Predicting Cancer Cells Diagnosis Using K Nearest-Neighbors

June 15, 2024

Marquez, Camila Kin
Concordia University
Montréal, Québec, Canada
ID: 40234852

***Abstract*—This paper describes and analyzes the efficiency of the data structures and algorithms used to predict the diagnosis of cancer cells. A K-D tree data structure was used for the organization of the training data. The k Nearest Neighbour (k-NN) machine learning model was used to predict the diagnosis of the testing data set. This implementation resulted in average diagnostic accuracy in the range of 85-100%. No significant correlation was found between the number of training and testing instances and the accuracy and computational efficiency of the program. All scenarios tested, on average, similarly to each other. However, the running time did increase depending on the number of testing instances and on the number of nearest neighbors found for each one.**

I. Introduction

A machine learning (ML) model was developed to predict the diagnosis of cancer cells based on The Breast Cancer Wisconsin Dataset[1], which is composed of 569 instances. Each instance of data includes an ID number, a diagnosis (B = benign, M = malignant) and a set of 10 attributes used to diagnose the cell. The ML model is composed of a K-D tree data structure that uses k-NN algorithm to predict the diagnosis of a given instance. For a given N instances, a 4:1 ratio of training:testing data was used. The accuracy of the model was assessed as a percentage of correct predictions from the testing data set; the computational efficiency was defined by the running time of the k-NN search and prediction; and the scalability of the solution was evaluated based on how flexible it is for different numbers of attributes and instances. The model was re-trained every time it was tested on a different number of instances (N = 100, 200, 300, 400, 500) and a different k-NN search (k = 1, 5, 7 for each N). This paper describes the ML model and conducts a thorough analysis of its performance, based on accuracy, timing and scalability.

[1] Wolberg,William, Mangasarian,Olvi, Street,Nick, and Street,W.. (1995). Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository. https://doi.org/10.24432/C5DW2B.

II. Description of Model

A. Main Data Structures

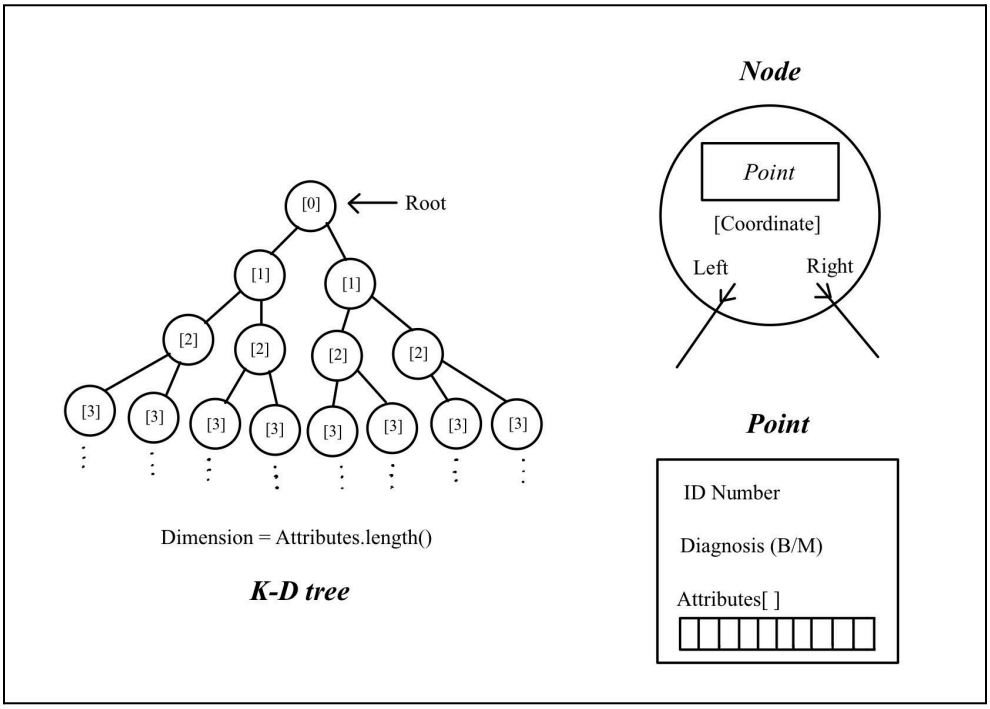

Figure 1: Annotated diagram of *K-D Tree* data structure and building blocks; *Node* and *Point.*

