KNN

Reading Material

PW SKILLS

Topics Covered

1. k-Nearest Neighbors (k-NN) Classifier
   - Definition and Significance
   - How k-NN Classification Works
   - Applications of k-NN Classifier
   - Advantages and Limitations
   - Practical Implementation

2. k-Nearest Neighbors (k-NN) Regressor
   - Definition and Significance
   - How k-NN Regression Works
   - Applications of k-NN Regressor
   - Advantages and Limitations
   - Practical Implementation

3. Variants of k-NN
   - Weighted k-NN
   - Distance-Weighted k-NN
   - Approximate k-NN

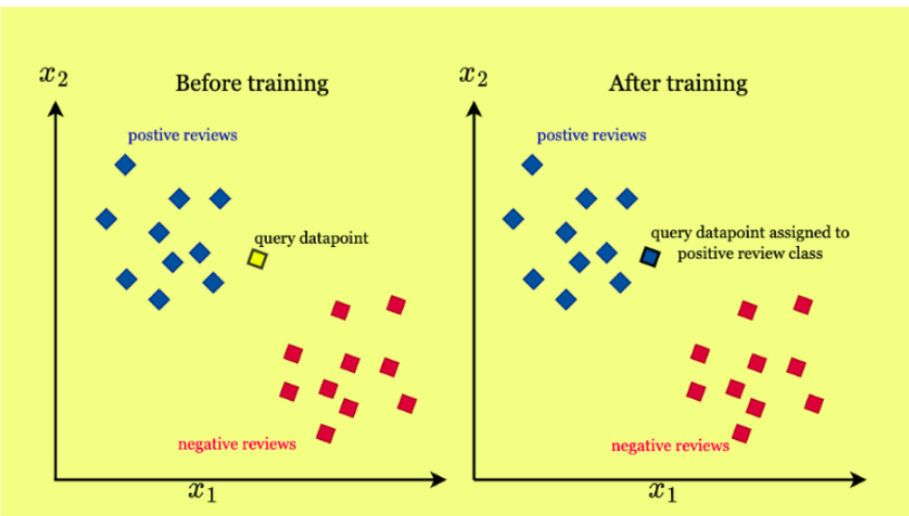
4. Brute Force k-NN
   - Definition and Significance
   - How Brute Force k-NN Works
   - Applications of Brute Force k-NN
   - Advantages and Limitations

5. k-Dimensional Tree (k-D Tree)
   - Definition and Significance
   - How k-D Tree Works
   - Applications of k-D Tree
   - Advantages and Limitations

What is the K-nearest neighbors (K-NN) Algorithm?

K-nearest neighbors (K-NN) is a popular supervised machine learning algorithm used for classification and regression. It works by finding the K most similar samples to a new, unseen data point and predicting its label or output based on the labels of the K nearest neighbors. The number of nearest neighbors, K, is a hyperparameter that must be chosen before training the model. K-NN is a simple yet effective algorithm that performs well on small to medium-sized datasets, particularly those with noisy or irregularly shaped boundaries.

k-Nearest Neighbors (k-NN) Classifier



1. Definition and Significance

The k-Nearest Neighbors (k-NN) algorithm is a simple, non-parametric, and supervised learning algorithm widely used for both classification and regression tasks. Unlike other algorithms that require a training phase, k-NN belongs to the family of "lazy learners," meaning it stores all available cases and classifies new cases based on a similarity measure. This method is significant due to its simplicity, effectiveness in various applications, and ability to make predictions without the need for a complex model.

2. How k-NN Classification Works

The core idea of k-NN classification is to classify a data point based on the majority class among its k nearest neighbors. The steps involved in k-NN classification are as follows:

1. **Determine the value of k:** Choose the number of neighbors to consider, typically an odd number to avoid ties.
2. **Calculate the distance:** For each data point in the dataset, calculate the distance between the new data point and existing data points. Common distance metrics include Euclidean, Manhattan, and Minkowski distances.
3. **Identify the k-nearest neighbors:** Sort the distances in ascending order and select the k nearest neighbors.
4. **Assign the class label:** The new data point is classified based on the majority vote of its k nearest neighbors.

