

DSC 190

DATA STRUCTURES & ALGORITHMS

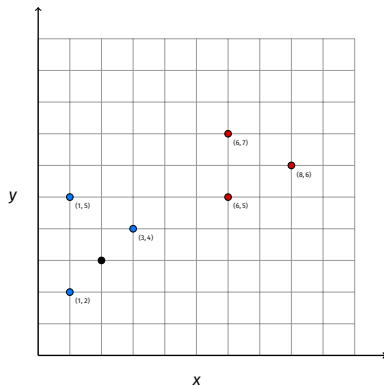
Lecture 6 | Part 1

Today's Lecture

Nearest Neighbors

- ▶ Finding the closest data point to a query point is a common operation.
- ▶ In machine learning, at the core of the **nearest neighbor classifier**.

NN Classifier



NN Query

- ▶ **Given:** a data set X of n points in \mathbb{R}^d and a **query** point, $p \in \mathbb{R}^d$.
- ▶ **Return:** the point in X that is nearest¹ to p

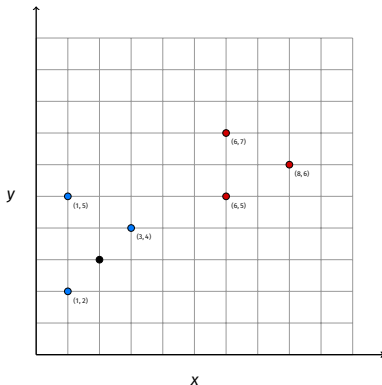
¹In terms of Euclidean distance, though other distances can be considered.

Approach #1: Brute Force

- ▶ Compute distance between p and every point $x \in X$, keep closest.
- ▶ Time: $\Theta(nd)$

Intuitively...

- ...we can do better. We only need to look at region close to p .

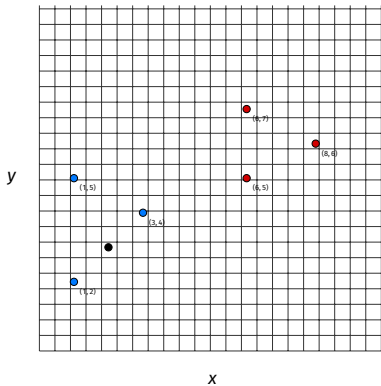


```
def brute_force_nn_search(data, p):  
    """Find nearest neighbor.  
  
    Parameters  
    ____  
    data : np.ndarray  
        An n x d array of points.  
    p : np.ndarray  
        A d-array representing the query point.  
  
    Returns  
    ____  
    nn : np.ndarray  
        The closest point.  
    nn_distance : float  
        Distance to closest point.  
  
    """  
    distances = np.sqrt(np.sum((data - p)**2, axis=1))  
    ix_of_nn = np.argmin(distances)  
    nn = data[ix_of_nn]  
    nn_distance = distances[ix_of_nn]  
    return (nn, nn_distance)
```

Approach #2

- ▶ Build a grid.
- ▶ To query NN, find cell containing p .
- ▶ Start search in p 's cell, move outwards.

Intuitively...



Problems

- ▶ How do we choose grid cell size?
 - ▶ Too big: cells contain a lot of points = brute force.
 - ▶ Too small: Most of the cells are empty.
 - ▶ “Just right” for one region might be too big/small for another region.
- ▶ Number of cells grows **exponentially** with dimension.

Today

- ▶ We'll refine the idea of a grid.
- ▶ Adapt cell placement/size to the data.
- ▶ Result: **k-d trees**.

k-d Trees

- ▶ Will speed up NN queries in low dimensions (<10) from $\Theta(n)$ to $\Theta(\log n)$.
- ▶ But will be just as bad as brute force in high dimensions.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 6 | Part 2

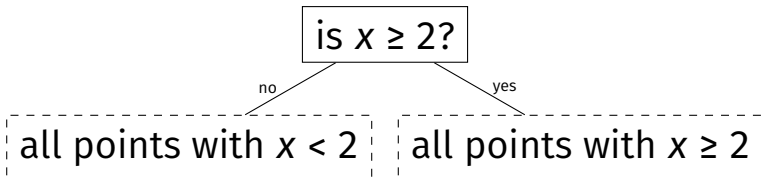
k-d Trees

k-d Trees

- ▶ **Binary search tree** for multidimensional data.
- ▶ Now: structure & properties.
- ▶ Next section: how to query them.
- ▶ Next next section: how to construct them.

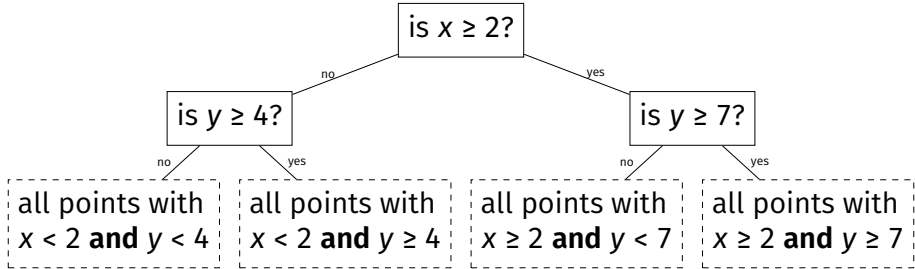
Internal Nodes

- ▶ Internal nodes are **threshold questions**.
 - ▶ can be of form $x \geq 1?$ or $y \geq \tau?$ in 2-d.
 - ▶ can be of form $x \geq \tau?$ or $y \geq \tau?$ or $z \geq \tau?$ in 3-d.
 - ▶ etc.



