Assignment 3
COMP 250, Fall 2019

Prepared by Prof. Michael Langer
Posted: Mon., Nov. 4, 2019
Due: Monday, Nov. 18, 2019 at 23:59 (midnight)

[document last modified: November 13, 2019]

General instructions

- The T.A.s handing this assignment are:
  - Abhisek (abhisek.konar@mail.mcgill.ca)
  - Brennan (brennan.nichyporuk@mail.mcgill.ca)
  - Charles (peiyong.liu@mail.mcgill.ca)
  - Akshatha (akshatha.arodi@mail.mcgill.ca)
  - Anand (anand.kamat@mail.mcgill.ca)
- Add code where instructed, namely in the ADD CODE HERE block. You may also add helper methods and extra fields if you wish.
- All import statements in your `KDTree.java` file will be removed, except for the ones that are in the starter code.
- The starter code includes a tester class that you can run to test if your methods are correct.
- Your code will be tested on valid inputs only.
- You may submit multiple times, e.g. if you realize you made an error after you submit. For any run of the Grader, you will receive the best score on your submissions.
- Late assignments will be accepted up to two days late and will be penalized by 10 points per day. If you submit one minute late, this is equivalent to submitting 23 hours and 59 minutes late, etc. So, make sure you are nowhere near that threshold when you submit.
- See the Submission Instructions at the end of this document for more details.
- Search for the keyword *updated* to find any the places where the PDF has been altered since the original posting.
- (updated Nov 5) We will remove all package declarations and import statements and add back only the imports that are given in the starter code. Therefore if you use other imports, then your code will not compile when we test it.

1 Introduction

In this assignment, we extend the powerful idea of binary search trees from one dimension to multiple dimensions. In particular, we will work with points in $\Re^k$ where k can be any positive integer. Such points might represent image data, audio data, or text. For example, machine learning methods often map such common data to *feature vectors* in a kD *feature space*, and then use these features in various ways.[1]

In binary search, one is given a key and tries to find that key amongst many that are stored in a data structure. We will examine a related but different problem for kD data. Our problem will be to take a given point, which we will call a *query point* rather than a "key", and search for the *nearest point* in the data set. In the field of Machine Learning, the nearest point is typically referred to as the *nearest neighbor*.

One way to find the nearest point would be to iterate by brute force over all data points, compute the distance from the query point to each data point, and then return the data point that has the minimum distance.[2] However, this brute force search is inefficient, for the same reason that linear search through a list is inefficient relative to binary search, since in the worst case it requires examining each data point. One would instead like to have a "nearest point" method that is must faster, by restricting the search to a small subset of the data points.

1.1 KD-Trees

One way to restrict the search for a query point is to use a tree data structure called a *kd-tree*. This is a binary tree which is similar to a binary search tree, but there are important differences. One difference between a kd-tree and a binary search tree concerns the data itself. In a binary search tree, the data points (keys) are ordered as if on a line, or in Java terms they are Comparable. In a kd-tree, there is no such overall ordering. Instead, we only have an ordering within each of the k dimensions, and this ordering can differ between dimensions. For example, consider $k = 2$, and let points be denoted (x_0, x_1). We can order two points by their x_0 value and we can order two points by their x_1 value, but we don't want to order (x_0, x_1) and (x'_0, x'_1) in general. For example, in the case $x_0 < x'_0$ but $x_1 > x'_1$, we do not want to impose an ordering of these two points.

Another important difference is how binary search trees and kd-trees are typically *used*. With a binary search tree, one typically searches for an *exact* match for the search key. With a kd-tree, one typically searches for the *nearest* data point. We will use the Euclidean distance. Give a query point $x_q \in \mathbb{Z}^k$ we want to find a data point $x[\,]$ that minimizes the squared distance,

$$sqrdist(x_q[\,], x[\,]) \equiv \sum_{i=0}^{k-1} (x_q[i] - x[i])^2.$$

We used squared distance rather than distance because taking the square root buys us no extra information about which point is closest.

