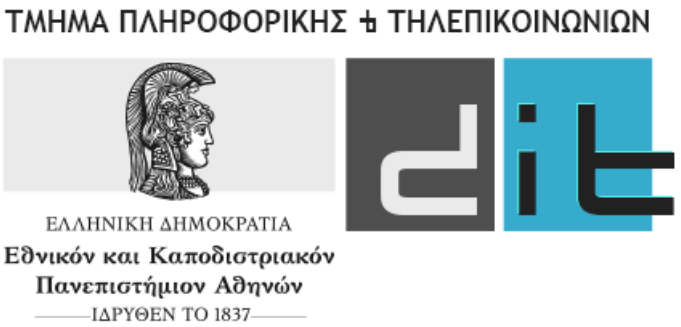

National and Kapodistrian University of Athens

Department of Informatics and Telecommunications

Software Development for Algorithmic Problems

Similarity Search and Clustering for multi-dimensional vectors

---

Project implemented by :

Nikolaos Galanis - sdi1700019
Sofoklis Strobolas - sdi1700153

Contents

1 Abstract

The goal of this project is to create 2 different programs:

- A program that predicts the nearest neighbour of a vector.
- A programs that performs Clustering given a dataset.

The nearest neighbor problem is solved using an approximate approach, as well as a brute force one. In the approximate, we are trying to predict the NN, by applying 2 different algorithms: **Locality Sensitive Hashing** and **Random Projections to the Hypercube**. We compare our timing results and our predictions with the BF model, in order to find out how well do our algorithms perform. In order to test the program, we use the MNIST dataset, that contains pictures as vectors, of size 28×28

The clustering problem is solved using a variety of algorithms. During the initialization stage we implement **k-means++**, during the update step we implement **k-median**, and during the assignment we implement direct assignment using the **Lloyd's algorithm**, as well as reverse assignment using the **LSH** and **Hypercube** Range Search that we implemented earlier.

The project was implemented using the C++ language, by taking advantage of the STL library.

2 Directories Structure and Files

```
/
├── algorithms
│   ├── Clustering ..... Clustering Implementation
│   ├── common ..... various metrics
│   └── Search
│       ├── BruteForce ... BF Implementation, to check accuracy of LSH and HC
│       ├── Hypercube ..... Hypercube Implementation
│       └── LSH ..... LSH Implementation
├── build ..... Contains the objective files, created with make
├── config ..... cluster config file
├── executables ..... Contains the exec files, created with make
│   ├── clustering
│   ├── hypercube
│   └── lsh
├── misc ..... Location of the input files
│   ├── datasets
│   └── querysets
├── programs ..... Location of the main fns of each program
│   ├── Clustering
│   ├── common
│   ├── Hypercube
│   └── LSH
├── utils ..... Utility functions
└── results ..... Execution results
```

We have organised our project as stated in the diagram above. In the programs directory are our main files for each method, and when compiled the objective files are being placed in the build directory, and the executables in the executables one. In the algorithms directory we have placed our implementation for the LSH, Hypercube, Clustering and Brute force, in the corresponding sub-directories.

Our Makefile is located at the root directory, alongside other .mk files, that contain akefile settings.

3 Compiling

For the compilation of all the programs we can use the Makefile. When being in the root directory of our project, in order to compile all our programs at once we can just call:

- **make** or **make all**

The **lsh**, **cube** and **cluster** executables will then be placed in the **executables** directory in their corresponding sub-directory and the **.o** files will be placed in the **build** directory.

We can delete these directories by using **make clean**.

In order to compile a program of our choice we have these options:

- LSH: **make lsh**
- Hypercube: **make cube**
- Clustering: **make cluster**

In order to delete the directories of a specific program we can use **clean_lsh** , **clean_cube**, or **clean_cluster** respectively.

We also use -Ofast argument for the gcc to optimize the performance of our programs.-Ofast - Disregard strict standards compliance. -Ofast enables all -O3 optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on -ffast-math and the Fortran-specific -fno-protect-parens and -fstack-arrays. This addition gave us significant improvement in the the performance of our programs.