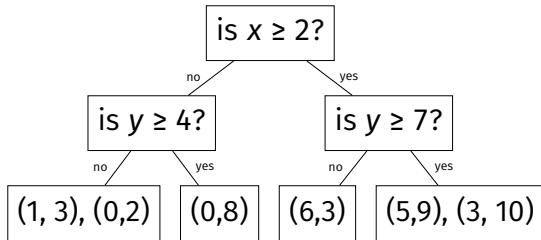
Internal Nodes

- A path forms a **conjunction**.



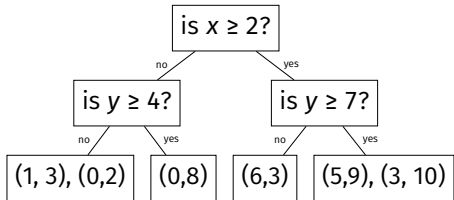
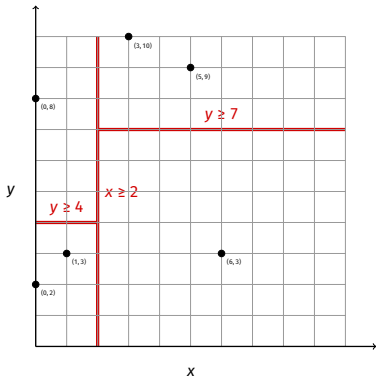
Leaf Nodes

- Leaf nodes are (collections of) points.



Partitioning

- Each internal node **splits** space.



k-d Trees in Python

```
from dataclasses import dataclass
from typing import Union, Optional
import numpy as np

@dataclass
class KDInternalNode:
    # the left and right children can be internal nodes
    # or numpy arrays of points (leaf nodes)
    left: Union['KDInternalNode', np.ndarray]
    right: Union['KDInternalNode', np.ndarray]

    # the threshold tau in the question
    threshold: float

    # the dimension used in the question
    dimension: int
```

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 6 | Part 3

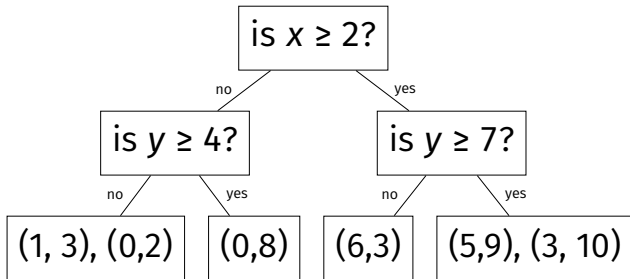
Queries on k-d Trees

Types of Queries

- ▶ Standard query:
 - ▶ Is $(1, 5)$ in the tree?
- ▶ Nearest neighbor query:
 - ▶ Return the nearest neighbor(s) of $(1, 5)$.

Standard Queries

- Is (6,3) in the tree? Is (1, 5) in the tree?



Standard Queries

- ▶ Similar to BST query.
 - ▶ Recursively choose left/right by answering question.
 - ▶ Brute-force linear search on leaf (if needed).
- ▶ Takes $O(h)$ time, where h is height of the tree².

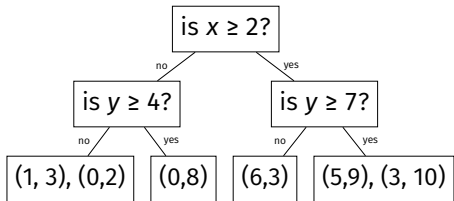
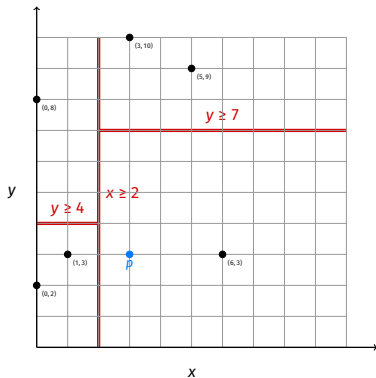
²Assuming each leaf has a bounded number of points.

Nearest Neighbor Queries

- ▶ Given query point $p = (x, y)$, find nearest neighbor in tree.
- ▶ Can we just do a standard query?
 - ▶ Find cell that *would* contain (x, y) .
 - ▶ Return closest neighbor within that cell.

No

► Example: $p = (3, 3)$.



Main Idea

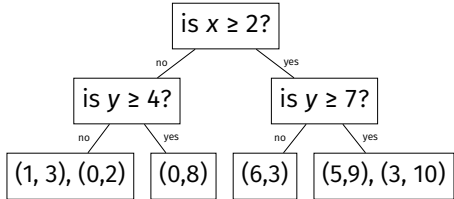
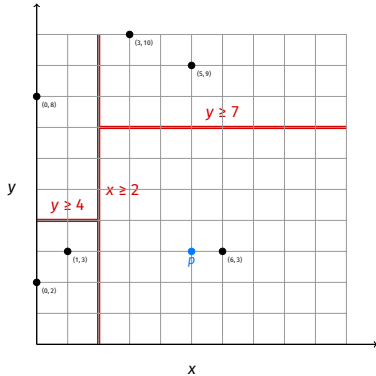
It is not sufficient to only check the cell that p *would* be placed in. You must also check neighboring cells (which can be very far away in the tree).

Brute Force?

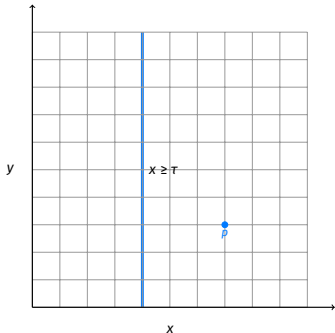
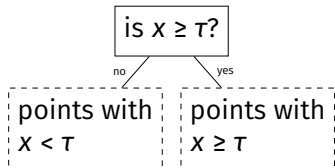
- ▶ This suggests we need to traverse the whole tree.
- ▶ But we can actually do much better.
- ▶ Intuitively, some branches can be ruled out (**pruned**).