Note that there may be multiple data points that are the same (squared) distance from a query point x_q. One could define the problem slightly differently by returning all points in this non-unique case. Or, one could ask for the nearest n points, and indeed this is common. We will just keep it simple and ask for a single nearest point. If multiple points lie at that same nearest distance, then any of these points may be returned.

Kd-trees which are a particular type of binary tree. Each *internal node* implicitly defines a $k-$dimensional region in $\Re^k$, which we refer to as a *hyperrectangle*, and which will be defined

[1] `https://en.wikipedia.org/wiki/Feature_(machine_learning)`

[2] Note that the nearest point may be non-unique.

below. An internal node has two (non-null) children which represent two spatially disjoint (non-intersecting) hyperrectangles that are contained within the parent hyperrectangle. We will refer to the two children as the low child and high child. [3]

Data points are stored at the leaf nodes, namely each leaf node has one `Datum` object. Internal nodes do not store references to `Datum` objects. Rather, for each internal node, we associate a set of data points, namely the `Datum` objects stored in the descendant leaf nodes.

Each internal node's hyperrectangle is defined by data points that are associated with each node, as follows. For each of the k dimensions, the *range* of the hyperrectangle in that dimension is the difference of the maximum and minimum values of the data points. An example will be given later to illustrate. The range of a node in each dimension will be used when constructing the tree, as discussed below.

1.1.1 Example

Let's consider a kd-tree node that is associated with ten 2D points ($k = 2$). The points are shown below. The horizontal axis is dimension x_0 and the vertical axis is dimension is x_1, which correspond to array representation $x[0]$ and $x[1]$. The red rectangle indicates the minimum and maximum values of the data points in the two dimensions. We will discuss the significance of these values below.

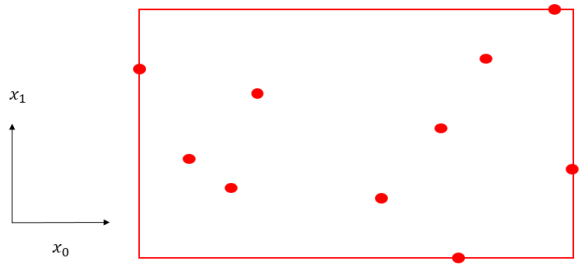

Figure 1: 10 data points in a 2D space. The red rectangle is the hyperrectangle defined by the low and high values in the two dimensions.

1.1.2 Constructing a KD-tree

We will not give a formal definition of kd-tree. Instead we give an algorithm for constructing one, given list of data points is $\Re^k$. The algorithm starts by constructing a root node and proceeds recursively. When constructing a node, we pass in an array of the data points associated with that node. The node is either a leaf or it is an internal node. Let's consider the Various possible cases.

First, the base case: if the array contains just one data point, then the node is a leaf node. Another base case is that that all the data points at a node are equal (duplicates). In this case, the node is again considered a leaf node, and the duplicate data points are removed.

In the remaining cases, the given array has more than one data point and at least two of the data points are different. These cases always define an internal node. The set of data points associated with an internal node is partitioned into two non-empty sets, which are used to define the two children of that node. If the internal node has two data points associated with it, then the two

[3] You can think of them as 'left' and 'right', respectively, if you like but keep in mind that we are in a k dimensional space. In a 3D space, you might have left/right, up/down, back/front. We will use the same words for each dimension, namely low/high.

children nodes will be leaf nodes. If the internal node has three or more data points in it, then these data points need to be split into two sets which are assigned to the two children respectively, and the node constructor is called recursively on the two sets of data points.

How should the data points at an internal node be split? We would like the data points at each node to be a compact as possible in $\Re^k$. Put another way, we would like the distances between points that are within one child to be as close to each other as possible, and the distances between points in two different children to be as far apart from each other as possible. As we will see later, this will allow us to sometimes only search the data points associated with one of the two children of a node, which speeds up the search.

