Group 2
Vishalsinh Gadhvi
Tommy Giang
Maryam Jamilurrehman

CMPE 297 Group 2 Project Report

The GitHub repository for our project code is here:
https://github.com/TGiang1/CMPE297_Group_Project

I. Project Description

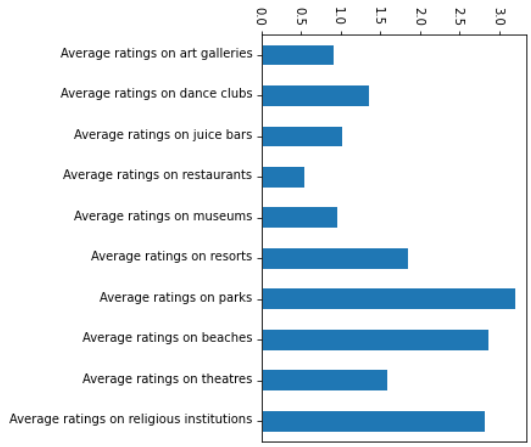
Our project topic revolves around data analysis, and in particular, nearest neighbor cost/runtime analysis of multi-dimensional data. The focus will be on using multi-dimensional data structures on indexing a large multidimensional dataset, and using the constructed data structure to explore runtime cost of nearest neighbor (NN) queries. The data structures that we will be using in our exploration are: KD-Tree, Ball-Tree, and M-Tree. We will compare with the standard "brute force" approach for nearest neighbor query searches as well. Lastly, we do additional exploration of how varying parameters (such as leaf_size, max_node_size, number of neighbors to query, and number of dimensions of the dataset) affect the runtime of the NN queries.

II. Dataset

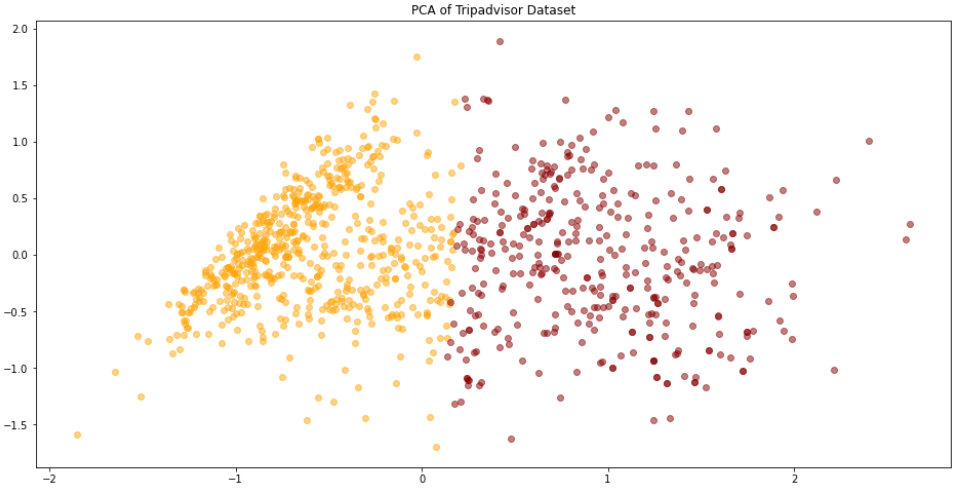
A. Details

- "tripadvisor_review.csv" file from the uci machine learning database: https://archive.ics.uci.edu/ml/datasets/Travel+Reviews
- Includes user reviews on destinations in 10 categories mentioned across East Asia
- Values range from 0 to 4
- The dataset has 980 instances (users), and 10 dimensions (ratings) after dropping the user ID column
- Preprocessing is simple – drop the user ID column and convert the dataframe into a 2 dimensional numpy array (with each row being a "data point" in 10 dimensions)

B. Data Statistics (Clustering Analysis)



[Figure 1]



[Figure 2]

For our initial analysis of the dataset to just understand our dataset a bit more before we proceed into the core exploration of our project, we decided to try and cluster (using Kmean module from sklearn.cluster) the users based on ratings into two groups (see Figure 2). We also found the ratings on parks, beaches, and religious institutions, and calculated the average ratings (see Figure 1) for each of the categories (dimensions). Beaches and religious institutions appear to be the highest amongst all.