Example

► Example: $p = (5, 3)$.

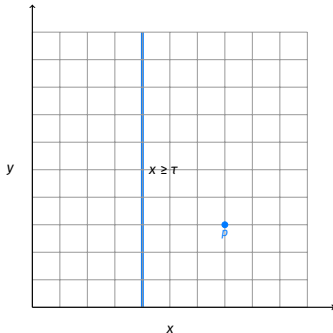
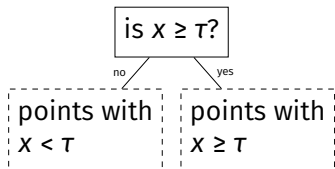


Bounding Branches



- **Observation:** let d_{bound} be distance from p to the boundary.
- Then the closest a point in the other branch can be to p is d_{bound}
- If we search and find a point whose distance to p is less than d_{bound} , **we do not need to search other branch.**

Bounding Branches



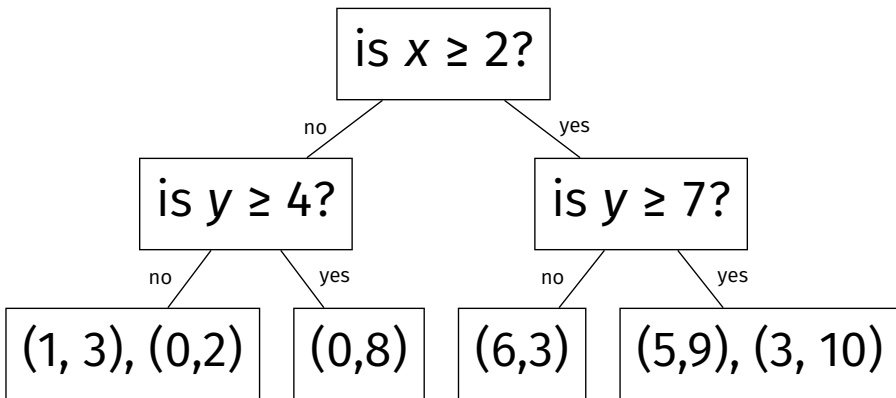
To query NN of (x, y) :

- ▶ Search right branch first if $x \geq t$, otherwise search left branch first.
- ▶ Let d_{nn} be the distance from p to the closest point found.
- ▶ Let d_{bound} be the distance from p to boundary.
- ▶ Search other branch only if $d_{bound} < d_{nn}$.

Apply this idea recursively.

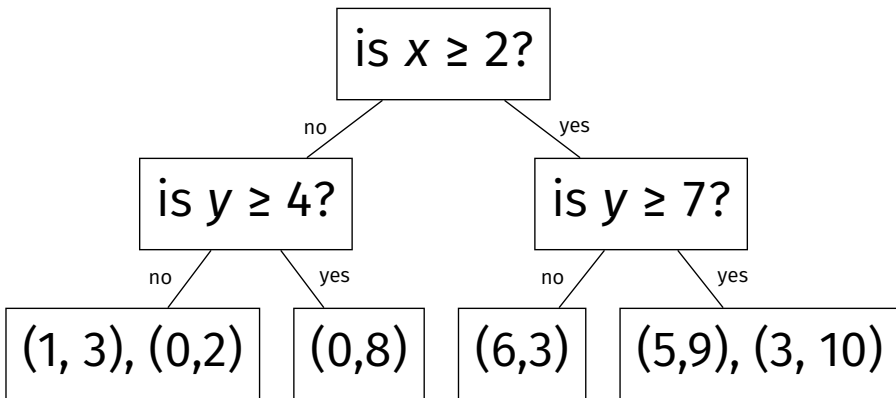
Example

- NN Query: (5, 3)



Example

- NN Query: (3,3)




```
def nn_query(node, p):
    if isinstance(node, np.ndarray):
        return brute_force_nn_search(node, p)
    else:
        # find the most likely branch
        if p[node.dimension] >= node.threshold:
            most_likely_branch, other_branch = node.right, node.left
        else:
            most_likely_branch, other_branch = node.left, node.right

        # compute distance to boundary
        distance_to_boundary = abs(p[node.dimension] - node.threshold)

        # find nn within most likely branch
        nn, nn_distance = nn_query(most_likely_branch, p)

        # check the other branch, but only if necessary
        if distance_to_boundary < nn_distance:
            nn_other, nn_other_distance = nn_query(other_branch, p)

            # check if the nn within this branch is closer
            if nn_other_distance < nn_distance:
                nn = nn_other
                nn_distance = nn_other_distance

    return nn, nn_distance
```

k-NN Search

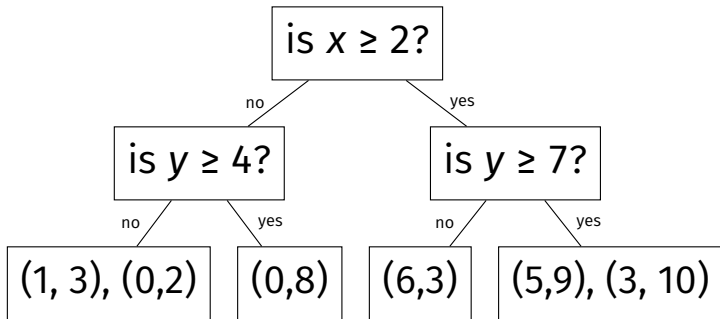
- ▶ Sometimes we want to find k nearest neighbors.
- ▶ Keep a max heap of best k so far.
- ▶ Check branch if distance to boundary $< k$ th closest.

