Université d'Ottawa
Faculté de génie

École de science
d'informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical
Engineering
and Computer Science

Nearest neighbor algorithms
Programming assignment
Part 1 (10%)
CSI2110 Algorithms and Data Structures

Fall 2023

Due: Saturday October 21 at 11:59PM (1min-24late: -30%; not accepted after 24hs)

This is an individual assignment. You can obviously discuss with your colleagues but do not show your code, do not share your code, do not publish your code, do not copy code. Doing so may lead to accusation of fraud.

Late assignment policy: ***1min-24hs late are accepted with 30% off; no assignments accepted after 24hs late.***

Problem description

In this project, we will explore the Nearest Neighbor search problem. Given a large number of vectors (the dataset) in a high-dimensional space and given a query vector, the objective is to find the vector that is the closest to the query vector, usually using a Euclidean distance (but not necessarily). This vector is called the nearest neighbor; in the general formulation, the objective could be to find the k nearest neighbors (kNN).

This is an important problem in computer science that is used in many applications. For example, for similar-image search, one strategy that is often used is to represent an image, through complex computations, using a high-dimensional vector (e.g. representing an image using a floating point vector of dimension 1024). If the mapping from an image to a vector works well, it is expected that two images that have similar content will be represented by two vectors that are relatively close to each other. Another application could be for music recommendation. Echo Nest is a company that produces a system extracting audio features and meta-data analysis of songs. Using these, one can build a vector representation of a song and use it to find similar songs in a music catalog. This company was acquired by Spotify in 2014. If you are interested in these subjects, here are two references:

- Douze, Matthijs & Jégou, Hervé & Sandhawalia, Harsimrat & Amsaleg, Laurent & Schmid, Cordelia. (2009). Evaluation of GIST descriptors for web-scale image search. International Conference on Image and Video Retrieval.
- Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. (2011). The Million Song Dataset. In Proceedings of the 12th International Society for Music Information Retrieval Conference (ISMIR 2011), 2011.

Nearest Neighbor algorithms

From an algorithmic point of view, this problem is easy to solve. You have a given query point of dimension D and you just go through the list of N points in your dataset in order to find the closest one. The complexity in this case is simply O(DN).

However, in the case of a very large dataset, this task could be very time consuming. It therefore exists algorithms that try to speed up the nearest neighbor search process. K-D trees is an example of such algorithm, however this one is not the most efficient when the search space is of high dimension since its complexity is O(DlogN). The impact of space dimension on search algorithm is often referred as the *curse of dimensionality*.

In some applications, the search for nearest neighbors can be accelerated even further if some inaccuracies can be tolerated. This category of algorithms is called Approximate Nearest Neighbor (ANN) search. The idea is to quickly return a point in the dataset that is very close to the query point without necessarily being the nearest one. We will explore one of these algorithms in the programming assignment Part 2 (P2) but before, let's solve the exact kNN problem. In fact, we will have to use a kNN module in our future ANN solution.

Finding the k Nearest Neighbors

We have a set of N points in a D-dimensional space. We have a query point Q which is not part of the set. We would like to find the k points in the set that are the nearest to Q. A simple algorithm is given next:

1. Create an empty set of neighbors to store the k-nearest neighbors.
2. For each point P in the set:
   a. Calculate the Euclidean distance dist(Q, P) between the query point Q and the point P.
   b. Add P to the neighbors set.
   c. If the size of the neighbors set exceeds k, remove the point with the farthest distance from Q.
4. Return the neighbors set as the k-nearest neighbors of point Q.

Looking at Step 2c., a wise programmer with good experience with data structures will observe that since we always have to remove the point with the highest distance to the query point, a priority queue of size k+1 appears to be a good choice for storing the current k nearest neighbors.

As a matter of fact, if we ask chatGPT to propose a Java code implementing this algorithm, it produces the code given on the next page (this is the unedited solution generated, we only added the comments). The complete source code is provided in a separate file.

```
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;

public class KNearestNeighbors {
    // finds the k nearest neighbors of
    // query point in list of points
    public static List<Point>
      findKNearestNeighbors(Point query, List<Point> points, int k) {

        // maxHeap to contain the neighbors
        PriorityQueue<Point> maxHeap = new PriorityQueue<>(
              // comparator used in Heap
              (a, b) -> Double.compare(b.distanceTo(query),
                                       a.distanceTo(query)));
              // * good, but this comparator means that we have
              // to compute the same distance many times…
              // may be a good idea to store each computed
              // distance value

        for (Point p : points) {

            maxHeap.offer(p);          // insert p in Heap
            // * well, it could be better to test is p will become
            // the root of the heap before we insert it… otherwise
            // we insert and remove the same point
            if (maxHeap.size() > k) {  // if Heap is full
                maxHeap.poll();        // removeMax from Heap
            }
        }

        // transfer neighbors to a list in order
        List<Point> neighbors = new ArrayList<>();
        while (!maxHeap.isEmpty()) {
            neighbors.add(maxHeap.poll());
        }

        return neighbors;
    }
```

Your task

Your mission for Programming Assignment P1 is to implement a few variations of the kNN algorithm. Our focus is on the priority queue that is used to update the list of currently found neighbors.