4 Executing

For the execution of programs **lsh**, **cube** or **cluster** we can use the commands bellow. :

- **make run_lsh**: Running the lsh with default arguments.
- **make run_cube**:Running the cube with default arguments.
- **make run_cluster_lloyds**: Running the cluster with lloyds.
- **make run_cluster_lloyds_complete**: Running the cluster with lloyds and complete.
- **make run_cluster_lsh**: Running the cluster with lsh.

- **make run_cluster_lsh_complete**: Running the cluster with lsh and complete.
- **make run_cluster_cube**: Running the cluster with hypercube.
- **make run_cluster_cube_complete**: Running the cluster with hypercube and complete.

We use the default arguments given to us:

- lsh: k=4, L=4, N=1, R=10000
- hypercube: k=14, M=10, probes=2, N=1, R=10000
- cluster: We use -m Lloyds or -m Range_Search_LSH or -m Range_Search_Hypercube for each method. Additionally, for the complete run we add -complete to the arguments.

We also experimented with the arguments in order to get the best time-accuracy trade-off.

We can also test the memory management by executing them using valgrind:

- **make valgrind_lsh**
- **make valgrind_cube**
- **make valgrind_cluster_lloyds**
- **make valgrind_cluster_lloyds_complete**
- **make valgrind_cluster_lsh**
- **make valgrind_cluster_lsh_complete**
- **make valgrind_cluster_cube**
- **make valgrind_cluster_cube_complete**

After the execution of each program the output file is being created and its placed next to the executable.

5 Implementation

We are going to briefly analyze our implementation for each program. A more detailed explanation of the code is provided through numerous comments on each function.

5.1 Hash Functions

Two classed are defined in the file "hashing.h", one for the Hash function h, and one for the amplified hash function g.

The HashFunction class has a constructor in order to initialize all the s vectors (distributed uniformly). There is also a $Hash()$ function, that hashes a vector in the margins given.

The AmplifiedHashFunction class has a constructor in order to initialize the k HashFunction classed. There is also a $Hash()$ function, that hashes a vector in the margins given.

5.2 LSH

All of the files regarding the LSH core are located in the file "lsh.h". The file contains the class LSH, which has as private members all the obligatory parameters for the initialization of the LSH, as well as a huge vector containing the images of the dataset.

While constructing the LSH tables, we use the constructor to call the appropriate constructor of the amplified hash function for each table, and then we hash all the feature vectors in the L tables.

The class offers 3 searching functions:

- **NearestNeighbour**: Finds the approximate nearest neighbour of the given vector, using the LSH algorithm
- **kNearestNeighbour**: Returns a sorted list of the n nearest neighbours and their distances to the vector
- **RangeSearch**: Returns all the neigbours of a vector that have a specific radius form the query.

5.3 Hypercube

All of the files regarding the Hypercube core are located in the file "hypercube.h". The file contains the class Hypercube, which has as private members all the obligatory parameters for the initialization of the HC, as well as a huge vector containing the images of the dataset.

While constructing the Hypercube table, we use the constructor to call the appropriate constructor of the hash functions, and also initialize the other fields of a class.

We have a *hash_to_hypercube*() function, that applies the methods we have learned in theory in order to hash a vector into the appropriate edge of the hypercube.

We also have the 3 search functions that we mentioned in the LSH implementation, but this time applied in the Hypercube algorithm.

5.4 Clustering

All of the files regarding the Clustering core are located in the file "Clustering.h". The file contains the class Clustering, which has as private members all the obligatory parameters for the initialization of the clustering algorithm, a huge vector containing the images of the dataset, and various data structures in order to access all the needed data in a handy way.

We also have various private functions in order to access those data structures, like the nearest centroid of a vector.