Analysis

- ▶ Assume each leaf has bounded number of points.
- ▶ Best case: $\Theta(h) \rightarrow \Theta(\log n)$ if balanced
- ▶ Worst case: $\Theta(n)$.
 - ▶ We may be unable to rule out many of the branches.
 - ▶ Can occur even if tree is balanced.
 - ▶ Especially if query point far from data.
- ▶ Note: balancing is difficult, but possible.

Example of Worst Case

- ▶ NN Query: (20, 20)
- ▶ Closest point is (5, 9) at distance ≈ 19



Performance Degradation

- ▶ In small dimensions, NN lookup usually takes $\Theta(\log n)$.
- ▶ We'll see performance **degrades** to $\Theta(n)$ (brute force) as dimensionality $\rightarrow \infty$.
- ▶ **Curse of Dimensionality**

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 6 | Part 4

Constructing k-d Trees

Construction

- ▶ **Given:** a set of n data points in \mathbb{R}^d
- ▶ **Construct:** a k-d tree containing these points.

Caveats

- ▶ There are many variations on k-d tree construction.
- ▶ We'll describe one popular approach.
- ▶ **Assumption:** **offline** construction.
 - ▶ Have all of the data at once (no insert/delete).

Idea

- ▶ Starting with n points, either:
 - ▶ make internal node by splitting ($x \geq \tau$?)
 - ▶ make leaf node containing the points
- ▶ Apply this strategy recursively.
- ▶ Questions:
 - ▶ Do we split, or do we make a leaf?
 - ▶ If we split:
 - ▶ What dimension to split on?
 - ▶ What threshold to use?

Q1: Do we split?

- ▶ Take parameter M (max leaf size).
- ▶ If $n < M$, don't split.
- ▶ **Reason:** For small n , brute force is actually faster (less overhead).

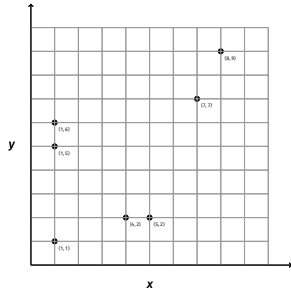
Q2: Which dimension to split on?

- ▶ Choose dimension with largest **spread**.
 - ▶ Difference between largest and smallest values.
 - ▶ Calculated using only points in **this** subtree.
- ▶ **Alternatively:** round-robin. Split x, y, z, x, y, \dots

Q3: What threshold to use?

- ▶ Need threshold, τ .
- ▶ Use median value in splitting dimension.
 - ▶ Calculated using only points in **this** subtree.
 - ▶ Guaranteed to produce balanced tree.
- ▶ **Alternatively:** randomly-selected pivot, or median of random selection

Set $M = 2$, use median and spread for splitting. We start with data:



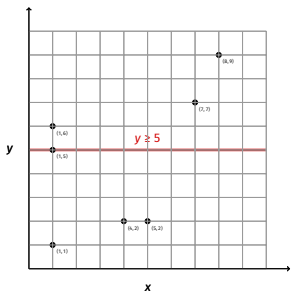
$(1,1), (4,2), (5,2), (1,5), (1,6), (7,7), (8,9)$

x	y
4	2
1	1
5	2
1	6
7	7
8	9
2	5

- Spread of x: 7
- Spread of y: 8
- Use y as splitting dimension.
- Median of y: 5.

Set $M = 2$, use median and spread for splitting. We start with data:

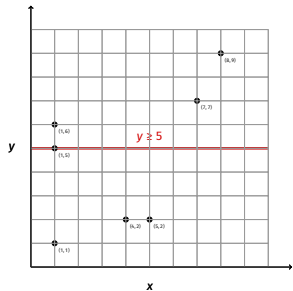
x	y
4	2
1	1
5	2
1	6
7	7
8	9
2	5



is $y \geq 5$?

$(1,1), (4,2), (5,2)$ $(1,5), (1,6), (7,7), (8,9)$

- Spread of x: 7
- Spread of y: 8
- Use y as splitting dimension.
- Median of y: 5.



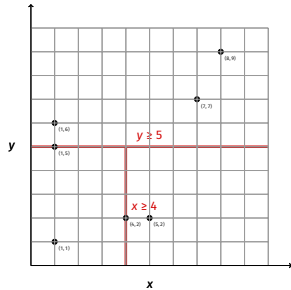
is $y \geq 5$?

(1,1), (4,2), (5,2) | (1,5), (1,6), (7,7), (8,9)

Recurse on left child. Data becomes:

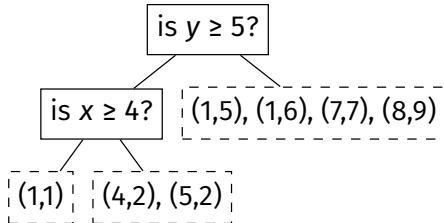
x	y
4	2
1	1
5	2

- Spread of x: 4
- Spread of y: 1
- Use x as splitting dimension.
- Median of x: 4.

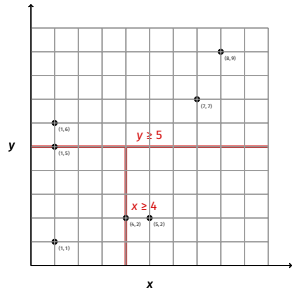


Recurse on left child. Data becomes:

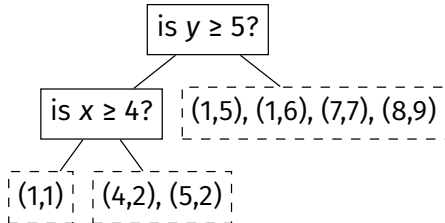
x	y
4	2
1	1
5	2

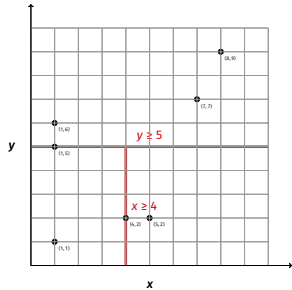


- Spread of x: 4
- Spread of y: 1
- Use x as splitting dimension.
- Median of x: 4.

