K-Nearest Neighbour (Continued)

Dr. Xiaowei Huang

https://cgi.csc.liv.ac.uk/~xiaowei/

A few things:

- No lectures on
  - Week 7 (i.e., the week starting from Monday 5th November), and
  - Week 11 (i.e., the week starting from Monday 3rd December)
- Labs will continue up to Week 7

- No Class Tests
- A final exam, MCQ, 80% (the other 20% are on the CAs)

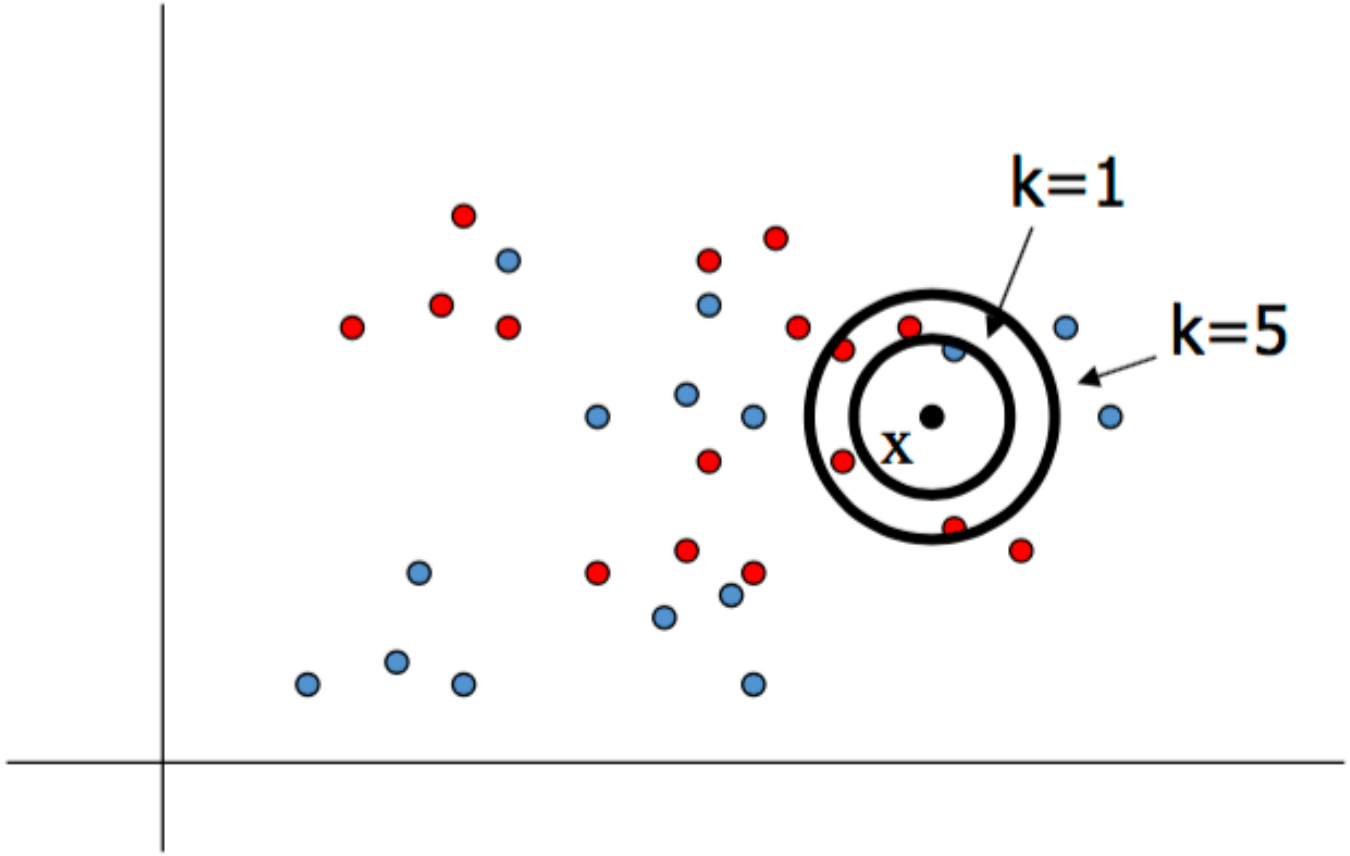
Up to now,

- Recap basic knowledge
- Decision tree learning
- k-NN classification
  - What is k-nearest-neighbor classification
  - How can we determine similarity/distance
  - Standardizing numeric features (leave this to you)

$$\hat{y} \leftarrow \underset{v \in \text{values}(Y)}{\text{argmax}} \sum_{i=1}^{k} \delta(v, y^{(i)})$$

To classify a new input vector x, examine the k-closest training data points to x and assign the object to the most frequently occurring class

$$\hat{y} \leftarrow \underset{v \in \text{values}(Y)}{\text{argmax}} \sum_{i=1}^{k} \delta(v, y^{(i)})$$



Nearest Neighbor

- **When to Consider**
  - Less than 20 attributes per instance
  - Lots of training data
- **Advantages**
  - Training is very fast
  - Learn complex target functions
  - Do not lose information
- **Disadvantages**
  - Slow at query time
  - Easily fooled by irrelevant attributes

Today's Topics

- K-NN regression
- Distance-weighted nearest neighbor
- Speeding up k-NN
  - edited nearest neighbour
  - k-d trees for nearest neighbour identification

k-nearest-neighbor *regression*

- learning stage
  - given a training set $(\mathbf{x}^{(1)}, y^{(1)}) \ldots (\mathbf{x}^{(m)}, y^{(m)})$, do nothing
    - (it's sometimes called a *lazy learner*)

- classification stage
  - **given**: an instance $x^{(q)}$ to classify
  - find the k training-set instances $(\mathbf{x}^{(1)}, y^{(1)}) \ldots (\mathbf{x}^{(k)}, y^{(k)})$ that are most similar to $x^{(q)}$
  - return the value $\hat{y} \leftarrow \frac{1}{k}\sum_{i=1}^{k} y^{(i)}$

Average over neighbours' values

Distance-weighted nearest neighbor

- We can have instances contribute to a prediction according to their distance from $x^{(q)}$
- classification:

$$\hat{y} \leftarrow \underset{v \in \text{values}(Y)}{\text{argmax}} \sum_{i=1}^{k} w_i \, \delta(v, y^{(i)}) \qquad w_i = \frac{1}{d(x^{(q)}, x^{(i)})^2}$$

Intuition: instances closer to the current one is more important.

reciprocal of the distance

- regression:

$$\hat{y} \leftarrow \frac{\sum_{i=1}^{k} w_i \, y^{(i)}}{\sum_{i=1}^{k} w_i}$$

Issues

- Choosing k
  - Increasing k reduces variance, increases bias
- For high-dimensional space, problem that the nearest neighbor may not be very close at all!
- Memory-based technique. Must make a pass through the data for each classification. This can be prohibitive for large data sets.

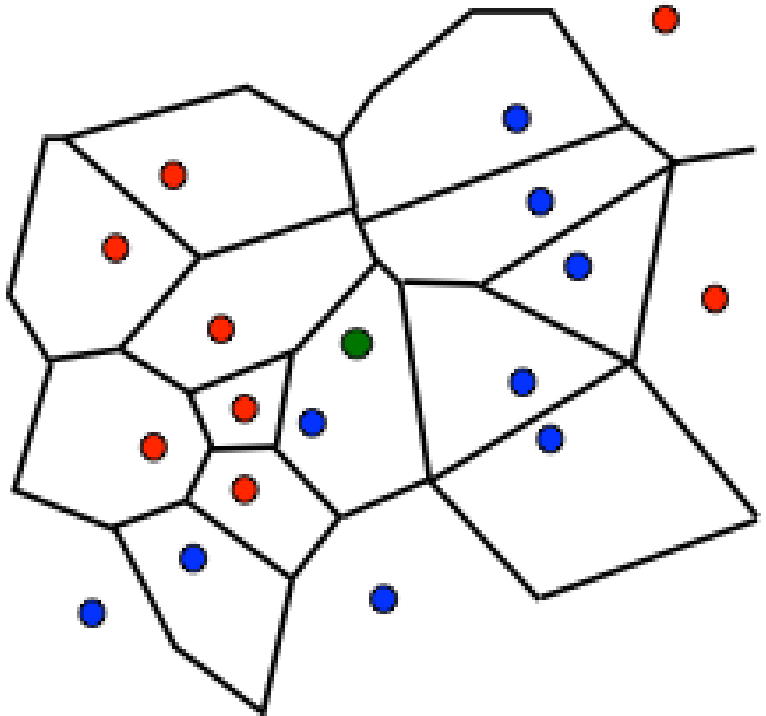
Nearest neighbour problem

- Given sample $S = ((x_1,y_1),\ldots,(x_m,y_m))$ and a test point x,
- it is to find the nearest k neighbours of x.

