

KD-TREE IMPLEMENTATION AND PERFORMANCE

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 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.

$$d_{t+1} = (d_t + 1) \% n$$

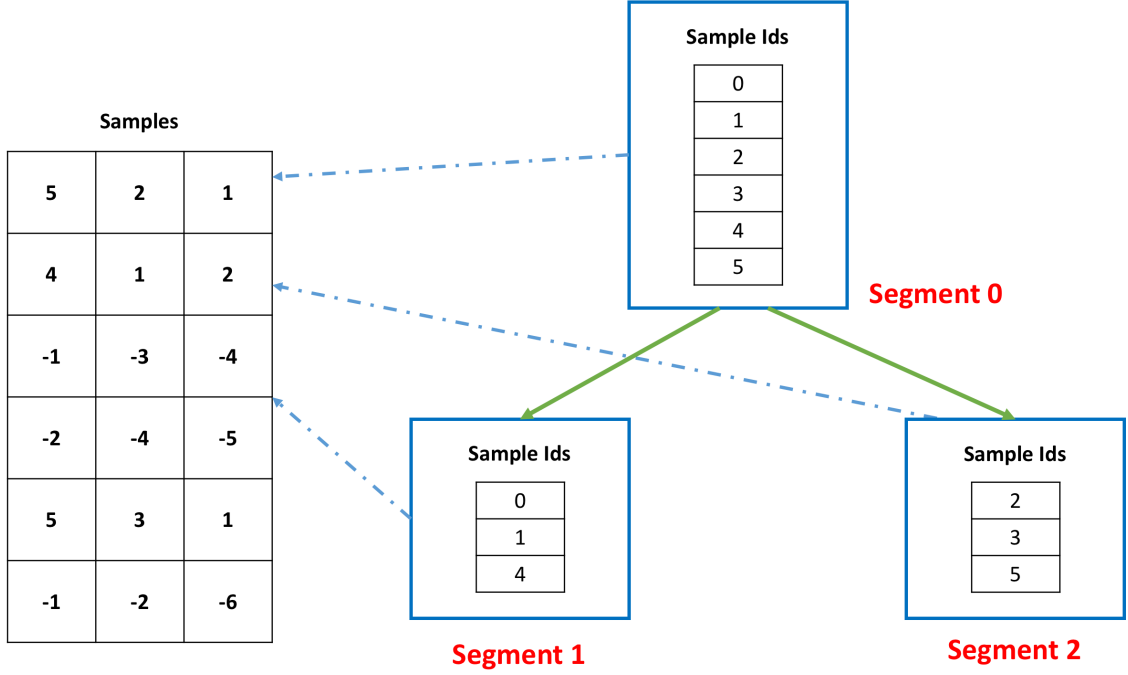


FIGURE 1. Example of how samples are stored in an Eigen matrix and then shared between segments.

- (2) Highest range axis: This method selects a dimension that has the maximum spread of points.

$$d_{t+1} = \arg \max_j (\max_value_along\ j - \min_value_along\ j)$$

- **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 $O(n \log n)$.
 - (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 $2k + 2$.

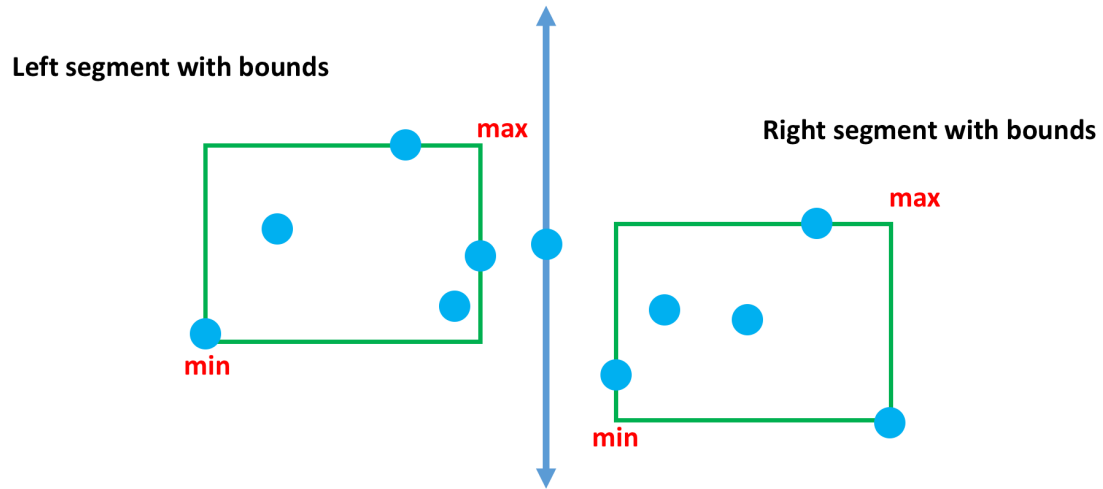


FIGURE 2. Example of a split, with split point, left and right bounding boxes.

REFERENCES

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

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( $S_r$ )

```

Algorithm 2 Tree building algorithm (parallel)

```

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: 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:  $Q \leftarrow Q + S$ 
11:  while  $Q$  is not empty do
12:     $p \leftarrow$  next free thread in  $P$ 
13:     $S_c \leftarrow$  Last item from  $Q$  // mimics stack
14:     $p \rightarrow$  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:      $S_l \leftarrow$  all samples less than split threshold; and
26:      $S_r \leftarrow$  all samples greater than the split threshold
27:
28:      $Q \leftarrow Q + S_r$ 
29:      $Q \leftarrow Q + S_l$ 
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:    $N \leftarrow$  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:       $d \leftarrow \text{distance}(N_i, q)$ 
16:      if  $d \leq d'$  then
17:         $d' \leftarrow d$ 
18:         $N' \leftarrow N_i$ 
19:    else
20:      Get current split:  $P \leftarrow N.\text{split}$ 
21:       $d_{\text{left}} \leftarrow \text{distance}(P.\text{leftBounds}, q)$ 
22:       $d_{\text{right}} \leftarrow \text{distance}(P.\text{rightBounds}, q)$ 
23:
24:       $\text{leftIsCloser} \leftarrow d_{\text{left}} \leq d'$ 
25:       $\text{rightIsCloser} \leftarrow d_{\text{right}} \leq d'$ 
26:
27:      if  $\text{leftIsCloser} \ \&\& \ \text{rightIsCloser}$  then
28:        Order in which to explore next side is decided by which is the closer side.
29:        if  $d_{\text{left}} \leq d_{\text{right}}$  then
30:           $Q \leftarrow Q + P.\text{rightNode}$ 
31:           $Q \leftarrow Q + P.\text{leftNode}$ 
32:        else
33:           $Q \leftarrow Q + P.\text{leftNode}$ 
34:           $Q \leftarrow Q + P.\text{rightNode}$ 
35:      else if  $\text{leftIsCloser}$  then
36:         $Q \leftarrow Q + P.\text{leftNode}$ 
37:      else if  $\text{rightIsCloser}$  then
38:         $Q \leftarrow Q + P.\text{rightNode}$ 

```