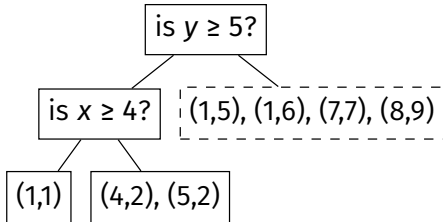


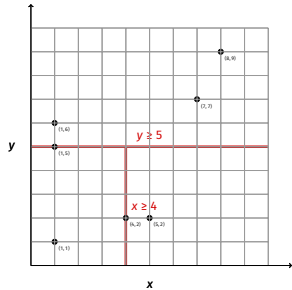
Recurse on children. Since size $\leq M$, these become leaf nodes.





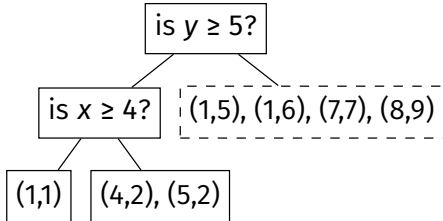
Recurse on children. Since size $\leq M$, these become leaf nodes.



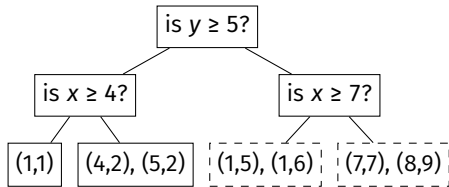
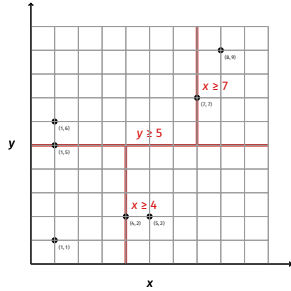


Unroll recursion, now recurse down right side of tree. Data becomes:

x	y
1	6
7	7
8	9
2	5



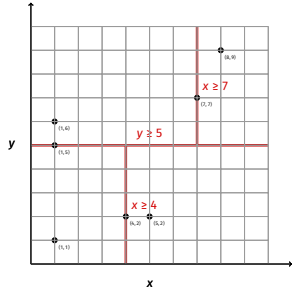
- Spread of x: 7
- Spread of y: 4
- Use x as splitting dimension.
- Median of x: 7 (or 2).



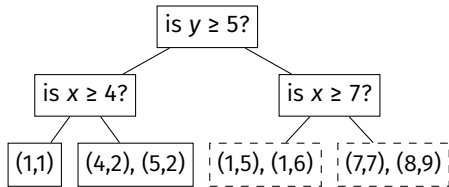
Unroll recursion, now recurse down right side of tree. Data becomes:

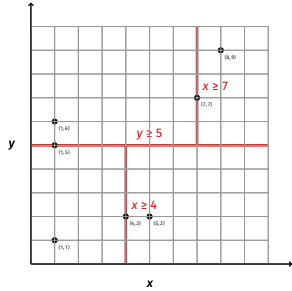
x	y
1	6
7	7
8	9
2	5

- Spread of x: 7
- Spread of y: 4
- Use x as splitting dimension.
- Median of x: 7 (or 2).

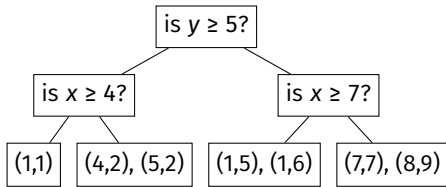


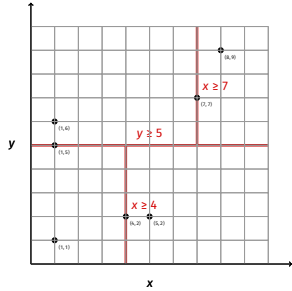
Make leaf nodes, since size $\leq M$.



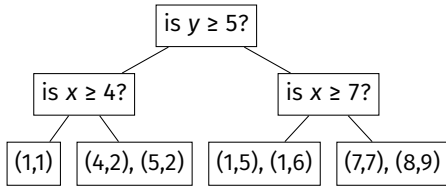


Make leaf nodes, since size $\leq M$.





Tree complete!



```
def build_kd_tree(data, m=2):
    if len(data) <= m:
        return data

    # find the dimension with greatest spread
    spread = data.max(axis=0) - data.min(axis=0)
    splitting_dimension = np.argmax(spread)

    # find the median along this dimension
    median = np.median(data[:, splitting_dimension])

    # separate the data into new left and right sets
    # note that this isn't the most efficient since it will
    # produce a copy... better to do an in-place partition
    left_data = data[data[:, splitting_dimension] < median]
    right_data = data[data[:, splitting_dimension] >= median]

    left = build_kd_tree(left_data)
    right = build_kd_tree(right_data)

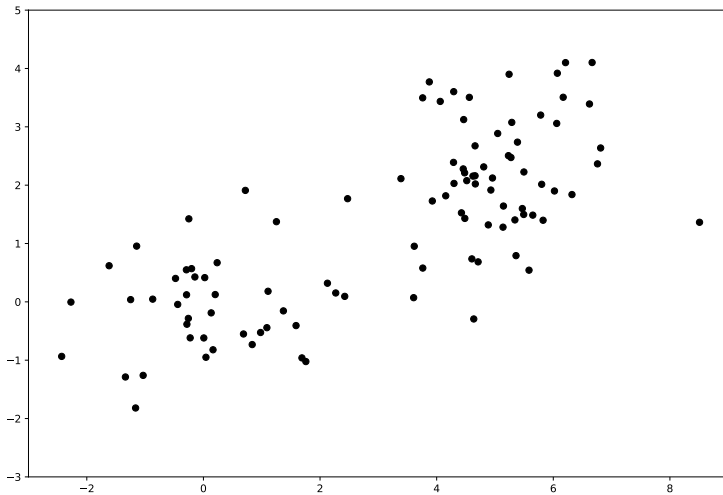
    return KDInternalNode(
        left=left, right=right, threshold=median,
        dimension=splitting_dimension
    )
```