- Note: for the algorithms, dimensionality N, i.e., number of features, is crucial.

Efficient Indexing: N=2

- Algorithm
  - compute Voronoi diagram in O(m log m)
    - See algorithm in https://en.wikipedia.org/wiki/Fortune's_algorithm
  - use point location data structure to determine nearest neighbours
  - complexity: O(m) space, O(log m) time.



Efficient Indexing: N>2

- Voronoi diagram: size in $O(m^{N/2})$
- Linear algorithm (no pre-processing):
  - compute distance $||x - x_i||$ for all $i \in [1, m]$.
  - complexity of distance computation: $\Omega(N\ m)$.
  - no additional space needed.

k-NN is a "lazy" learning algorithm – does virtually nothing at training time

but classification/prediction time can be costly when the training set is large

Efficient Indexing: N>2

- two general strategies for alleviating this weakness
  - don’t retain every training instance (edited nearest neighbor)
  - pre-processing. Use a smart data structure to look up nearest neighbors (e.g. a k-d tree)

Edited instance-based learning

- select a subset of the instances that still provide accurate classifications
- *incremental deletion*

  start with all training instances in memory
  for each training instance $(\boldsymbol{x}^{(i)}, y^{(i)})$
  if other training instances provide correct classification for $(\boldsymbol{x}^{(i)}, y^{(i)})$
  delete it from the memory

- *incremental growth*

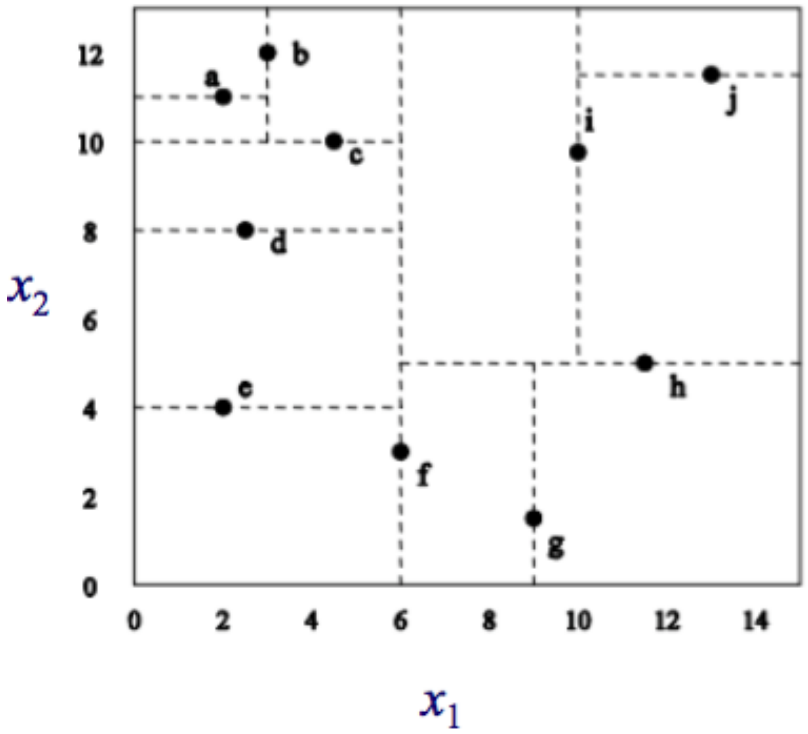
  start with an empty memory
  for each training instance $(\boldsymbol{x}^{(i)}, y^{(i)})$
  if other training instances in memory **don't** correctly classify $(\boldsymbol{x}^{(i)}, y^{(i)})$
  add it to the memory

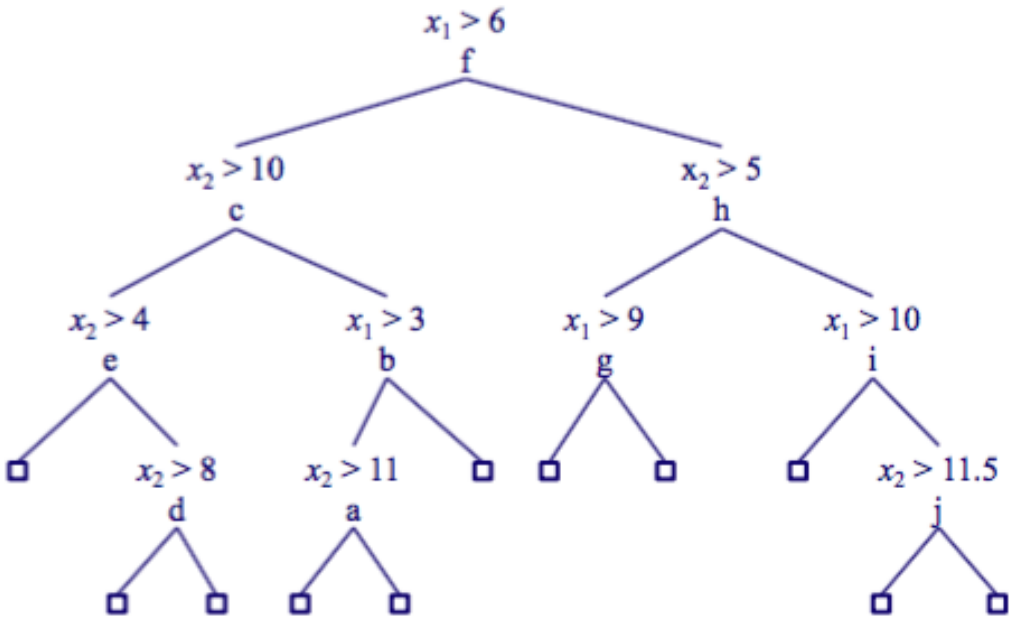
Q1: Does ordering matter?

Q2: If following the optimal ordering, do the two approaches produce the same subset of instances?