The data

We give you four files. Two files containing point datasets; you use the small dataset for your tests and the larger one to generate the experimental results. Two files containing query points for your kNN searches; in this first programming assignment, just use the small file. We also give you a utility file, `PointSet.java`, containing a class that reads a point set from these files.

The programming task

We ask you to create three versions of a program that finds the kNN of a query point. You have to define the class `KNN` with the following methods:

- An instance of this class is constructed by providing a reference to a `PointSet` object;
- A setter method `setK` is used to set the value of k;
- A method `findNN` that finds the k nearest neighbors of a query point, specified with a reference to a `LabelledPoint` instance. The method returns the list of nearest neighbors in an `ArrayList<LabelledPoint>` object.

You can also create other methods or classes.

These three versions will contain different implementations for the priority queue. More specifically, you have to define the class `PriorityQueue`, in three versions, all implementing the `PriorityQueueIF` interface included in the project. Therefore, these classes will have the following methods:

- A constructor that creates an empty queue by specifying the capacity of this queue;
- A constructor that creates a queue from a specified ArrayList instance and a capacity;
- The methods `offer`, `poll`, `peak`, `size`, `isEmpty` as they are defined in the standard `java.PriorityQueue` class;

The three versions are as follows:

- `PriorityQueue1` : this version will be based on a priority queue implemented with a simple array; the max element being the last element in the array.
- `PriorityQueue2` : this version will be based on a priority queue implemented with an array representing a heap; the max element being at the root of the heap tree. Each insertion in the heap will trigger a upheap operation. Each removal from the tree will trigger a downHeap operation.

- `PriorityQueue3`: this version will simply use the `java.PriorityQueue` standard class as an instance variable in this class.

The program should simply output to the console the list of nearest vectors for each query vector, contained in the query file. The vectors are specified using their labels. Each line contains the query vector label (in order, starting at 0), followed by : and the list of neighbors, comma separated. For example:

```
0 : 10, 3, 77, 4, 877
1 : 130, 43, 57, 324, 87
…
```

You should create your solutions from the instructions given here and from your class notes. Do not use fancy code found on the Internet or code produce by generative AI tools. You must cite any external resources you may have used to inspire your solution; if we find out that you used an uncited resource or tool, you automatically get 0 for this assignment.

The experiments

Once you have defined and tested your three version of the KNN class, you now have to perform the following experiments:

- Using the dataset of 1,000,000 points, we ask you to measure the execution for finding the k nearest neighbors of 100 query points;
  - Use the two files provided matching this description.
- You perform this experiment for k= 1, 10, 50, 100, 500 and 2000;
  - The results printed to the console are saved in a text file following the naming convention:
    - knn_version_k_numberOfquery_databaseSize
    - for example, the results of the experiment using version 1 to find the 50 nearest neighbors of 1000 query vectors for a dataset of 1000000 vectors should stored in the file: `knn_1_50_1000_1000000.txt`
- Report your results on a graph showing k-value on the x-axis and execution time in ms on the y-axis;
- Comment on the results obtained.

FYI: On a Windows LapTop XPS13 with Intel i7-10710U CPU @ 1.10GHz, 6 Cores, the execution time for finding the 1-NN of 100 query points in a point set of 1000000 points was 43857 milliseconds.

Submission instructions

For this first programming assignment, you have to submit the following items:

- **All your Java classes in a zip file**, in particular
  - the `KNN` class;
  - the `PriorityQueue1`, `PriorityQueue2`, `PriorityQueue3` classes;
  - remember that these classes must implement `PriorityQueueIF`
  - and all the other classes required to run your solution
  - to run your program, the following arguments should be provided to the main program (via: `public static void main(String[] args)`)
    - version number
    - value of k
    - dataset filename
    - query point filename

    for example, the command line for running this class would look like:
    ```
    java KNN 1 10 sift_base.fvecs siftsmall_query.fvecs
    ```

Marking scheme

KNN class	10%
PriorityQueue 1,2,3 and other classes	34%
Quality of programming (structures, organisation, etc)	10%
Program efficiency (time and memory)	10%
Quality of documentation (report, comments and headers)	6%
Computational time experiment	20%
Discussion and comments on results	10%

All your files must include a header that includes your name and student number. All the files must be submitted in a single zip file.