Project 1: *ANN and Clustering in MNIST digit Dataset*

Team Members:

1. Dimitrios Konstantinidis (sdi1700065@di.uoa.gr)
2. Andreas Spanopoulos (sdi1700146@di.uoa.gr)

A little bit about our work

In this Project we implement algorithms and data structures that can be used to solve the following 2 problems on any given Dataset:

1. Find *N* Approximate Nearest Neighbors (ANN) for any query point
2. Group the data points together by performing Clustering

Thus, the Project is split in two parts, the first one solves the (ANN) problem and the second one solves the Clustering problem. An experimental Analysis for each part of the project can be found in the directory Analysis.

A little bit about our Dataset

The Datasets used to test the correctness of our algorithms is the *MNIST Database of handwritten digits*, and be found here. This database contains images of size 28 x 28 pixels, that is, 784 pixels if we flatten the image (spoiler alert: we do). The Training set consists of 60000 images, and the test set of 10000. Note that the images are stored in Big-Endian format. Our code makes sure that they are converted to Little-Endian.

Approximate Nearest Neighbors (ANN)

Approximate Nearest Neighbors (or ANN) is a problem in which we want to search for similar items, without enduring the huge cost of a brute force algorithm. The program implements two data structures which drastically reduce time complexity, while also retaining respectable accuracy. The data structures are the following: - Locality Sensitive Hashing: Hash table which maps nearby items to the same bucket - Randomized Projection Hypercube: Hypercube which maps similar items to the same (or nearby) vertices

Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing is an algorithmic technique that hashes similar input items into the same buckets with high probability. The layers in which this algorithm is implemented are the following: 1. HashFunction: This is our baseline hash function. It takes in as an input an item (image), shifts each feature (pixel) by a small amount and then divides it by the given window size. The "shifting" is constant for each for each item given, however it differs among other "sibling" functions. After this shifting is complete, it is time to calculate the following formula:
$h(p)=a_{d-1}+ma_{d-2}+...+m^{d-1}a_0$ 2. AmplifiedHashFunction: This class is called an amplified hash function as it "combines" k HashFunction instances to form a new (amplified) hash function.
Specifically, on creation, k HashFunction objects are initialized. When an item needs to be hashed, it gets hashed by all k of those hash functions, in the end, the results get concatenated bitwise to produce the "final" hash. 3. LSH: This is the part where "everyting makes sense". It contains L hash tables (so L AmplifieadHashFunctions) and whenever an item is inserted, it is added for each hash table into its respective bucket.
To summarize, when LSH is initialized, we have L hash tables where each one contains one instance of each of our items. Whenever a query item arrives, we hash it for every hash table and search for its nearest neighbors among the L buckets (one from each hash table).

Projection in Hypercube

Randomized projection into Hypercube is a similar algorithmic technique to LSH. However, instead of L hash tables with their own AmplifiedHashFunction, our dataset is stored into the vertices of a Hypercube. Layers: 1. HashFunction: This is the same class implemented in LSH, whenever an item comes as input, it is shifted and then the "h" formula is calculated on it. 2. FFunc: This class has a relatively simple work to do. All it has to do is map the output from HashFunction to {0,1}. To ensure that the values are distributed uniformly, a random integer among the range of HashFunction values is selected. Once selected, whenever a HashFunction value is inserted, if the random value is greater than the "h" value, it returns 0. Else, it returns 1. 3. Hypercube: This is the final layer. A hypercube of dimension **d'** is created alongside d' HashFunctions and FFuncs. This hypercube is represented as an array of size $2^{d'}$. Whenever an item is inserted, it is first hashed by each HashFunction and then mapped to {0,1} by FFunc. The result is a binary number of length d' (which is equivalent to a decimal number i.e. the index of the vertex in which the item will be inserted). The process is similar whenever a query item arrives, as it is hashed, then mapped. Once mapped, we have a starting vertex from which we want to start our search for near neighbors. However, we are not limited to only this vertex, it is possible to traverse nearby hypercube vertices (e.g. vertices with hamming distance of 1 from the starting vertex).

Clustering

In this part of the Project we implement the K-medians algorithms via 3 different ways: - Lloyds Algorithm - Reverse Assignment using Range Search LSH - Reverse Assignment using Range Search Hypercube

The centroids are initialized using initialization++ in every algorithm.

Every algorithm consists of the same 2 steps: 1. Assignment (Expectation): Assign each object to its nearest center. 2. Update (Maximization): Calculate Median of each dimension per cluster and make it new center.

Note that the second step is the same for every algorithm. These steps keep repeating until no change or small change is made in the centroids.