The main data structure of this ML model is a K-D Tree, whose basic structure can be seen in Figure 1. This data structure is a full, balanced binary tree which organizes data by comparing different attributes at each depth level. The building blocks of the tree are Nodes, which contain a Point and have references to the left and right Nodes. Finally, a Point is defined specifically for this application, as an object with an ID, a diagnosis and a list of attributes, that represents a single instance in the data set. The following section describes how the K-D tree is built recursively and used to find the k nearest neighbors to an external Point.

B. Main Algorithms Operating on the Data Structures

a. Reading and processing data from file

The flowchart depicted in Figure 2 shows the process through which data is read from the file, separated into training and testing arrays, and finally normalized. Although this algorithm is not applied directly on the K-D tree, it is imperative that data undergoes this process before building the tree upon it. The most important part for increasing accuracy is the normalization of the data, as it gives equal weights to all attributes when distance is being calculated to find the k-NN to a Point. The min-max normalization method was used, and the pseudocode for it can be found after Figure 2.

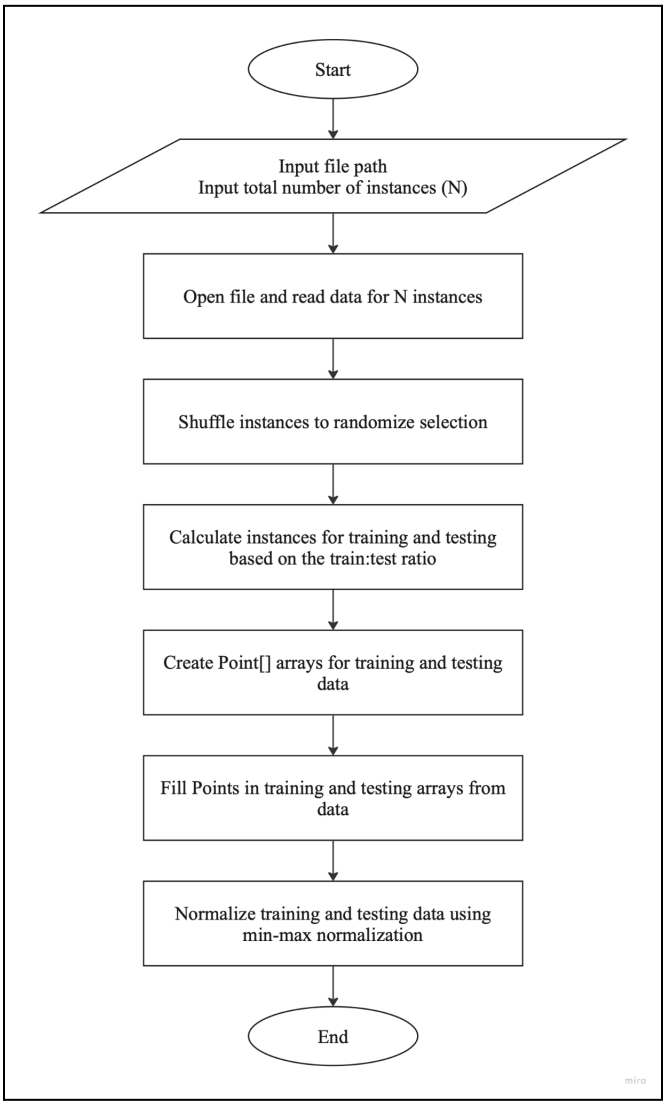

Figure 2: Flowchart description of processing of data from a file, into training and testing Point arrays

