Nearest Neighbor Graph

Myrto Paraforou
Danae Karageorgopoulou

January 14, 2024

1 Problem

Nearest Neighbor Search (NSS) is an optimization problem involving the identification of a point within a given dataset that is closest (or most similar) to a specified point. Proximity is typically expressed through a dissimilarity metric, signifying that objects are less similar as the dissimilarity value increases.

The conventional definition of the Nearest Neighbor Search problem is as follows: consider a set S of points in a space M and a query point q $\in$ M. Determine the point in S that is closest to q. In most cases, M represents a metric space, and dissimilarity is articulated as a distance metric, which is a symmetric measure adhering to the triangular inequality. Additionally, M can be a space of d dimensions, where dissimilarity is gauged using metrics such as Euclidean distance between vectors.

In the context of this project report, we explore the principles and methodologies associated with solving the Nearest Neighbor Search problem with the use of k-Nearest Neighbor Graph, emphasizing the use of dissimilarity metrics and distance measures within the given dataset.

2 Introduction

In this project, the main idea revolves around the creation of a nearest neighbor graph, crucial for finding approximate nearest neighbors later. This involves transforming each data point into a graph node and connecting it to the k most similar or closest other points based on metrics like Euclidean distance in n-dimensional space.

3 Assumptions and Disclaimers

The KNN graph is designed specifically for datasets from the SIGMOD 2023 programming contest, and it works within a 100-dimensional space.
The entire project is developed in C++, and notably, it does not rely on STL data structures.

To manage the storage of both neighbors and data points, we utilize a custom data structure called ADTSet in C, which is implemented internally within the project. The foundation of ADTSet is built upon the open-source code of the k08 class (optimized version performed), accessible at https://github.com/chatziko-k08/lecture-code.
All the methods, routines and sub-routines have been thoroughly tested using unit testing methods by the use of the open source testing library https://github.com/mity/acutest.

4 Algorithm Review

The implementation relies on the KNNDescent class, which encapsulates essential information for representing the k-NN graph. The creation and maintenance of this graph utilize the Vertex and Neighbor classes. These classes store crucial details about each graph vertex, including pre-calculated distances between data points in space. The graph is constructed in the class member method createKNNGraph.

Additionally, there is the KNNBruteForce class, responsible for computing the accurate k-NN graph. It operates independently from the KNNDescent class and is later used as a similarity measure to compare how accurate is the approximate calculation of the KNNDescent method.

4.1 Graph Initialization

The implementation offers two separate methods for initializing the graph before any procedure begins to operate:

- Random Initialization
- Random Projection Forest Initialization

In the **random initialization method**, each vertex (datapoint) is connected to k completely random vertices inside the dataset. As a result, the accuracy of the randomly initialized graph is pretty low (approximately 10%).

The **random projection forest initialization method** will be further examined in the next section of the report.

4.2 Potential Neighbor Calculation

Before any modifications are made to the graph, the potential neighbors for updating each vertex are computed. The potential neighbor calculation method determines prospective neighbors for each vertex by evaluating both neighbors and reverse neighbors, taking their distances into account. The selection of potential neighbors is accomplished by executing an all-pairs distance comparison between the combined set of neighbors and reverse neighbors of a particular vertex (local join). The local join process involves the following steps for each pair of neighbors in the combined set:

1. Compute the distance between the two vertices using the specified metric.
2. Verify if the calculated distance exceeds the distance to the furthest neighbor.
3. If the distance is greater, the comparison is skipped, indicating that the new potential neighbor is not closer than the current furthest neighbor.
4. If the distance is not greater, it signifies a closer potential neighbor. Consequently, the new potential neighbor is added to the set of potential neighbors associated with the respective vertex.

Due to the high computational cost of local join, a sequence of optimizations has been implemented to enhance the efficiency of the fundamental algorithm:

- **Incremental Search**
  As the algorithm continues, fewer new items are added to the K-NN graph in each iteration. Therefore, conducting a complete local join in every iteration is inefficient, given that many pairs have already been compared in earlier iterations. As a result, each neighbor carries a boolean flag that signifies whether or not it has been compared at least once or not. When a vertex is initially assigned as a neighbor to another vertex, its flag is true. Two neighbors (or reverse neighbors) will be compared with each other only if at least one of them is new to the set (meaning that at least one flag must be true).
- **Sampling**
  The expense of each iteration amounts to K^2N similarity comparisons. To reduce the computational workload, we use a sampling strategy. This involves, marking nodes for comparison and constructing sampled lists of neighbors and reverse neighbors. Adjusting the sampling rate p allows for finding a balance between precision and computational time.