To try to achieve this compactness property when splitting the set of data points at a node, we compute which of the k dimensions has the largest range of values, where *range* of each dimension is defined by the maximum minus the minimum value of the data point values. We then partition or *split* the data points into two sets (corresponding to the two children) based only on the coordinate values of the data points in that splitting dimension. For this assignment, *we require that you choose the splitting dimension* $d \in \{0, \ldots, k-1\}$ *according to this maximum range criterion.* This guarantees, in particular, that the range of values in this splitting dimension is greater than zero. Also note that if there are two dimensions d and d' that both have the greatest range of values, then either may be chosen.

Once the *splitting* dimension d of a node is chosen, the algorithm chooses a *splitting value*: data points associated with that node whose value $x[d]$ within dimension d is less than or equal to the splitting value go in the low child's subtree, and the remaining data points go into the high child's subtree. How to choose the splitting value? We would like the tree to be as balanced as possible, and so we try to choose a splitting value that partitions the data points at a node such that there are a roughly equal number of data points associated with the two children. (This is similar to the role of the pivot in quicksort.) However, it may not be possible to choose a rule that is fast and that always achieves this goal. For example, choosing splitting value to be the median value works well if the data point values in dimension d are distinct, but if there are repeats then the median method might not work so well. (Also, choosing the median quickly is not obvious.) We will leave it up to you how to choose this splitting value. We suggest a simple choice for the splitting value to be the average of the min and max values in the splitting dimension. This choice at least guarantees that we partition the data points at a node into two sets (assuming at least two points are different).

1.1.3 Example (continued) - wording updated Nov. 8

For our figures, the *splitting plane* is defined by the median of data values in the dimension with the largest range. For the original 10 values, the splitting dimension is x_0. See Figure 2.

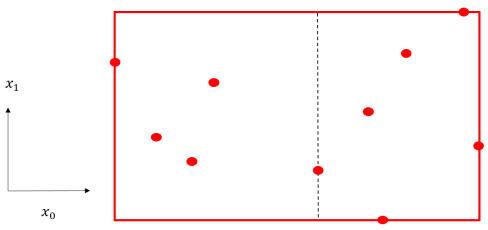

Figure 2: The original data with 10 points has a larger range in the x_0 dimension than in the x_1 dimension and so we split the 10 points in the root node using the x_0 dimension.

By convention, points on the splitting plane belong to the low half. So for Figure 2, there are 5 points in the low child and 5 points in the high child.

Although the median split is very common, for your implementation, we strongly suggest that you do *not* attempt to split using the median, but instead split using some other value such as the average of the low and high values in the splitting dimension. This will be easier to do.

Figure 3 below shows the result of splitting recursively, as described above. (Here I have labelled the points with variable p rather than x. The subscript will be an index on point $1, .., 10$ rather than dimension $0, ..., k-1$.) On the left, the original bounding rectangle is partitioned into a nested set of rectangles. This is done by choosing a set of splitting planes (dashed black lines) for each rectangle. The splitting planes are labelled with letters **a** to **h**. On the right is shown the KD-tree structure that is defined by the splitting planes and the corresponding nesting of rectangles. For example, the large bounding rectangle is partitioned into two rectangles by the splitting plane **a**, which corresponds to the root node of the tree.

Consider an internal node **d** and its subtree, and the corresponding splitting plane labelled **d** in the figure on the left. This subtree defined by **d** is the low (left) child of node **b** and thus the points associated with this subtree lie on the low side of splitting plane **b**, namely these are the points $\{p_4, p_3, p_5\}$. (Recall that points that lie on a splitting plane are included on the low side, not the high side.) The splitting plane **d** splits this set into two sets $\{p_4, p_3\}$ and $\{p_5\}$. The set $\{p_4, p_3\}$ is in turn split by plane **e**, and corresponding node **e**'s children are two leaves containing these two points. The point $\{p_5\}$ becomes a leaf which is the high child of node **d**. We encourage to examine other internal nodes and ask which regions they correspond to and what are the data points within these regions.

