ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

SOFTWARE DEVELOPMENT FOR ALGORITHMIC PROBLEMS. 7TH SEMESTER 2020, ASSIGNMENT 1: SEARCH AND CLUSTERING VECTORS

National and Kapodistrian University of Athens, DiT. 1/11/2020

Dimitrios Foteinos | sdi1700181
Evgenia Kalyva | sdi1600256

Contents

1 Abstract

This assignment is the 1st part of the set. The subject of this particular homework , belongs purely in the field of Data Science. In this assignment, we were asked to implement some interesting methods for **searching** and **clustering** $d-$dimension vectors.

For this project, we were confidential about the tools that we may have needed. Ofcourse, this project is implemented in **C++** , because we needed a lot of the structures, utilities and the generic nature of this specific language. Thus, our design was made up in order to have any type of vectors, (such as **int** or **double**). The metrics for the distance that we were asked to choose is the famous Manhattan Distance and Hamming Distance. To be more detailed, this project was splited up in two unique parts: A) **Searching** and B) **Clustering**. For each part, we have the following:

A. **Locality Sensitive Hashing and Randomized Projections.**

The algorithms that we implemented for Searching, were the famous LSH and HyperCube.

For LSH, we used as distance function the Manhttan Distance. The main idea is that we have some **training data** and some **test data**. First, we choose the number of the Hashtables we want to use. Then, for each Hashtable, we choose from some specific formulas the number of buckets that we want to use. With these being said, we must declare that in this program, there is a hyperparameter: **w**. Stuff can be really tricky about the choosing of w, because it can be shown that this parameter can decide crucial things about our program, such as **Accuracy Complexity** and **Time**. As we expected, if we want to increase the accuracy , then the time will be increased too. So, how can we choose the best **w**, in order to achieve the best trade-off between Accuracy and Time?

There is not only one way to answer this question, because some may want to achieve the best accuracy regardless of the time (in this case the computational power may be higher than usual) , and some may want to achieve the best possible accuracy with the minimum possible time. Let's assume that we agree with the second case, how can we achieve the best accuracy with the minimum time? In order to accomplish this behaviour, we created a cost function:

$$C(t, d) = t * d$$

Given inputs are: t= the ratio of the average time of LSH and the average time of Bruteforce

$$t = \frac{mean_{LSH}}{mean_{TRUE}}$$

and d = the absolute value of the difference between the average distance of LSH and the average distance of Bruteforce,

$$d = |LSH - TRUE|$$

So , the lower the cost function, the best for us. We did some "grid search" in our data. We used whole the training set (60.000 points) and we took a tiny sample of the test data (100 points), also we looked for 3-Nearest Neighbours. For **LSH** we have the following:

Cost Value/W	**52**	**100**	**250**	**500**	**1000**	**4.000**	**20.000**	**40.000**
LSH	61.0441	63.7806	62.7264	65.6498	57.0376	56.8202	**19.5274**	37.4881

As we can see, the best possible w is 20.000 after our research. We must declare that, if we set w up to 20.000, we may increase the time, but in this value we have the best possible accuracy. So, if someone wants to increase the time of the algorithm but still achieve some satisfying results, it's up to him to play with the hyperparameter and see which one fits perfect for him.

For the Hypercube, we also use a training data set and an input data set. This time, every image will be considered a point in a d'(k) dimensional Hypercube, and for every input image we will search for the approximate Nearest Neighbor within the points closer to the one the image belongs to, according to the calculated Hamming Distance between all points. In our implementation, the Hypercube is essentially a map, with the keys being the points(strings of 0's and 1's of d' length) and the values being the image/images that belong to each point. To calculate the nearest neighbors, we perform a search in a given number of points(probes) and a given number of images(M) in total, with the Manhattan distance, withing a R-range radius.

2 File Contents

main.cpp : the main function , that initializes our program according to given arguments

arguments.cpp : implementations of functions handling the arguments given from the command line

arguments.h : the header file for arguments.cpp

util.cpp : implementations of utility functions that initiate LSH, Hypercube or Clustering

util.h : the header file for util.cpp

hash-functions.cpp : implementations of all the needed hash functions, g for LSH, f for Hypercube and h for both

hash-functions.h : the header file for hash-functions.cpp

hash-tables.cpp : implementations of our basic generic structures and methods for LSH and Hypercube

cube-functions.cpp : implementations of specific methods for Hypercube

lsh-functions.cpp : implementations of specific methods for LSH

hash-tables.h : the header file for hash-tables.cpp, cube-functions.cpp and lsh-functions.cpp

cluster-functions.cpp : implementations for clustering

Makefile : makefile for our program

cluster.conf : basic parameters for clustering

3 Compile and Run

In this assignment, we have in total 2 main concepts as we said before: 1) **Searching** and **Clustering**. We can break down these 2 concepts into more complex ones:

Searching

- **LSH:**

1.1 Just type **make** , and then simply to give all the parameters : $./lsh –d <input file> –q <query file> –k <int> -L <int> -o <output file> -N <number of nearest> -R <radius>

1.2 Just type **make** , and then simply(if you want our program to take as input default parametrs) : $./lsh –d <input file> –q <query file> -o <output file>

1.3 Just type **make** , if you want to run lsh with gdb $ **make** gdblsh

1.4 Just type **make** , and then simply(if you want to run lsh with valgrind): $ **make** vallsh

– **Hypercube:**

2.1 Just type **make** , and then simply to give all the parameters : $ ./cube –d <input file> –q <query file> –k <int> -M <int> -probes <int> -o <output file> -N <number of nearest> -R <radius>

2.2 Just type **make** , and then simply(if you want our program to take as input default parameters) : $./cube –d <input file> –q <query file> -o <output file>

2.3 Just type **make** , if you want to run Hypercube with gdb $ **make** gdbcube

2.4 Just type **make** , and then simply(if you want to run Hypercube with valgrind): $ **make** valcube

– **Clustering:**

In this point we must declare that, we have a configuration file that help us give the clustering programs arguments.

3.1 Just type **make** , and then simply to give all the parameters : $./cluster –i <input file> –c <configuration file> -o <output file> -complete <yes OR no> -m <method: Classic OR LSH or HyperCube>

3.2 Just type **make** , if you want to run Clustering with gdb $ **make** gdbclu

3.3 Just type **make** , and then simply(if you want to run Clustering with valgrind): $ **make** valclu