Lloyds Algorithm

Lloyds Algorithm is basically the brute-force approach to the Clustering Problem. - In the Assignment step, every data point is compared with every cluster center to

find the closest one and assign that point to it. - In the update step, for every cluster the new center is computed by taking the median of every component of all the vectors in the cluster.

Reverse Assignment using Range Search LSH

The goal behind this algorithm (and the Hypercube version respectively) is to avoid brute-force checking every data point in the assignment step. - In the Assignment step, we use Range Search LSH with increasing radius to find the nearby points, until some threshold is achieved. After that, every non-assigned point in the end is then assigned manually by computing the distance to each centroid and taking the minimum. - In the update step, as in Lloyds, we compute the median of every dimension of all the vectors in the cluster, and assign it to the centroid.

Reverse Assignment using Range Search Hypercube

Again, the steps are pretty much similar. - In the Assignment step, we use Range Search Hypercube with increasing radius to find the nearby points, until some threshold is achieved. After that, every non-assigned point in the end is then assigned manually by computing the distance to each centroid and taking the minimum. - In the update step, as before, the new centroid is computed by taking the median of every dimension of all the vectors in the cluster.

Implementation

How the Repository is organized

The main directory contains the entities: - bin: Target directory where binary files (executables) are placed - config: Directory that contains configuration files for the Clustering Algorithms. - ANN: Directory that contains the main .cpp files of LSH and Hypercube programs. - Clustering: Directory that contains the main .cpp file of the Clustering program. - include: Directory that contains the libraries (.h files) used in both parts of the Project. More on this directory in a bit. - object: Directory that contains the Object (compiled) files that are created during compilation. - output: Directory that contains output .txt files. - src: Directory that contains source code for both parts of the Project. More on it a in a bit. - Makefile: A makefile used to build the executables of the Project.

The include directory contains sub-directories with .h or .hpp files. Their purpose is pretty much self-explanatory. The sub-directories are: - include/BruteForce: Contains generic library for running Brute Force algorithm to solve the exact Nearest Neighbors Problem. - include/Clustering: Contains generic library for the different clustering algorithms. - include/core: Contains a header that defines a struct for our data points (objects/items). - include/Hypercube: Contains a generic library that implements the Hypercube Data structure used in the ANN problem. - include/interfaces: Contains header files that define the different utilities used to read input/write output, and move around the Data from the Dataset in our programs. - include/LSH: Contains a generic library that implements the LSH Data structure used in the ANN problem. - include/metrics: Contains a generic library used to implement the different metric functions. - include/utils: Contains generic libraries with utilities used by other parts of the project.

The src directory contains sub-directories with .cpp files. Again, their purpose is pretty much self-explanatory. The sub-directories are: - src/interfaces: Contains implementations of different interface function defined in the respective .h header files.

How ANN is implemented

1. LSH The code for LSH can be found under the include/LSH directory. It contains two files, include/LSH/LSH.hpp and include/LSH/LSHFun.hpp.
    1. include/LSH/LSHFun.hpp
        1. HashFunction is used to represent the "h" function in which an item is at first shifted and then applied the following fomula: $h(p)=a_{d-1}+ma_{d-2}+...+m^{d-1}a_0$
        2. AmplifiedHashFunction represents the "g" function which "combines" k number of "h" functions by concatenating bitwise their results.
    2. include/LSH/LSH.hpp
        1. LSH is used to combine the aforementioned HashFunction and AmplifiedHashFunction, it represents the entire LSH data structure. It contains L hash tables, which have their respective amplified hash functions to map items into buckets. This class contains the methods kNN and rangeSearch.
            1. kNN searches for the k (given as argument) nearest neighbors of the query input. At first, it hashes the query with each amplified hash function, it then searches inside the respective buckets (to which "query" is mapped) of each hash table. It returns a sorted array (on ascending distance) of pairs of distance and item.
            2. rangeSearch is similar to kNN. It searches inside the same buckets as kNN however, it appends to an array (similar to the one used in kNN) every item which has a distance from the query item of less than R (given as argument).
2. Hypercube The code for the randomized projection to a hypercube can found under include/Hypercube. It contains one file:
    1. include/Hypercube/Hypercube.hpp
        1. FFunc is used to represent the "f" function in which the result of an "h" function (implemented in include/LSH/LSHFun.hpp as HashFunction) is mapped to {0,1}. When initialized, a random integer is created within range (min, max) (given as arguments). Whenever an "h" value arrives, if that h value is less than the random integer, 0 is returned, else, 1 is returned. This range can be calculated through the mean and deviation of the "h" values. Specifically: include/utils/ANN.hpp implements a function which calculates those values. The final range could be (mean-deviation, mean+deviation).
        2. Hypercube combines include/LSH/LSHFun.hpp->HashFunction and FFunc to implement a Hypercube whose vertices contain items. Whenever an item arrives, all "h" and "f" function values are calculated. From that, a vector of {0,1} is constructed (represented as an integer) which is the vertex of the hypercube. This class also contains kNN and rangeSearch methods.
            1. kNN takes in 4 arguments: a query item, k (how many neighbors to be returned), probes (how many vertices to search), thresh(old) (how many total items to search). Until the probe and thresh end, starting from vertices with hamming distance = 0 (the startingVertex itself), then with hamming distance = 1 (the vertices next to startingVertex who only differ by one bit) etc. we search the whole vertex to find the k most similar vertices. This returns a sorted array (on ascending distance) of pairs of distance and item.
            2. rangeSearch instead of k, it takes in R as an argument which is the maximum distance an item to be returned must have. The probes are traversed in the same way as kNN, only this time, instead of searching for the k most similar, all items which are inside the "ball" created by R are returned (with their respective distances)