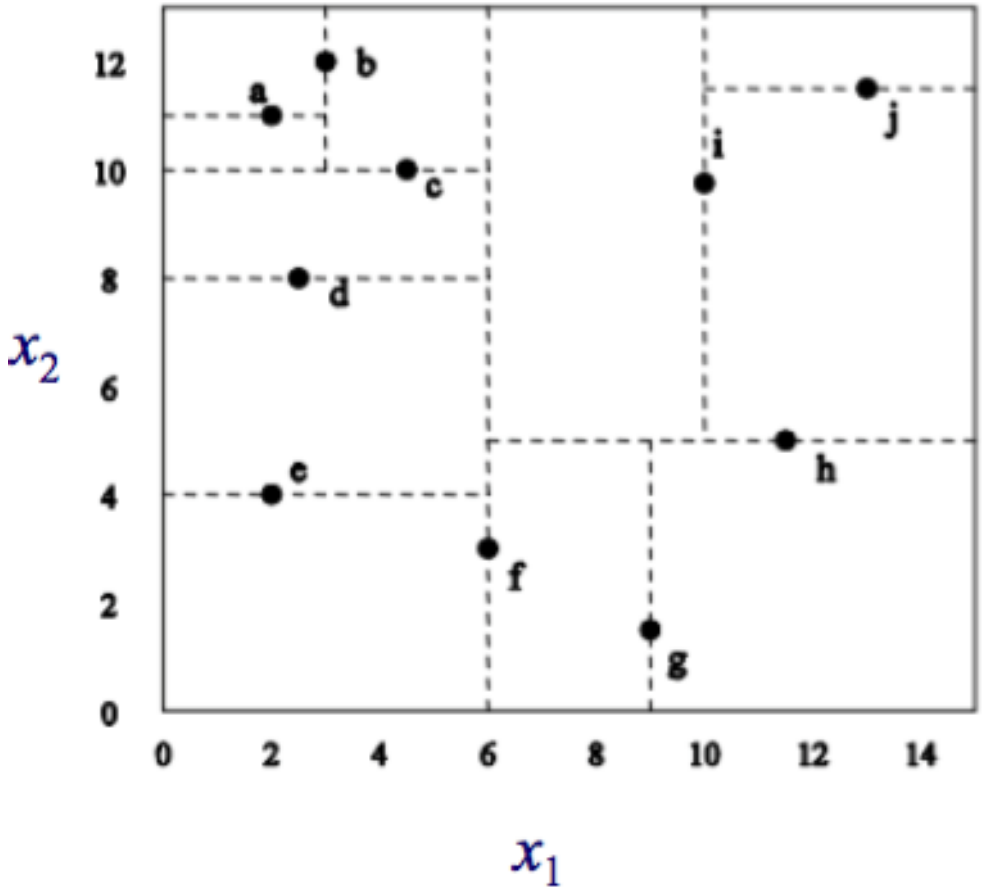
k-d trees

- a *k-d tree* is similar to a decision tree except that each internal node
  - stores one instance
  - splits on the median value of the feature having the highest variance





Construction of k-d tree



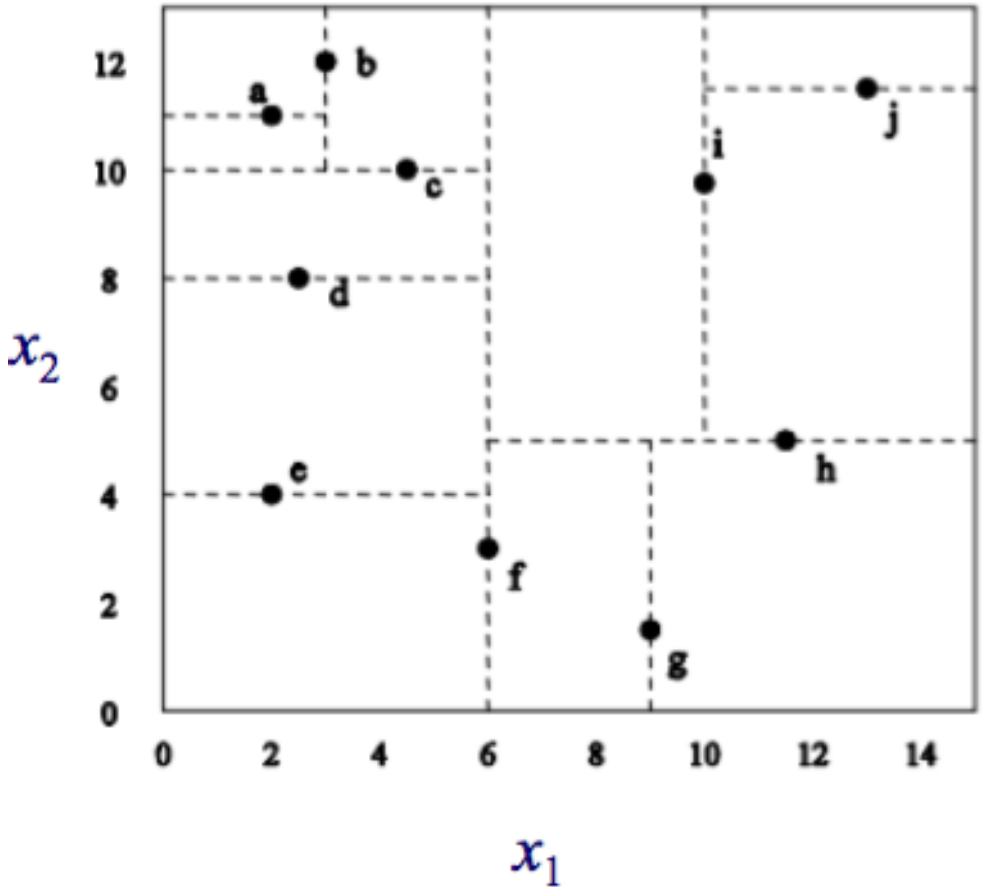
median value of the feature having the highest variance?

-- point f, $x_1 = 6$

$x_1>6$

f

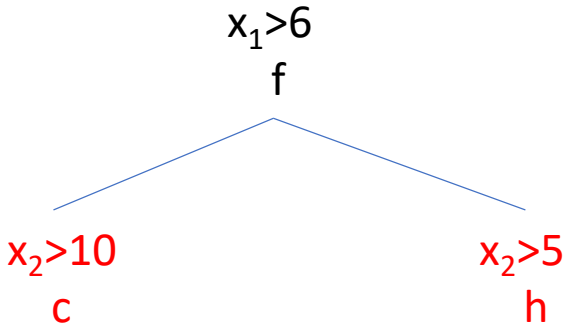
Construction of k-d tree



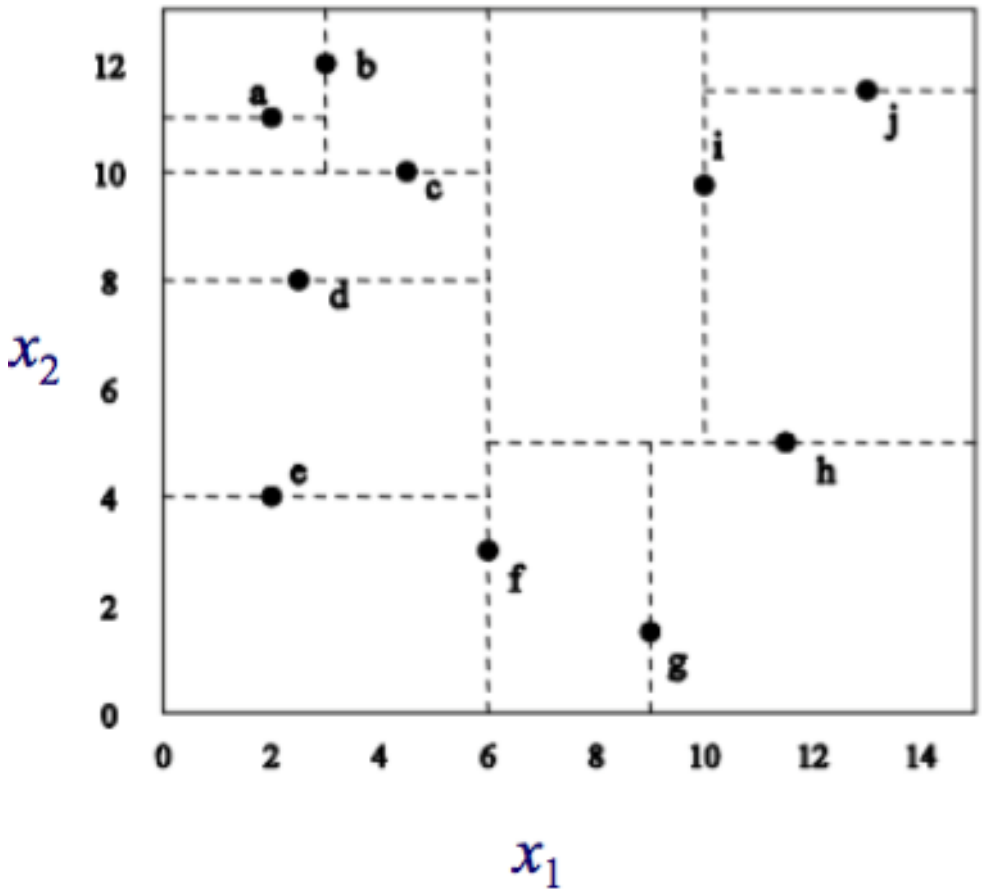
median value of the feature having the highest variance?

