IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Date of publication August 25, 2023

Digital Object Identifier TBD

A comparative study of algorithms to determine the K-Nearest Neighbours

PRATIGYA PAUDEL[1], SUSHANK GHIMIRE[1]

[1]Institute of Engineering, Thapathali Campus, Bagmati 44600 Nepal (e-mail: pratigyapaudel0@gmail.com)

Corresponding author: Pratigya Paudel (e-mail: pratigyapaudel0@gmail.com).

"This work was completed as a part of a college practical for Data Mining (CT725)."

ABSTRACT KD-Tree is a powerful data structure that finds its utility in optimizing searches and retrievals in multi-dimensional spaces. In this study, we delve into the realm of KD-Trees by applying them to the analysis of the Iris dataset, which encompasses data points defined by three distinct features. The primary objective is to exploit the KD-Tree's ability to accelerate searches and nearest neighbor queries within multi-dimensional datasets. By constructing a KD-Tree from the Iris dataset and effectively partitioning the data points, we aim to enhance the efficiency of various spatial operations. This experiment involves creating a KD-Tree to efficiently organize the Iris dataset in a hierarchical manner. Each node within the KD-Tree delineates a specific region in the multi-dimensional space defined by the features. We are particularly interested in how KD-Trees expedite nearest neighbor searches, making them an invaluable asset when dealing with complex data structures. We will finally look at ball trees, that outperform both the brute-force algorithm and the KD trees for building KNN models quickly.

INDEX TERMS KD Tree, Nearest Neighbour, Supervised Machine Learning

I. INTRODUCTION

KD Tree is a binary tree structure where each node represents a k-dimensional point. Nodes in this tree can be seen as generating a dividing hyperplane that partitions space into two segments, often referred to as half-spaces. Data points situated on the left side of this hyperplane are associated with the left subtree of the node, while those on the right side are linked to the right subtree. The choice of the hyperplane's orientation is determined by the node's specific dimension association within the tree.

More precisely, every node in the k-d tree corresponds to one of the k dimensions, and the hyperplane aligned with this dimension's axis is orthogonal to it. For instance, if the dimension associated with the "x" axis is selected for a particular division, data points with smaller "x" values compared to the node would be located within the left subtree. Conversely, data points with larger "x" values would be situated in the right subtree. In this scenario, the hyperplane's position is determined by the x value of the point, and its normal direction coincides with the unit x-axis. KD-Tree is an effective data structure for performing nearest neighbor search efficiently in multi-dimensional spaces. This method significantly reduces the search space and accelerates the retrieval of the closest data point to a given query point.

A Ball Tree is a data structure used in machine learning and computational geometry for nearest neighbor search and range query operations. It's particularly well-suited for high-dimensional spaces where the distribution of data points is uneven or non-uniform. The Ball Tree organizes the data points in a hierarchical structure, where each node represents a bounding hypersphere containing a subset of the data points. The splitting process involves selecting a center point and calculating the radius such that all data points within the hypersphere are enclosed. This structure efficiently partitions the data space and allows for faster nearest neighbor searches by quickly identifying regions that potentially contain the nearest neighbors. Ball Trees are advantageous in scenarios where the data is irregularly distributed and Euclidean space might not be the most suitable metric. However, constructing a Ball Tree can be more computationally intensive compared to KD Trees, especially in lower-dimensional spaces. Overall, Ball Trees provide an effective way to accelerate nearest neighbor searches in high-dimensional datasets with non-uniform distributions.

The Iris dataset is a well-known and frequently used dataset in the field of machine learning and statistics. It contains information about various attributes of iris flowers belonging to three different species: Setosa, Versicolor, and Virginica. In this modified version of the Iris dataset, the number of features has been pruned to three, providing a concise representation of the data. The three retained features are typically the sepal length, sepal width, and petal length of the flowers. By focusing on these three features, the pruned Iris dataset maintains its ability to discriminate between different iris species while reducing the complexity introduced by additional attributes. This trimmed dataset is still highly valuable for classification using KD tree.

II. METHODOLOGY

A. THEORY

A KD-Tree (K-Dimensional Tree) is a data structure used for efficient k-nearest neighbor (k-NN) search. It organizes data points in a hierarchical manner, partitioning the space into regions. At each node, a splitting hyperplane is defined perpendicular to a chosen dimension, dividing data points into two subsets. During search, the tree is traversed by comparing the query point's coordinates to the hyperplane, allowing for efficient pruning of search paths. This process minimizes the number of distance calculations, enabling KD-Trees to swiftly identify the k-nearest neighbors based on their proximity in the feature space.

In the brute-force technique for k-nearest neighbor (k-NN) search, each query point is compared to all data points in the dataset. Distances between the query point and every data point are calculated, and the k-nearest neighbors are determined by selecting the points with the smallest distances. This method exhaustively evaluates all data points, making it straightforward but computationally expensive, particularly for large datasets and high dimensions. While conceptually simple, the brute-force approach becomes less efficient as the dataset size increases, as it involves computing distances to all points regardless of their actual proximity to the query point.

The KD-Tree algorithm relies on the concept of splitting hyperplanes to efficiently organize data points. At each node of the tree, a hyperplane is established perpendicular to a chosen dimension. This division separates the data points into two subsets, facilitating focused search operations. The dimension to split along is determined by the depth of the tree, creating a hierarchical structure that guides the search process.

Computing the distance between a query point and data points is fundamental in KD-Tree k-NN search. The most commonly used metric is the Euclidean distance formula. By calculating the distance based on the coordinates of each point, the algorithm gauges their relative proximity. This allows the KD-Tree to identify potential neighbors efficiently and rank them based on their distances.

During traversal of the KD-Tree, a critical optimization involves pruning unnecessary branches. This is achieved by comparing the distance from the query point to the splitting hyperplane with the current minimum distance to a known neighbor. If the distance exceeds the minimum, the algorithm can confidently discard that branch of the tree, avoiding unnecessary computations and focusing on potential nearest neighbors.

