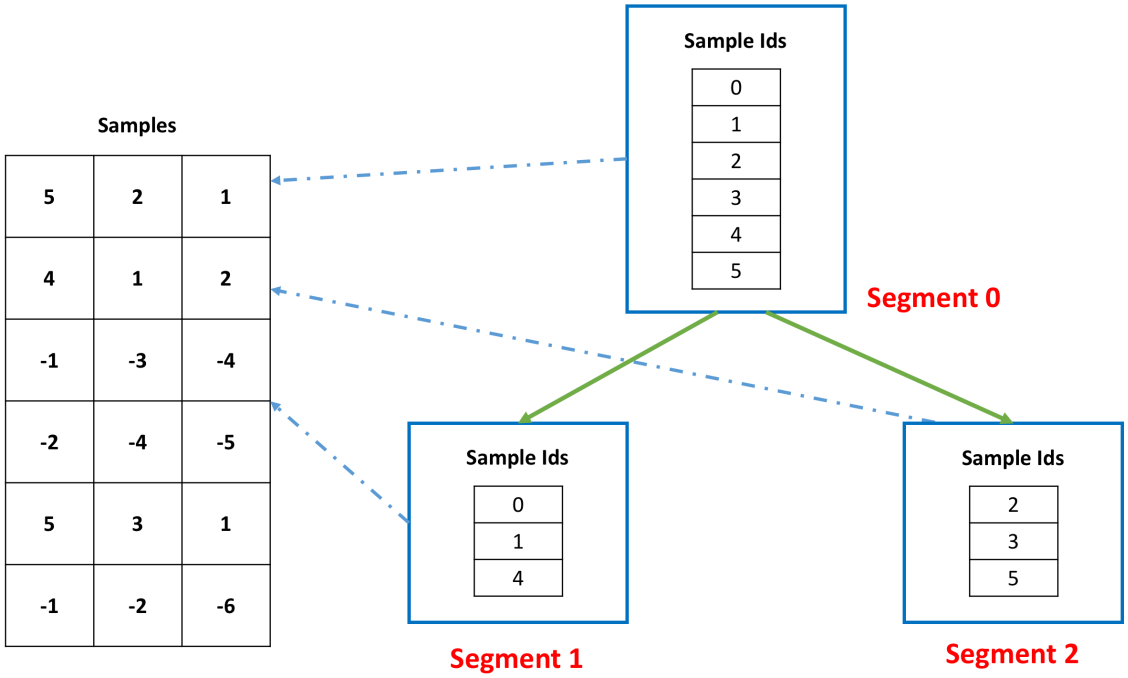

FIGURE 1. Example of how samples are stored in an Eigen matrix and then shared between segments.

KD-TREE IMPLEMENTION AND PERFORMANCE

1. 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. Example of segments is shown in Figure 1. The six samples are stored in memory as a 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 contains a reference to the main matrix and have a vector of sample ids

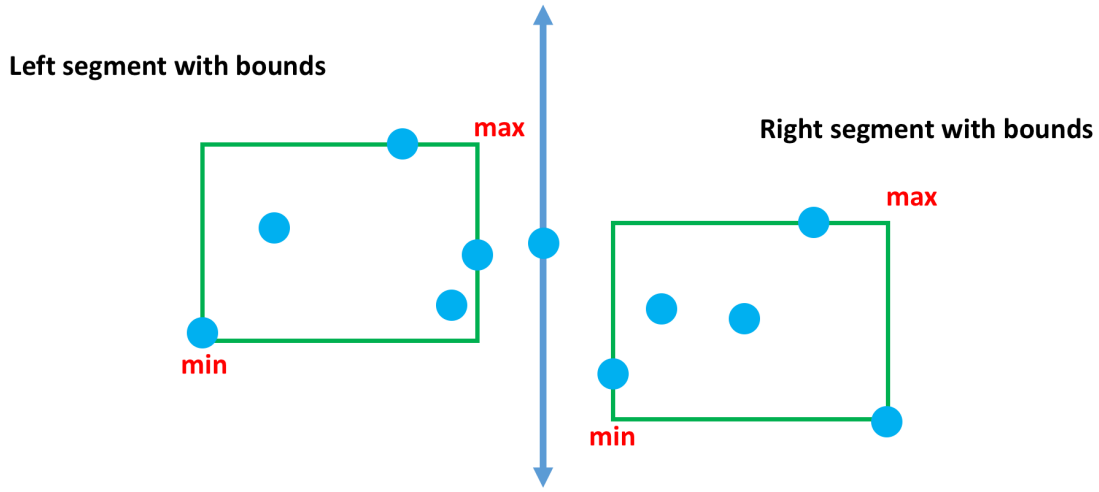

FIGURE 2. Example of a split, with split point, left and right bounding boxes.

that indicate the samples from matrix with 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 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 centre of Figure 2.
  (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 $2k + 2$.

