K-nearest Neighbors Data Structures

Kun He (何琨)

Data Mining and Machine Learning Lab
(John Hopcroft Lab)
Huazhong University of Science & Technology

brooklet60@hust.edu.cn

2022 年 6 月

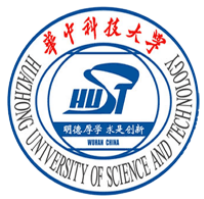


Table of Contents

1. k-NN Review
   - Time Complexity of k-NN
   - Goal
2. K-Dimensional Trees
   - General Idea
   - Proof of Effectiveness
   - KD-tree Data Structure
3. Ball-Trees
   - Ball-Tree Construction
   - Ball-Tree Use
4. Summary

Table of Contents

1. k-NN Review
   - Time Complexity of k-NN
   - Goal
2. K-Dimensional Trees
   - General Idea
   - Proof of Effectiveness
   - KD-tree Data Structure
3. Ball-Trees
   - Ball-Tree Construction
   - Ball-Tree Use
4. Summary

Assumptions

Assumptions

- Let's say we are in a d-dimensional space.
- To make it easier, let's assume we've already processed some number of inputs, and we want to know the time complexity of adding one more data point.

Time Complexity of k-NN

- **When training**, k-NN simply memorizes the labels of each data point it sees. This means adding one more data point is $O(d)$.
- **When testing**, we need to compute the distance between our new data point and all of the data points we trained on. If n is the number of data points we have trained on, then our time complexity for training is $O(dn)$.
- **Classifying one test input** is also $O(dn)$.

Goal

Goal

To achieve the best accuracy we can, we would like our training data set to be very large ($n \gg 0$), but this will soon become a serious bottleneck during test time.
Can we make k-NN faster during testing?
We can if we use clever data structures.

Table of Contents

1. k-NN Review
   - Time Complexity of k-NN
   - Goal
2. K-Dimensional Trees
   - General Idea
   - Proof of Effectiveness
   - KD-tree Data Structure
3. Ball-Trees
   - Ball-Tree Construction
   - Ball-Tree Use
4. Summary

General Idea

The general idea of KD-trees is to partition the feature space. We want to discard lots of data points immediately because their partition is further away than our k closest neighbors.
We partition the following way:

1. Divide your data into two halves, e.g. left and right, along one feature.
2. For each training input, remember the half it lies in.

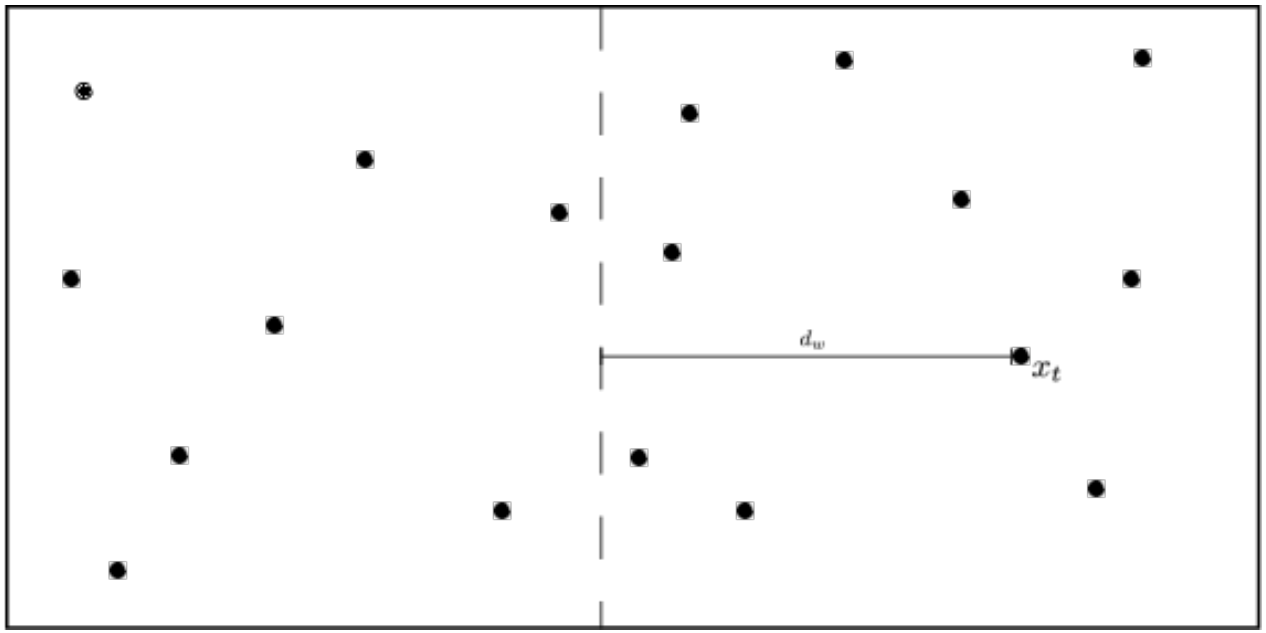
Partitioning the Feature Space

How can this partitioning speed up testing?
Let's think about it for the one neighbor case.

