Uber Coding Assignment

The implementation is that of kd tree in C++. Some of the design parameters are described herewith.

Code Structure

- The **class node** is a class and hence holds information related to one single data point in the space, which include, the index of the data point in the input file, the data itself, collision if any, and the pointers to it's left and right. By default, the pointers are initialized to nullptr.
- The **class kdtree** is a class and hence holds methods and members pertaining to the kdtree. The only member is the root node or the head node which can used anywhere to access anything or manipulate the tree as and when required.

  The important class methods include,
  - **search_kdtree**, which finds the nearest neighbor to the given point in the tree. Just doing traversal would only result in the approximate neighbor, as in the higher dimension, one could be nearer but still be in another branch of th tree and hence inaccessible. The described problem is already solved and there exists an established method as described in https://web.stanford.edu/class/cs106l/handouts/assignment-3-kdtree.pdf. This file has been used as a guide and inspiration to learn more about kd trees. The complexity for the suggested method to find the accurate nearest neighbor, is O(log n) given that the tree is balanced and the data is randomly enough. The worst can be that the whole tree is searched in cases such completely skewed tree.
  - **insert_kdtree**, which inserts the a nodes one at a time in the given order and this has a complexity of O(log n), but in high dimensions, it is important to find a good splitting axis and a splitting position in order to create a balanced tree, as ceasing to do so, affects the search. Common methods of splitting axis include range (or) variance based which has been adopted to be the default case unless the user specifies some other strategy. One other heuristic is provided as well, which returns the dimension with the least variance. The other way provided is where the user writes a function which specifies the dimension or axis of splitting thereby overriding the split axis function. One other case is the split position and the common way is to take the median as this maintains equal weight for the tree, but it is possible to do it at any position as well such as 0.75 of the weight to be distributed to one direction and remaining to the other. This could be used as per the users requirements as and when required. It is even possible to develop a function that intelligently varies these parameters to maintain good balance of the tree, without relying on the conventional central methods. In order to find median(middle element) or the element at the 0.75 position or 0.25 position, it is necessary to sort and that adds to the complexity heavily. The worst case complexity is O(kn logn) where k is the dimension and n is the count on the data points.
  - There are other few methods which facilitate usage and debugging such as print function, checking if an element is present in the tree or not and so on.
  - De serialization and serialization is done in binary files with format .kd. This maintains the index as the first element and the following as the data, which are all comma separated. Data is differentiated from one another using '\n' command and the end of file is depicted using '/'. The same strategy is used for parsing as well.
- A **namespace kdspace** has been created which handles some of the interfacing function and those required for necessary functions of the kd tree. These include, parsing a file and creating a vector of vector of data, finding the split position, split axis, identifying the node to be inserted first so that the insert_kdtree function can be used.

Wall Time

Time as measured for the provided dataset, **sample_data.csv (1000x3).**

- Parse the data from sample_data.csv with index and data alongside -> **0.013s**
- Building the balanced tree from the parsed data by finding the highest range split axis and median split position in each recursion -> **0.024s.**
- Reconstructing a tree from the deserialized .kd file → **0.05s**

Memory leaks

Smart pointers have been used as required and hence, when the counter hits zero(shared_ptr), it is automatically destroyed. The memory leak was checked using Valgrind and it returned the data as 0 bytes of memory leak as can be seen in the attached image.

```
Stroring the tree to disk as 'tree.kd'.
Loading the tree from the serialized file 'tree.kd'.
Parsing the 'query_data.csv' file...
Parsed
Querying to get index and Euclidean distance and stroring the requested data to disk at 'output.txt'.
Done!
==11614==
==11614== HEAP SUMMARY:
==11614==     in use at exit: 0 bytes in 0 blocks
==11614==   total heap usage: 47,056 allocs, 47,056 frees, 7,800,007 bytes allocated
==11614==
==11614== All heap blocks were freed -- no leaks are possible
==11614==
```

Compilation

Complied using g++ and does not give any error with the compilation features namely, -Wall -Wextra -Werror.

Testing

Necessary tests were performed to make sure the code does not crash or go unstable. Necessary errors and exceptions were included. Due to time constraints, the same could not be included in the cmake as Ctests.

Updates

Intelligent heuristics can be created to balance the tree dynamically using th provided interfaces, thus making the tree more balanced and hence making the search more faster and efficient.

More interfaces can be created for the code other than the '-d' and '-b' commands, thereby making it much more comfortable and flexible to use.

Different serialization techniques could be explored to make it faster and effective.