k-nearest neighbors and kd-trees

CS 360 Machine Learning
Week 2, Day 1

January 30, 2024

Contents

1 kd-trees and nearest neighbors

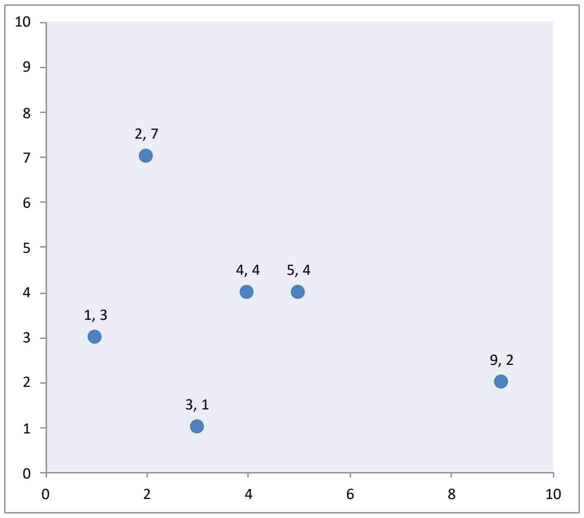

Figure 1: kd-tree example initial training points

Continuing with our example from last time (Figure 1) recall that we always start with a single dimension (e.g., the x-axis) at a time. We choose the median along each dimension, starting with the x-axis. So we started with the root node as $(4, 4)$ and then we recurse where the left child is the region on the left and the right similarly. With each recursive call, we switch the dimension, so next we switch to y. Based on the sorted order, the middle point is $(1, 3)$. We find that $(3, 1)$ is the left child and $(2, 7)$ is the right child since $(3, 1)$ is smaller in terms of the y-coordinate. So these are ordered in terms of a tree traversal in their y-coordinate. The split point is stored at the root node of the subtree.

When you write the code for median finding, you can sort, find the middle element, and then make sure not to include the middle point when you recurse on the left and right sides. (While we'll go ahead and sort to find the median, that takes $O(n \log n)$ time, and if we were really concerned with the time complexity of this *kd*-tree creation algorithm we'd use an $O(n)$ median-finding algorithm instead.) Note that if we were continuing further down the tree, we'd switch back to consider the x-coordinate after considering the y-coordinate, and just keep switching back and forth (we use the `mod` operator to do this).

Considering the other side of the created *kd*-tree (the right of the root node $(4, 4)$), then we end up with $(5, 4)$ as the next node and $(9, 1)$ as its left child.

What if we had a query point at $(7, 7)$? We would start by considering the distance to $(4, 4)$ since it's at the root – that's our current known min distance. When we consider the hypersphere (circle in 2-dimensions) centered at the query point and with a radius of minimum distance, that's all the points that could be closer to $(7, 7)$ than $(4, 4)$. Since that hypersphere intersects the split, we need to be sure to recurse on both sides since there could be a point on the other side of the recursive call (based on a distance calculation to its x-value defining the vertical boundary) that's closer to the query point. Recursing down, we again update the minimum distance based on the next point considered. In this case, we next consider $(5, 4)$ since the query point is on that side of the tree, and we update the minimum distance based on the distance to $(5, 4)$. We then consider the other side of the tree, checking the distance from the query point to the boundary in one dimension, eventually recursing down to check $(1, 3)$ and $(2, 7)$ but not needing to recurse down to the subtree rooted at $(3, 1)$ because the distance from $(7, 7)$ to $(5, 4)$ (the current minimum distance) is less than the distance to the boundary with the region containing $(3, 1)$. Note that in order to determine whether to check a subtree, we consider only the distance in a single dimension, e.g., checking only whether the distance between the query point's x-value and the boundary to the *kd*-tree cell is more or less than the current calculated minimum distance.

Now let's consider some *incomplete* pseudocode for this query process:

```
"""
root is the root of the subtree currently being considered
q is the query point
min_d is the current minimum distance
nn is the current nearest neighbor
d is a function that calculates the distance between two points

Note that this will also need to include notions of depth and dimension
that are not included in the below, along with other missing details!
When considering dimension we'll need to include a calculation that:
    dim = depth % d
where d is the total number of dimensions,
depth is always increasing as the tree grows, and
dim cycles through the dimensions.
"""
get_nn(root, q, min_d, nn):
    if root is a leaf:
        if d(root, q) < min_d:
            return d(root, q) , root
        else:
            return min_d , nn
    else: # non-leaf case
        if d(root, q) < min_d:
            min_d, nn = d(root, q) , root
        child = root of the subtree containing q which will always need to be checked
        min_d, nn = get_nn(child, q, min_d, nn)
        # hypersphere comparison to the boundary
        # q.dim is the current dimension coordinate
        # root.dim is the relevant split coordinate of the root
        if |q.dim - root.dim | < min_d:
             # the query min_d hypersphere overlaps the boundary for that subtree
            return get_nn(other_child, q, min_d, nn)
        else:
            return min_d, nn
```

See the handout to try running some example queries on the below example training set and resulting *kd*-tree (Figure 2).

1.1 Handling $k > 1$

How do we extend this process to query for multiple (k) nearest neighbors? Consider the naive case (not using *kd*-trees) where we just look at all the points. We could just sort by distance and take the top k. But every time we queried we'd need to sort, so we'd no longer have the $O(n)$ runtime we previously could have achieved for the naive algorithm. Instead, we could put all the points into a heap as we considered the distances and take the k with minimum distance after processing all points.