The order in which nodes of the KD-Tree are traversed plays a pivotal role in the algorithm's efficiency. The traversal order is determined by comparing the query point's coordinates to the value of the splitting hyperplane at the current node. If the query point's coordinate along the splitting dimension is smaller than the node's value, the algorithm proceeds to the left subtree; otherwise, it moves to the right subtree. This strategic traversal ensures that relevant portions of the tree are explored first, enhancing the likelihood of finding accurate nearest neighbors quickly.

B. ALGORITHM FOR KD TREE

Input: Dataset *data*, Current depth *depth*
Output: Root of KD-Tree
If data is empty:
  Return null
$axis \leftarrow depth \mod$ number of dimensions
Sort *data* along *axis*
median $\leftarrow$ middle element of sorted data
node $\leftarrow$ new KD-Tree node
$node.value \leftarrow median$
$node.left \leftarrow$ build_kd_tree$(data[:\text{median index}], depth+1)$
$node.right \leftarrow$ build_kd_tree$(data[\text{median index}+1:], depth+1)$
Return *node*

C. ALGORITHM FOR BRUTE FORCE APPROACH

Input: Query point q, Dataset *data*, Number of neighbors k
Output: List of k nearest neighbors
Initialize an empty list *neighbors*
For each data point p in *data*:
  Calculate the distance between q and p
  Add $(p, \text{distance})$ to *neighbors*
Sort *neighbors* based on distance in ascending order
Select the first k elements from *neighbors* as the k nearest neighbors
Return k nearest neighbors

D. ALGORITHM FOR BALL TREES

Input: Query point q, Dataset *data*, Number of neighbors k
Output: List of k nearest neighbors
Initialize an empty list *neighbors*
For each data point p in *data*:
  Calculate the distance between q and p
  Add $(p, \text{distance})$ to *neighbors*
Sort *neighbors* based on distance in ascending order
Select the first k elements from *neighbors* as the k nearest neighbors
Return k nearest neighbors

E. TIME COMPLEXITY OF THE APPROACHES

Brute Force Method:
The time complexity for a single query point in the brute force

KNN method is $\mathcal{O}(N \times d)$, where N is the number of data points and d is the number of dimensions.

KD Tree:

The KD tree-based KNN method has an average time complexity of $\mathcal{O}(\log N)$ for querying the k nearest neighbors once the KD tree is constructed. Constructing the KD tree initially takes $\mathcal{O}(N \times \log N)$ time.

Ball Trees:

The time complexity of constructing a ball tree is typically around $\mathcal{O}(N \log N)$, where N represents the number of data points. This complexity arises from the recursive partitioning process that involves calculating bounding hyper-spheres for subsets of the data. The actual time taken may be influenced by the data's dimensionality and distribution. The process aims to efficiently organize the data to enable faster nearest neighbor queries.

F. MATHEMATICAL FORMULAE

1) Calculation of Euclidean Distance

Euclidean distance is a measure of the straight-line distance between two points in a multi-dimensional space. It is a commonly used distance metric in various fields to quantify the similarity or dissimilarity between data points.

The Euclidean distance between two points $\mathbf{p} = (p_1, p_2, \ldots, p_n)$ and $\mathbf{q} = (q_1, q_2, \ldots, q_n)$ in n-dimensional space can be calculated using the following formula:

$$\text{Euclidean Distance} = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{1}$$

G. INSTRUMENTATION TOOLS

The entirety of the process is done using Python. Google Colab, short for Google Colaboratory, is an online platform provided by Google for running and sharing Jupyter notebook environments and it was used for all of the coding. Google colab provides a number of built-in functions for data analysis. The dataset is loaded through scikit-learn and visualized using pandas. The results are then visualized using different visualization tools like Seaborn and matplotlib.

H. WORKING PRINCIPLE

1) Dataset Collection

The dataset used for the comparison of speed between the two approaches has been the popular iris dataset. The first three features, namely sepal length, sepal width, and petal length of the flowers have been extracted to train the models with.

2) Brute-Force Algorithm

Training the brute force model for k-nearest neighbors (KNN) involves building a direct and straightforward approach to classify data points. During training, the algorithm simply memorizes the entire training dataset, creating a reference to each data point and its corresponding class label. This reference allows the algorithm to quickly access the dataset during the classification phase. While the brute force method is conceptually simple and doesn't involve complex optimization, it can be computationally intensive and less efficient as the dataset size grows. The classification step in this method involves calculating the distance between the query point and all data points in the dataset, selecting the k-nearest neighbors based on distance, and determining the majority class among those neighbors.

3) KD Tree

Training the k-nearest neighbors (KNN) model using a KD tree introduces a more efficient approach to classification by optimizing the search for nearest neighbors. Unlike the brute force method, the KD tree constructs a balanced binary tree that partitions the feature space into smaller regions, facilitating quicker nearest neighbor searches. During the construction phase, the algorithm recursively selects pivot points along different dimensions to create a hierarchical tree structure. This tree significantly reduces the number of distance calculations required during classification. When a query point is provided, the KD tree traversal efficiently narrows down the search space by navigating the tree based on the pivot points. As a result, the KNN algorithm only evaluates distances for points located in the vicinity of the query, minimizing computation. The KD tree method is particularly advantageous in high-dimensional spaces where the brute force approach becomes computationally prohibitive. By optimizing the search process, the KD tree enhances the speed and efficiency of KNN classification without compromising accuracy.

4) Ball Tree

Diverging from the brute force method, the ball tree constructs a hierarchical structure that encompasses the dataset through bounding hyperspheres, resulting in faster proximity searches. During the creation phase, the algorithm iteratively selects pivot points to generate a tree that encapsulates data points within these hyperspheres. This innovative structure effectively diminishes the number of distance calculations necessary during classification. When a query point is presented, the ball tree navigation efficiently prunes the search area by navigating through the tree's nested hyperspheres. Consequently, the KNN algorithm only calculates distances for points enclosed within the vicinity of the query, significantly curtailing computational burden. The ball tree strategy is particularly effective in scenarios involving high-dimensional spaces, where the brute force method's efficiency diminishes. By optimizing the search process, the ball tree method elevates the velocity and effectiveness of KNN classification, all while maintaining the precision and reliability of the model.

