K Nearest Neighbor (K-NN)

Table of Contents

Harrison Lloyd

CSCI 411

Professor Tillquist

8 Dec 2024

I. Introduction

K-NN, short for K-Nearest Neighbors, is a machine learning algorithm that classifies or predicts data based on proximity. It works on the idea that data points close together in terms of distance (i.e., two points on a graph) are likely to have similar characteristics. For example, in a group of animals, if one unknown animal is near others and is labeled as a cat, the algorithm will likely classify it as a cat, too. This makes K-NN useful for tasks like grouping similar items or making predictions based on patterns in the data.

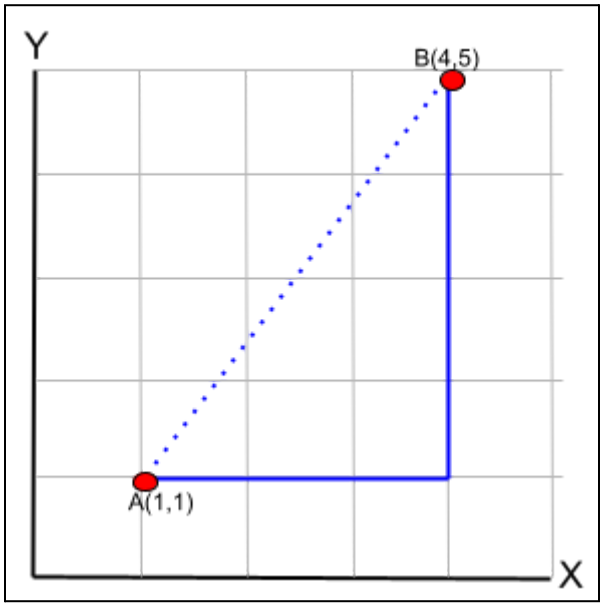
Evelyn Fix and Joseph Hodges developed this algorithm in 1951 as part of their research for the U.S. Air Force. Their work introduced the concept of nearest-neighbor classification, which laid the math for non-parametric approaches to pattern recognition. Sixteen years later, in 1967, Thomas Cover and Peter Hart expanded on the algorithm by broadening its use cases and going deeper into its theory. K-NN is important because of its simplicity and flexibility. It isn't too hard to understand and implement, making it well-versed in solving problems for image recognition, predicting customer behavior, recommendation systems, and detecting diseases through X-ray images. Furthermore, it can also be adaptable to data. For example, since it keeps all of its data in memory, it automatically includes and uses it to improve future predictions when a new data point is added.

While K-NN is an effective algorithm, it can be inefficient when handling large datasets or higher dimensions using the naive method. To combat this, the data structure known as K-d tree (k-dimensional tree) was introduced. It was invented by Jon Louis Bentley in 1975 and is used to organize points in a k-dimensional space. The K-d tree is important because it speeds up the process of finding nearest neighbors, allowing it to be more applicable when larger datasets are needed.

II. Intuition

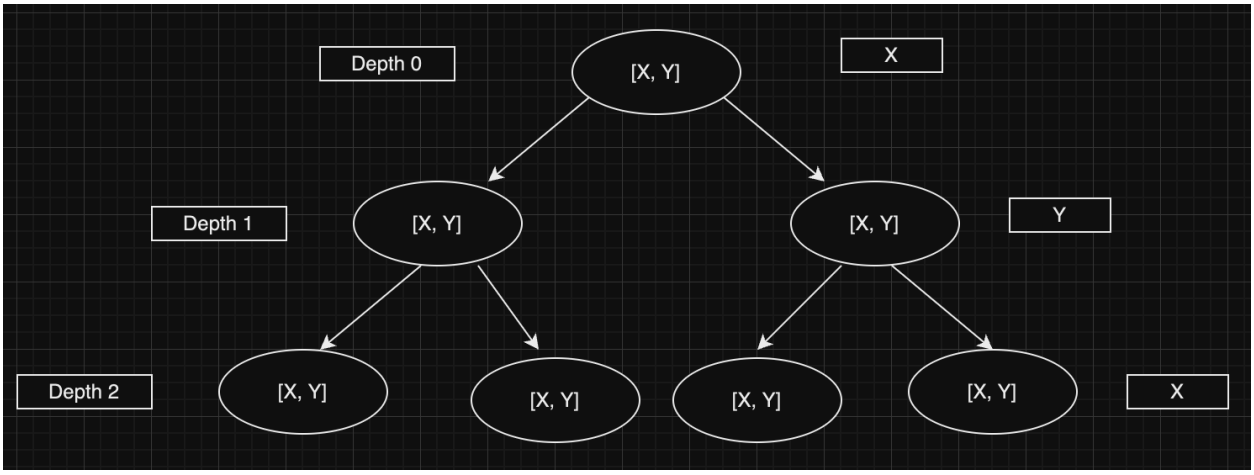
The wonderful part about the K Nearest Neighbors algorithm is its simplicity. It has three key components: calculating the distance, finding the nearest K neighbors, and making a prediction. The algorithm first calculates the distance between the query (the point to classify) and all other points on the dataset. There are several ways to calculate the distance, but the two most commonly used methods are the Euclidean and the Manhattan distance. The Euclidean distance formula is written as follows: d(p,q) = $\sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$. It measures the shortest possible line between two points and best suits data with lower dimensions. Another method to find this distance is the Manhattan distance, written as d(p,q) = $\sum_{i=1}^{n} |p_i - q_i|$. The Manhattan distance differs from the Euclidean in that it measures the horizontal and vertical movement between two points, making it better when movement along axes is more relevant. The next step is defining K. The K parameter determines the number of neighbors the algorithm considers when making predictions, so choosing the right K is important. When K is a small number, such as 1, it can be sensitive to outliers and noise, making it vulnerable to overfitting. On the other