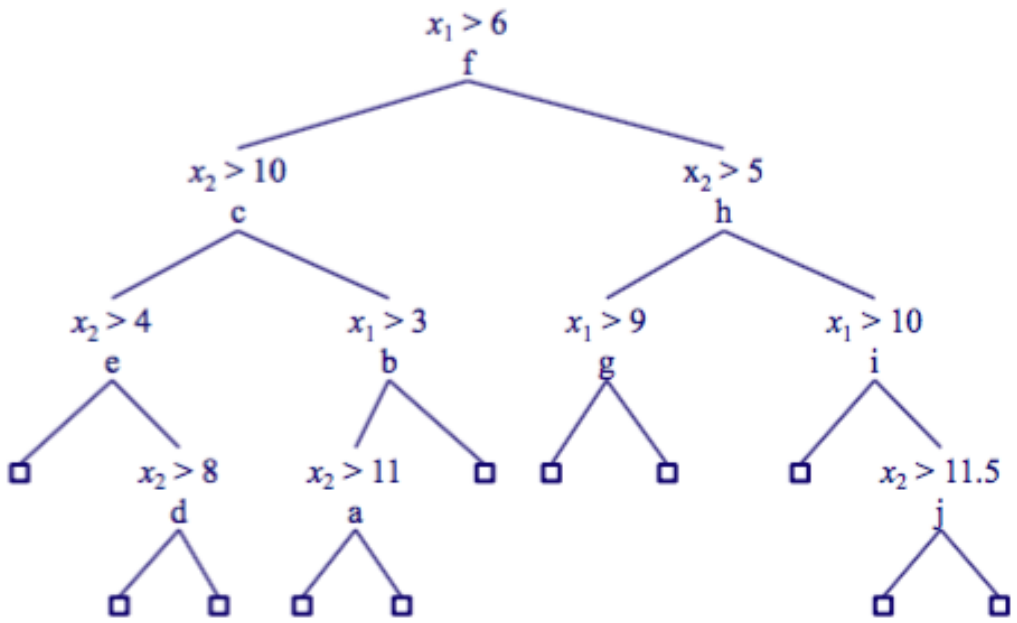
-- point f, $x_1 = 6$

-- point c, $x_2 = 10$ and point h, $x_2 = 5$



Construction of k-d tree





There can be other methods of constructing k-d trees, see e.g., https://en.wikipedia.org/wiki/K-d_tree#Nearest_neighbour_search

Finding nearest neighbors with a k-d tree

- use branch-and-bound search
- priority queue stores
  - nodes considered
  - lower bound on their distance to query instance

- lower bound given by distance using a single feature

- average case: $O(\log_2 m)$
- worst case: $O(m)$ where m is the size of the training-set

Finding nearest neighbours in a k-d tree

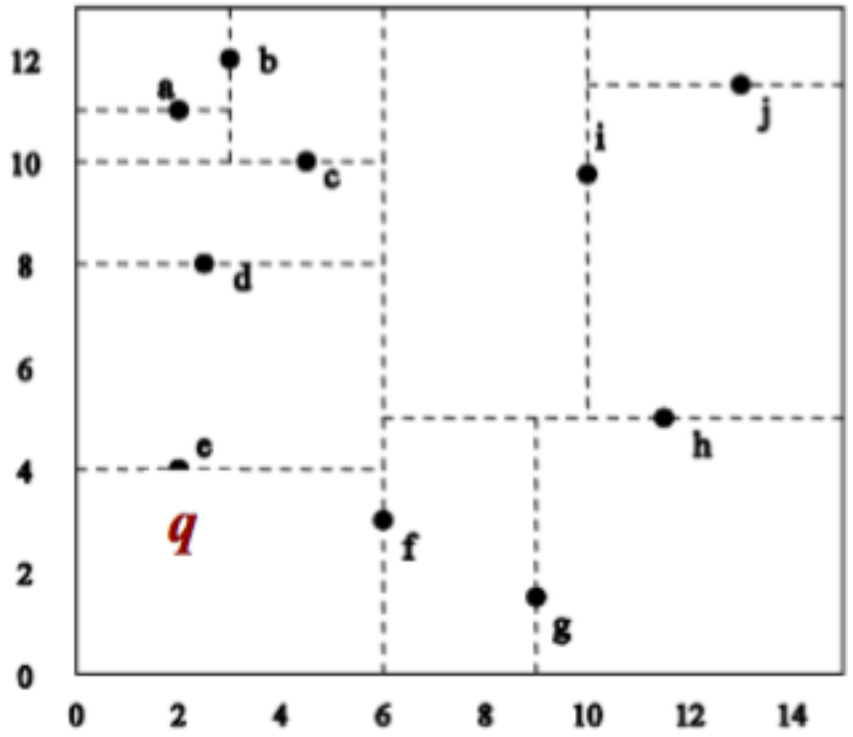
```
NearestNeighbor(instance x^(q))
    PQ = { }                                        // minimizing priority queue
    best_dist = ∞                                   // smallest distance seen so far
    PQ.push(root, 0)
    while PQ is not empty
        (node, bound) = PQ.pop();
        if (bound ≥ best_dist)
            return best_node.instance               // nearest neighbor found
        dist = distance(x^(q), node. instance)
        if (dist < best_dist)
            best_dist = dist
            best_node = node
        if (q[node.feature] – node.threshold > 0)
            PQ.push(node.left, x^(q)[node.feature] – node.threshold)
            PQ.push(node.right, 0)
        else
            PQ.push(node.left, 0)
            PQ.push(node.right, node. threshold - x^(q) [node.feature])
    return best_node. instance
```

Intuitively, for a pair (*node*,*value*), *value* represents the smallest guaranteed distance, i.e., greatest lower bound up to now, from the instance $x^{(q)}$ to the set of instances over which *node* is the selected one to split

For example, the set of instances where *root* is the selected one to split over is the whole training set.

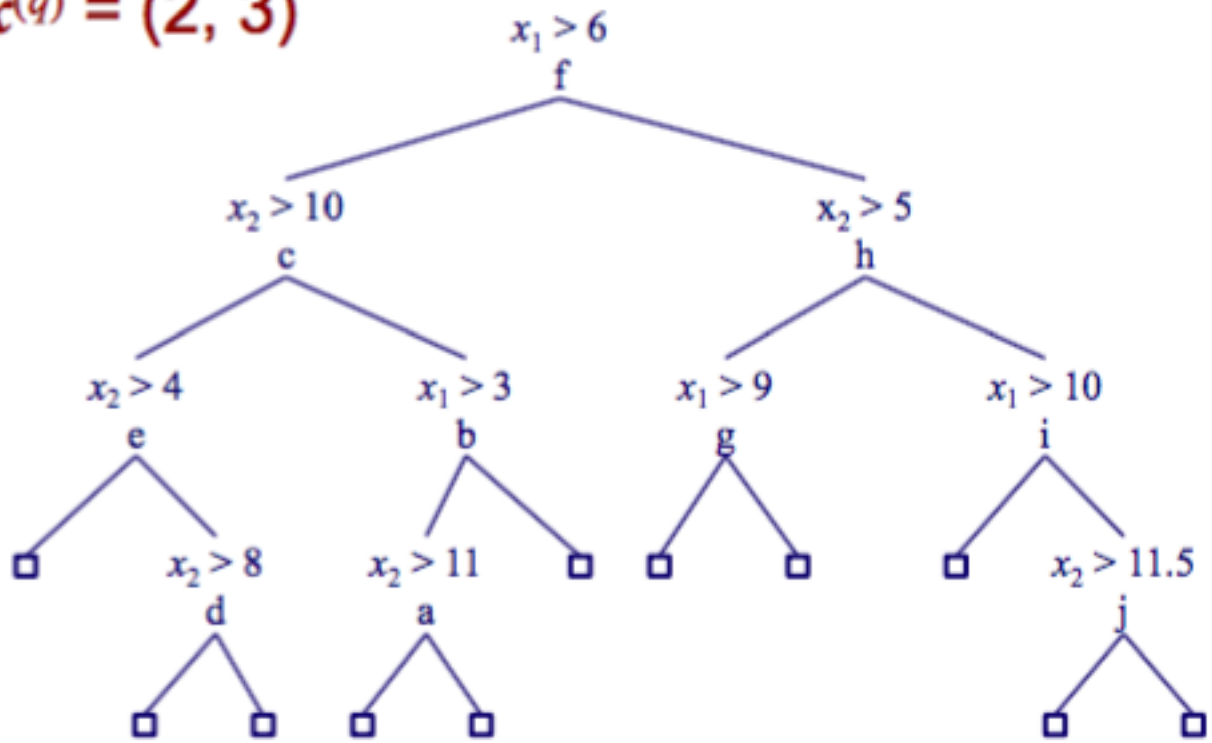
(root,0) means that at the beginning, the guaranteed smallest distance to the training set is 0

k-d tree example (Manhattan distance)

