FLANN - Fast Library for Approximate Nearest Neighbors

User Manual

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

May 13, 2008

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 D programming language (D is a C/C++ like language with a cleaner syntax, more features and that compiles to native code achieving the same performnace as C/C++). FLANN can be easily used in many contexts through the C, MATLAB and Python bindings provided with the library. It can also be used as a standalone console application.

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
   int* result = new int[t_rows*nn];
   // initialize the FLANN library
   flann_init();
   // index parameters are stored here
   IndexParameters 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, nn, &p, NULL);

   // ...

   delete dataset;
   delete testset;
   delete result;

   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 = 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 = 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
Info  log - Reading input data from dataset.dat
Info  log - Algorithm: kdtree
Info  log - Building index...
Info  log - Time to build 8 trees for 10000 vectors: 0.57 seconds
Info  log - Reading test data from testset.dat...
Info  log - Searching...
Info  log - Time to search 1000 vectors: 0.07 seconds
Info  log - Wrote the nearest neighbors to nn.dat
```

2 Getting 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:

- `Makefile`: makefile used to compile the application/library.
- `src`: directory containg the source files
- `doc`: directory containg this documentation
- `bin`: directory containg the gdc D compiler and build tools. Since not many system have the gdc compiler installed by default, it is distributed with the library.
- `build`: directory where the compiled application, library, C bindings and matlab wrapper functions are stored. It has the following subdirectories:
  - `bin`: contains the command line application (`flann`)
  - `lib`: contains the compiled library (`libflann.a`)
  - `include`: contains the C bindings for the library (`flann.h`)
  - `matlab`: contains the MATLAB wrapper functions and a MEX (Matlab EXecutable) file
  - `python`: contains the python bindings

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.

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 searching takes longer. The optimum value usually depends on the application.

`build_weight` - specifies the importance of the index build time reported 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.

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'.

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 = 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 a matrix of size $k \times m$ in which each column contains the indexes (in the dataset matrix) of the k nearest neighbors of the corresponding point from testset.

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 = 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 = 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.a` and `libgphobos.a` 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 -lgphobos
```

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

flann_init()

```
void flann_init();
```

This function performs various initializations needed by the library. It must be called before any other function from the library is called.

flann_term()

```
void nn_term();
```

This is the pair of `flann_init()` and runs finalization operations, such as freeing memory used by the library.

flann_build_index()

```
FLANN_INDEX flann_build_index(float* dataset, int rows, int cols,float* speedup,
   IndexParameters* index_params, 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)

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

```
struct IndexParameters {
    int algorithm;
    int checks;
    int trees;
    int branching;
    int iterations;
    int centers_init;
    float target_precision;
    float build_weight;
    float memory_weight;
};
```

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

```
const int LINEAR    = 0;
const int KDTREE    = 1;
const int KMEANS    = 2;
const int COMPOSITE = 3;

const int CENTERS_RANDOM = 0;
const int CENTERS_GONZALES = 1;
const int 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 invital 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 rest of the fields of the `IndexParameters` structure have the same meaning as described in 3.1.1.

flann_params - is a structure containing parameters that influence the behavior of the FLANN library functions. If a NULL value is passed, this argument is ignored.

```
struct FLANNParameters {
    int log_level;
    char* log_destination;
    long random_seed;
};
```

random_seed - contains the random seed to use to initialize the random number generator.

log_level - specifies the amount of messages generated by the FLANN library functions It can take the following values:

```
const int LOG_NONE  = 0;
const int LOG_FATAL = 1;
const int LOG_ERROR = 2;
const int LOG_WARN  = 3;
const int LOG_INFO  = 4;
```

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 tcount, int* result, int nn, int checks, 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 `tcount` 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 `result` matrix (which is a tcount × 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 `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 count, int length,
   float* testset, int tcount, int* result, int nn,
   IndexParameters* index_params, 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_free_index()

```
void flann_free_index(FLANN_INDEX index_id, 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 and scipy.weave packages 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 = 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 [command command_options]

Commands:
        autotune_params
        sample
        help
        compute_clusters
        compute_nn
        generate_random
        run_test
        compute_gt
        convert

For more info type: flann help [command]
```

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

```
$ flann help compute_nn
Command: compute_nn [options]
Builds an index from a dataset and searches that index for the nearest
neighbors of all the features in a testset.

Options:
  --verbosity, -v             The program verbosity (trace > info > warn > error
> fatal > none) (Default: info)
  --reporters, -e             Comma-delimited list of reporters to use.
  --help, -h                  Display help message
  --input-file, -i            Name of file with input dataset.
  --params-file, -p           File containing 'optimum' input dataset
parameters.
  --algorithm, -a             The algorithm to use when constructing the index
(kdtree, kmeans...).
  --trees, -r                 Number of parallel trees to use (where available,
for example kdtree).
  --branching, -b             Branching factor (where applicable, for example
kmeans) (default: 2).
  --byte-features, -B         Use byte-sized feature elements.
  --centers-init, -C          Hot to choose the initial cluster centers for
kmeans (random, gonzales) (default: random).
  --max-iterations, -M        Max iterations to perform for kmeans (default:
until convergence).
  --test-file, -t             Name of file with test dataset.
  --output-file, -o           Output file to save the features to.
  --nn, -n                    Number of nearest neighbors to search for.
  --checks, -c                Number of times to restart search (in
best-bin-first manner).
  --skip-matches, -K          Skip the first NUM matches at test phase.
```

4 Acknowledgments

Many thanks to Hoyt Koepke for his work on the python bindings.

References

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *SODA '07: Proceedings of the eighteenth an-*

nual ACM-SIAM symposium on Discrete algorithms, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.