KD Tree Library

Author: Mohana Prasad
Time Spent: 5-6 hrs

The KD Tree library has methods for building the KD tree, searching for nearest element, saving/loading the tree, delete/clearing the tree. It also has the option to choose different axis selection methods and split position methods at runtime. This has been implemented using c++11 function pointers. Other C++11 constructs were also extensively used. The library also supports double, floating point and integer data point values. This has been implemented using function and class templates feature of C++. STL vectors were used to represent K dimensional data points. DoublePoint, FloatPoint and IntPoint typedefs are used for representing our data points.

“Tests.cpp” has tests for testing each module and for testing different algorithms for axis selection (maximum range, cycle/round robin) and split position selection (median, median of medians).

Implementation Details and Design Decisions:

The KDNode class represents a node in our tree. An internal node stores the split position, split axis and pointers to its children. If it is a leaf node then, it stores a vector of indexes to the data points. The boolean variable leaf_node indicates whether it is a leaf node. The number of points to be stored in a leaf node can be decided at runtime when we build the tree.

The KDTree class stores the entire list of data points and a pointer to the root KDNode. An alternative design would have been to store the data points themselves at the leaf nodes instead of storing indexes. For small K (dimension) that option might work well, but as K increases we may have to copy and manipulate large vectors while building the tree as we would have to pass the entire data points list down the tree. But as we store only the indexes, we manipulate, copy and pass down lesser items while we build the tree. We would be able to realize the gains of this decision when K grows large (maybe 10 or more). One drawback with storing indexes is that, when we save our tree we have to separately store the indexes at the leaf nodes along with the list of data points. But, if we had chosen to directly store the data points at the leaf nodes, we need not have to waste extra memory in storing the indexes. But since indexes are integers, the overhead in memory is going to be insignificant.

Build function: The build function gets the data points, sets the correct axis and split position functions pointers, makes sanity checks on the dimensions of the data points and calls the recursive build_tree function that builds the tree.

The recursive build_tree and nearest_neighbor_search functions are implemented as described in https://en.wikipedia.org/wiki/K-d_tree

The median split position algorithm is O(N logN) where N is the number of points as it sorts the entire list on the given axis and finds the median. The median of median method is recommended which is O(N). We use the n_element method from C++ standard template library for doing this.

The maximum range axis selection and cycle based axis selection were straightforward implementations. Cycle based is O(1) and range based is O(NK).

Saving a tree starts off by first saving the list of data points. We subsequently store the nodes of the tree in in-order traversal where we save the axis and split position for non leaf nodes and the size and list of indexes for leaf nodes. (note: leaf can have indexes to more than one point. We can set it when we build the tree).

Loading a tree just loads things in the order in which it was saved.

Complexity Analysis:
I would show for cycle/round robin axis selection and median of medians split position algorithm. Complexity for other cases can be found by plugging in the values from the previous section.

Build tree:
Given a list of ‘P’ points at a node,

For cycle/round robin axis selection: O(1)
For median of medians split position: O(P)
For making two lists: O(P)
Total: O(P)

While building, the first root node gets N points, then the next level nodes get N/2 each and so on.
So, total time complexity = 1*N + 2*N/2 + 4*N/4 + ….
= **O(N log N)** where N is the number of data points.
In case of max range axis selection, it will be O(KN log N)
Storage: O(NK) for storing the data + O(N) for the tree = O(NK) space

Search:
Computing distance between two points is O(K). In the worst case it looks like we may have to search the entire list which will make it O(NK).
But in the average case, we will be pruning a lot and would end up doing in O(log N) as we would come down the tree and will look at only a constant number of leaf nodes.

Suggested Improvements:

To speed up search more, we can go for approximate methods where we look at only a limited number of leaf nodes and then stop.

We can also throw away the restriction of splitting parallel to the axis hyperplanes and try methods that split in directions with maximum variance.