III. RESULTS

A. BRUTE FORCE KNN

The results from Brute Force KNN displayed a long waiting time for obtaining the predictions. The time needed to query a given point using Brute-force KNN came out to be around

0.047 seconds. This waiting time resulted in accuracy of 93.3%. The plot for the data points and the query points can be used to visualize the way nearest neighbours are selected. Clearly, the data points nearest to the query points are selected as the nearest neighbours.

B. KD TREE

While there is no build time for Brute-force KNN, the highest amount of time taken in a KD tree is when building it. From the results, KD tree took 0.0048 seconds to build, which is still a lot faster than the time needed for Brute-force KNN to query a result. The query time once the tree is built is almost non-existent. The KD tree approach surpasses the accuracy of the Brute-force algorithm to reach new heights of 98% accuracy. The partition space created by the points while building the KD tree can be visualized. Also, the tree can be visualized in itself as well. The KD tree also helps separate the areas for the different classes which can help classify a given point with more ease.

C. BALL TREE

Ball tree took the performance of the model to new heights with much improved time to build the tree and to query a point. It took roughly 0.004 seconds to build the ball tree from the scratch and the time taken for k-NN query came down to less than 0.000017 seconds. The tree, like the KD tree can be visualized with its nodes and branches as well. The accuracy for the approach is still quite high at 95%.

IV. DISCUSSION AND ANALYSIS

The comparison between Brute Force KNN, KD Tree, and Ball Tree methods for nearest neighbor search reveals intriguing insights into their performance. Brute Force KNN demonstrated accurate predictions but exhibited a drawback in terms of query time, taking around 0.047 seconds. However, this approach achieved an accuracy of 93.3%. In contrast, the KD Tree approach showcased remarkable efficiency by reducing query time to nearly negligible levels once the tree was built, though the build time was 0.0048 seconds. This approach excelled in accuracy, reaching 98%. The visualization of the KD tree's partition spaces and structure emphasized its ability to segregate classes and assist in accurate classifications.

The Ball Tree method emerged as a game-changer, significantly improving both build and query times. Constructing the ball tree took a mere 0.004 seconds, and querying a point required less than 0.000017 seconds. This striking efficiency didn't come at the cost of accuracy, with the model achieving an impressive 95% accuracy. Visualization of the ball tree, akin to KD tree, illustrated its hierarchical structure. These results indicate that advanced tree-based methods, such as KD Tree and Ball Tree, offer substantial advantages over Brute Force KNN in terms of efficiency and accuracy. The disparity in performance arises from their inherent data structure and partitioning strategies, allowing KD Tree and Ball Tree to significantly accelerate the nearest neighbor search process while maintaining or even improving predictive accuracy. This stays in line with the theory on KD trees and Ball trees. However, ball trees are known to be more computationally heavy on smaller datasets than KD trees.

V. CONCLUSION

In summary, the comparison and analysis of the Brute Force KNN, KD Tree, and Ball Tree methods for nearest neighbor search provide valuable insights into the trade-offs between accuracy and efficiency within this essential machine learning task. The Brute Force KNN method, although accurate, reveals its limitations through prolonged query times, making it less suitable for scenarios demanding rapid response times. On the other hand, the KD Tree method introduces a significant improvement in efficiency by substantially reducing query times once the tree is constructed, culminating in an impressive accuracy rate of 98%. The visualization of the KD Tree's partitioned spaces highlights its capacity to effectively separate data points and enhance classification accuracy.

Surprisingly, the Ball Tree method outshines expectations by achieving a balance between rapid tree construction and query times, resulting in a remarkable improvement in both efficiency and accuracy. Its ability to construct the tree in approximately 0.004 seconds and query a point in under 0.000017 seconds showcases its potential for real-time applications. Visualizing the Ball Tree's hierarchical structure emphasizes its effective representation of data relationships, contributing to its superior performance.

In light of these findings, it is evident that the choice of the nearest neighbor search method must be made judiciously based on the specific demands of the application. While Brute Force KNN remains a reliable choice for accuracy-focused tasks, KD Tree and Ball Tree methods provide efficient alternatives for scenarios demanding faster response times and processing large datasets. This exploration underscores the significance of algorithm selection in achieving a harmonious balance between accuracy and efficiency, driving the advancements of machine learning in practical applications.

VI. REFERENCES

- David Bowser-Chao and Debra L. Dzialo. "Comparison of the use of nearest neighbours and neural networks in top-quark detection." *Physical Review D*, vol. 47, no. 5, pp. 1900–1905, Mar. 1993. doi: 10.1103/physrevd.47.1900.

PRATIGYA PAUDEL is a fourth year student, studying computer engineering under IOE, Thapathali Campus. She has been involved in a lot of machine learning projects and has a keen eye for data analysis and AI related stuff. With the enthusiasm for Artificial Intelligence (AI), she is driven by the potential of AI to transform industries and tackle complex challenges. Her academic journey has equipped her with a strong foundation in AI concepts, including machine learning and data analysis. She possesses a relentless curiosity and is always eager to explore the latest advancements in AI. Her goal is to apply her knowledge and make a meaningful contribution in the field.

SUSHANK GHIMIRE is a fourth year student, studying computer engineering under IOE, Thapathali Campus. He possesses a lot of interest, working with data. His educational path has provided him with a solid understanding of AI concepts, encompassing machine learning and data analysis. He possesses an unwavering curiosity and is constantly eager to delve into the latest advancements in AI. His objective is to leverage his knowledge and expertise to create a significant impact in the field.

