Exact and Approximate k-Nearest Neighbors Implementation and Analysis

Aristeidis Daskalopoulos

Introduction

The k-Nearest Neighbors (k-NN) algorithm is a fundamental method in machine learning and data analysis, commonly used for classification, regression, and clustering tasks. It works by identifying the k closest points to a given query point based on a distance metric, which in this project is the Euclidean distance.

We have implemented both **exact** and **approximate** approaches to solve the k-NN problem, with a focus on time and memory efficiency, particularly when the project runs on large datasets. All of the *exact* methods/functions we implemented guarantee correct results, while the *approximate* methods trade some accuracy for improved speed, especially when working with large datasets. Our solutions are designed to efficiently handle large-scale datasets by utilizing parallel computing techniques.

This report provides a theoretical analysis and basic test results. For a deeper understanding of the code, see `README.md` file.

The Exact k-NN Algorithms

Overview

The exact k-NN algorithm computes the pairwise distances between query points and all corpus points to construct a complete distance matrix. From this matrix, the k smallest distances for each query are selected, along with their corresponding indices. Mathematically, given a corpus $C \in R^{n \times d}$ and queries $Q \in R^{m \times d}$, we compute the squared distance matrix:

$$D[i, j] = \|Q[i, :] - C[j, :]\|^2 = \sum_{k=1}^{d} (Q[i, :] - C[j, :])^2.$$

This is an explanatory/mathematical representation of the calculation we perform in the code. As we can see, the matrix D has dimensions $m \times n$, and for each query q_i in every row, the k indices corresponding to the smallest values in $D[i, :]$ are the ones that we need to find.

The issue with matrix D is that it contains $m \times n$ floating-point values. This results in an extremely high memory requirement to calculate and store such a matrix in its entirety. To handle this on a typical computer, we split the matrix D into smaller chunks along the query dimension. Specifically, we divide the queries into smaller subsets, or "query-chunks," and for each chunk, we compute the k-NN based on the full corpus using a fast selection algorithm. This approach breaks the problem into smaller subproblems, which can be solved independently. The final result of our functions is two complete k-NN matrices (one matrix holds the indices of the neighbors and the other is for the corresponding distances), where the structure of the output is similar to that of MATLAB's `knnsearch`.

As is already clear, each subproblem created with this approach is independent of the others. Therefore, instead of solving them sequentially, we can solve as many as possible in parallel, depending on the number of available threads. This parallelization slightly accelerates the computation of exact k-nearest neighbors (k-NN), yielding a measurable improvement—approximately a 25-30% reduction in runtime for 4 threads on large datasets. However, when multiple threads are executed simultaneously, the memory available to each thread decreases. For instance, if 4 threads are used and 10GB of memory is available, each thread can only access one-quarter of the total memory, which may slow down the sequential k-NN computation.

Implemented Functions

We implemented four exact k-NN functions inside the `exact/` folder. All of them solve the general k-NN problem, where the test set does not have to be equal to the training set, and are designed to handle large datasets, especially when it is not feasible to calculate and store all the distances at once. These functions are fundamental because we will relay on them when we will start implementing the approximate methods.

- **knn_exact_serial:** A straightforward implementation processes the entire dataset sequentially. In each loop, it loads as many queries as the system's available memory allows in order for each query to find the k-nearest neighbors (k-NN) based on the entire corpus.
- **knn_exact_pthread:** The queries are divided into sub-blocks - based on the number of threads we asked for - and multiple threads independently compute the k-NN for their assigned query subsets using the `knn_exact_serial`.
- **knn_exact_openmp:** The loop that we introduce in the `knn_exact_serial` over queries is parallelized using OpenMP's work-sharing constructs, enabling efficient multi-threading with shared memory.

- **knn_exact_opencilk:** A task-based approach where parallelism is expressed via dynamic task creation and scheduling, using the same logic as we follow in the `knn_exact_openmp`.

Tests

To run all the exact functions and verify that the results are correct, we download and use the 'sift-128-euclidean.hdf5' file, which can be found and installed from the following repository: `https://github.com/erikbern/ann-benchmarks`. Building, executing, and testing all exact methods can be easily achieved by running the following script from the project folder:

```
./run_knn.sh 0 4 data/sift-128-euclidean.hdf5 train test 100 data/sift-128-euclidean.hdf5 neighbors distances
```