Suppose we had the following calculated distances to the query point:

$$7, 2, 3, 4, 8, 5, 1$$

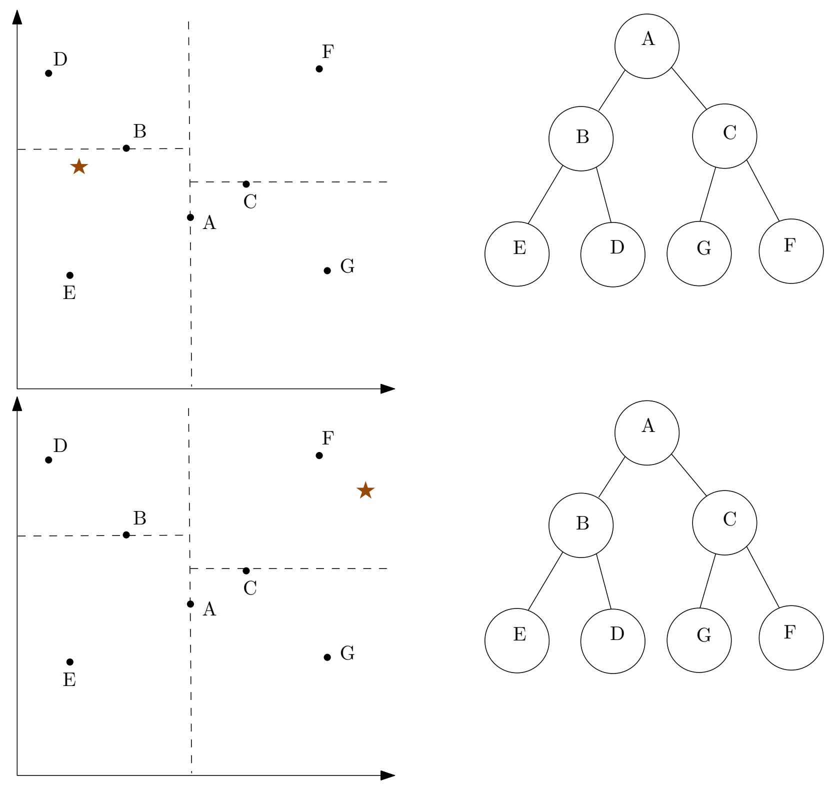

Figure 2: A running example set of training points for the *kd*-tree, with the full tree and space partitioning shown. (See the handout for associated questions.)

In this case, entering these distances into a min-heap in the given order, we'd start with 7 as the root, then when we add 2 we'd need to bubble up to maintain the heap property, with 2 as the new root of the heap. We'd continue in this way, maintaining the heap property as we go. Finally at the end, we'd enter 1 and find that we bubble up the heap so that it's the final root node.

Once built, in order to get the k nearest neighbors we would use the get minimum operation k times for the heap, making sure to maintain the heap property each time. In this example, that means we'd extract 1, followed by 2, and so on. When implementing it, each of these distances would actually be part of an object or tuple including both the distance and the associated point, which would allow us to retrieve these k nearest neighboring points.

If we were going to do this in the *kd*-tree implementation, we'd need to think about maintaining the bounding ball for the kth nearest neighbor instead of the nearest neighbor distance as discussed above, but would still be able to use this basic strategy of maintaining the points by distance to the query point in a heap.

2 Evaluation methods for binary classification (continued)

2.1 AUC (area under the curve)

Consider a ROC curve – ideally, the false positive rate is low and the true positive rate is high. Looking at the area under the curve can help you to compare various ROC curves, for example if the curves themselves overlap or have benefits in different parts of the false positive rates. In some cases you might still care about the specific values of the ROC curve for some regions – the AUC will not help you understand this more specific shape information, but can be a useful way overall to compare different methods.

How can you calculate the AUC? One way to do it is based on discrete thresholds, like the manual process of integration you might have learned before learning about Calculus integration methods. Of course there are also packages to do this for you.

2.2 Precision / recall curves

Another way to compare models and examine their benefits and drawbacks is to consider precision / recall curves (see, e.g., textbook Figure 3-8). In a precision / recall curve, the x-axis has the calculated recall and the y-axis shows precision. Why might we be interested in also looking at precision and recall curves? In cases where you have low precision, this gives you the chance to identify that, though there is still information a precision / recall curve won't show you.

Consider the below confusion matrix:

Table 1: An example confusion matrix, where the rows give the true values and the columns give the predictions.

	0	1
0	385	10
1	3	7

Looking at the false positive rate (FPR) versus the true positive rate (i.e., the ROC curve) looks good for this matrix, since here $FPR = 10/(10 + 385)$ which is low and $TPR = 7/(7 + 3)$, which is fairly high.

However, if I'm interested in the true positive rate, I may also want to look at this number relative to all the things I said were positive; in this case I might consider precision, which for this example is $7/(10 + 7)$. Recall is the same as true positive rate, so you want precision to be high and recall to be high; this is what the precision / recall curve will allow you to see. However, the precision / recall curve doesn't make use of the number of true negatives, which in some cases you might care about.

Precision can be less than 50%. For a ROC curve we should never see something below the center line since that's worse than random guessing. Some people prefer precision / recall curves because you use the whole space; you might genuinely end up with values below 50%. Overall, precision / recall curves provide another way to look at the data.