APPENDIX

A. TABLES

TABLE 1. Instances from the Iris Dataset

Sepal Length	Sepal Width	Petal Length	Class
5.1	3.5	1.4	Setosa
4.9	3.0	1.4	Setosa
4.7	3.2	1.3	Setosa
7.0	3.2	4.7	Versicolor
6.4	3.2	4.5	Versicolor
6.9	3.1	4.9	Versicolor
6.3	3.3	6.0	Virginica
5.8	2.7	5.1	Virginica
7.1	3.0	5.9	Virginica

B. FIGURES AND PLOTS

FIGURE 1. System Block Diagram

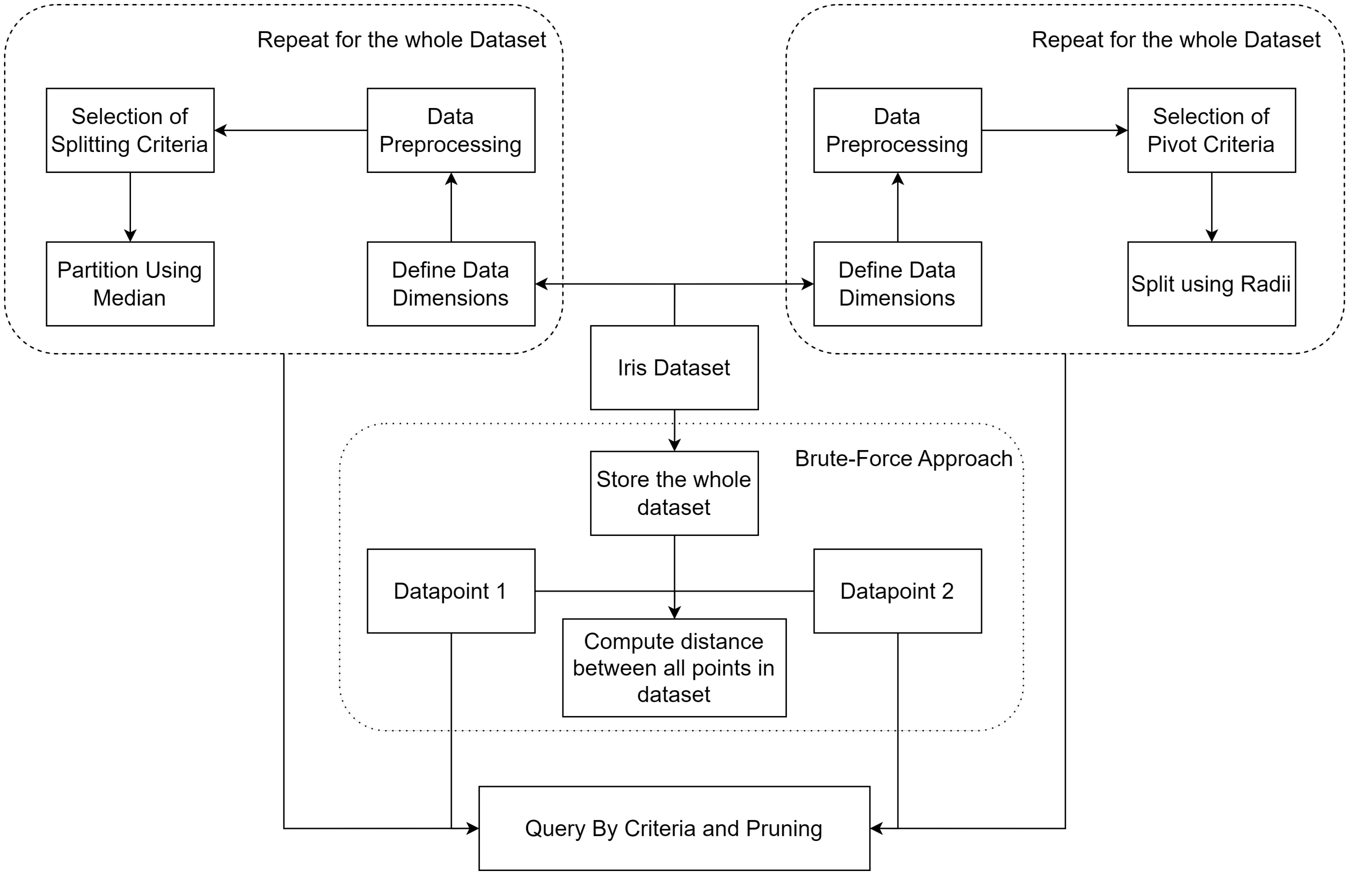


FIGURE 2. Dataset Distribution

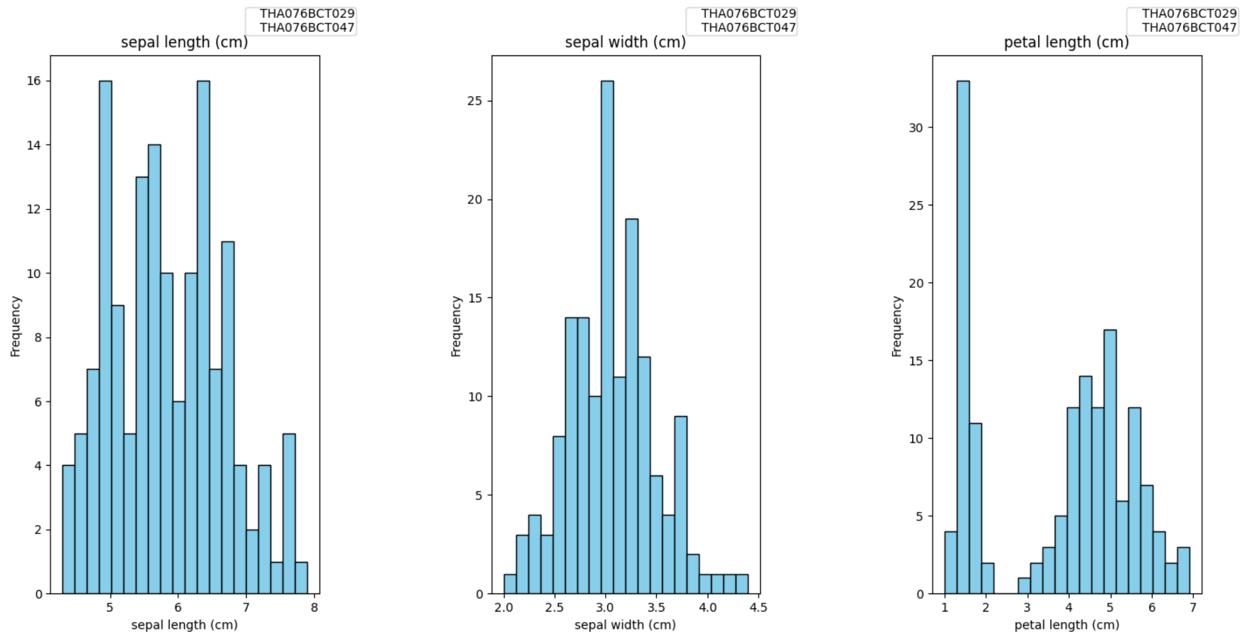


FIGURE 3. KNN formation using Brute-force algorithm

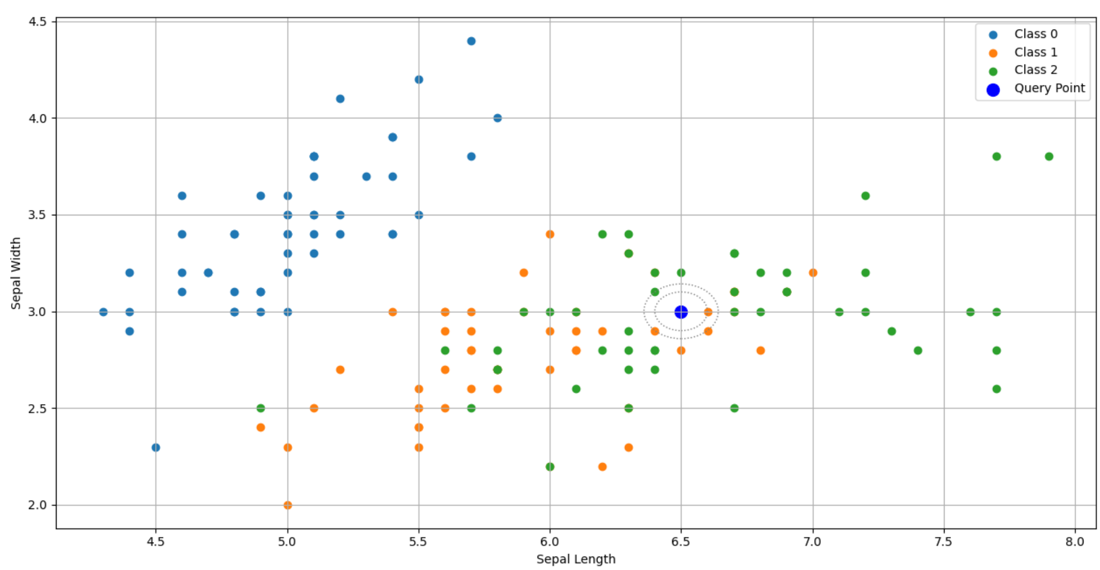


FIGURE 4. Query and Build times for KD tree and Brute-force algorithm

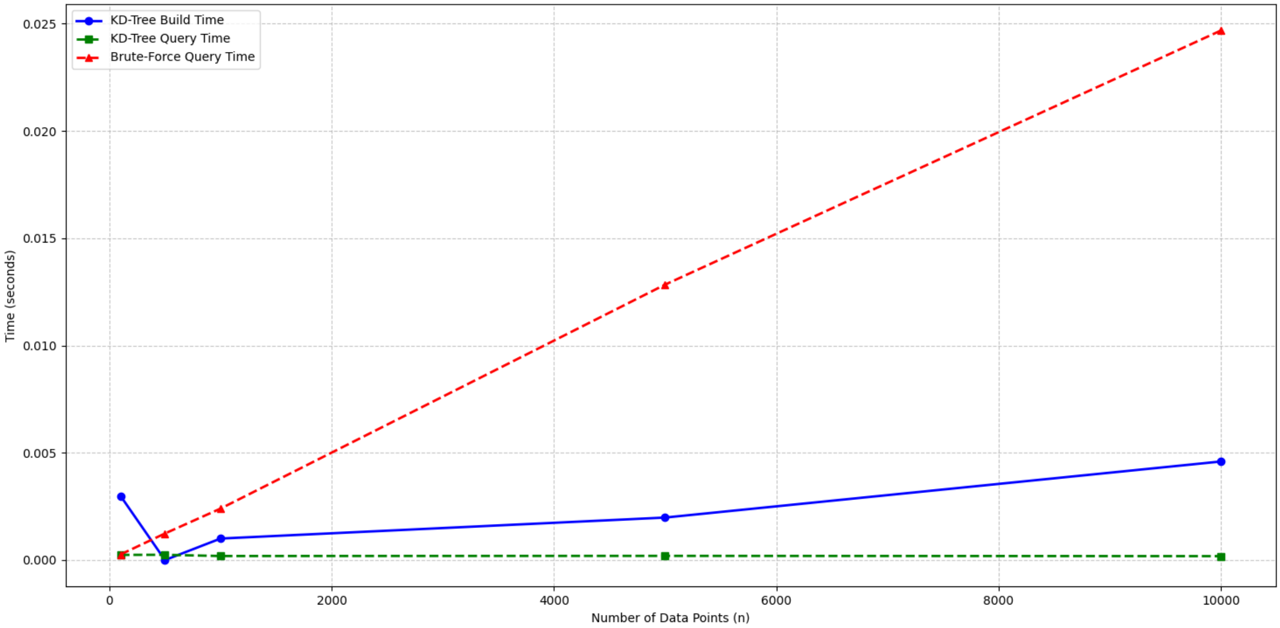