Augmented view of min-max normalization used, in the form of pseudocode:

```
function normalizeData(trainList, testList){
        (minValues, maxValues) = minmaxValues(trainList)

        for each point in trainList for j from 0 to numAttributes - 1
                x = point.attributes[j]
                point.attributes[j] = (x - minValues[j]) / (maxValues[j] - minValues[j])

        for each point in testList for j from 0 to numAttributes - 1
                x = point.attributes[j]
                point.attributes[j] = (x - minValues[j]) / (maxValues[j] - minValues[j])
}

function minmaxValues(data) {
        initialize minValues array
        initialize maxValues array

        for j from 0 to numAttributes - 1
                min = data[0].attributes[j]
                max = data[0].attributes[j]

                for each point in data
                        if point.attributes[j] < min then
                                min = point.attributes[j]
                        if point.attributes[j] > max then
                                max = point.attributes[j]

                minValues[j] = min
                maxValues[j] = max

                return (minValues, maxValues)
}
```

b. Building a balanced K-D tree

The algorithm to build a balanced K-D tree of any dimension is depicted in the flowchart in Figure 3. At each depth, the attribute used for comparison is determined and the data is sorted based on it. Then, the midpoint is found and set as the current Node, and the tree is built recursively to the left and to the right using the corresponding left and right subarrays.

