Vector Similarity Search & Clustering

George Sittas, Dimitra Kousta

October-November 2021

1 Foreword

In this project we demonstrate two different methods for solving the approximate k-NN problem: Locality-Sensitive Hashing (LSH) and Randomized Projection on Hypercubes. We then implement three variants of vector clusterers with the K-Means++ initialization algorithm, using the above methods and the standard Lloyd's algorithm for the expectation (assignment) step. Lastly, we conduct a Silhouette analysis on the results of these variants to evaluate their overall effectiveness.

2 Structure

The following diagram represents the project's directory organization:

```
assignment_1
├── bin - executables created by Makefile
├── config - configuration files
├── input - dataset and query files
├── modules - reusable code
│   ├── clusterers - clusterer interface & its implementations
│   ├── common_utils - tools that are used throughout the project
│   ├── nn_solvers - k-NN problem solver interface & its implementations
│   ├── parsers - code related to parsing
│   └── types - representations of entities used in the project
└── src - demonstrations of all implemented algorithms
    ├── cluster - vector clustering and its evaluation
    ├── cube - Hypercube approach for approximately solving the k-NN problem
    └── lsh - LSH approach for approximately solving the k-NN problem
```

3 Compilation

The following commands are available in a shell environment for building the project:

```
make lsh     # --> approximate k-Nearest Neighbor solver (LSH)
make cube    # --> approximate k-Nearest Neighbor solver (Hypercube)
make cluster # --> vector clusterer / clustering evaluator
make clean   # --> remove all executables & object files
```

4 Usage

4.1 Execution

After the executables are created, they can be used as follows:

```
./bin/lsh     OPT_ARGS_LSH
./bin/cube    OPT_ARGS_CUBE
./bin/cluster ARGS_CLUSTER

OPT_ARGS_LSH:

    -i <input file>  : set input file
    -q <query file>  : set query file
    -o <output file> : set output file
    -k <int>         : set number of LSH hash functions
    -L <int>         : set number of hash tables
    -N <int>         : set number of neighbors to be searched
    -R <real>        : set search radius

OPT_ARGS_CUBE:

    -i <input file>  : set input file
    -q <query file>  : set query file
    -o <output file> : set output file
    -k <int>         : set hypercube's dimension
    -M <int>         : set max number of points to be checked
    -N <int>         : set number of neighbors to be searched
    -R <real>        : set search radius
    -probes <int>    : set max number of vertices to be checked

ARGS_CLUSTER:

    -i <input file>             : set input file
    -c <configuration file>     : set configuration file
    -o <output file>            : set output file
    -m (Classic|LSH|Hypercube) : set clustering method
    -complete                   : show final clustering (optional)
```

4.2 File Formats

Datasets and query files provided to the above executables ("input_file" and "query_file") need to follow the format shown below:

```
item_id1    X11   X12   ...   X1d
.     .     .     ...   .
item_idN    XN1   XN2   ...   XNd
```

Where each item_id is unique, can either be an int or a string, and Xij is a real coordinate of a d-dimension vector in the Euclidean space. The input tokens are separated by some kind of delimiter, the default one being a single space (the delimiter can be changed in main.cc files under src). Also, the configuration files ("config_file") used by the clusterer need to follow the format shown below:

```
number_of_clusters: <int>                 // K of K-medians
number_of_vector_hash_tables: <int>       // default: L=3
number_of_vector_hash_functions: <int>    // k of LSH for vectors, default: 4
max_number_M_hypercube: <int>             // M of Hypercube, default: 10
number_of_hypercube_dimensions: <int>     // k of Hypercube, default: 3
number_of_probes: <int>                   // probes of Hypercube, default: 2
```

Here, lines that include a default value for the corresponding option can be omitted completely (hence, the only mandatory line is the one specifying the parameter K in K-means).

5 Evaluation

In this section, we shall present and explain the metrics that were used to evaluate the efficiency and accuracy of both k-NN solvers, as well as demonstrate some statistic results acquired during the process of parameter tuning for both the k-Nearest Neighbors and the Clustering problems.