Analysis

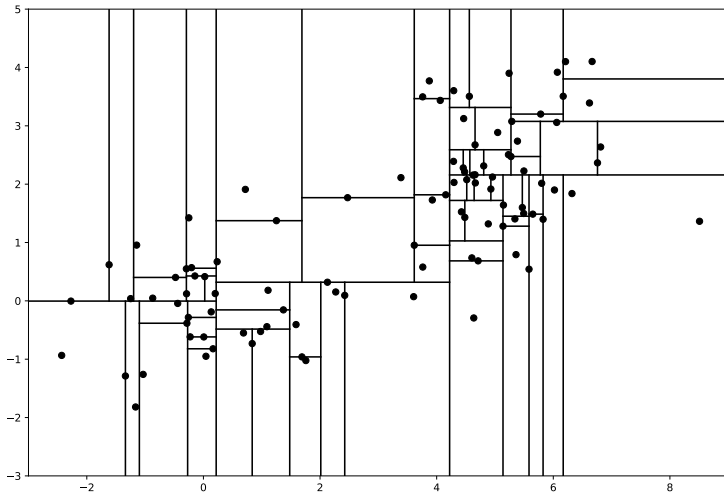
- ▶ $\Theta(k)$ to find median, perform copies, where k is number of points in subtree.
- ▶ Tree has $\Theta(\log n)$ levels (since it is balanced).
- ▶ Total time:

$$\underbrace{n}_{\text{level 1}} + \underbrace{(n/2 + n/2)}_{\text{level 2}} + \underbrace{(n/4 + n/4 + n/4 + n/4)}_{\text{level 3}} + \dots = \Theta(n \log n)$$

Example



Example



DSC 190

DATA STRUCTURES & ALGORITHMS

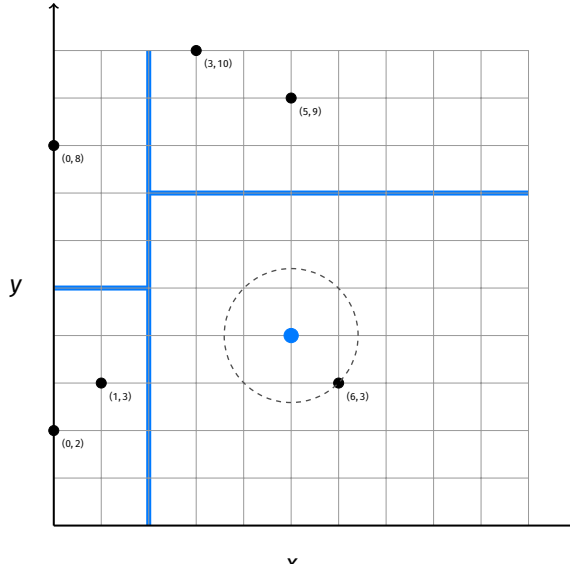
Lecture 6 | Part 5

Curse of Dimensionality

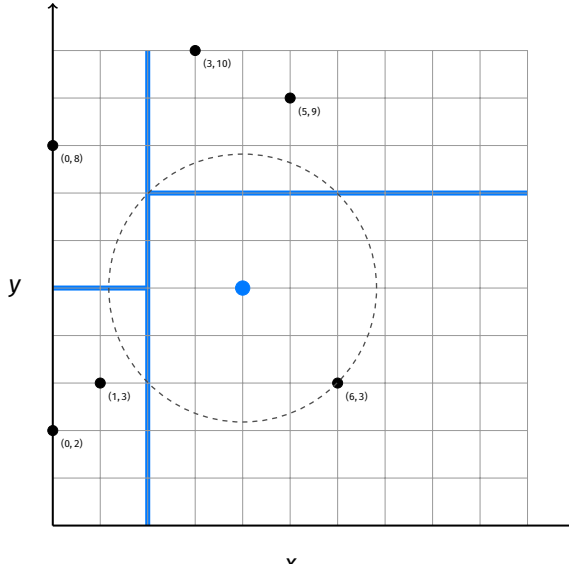
Performance Degradation

- ▶ Brute force NN search takes $\Theta(n)$ time.
- ▶ If dimensionality is small, k-d trees take $\Theta(\log n)$.
 - ▶ Great speedup!
- ▶ As dimensionality grows, performance **degrades**.
 - ▶ At worst, it is $\Theta(n)$.
 - ▶ Becomes just as bad as brute force!
- ▶ Why?