We provide 3 different constructors of the class, using function overloading, in order to initialize differently the class given the choice of assignment method by the user. The different choices of assignment methods are:

- Direct assignment using the Lloyd's algorithm
- Reverse assingment using LSH algorithm
- Reverse assingment using random projections to the Hypercube

Then, we implement the 3 steps of the Clustering algorithm:

- **Initialization**, using the kmeans++ algorithm
- **Assignment**, using the method chosen by the user
- **Update**, by each time computing the median of all the vectors assigned to a centroid.

We have also implemented the **Silhouette** metric, in order to evaluate the quality of our clustering. It returns the silhouette of each cluster, and finally the total silhouette of the program.

Finally, there is a function for running the clustering algorithm, that runs the above steps, until a very small amount of changes during assingment is detected.

5.5 Utility Functions

We have implemented various utils in order to help us with the building of the program. All of them are in the utils directory, as well as the metrics directory.

In the **metrics** directory, there are 2 different metrics: the Manhattan, which is used by our programs, and the Euclidean.

In the **utils** directory, there are print and math utils. The print utils help us visualize various data structures, and the math to implement complex math functions that are needed throughout our functions.

6 Input

6.1 Search

For both of the search algorithms(LSH and Hypercube) the program takes as an input the MNIST dataset, and the MNIST queryset. The dataset contains 60.000 images of size 28×28, and the queryset contains 10.000 similar images. The images are a set of bytes(range 0 to 255), thus one image, is a vector of size 28×28, containing integer values from 0 to 255. For our convenience, and because of the idiomorphies of the MNIST dataset, we store the data as double, and not as integers. The dataset and the queryset look like the above set:

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  60000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel
```

6.2 Clustering

For the clustering algorithm, the input is once again the MNIST dataset. We are going to use those 60.000 vectors in order to assign them to a cluster. Also, we take as an input a configuration file, of the following shape:

```
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
```

7 Output

Our programs produce both an output file, and write results to the stdout stream.

7.1 Search

For the search algorithms, we produce an output file, with the following format:

```
Query: image_number_in_query_set
Nearest neighbor-1: image_number_in_data_set
distanceLSH: <double> [ή distanceHypercube αντίστοιχα]
distanceTrue: <double>
...
Nearest neighbor-N: image_number_in_data_set
distanceLSH: <double> [ή distanceHypercube αντίστοιχα]
distanceTrue: <double>
tLSH: <double>
tTrue: <double>
R-near neighbors:
image_number_A
image_number_B
. . .
image_number_Z
```

When the program is run, it produces the following output in the stdout:

```
LSH initialization with parameters: L = 4, m = 4294967291, M = 256, n_points = 60000 k = 4, w = 12346, space_dim = 784
Initialization completed in 9 seconds
mean lsh time 0.0274191
mean bf time 0.0875661
corectly computed neighbours 902 out of 10000
```

7.2 Clustering

The output of the clustering algorithm is of the following format:

```
Algorithm: Lloyds OR Range Search LSH OR Range Search Hypercube
CLUSTER-1 {size: <int>, centroid: πίνακας με τις συντεταγμένες του centroid}
.     .     .     .     .     .     .
CLUSTER-K {size: <int>, centroid: πίνακας με τις συντεταγμένες του centroid}
clustering_time: <double> //in seconds
Silhouette: [s1,...,si,...,sK, stotal]
/* si=average s(p) of points in cluster i, stotal=average s(p) of points in
dataset */

/* Optionally with command line parameter -complete */
CLUSTER-1 {centroid, image_numberA, ..., image_numberX}
.     .     .     .     .     .     .