FIGURE 5. Partition Space Visualization using KD Trees

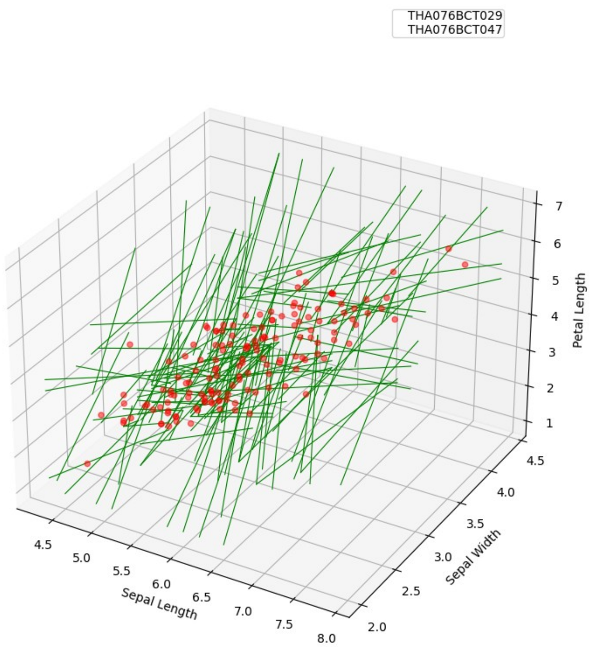


FIGURE 6. KD Tree Decision Boundary Visualization

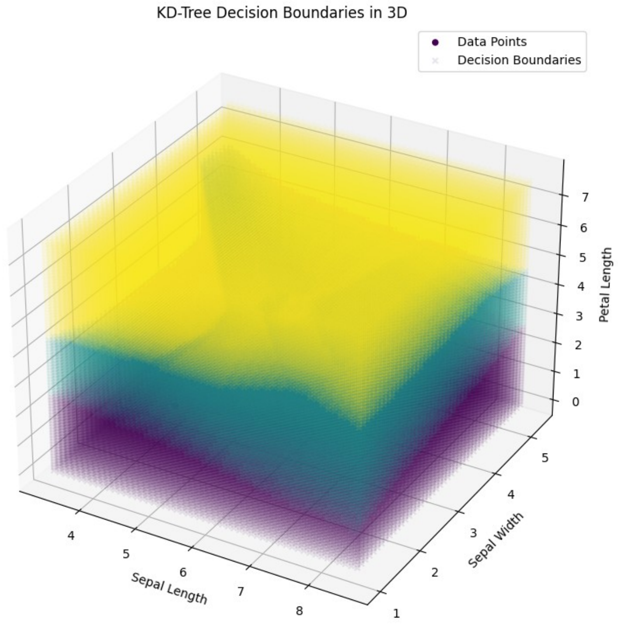


FIGURE 7. KD Tree Visualization

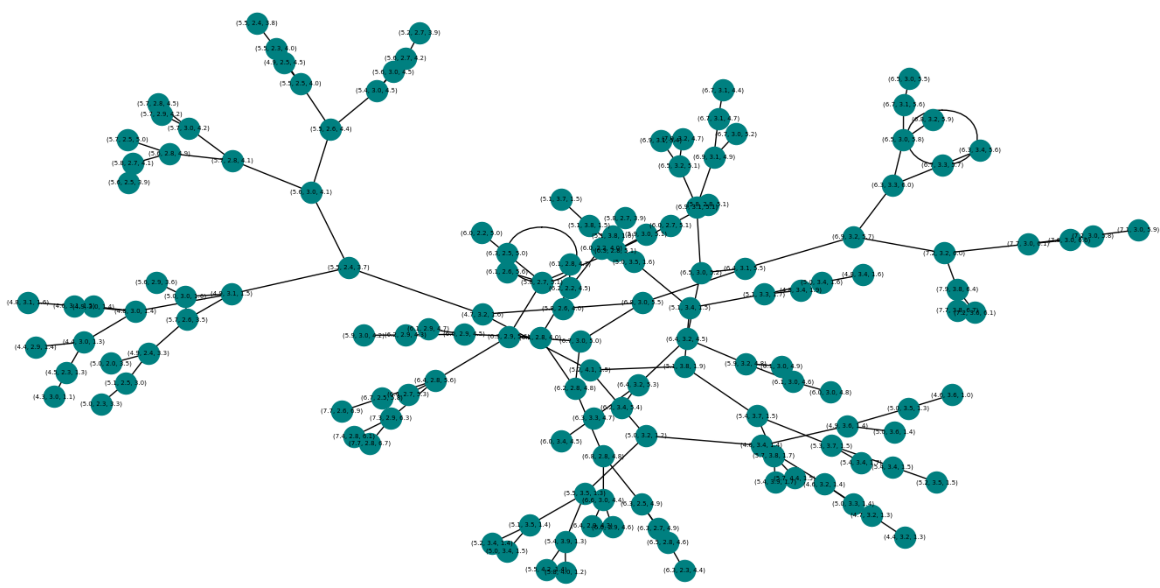
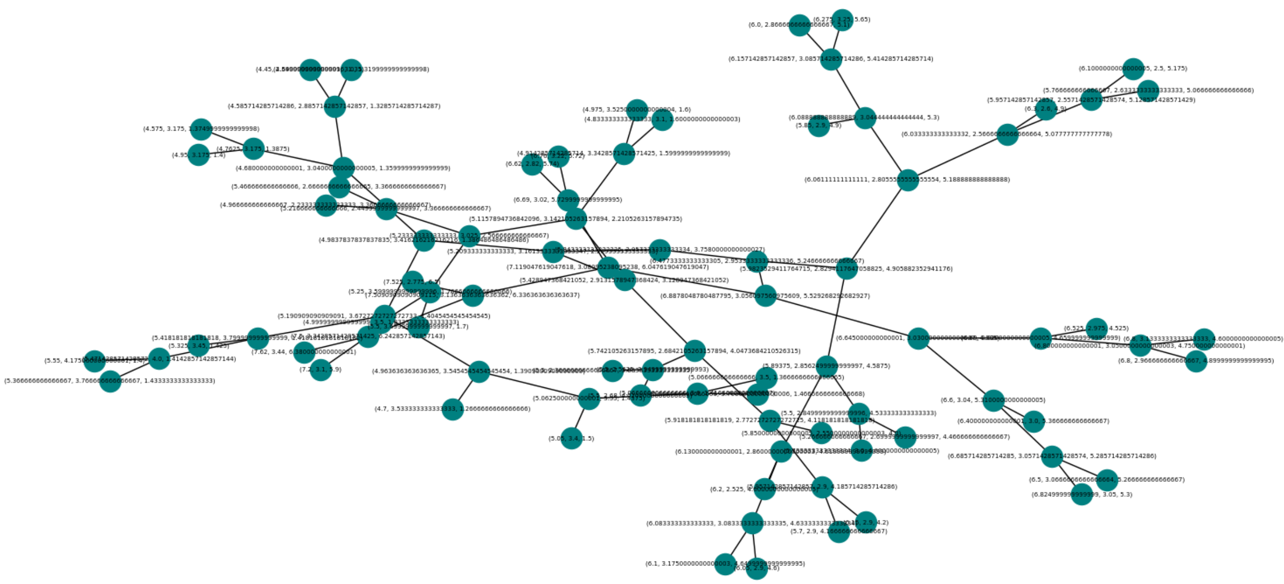