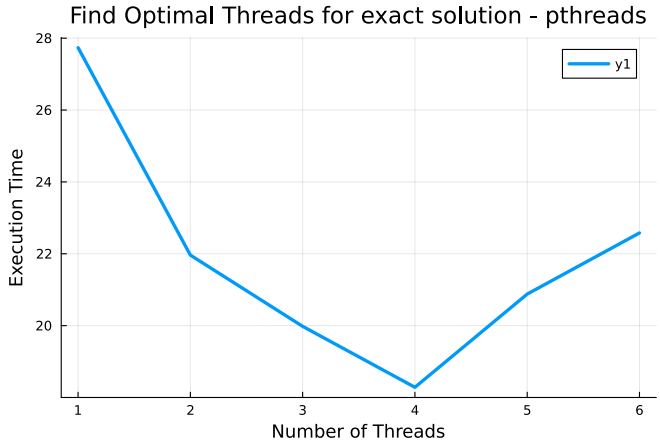
The second parameter specifies the number of threads to be utilized. In this example, we calculate and evaluate the 100-nearest neighbors (100-NN) of the test set, using the train set as a reference. We then compare our output matrices (indices and distances) against the expected results from the dataset to validate accuracy.

The results of running the 100-NN for all the exact implementations on our machine are as follows:

- **Running knn_exact_serial:**
  - Running time: 27.736545 seconds, Queries per second: 360.535171
- **Running knn_exact_pthread with 4 threads:**
  - Running time: 18.980681 seconds, Queries per second: 526.851487
- **Running knn_exact_openmp with 4 threads:**
  - Running time: 18.835897 seconds, Queries per second: 530.901183
- **Running knn_exact_opencilk with 4 threads:**
  - Running time: 18.986052 seconds, Queries per second: 526.702445

  We also ran the Julia exact serial implementation, which is quite slow, with a running time of 3 minutes for the same dataset. However, it serves as a valuable tool for verifying the correctness of the C results when the expected outputs are not available.

By changing the number of threads we observe how the program's speed changes without affecting the accuracy of the results (as expected - the methods always find the exact solution). In more detail, we observe that increasing the number of threads reduces the runtime, but this effect diminishes after using 4 or 5 threads (depending on the available memory that our computer had at the method's execution time). This behavior occurs because, as the number of threads increases, the memory available to each thread decreases (as discussed in the theoretical overview). By analyzing the diagrams, we can determine the optimal number of threads for our hardware (see that the y-axis starts at 18secs):



The dataset used has the expected results, with which we compare our outcomes. All of these results match (since the exact method is deterministic):
`Exact: Neighbors Mismatch Percentage: 1.994100%, Distances Mismatch Percentage: 0.000000%`

As we can see, the results are not perfectly matching, even with the exact solution. This discrepancy is due to the accuracy limitations imposed by floating-point rounding errors. For example, if two corpus samples have the same or nearly the same distance from one query, the handler responsible for sorting the indices (neighbors) based on the corresponding distances cannot reliably choose which one is closer. This outcome aligns with the 0% distance mismatch result, confirming that the distances of each of the k-NN found using our method are correct.

The Approximate k-NN Algorithms

While exact k-NN methods guarantee correct results, they can be computationally expensive and memory-intensive for large datasets. To address these limitations, we developed approximate k-NN algorithms that trade a small degree of accuracy for significant gains in speed and efficiency. These algorithms are particularly developed and run only for scenarios where $C = Q$, i.e., the corpus and query sets are identical.

Overview

The approximate methods aim to reduce the problem size by partitioning the dataset into smaller subsets. These subsets are processed independently, and the results are merged to approximate the k-NN for the entire dataset. The splitting and merging operations introduce a degree of uncertainty but significantly reduce computational overhead.

Mathematical Problem Formulation

Given a dataset $S \in R^{n \times d}$, the goal is to find the approximate k-NN for every data point $s_i \in S$. Instead of computing the complete distance matrix as in exact methods, the approximate approach divides S into 3 subsets:

$$S = S_1 \cup S_2, \quad S_1 \cap S_2 = \emptyset, \quad S_3.$$

For each one of the S_1, S_2 and S_3 we will find the local k-NN (we can do that using the exact method or by running the approximate method recursively). The final k-NN for each point $s_i \in S$ is derived by merging results effectively. Lets see now how exactly do we split the dataset S.

Dataset Splitting and Hyperplane Distance

The dataset is split based on a hyperplane defined by the geometric properties of the data. The process involves:

1. Computing two helper midpoints, μ_1 and μ_2, which are the centroids of two halves of the dataset.
2. Calculating the true midpoint $\mu = \frac{\mu_1 + \mu_2}{2}$.
3. Using the vector $\mathbf{v} = \mu_2 - \mu_1$ to define a hyperplane orthogonal to $\mathbf{v}$ that passes through μ.
4. Determining the distance of each point from the hyperplane as:

$$d_i = \frac{\mathbf{v} \cdot (\mathbf{s}_i - \mu)}{\|\mathbf{v}\|}.$$

Points are assigned to subsets based on the sign and magnitude of d_i. Points near the hyperplane (the programmer determines the definition of "near" in this approach by configuring the `__ZERO__` value within the `knn_approx_serial.h` file) are included in multiple subsets to account for edge cases.