This process is computationally intensive, especially for large datasets, as it involves calculating the distance between the query point and all other points in the dataset.

3. Applications of k-NN Classifier

k-NN has been successfully applied in various domains:

- **Data Imputation:** Filling in missing data by predicting the missing values based on the k nearest neighbors.
- Recommendation Systems: Recommending products or content to users based on the behavior of similar users.
- **Finance:** Credit risk assessment and stock market prediction using historical data to classify future outcomes.
- **Healthcare:** Predicting the likelihood of diseases based on patient data by comparing with previous cases.
- **Pattern Recognition:** Identifying patterns such as handwritten digits or text classification.

4. Advantages and Limitations

Advantages:

- **Simplicity:** Easy to understand and implement, making it one of the first algorithms learned by data scientists.
- **Flexibility:** Adaptable to various data types and distance metrics, making it versatile in application.
- **No Training Phase:** Since k-NN is a lazy learner, it doesn't require a training phase, which can be advantageous for certain applications.

Limitations:

- **Computationally Expensive:** The algorithm requires significant computation, especially with large datasets, due to the need to calculate the distance for every data point.
- **Curse of Dimensionality:** Performance degrades with high-dimensional data, as the algorithm becomes less effective in distinguishing between neighbors.
- **Prone to Overfitting:** With a small value of k, the algorithm may overfit the data, capturing noise rather than the actual pattern.
- **Sensitive to Irrelevant Features:** The presence of irrelevant or redundant features can significantly affect the accuracy of the model.

5. Practical Implementation

Implementing k-NN in Python using the scikit-learn library is straightforward. Here is a basic example:

```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score,
confusion_matrix, classification_report

# Load the Titanic dataset (assuming it is available as a
CSV file)
url = "https://raw.githubusercontent.com/datasciencedojo/
datasets/master/titanic.csv"
df = pd.read_csv(url)

# Select relevant features and the target variable
features = ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch',
'Fare', 'Embarked']
df = df[features + ['Survived']]
# Data preprocessing
# Convert categorical features to numeric
df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})
df['Embarked'] = df['Embarked'].map({'C': 0, 'Q': 1, 'S':
2})

# Handle missing values
df['Age'].fillna(df['Age'].median(), inplace=True)
df['Embarked'].fillna(df['Embarked'].mode()[0],
inplace=True)

# Split the data into features (X) and target (y)
X = df.drop('Survived', axis=1)
y = df['Survived']
```

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize and train the k-NN classifier
knn = KNeighborsClassifier(n_neighbors=5,
metric='minkowski', p=2)
knn.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy:.2f}")
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(class_report)
```

In this example, n_neighbors=5 indicates that the algorithm considers the 5 nearest neighbors. The metric='minkowski' with p=2 specifies the use of Euclidean distance.

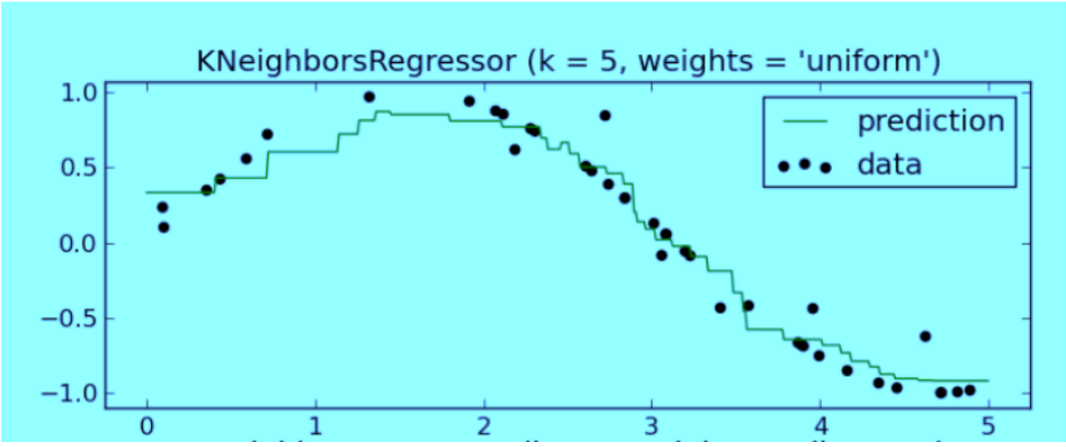
TheOutput:

```
Accuracy: 0.80
Confusion Matrix:
[[90 15]
 [20 54]]
Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.86      0.84       105
           1       0.78      0.73      0.76        74

    accuracy                           0.80       179
   macro avg       0.80      0.79      0.80       179
weighted avg       0.80      0.80      0.80       179
```

k-Nearest Neighbors (k-NN) Regressor



1. Definition and Significance

The k-Nearest Neighbors (k-NN) Regressor is a non-parametric, instance-based machine learning algorithm used for regression tasks. Unlike k-NN classification, which predicts a class label, k-NN regression predicts a continuous output based on the average (or weighted average) of the k-nearest neighbors to the input point.

The significance of k-NN regression lies in its simplicity and effectiveness in making predictions based on proximity to known data points. It is particularly useful when the relationship between the input features and the target variable is complex and non-linear, as k-NN can capture local patterns in the data.

2. How k-NN Regression Works

k-NN regression works by following these steps:

1. **Determine the Value of k:** The first step is to choose the number of neighbors (k) to consider when making a prediction. This value can be tuned using cross-validation.
2. **Calculate the Distance:** For a given input data point, the algorithm calculates the distance to all data points in the training set. Common distance metrics include Euclidean distance, Manhattan distance, and Minkowski distance.
3. **Identify Neighbors:** The algorithm then selects the k data points in the training set that are closest to the input point, based on the calculated distance.
4. **Predict the Output:** The predicted value for the input point is computed as the average of the target values of the k nearest neighbors. In some cases, a weighted average is used, where closer neighbors have a higher influence on the prediction.
5. **Return the Prediction:** The output is returned as the predicted value for the input data point.

3. Applications of k-NN Regressor

The k-NN Regressor can be applied in various domains where predicting a continuous variable is necessary:

- **House Price Prediction:** k-NN regression can predict house prices based on features like location, square footage, number of bedrooms, etc., by averaging the prices of nearby houses.
- **Stock Price Prediction:** It can be used to predict future stock prices by averaging the prices of similar stocks in the past.
- **Weather Forecasting:** k-NN can be employed to predict temperatures or other weather conditions by analyzing historical data from nearby locations.
- **Customer Behavior Analysis:** k-NN regression can predict customer behavior, such as spending patterns, by analyzing the behavior of similar customers.

4. Advantages and Limitations

4.1. Advantages

- **Simplicity:** k-NN regression is easy to implement and understand, making it an accessible algorithm for beginners and practical for many applications.
- **No Assumptions:** Unlike linear regression, k-NN does not assume a specific form for the relationship between the features and the target variable, making it flexible in capturing complex patterns.
- **Adaptability:** The algorithm easily adapts to new data since it doesn't require a training phase. All computation occurs during the prediction phase.

4.2. Limitations

- **Computationally Intensive:** k-NN regression can be slow, especially with large datasets, because it requires calculating the distance to every data point in the training set.
- **Curse of Dimensionality:** As the number of features increases, the distance between points becomes less meaningful, leading to decreased performance.
- **Sensitive to Noisy Data:** Outliers or noisy data points can significantly affect the predictions, especially when k is small.

5. Practical Implementation Using a Housing Dataset

5.1. Introduction to the Dataset

Let's demonstrate k-NN regression using a housing dataset, where the goal is to predict house prices based on various features like the number of rooms, square footage, and location.

5.2. Implementation of k-NN Regression

Below is a Python implementation using the scikit-learn library:

```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Load the dataset (assuming it is available as a CSV file)
url="https://raw.githubusercontent.com/selva86/datasets/
master/BostonHousing.csv"
df = pd.read_csv(url)
# Select relevant features and the target variable
features = ['rm', 'lstat', 'ptratio', 'indus', 'nox',
'tax', 'crim']
target = 'medv'
X = df[features]
y = df[target]
```

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize and train the k-NN regressor
knn = KNeighborsRegressor(n_neighbors=5,
metric='minkowski', p=2)
knn.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error: {mse:.2f}")
print(f"R-squared: {r2:.2f}")
```