hand, a large K value can smooth predictions by averaging over many points, however, this can lead to underfitting. Another important note is that the K value should always be odd. This is because when K is even, i.e., K=2, there's a chance it could be forced to break ties where an equal number of neighbors belong to different classes. Once the distance has been calculated and K has been defined, the algorithm identifies the K closest points to the query point. In terms of classification, the query point gets assigned to the class that occurs most often among its K neighbors. Meanwhile, for regression, the value gets predicted by averaging the values from the nearest K neighbors. Here is how to use the Euclidean to find the distance from point A to point B, $d(A, B) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \rightarrow d(A, B) = \sqrt{(4 - 1)^2 + (5 - 1)^2} \rightarrow d(A, B) = \sqrt{9 + 16} \rightarrow 5$.



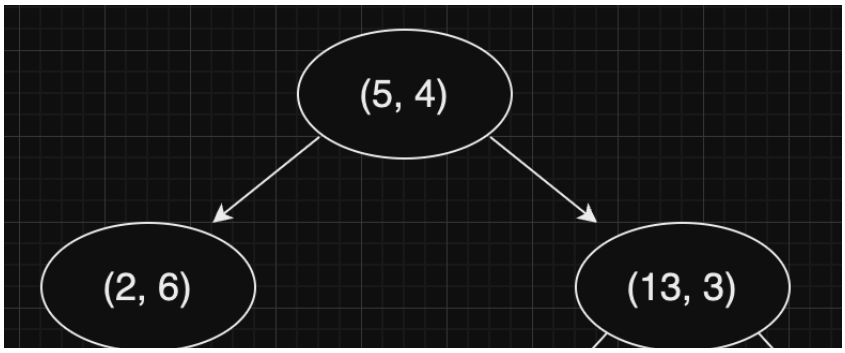
Using the same graph, here is how to use the Manhattan to find the distance from point A to point B, $d(A, B) = |X_2 - X_1| + |Y_2 - Y_1| \rightarrow d(A, B) = |4 - 1| + |5 - 1| \rightarrow d(A, B) = 3+4 = 7$.

The K-d tree is a binary search tree (BST) that organizes and searches points in a K-dimensional space. Imagine you have a set of points spread across a 2D plane and want to quickly find the nearest neighbors to a given query point. Searching through each point individually, as the naive method does, becomes increasingly less efficient as the dataset gets larger or the number of dimensions increases. Once more, imagine that we have a space with a number of points. We start by finding the median point's X values and splitting the set into two branches. In each branch, we will find a median along the Y-axis and divide it into branches once again. We will continue this alternating splitting pattern along the X and Y axes until each cell only contains one point, giving us the finished tree structure. This is much more efficient as we divide the search space and focus only on the regions most likely to contain the nearest neighbors. An example of a K-d tree in a real-world situation could be used for a dataset in a hospital. For instance, say a hospital wanted to record each patient's blood type, white/red blood cell count, and blood pressure. Then, a new patient came in with a mysterious illness. Using a K-d tree, you could quickly discover this unknown illness by looking at patients with similar characteristics in blood type, cell count, and blood pressure (nearest neighbors). The underlying structure of the k-d tree is seen as follows:

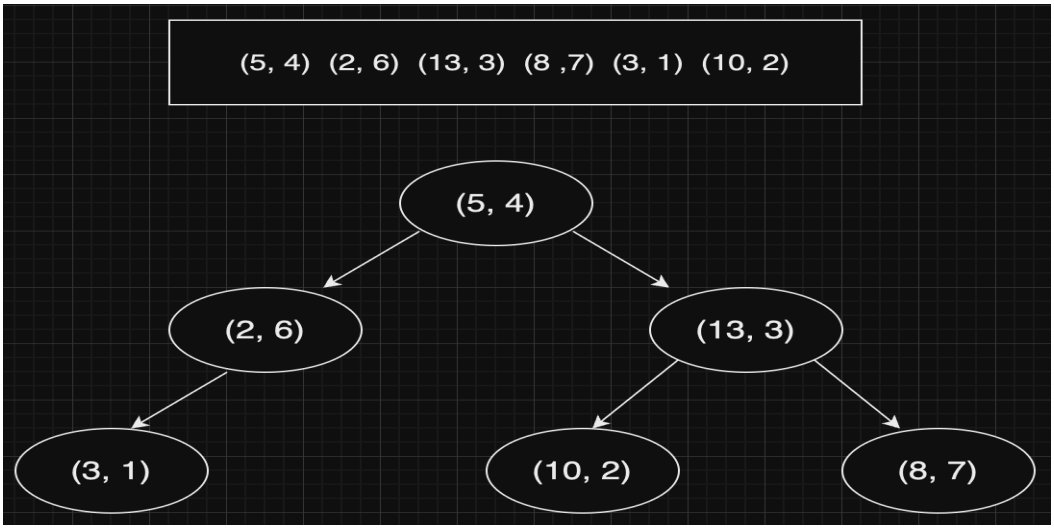


Consider this scenario: We have the points (5, 4), (2, 6), (13, 3), (8, 7), (3, 1), and (10, 2). We start constructing the tree by sorting these points along the X-axis to find the median value, which, in this case, is (5, 4).

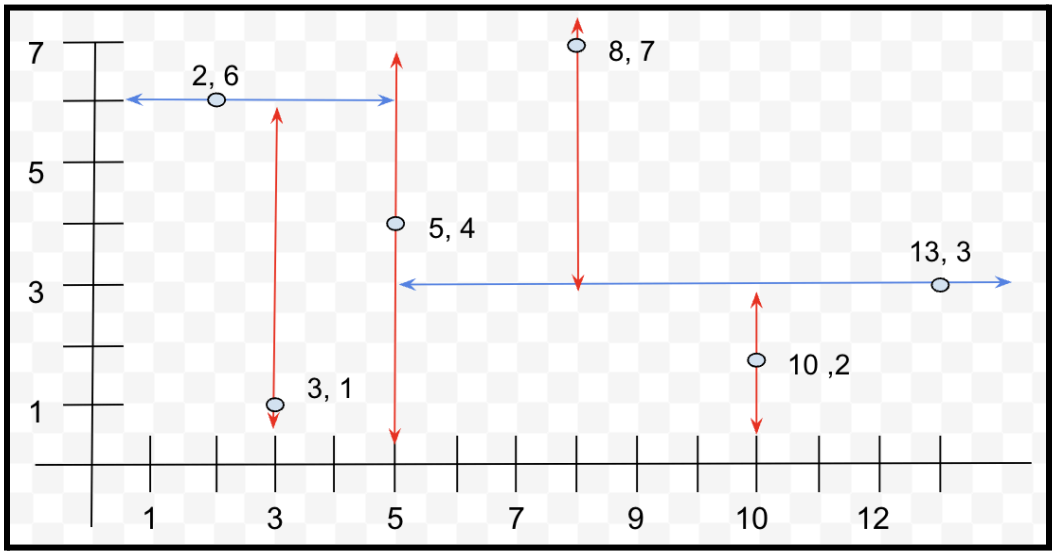
From here, the point (5, 4) became the root, and those with X values smaller than five branches to the left, while those greater than five branches to the right. Next, within each branch, the points are sorted along the Y-axis, where we again find the median to divide it further. Here, the points to the left of the root are (2, 6), and (3, 1), while the points to the right are (13, 3), (10, 2), and (8, 7). The median on the left would be (2, 6) and the median on the right (13, 3).