III. Test Set

The main test set that we will use ("queries1.txt" in our repository) is a generated txt file that we import as a numpy array to use an input for querying our NN searches on our data structures.

The function generates random uniform values between 0 and 5, rounded to 2 decimals (figure 3).

```python
def generateQueries(numQueries, upperLimit=5, numDimensions=10):
    queriesArray = np.random.uniform(0, upperLimit, size=(numQueries, numDimensions))
    queriesArray = np.round(queriesArray, 2)

    return queriesArray
```

[Figure 3]

For our purposes, we generate “queries1.txt” that can be found in our repository. The shape is (50k, 10), for 50k queries, with each query being a point in 10 dimensional space (figure 4).

```
2.440 0.530 1.160 4.950 1.670 1.230 0.440 4.980 2.500 1.160
4.870 1.520 1.950 0.350 4.310 2.900 4.320 1.600 4.210 4.180
3.840 2.140 4.980 3.200 1.610 3.730 1.660 0.050 0.790 3.830
0.760 0.760 4.300 3.530 1.860 4.330 2.530 3.750 3.410 0.880
2.050 1.620 3.320 4.960 0.220 4.630 2.720 3.660 2.050 2.490
1.400 3.160 0.110 3.790 0.070 1.690 1.660 3.320 1.870 4.510
4.060 4.690 4.190 3.180 1.450 2.210 2.470 3.560 2.920 2.210
4.590 0.870 3.200 2.470 4.790 3.030 0.800 3.730 1.740 1.050
1.410 1.440 2.820 1.730 4.930 2.810 2.930 0.090 4.580 4.910
2.380 2.660 3.990 4.480 1.310 3.570 1.800 0.600 4.330 0.620
0.210 4.270 2.660 1.130 3.480 1.250 2.970 3.320 3.310 1.480
2.120 1.310 0.860 0.810 2.450 0.010 0.450 4.020 1.190 3.380
4.360 0.290 3.210 3.050 0.770 4.400 1.330 4.430 1.490 2.510
0.370 3.910 2.160 3.710 2.900 0.860 1.020 4.750 2.820 0.440
1.760 2.080 2.600 2.750 1.290 3.350 0.690 0.530 1.630 3.600
4.410 3.150 2.270 1.520 3.540 1.220 1.330 4.200 2.680 1.990
```

[Figure 4]

IV. Implementation Details

A. Workflow

Brute Force

Our brute force (see BruteForce folder’s main file “BruteForce.py”) procedure is as follows:

1. For each query (in queries1.txt), for each data point, calculate the distance between the query and the data point
    a. We use the euclidean distance formula
2. Find the k array indexes of the k minimum values
    a. We use np.argmin() and np.argpartition()

KD-Tree

Our KD-Tree (see KDTreeNN folder’s main file “KDTreeImplementation.py”) procedure is as follows:

1. Iterate through the parameter “leaf_size” and examine how the leaf size of the built KD-Tree affects the NN search (k=1, meaning searching for 1 neighbor per query) time for 50k queries (10 dimensions)

2. Select a reasonable leaf_size, and iterate through the number of nearest neighbors to search for (parameter k) and examine how this affects the NN search time for 50k queries (10 dimensions)
3. Select a reasonable leaf_size and k, and examine how the number of dimensions of the dataset (from 2 to 10) affect the NN search time for 50k queries

Ball-Tree

Our Ball-Tree (see BallTreeNN folder's main file "BallTreeImplementation.py") procedure is the same as the KD-Tree procedure:

1. Iterate through the parameter "leaf_size" and examine how the leaf size of the built Ball-Tree affects the NN search (k=1, meaning searching for 1 neighbor per query) time for 50k queries (10 dimensions)
2. Select a reasonable leaf_size, and iterate through the number of nearest neighbors to search for (parameter k) and examine how this affects the NN search time for 50k queries (10 dimensions)
3. Select a reasonable leaf_size and k, and examine how the number of dimensions of the dataset (from 2 to 10) affect the NN search time for 50k queries

M-Tree

Our M-Tree (see MTreeNN folder's main file "MTreeImplementationNew.py") procedure is as follows:

1. Examine run-time procedure of NN queries with reasonable leaf_size
2. Evaluate the results and proceed accordingly

As the M-Tree implementation is similar to a B-Tree, we expect it to be a 'naive' data structure to use for our multidimensional data analysis. Thus, we should check the runtime results of a simple NN procedure first and evaluate before continuing with the M-Tree exploration.