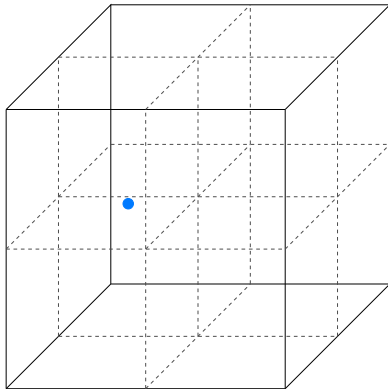
Explanation #1



Explanation #1



Explanation # 1

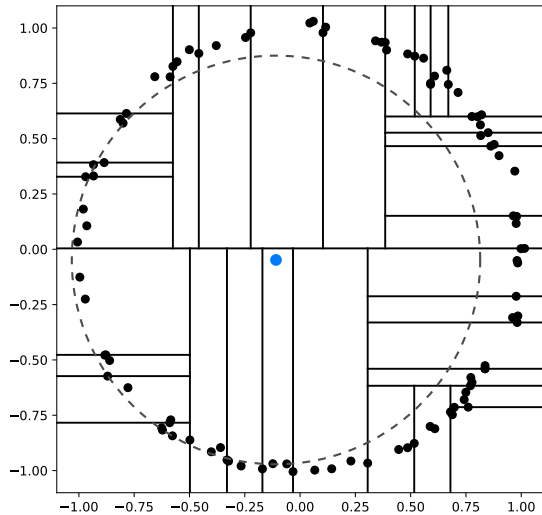


Main Idea

As d grows, the number of neighboring cells that we may need to check grows like 2^d .

Explanation #2

- ▶ We saw that if query point is far away, we cannot rule out branches.
- ▶ The reason? Distance from query to NN is not significantly different from distance between query and other points.



Surprising Fact

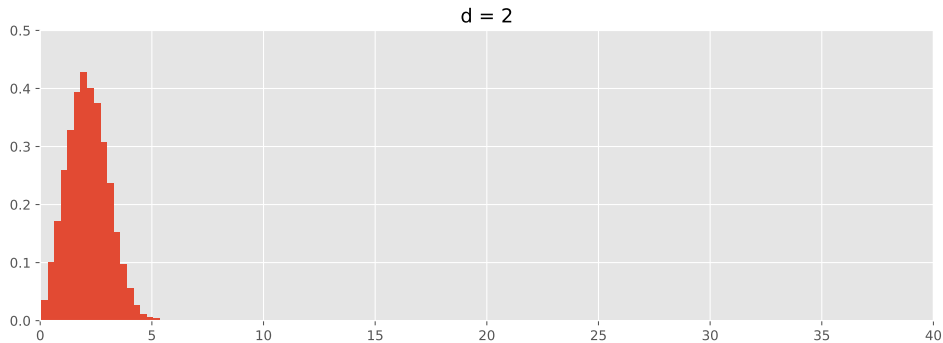
- ▶ In high dimensions³, the ratio of the distance to nearest neighbor and distance to furthest neighbor $\rightarrow 1$.

³Under some assumptions on distribution of data.

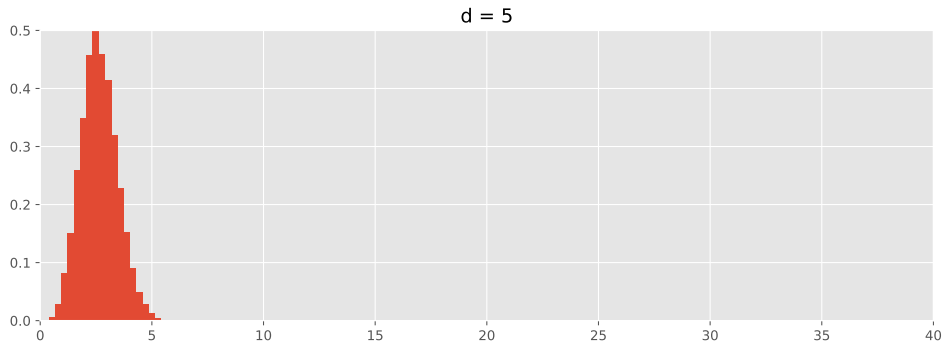
Experiment

- ▶ Generate random d -dimensional query vector from multivariate Gaussian.
- ▶ Generate 1000 d -dimensional data points from same Gaussian.
- ▶ Plot distribution of distances.