The Output:

```
Mean Squared Error: 13.41
R-squared: 0.82
```

Variants of k-Nearest Neighbors (k-NN)

The standard k-NN algorithm is a powerful tool for both classification and regression tasks, but it has some limitations, such as sensitivity to the choice of k and the influence of outliers. To address these issues, several variants of k-NN have been developed. This section explores some of the most popular variants: Weighted k-NN, Distance-Weighted k-NN, and Approximate k-NN.

1. Weighted k-NN

Weighted k-NN is a modified version of the traditional k-NN algorithm that addresses one of the key issues: the equal contribution of all neighbors, regardless of their distance from the query point. In standard k-NN, the predicted output is simply the average or majority vote of the k nearest neighbors. However, this can lead to inaccuracies if the nearest neighbors vary widely in their distances.

Intuition: The closer a neighbor is to the query point, the more relevant it should be in making predictions. Weighted k-NN assigns a weight to each of the k nearest neighbors, giving more influence to neighbors that are closer to the query point and less influence to those that are farther away.

Algorithm:

1. **Calculate Distance:** For a given query point, compute the distance between the query point and all points in the training set.
2. **Select k Nearest Neighbors:** Identify the k nearest neighbors based on the calculated distances.
3. **Assign Weights:** Assign a weight to each neighbor, typically using a kernel function such as the inverse distance function. The weight is higher for points closer to the query point.
4. **Predict Output:** The final prediction is made by taking a weighted average (for regression) or weighted vote (for classification) of the k nearest neighbors.

Example Implementation:

```
import numpy as np
from sklearn.neighbors import KNeighborsRegressor

# Define the kernel function (inverse distance)
def inverse_distance_weight(distances):
    with np.errstate(divide='ignore'):  # Handle division
by zero
        weights = 1 / distances
    weights[distances == 0] = 1.0  # Assign weight of 1 to
zero distances
    return weights

# Example distances and target values
distances = np.array([0.1, 0.2, 0.3])
targets = np.array([1, 0, 1])

# Calculate weights and weighted prediction
weights = inverse_distance_weight(distances)
weighted_prediction = np.dot(weights, targets) /
weights.sum()
print(f"Weighted Prediction: {weighted_prediction}")

# The Output:Weighted Prediction: 0.7272727272727274
```

Advantages:

- Weighted k-NN reduces the impact of outliers and distant neighbors.
- It provides more accurate predictions in cases where the nearest neighbors vary significantly in distance.

Limitations:

- It still requires calculating distances to all points in the training set, which can be computationally expensive.

2. Distance-Weighted k-NN

Distance-Weighted k-NN is a specific type of weighted k-NN where the weights are directly proportional to the inverse of the distance from the query point. The idea is that points closer to the query point should have a stronger influence on the prediction.

Intuition: The closer a data point is to the query point, the more likely it is to share similar characteristics. By assigning weights inversely proportional to the distance, distance-weighted k-NN emphasizes the importance of closer neighbors.

Algorithm:

1. **Calculate Distance:** Compute the distance from the query point to all points in the training set.
2. **Assign Weights:** Calculate the weight for each neighbor using the inverse of the distance.
3. **Make Prediction:** Use the weighted average or weighted vote of the nearest neighbors to make the final prediction.

Example: Given a set of neighbors at different distances, we can calculate the weights as the inverse of the distance and then predict the output by taking a weighted average.

Advantages:
- Reduces the impact of neighbors that are farther away.
- Can lead to more accurate predictions in datasets with non-uniform distributions.

Limitations:
- Similar to weighted k-NN, it can be computationally intensive, especially with large datasets.

3. Approximate k-NN

Approximate k-NN is designed to speed up the k-NN algorithm by finding an approximate set of nearest neighbors rather than the exact k nearest neighbors. This is particularly useful in scenarios with large datasets or high-dimensional data, where exact nearest neighbor search is computationally prohibitive.