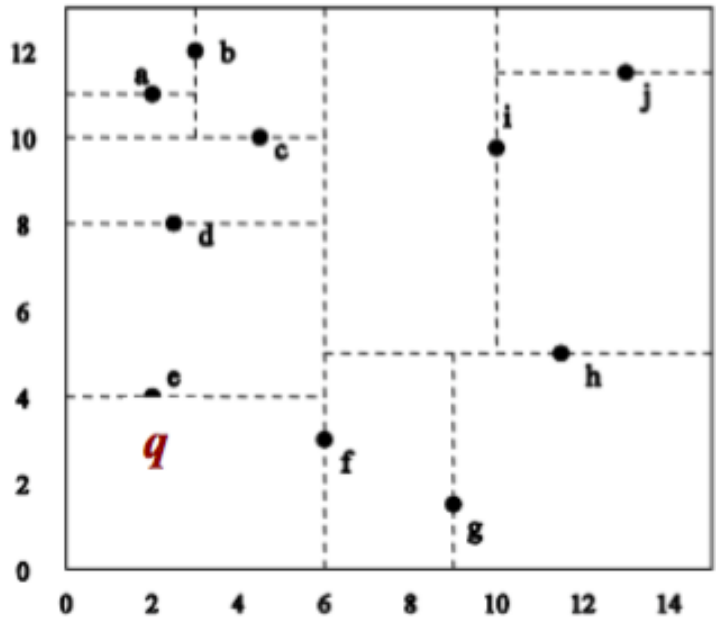


given query

$x^{(q)} = (2, 3)$

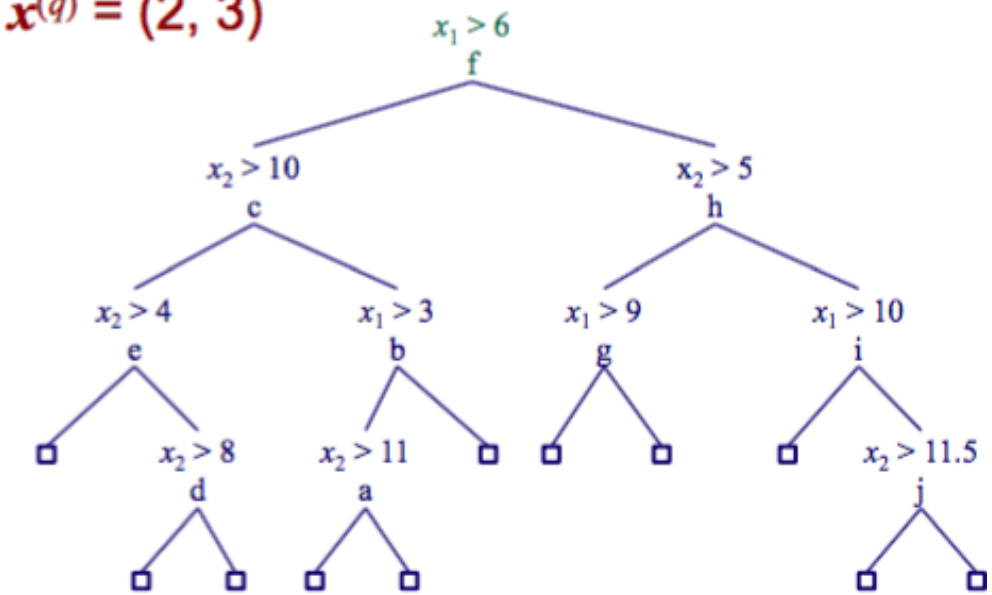


k-d tree example (Manhattan distance)



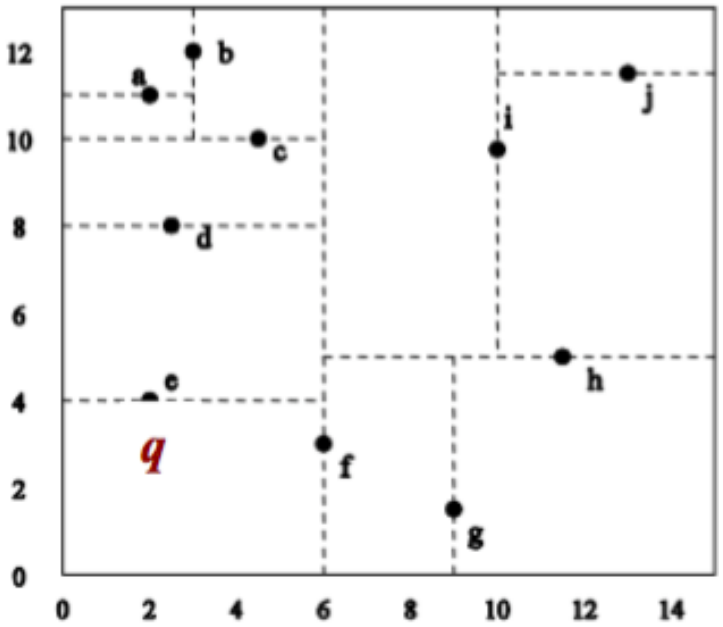
given query

$x^{(q)} = (2, 3)$



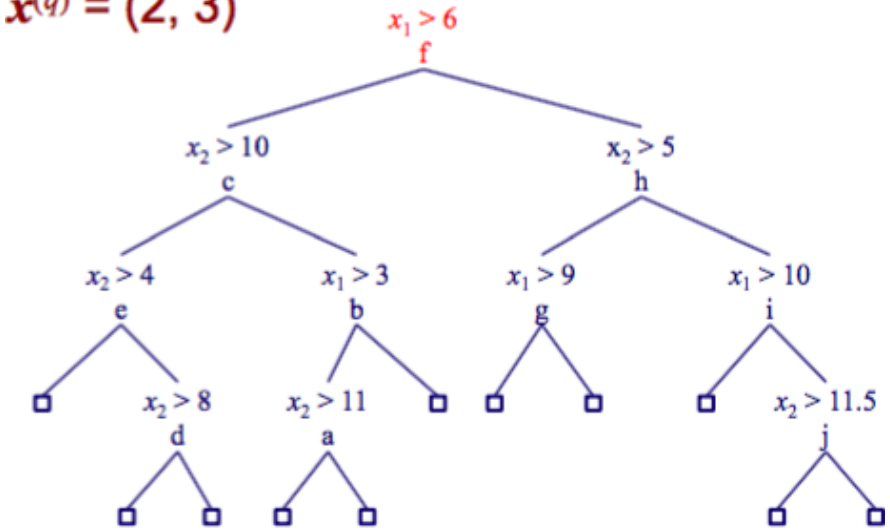
distance	best distance	best node	priority queue
	∞		(f, 0)

k-d tree example (Manhattan distance)