5.1 k-Nearest Neighbors

The following metrics were used, in order to compare the two approximate methods to the brute-force algorithm used for solving the k-NN problem:

```
- avg_bf_duration : how long it took, on average, for the brute
                    force algorithm to compute the k neighbors.

- avg_(lsh|cube)_duration : same, but for each approximate method.

- avg_duration_rate : rate that depicts the relative speed of
                      each method, compared to the brute force
                      algorithm, measured (per query) as:

            average((lsh|cube)_duration / bf_duration)
```

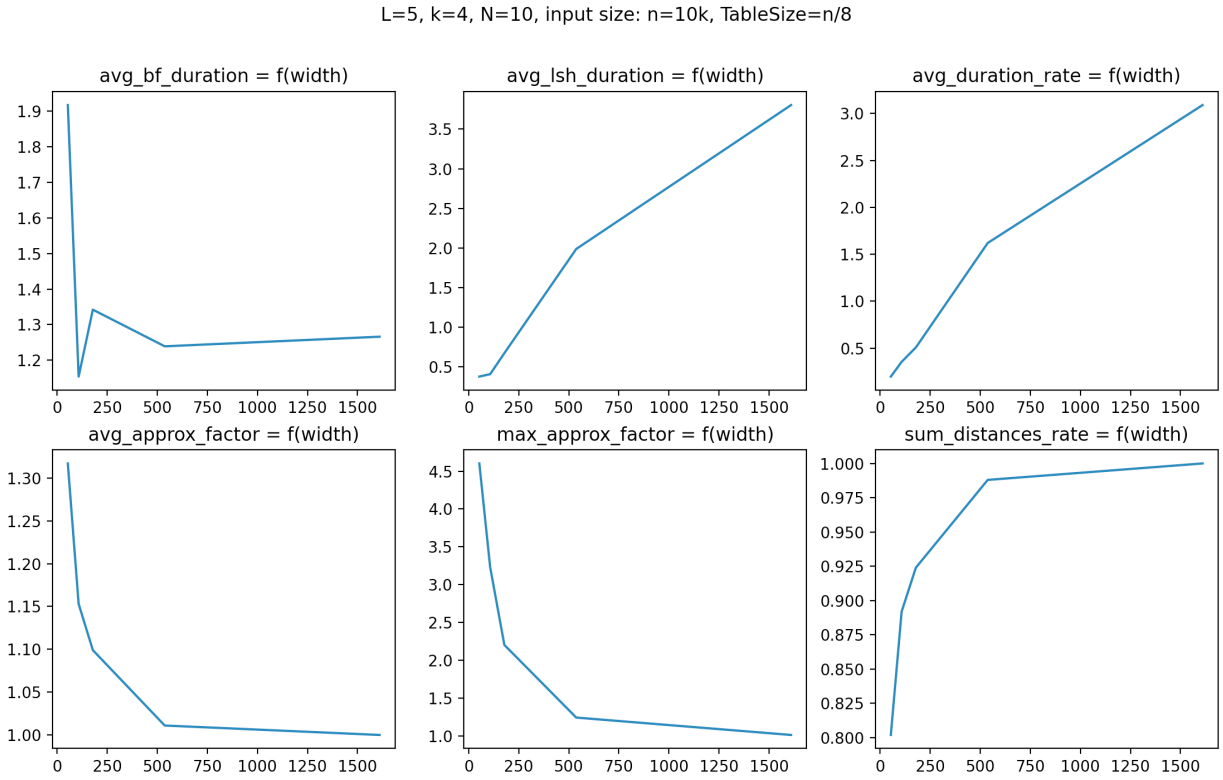
```
- avg_approx_factor : rate that depicts the relative accuracy of
                      each method, compared to the brute force
                      algorithm, measured (per query) as:

            average((lsh|cube)_distance / bf_distance)

- max_approx_factor : same, but we measure the maximum such rate.

- sum_distances_rate : similar metric to avg_approx_factor, but
                       here we measure the following:

           Sum (bf_distance) / Sum ((lsh|cube)_distance)
```

Note that, in the last three metrics, "distance" refers to the distance between the query and the 1-nearest neighbor computed by each method (approximate or brute-force). If we have values below 1 for the average duration rate, it means that the approximate methods are faster, compared to the brute-force approach. Similarly, as values get closer to 1 (but bigger than it) for the average/max approximation factors, it means that the approximate methods get progressively more accurate, with the perfect accuracy reached when the rates are equal to 1. The final metric simply measures how close (percentage) the approximate methods are to reaching the perfect accuracy. Besides these metrics, we also count the number of neighbors missed by each of the approximation methods. The data set used in all cases mentioned below is the one corresponding to the 10000 entries.