Intuition: Instead of finding the exact k nearest neighbors, approximate k-NN uses data structures and algorithms that can quickly identify a subset of neighbors that are likely to be close to the query point. While these may not be the exact nearest neighbors, they are sufficiently close for practical purposes.

Common Techniques:

- **KD-Trees:** A data structure that partitions the data into a tree-like structure, enabling faster search for nearest neighbors.
- **Ball Trees:** A tree structure that organizes data points into a series of nested hyperspheres, making it easier to find nearby points.
- **LSH (Locality-Sensitive Hashing):** A hashing-based technique that groups nearby points together in the same bucket, allowing for quick identification of close neighbors.

Advantages:
- Significant speed-up in the nearest neighbor search.
- Useful for real-time applications or when dealing with large datasets.

Limitations:
- The trade-off between speed and accuracy: Approximate k-NN may not always find the exact nearest neighbors, potentially leading to slightly less accurate predictions.

Brute Force k-NN

Definition and Significance

Brute Force k-Nearest Neighbors (k-NN) is a straightforward and intuitive method for implementing the k-NN algorithm. In this variant, the algorithm computes the distance between the query point and every other point in the dataset to determine the k closest neighbors. The term "brute force" refers to the exhaustive nature of this approach, as it doesn't incorporate any optimization techniques to reduce the computational load. Despite its simplicity, Brute Force k-NN remains significant in scenarios with small to medium-sized datasets where computational resources are not a limiting factor.

How Brute Force k-NN Works

The Brute Force k-NN algorithm operates by following these steps:

1. **Distance Calculation:** For a given query point, the algorithm calculates the distance between this point and every other point in the dataset. Common distance metrics include Euclidean, Manhattan, and Minkowski distances.
2. **Neighbor Selection:** Once the distances are computed, the algorithm sorts all points based on their distance to the query point and selects the top k points as the nearest neighbors.
3. **Classification/Regression:** In classification tasks, the algorithm assigns the class label based on a majority vote among the k nearest neighbors. In regression tasks, the predicted value is typically the average of the values of the k nearest neighbors.

Applications of Brute Force k-NN

While Brute Force k-NN is computationally expensive for large datasets, it is still used in various applications where the simplicity and interpretability of the algorithm outweigh its inefficiency:

- **Small-Scale Machine Learning Tasks:** Brute Force k-NN is often used in educational contexts and small-scale projects where the dataset size is manageable.
- **Prototype Development:** It is useful in the early stages of algorithm development or prototyping, where quick implementation is more critical than optimization.
- **Benchmarking:** Brute Force k-NN serves as a baseline for evaluating the performance of more complex and optimized algorithms.

Advantages and Limitations

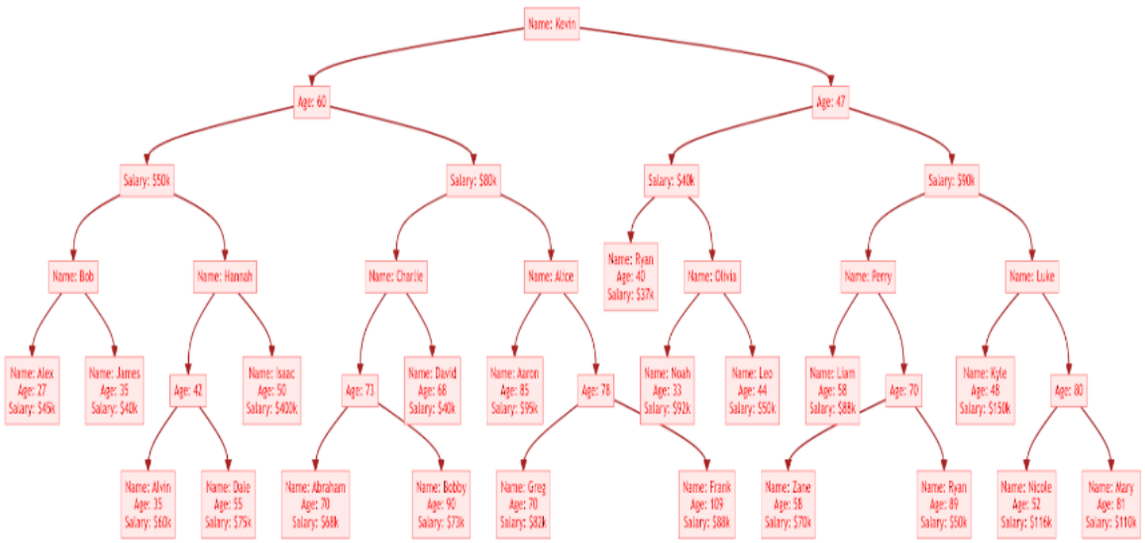
Advantages:

- **Simplicity:** The algorithm is easy to understand and implement, making it an excellent choice for beginners and quick prototypes.
- **No Assumptions:** Brute Force k-NN does not make any assumptions about the underlying data distribution, which makes it a versatile algorithm.

Limitations:

- **Computationally Expensive:** The brute force nature of the algorithm means it has a time complexity of O(n) for distance calculations, making it inefficient for large datasets.
- **Memory-Intensive:** As the dataset size grows, the memory required to store all distances increases significantly.
- **Scalability Issues:** The algorithm does not scale well with the size of the dataset, leading to longer processing times as the dataset grows.

k-Dimensional Tree (k-D Tree)



Definition and Significance

A k-Dimensional Tree, commonly known as a k-D Tree, is a data structure used to organize and search for points in a multi-dimensional space. It is particularly effective for nearest neighbor searches, a common problem in machine learning where the goal is to find the closest data points to a given query point. The k-D Tree is a binary tree, where each node represents a point in the space, and the tree is constructed by recursively partitioning the data points along different dimensions. This hierarchical structure enables efficient search and retrieval of points, making k-D Trees valuable in handling high-dimensional data often encountered in fields like computer vision, natural language processing, and bioinformatics.

How k-D Tree Works

The construction of a k-D Tree involves the following steps:

1. **Dimension Selection:** The data is split by choosing a dimension (axis) at each level of the tree. This can be done cyclically, by choosing the dimension with the highest variance, or by another criterion.
2. **Data Partitioning:** The data points are split into two groups based on the median value of the selected dimension. This median becomes the root node of the tree at that level.
3. **Recursive Division:** The process is repeated recursively for each of the two groups, with the next level splitting on a different dimension. This continues until all points are partitioned into leaf nodes.

For nearest neighbor search, the k-D Tree is traversed from the root to the leaf node that contains the query point. During backtracking, the algorithm checks whether any closer points might exist in other subtrees, ensuring that the nearest neighbors are found.

Applications of k-D Tree

k-D Trees are used in various real-world applications where efficient nearest neighbor search is crucial:

- **Computer Vision:** In tasks like image recognition and object detection, k-D Trees enable efficient searches through large image databases to find similar features or objects.
- **Geographic Information Systems (GIS):** k-D Trees help in quickly locating the nearest geographical features or points of interest, such as restaurants or hospitals, based on a user's current location.
- **Bioinformatics:** k-D Trees are used to find similar genetic sequences or protein structures, assisting in the identification of functional relationships and evolutionary patterns.

Advantages and Limitations

Advantages:

- **Efficient Search:** k-D Trees allow for efficient nearest neighbor search in high-dimensional spaces by reducing the search space at each tree level.
- **Handling High-Dimensional Data:** k-D Trees are particularly useful in scenarios where the data has multiple dimensions, as they mitigate some of the challenges posed by the "curse of dimensionality."

Limitations:

- **Performance Degradation in High Dimensions:** As the number of dimensions increases, the performance of k-D Trees can degrade, especially when the data is not uniformly distributed. The tree can become unbalanced, leading to longer search times.
- **Poor Suitability for Dynamic Datasets:** Inserting or deleting points in a k-D Tree is computationally expensive and may require substantial restructuring of the tree, making it less ideal for dynamic datasets.
- **Curse of Dimensionality:** Although k-D Trees help manage high-dimensional data, they are not immune to the curse of dimensionality, where the volume of space increases exponentially with the number of dimensions, potentially diminishing the tree's efficiency.