CLUSTER-K {centroid, image_numberR, ..., image_numberZ}
```

8 Results

In this section we present the results that our programs have in the standard output. For each program we tested with the argument $-Ofast$ for a better speed performance. Additionally, we tested both 1000 queries and 10000 queries.

8.1 LSH

- 10000 queries

First attempt

```
Building lsh...
lsh ready
././executables/lsh/lsh -d misc/datasets/train-images-idx3-ubyte  -q misc/querysets/t10k-images-idx3-ubyte -k 4 -L 4 -o executables/lsh/lsh_out -N 1 -R 10000
LSH initialization with parameters: L = 4, m = 4294967291, M = 256, n_points = 60000 k = 4, w = 12346, space_dim = 784
Initialization completed in 8 seconds
mean lsh time 0.262565
mean bf time 1.20416
corectly computed neighbours 8698 out of 10000
```

Second attempt

```
././executables/lsh/lsh -d misc/datasets/train-images-idx3-ubyte  -q misc/querysets/t10k-images-idx3-ubyte
 -k 4 -L 4 -o executables/lsh/lsh_out -N 1 -R 10000
LSH initialization with parameters: L = 4, m = 4294967291, M = 256, n_points = 60000 k = 4, w = 12346, spa
ce_dim = 784
Initialization completed in 9 seconds
mean lsh time 0.0983048
mean bf time 1.3373
corectly computed neighbours 7277 out of 10000
```

1000 queries

```
Building lsh...
lsh ready
././executables/lsh/lsh -d misc/datasets/train-images-idx3-ubyte  -q misc/querysets/t10k-images-idx3-ubyte -k 4
-L 4 -o executables/lsh/lsh_outq1000 -N 1 -R 10000
LSH initialization with parameters: L = 4, m = 4294967291, M = 256, n_points = 60000 k = 4, w = 12346, space_dim
 = 784
Initialization completed in 15 seconds
mean lsh time 0.0331527
mean bf time 0.183246
corectly computed neighbours 895 out of 1000
```

8.2 Hypercube

- 1000 queries

```
Hypercube initialization with parameters: k = 13, m = 4294967291, threshold = 3000 max_probes = 100, npoints = 60000, w = 246920, spacedim = 784
Initialization completed in 14 seconds
mean hc time 0.0112273
mean bf time 0.174348
corectly computed neighbours 237 out of 1000
```

-

- 10000 queries

```
Hypercube initialization with parameters: k = 13, m = 4294967291, threshold = 1000 max_probes = 50, npoints = 60000, w = 246920, spacedim = 784
Initialization completed in 13 seconds
mean hc time 0.00690036
mean bf time 0.174981
corectly computed neighbours 737 out of 10000
```

8.3 Clustering

All the runs were made for 10 clusters, and the other arguments with their default values

8.3.1 Lloyd's assignment

Running with all the vectors: Clustering time: 383.486 secs

Running with 1000 vectors: Clustering time: 7.01293 secs
Silhouette: [0.0920564, 0.259612, 0.473002, 0.558257, 0.341821, 0.429571, 0.544843, 0.347369, 0.212882, -0.395158]
Total Silhouette: 0.181951

8.3.2 LSH reverse assignment

Running with all the vectors: Clustering time: 331.486 secs

Running with 1000 vectors: Clustering time: 5.81206 secs
Silhouette: [0.464598, 0.207147, -0.0251963, 0.396038, 0.237209, 0.337994, 0.493729, -0.338102, -0.0393306, 0.409013]
Total Silhouette: 0.104314

8.3.3 Hypercube reverse assignment

Running with all the vectors: Clustering time: 302.536 secs

Running with 1000 vectors: Clustering time: 4.67603 secs
Silhouette: [0.544338, 0.175981, 0.433266, -0.393804, 0.0193938, -0.0509101, 0.301775, 0.413968, 0.535144, 0.22142]
Total Silhouette: 0.101317

8.4 Complete results

All the results of the various executions of our programs are located in the directory results.

9 Conclusion

During this project, the effect of approximation in big data was made clear to us. With the observation of the time differences, we found out that while brute force algorithms take a significant amount of time to complete, the approximating algorithms predict the results with almost the same accuracy.

In the clustering matter, we learned the algorithms that were used, and the advantages of each one when it comes down to accuracy and successfully predicting the correct cluster for each vector.