B. Tools and Algorithms Used

The top level "requirements.txt" file includes the packages and libraries needed for our project.

Pandas is mainly used for reading the dataset. Numpy is used for processing and manipulating array values and query files, as well as mathematical operations. We use matplotlib to generate the graphs for our exploration above (how the varying parameters affect the NN search time for each data structure).

For the KD-Tree and Ball-Tree exploration, we use sklearn's neighbors module to import the tree implementations.

For the MTree exploration, we use an open source implementation found here: https://github.com/tburette/mtree

For the Brute Force, KD-Tree, and MTree implementations, we define the distance function as the euclidean distance. For the Ball-Tree, we use the default distance function that sklearn uses for the Ball-Tree implementation (radial distance).

V. Results and Analysis

A. Evaluation Method

Our evaluation methodology includes analyzing the NN search times (in units of seconds) for our prescribed procedure above for each of the data structures (and parameter combinations) that we chose to explore. For the displayed results below, we will include this evaluation and analysis where appropriate.

B. Results

Brute Force

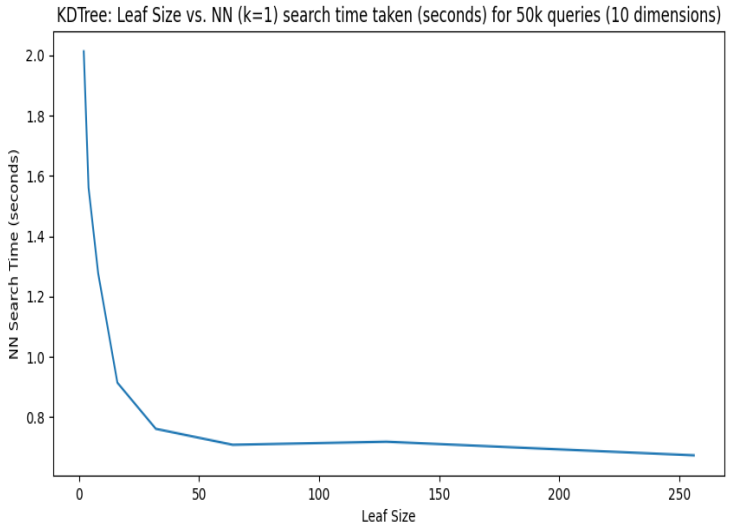
```
PS E:\Documents\CMPE297\CMPE297_Project\CMPE297_Group_Project\BruteForce> python .\BruteForce.py
For brute force method and data of 10 dimensions, K=1 nearest neighbors for queries1.txt 50K queries are outputted to BruteForce_NN_indexes.txt
For brute force method, searching for k =1 nearest neighbors for each of 50K queries took 245.33812189102173


For brute force method, searching for k =40 nearest neighbors for each of 50K queries took 255.51579928398132
```

[Figure 5]

From figure 5, we can conclude from our brute force procedure that the runtime of NN search is long, as expected. However, the upside is that varying of the parameter k (# of neighbors to search for per query) does not really affect the runtime. This is because we can simply locate the k indexes of the k minimum values after storing the distances between the query and each of the data points.

