neighbors found and the distances to them are stored in the `indices` and dists `arrays`. The `max_nn` parameter sets the limit of the neighbors that will be returned (the size of the `indices` and `dists` arrays must be at least `max_nn`).

3.2.5 flann_save_index()

```
int flann_save_index(flann_index_t index_id,
        char* filename);
```

This function saves an index to a file. The dataset for which the index was built is not saved with the index.

3.2.6 flann_load_index()

```
flann_index_t flann_load_index(char* filename,
        float* dataset,
        int rows,
        int cols);
```

This function loads a previously saved index from a file. Since the dataset is not saved with the index, it must be provided to this function.

3.2.7 flann_free_index()

```
int flann_free_index(FLANN_INDEX index_id,
        struct FLANNParameters* flann_params);
```

This function deletes a previously constructed index and frees all the memory used by it.

3.2.8 flann_set_distance_type

This function chooses the distance function to use when computing distances between data points.

```
void flann_set_distance_type(enum flann_distance_t distance_type, int order);
```

distance_type The distance type to use. Possible values are

```
enum flann_distance_t {
        EUCLIDEAN = 1,  // squared euclidean distance
        MANHATTAN = 2,
        MINKOWSKI = 3,
        HIK       = 5,
        HELLINGER = 6,
        CS        = 7,   // chi-square
        KL        = 8,   // kullback-leibler divergence
};
```

order Used in for the `MINKOWSKI` distance type, to choose the order of the Minkowski distance.

3.2.9 flann_compute_cluster_centers()

Performs hierarchical clustering of a set of points (see 3.1.6).

```
int flann_compute_cluster_centers(float* dataset,
      int rows,
      int cols,
      int clusters,
      float* result,
      struct FLANNParameters* flann_params);
```

See section 1.1 for an example of how to use the C/C++ bindings.

3.3 Using FLANN from MATLAB

The FLANN library can be used from MATLAB through the following wrapper functions: `flann_build_index`, `flann_search`, `flann_save_index`, `flann_load_index`, `flann_set_distance_type` and `flann_free_index`. The `flann_build_index` function creates a search index from the dataset points, `flann_search` uses this index to perform nearest-neighbor searches, `flann_save_index` and `flann_load_index` can be used to save and load an index to/from disk, `flann_set_distance_type` is used to set the distance type to be used when building an index and `flann_free_index` deletes the index and releases the memory it uses.

The following sections describe in more detail the FLANN matlab wrapper functions and show examples of how they may be used.

3.3.1 flann_build_index

This function creates a search index from the initial dataset of points, index used later for fast nearest-neighbor searches in the dataset.

```
[index, parameters, speedup] = flann_build_index(dataset, build_params);
```

The arguments passed to the `flann_build_index` function have the following meaning:

`dataset` is a $d \times n$ matrix containing n d-dimensional points

`build_params` - is a MATLAB structure containing the parameters passed to the function.

The `build_params` is used to specify the type of index to be built and the index parameters. These have a big impact on the performance of the new search index (nearest-neighbor search time) and on the time and memory required to build the index. The optimum parameter values depend on the dataset characteristics (number of dimensions, distribution of points in the dataset) and on the application domain (desired precision for the approximate nearest neighbor searches). The `build_params` argument is a structure that contains one or more of the following fields:

`algorithm` - the algorithm to use for building the index. The possible values are: `'linear'`, `'kdtree'`, `'kmeans'`, `'composite'` or `'autotuned'`. The `'linear'` option does not create any index, it uses brute-force search in the original dataset points, `'kdtree'` creates one or more randomized kd-trees, `'kmeans'` creates a hierarchical kmeans clustering tree, `'composite'` is a mix of both kdtree and kmeans trees and the `'autotuned'` automatically determines the best index and optimum parameters using a cross-validation technique.

Autotuned index: in case the algorithm field is `'autotuned'`, the following fields should also be present:

`target_precision` - is a number between 0 and 1 specifying the percentage of the approximate nearest-neighbor searches that return the exact nearest-neighbor. Using a higher value for this parameter gives more accurate results, but the search takes longer. The optimum value usually depends on the application.

`build_weight` - specifies the importance of the index build time raported to the nearest-neighbor search time. In some applications it's acceptable for the index build step to take a long time if the subsequent searches in the index can be performed very fast. In other applications it's required that the index be build as fast as possible even if that leads to slightly longer search times. (Default value: 0.01)

`memory_weight` - is used to specify the tradeoff between time (index build time and search time) and memory used by the index. A value less than 1 gives more importance to the time spent and a value greater than 1 gives more importance to the memory usage.

