KD-TREE

1. Introduction

This document describes the solution details of KD-tree project. In Section 2 we describe the basic data structures and components used in the project. In Section 3 we describe the algorithms used to build and search KD-trees. In Section 4 we describe some of the design decisions we made during this project. We conclude the document by describing some of the potential improvements we could have done in this project.

2. Basic components

The various components used in this project are as follows:

- **Sample matrix**: Input samples are represented as a matrix $M \in R^{m\times n}$, where we have m samples and n dimensions and M_{ij} represents the value for i^{th} sample in j^{th} dimension. We use Eigen library [1] to represent this matrix.

- **Segment**: Segment is the logical partitioning of samples that is used by the KD-tree algorithm during tree construction. An example of segments is shown in Figure 1. The six samples are stored in memory as an Eigen matrix and shared by all segments. Each segment contains a vector of sample ids that are present in the segment. The first segment contains all the sample ids.

  After a split, during tree building process, we get two almost equal segments which contain a reference to the main matrix and have a vector of sample ids that indicate the samples from matrix which belong in the segment. Partitioning samples this way allows us to develop recursive or parallel algorithms to build trees efficiently.

- **Split**: This contains the information that we need to split samples into segments. This is created during tree building process and used when we are searching for a query in the tree. Every internal node in the tree contains this split object. Split contains the following details (Figure 2):
  1. Split dimension (d_{t+1}): This the dimension along which we will be splitting the samples at $(t+1)^{th}$ node during the search.
  2. Split point: This is the point that is used to split the samples in the segment into almost equal size. If the number of points in the segment are odd then the split point is the one in middle otherwise, it is the average of the 2 points in the middle. This is the point in the center of Figure 2.

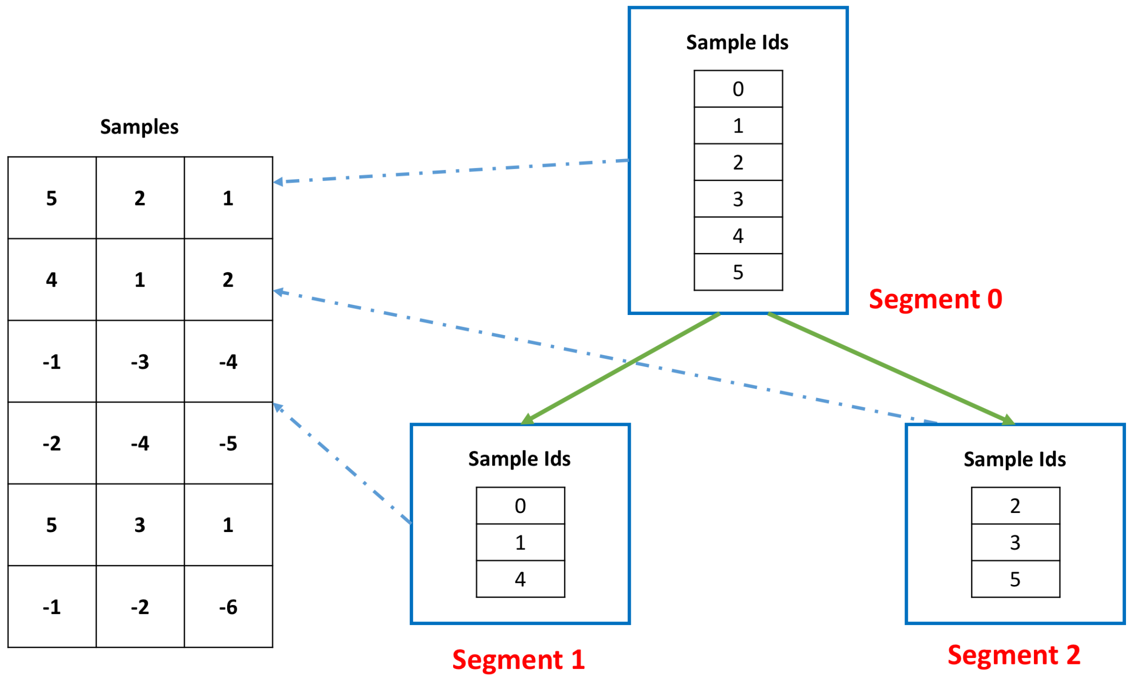

FIGURE 1. Example of how samples are stored in an Eigen matrix and then shared between segments.