KDTree

1. Leaf_size vs. NN search time



[Figure 6]

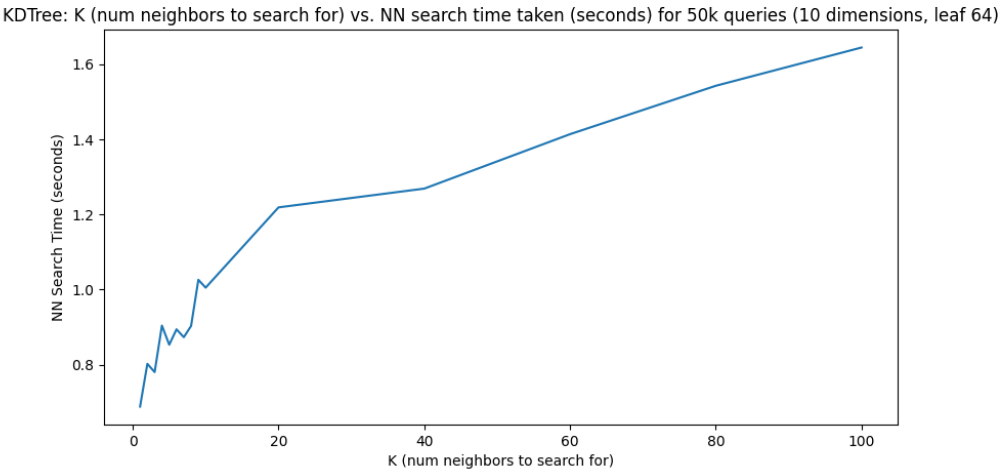
```
For 50k queries and 10 dimensions and kd tree with leaf size of 2, the time taken for nearest neighbor search (k=1) was  2.012998342514038
For 50k queries and 10 dimensions and kd tree with leaf size of 4, the time taken for nearest neighbor search (k=1) was  1.5610122680664062
For 50k queries and 10 dimensions and kd tree with leaf size of 8, the time taken for nearest neighbor search (k=1) was  1.2770004272460938
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=1) was  0.9139995574951172
For 50k queries and 10 dimensions and kd tree with leaf size of 32, the time taken for nearest neighbor search (k=1) was  0.7609963417053223
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=1) was  0.7079997062683105
For 50k queries and 10 dimensions and kd tree with leaf size of 128, the time taken for nearest neighbor search (k=1) was  0.7181088924407959
For 50k queries and 10 dimensions and kd tree with leaf size of 256, the time taken for nearest neighbor search (k=1) was  0.6729996204376221
```

[Figure 7]

The above figure 6 and figure 7 show our KD-Tree's procedure's output for how the leaf size affects the NN search time (for k=1 and number of dimensions = 10).

Interestingly, there appears to be a y=(1/x) correlation. We should be smart about choosing the leaf_size, and we decide to pick leaf_size = 64 to continue.

2. K vs. NN search time



[Figure 8]

```
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=1) was  0.6880102157592773
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=2) was  0.8019990921020508
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=3) was  0.7799978256225586
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=4) was  0.9040102958679199
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=5) was  0.8529877662658691
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=6) was  0.8940014839172363
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=7) was  0.8729984760284424
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=8) was  0.9029977321624756
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=9) was  1.0259959697723389
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=10) was  1.0050005912780762
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=20) was  1.2189998626708984
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  1.2689931392669678
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=60) was  1.4140410423278809
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=80) was  1.5429964065551758
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=100) was  1.6449987888336182
```

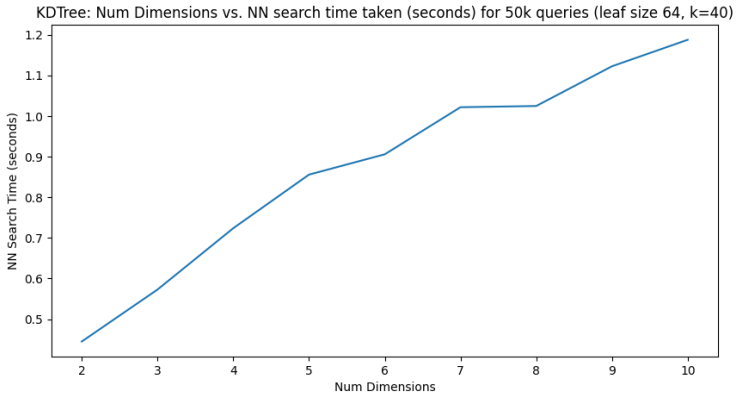
[Figure 9]

The above figure 8 and figure 9 show our KD-Tree's procedure's output for how k (number of neighbors to search for per query) affects the NN search time (taking the leaf_size = 64 and num dimensions = 10).