`sample_fraction` - is a number between 0 and 1 indicating what fraction of the dataset to use in the automatic parameter configuration algorithm. Running the algorithm on the full dataset gives the most accurate results, but for very large datasets can take longer than desired. In such case, using just a fraction of the data helps speeding up this algorithm, while still giving good approximations of the optimum parameters.

Randomized kd-trees index: in case the algorithm field is `'kdtree'`, the following fields should also be present:

`trees` - the number of randomized kd-trees to create.

Hierarchical k-means index: in case the algorithm type is `'means'`, the following fields should also be present:

`branching` - the branching factor to use for the hierarchical kmeans tree creation. While kdtree is always a binary tree, each node in the kmeans tree may have several branches depending on the value of this parameter.

`iterations` - the maximum number of iterations to use in the kmeans clustering stage when building the kmeans tree. A value of -1 used here means that the kmeans clustering should be performed until convergence.

`centers_init` - the algorithm to use for selecting the initial centers when performing a kmeans clustering step. The possible values are 'random' (picks the initial cluster centers randomly), 'gonzales' (picks the initial centers using the Gonzales algorithm) and 'kmeanspp' (picks the initial centers using the algorithm suggested in [AV07]). If this parameters is omitted, the default value is 'random'.

`cb_index` - this parameter (cluster boundary index) influences the way exploration is performed in the hierarchical kmeans tree. When `cb_index` is zero the next kmeans domain to be explored is choosen to be the one with the closest center. A value greater then zero also takes into account the size of the domain.

Composite index: in case the algorithm type is `'composite'`, the fields from both randomized kd-tree index and hierarchical k-means index should be present.

The `flann_build_index` function returns the newly created `index`, the `parameters` used for creating the index and, if automatic configuration was used, an estimation of the `speedup` over linear search that is achieved when searching the index. Since the parameter estimation step is costly, it is possible to save the computed parameters and reuse them the next time an index is created from similar data points (coming from the same distribution).

3.3.2 flann_search

This function performs nearest-neighbor searches using the index already created:

```
[result, dists] = flann_search(index, testset, k, parameters);
```

The arguments required by this function are:

`index` - the index returned by the `flann_build_index` function

`testset` - a $d \times m$ matrix containing m test points whose k-nearest-neighbors need to be found

`k` - the number of nearest neighbors to be returned for each point from `testset`

`parameters` - structure containing the search parameters. Currently it has only one member, `parameters.checks`, denoting the number of times the tree(s) in the index should be recursively traversed. A higher value for this parameter would give better search precision, but also take more time. If automatic configuration was used when the index was created, the number of checks required to achieve the specified precision is also computed. In such case, the parameters structure returned by the `flann_build_index` function can be passed directly to the `flann_search` function.

The function returns two matrices, each of size $k \times m$. The first one contains, in which each column, the indexes (in the dataset matrix) of the k nearest neighbors of the corresponding point from testset, while the second one contains the corresponding distances. The second matrix can be omitted when making the call if the distances to the nearest neighbors are not needed.

For the case where a single search will be performed with each index, the `flann_search` function accepts the dataset instead of the index as first argument, in which case the index is created searched and then deleted in one step. In this case the parameters structure passed to the `flann_search` function must also contain the fields of the `build_params` structure that would normally be passed to the `flann_build_index` function if the index was build separately.

```
[result, dists] = flann_search(dataset, testset, k, parameters);
```

3.3.3 flann_save_index

This function saves an index to a file so that it can be reused at a later time without the need to recompute it. Only the index will be saved to the file, not also the data points for which the index was created, so for the index to be reused the data points must be saved separately.

```
flann_save_index(index, filename)
```

The argumenst required by this function are:

`index` - the index to be saved, created by `flann_build_index`

`filename` - the name of the file in which to save the index

3.3.4 flann_load_index

This function loads a previously saved index from a file. It needs to be passed as a second parameter the dataset for which the index was created, as this is not saved together with the index.

```
index = flann_load_index(filename, dataset)
```

The argumenst required by this function are:

`filename` - the file from which to load the index

`dataset` - the dataset for which the index was created

This function returns the index object.