How Clustering is implemented

The code for the Clustering part can be found in the file include/Clustering/clustering.hpp. There are 2 objects created in this file: 1. A struct **ClusterCenter**, used to represent a centroid and the vectors in its cluster. 2. A class **Clustering**, used to provide and interface between the different methods in the clustering procedure.

In our Dataset, the data points are grayscale images of size 28x28. The images are flattened to an array of size 28 x 28 = 784 pixels (or bytes). Thus, in our case, the vectors contain pixels (unsigned chars == uint8_t), but this can change since we have used templates.

When the constructor of a Clustering object is called, the centroids get initialized with the method initialization++. After that, we can call the method Clustering::perform_clustering() with the correct parameters to begin one of the clustering algorithms descriven above. Then, we can call the method Clustering::compute_average_silhouette() to compute the Silhouette values from the Clustering that has been produced.

You can take a look at the main function for the Clustering, to get a taste of how it works.

How to Compile and Run the Code

When compiling the code, the objects will always go in the object directory, and the executables will go to the bin directory.

The are 4 options when compiling the code. 1. To compile the LSH code: `bash $ make LSH`

2. To compile the Hypercube code:

```
$ make HC
```

3. To compile the Clustering code:

```
$ make CLUSTER
```

4. To compile all of the above:

```
$ make ALL
```

Here are some examples on how to run the compiled code:

1. Run the LSH with
    - d = ./Dataset/train-images-idx3-ubyte, the path to the input file
    - q = ./Dataset/t10k-images-idx3-ubyte, the path to the query file
    - k = 6, the number of LSH hash functions
    - L = 4, the number of HashTables
    - o = ./output/lsh_output.txt, the path to the output file
    - N = 10, the number of Approximate Nearest Neighbors to find
    - R = 10000, the radius used to perform the Range Search LSH

```
$ ./bin/lsh -d ./Dataset/train-images-idx3-ubyte
            -q ./Dataset/t10k-images-idx3-ubyte
            -k 6
            -L 4
            -o ./output/lsh_output.txt
            -N 10
            -R 10000
```

2. Run the Hypercube with
    - d = ./Dataset/train-images-idx3-ubyte, the path to the input file
    - q = ./Dataset/t10k-images-idx3-ubyte, the path to the query file
    - k = 14, the dimension in which the points will be projected
    - M = 10, the maximum number of points to check
    - probes = 2, the number of edges of the hypercube to examine
    - o = ./output/lsh_output.txt, the path to the output file
    - N = 10, the number of Approximate Nearest Neighbors to find
    - R = 10000, the radius used to perform the Range Search LSH

```
$ ./bin/cube -d ./Dataset/train-images-idx3-ubyte
                -q ./Dataset/t10k-images-idx3-ubyte
                -k 6
                -M 10
                -probes 2
                -o ./output/lsh_output.txt
                -N 10
                -R 10000
```

3. Run the Clustering with
    - d = ./Dataset/train-images-idx3-ubyte, the path to the input file
    - c = ./config/cluster.conf, the path to the configuration file
    - o = ./output/lsh_output.txt, the path to the output file
    - complete, flag given to log more information
    - m = LSH, the method to perform the Clustering with (available options are: Classic, LSH, Hypercube)

```
$ ./bin/cluster -d ./Dataset/train-images-idx3-ubyte
                -c ./config/cluster.conf
                -o ./output/lsh_output.txt
                -complete
                -m LSH
```

To remove the objective and the executable files, just give the command

```
$ make clean
```

GitHub Repository: https://github.com/DemetrisKonst/ANN-Clustering