Recursive Partitioning for Increased Accuracy

The splitting process can be recursively applied to each subset, resulting in finer partitions. The recursion depth is controlled by the accuracy parameter and the total number of threads that we have (the total number of threads plays a much more significant role in our implementations), which represents the trade-off between runtime and the granularity of the approximation. Finer partitions increases computation time but degrades result fidelity.

Merging Results

After solving the k-NN subproblems for S_1, S_2 and S_3, the results are merged. The merging process is as follows:

- We initially assign the results of S_1 and S_2 as the final results (remember that $S = S_1 \cup S_2, \quad S_1 \cap S_2 = \emptyset$).
- Then we check for each point $s \in S_3$ if any of the k-NN we found in the S_3 subproblem has lower distance than the one we calculate in the S_1 or S_2 subproblem
- Is so, we update the k-NN for this point by merging its initial k-NN and the one inside the S_3 sub-solution (we return the k-smallest of the union of these).

To conclude, for the initial dataset $S \in R^{n \times d}$, we needed to solve an $n \times n$ matrix (with a total of n^2 values). However, in the new approach, we calculate two matrices of size $n/2 \times n/2$ (with a total of $n^2/2$ values cumulatively) and also solve a small subset S_3, which contains values determined to be close to our hyperplane (where "close" is defined by the `__ZERO__` value, as discussed earlier). In most cases, this subproblem holds no more than $n/4$

points, as we only consider values that are "very" close to the hyperplane. Overall, this approach reduces the total number of calculations to approximately $\frac{9}{16} \cdot n^2$, which corresponds to a nearly 45% reduction in the total number of computations.

Parallel Approximate all-to-all-k-NN

The parallel implementation further accelerates the approximate method by leveraging multi-threading. To explain how its done we will introduce one by one all the implemented functions:

The Approximate k-NN Algorithms

The approximate k-Nearest Neighbors (k-NN) methods we implemented are designed to accelerate the k-NN search process by splitting the dataset into smaller sub-problems that can be solved independently and in parallel. While this approach trades some accuracy for speed, it maintains high efficiency for large datasets. Below, we provide an in-depth explanation of how our approximate methods work and the theoretical concepts they are based on.

Overview of the Approximate Approach

The goal of our approximate k-NN methods is to estimate the nearest neighbors for each query point without constructing a complete distance matrix between all query and corpus points. Instead, we split the dataset into smaller, more manageable subsets, perform k-NN computations on these subsets, and merge the results. This strategy allows us to parallelize the computation and reduce memory overhead significantly.

Dataset Splitting and Sub-Problems

The approximate k-NN algorithm starts by partitioning the dataset based on a splitting strategy. Given a dataset C of n points in R^d, we split C into smaller subsets by dividing the data space using hyperplanes:

1. Compute two helper midpoints of the dataset by averaging subsets of the data.
2. Use these midpoints to define a hyperplane that approximately splits the data into three parts:
    - **Part 1:** Points that fall on one side of the hyperplane.
    - **Part 2:** Points that fall on the opposite side.
    - **Part 3:** Points that lie near the hyperplane, defined by a tolerance parameter (i.e., the dataset is split with a margin around the hyperplane).
3. The programmer defines what is considered "near" using a parameter `__ZERO__` in the implementation, allowing for a margin of error when determining which points lie close to the hyperplane.

By dividing the dataset into these parts, we reduce the complexity of the problem to smaller sub-problems that can be solved independently.

Parallelization Strategies

To make the approximate k-NN algorithms faster, we employ parallel computing techniques to process each subset of the data concurrently. Below, we discuss the specific parallelization strategies used in each implementation:

Pthread-Based Parallelization

In the pthread-based implementation, the dataset is divided into several chunks, and each chunk is assigned to a separate thread. The main steps are:

1. Split the dataset into subsets using the hyperplane-based strategy described above.
2. Each thread is responsible for processing one of the subsets:
    - Each thread initializes a local data structure to store distances and indices.
    - The thread performs an exact k-NN search on its subset using a serial method.

3. The results from all threads are merged to produce the final k-NN results. If a point lies near the hyperplane (Part 3), it might belong to multiple subsets, and a verification step is performed to ensure the closest neighbors are accurately recorded.