3.3.5 flann_set_distance_type

This function chooses the distance function to use when computing distances between data points.

```
flann_set_distance_type(type, order)
```

The argumenst required by this function are:

`type` - the distance type to use. Possible values are: `'euclidean'`, `'manhattan'`, `'minkowski'`, `'max_dist'` ($L\_infinity$ - distance type is not valid for kd-tree index type since it's not dimensionwise additive), `'hik'` (histogram intersection kernel), `'hellinger'`,`'cs'` (chi-square) and `'kl'` (Kullback-Leibler).

`order` - only used if distance type is `'minkowski'` and represents the order of the minkowski distance.

3.3.6 flann_free_index

This function must be called to delete an index and release all the memory used by it:

```
flann_free_index(index);
```

3.3.7 Examples

Let's look at a few examples showing how the functions described above are used:

3.3.8 Example 1:

In this example the index is constructed using automatic parameter estimation, requesting 90% as desired precision and using the default values for the build time and memory usage factors. The index is then used to search for the nearest-neighbors of the points in the testset matrix and finally the index is deleted.

```
dataset = single(rand(128,10000));
testset = single(rand(128,1000));

build_params.target_precision = 0.9;
build_params.build_weight = 0.01;
build_params.memory_weight = 0;

[index, parameters] = flann_build_index(dataset, build_params);

result = flann_search(index,testset,5,parameters);

flann_free_index(index);
```

3.3.9 Example 2:

In this example the index constructed with the parameters specified manually.

```
dataset = single(rand(128,10000));
testset = single(rand(128,1000));

index = flann_build_index(dataset,struct('algorithm','kdtree','trees',8));

result = flann_search(index,testset,5,struct('checks',128));

flann_free_index(index);
```

3.3.10 Example 3:

In this example the index creation, searching and deletion are all performed in one step:

```
dataset = single(rand(128,10000));
testset = single(rand(128,1000));

[result,dists] = flann_search(dataset,testset,5,struct('checks',128,'algorithm',...
    'kmeans','branching',64,'iterations',5));
```

3.4 Using FLANN from python

FLANN can be used from python programs using the python bindings distributed with the library. The python bindings can be installed on a system using the distutils script provided (setup.py), by running the following command in the build/python directory:

```
$ python setup.py install
```

The python bindings also require the numpy package to be installed.

To use the python FLANN bindings the package `pyflann` must be imported (see the python example in section 1.1). This package contains a class called FLANN that handles the nearest-neighbor search operations. This class containg the following methods:

`def build_index(self, dataset, **kwargs) :`

This method builds and internally stores an index to be used for future nearest neighbor matchings. It erases any previously stored index, so in order to work with multiple indexes, multiple instances of the FLANN class must be used. The `dataset` argument must be a 2D numpy array or a matrix. The rest of the arguments that can be passed to the method are the same as those used in the `build_params` structure from section 3.3.1. Similar to the MATLAB version, the index can be created using manually specified parameters or the parameters can be automatically computed (by specifying the target_precision, build_weight and memory_weight arguments).

The method returns a dictionary containing the parameters used to construct the index. In case automatic parameter selection is used, the dictionary will also contain the number of checks required to achieve the desired target precision and an estimation of the speedup over linear search that the library will provide.

`def nn_index(self, testset, num_neighbors = 1, **kwargs) :`

This method searches for the `num_neighbors` nearest neighbors of each point in `testset` using the index computed by `build_index`. Additionally,

a parameter called checks, denoting the number of times the index tree(s) should be recursivelly searched, must be given.

Example:

```
from pyflann import *
from numpy import *
from numpy.random import *

dataset = rand(10000, 128)
testset = rand(1000, 128)

flann = FLANN()
params = flann.build_index(dataset, target_precision=0.9, log_level = "info");
print params

result, dists = flann.nn_index(testset,5, checks=params["checks"]);
```

`def nn(self, dataset, testset, num_neighbors = 1, **kwargs) :`
This method builds the index, performs the nearest neighbor search and deleted the index, all in one step.

`def save_index(self, filename) :`
This method saves the index to a file. The dataset form which the index was build is not saved.

`def load_index(self, filename, pts) :`
Load the index from a file. The dataset for which the index was build must also be provided since is not saved with the index.

`def set_distance_type(distance_type, order = 0) :`
This function (part of the pyflann module) sets the distance type to be used. See 3.3.5 for possible values of the distance_type.

See section 1.1 for an example of how to use the Python bindings.

References

[AV07] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2007.