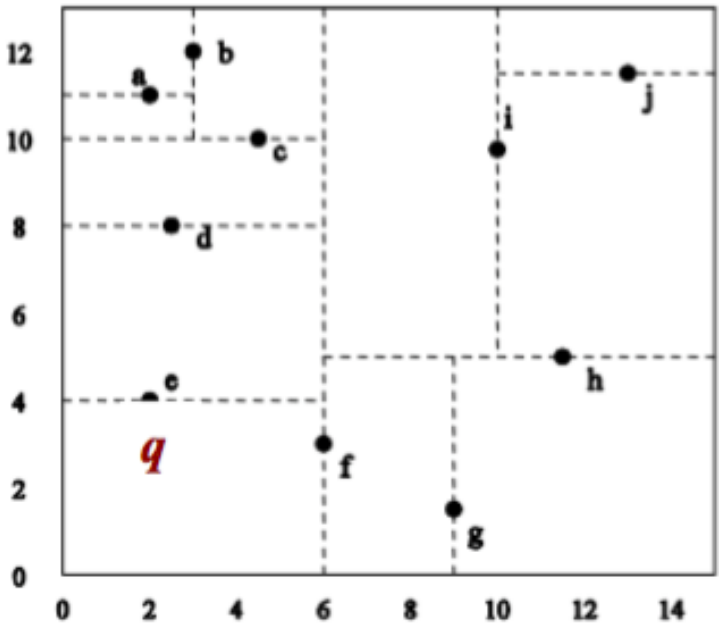
given query

$x^{(q)} = (2, 3)$

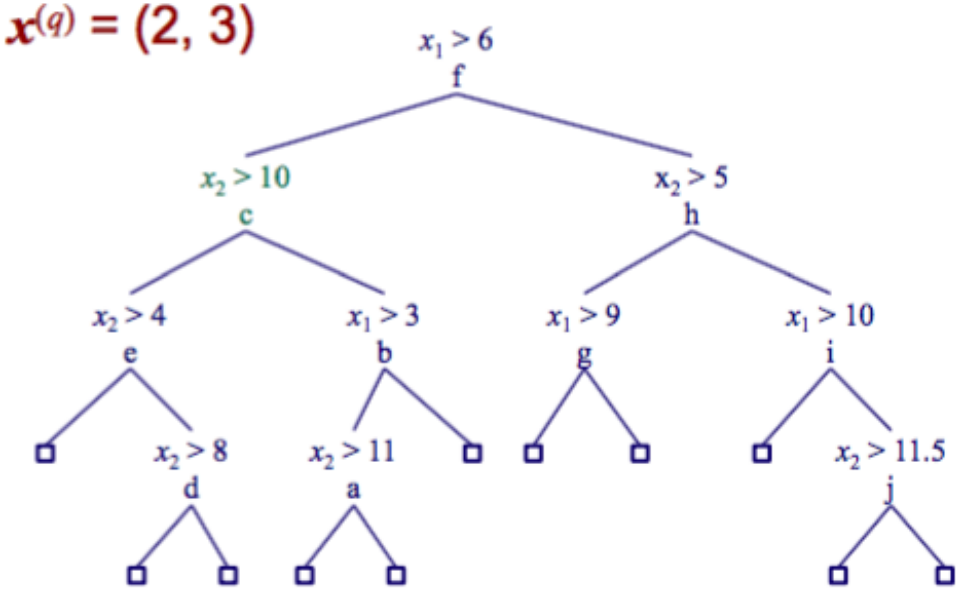


	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	

k-d tree example (Manhattan distance)

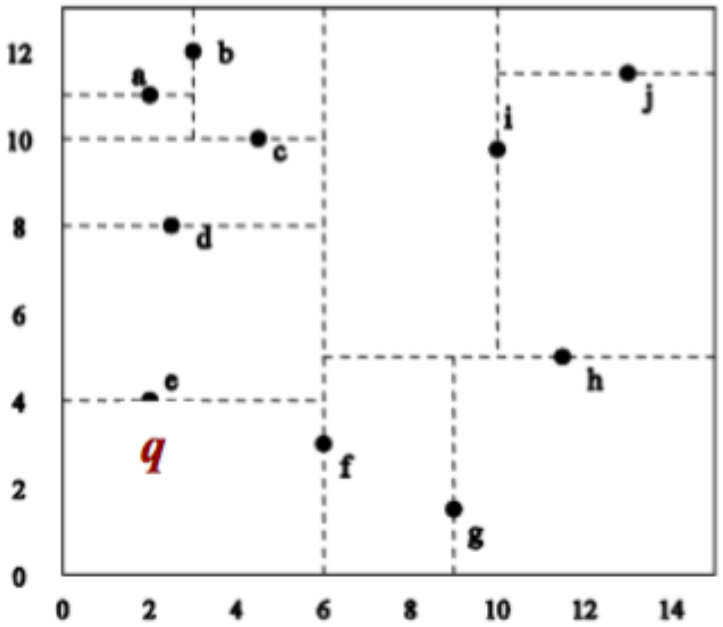


given query
$x^{(q)} = (2, 3)$

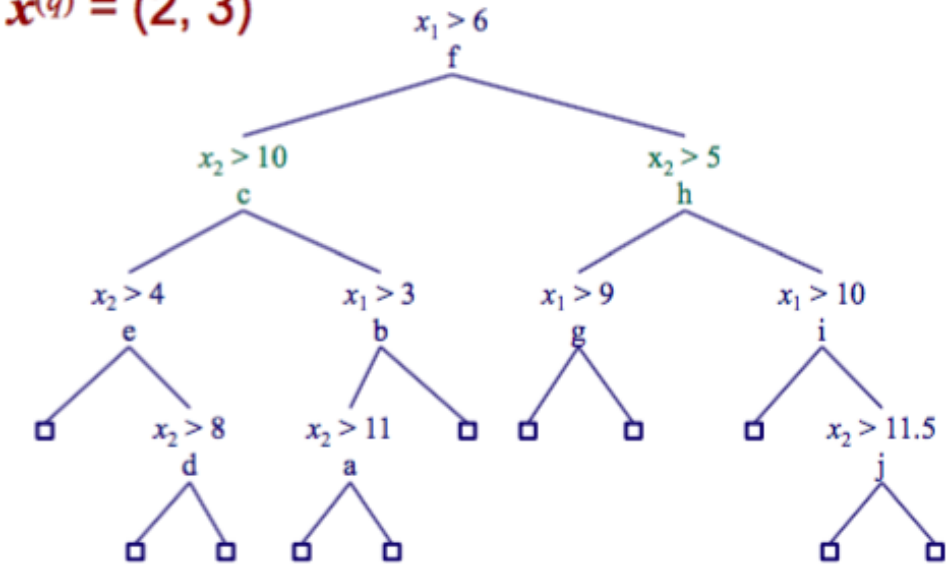


	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	(c, 0)

k-d tree example (Manhattan distance)