4.3 Graph Update

The `updateGraph` method refines a k-NN graph by iteratively updating vertex neighbors based on potential neighbors, enhancing graph quality. The algorithm compares distances between potential and current neighbors for each vertex, adjusting the graph until no further improvements can be made.

1. For each vertex, identify potential and current neighbors.
2. While there are potential neighbors closer than the furthest current neighbor, update the graph by adding new neighbors and adjusting reverse neighbors.
3. The method concludes employing an early termination technique (explained below:)

- **Early Termination**
  In response to the observation that the number of updates in each iteration of the k-NN graph update decreases over time, a strategic approach has been implemented. Instead of relying on the criterion of zero updates to terminate the updateGraph calls, the algorithm uses δ as a precision parameter, deciding to stop when the count of K-NN list updates in each iteration falls below δKN. This adjustment allows for a more flexible and efficient termination strategy, where the algorithm adapts to the decreasing returns in terms of updates and terminates based on a threshold that considers both the precision parameter δ and the product of K and N.

5 Optimizations

The following methods and algorithms have been integrated into the code to optimize its performance with large datasets:

5.1 Random Projection Forests

The creation of a random projection forest has been used for optimally initializing the KNN Graph before any further actions on the basic NNDescent algorithm. The random projection forest is based on the formation of a **single random projection tree**.

The initial step involves constructing a random projection tree using the vertices of the graph as a whole data space. In this process, the high-dimensional space is randomly partitioned into smaller subsets or clusters using random hyperplanes. The leaves of the tree are constrained to be equal to or smaller than a specified size, typically around 80% of the size of k. Subsequently, each vertex in the graph is designated as a neighbor exclusively to the other neighbors within its specific cluster. Given that the size of each cluster is less than 80% of k, the remaining neighbors required for initializing the graph are randomly assigned from the entire dataset.

The initialization relying on a single random projection tree is insufficient to generate an initial graph with a significant enough similarity percentage for the random projection method to be considered successful. In fact, using only one tree results in a mere 3-4% improvement in similarity compared to the random initialization method. Therefore, the initial graph keeps being updated until it achieves a similarity percentage of approximately 50-60%. This is accomplished by generating additional random projection trees and updating the existing nearest neighbors of each vertex based on the cluster information of each tree. To attain this level of similarity, it is estimated that approximately 7-8 random projection trees are required.

5.2 Parallelization

To enhance the efficiency of our algorithm and take advantage of the capabilities of multi-core processors, we use the `std::thread` library provided by C++. This library made the concurrent execution of tasks easier, making our algorithm faster by parallelizing subroutines with high computational load. The decision to use `std::thread hardware concurrency` was strategic, dynamically determining the number of available hardware threads and adapting our parallelization strategy accordingly. By aligning the number of threads with the hardware capabilities, we aimed to achieve optimal resource performance.

Three critical methods within the implementation underwent parallelization: the one responsible for calculating the potential neighbors of each vertex, the method handling graph updates, another one that is used to create a random projection forest and the function dedicated to computing the squares of each data point. The key is to efficiently share the computational load among multiple threads, breaking down these intensive tasks. This parallelization strategy contributes to improved performance and faster execution, especially when dealing with large datasets and computationally demanding operations.

In the method responsible for :

1. calculating the potential neighbors of each vertex, the breakdown involved assigning a specific amount of vertices to each thread for the computation of their potential neighbors. Each datapoint has its own mutex, strategically blocked to ensure safe access to shared data.

2. graph updates, has a similar threading pattern, but with a more complex locking mechanism due to the amount of shared resources.

3. updating the random projection graph, a similar threading strategy is used. The task is to manage the random projection forest with a specified number of trees. So we divide the work among multiple threads, each responsible for updating a part of the graph.

4. computing the squares of each data point, accordingly assigns a specific amount of datapoints to each thread.

5.3 Distance Calculation

The Euclidean distance between two points in an N-dimensional space is defined as the square root of the sum of the squared differences of their components. In our specific case, the absolute value of the distance is not particularly significant; rather, its comparison with other distances matters. Therefore, the precise calculation of the square root is unnecessary. In this scenario, we can deal with computing the squared distance. Using the binomial theorem, we observe that the calculation of the distance between two points will always involve the square norm of each vector (inner product of the vector with itself). These norms can be precomputed and stored for later use, reducing the computational workload during the distance calculation process.

