FLANN - Fast Library for Approximate Nearest Neighbors

User Manual

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

February 27, 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) and from the command line.

- **C/C++**

```
// file flann_example.cc

#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 = new int[t_rows*nn];
```

```
   // allocate memory for the distances
   float* dists = new float[t_rows*nn];

   // index parameters are stored here
   FLANNParameters p;
   // want 90% target precision
   // the rest of the parameters are automatically computed
   p.target_precision = 0.9;
   // 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);

   // ...

   delete[] dataset;
   delete[] testset;
   delete[] result;
   delete[] 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);
```

- **Command line application**

```
$ flann compute_nn --input-file=dataset.dat --test-file=testset.dat
--algorithm=kdtree --trees=8 --checks=64 --nn=5 --output-file=nn.dat
Reading input dataset from dataset.dat
Building index
Building index took: 0.76
Reading test dataset from testset.dat
Searching for nearest neighbors
Searching took 0.06 seconds
Writing matches to nn.dat
```

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:

- `src`: directory containg the source files
- `doc`: directory containg this documentation
- `bin`: directory various for scripts and binary files

In addition, the FLANN libary will install itself in the directory `build` (unless overridden by the `CMAKE_INSTALL_PREFIX` variable). The `build` directory will have the following structure:

- `bin`: contains the command line application (`flann`) and other binary files
- `lib`: contains the compiled libraries (`libflann.so` and `libflann_s.a` for linux)
- `include`: contains the C header files
- `matlab`: contains the MATLAB wrapper functions and a MEX (Matlab EXecutable) file
- `python`: contains the python bindings

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
$ mkdir tmp
$ cd tmp
$ cmake ../src -DCMAKE_BUILD_TYPE=release
$ make install
```

On windows the steps are similar:

```
> cd flann-x.y-src
> md tmp
> cd tmp
> cmake ..\src -G ``NMake Makefiles'' -DCMAKE_BUILD_TYPE=release
> nmake install
```

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

3 Using FLANN

3.1 Using FLANN from MATLAB

The FLANN library can be used from MATLAB through the following wrapper functions: `flann_build_index`, `flann_search` 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 and `flann_free_index` deletes the index and releases the memory it uses.

Note that in the binary distribution of FLANN the MEX file is linked against the shared version of FLANN (`flann.so` or `flann.dll`), so on Linux you must set the LD_LIBRARY_PATH environment variable accordingly prior to starting MATLAB. On Windows is enough to have `flann.dll` in the same directory with the MEX file.

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

3.1.1 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.

Depending on the contents of the `build_params` structure, the function has two different behaviors. If the structure contains a field that specifies the index type to create the function will create an index of that type (using index parameters which also have to be included in the `build_params` structure). If the index and index parameters are not specified directly the function will first try to automatically detect the best index and index parameters to use for nearest neighbor search in the provided dataset.

Using automatic index and parameter configuration When using automatic configuration the `build_params` structure must contain the following fields:

`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.

Specifying the index type and parameters manually Because the parameter estimation step is costly, it is possible to skip this step and reuse the already computed parameters the next time an index is created from similar data points (coming from the same distribution). To specify the index type and the index parameters manually, the `build_params` structure must contain the following fields:

`algorithm` - the algorithm to use for building the index. The possible values are: `'linear'`, `'kdtree'`, `'kmeans'` and `'composite'`. 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 and `'composite'` is a mix of both kdtree and kmeans trees.

`trees` - the number of randomized kd-trees to create. This parameter is required only when the algorithm used is `'kdtree'`.

`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. This parameter is required only when the algorithm used is `'kmeans'`.

`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. This parameter is required only when the algorithm used is `'kmeans'`.

`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.

The above parameters 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 `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.

3.1.2 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);
```

3.1.3 flann_free_index

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

```
flann_free_index(index);
```

3.1.4 Examples

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

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);
```

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);
```