In order to find a value for the denominator involved in the computation of a hash function in the LSH family, we extract a random sample of 1000 data points uniformly from the data set and calculate the average distance between two points in that sample. Then, we calculate the denominator as a function of that average distance.

Empirically, we've found that as the value of the denominator grows, so does the running time of the LSH method, but we get better precision (see the graphs in the following page – note that the upper left one is not supposed to give a correlation between the average brute force algorithm's duration and the width, since the brute force algorithm doesn't use it for its calculations). This is to be expected, since more data points are hashed into the same buckets, and thus we end up comparing the query point to more points in the data set. The Hypercube method isn't as sensitive to the value of the denominator as the LSH method is, and this is also to be expected, since the determining factors here are the total number of points to be compared to the query point and the number of probes to be done. For this reason, we've also found that the Hypercube method misses neighbors more frequently, as opposed to the LSH method, which constantly returns most of the requested number of neighbors, if not all of them.

We have also noticed that the LSH method works best with around 5 to 10 LSH hash functions, as well as 5 to 8 hash tables. If we try to go past these ranges, the good effects start to diminish and we either get a computational overhead, resulting in both bigger delays and a larger memory consumption, or a loss in accuracy. Regarding the table size, we've noticed that it slightly affects the method, and it's frequently better to have it be equal to 1/8-th of the data set's size.



5.2 Clustering

We compared the three clustering methods, and realized (empirically) that the approximate methods are not much faster, compared to the Lloyd's algorithm. Also, we have noticed that the number of clusters that produces the greatest Silhouette scores is around 4 to 6, as values outside this range start giving negative scores. The following statistics were computed for: 4-cluster clustering, width = 107, input size: n = 10k, hash table size = n/8, default parameters for the LSH method, and 1000 points, 5 probes, 10 vertices for the Hypercube method:

```
Lloyd's    : average Silhouette score = 0.192
LSH        : average Silhouette score = 0.036
Hypercube  : average Silhouette score = 0.064
```

And the corresponding running times are:

Method/Times	Classic	LSH	Hypercube
Clustering vectors	0.228 sec	0.123 sec	0.101 sec
Computing Silhouette	7.649 sec	7.602 sec	6.850 sec

For both Reverse Assignment methods and the Lloyd's method, the clustering is assumed to converge to a fix-point (and is thus terminated) if the clusters remain unchanged for one whole iteration. Also, in the Reverse Assignment methods, we stop the range search if we find that there are no new assignments after an iteration, and then we assign all remaining points using the Lloyd's method. Another criterion for clustering convergence could be: comparing the new centroid positions to their old ones, and if they are epsilon-distant apart, we can say that the clustering has converged. We tried this, and noticed it gives better clustering results for small distance thresholds, but can be slower than the other criterion, mentioned at the start of this paragraph. This is reasonable, since letting the centroids "move around" until they've actually converged to a location provides greater stability, and also needs more iterations to happen. But, there could be edge cases where the centroids would be moving back and forth, thus being unable to converge to a location. In such cases, a trigger could be used to stop the algorithm after being stuck for an amount of iterations.

6 Conclusion

After having implemented the proposed methods, it's become apparent that parameter tuning is a difficult task and can vary heavily, depending on the data set at hand. Moreover, we have concluded that these methods offer a great alternative for tackling difficult problems, such as those of clustering and similarity search in high dimensional vector spaces, that would otherwise be unsolvable.

7 References

- Priority Queue in C++
- How to Measure C++ Time Intervals
- Generate random numbers uniformly over an entire range
- How to create random Gaussian vector in c++?
- Locality-sensitive hashing
- How to pick item by its probability