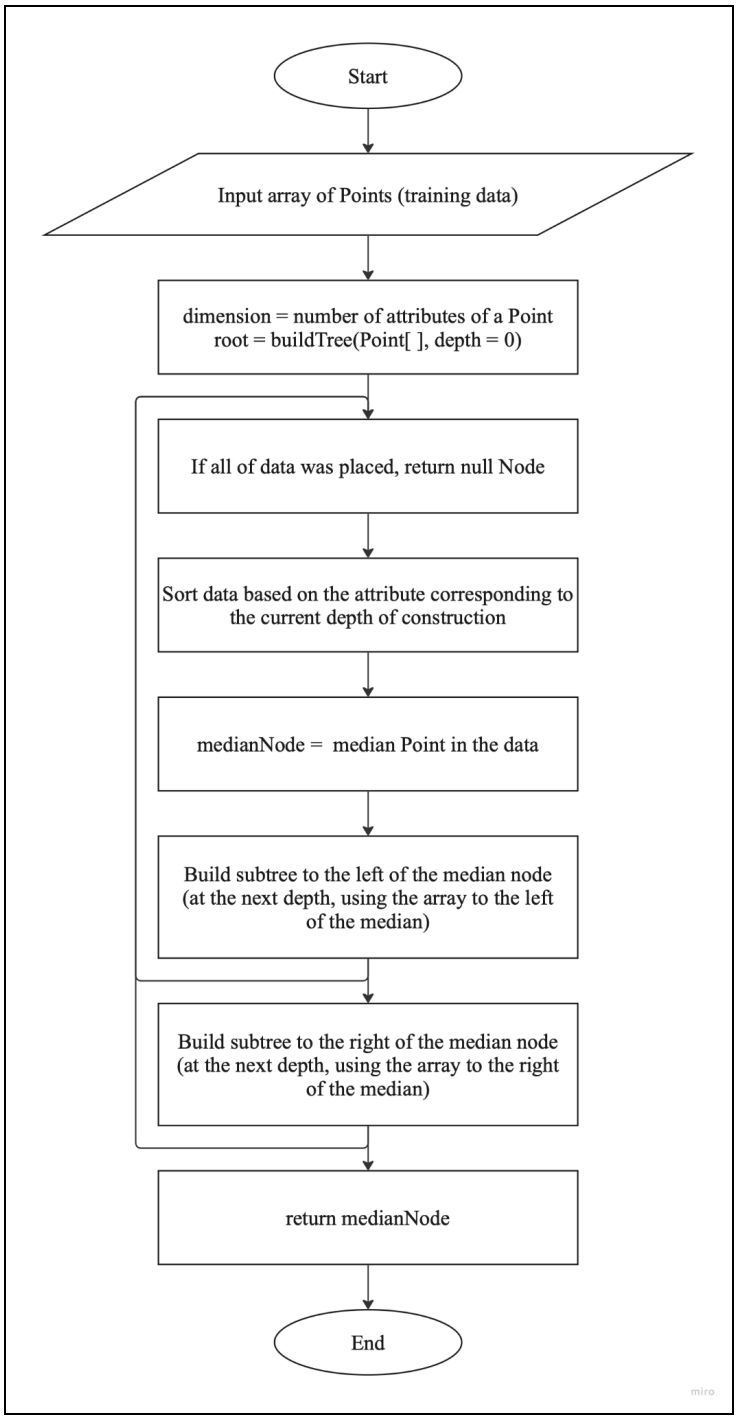

Figure 3: Flowchart description of building a balanced K-D tree

For further clarity, the pseudocode for the recursive BuildTree function is the following:

```
function buildTree(points, depth) {
        if points is empty then
                return null

        coordinate = depth % dimension
        midIndex = length of points / 2

        sort points based on attribute[coordinate]

        node = create new Node with points[midIndex] and coordinate

        node.left = buildTree(subarray of points from 0 to midIndex, depth + 1)
        node.right = buildTree(subarray of points from midIndex + 1 to end, depth + 1)

        return node
}
```

c. Testing the tree

The tree is tested by passing an array of Points (testing list) to it. The algorithm depicted as a flowchart in Figure 4 tests the accuracy and runtime of predicting diagnoses for the testing list. For every instance in the testing Points array, its k nearest neighbors are found in the K-D tree. Then, through majority voting, a diagnosis (B for benign or M for malignant) is predicted for that Point. Finally, the predicted and actual diagnoses of each Point are compared, and the total number of correct and incorrect predictions is found. The percentage of accuracy is then calculated and output.

The k-NN procedure used the Euclidean equation to calculate the distance between different points. A max-heap priority queue data structure was used to store the Nodes with the shortest distance to the testing instance.