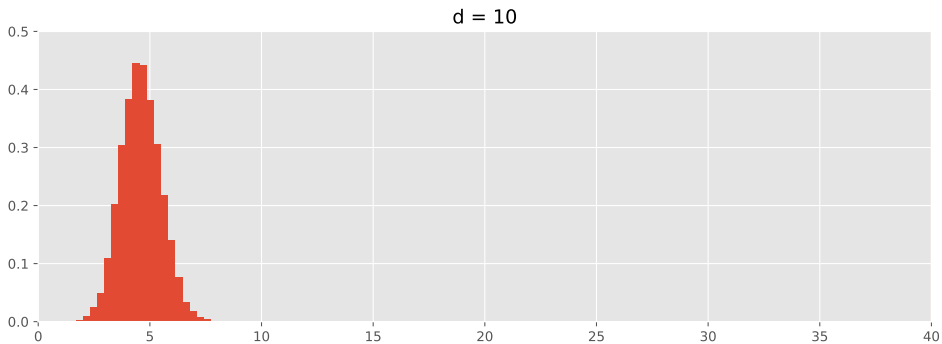
Experiment



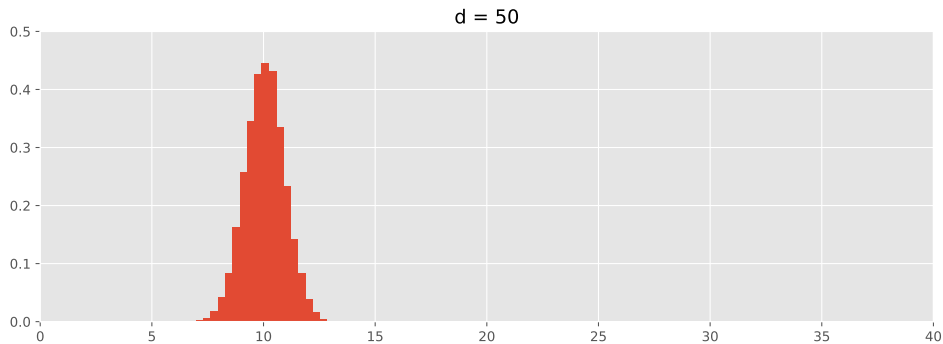
Experiment



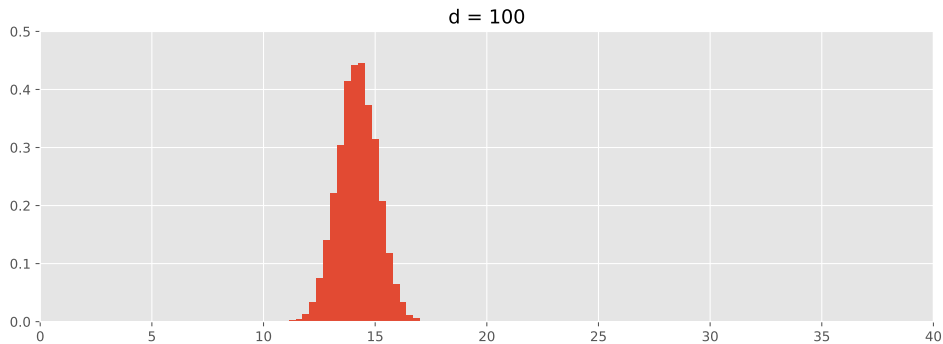
Experiment



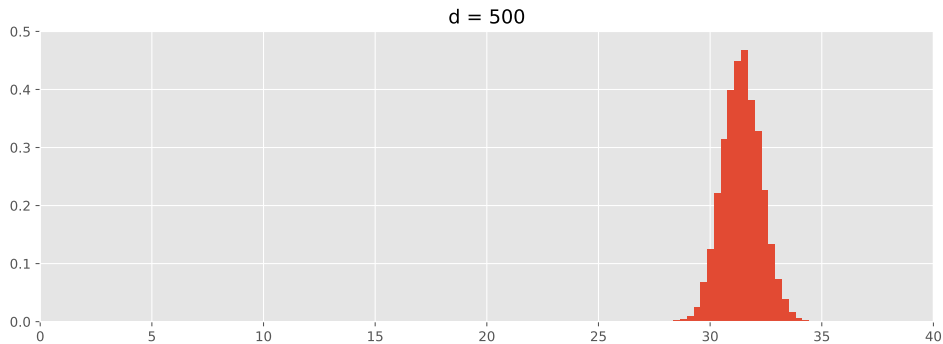
Experiment



Experiment



Experiment



Experiment

- ▶ Notice: width doesn't change, but center increases.
- ▶ So $\min = \max - \delta$, with δ constant.

$$\frac{\min}{\max} = 1 - \frac{\delta}{\max}$$

Explanation #2

- ▶ Every point in data set is approximately equidistant to query point.
- ▶ Can't rule out branches.
- ▶ Have to perform a brute force search.

Main Idea

In high dimensions, every data point is approximately equidistant to the query point, meaning we can't rule out most branches.

Main Idea

Not only are k-d trees **inefficient** in high dimensions, Euclidean distance is **less meaningful** in high dimensions, and therefore so is the concept of NN search itself.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 6 | Part 6

Approximate Nearest Neighbors

Why, exactly?

- ▶ Why do we need the **exact** NN?
- ▶ Often something close would do.
- ▶ Especially if not confident in distance measure.
 - ▶ As is the case in high dimensions.
- ▶ Maybe this can be done faster?

ANN

- ▶ **Given:** A set of points and a query point, p .
- ▶ **Return:** An **approximate nearest neighbor**.

k-D ANNs

- ▶ So far, our k-d trees find **exact** nearest neighbor.
- ▶ But there's a **very** simple way to do ANN query.
- ▶ Idea: prune more aggressively.

Before

- ▶ Let d_{nn} be distance from query point to best so far.
- ▶ Let d_{bound} be distance from query point to boundary.
- ▶ Search branch only if $d_{bound} < d_{nn}$.

Now

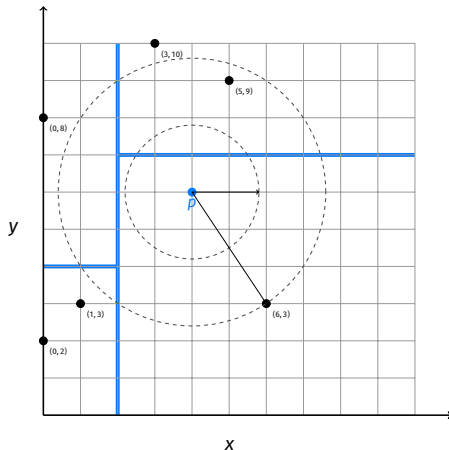
- ▶ Take $\alpha \geq 1$ as a parameter.
- ▶ Search branch only if $d_{\text{bound}} < d_{\text{nn}}/\alpha$.
- ▶ **Idea:** make it easier to toss out branch.
- ▶ If $\alpha = 1$; exact search.
- ▶ If $\alpha > 1$; approximate, faster as α grows.

Theory

- Let q be exact NN, let q_{ann} be that found by this strategy.

- Then:

$$d(p, q_{\text{ann}}) \leq \alpha \cdot d(p, q)$$



Next Time

- ▶ ANNs via **Locality Sensitive Hashing**.