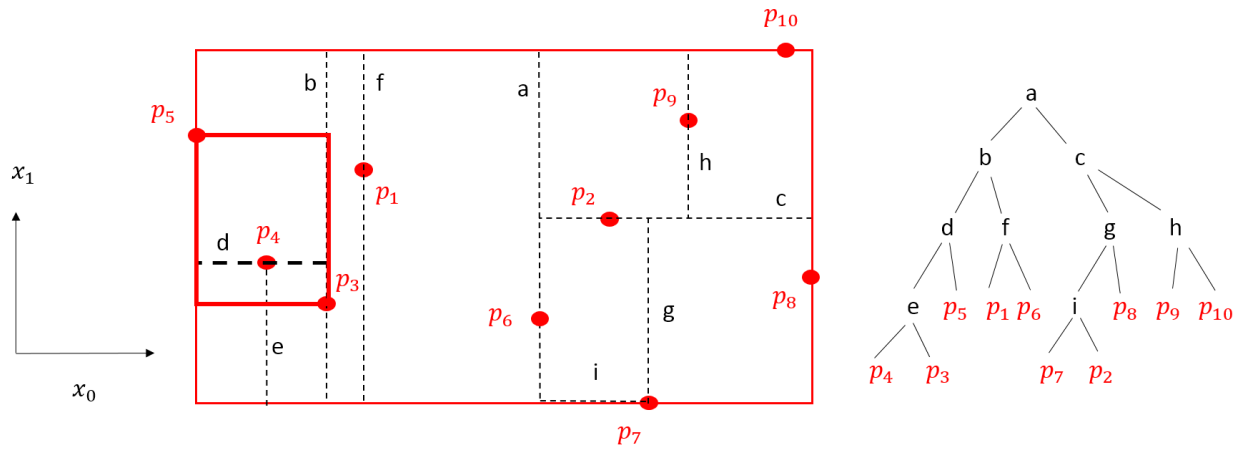

Figure 3: The outer red rectangle is the region defined by the low and high bounds of the 10 data points. The inner red rectangle is the region bounded by the data points below splitting node **d** , namely points p_3, p_4, p_5. Notice that this region does not span the entire region to the left of the **b** splitting plane.

1.2 Finding the nearest point

Given a kd-tree, one would like to to find the nearest data point to a given (input) query point. The query point need not belong to the set of data points in the tree.

The search algorithm for the nearest point is recursive, starting at the root node. For any node during the search, the query point's coordinate value $x_q[d]$ in the splitting dimension d is compared to the node's splitting value. You might think that you could restrict your search for the closest data point to the query point x_q by examining only the data points on the same side of the splitting plane as x_q. This approach would indeed be sufficient if we were trying to find an exact match of

the query in the tree. However, the query point may not have an exact match, and in that case the nearest point to the query point might not always lie on the same side of the splitting plane of a node, and so one might have to search through the points on both sides of the splitting plane.

If one were *always* to search through all points associated with a node, then the tree structure would serve no purpose, and one would be better off just brute force searching the list of data points. The clever insight that makes kd-trees useful is that we can sometimes restrict our search only to those data points on the same side of the splitting plane as the query point. The crucial condition for restricting the search is as follows: *if the distance from the query point to the closest point on the same side of the splitting plane is less than the distance from the query point to the splitting plane itself, then points on the opposite side of the splitting plane cannot be the nearest point and hence do not need to be considered.* If, however, the distance from the query point to the nearest point on the same side is greater than the distance from the query point to the splitting plane, then one does need to consider the possibility that a data point on the other side of the splitting plane might be the closest point.

Note that if there are two data points that are the same distance from the target, then we might ask how to break the tie to get a unique answer. It is possible to come with rules for doing so, but we will not concern ourselves with this. Instead, in the case that are multiple closest points to some given query point, if you return one of the points, then this is considered to be correct. This avoids us having to specify a tie breaking policy.