6 Graph Implementation

6.1 Graph Representation

The directed nearest neighbor graph uses **adjacency sets** in order to store the incoming and outcoming edges of each vertex. The adjacency set is a versatile alternative for the classic adjacency list approach. Each vertex of the graph has a set for its outcoming edges (nearest neighbors) and a set for its incoming edges (reverse nearest neighbors). The selection of this particular representation was chosen for the graph is because spacewise, the set occupies precisely the number of neighbors (outcoming edges) and the number of reverse neighborr (incoming edges) in a given space unit, and timewise, although the iteration might be longer than the linear iteration of a linked list, the random access of an item of the adjacency set is much quicker.

6.2 Graph Data Structure

As previously noted, for the storage management of both neighbors and data points, a custom data structure named **ADTSet** is implemented and utilized. This data structure offers an instance of a Set, equipped with various set methods for the insertion and removal of items, retrieval of items within the set, and other essential functionalities such as determining the size of the set. The items that are being inserted in the set are ordered, and for that reason we must provide a value based on which the items will be sorted, as well as a custom made compare function for the specific item type's needs.
The implementation of the Set structure is based on the formation of an AVL Tree, so, each insertion takes O($logn$) time, where n is the size of the set at a specific moment. To resemble a max heap, we have altered the internal implementation of the ADTSet, so that it has direct and easy access to the item with the maximum value. This appeared to be helpful for the KNNDescent methods, because the "furthest neighbor" or, in other words, the neighbor in a set of neighbors with the maximum distance from the source/vertex, is often requested.

7 Experimental Results

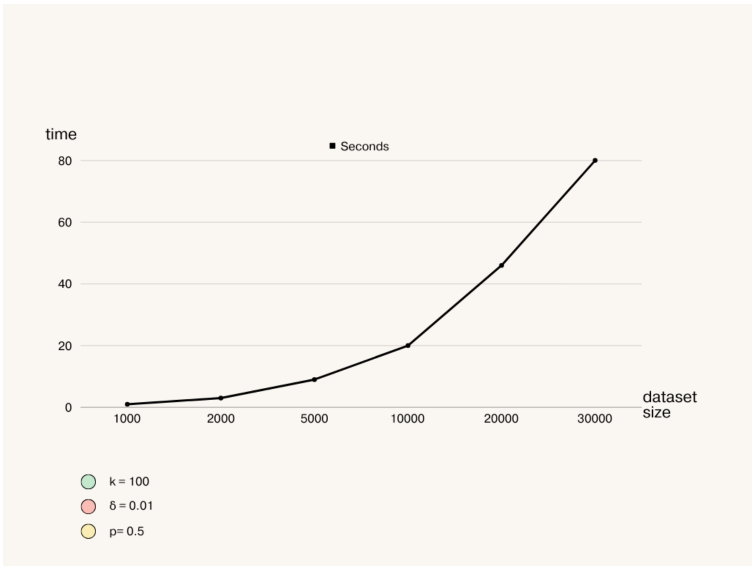

Figure 1: Impact of Dataset Size on Computational Time

As the dataset size varies, the impact on computational time is visually represented. The plotted data illustrates a non-linear relationship, showing that as the dataset size grows, computational time experiences an exponential increase.

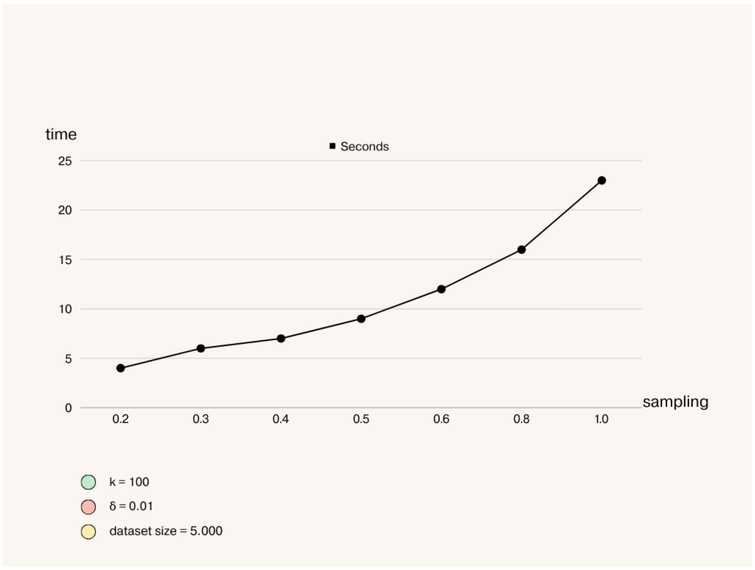

Figure 2: Execution Time Across Different Sampling Factors

The figure illustrates the relationship between execution time and various sampling factors. It's interesting to observe that lower sampling factors result in efficient computation. However, note that: very low sampling factors may compromise accuracy.