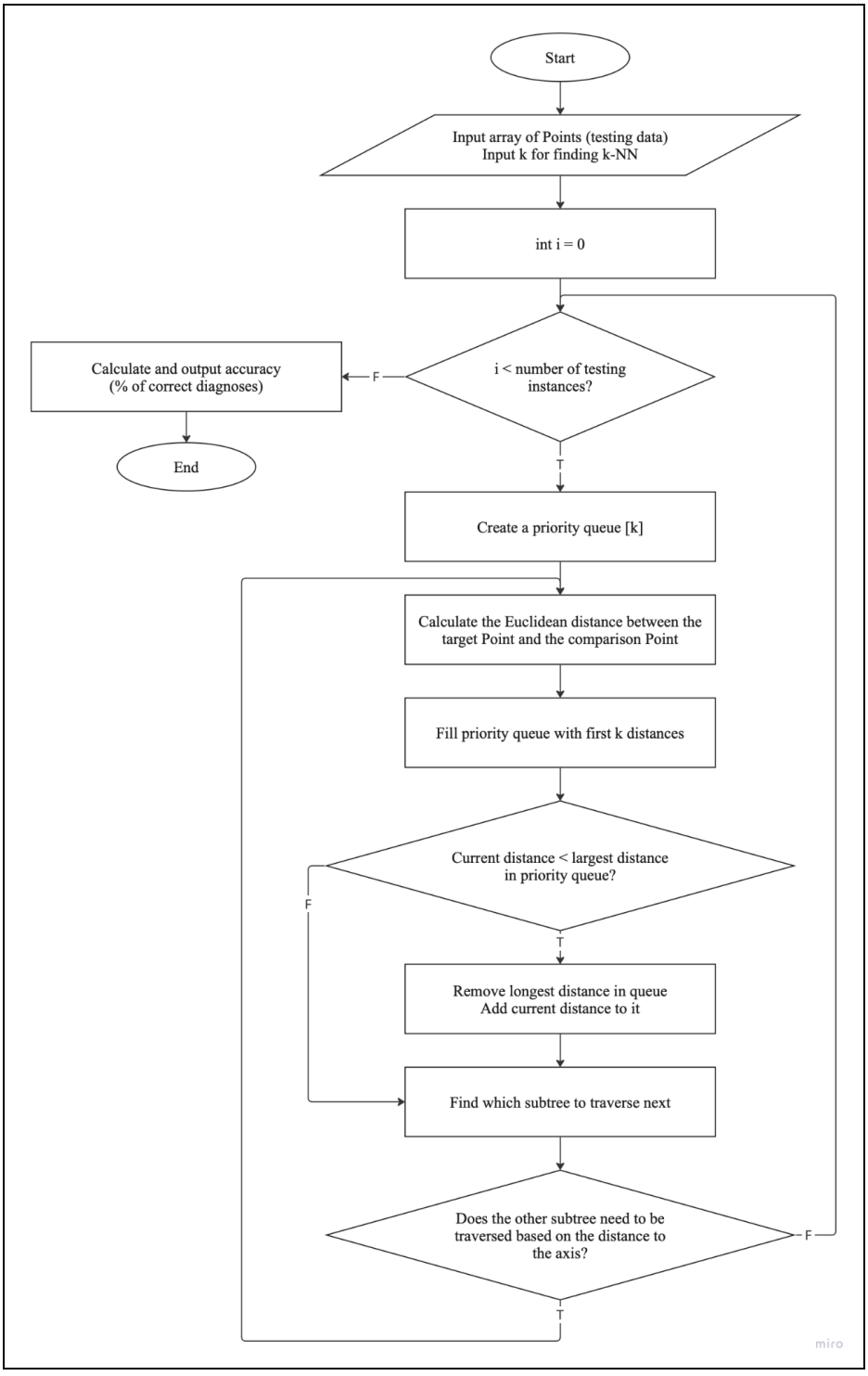

Figure 4: Flowchart description of testing the K-D tree using k-NN algorithm

The following pseudocode is provided as an augmentation for the algorithm of finding the k nearest neighbors to a testing point:

```
function kNearestNeighbours(targetPt, k){
	initialize priority queue NN of size k

	call kNearestNeighboursRecursive(root, targetPt, k, 0, NN)

	initialize nearestNeighbours array of size k
	for i from 0 to size of NN - 1
		nearestNeighbours[i] = NN.poll().point

	return nearestNeighbours
}

function kNearestNeighboursRecursive(currentNode, targetPt, k, depth, priority queueNN){
	if currentNode is null then
		return

	currentPoint = currentNode.point
	distance = Euclidean distance between targetPt and currentPoint

	if size of NN is not k then
		add currentNode to NN
	else if distance is less than distance between targetPt and furthest point in NN then
		remove farthest node from NN
		add currentNode to NN

	coordinate = depth % dimension
	left = targetPt.attributes[coordinate] < currentPoint.attributes[coordinate]

	if left then
		nextNode = currentNode.left
		otherNode = currentNode.right
	Else
		nextNode = currentNode.right
		otherNode = currentNode.left

	call kNearestNeighboursRecursive(nextNode, targetPt, k, depth + 1, NN)

	if (size of NN < k) or (distance between targetPt and currentPoint's attributes < distance
between targetPt and furthest distance in NN then
	call kNearestNeighboursRecursive(otherNode, targetPt, k, depth + 1, NN)
}
```

III. Results and Analysis

A. Accuracy of Diagnosis

Figure 5 presents the percentage accuracy of predicting the test instances' diagnosis, as a function of the total number of instances used to train the ML model. The three series presented in the graph correspond to the number k of nearest neighbors found to predict the diagnosis.