1. Identify which side the test point lies in, e.g. the right side.
2. Find the nearest neighbor x^R_{NN} of x_t in the same side. The R denotes that our nearest neighbor is also on the right side.
3. Compute the distance between x_y and the dividing "wall". Denote this as d_w. If $d_w > d(x_t, x^R_{\mathrm{NN}})$ you are done, and we get a 2x speedup.

In other words: if the distance to the partition is larger than the distance to our closest neighbor, we know that none of the data points *inside* that partition can be closer. We can avoid computing the distance to any of the points in that entire partition.

Partitioning the Feature Space



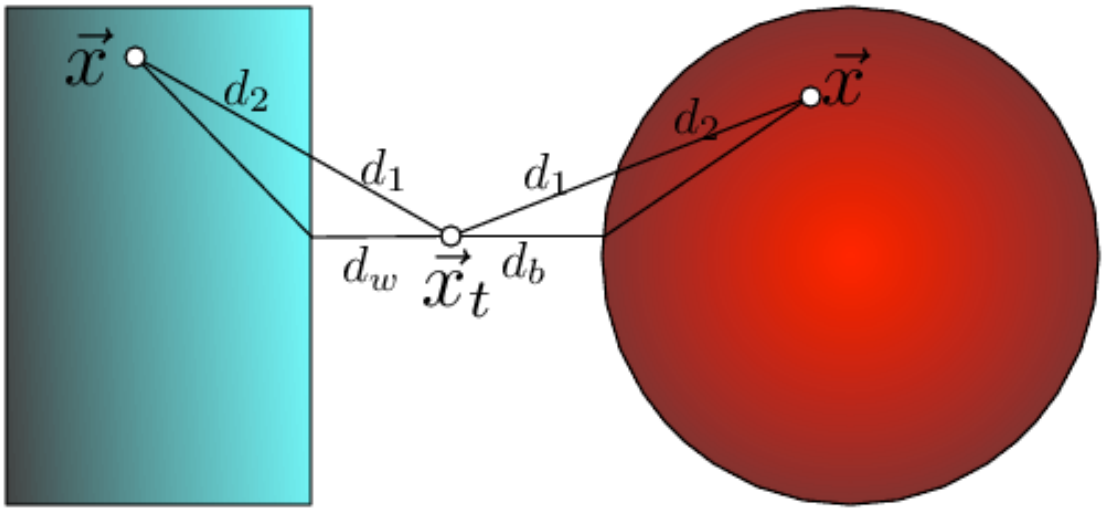
Proof

We can prove this formally with the triangular inequality.
Proof.
Let $d(x_t, x)$ denote the distance between our test point x_t and a candidate x. We know that x lies on the other side of the wall, so this distance is dissected into two parts $d(x_t, x) = d_1 + d_2$, where d_1 is the part of the distance on $x_t's$ side of the wall and d_2 is the part of the distance on $x's$ side of the wall. Also let d_w denote the shortest distance from x_t to the wall. We know that $d_1 > d_w$ and therefore it follows that

$$d(x_t, x) = d_1 + d_2 \geq d_w + d_2 \geq d_w.$$

This implies that if d_w is already larger than the distance to the current best candidate point for the nearest neighbor, we can safely discard x as a candidate.

Illustration of the Triangular Inequality



The bounding of the distance between $\vec{x}_t$ and $\vec{x}$ with KD-trees and Ball trees (here $\vec{x}$ is drawn twice, once for each setting). The distance can be dissected into two components $d(\vec{x}_t, \vec{x}) = d_1 + d_2$, where d_1 is the outside ball/box component and d_2 the component inside the ball/box. In both cases d_1 can be lower bounded by the distance to the wall, d_w, or ball, d_b, respectively i.e. $d(\vec{x}_t, \vec{x}) = d_1 + d_2 \geq d_w + d_2 \geq d_w$.