2. 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)

Recursive tree building algorithm: We first load samples into a Eigen matrix and then initialize KD-tree as a 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 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 recursive algorithm in terms of functionality but doesn't have the additional overhead 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.

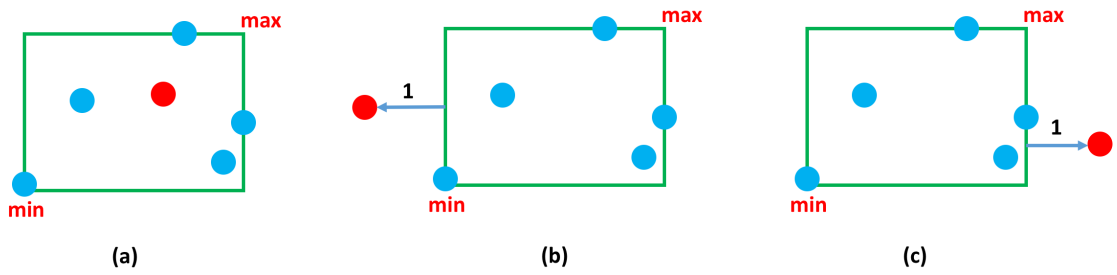

FIGURE 3. Example of a distance between query (red point) and bounding box is shown. Case (a) has the query inside the bound so it has a distance of 0. In case (b) and (c) the query is about 1 unit away from the bounding box. Hence, is at a distance of 1.

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 leaf node, we check if the sample at that node is closer than the nearest neighbor found so far. If it is, we update nearest neighbor and the distance from it.

Important aspect of this search algorithm is the approach used to find the distance between 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:

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

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

This function keeps the distance between two points to be 0 if they are within the range other wise 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.

3. IMPLEMENTATION/DESIGN

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

Algorithm 1 Tree building algorithm (recursive)

```
1: Load samples
2: Initialize KD-tree as an array of nodes of size 2^(⌊log2(num samples)⌋+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 ← all samples less than split threshold; and
17:        S_r ← all samples greater than the split threshold
18:
19:        BuildTreeRecursive(S_l)
20:        BuildTreeRecursive(S_r)
```

- **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 to test different components in a modular way resulting in cleaner code and isolation of issues. I achieved function aspects by making most of my classes immutable, i.e, the class attributes cannot be changed once they are created. In addition 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.

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.

Algorithm 2 Tree building algorithm (parallel)

```
1: Load samples
2: Initialize KD-tree as an array of nodes of size 2^{⌊log2(num samples)⌋+1}
3: Create Segment S_0 which contains all samples
4:
5: BuildTreeParallel(S_0)
6:
7: procedure BuildTreeParallel(Segment S)
8:     Create a thread pool P
9:     Initialize a thread-safe queue Q
10:    Push segment onto queue: Q ← Q + S
11:    while Q is not empty do
12:        p ← next free thread in P
13:        S_c ← Last item from Q          // mimics stack
14:        p → ProcessSegment(S_c, Q)
15:
16: procedure ProcessSegment(Segment S, Queue Q)
17:    if S contains just one sample then
18:        Create a leaf node with one sample and insert it to the KD-tree
19:    else
20:        Get dimension to split by using dimension selector.
21:        Get the split threshold using split point selector.
22:        Insert an internal node with split information into KD-tree.
23:
24:        Generate the segments:
25:        S_l ← all samples less than split threshold; and
26:        S_r ← all samples greater than the split threshold
27:
28:        Q ← Q + S_r
29:        Q ← Q + S_l
30:
```

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 ← First node in KD-tree
6:     Push node onto queue: Q ← Q + N
7:
8:     Initialize current nearest neighbor N′ and Euclidean distance from that neighbor d′
9:     N′ ← N
10:    d′ ← ∞
11:
12:    while Q is not empty do
13:        N_i ← Last item from Q            // mimics stack
14:        if N_i is a leaf node then
15:            d ← distance(N_i, q)
16:            if d ≤ d′ then
17:                d′ ← d
18:                N′ ← N_i
19:        else
20:            Get current split: P ← N.split
21:            d_left ← distance(P.leftBounds, q)
22:            d_right ← distance(P.rightBounds, q)
23:
24:            leftIsCloser ← d_left ≤ d′
25:            rightIsCloser ← d_right ≤ 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 ≤ d_right then
30:                    Q ← Q + P.rightNode
31:                    Q ← Q + P.leftNode
32:                else
33:                    Q ← Q + P.leftNode
34:                    Q ← Q + P.rightNode
35:            else if leftIsCloser then
36:                Q ← Q + P.leftNode
37:            else if rightIsCloser then
38:                Q ← Q + P.rightNode
```