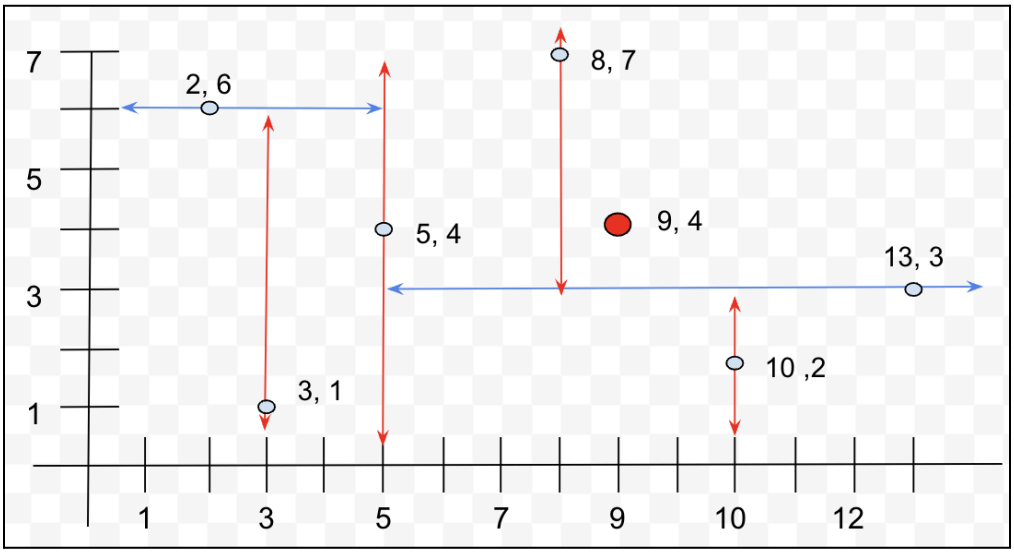
This pattern continues until we run out of points, leaving us with the final structure:



This image shows the K-D tree being split on a 2D graph. The first split, made by the root (5, 4), divides the space vertically at X=5. Then, the left child (2, 6) splits the left region horizontally at y=6, while the right child (13, 3) splits the right region horizontally at Y=3. This continues until each region contains just a single point.



Suppose our target point is at (9, 4), and our goal is to find the nearest neighbor. We walk down the tree and calculate the shortest distance found so far. On the way down the tree from the right side, we missed the point (10, 2). This happened because when we compared the second node of the tree (13, 3), the Y coordinate was smaller than that of the Y coordinate belonging to the target. After we've gone as deep as we can in the tree, we recurse back. Here, we have a candidate's closest neighbor (8, 7), but it's still possible that there's an even closer neighbor. As we recurse back, we also calculate the distance from the part of the tree that hasn't yet been calculated.



III. Pseudocode of naive approach

```
Struct
        Val = 0   // value of the point (0 or 1)
        X = 0   // X-coordinate
        Y = 0   // Y-coordinate
        Distance = 0   // Distance from point to query

Function comparison(pointA, pointB)   // Sort points by distance
        Return pointA.distance < pointB.distance

// Calculate the distance from each point to the query
Function classifyPoint(arr, n, k, test_point)
        For i from 0 to n-1
                Arr[i].distance = sqrt(
                        pow((arr[i].x - test_point.x), 2) +
                        pow((arr[i].y - test_point.y), 2) )

        sort(arr, n, comparison) // Sort the array of points by distance using the comparison function

        num_group0 = 0
        num_group1 = 0
        // Count the occurrences of the two groups among the K closest points
        For i from 0 to k-1
                If arr[i] == 0
                        num_group0++
                Else if arr[i] == 1
                        num_group1++
If num_group0 > num_group1  // Return the class with the highest frequency
                Return 0;
        Else
                Return 1
```

Tests:
Here are the data points on the graph:
(0,1,12), (0,2,5), (1,5,3), (1,3,2) (0,3,6), (1,7,2), (1,6,1), (0,3,10), (1,4,2), (0,2,11), (1,2,5)

1) Query 1 at {0, 10, 10}, expected output: 0
2) Query 2 at {0, 3, 4}, expected output: 1
3) Query 3 at {0, 5, 5}, expected output: 0
4) Query 4 at {0, 7, 7}, expected output: 1
5) Query 5 at {0, 9, 8}, expected output: 0