The relationship does not seem completely linear (perhaps either slightly logarithmic, or 2 piecewise linear functions).

We decide to choose a middle-ground of k = 40 to proceed.

3. Number of dimensions vs. NN search time



[Figure 9]

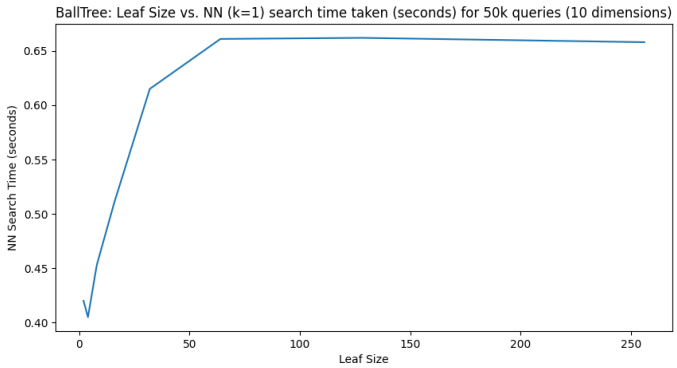
```
For 50k queries and 2 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  0.44501328468322754
For 50k queries and 3 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  0.5730025768280029
For 50k queries and 4 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  0.7239980697631836
For 50k queries and 5 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  0.8559980392456055
For 50k queries and 6 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  0.9059984683990479
For 50k queries and 7 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  1.0219993591308594
For 50k queries and 8 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  1.0249953269958496
For 50k queries and 9 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  1.1228032112121582
For 50k queries and 10 dimensions and kd tree with leaf size of 64, the time taken for nearest neighbor search (k=40) was  1.188000202178955
```

[Figure 10]

As expected, from figure 9 and figure 10, there appears to be a pretty linear relationship between the number of dimensions and the NN search time for our KD-Tree data structure (taking leaf_size=64 and k=40) and dataset.

<u>BallTree</u>

1. Leaf_size vs. NN search time



[Figure 11]

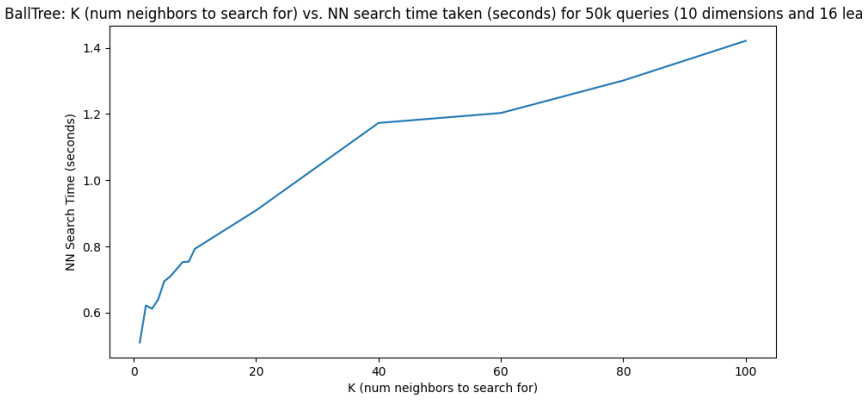
```
For 50k queries and 10 dimensions and Ball tree with leaf size of 2, the time taken for nearest neighbor search (k=1) was  0.4200127124786377
For 50k queries and 10 dimensions and Ball tree with leaf size of 4, the time taken for nearest neighbor search (k=1) was  0.4050095081329345 7
For 50k queries and 10 dimensions and Ball tree with leaf size of 8, the time taken for nearest neighbor search (k=1) was  0.4529933929443359 4
For 50k queries and 10 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=1) was  0.5109965801239014
For 50k queries and 10 dimensions and Ball tree with leaf size of 32, the time taken for nearest neighbor search (k=1) was  0.6149997711181641
For 50k queries and 10 dimensions and Ball tree with leaf size of 64, the time taken for nearest neighbor search (k=1) was  0.6609997749328613
For 50k queries and 10 dimensions and Ball tree with leaf size of 128, the time taken for nearest neighbor search (k=1) was  0.6620116233825684
For 50k queries and 10 dimensions and Ball tree with leaf size of 256, the time taken for nearest neighbor search (k=1) was  0.6580119132995605
```

[Figure 12]

The above figure 11 and figure 12 show our Ball-Tree's procedure's output for how the leaf size affects the NN search time (for k=1 and number of dimensions = 10).