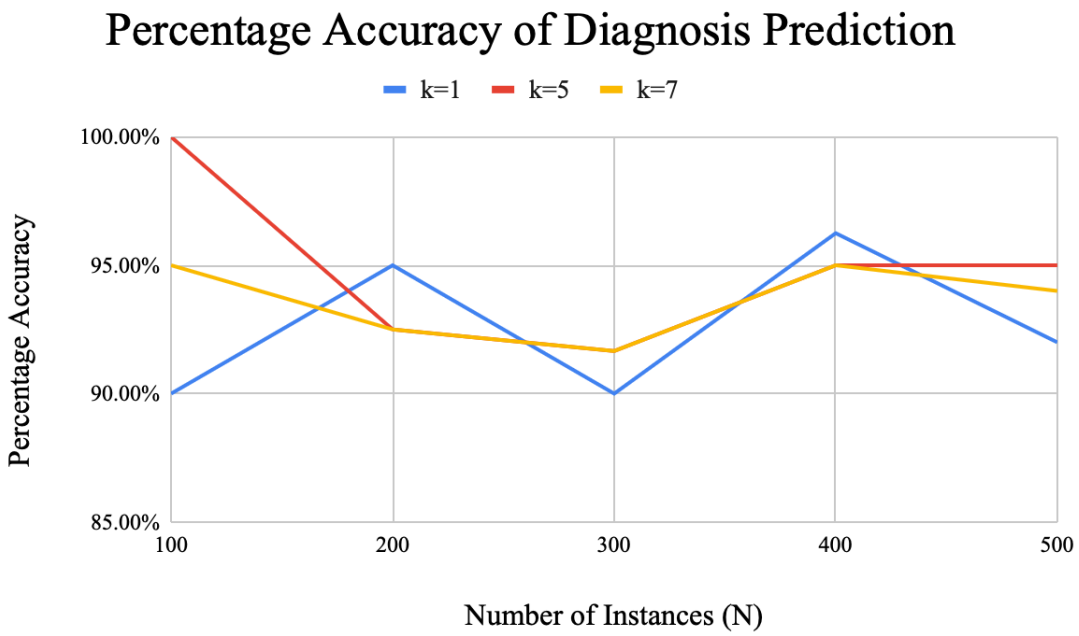

Figure 5: Graph presenting the relationship between number of instances, k-NN and percentage accuracy

There was no significant correlation found between the number of instances used for training nor for the number of nearest neighbors used. The series k=5 and k=7 even converge for values of N=200, 300 and 400. Every run has slightly different results, but all of them stay within the range of 85%-100%, and they all average at around 92% accuracy. This means that the implementation is highly accurate for even small amounts of training data, and that increasing the number of nearest neighbors does not lead to increased accuracy.

High accuracy can mainly be attributed to the normalization of data before constructing the tree, to the precise calculation of Euclidean distance between points, and to the correct traversal of the K-D tree to find the closest points based on all attributes.

B. Computational Efficiency

The running time of predicting diagnoses for the array of test instances was taken. This includes finding k nearest neighbors and asserting a diagnosis for each instance in the array, and does not include calculating the percentage accuracy. The results are presented in the graph in Figure 6, where running time (in ms) is presented as a function of the number of instances used to build the tree. As before, the three series represent different numbers of nearest neighbors used to predict the diagnosis.