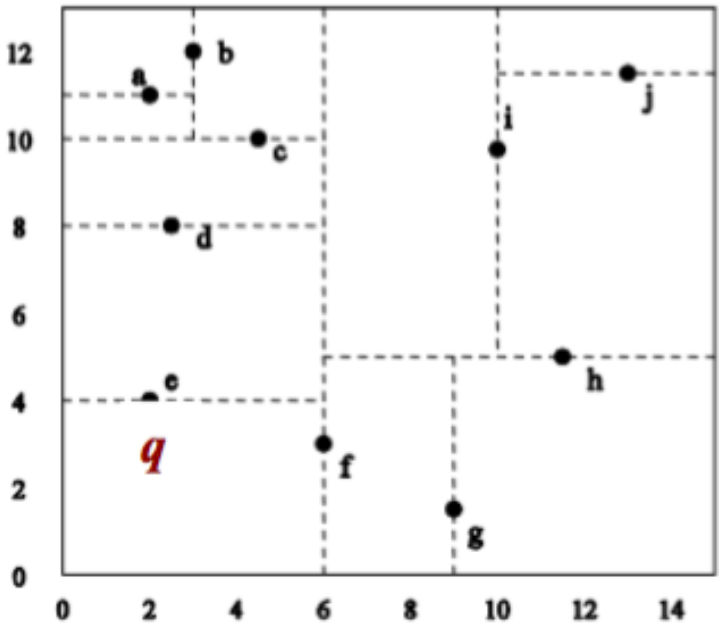


given query
$x^{(q)} = (2, 3)$

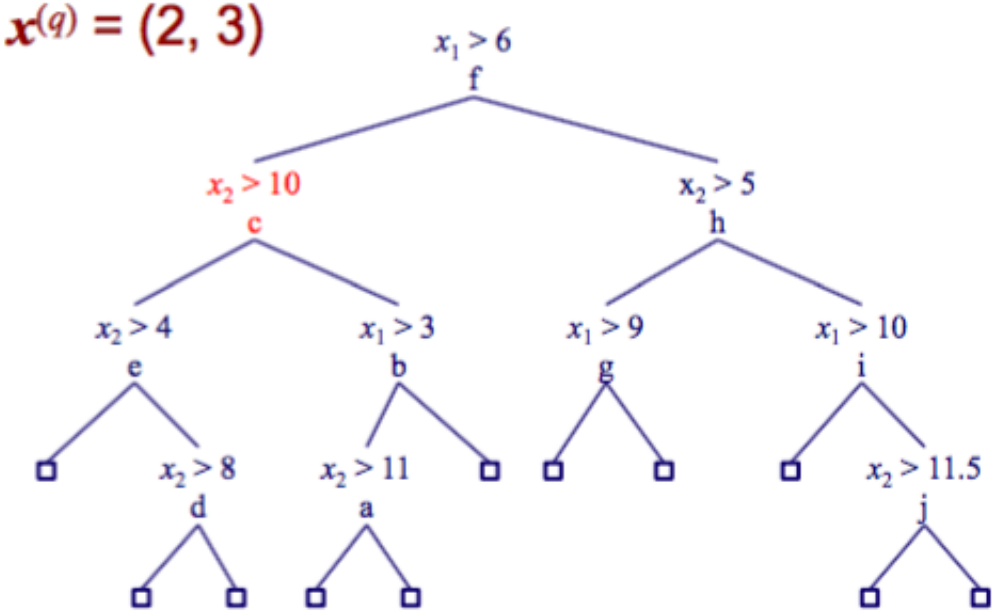


	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	(c, 0) (h, 4)

k-d tree example (Manhattan distance)

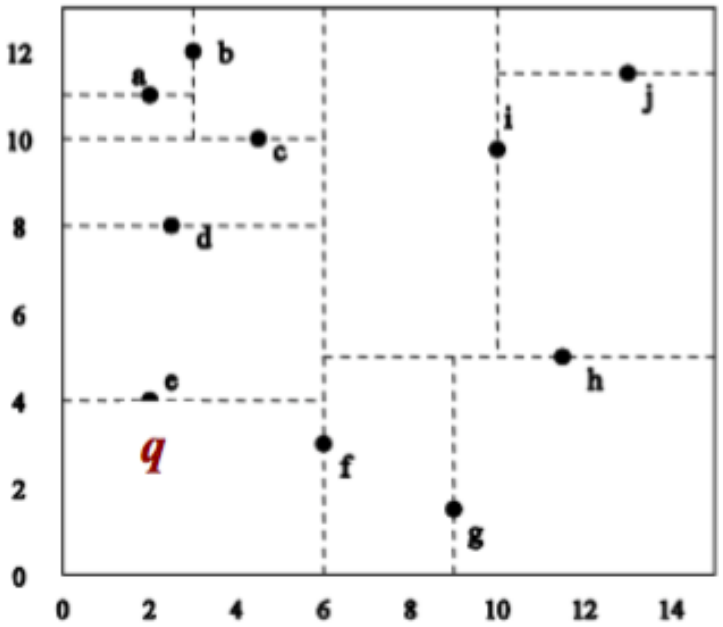


given query
$x^{(q)} = (2, 3)$



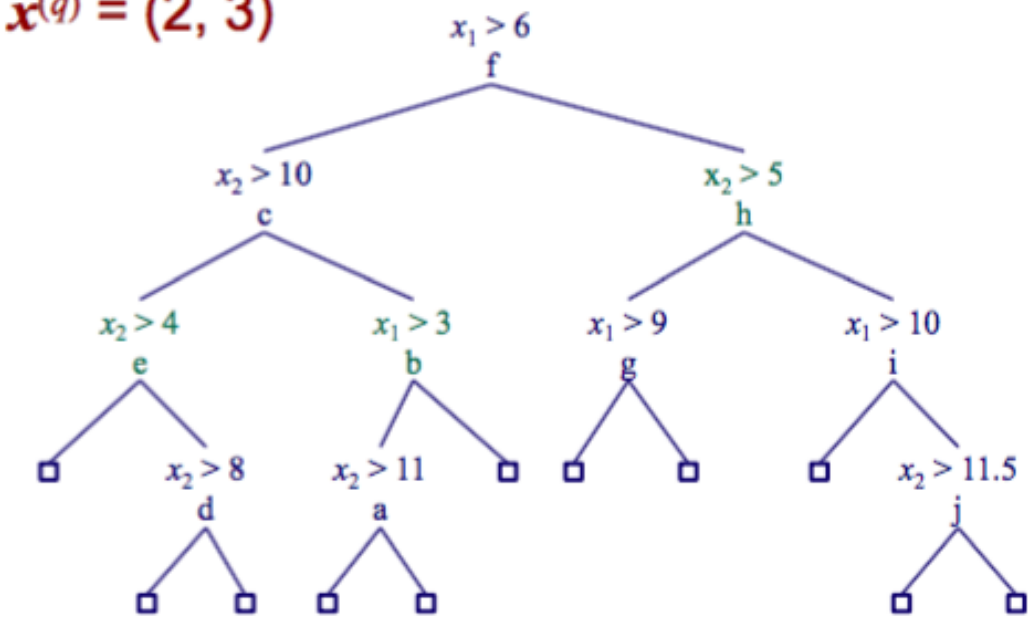
	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	(c, 0) (h, 4)
pop c	10.0	4.0	f	

k-d tree example (Manhattan distance)



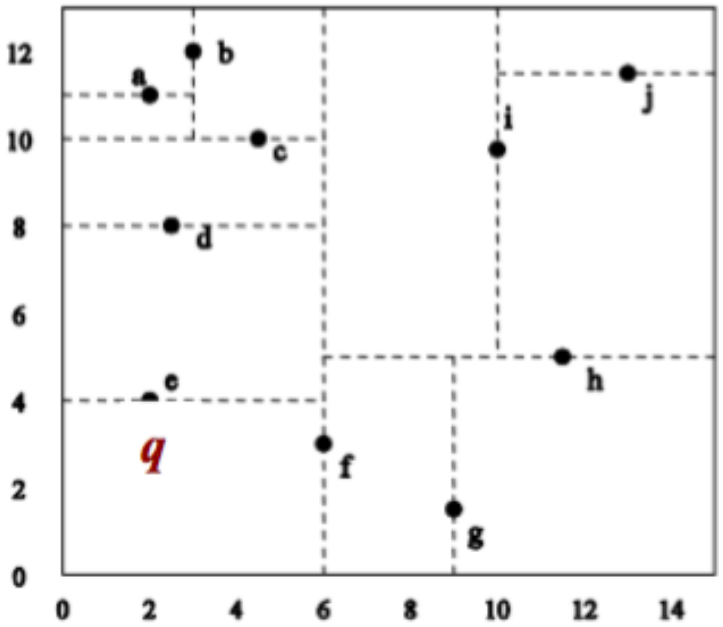
given query

$x^{(q)} = (2, 3)$

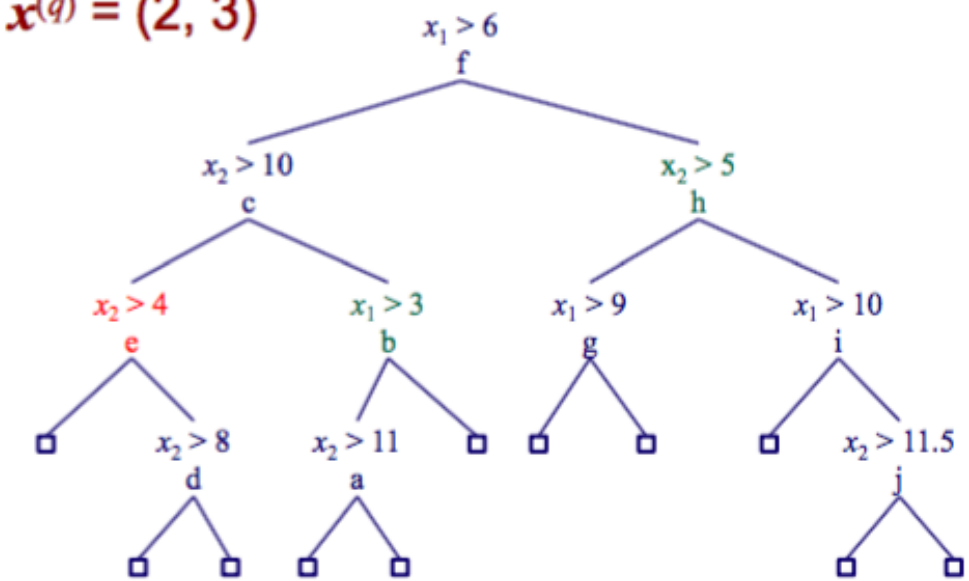


	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	(c, 0) (h, 4)
pop c	10.0	4.0	f	(e, 0) (h, 4) (b, 7)

k-d tree example (Manhattan distance)