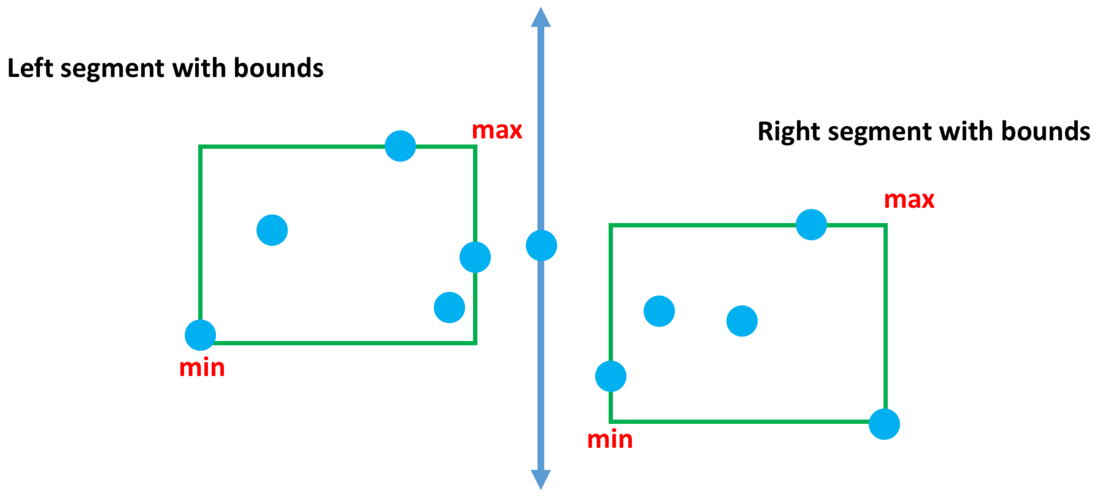

FIGURE 2. Example of a split, with split point, left and right bounding boxes.

(3) Bounds: These define the boundaries of left and right segments generated after the current segment is split. We store: (i) min point, which is the min value in all dimensions for samples in the segment; and, (ii) max point, which is the max value in all dimensions for samples in the segment.

- **Split Dimension Selector**: This is used to select the dimension that will be used to split the samples next. The selector supports 2 methods:
  (1) Cycle through axis: This method cycles through dimensions.
  $$d_{t+1} = (d_t + 1)\%n$$
  (2) Highest range axis: This method selects a dimension that has the maximum spread of points.
  $$d_{t+1} = \arg\max_j(\text{max\_value\_along j - min\_value\_along j})$$
- **Split Point Selector**: This is used to select the threshold that will be used to separate points. These methods return the same value but the complexity of finding them is different depending on the method. This selector supports 2 methods:
  (1) Median: Sorts the values for dimensions we are splitting across and picks the median. This has a complexity of $O(nlogn)$.
  (2) Median of median: This uses a selection algorithm that picks the k^{th} biggest item with $O(n)$ complexity.
- **KD-tree**: The KD-tree object is represented as an array of nodes. A node can either be an internal node containing information of how to split samples or a leaf node that contains id for a sample. The array is indexed from 0, so for internal nodes at index k, the left child is at $2k + 1$ and the right child is at $2k + 2$.

3. Algorithms

I implemented three algorithms in this project. First two of them are related to building KD-tree and the third one is about searching a query in the tree. Of the two tree-building algorithms one uses recursion (Algorithm 1) and another one is a parallel (Algorithm 2). The third algorithm is used to search for a query in a KD-tree (Algorithm 3). The details are as follows:

- **Recursive tree building algorithm**: We first load samples into an Eigen matrix and then initialize KD-tree as an array of nodes. We then create a big segment containing all the samples and call the function to build the tree. This recursive function gets the dimension to split by and splits the big segment into 2 segments of almost equal parts using the split point. It inserts the split point into KD-tree and then calls the recursive function again with the smaller segments. If the

Algorithm 1 Tree building algorithm (recursive)

```
1:  Load samples
2:  Initialize KD-tree as an array of nodes of size $2^{\lfloor log_2(\text{num samples}) \rfloor + 1}$
3:  Create Segment $S_0$ which contains all samples
4:
5:  BuildTreeRecursive($S_0$)
6:
7:  procedure BuildTreeRecursive(Segment S)
8:      if S contains just one sample then
9:          Create a leaf node with one sample and insert it to the KD-tree
10:     else
11:         Get dimension to split by using dimension selector.
12:         Get the split threshold using split point selector.
13:         Insert an internal node with split information into KD-tree.
14:
15:         Generate the segments:
16:         $S_l \leftarrow$ all samples less than split threshold; and
17:         $S_r \leftarrow$ all samples greater than the split threshold
18:
19:         BuildTreeRecursive($S_l$)
20:         BuildTreeRecursive($S_r$)
```

segment contains just one sample then the function inserts a leaf node with that segment into the tree.