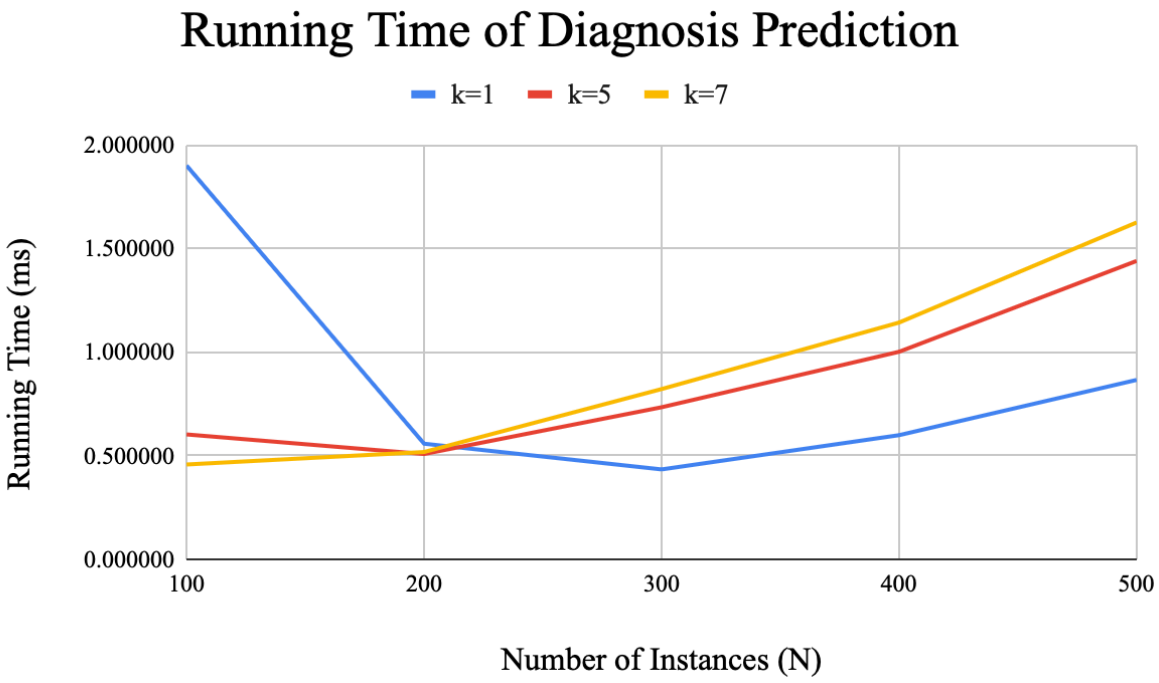

Figure 6: Graph presenting relationship between number of instances, k-NN and running time

A higher number of training instances meant a higher number of testing instances, since the ratio was maintained at 4:1 for every run. Therefore, it is expected that the runtime for predicting the diagnoses of more instances would increase too, since finding kNN for a larger number of instances requires more time. This is confirmed by the general trends of the series, where the runtime increases as N does. This linear relationship also suggests that the runtime per instance remained constant despite the number of training instances. Furthermore, as expected, runtime increased when k increased. This is because more Points' diagnoses had to be compared to predict a diagnosis, and because more nearest neighbors needed to be found for each instance. There is an outlier to this trend, for N=100 and k=1, where the runtime is higher than all others. This occurs every time the program is run, and is completely opposite to the trends otherwise

visible in runtime. As such, it is not considered when asserting the conclusion that runtime increases both with increased number of instances, as with increased number of nearest neighbors found.

C. Scalability of Solution

This solution was designed to be as scalable as possible. The K-D tree can be of any dimension needed to accommodate the attributes of a patient, and the only change that needs to be made is in the array of Points passed to the constructor of the tree. The number of attributes read from a file can also be changed in the constructor of the Ass3 class. Any specified number of instances (not larger than the available data) can be read from the file, processed, divided into a specified ratio and normalized. The tree will be built in a balanced manner despite the number of instances used to build it, and the k for finding k nearest neighbors can always be specified by the user. Overall, this solution is highly adaptable to other applications that also need a Point to represent an array of attributes on which the K-D tree is based. By tweaking the number of attributes, the train:test ratio, the number of instances used and the number of nearest neighbors found, it can be scaled to fit different needs.

IV. Conclusion

The machine learning model developed using a K-D tree and the k-NN algorithm resulted in high accuracy when predicting a patient's diagnosis. On average, this accuracy was not affected by the number of training instances used, nor by the number of nearest neighbors found. The runtime of predicting the diagnoses of an array of patients increased proportionally to both the number of instances used for training and testing, and to the number of nearest neighbors used to predict the diagnosis of each of them. The model was built with scalability in mind, allowing for adjustments to the number of attributes per patient, the dimension of the tree, the training:testing ratio and other considerations. Overall, this solution met its objective of accurate prediction of malignant or benign tumors, based on the 10 attributes provided by the data set.