Interestingly, as opposed to the KD-Tree's y=(1/x) relationship, there appears to be a y=log(x) correlation here. We should choose a relatively smaller leaf_size, and we pick leaf_size = 16.

2. K vs. NN search time



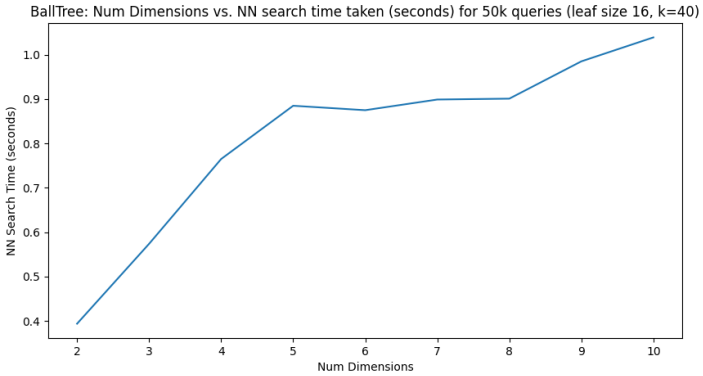
[Figure 13]

```
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=1) was  0.5100138187408447
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=2) was  0.6220111846923828
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=3) was  0.6120002269744873
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=4) was  0.6399993896484375
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=5) was  0.6950125694274902
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=6) was  0.7099289894104004
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=7) was  0.7310054302215576
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=8) was  0.7530026435852051
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=9) was  0.7540738582611084
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=10) was  0.7929973602294922
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=20) was  0.9089996814727783
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  1.1730012893676758
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=60) was  1.2030112743377686
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=80) was  1.3011643886566162
For 50k queries and 10 dimensions and kd tree with leaf size of 16, the time taken for nearest neighbor search (k=100) was  1.4210128784179688
```

[Figure 14]

The above figure 13 and figure 14 show our Ball-Tree's procedure's output for how k (number of neighbors to search for per query) affects the NN search time (taking the leaf_size = 16 and num dimensions = 10).
Overall, there is a pretty linear relationship. We pick a reasonable k = 40.

3. Number of dimensions vs. NN search time



[Figure 15]

```
For 50k queries and 2 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  0.39399027824401855
For 50k queries and 3 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  0.5739986896514893
For 50k queries and 4 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  0.7649991512298584
For 50k queries and 5 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  0.8849990367889404
For 50k queries and 6 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  0.8750123977661133
For 50k queries and 7 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  0.8989956378936768
For 50k queries and 8 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  0.9009974002838135
For 50k queries and 9 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  0.9850115776062012
For 50k queries and 10 dimensions and Ball tree with leaf size of 16, the time taken for nearest neighbor search (k=40) was  1.038996696472168
```

[Figure 16]

From figure 15 and figure 16, there appears to be 2 linear functions (piecewise functions) describing the relationship between number of dimensions and NN search time for Ball-Tree (taking leaf_size = 16 and k=40). For k<5, the NN search time increases faster (high slope) as k increases than for k>5 (where the NN search time increases at a slower rate as k increases). This perhaps shows that Ball-Tree performs better than KD-Tree on data with more dimensions.

<u>MTree</u>
For our MTree exploration, we used arbitrary default values of max_node_size of 32 and 64 and just did a simply NN neighbor search for our 50K queries. The resulting search time was much too slow in comparison to the KDTree and BallTree implementation (as we can see from figure 17).

```
For 50k queries and 10 dimensions and M tree with max node size of 32, the time taken for nearest neighbor search (k=1) was  133.1019983291626
For 50k queries and 10 dimensions and M tree with max node size of 64, the time taken for nearest neighbor search (k=1) was  281.43636536598206
```

[Figure 17]

Thus, we did not proceed with further exploration of the MTree, and confirm that the MTree, which is similar in structure to a BTree, does not work well for multidimensional data and is a naive data structure to use in this case.