FIGURE 8. Ball Tree Visualization



C. CODING

```
#Import necessary libraries
import numpy as np
import time
import sys
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
import networkx as nx
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from collections import Counter
import time
import matplotlib.pyplot as plt

#Brute Force method
def brute_force_knn(train_X, train_y, test_X, k):
    predictions = []
    for test_point in test_X:
        distances = np.sqrt(np.sum((train_X - test_point)**2, axis=1))
        nearest_indices = np.argsort(distances)[:k]
        nearest_labels = train_y[nearest_indices]
        most_common = Counter(nearest_labels).most_common(1)
        predictions.append(most_common[0][0])
    return np.array(predictions)

#Time for predicting
k_neighbors = 3
iris = load_iris()
X = iris.data[:, :3]   # Truncate to 3 features
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

start_time = time.time()
brute_force_predictions = brute_force_knn(X_train, y_train, X_test, k_neighbors)
brute_force_time = time.time() - start_time
accuracy1 = accuracy_score(y_test,brute_force_predictions)

#Brute Force Visualization for 2 features
plt.figure(figsize=(10, 6))

# Plot all data points with colors according to classes
for class_num in np.unique(iris_target):
    class_indices = np.where(iris_target == class_num)
    plt.scatter(iris_data[class_indices, 0], iris_data[class_indices, 1], label=f'Class {c

# Plot query point
plt.scatter(query_point[0], query_point[1], color='blue', marker='o', s=100, label='Query P

# Draw circles to indicate distance for k-NN points
for idx in knn_indices:
    circle = Circle((knn_data[0][0], knn_data[0][1]), radius=distances[idx], color='gray',
    plt.gca().add_patch(circle)

plt.xlabel('Sepal Length')
```

```
plt.ylabel('Sepal Width')
plt.legend()
plt.grid(True)
plt.show()

#KD Tree

class Node:
    def __init__(self, point, left=None, right=None):
        self.point = point
        self.left = left
        self.right = right


def build_kdtree(points, depth=0):
    if len(points) == 0:
        return None

    k = points.shape[1]
    #k = 3
    axis = depth % k
    sorted_points = points[points[:, axis].argsort()]
    median_idx = len(sorted_points) // 2
    median_point = sorted_points[median_idx]

    left_points = sorted_points[:median_idx]
    right_points = sorted_points[median_idx + 1:]

    return Node(
        median_point,
        build_kdtree(left_points, depth + 1),
        build_kdtree(right_points, depth + 1)
    )

def _distance(p1, p2):
    return np.sqrt(np.sum((p1 - p2)**2))

def nearest_neighbor(tree, query, depth=0, best=None):
    if tree is None:
        return best

    if best is None or _distance(query, tree.point) < _distance(query, best.point):
        best = tree

    k = query.shape[0]
    axis = depth % k

    if query[axis] < tree.point[axis]:
        return nearest_neighbor(tree.left, query, depth + 1, best)
    else:
        return nearest_neighbor(tree.right, query, depth + 1, best)

def visualize_kdtree_3d(node, graph, parent=None, side=None, depth=0):
    if node is None:
        return