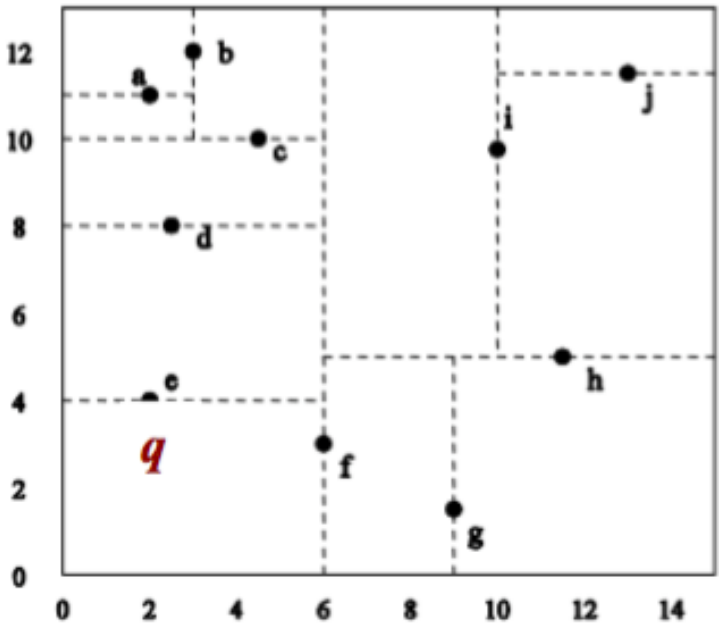


given query
$x^{(q)} = (2, 3)$

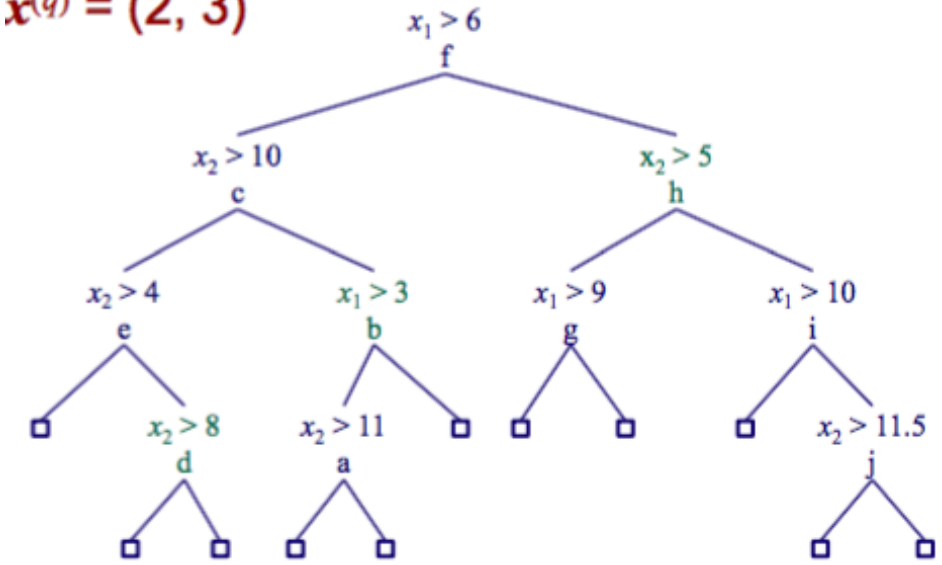


	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	(c, 0) (h, 4)
pop c	10.0	4.0	f	(e, 0) (h, 4) (b, 7)
pop e	1.0	1.0	e	

k-d tree example (Manhattan distance)

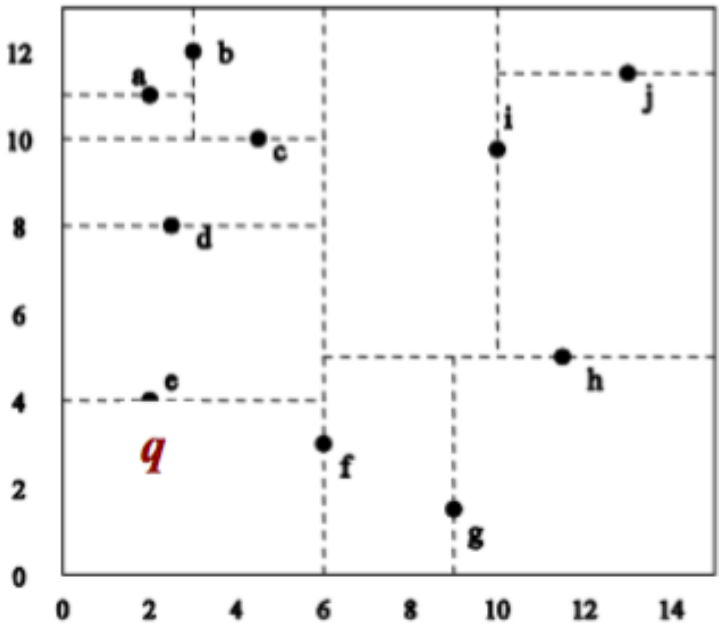


given query
$x^{(q)} = (2, 3)$

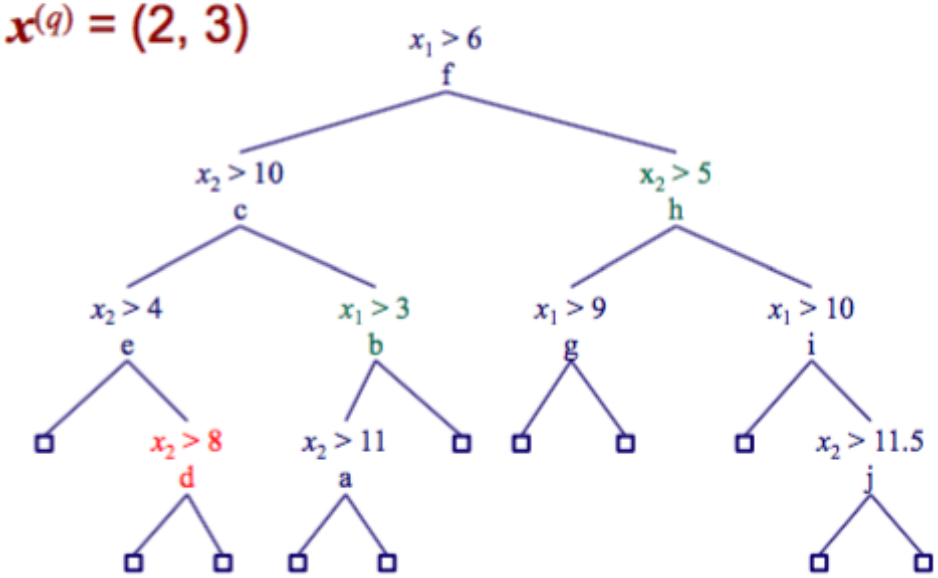


	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	(c, 0) (h, 4)
pop c	10.0	4.0	f	(e, 0) (h, 4) (b, 7)
pop e	1.0	1.0	e	(d, 1) (h, 4) (b, 7)

k-d tree example (Manhattan distance)



given query
$x^{(q)} = (2, 3)$



	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	(c, 0) (h, 4)
pop c	10.0	4.0	f	(e, 0) (h, 4) (b, 7)
pop e	1.0	1.0	e	(d, 1) (h, 4) (b, 7)
pop d	return e			

Extended Materials: Voronoi Diagram Generation

- https://en.wikipedia.org/wiki/Voronoi_diagram
- https://courses.cs.washington.edu/courses/cse326/00wi/projects/voronoi.html