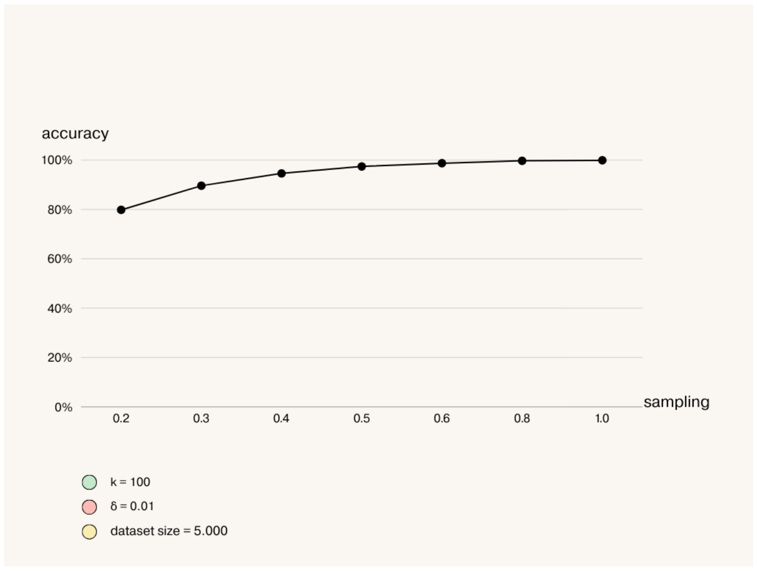

Figure 3: Effects of Sampling Factors on Accuracy

The figure shows how accuracy improves with higher sampling factors, reaching exceptional levels. However this observation, in connection with the previous plot, emphasizes the need to strike a balance between fast computation and maintaining accuracy in different sampling scenarios.

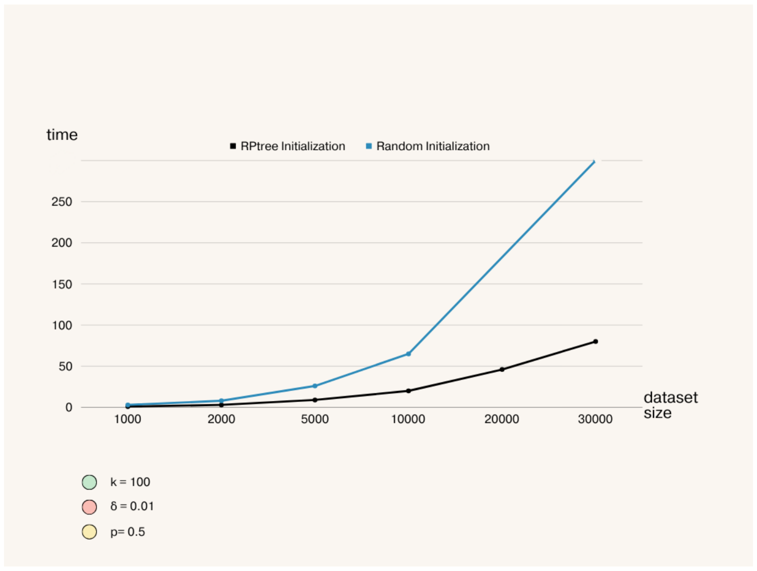

Figure 4: Time Comparison: RP-Tree Initialization vs. Random Initialization with Increasing Dataset Size

This comparative analysis contrasts the computation time between RP-Tree Initialization and Random Initialization as the dataset size grows. Notice that the implementation that uses RP-Tree Initialization requires much less computation time compared to Random Initialization. Additionally, for dataset sizes exceeding 20,000, the computation time for the Random Initialization implementation remains uncalculated due to its excessive demands.