```

```
    graph.add_node(tuple(node[0]), depth=depth)
    if parent is not None:
        graph.add_edge(tuple(parent[0]), tuple(node[0]), side=side)

    k = len(node[0])   # Number of dimensions
    axis = depth % k

    visualize_kdtree_3d(node[1], graph, node, 'left', depth + 1)
    visualize_kdtree_3d(node[2], graph, node, 'right', depth + 1)

G = nx.Graph()
visualize_kdtree(iris_kdtree, G)

# Position the nodes for better visualization
pos = nx.spring_layout(G, seed=42)

# Draw the tree structure
plt.figure(figsize=(10,6))
nx.draw(G, pos, with_labels=True, node_size=300, node_color='teal', font_size=5,  font_colo
plt.title("KD-Tree Visualization")
plt.show()

#Plot 3d partitioning space
def plot_tree(ax, node, xmin, xmax, ymin, ymax, zmin, zmax, depth=0):
    if node is None:
        return

    k = len(node.point)
    axis = depth % k

    if axis == 0:
        ax.plot([node.point[0], node.point[0]], [ymin, ymax], [zmin, zmax], color='green',
        plot_tree(ax, node.left, xmin, node.point[0], ymin, ymax, zmin, zmax, depth + 1)
        plot_tree(ax, node.right, node.point[0], xmax, ymin, ymax, zmin, zmax, depth + 1)
    elif axis == 1:
        ax.plot([xmin, xmax], [node.point[1], node.point[1]], [zmin, zmax], color='green',
        plot_tree(ax, node.left, xmin, xmax, ymin, node.point[1], zmin, zmax, depth + 1)
        plot_tree(ax, node.right, xmin, xmax, node.point[1], ymax, zmin, zmax, depth + 1)
    else:
        ax.plot([xmin, xmax], [ymin, ymax], [node.point[2], node.point[2]], color='green',
        plot_tree(ax, node.left, xmin, xmax, ymin, ymax, zmin, node.point[2], depth + 1)
        plot_tree(ax, node.right, xmin, xmax, ymin, ymax, node.point[2], zmax, depth + 1)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(iris_data[:, 0], iris_data[:, 1], iris_data[:, 2], c='red', label='Data Points',
ax.set_xlabel('Feature 1')
ax.set_ylabel('Feature 2')
ax.set_zlabel('Feature 3')

plot_tree(ax, iris_kdtree,
          min(iris_data[:, 0]), max(iris_data[:, 0]),
          min(iris_data[:, 1]), max(iris_data[:, 1]),
          min(iris_data[:, 2]), max(iris_data[:, 2]))
```

```
plt.legend()
plt.show()

#Comparison of time
build_times_kdtree = []
query_times_kdtree = []
query_times_bruteforce = []

data_sizes = [20,40,60,80,100,120,150]

for num_points in data_sizes:
    data_points = iris_data[:num_points]

    # Build a KD-Tree
    start_time = time.time()
    knn_kdtree = KNeighborsClassifier(n_neighbors=1, algorithm='kd_tree')
    knn_kdtree.fit(data_points, np.zeros(num_points))  # Dummy labels for building KD-Tree
    build_time = time.time() - start_time
    build_times_kdtree.append(build_time)

    # Perform nearest neighbor queries using KD-Tree
    query_points = iris_data[num_points:num_points+1000]
    total_query_time_kdtree = 0
    for query_point in query_points:
        start_time = time.time()
        knn_kdtree.kneighbors([query_point])
        query_time = time.time() - start_time
        total_query_time_kdtree += query_time
    average_query_time_kdtree = total_query_time_kdtree / 1000
    query_times_kdtree.append(average_query_time_kdtree)

    # Perform nearest neighbor queries using Brute-Force
    total_query_time_bruteforce = 0
    for query_point in query_points:
        start_time = time.time()
        min_distance = np.inf
        for train_point in data_points:
            distance = np.linalg.norm(train_point - query_point)
            if distance < min_distance:
                min_distance = distance
        query_time = time.time() - start_time
        total_query_time_bruteforce += query_time
    average_query_time_bruteforce = total_query_time_bruteforce / 1000
    query_times_bruteforce.append(average_query_time_bruteforce)

# Plot the empirical time complexity
plt.figure(figsize=(10, 6))

# Plot build times
plt.plot(data_sizes, build_times_kdtree, label='KD-Tree Build Time', color='blue', marker=

# Plot query times
plt.plot(data_sizes, query_times_kdtree, label='KD-Tree Query Time', color='green', marker
plt.plot(data_sizes, query_times_bruteforce, label='Brute-Force Query Time', color='red', r

plt.xlabel('Number of Data Points (n)')
```

```
plt.ylabel('Time (seconds)')
legend_handles = [
    plt.Line2D([], [], color='black', marker='o', markersize=10, label='THA076BCT029\nTHA07
]
plt.legend(handles=legend_handles, loc='upper left', bbox_to_anchor=(0.7, 1.1), ncol=len(le
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()


# Ball Tree Imlementation
import numpy as np
import time
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
import networkx as nx

class BallNode:
    def __init__(self, center, radius, left=None, right=None, points=None):
        self.center = center
        self.radius = radius
        self.left = left
        self.right = right
        self.points = points

# Build the Ball Tree
# Build the Ball Tree
def build_balltree(points, min_points=5):
    points = np.array(points)   # Convert points to a NumPy array

    if len(points) == 0:
        return None

    center = np.mean(points, axis=0)
    radius = max(np.linalg.norm(point - center) for point in points)

    if len(points) <= min_points:
        return BallNode(center, radius, points=points)

    left_points = []
    right_points = []

    split_dim = np.argmax(np.ptp(points, axis=0))
    sorted_points = points[np.argsort(points[:, split_dim])]
    median_idx = len(sorted_points) // 2
    median_point = sorted_points[median_idx]

    for point in sorted_points:
        if point[split_dim] < median_point[split_dim]:
            left_points.append(point)
        else:
            right_points.append(point)

    return BallNode(center, radius, build_balltree(left_points), build_balltree(right_poin
```

```
# Query the Ball Tree for k nearest neighbors within a given radius
# Query the Ball Tree for k nearest neighbors within a given radius
def ball_tree_knn(node, query, k, radius, neighbors=None):
    if neighbors is None:
        neighbors = []

    if node is None:
        return neighbors

    dist = np.linalg.norm(query - node.center)

    if dist <= radius + node.radius:
        if node.points:
            for point in node.points:
                neighbors.append(point)
                if len(neighbors) >= k:
                    return neighbors
        else:
            neighbors = ball_tree_knn(node.left, query, k, radius, neighbors)
            neighbors = ball_tree_knn(node.right, query, k, radius, neighbors)

    elif query[0] < node.center[0]:
        neighbors = ball_tree_knn(node.left, query, k, radius, neighbors)
    else:
        neighbors = ball_tree_knn(node.right, query, k, radius, neighbors)

    return neighbors


# Load Iris dataset
iris = load_iris()
iris_data = iris.data[:, :3]

# Build the Ball Tree from the Iris dataset
start_time = time.time()
iris_balltree = build_balltree(iris_data)
build_time = time.time() - start_time

# Perform k-NN queries and measure time
num_queries = 1000
k = 3
radius = 0.5
query_points = np.random.rand(num_queries, 3)

total_query_time = 0
for query_point in query_points:
    start_time = time.time()
    knn_neighbors = ball_tree_knn(iris_balltree, query_point, k, radius)
    query_time = time.time() - start_time
    total_query_time += query_time

average_query_time = total_query_time / num_queries

print(f"Time taken to build Ball Tree: {build_time:.6f} seconds")
print(f"Average time taken for k-NN query: {average_query_time:.6f} seconds")
```

```
# Visualize Ball Tree
def visualize_balltree(node, graph, parent=None, side=None, depth=0):
    if node is None:
        return

    graph.add_node(tuple(node.center), depth=depth)
    if parent is not None:
        graph.add_edge(tuple(parent.center), tuple(node.center), side=side)

    visualize_balltree(node.left, graph, node, 'left', depth + 1)
    visualize_balltree(node.right, graph, node, 'right', depth + 1)

G = nx.Graph()
visualize_balltree(iris_balltree, G)

# Position the nodes for better visualization
pos = nx.spring_layout(G, seed=42)

# Draw the tree structure
plt.figure(figsize=(10,6))
nx.draw(G, pos, with_labels=True, node_size=300, node_color='teal', font_size=5, font_colo
plt.show()
```