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));
```

3.2 Using FLANN from C/C++

FLANN can be easily used in C/C++ programs through the C bindings provided with the library. To use the C bindings, the library header file `flann.h` (located in the `build/include` directory) must be included. Also the compiler must be told where to look for that file, by adding the `build/include` directory to the compiler include path (all the paths in this document are specified relative to the library main directory). The directory can be added to the compiler include path using the `-I` compiler flag (on most Unix/Linux systems). When linking the C/C++ application the `libflann_s.a` (for linking statically) or the `libflann.so` (for linking dynamically) libraries must be linked in. This is done using the `-l` compiler flag followed by the library name (eg. `-lflann`) and by specifying the library search path (`build/lib`) with the `-L` flag. The entire compile command that must be used will look like this:

```
g++ flann_example.cc -I build/include -L build/lib -o flann_example -lflann
```

The following section describes the C bindings offered by the FLANN library:

flann_build_index()

```
FLANN_INDEX 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 3.1.1)

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

```
struct FLANNParameters {
    flann_algorithm_t algorithm; // the algorithm to use (see constants.h)
    int checks;                  // how many leafs (features) to check in one search
    float cb_index;              // cluster boundary index. Used when searching the kmeans tree
    int trees;                   // number of randomized trees to use (for kdtree)
    int branching;               // branching factor (for kmeans tree)
    int iterations;              // max iterations to perform in one kmeans cluetering (kmeans tree)
    flann_centers_init_t centers_init;   // algorithm used for picking the initial cluster centers for kmea
    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

    flann_log_level_t log_level;             // determines the verbosity of each flann function
    char* log_destination;       // file where the output should go, NULL for the console
    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,
};

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
};
```

The field `log_destination` contains the name of a file where these messages should be generated or NULL for the console.

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.

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.

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`).

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.

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

3.3 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 3.1.1. 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 delete_index(self, **kwargs) :`
This method deletes the current index and all the data associated with it. It should be called to free all the memory used by the index.

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

3.4 Using the `flann` command line application

The FLANN distribution also contains a command line application that can be used to perform nearest-neighbor searches using datasets stored in files. The application can read datasets stored in CSV format, space-separated values or raw binary format.

The command line application takes a command name as the first argument and then the arguments for that command:

```
$ flann
Usage: flann.py [command commans_args]

Comamnds:
      generate_random
      compute_gt
      compute_nn
      autotune
      sample_dataset
      cluster
      run_test

For command specific help type: flann.py <command> -h
```

To see the possible arguments for each command, use `flann help <command>`. For example:

```
$ flann run_test -h
Usage: flann.py [command command_args]

Options:
  -h, --help            show this help message and exit
  -i FILE, --input-file=FILE
                        Name of file with input dataset
  -a ALGORITHM, --algorithm=ALGORITHM
                        The algorithm to use when constructing the index
                        (kdtree, kmeans...)
  -r TREES, --trees=TREES
                        Number of parallel trees to use (where available, for
                        example kdtree)
  -b BRANCHING, --branching=BRANCHING
                        Branching factor (where applicable, for example
                        kmeans) (default: 2)
  -C CENTERS_INIT, --centers-init=CENTERS_INIT
                        How to choose the initial cluster centers for kmeans
                        (random, gonzales) (default: random)
  -M MAX_ITERATIONS, --max-iterations=MAX_ITERATIONS
                        Max iterations to perform for kmeans (default: until
                        convergence)
  -l LOG_LEVEL, --log-level=LOG_LEVEL
                        Log level (none < fatal < error < warning < info)
                        (Default: info)
  -t FILE, --test-file=FILE
                        Name of file with test dataset
  -m FILE, --match-file=FILE
                        File with ground truth matches
  -n NN, --nn=NN        Number of nearest neighbors to search for
  -c CHECKS, --checks=CHECKS
                        Number of times to restart search (in best-bin-first
                        manner)
  -P PRECISION, --precision=PRECISION
                        Run the test until reaching this precision
  -K NUM, --skip-matches=NUM
```

```
Skip the first NUM matches at test phase
```

4 Acknowledgments

Many thanks to Hoyt Koepke for his initial work on 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.