Quiz

Construct a case where this does not work.

KD-tree Data Structure

KD-tree

- The KD-tree is a binary tree in which every node is a k-dimensional point.
- Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces.
- Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree.
- The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k dimensions, with the hyperplane perpendicular to that dimension's axis.

Tree Construction

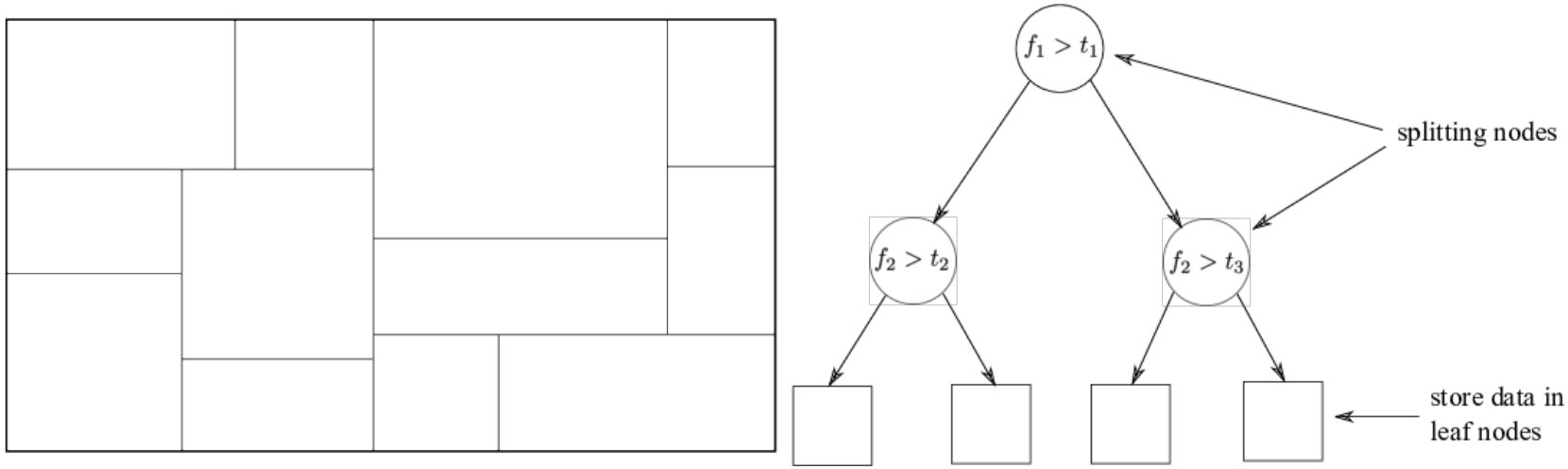


Fig. The partitioned feature space with corresponding KD-tree.