III. Pseudocode of k-d tree approach

```
Node:
        X       // X-coordinate
        Y       // Y-coordinate
        Label  // its category

KD:
        Point  // node object
        L    //  Left child
        R    //  Right child

function distance(a, b):
        return (a.x - b.x)^2 + (a.y - b.y)^2   // uses the Euclidean to find the distance

function nearest(target, a, b):      // Finds which of two nodes is closer to the target
        if a == null:
                return b
        if b == null:
                return a

        distA = distance(target, a.point)   // Finds the distances from the target to a and b
        distB = distance(target, b.point)
        if distA < distB:
                return a else: return b

function buildKDTree(points, depth):       //recursively builds K-d tree
        if points empty:
                return null



        axis = depth % 2   // This alternates between X and Y

         if axis == 0:
                sort(points by x)
        else:
                sort(points by y)

        median_index = length(points)/2   // Finds the median point
```

```
        tree = new KD(medianPoint)  // Creates new K-d tree node for the median point
        left_point = [ ]
        right_point = [ ]

        for i = 0 to medianIndex - 1:
                append points[i] to left_point    //adds to the left of the tree

        for i = medianIndex + 1 to length(points) - 1:
                append points[i] to right_point     //adds to the right of the tree

        tree.L = buildKDTree(left_point, depth + 1) // Recursively builds the tree
        tree.R = buildKDTree(right_point, depth + 1)

function nearNeigh(root, target, depth):
        if root is null:
                return null

        axis = depth % 2   // This alternates between X and Y

         // Finds which branch to search first
        if (axis == 0 and target.x < root.point.x) or (axis == 1 and target.y < root.point.y):
                next_branch = root.L
                other_branch = root.R
        Else:
                next_branch = root.R
                other_branch = root.L

         // Recursively searches in the other branch
        temp = nearNeigh(next_branch, target, depth + 1)


        // Finds the nearest node between the current node and the best so far
        best = nearest(target, temp, root)

        radius = distance(target, best.point)   // Distance to the best so far

        if axis == 0:
                dist_plane = (target.x - root.point.x)^2
        Else:
                dist_plane = (target.y - root.point.y)^2
```

```
        // If the split is within the radius, then search the other branch
        if radius >= dist_plane:
                temp = nearNeigh(other_branch, target, depth + 1)
                best = nearest(target, temp, best)

        return best
```

Tests:

Dataset Used: wdbc.data from UC Irvine (30 features, labels: 0=benign, 1=malignant)

1) The input query is at index 10. The expected output is 0.
2) The input query is at index 50. The expected output is 0.
3) The input query is at index 100. The expected output is 0.
4) The input query is at index 200. The expected output is 0.
5) The input query is at index 300. The expected output is 0.

IV. Asymptotic Run Times

The run time of K Nearest Neighbors is O(n*d), where n represents the number of data points, and d represents the number of dimensions. This is when there is only a single query point, however, when there is more than one, the run time increases to O(q*n*d). When using a K-d tree, the run time is O(d * log(n)) when there's only one query, and otherwise, O(d * q * log(n)).

Presentation

Room 1 -
https://docs.google.com/document/d/1LUiCef1BnCxdWCEis_wbScdK4tIfR3fxYuvnwdDimYE/edit?usp=sharing

Here is the Google Slide I used for my presentation
https://docs.google.com/presentation/d/1P65WVJPIPFQsCMw9q7Wg9pMbMAmAB8JYBdx_Re5o2mg/edit#slide=id.g316a0e73744_0_10

From my peer's input, I improved upon my previous work by finding a strong dataset from UC Irvine and clearing up any previously confusing or misunderstood information.

Work Cited

"Search and Insertion in K Dimensional Tree." *GeeksforGeeks*, GeeksforGeeks, 13 June 2023, www.geeksforgeeks.org/search-and-insertion-in-k-dimensional-tree/.

Ibm. "What Is the K-Nearest Neighbors Algorithm?" *IBM*, 28 Oct. 2024, www.ibm.com/topics/knn#:~:text=The%20k%2Dnearest%20neighbors%20(KNN)%20algorithm%20is%20a%20non,used%20in%20machine%20learning%20today.

GeekforGeeks. "K-Nearest Neighbor(KNN) Algorithm."*GeeksforGeeks*, 15 July 2024, www.geeksforgeeks.org/k-nearest-neighbours/.

Christopher, Antony. "K-Nearest Neighbor." *Medium*, The Startup, 3 Feb. 2021, medium.com/swlh/k-nearest-neighbor-ca2593d7a3c4.