This method relies on explicit thread management, where memory is dynamically allocated for each thread's subset, and synchronization mechanisms are used to ensure the results are correctly merged.

OpenMP-Based Parallelization

The OpenMP-based implementation leverages the OpenMP framework for loop parallelism and shared memory:

1. The dataset is divided into chunks based on the total number of available threads, similar to the pthread approach.
2. Using `#pragma omp parallel` constructs, each thread processes its chunk of the dataset independently:
   - A serial k-NN search is performed for each chunk.
   - A critical section is used to update the global result arrays, ensuring no race conditions occur during the merging of results.
3. A final pass is performed after parallel processing to ensure the accuracy of the merged results.

This method simplifies the parallelization using OpenMP's built-in constructs, reducing the complexity of thread management while maintaining efficient parallelism.

OpenCilk-Based Parallelization

1. The dataset is divided into chunks similarly to the previous implementations, but the parallel processing is handled using `cilk_for` constructs:
   - Each task is dynamically spawned to handle a subset of the dataset.
   - A serial k-NN search is performed for each task, and results are merged back into global arrays.
2. The use of dynamic task creation allows for efficient load balancing, especially when the dataset contains irregular patterns.
3. After parallel processing, a verification step is performed to ensure all k-NN results are accurate, especially for data points near the splitting hyperplane.

OpenCilk's dynamic task scheduling simplifies the parallelization, allowing the algorithm to scale well with the number of available cores without explicit thread management.

Tests

To run all the approximate functions and verify that the results are correct, we generate a random $n \times n$ matrix and store it in the `data/tmp` folder. Then for verification we use one of the exact fucntion which we already test that it computes the results correctly. Building, executing, and testing all exact methods can be easily achieved by running the following script from the project folder:

```
./run_knn.sh 1 4 null null null 100
```

The second parameter specifies the number of threads to be utilized. In this example, we calculate and evaluate the 100-nearest neighbors (100-NN) of the random matrix. We then compare our output matrices (indices and distances) against the expected results that the exact function gave as to validate accuracy.

The results of running the 100-NN for all the approximate implementations on our machine for random data that have `data_length = 118148`, `dim = 176` are as follows:

- **Running knn_exact_pthread with 4 threads:**
  - Running time: 24.461436 seconds, Queries per second: 4829.969917
- **Running knn_approx_serial:**
  - Running time: 15.675810 seconds, Queries per second: 7536.963002
- **Running knn_approx_pthread:**
  - Running time: 10.854054 seconds, Queries per second: 10885.149457
- **Running knn_approx_openmp with 4 threads:**
  - Running time: 7.843689 seconds, Queries per second: 15062.810369
- **Running knn_approx_opencilk with 4 threads:**
  - Running time: 8.782857 seconds, Queries per second: 13452.114728

```
Compare knn_approx_serial results with expected:
Approximate: Neighbors Hit Rate: 58.317932%, k-NN Average Distances Rate (approx/exact): 1.008012
Compare knn_approx_pthread results with expected:
Approximate: Neighbors Hit Rate: 25.742374%, k-NN Average Distances Rate (approx/exact): 1.020895
```

The comparison of the *OpenMP's* and *OpenCilk's* functions with the expected results gives us exactly the same results. This happens because to every parallel approximate method we implement the exact same code, which meaning that for the same dataset we will have the same or nearly the same results.

By increasing the number of threads we observe that although using more threads can speed up the computation by parallelizing the work, it can also introduce additional inaccuracies due to the need for merging the results of smaller sub-sets. To practically see the influence that the total number of threads have over accuracy and time we run for a **larger problem** of `data_length = 129546`, `dim = 191` the command:

```
./run_knn.sh 1 8 null null null 100
```

- **Running knn_exact_pthread with 8 threads:**
  - Running time: 33.003033 seconds, Queries per second: 3925.275595
- **Running knn_approx_serial:**
  - Running time: 18.589969 seconds, Queries per second: 6968.596881
- **Running knn_approx_pthread with 8 threads:**
  - Running time: 5.038039 seconds, Queries per second: 25713.576255
- **Running knn_approx_openmp with 8 threads:**
  - Running time: 4.826676 seconds, Queries per second: 26839.588984
- **Running knn_approx_opencilk with 8 threads:**
  - Running time: 5.057512 seconds, Queries per second: 25614.570959

```
Compare knn_approx_serial results with expected:
Approximate: Neighbors Hit Rate: 58.059330%, k-NN Average Distances Rate (approx/exact): 1.007634
Compare knn_approx_pthread results with expected:
Approximate: Neighbors Hit Rate: 13.385284%, k-NN Average Distances Rate (approx/exact): 1.030374
```