1.2.1 Example (continued)

The figure below shows the first splitting plane, which is defined at the root node of the tree. It also shows a query point x_q in blue. Suppose we first search the data points on the same side of the splitting plane as the query point. The nearest data point on the same side of the splitting plane is found to be p_1. A circle is drawn that is centered on the query point and passes through p_1. Notice that the distance from the query point to p_1 is less than the distance from the query point to the splitting plane. That is, the ball shown around the query point contains the nearest point but doesn't intersect the splitting plane. In this case, we can be sure that the nearest of all 10 data points is *not* on the other side of the splitting plane, and so we don't need to search the data points other side.

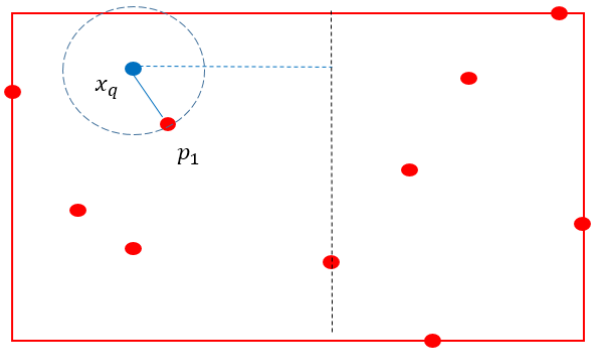

Figure 4: A query point x_q (blue) and the nearest data point p_1 on the same side as the splitting plane.

Let's now take an example with a different query point x'_q. Now the ball around the closest point p_1 to the query point that is on the same side of the splitting plane which has radius $\|x'_q - p_1\|$ intersects the splitting plane. In this case, we cannot be sure that p_1 is indeed the closest point in the data set, as there may be a point on the other side of the splitting plane that falls strictly within

the ball and hence is closer. We therefore also have to search the points on the other side of the splitting plane too. (Indeed in this example, p_2 is such a point.)

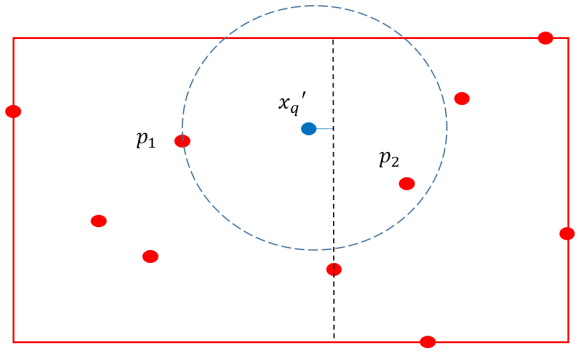

Figure 5: A query point x'_q (blue) and the nearest data point p_1 on the same side as the splitting plane, and the nearest point p_2 on the opposite side of the splitting plane.

The description above should be sufficient for you to write a method for building a kd-tree for a given set of data points, and for finding the closest data point to a given target.

1.3 More info about kd-trees

Kd-trees were invented in the 1970's. *If you are interested*, have look at some of the original research papers that presented the method. (Click to access the paper.) As you can see from just these two papers, there are many possible ways to define kd-trees.

- Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching". Communications of the ACM.
- Friedman, J. H.; Bentley, J. L.; Finkel, R. A. (1977). "An Algorithm for Finding Best Matches in Logarithmic Expected Time". ACM Transactions on Mathematical Software.

There are ample resources about kd-trees on the web. For example, the KD-tree `Wikipedia page` . While we encourage you to learn more if you are interested, be aware that these resources will contain more information than you need to do this assignment, and so you would need to sift through it and figure out what is important and what can be ignored.

This PDF should have all you need to do the assignment. If you require clarification, use the discussion board. Also, share links to good resources and to resolve questions you might have. (But please, before posting, use the search feature to check if you question has already been asked and answered.)

2 Instructions

2.1 Starter code

In our implementation, the tree will store the data points at the leaves. We assume that the coordinate values of the data and query points are all `int` values, so the data and query points are all in $\mathbb{Z}^k$ rather than $\Re^k$, where $\mathbb{Z}$ is a common symbol for the set of integers.

The starter code contains three classes:

- `KDTree`: This class has several fields including `rootNode` which is the root of the tree and is of type `KDNode` which is an inner class. It also has an integer field `k` which is the number of dimensions, i.e. $\mathbb{Z}^k$.
- `Datum`: This class holds the coordinate values $x[\,]$ of a data point or query point.
- `Tester`: Some example tests to verify the correctness of your code.