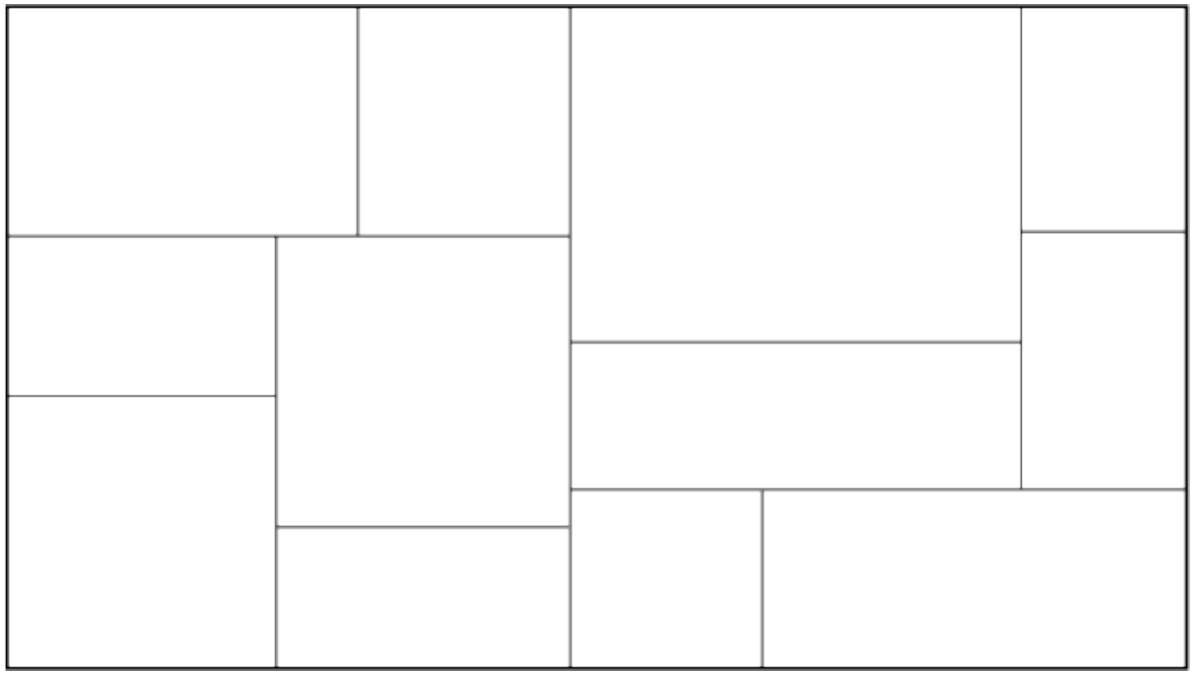
Tree Construction

Tree Construction:

1. Split data recursively in half on exactly one feature.
2. Rotate through features.

When rotating through features, a good heuristic is to pick the feature with maximum variance.

Example



Question:

- Which partitions can be pruned?
- Which must be searched and in what order?

Pros and Cons

Pros:

- Exact.
- Easy to build.

Cons:

- Curse of Dimensionality makes KD-Trees ineffective for higher number of dimensions.
- All splits are axis aligned.

Approximation: Limit search to m leafs only.

Table of Contents

1. k-NN Review
   - Time Complexity of k-NN
   - Goal
2. K-Dimensional Trees
   - General Idea
   - Proof of Effectiveness
   - KD-tree Data Structure
3. Ball-Trees
   - Ball-Tree Construction
   - Ball-Tree Use
4. Summary

Concept of Ball-Trees

Concepts

- Similar to KD-trees, but instead of boxes use hyper-spheres (balls).
- As before we can dissect the distance and use the triangular inequality

$$d(x_t, x) = d_1 + d_2 \geq d_b + d_2 \geq d_b$$

- If the distance to the ball, d_b, is larger than distance to the currently closest neighbor, we can safely ignore the ball and all points within.
- The ball structure allows us to partition the data along an underlying manifold that our points are on, instead of repeatedly dissecting the entire feature space (as in KD-Trees).

Ball-Tree Construction

Input: set S, $n = |S|$, k

Algorithm 1 Balltree in Pseudo-code

1: **procedure** BALLTREE(S,k)
2: **if** $|S| < k$ **then** stop **end if** ▷ Return leaf containing S
3: pick $x_0 \in S$ uniformly at random
4: pick $x_1 = \text{argmax}_{x \in S}\ d(x_0,x)$
5: pick $x_2 = \text{argmax}_{x \in S}\ d(x_1,x)$
6: $\forall i = 1 \ldots |S|, z_i = (x_1 - x_2)^T x_i$ ← project data onto $(x_1 - x_2)$
7: $m = \text{median}(z_1, \cdots, z_{|S|})$
8: $S_L = \{x \in S : z_i < m\}$
9: $S_R = \{x \in S : z_i \geq m\}$
10: **Return** tree:
- center $c = \text{mean}(S)$
- radius $r = \max_{x \in S} d(x,c)$
- children: Balltree(S_L,k) and Balltree(S_R,k)

11: **end procedure**

Note: Steps 3 & 4 pick the direction with a large spread $(x1-x2)$

Ball-Tree Use

Same as KD-Trees:
Slower than KD-Trees in low dimensions $(d\ 3)$ but a lot faster in high dimensions. Both are affected by the curse of dimensionality, but Ball-trees tend to still work if data exhibits local structure (e.g. lies on a low-dimensional manifold).

Table of Contents

1. k-NN Review
   - Time Complexity of k-NN
   - Goal
2. K-Dimensional Trees
   - General Idea
   - Proof of Effectiveness
   - KD-tree Data Structure
3. Ball-Trees
   - Ball-Tree Construction
   - Ball-Tree Use
4. Summary

Summary

- k -NN is slow during testing because it does a lot of unecessary work.
- KD-trees partition the feature space so we can rule out whole partitions that are further away than our closest k neighbors. However, the splits are axis aligned which does not extend well to higher dimensions.
- Ball-trees partition the manifold the points are on, as opposed to the whole space. This allows it to perform much better in higher dimensions.

The End