C. Verification

We also verified the validity of our data structure's outputs to make sure the nearest neighbor operations for the queries are indeed operating correctly.

In each of the Brute Force, KD-Tree, and Ball-Tree implementations (see respective folders), there is a procedure (checkNearestNeighbors method) in the main file that does a simple NN search using the tree (with arbitrary parameters) upon the queries1.txt's 50k queries. The output files (for example, "BallTree_NN_indexes.txt" in BallTreeNN folder) contains the indexes of the nearest neighbor for each of the 50K queries.

We verified that the Brute Force, KD-Tree, and Ball-Tree NN procedure on queries1.txt found the exact same neighbors for each of the queries.

For the M-Tree implementation, however, we were not able to verify the correctness of their NN implementation, as the code returns the value of the closest neighbor rather than the index. Nonetheless, the M-Tree procedure results were much too slow for practical multi-dimensional analysis purposes.

D. Summary of Results

The below table (table 1) shows the best NN search time (seconds) achieved for k=1 and k=40 for each data structure/method explored, using our 50k queries of 10 dimensions each (queries1.txt).

Data Structure	**K = 1**	**K = 40**
Brute Force	245s	255s
KD-Tree	0.67s	1.18s
Ball-Tree	0.41s	1.04s
M-Tree	133s	–

[Table 1]

We can see that kd-tree and ball-tree both performed very well, with the best times going to the ball-tree.

VI. Lessons Learned

- Optimized data structures for multi-dimensional indexing (such as KDTree and BallTree) performs well on multidimensional data
    - In our case of dimension=10, BallTree performs NN search better
    - For lower # dimensions (k<5), KDTree performs NN search better

  Reasoning for this could be: We have learned about the KD-Tree structure in class. The BallTree implementation is similar to KDTree, except instead of dividing the spatial domain in one dimension at a time, and seemingly creating nodes that represent regions of rectangles in space, the KDTree divides the spatial domain in hyperspheres. So the root node would be a hypersphere capturing all the points, and inside that would be 2 hyperspheres containing a set of points, and so on and so forth. So rather than Euclidean distance, the Balltree can do NN by radius search, which makes it more effective in higher dimensional space, but including more overhead for lower dimension data.
- For KDTree, selection of leaf_size is important due to the y=(1/x) relationship
- For BallTree, the leaf_size should probably be a relatively small size
- Naive data structures for low dimension data does not work well on high dimensional data
    - Still better than brute force approach

  Reasoning for this could be: Mtree is similar to the Btree data structure that we learned in class. It appears to be a method of also dividing 2 dimension space into tree structure. It is also like a mix of R-tree and B-tree. However, it seems to only work well for low dimension data (such as 2 dimensions), and with properties like a Btree, does not perform well in high dimensional space. For our project, we saw it on a list of data structures used for multidimensional data, and decided to try it out; however, the open source implementation did not perform well as seen from our results.

VII. Open Issues and Future Work

- Explore further relations/combinations between data structure and dataset parameters such as leaf_size, k, and num_dimensions
- Try out more multidimensional data structures optimized for NN queries in high-dimensional space (such as LSH and R-tree)
- Test with different datasets, and skewness of the dataset / test queries to see how results are affected for each data structure
- Use larger and higher dimension dataset to see if this affects our results
- Measure Cost of NN queries, and not just the runtime.

VIII. Environment Setup

A. For .ipynb files, simple use of Google Colab will allow running of the code and exploration
B. In order to run the main python files (BruteForce.py, KDTreeImplementation.py, BallTreeImplementation.py, MTreeImplementationNew.py) to generate our output/results that can be seen in the images folder, follow the steps to create the environment
    1. Use pipenv to manage the dependencies: https://realpython.com/pipenv-guide/
    2. Install the required libraries at the top level via “pipenv install -r requirements.txt”
    3. In your terminal, activate the shell via “pipenv shell”, and then cd to any of the data structure folder and run “python {filename.py}” to run our main python files