- **Parallel tree building algorithm**: This is similar to the recursive algorithm in terms of functionality but doesn't have the additional overhead of maintaining system stack during recursion. Instead, it uses a thread-safe queue that can be used like a stack. In addition to this, the algorithm is multithreaded which allows for faster tree building on a big dataset when enough resources are available.

- **Query search algorithm**: This algorithm is used to search for a query in a KD-tree. Like the iterative (parallel) version, this uses a stack to track the sample space that needs to be explored. We start by loading the samples on which the tree was built and the KD-tree. We then initialize the stack with the root node of KD-tree. In addition to this, we also initialize 2 variables - the nearest neighbor discovered so far during the search and the euclidian distance from it. The nearest neighbor is initialized to the first node and its distance is set to Infinity. We then start the search until the stack is empty. During the search, if we reach the leaf node, we check if the sample at that node is closer than the nearest neighbor found so far. If it is, we update the nearest neighbor and the distance from it.

Algorithm 2 Tree building algorithm (parallel)

```
Load samples
Initialize KD-tree as an array of nodes of size $2^{\lfloor log_2(\text{num samples})\rfloor+1}$
Create Segment $S_0$ which contains all samples

BuildTreeParallel($S_0$)

procedure BuildTreeParallel(Segment S)
    Create a thread pool $P$
    Initialize a thread-safe queue Q
    Push segment onto queue: $Q \leftarrow Q + S$
    while Q is not empty do
        $p \leftarrow$ next free thread in P
        $S_c \leftarrow$ Last item from $Q$            // mimics stack
        $p \rightarrow$ ProcessSegment($S_c$, $Q$)

procedure ProcessSegment(Segment S, Queue Q)
    if S contains just one sample then
        Create a leaf node with one sample and insert it to the KD-tree
    else
        Get dimension to split by using dimension selector.
        Get the split threshold using split point selector.
        Insert an internal node with split information into KD-tree.

        Generate the segments:
        $S_l \leftarrow$ all samples less than split threshold; and
        $S_r \leftarrow$ all samples greater than the split threshold

        $Q \leftarrow Q + S_r$
        $Q \leftarrow Q + S_l$

```

An important aspect of this search algorithm is the approach used to find the distance between the query and the bounds identified during the KD-tree building process. A bound is represented using 2 points $(point_{min}, point_{max})$, where $point_{min}$ and $point_{max}$ and n-dimensional vectors, similar to query q. The split distance between the two is defined as follows:

$$d(i) = \begin{cases} point_{min}[i] > q[i] & (point_{min}[i] - q[i]) \\ point_{max}[i] < q[i] & (q[i] - point_{max}[i]) \\ otherwise & 0 \end{cases}$$

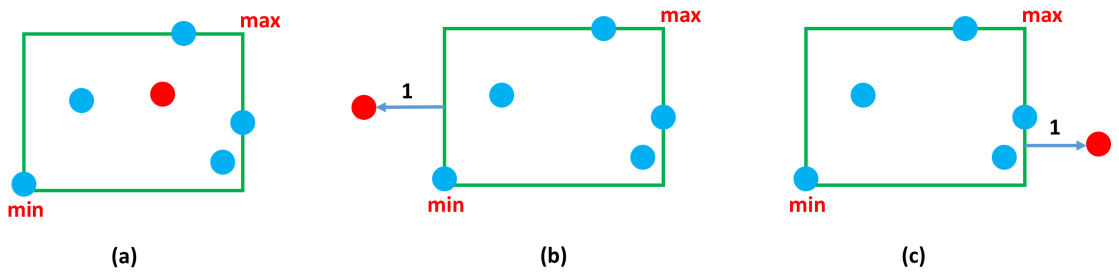

FIGURE 3. An example of a distance between query (red point) and the bounding box is shown. Case (a) has the query inside the bound so it has a distance of 0. In case (b) and (c) the query is about 1 unit away from the bounding box. Hence, is at a distance of 1.

$$distance(bound, q) = \sum_{i=0}^{n-1} d(i)^2$$

This function keeps the distance between two points to be 0 if they are within the range otherwise we add the squared distance from the region. This is similar to the approach discussed in [2]. An example of this is shown in Figure 3. Using this definition of distance, the search algorithm continues to decide if a branch needs to be explored in the search tree.

4. Implementation/Design

I made the following design decisions during the implementation of the algorithms.