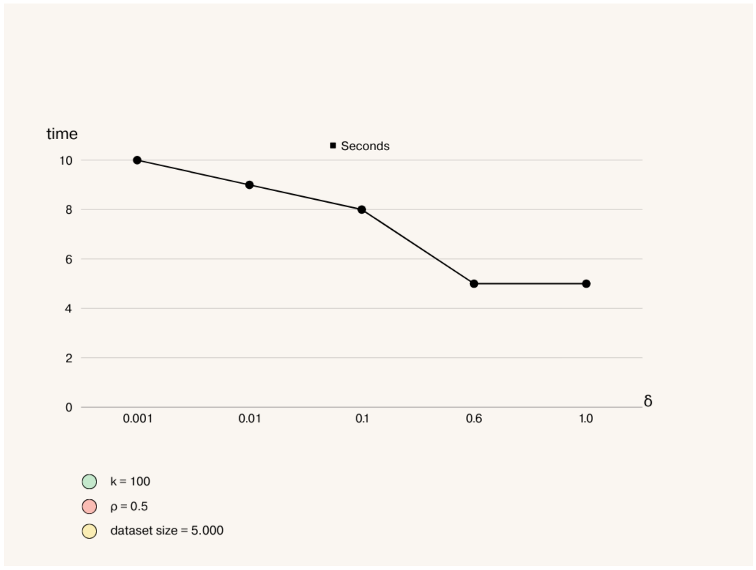

Figure 5: Execution Time Across Different Delta Factors

The figure illustrates the Execution Time Across Different Delta Factors, revealing a reduction in execution time as the delta factor increases. It is obvious, that there is a critical point where the time experiences a significant decrease. However, it's crucial to note that very low delta factors may compromise accuracy.

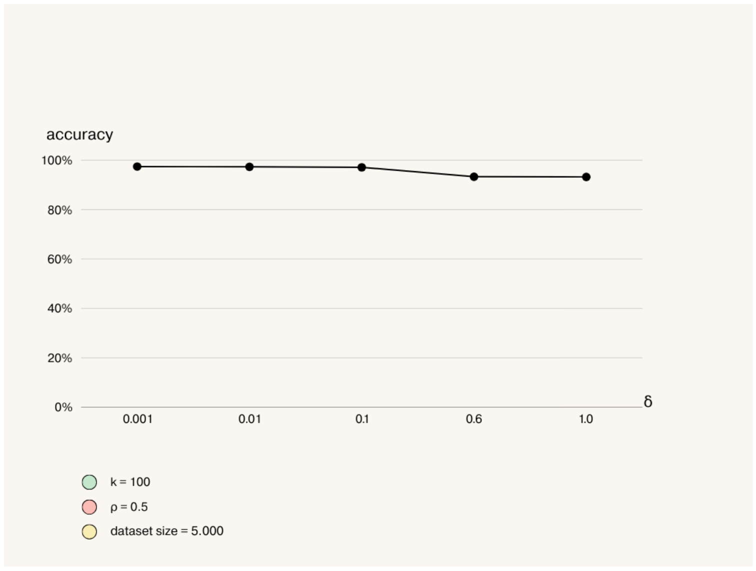

Figure 6: Effects of Delta Factors on Accuracy

The figure shows how accuracy reveals a decrease with higher delta factors, reaching unwanted levels. Notably, at the same point where the previous plot showcased a significant reduction in time, the accuracy appears to be compromised. This underscores the crucial trade-off between computational efficiency and accuracy. Thoughtful delta factor selection is crucial for achieving a balanced result.

8 Testing Environment

- **Machine Hostname:** Linux08
- **Linux Kernel Version:** 5.4.0-169-generic
- **Distribution Information:** Ubuntu SMP #187
- **System Architecture:** x86_64 (multiprocessor core)
- **Operating System:** GNU/Linux
- **Number of CPUs:** 4
- **CPU Model:** Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
- **Total RAM:** 16GB
- **Threads per Core:** 1

References

[1] Wei Dong, Moses Charikar, and Kai Li. 2011. *Efficient k-nearest neighbor graph construction for generic similarity measures.* In Proceedings of the 20th international conference on World Wide Web (WWW '11). Association for Computing Machinery, New York, NY, USA, 577–586.

[2] Dan Kluser, Jonas Bokstaller, Samuel Rutz, and Tobias Buner. *Fast Single-Core K-Nearest Neighbor Graph Computation.*

[3] Description of the implementation of the algorithm in Python for the PyNNDescent project.