2.2 Your Task

Implement the following in the `KDTree` class:

- **(50 points)** `KDNode` constructor

  The constructor `KDNode` is given an array of `Datum` objects, and it constructs a `KDNode` object that is the root of a subtree whose leaves contain the given `Datum` objects.

  The `KDNode` constructor should split the given data points such that, on average, about half go into the low child and the other about half go into high child, and so the resulting tree is approximately balanced. The height of the tree should be roughly $\log_2(N)$ where N is the number of data points.

  It is difficult to choose a splitting rule that guarantees an equal (half-half) split, and so we will not require this. In particular, even if you were to choose the splitting value to be the median value of the data points in the splitting dimension, a very unbalanced tree could still result if many data points had that splitting value at many of the nodes. It is good enough for you to let the split be the average of the min and max value in the splitting dimension. If properly implemented, this will lead to roughly $\log_2(N)$ height of the test trees. We have provided you with a helper method `KDTree.height` that returns the height of the tree, and `KDTree.meanDepth` that computes mean depth of the leaves of a tree.

  If there are duplicate data points in the input, then your constructor must put only one copy of the duplicate points into the tree. It is up to you to find an efficient way to remove duplicates. What you should *not* do is try to remove duplicates from the initial list of data points, since this would take time $O(N^2K)$ – namely comparing all pairs of data points for equality on each dimension – which is too slow for large data sets. We will test that you have correctly and efficiently removed duplicates.

  (Updated Nov. 12) You should insure that your tree is constructed efficiently. Guidelines on tree construction times can be found in the tester.

  (Updated Nov. 12) Although you are free to create your own variables / methods, you should also set the value of the provided fields appropriately. For example, splitDim / splitValue should be set by your code if a given KDNode is not a leaf.

- **(20 points)** `KDTreeIterator` inner class [EDIT: wording below updated Nov. 10]

  The method `KDTree.iterator()` returns an `kDTreeIterator` object that can be used to iterate through all the data points. *The order of points in your iterator must correspond to an in-order traversal of your kd-tree, namely it must visit all data points in the low subtree of any node before it visits any of the data points in the high subtree.*

  We will test your iterator by examining the order of data points. (The Grader will have access to the `KDNode` objects and various fields of your tree: we have given these fields package visibility.)

The tester that we provide only tests your iterator for the 1D case. We suggest you create your own iterator tests for the 2D and 3D cases, and share these testers with each other.

It is fine if your iterator uses an `Arraylist` to store references to all the data points (`Datum` objects). There are more space efficient ways to make iterators from trees, which represent only one path in the tree at any time. But we do not require this space efficiency.

- **(30 points)** `KDNode.nearestPointInNode` method

  The method `nearestPointInNode` takes a query point `Datum` and returns a data point `Datum` that is as close to the query point as any other data point. There may be multiple points that have the same nearest distance to the query point, and in this case your method should return one of them.

  (Updated Nov. 12) Given a tree of sufficient size (# Datums $\geq$ 50000), your implementation must (on average) beat a brute force approach. We provide a few such tests in the tester.

2.3 Submission

- Submit a single zipped file A3.zip that contains the `KDTree.java` file to the myCourses assignment A3 folder. If you submit the other files in the directory, then they will be ignored.

  Include your name and student ID number within the comment at the top of each file.

- Xiru (TA) will check for invalid submissions once before the solution deadline and once the day after the solution deadline. He will notify students by email if their submission is invalid. It is your responsibility to check your email.

- Invalid solutions will receive a grade of 0. You may resubmit a valid solution, but only up to the two days late limit, and you will receive a late penalty. As with Assignments 1 and 2, the grade you will receive is the maximum grade from your submissions.

- If you have any issues that you wish the TAs (graders) to be aware of, please include them in the comment field in mycourses along with your submission. Otherwise leave the mycourses comment field blank.