- **Functional design**: During the design of classes I tried to follow the concepts of functional programming as much as possible. This decision allowed me to reuse code that was used to implement recursion based solution to a mutli-threaded one. It also allowed me to test different components in a modular way resulting in cleaner code and isolation of issues. I achieved functional aspects by making most of my classes immutable, i.e, the class attributes are all private and cannot be changed once they are created. The attributes are available, only if needed, through getter methods. In addition to immutable classes I made sure all the functions are pure, i.e, the value returned by the function is just dependent on the input parameters to the function and not any variables outside of it.

- **Test driven development**: I took a test-driven approach while implementing the modules, which mean any module that I added was thoroughly unit tested. Anytime there were dependencies between the modules I was able to test them by mocking (not gmock) the other modules. I also wrote integration tests to test the end-to-end correctness of the implementation. The datasets for the integration tests were generated using a python script, where the samples to build the tree and queries were generated randomly and the labeled data was generated by doing

Algorithm 3 Query searching algorithm

```
1:  Load samples
2:  Load KD-tree as an array of nodes
3:
4:  procedure SearchQuery(Query q)
5:      $N \leftarrow$ First node in KD-tree
6:      Push node onto queue: $Q \leftarrow Q + N$
7:
8:      Initialize current nearest neighbor $N'$ and Euclidean distance from that neighbor $d'$
9:      $N' \leftarrow N$
10:     $d' \leftarrow \infty$
11:
12:     while Q is not empty do
13:         $N_i \leftarrow$ Last item from $Q$            // mimics stack
14:         if $N_i$ is a leaf node then
15:             $d \leftarrow distance(N_i, q)$
16:             if $d \leq d'$ then
17:                 $d' \leftarrow d$
18:                 $N' \leftarrow N_i$
19:         else
20:             Get current split: $P \leftarrow N.split$
21:             $d_{left} \leftarrow distance(P.leftBounds,\ q)$
22:             $d_{right} \leftarrow distance(P.rightBounds,\ q)$
23:
24:             $leftIsCloser \leftarrow d_{left} \leq d'$
25:             $rightIsCloser \leftarrow d_{right} \leq d'$
26:
27:             if $leftIsCloser$ && $rightIsCloser$ then
28:                 Order in which to explore next side is decided by which is the closer side.
29:                 if $d_{left} \leq d_{right}$ then
30:                     $Q \leftarrow Q + P.rightNode$
31:                     $Q \leftarrow Q + P.leftNode$
32:                 else
33:                     $Q \leftarrow Q + P.leftNode$
34:                     $Q \leftarrow Q + P.rightNode$
35:             else if $leftIsCloser$ then
36:                 $Q \leftarrow Q + P.leftNode$
37:             else if $rightIsCloser$ then
38:                 $Q \leftarrow Q + P.rightNode$
```

a n^2 search find the nearest neighbor and the distance from the neighbor. I then verified the correctness of my implementation by checking we are able to find nearest neighbors correctly (KDTreeHandlerTest.cpp).

- **Storing sample ids instead of samples**: In the built model, I decided to keep just the sample ids instead of the entire sample. From an implementation point of view either option was viable but I decided to keep just the sample ids as it avoids duplication of samples. The drawback of this approach is that the samples dataset is tied to the model.

- **Squared distance instead of Euclidean**: In the query search algorithm I use squared distance to maintain the nearest neighbor discovered so far instead of Euclidean distance. I use Euclidean distance only in the final step while writing the results. I did this as squared distance metric doesn't change the ordering of the nearest neighbors and is computationally faster as it doesn't have to calculate the square-root required in the Euclidean distance. This is similar to the approach taken in KD-tree in scikit-learn [3].

5. Implementation/Design

In this document I described the various data structures and components, the algorithms used to build and search KD-trees and some of the design decisions I made during the project. If I had more time, I would have liked to do the following things:

- **Store multiple samples in leaf nodes**: Right now the KD-tree continues to build the tree until leaf contains just one sample. Alternatively, we can stop the iteration when we have reached a threshold, for example, stop building tree when a node contains 10 samples. During the search process we can then do a linear search of the 10 samples to find the nearest neighbor. This can be implemented in my design without too many changes.

- **Evaluation of parallel algorithm**: I have currently evaluated the correctness of my parallel implementation. I would like to spend more time trying different datasets to see when using parallel approach beats the recursive approach. For small datasets, there might not be too much difference between the two implementation. In addition to the size of the dataset, I might also want to see if there are specific parameters in TBB [4] (currently used for parallelization) library that can help me improve the performance.

- **More documentation**: I have added comments in places where I felt code needed clarity. But I would have done another round of documentation making the code more clear.

References

[1] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[2] Nearest neighbor search using kd trees. http://programmizm.sourceforge.net/blog/2011/nearest-neighbor-search-using-kd-trees, 2011.

[3] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[4] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.