K-Nearest Neighbour (Continued)

Dr. Xiaowei Huang

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

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)

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

Today’s Topics

- Definition
- Speeding up k-NN
  - edited nearest neighbour
  - k-d trees for nearest neighbour identification
- Variants of k-NN
  - K-NN regression
  - Distance-weighted nearest neighbor
  - Locally weighted regression to handle irrelevant features
- Discussions
  - Strengths and limitation of instance-based learning
  - Inductive bias

Definition

k-nearest-neighbor classification

- classification task
  - **given**: an instance $x^{(q)}$ to classify
  - find the k training-set instances $(\mathbf{x}^{(1)}, y^{(1)}) \ldots (x^{(k)}, y^{(k)})$ that are the most similar to $x^{(q)}$
  - return the class value

$$\hat{y} \leftarrow \underset{v \in \text{values}(Y)}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, y^{(i)}) \qquad \delta(a,b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$

  - (i.e. return the class that have the most number of instances in the k training instances

How can we determine similarity/distance

- suppose all features are discrete
  - Hamming distance (or L^0 norm): count the number of features for which two instances differ

- Example: X = (Weekday, Happy?, Weather) Y = AttendLecture?
  - D : in the table
  - New instance: <Friday, No, Rain>
  - Distances = {2, 3, 1, 2}
  - For 1-nn, which instances should be selected?
  - For 2-nn, which instances should be selected?
  - For 3-nn, which instances should be selected?

v1	**v2**	**v3**	**y**
Wed	Yes	Rain	No
Wed	Yes	Sunny	Yes
Thu	No	Rain	Yes
Fri	Yes	Sunny	No

New datum

Fri	No	Rain	

How can we determine similarity/distance

- Example: X = (Weekday, Happy?, Weather) Y = AttendLecture?
  - New instance: <Friday, No, Rain>
  - For 3-nn, selected instances: {(<Wed, Yes, Rain>, No), (<Thu, No, Rain>, Yes), (<Fri, Yes, Sunny>, No)}
- Classification:

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

  - v = Yes. $\sum_{i=1}^{k} \delta(v, y^{(i)}) = 0 + 1 + 0 = 1$
  - v = No. $\sum_{i=1}^{k} \delta(v, y^{(i)}) = 1 + 0 + 1 = 2$

So, which class this new instance should be in?

How can we determine similarity/distance

- suppose all features are continuous
  - Euclidean distance:

$$d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sqrt{\sum_f \left(x_f^{(i)} - x_f^{(j)}\right)^2}$$

where $x_f^{(i)}$ represents the f -th feature of $x^{(i)}$

  - Manhattan distance:

$$d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sum_f \left|x_f^{(i)} - x_f^{(j)}\right|$$

Recall the difference and similarity with L^p norm

How can we determine similarity/distance

- Example: X = (Height, Weight, RunningSpeed) Y = SoccerPlayer?
  - D: in the table
  - New instance: <185, 91, 13.0>
  - Suppose that Euclidean distance is used.
  - Is this person a soccer player?

v1	**v2**	**v3**	**y**
182	87	11.3	No
189	92	12.3	Yes
178	79	10.6	Yes
183	90	12.7	No

New datum

185	91	13.0	

How can we determine similarity/distance

- if we have a mix of discrete/continuous features:

$$d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sum_{f} \begin{cases} \left| x_f^{(i)} - x_f^{(j)} \right| & \text{if } f \text{ is continuous} \\ 1 - \delta\left(x_f^{(i)}, x_f^{(i)}\right) & \text{if } f \text{ is discrete} \end{cases}$$

- typically want to apply to continuous features some type of normalization (values range 0 to 1) or standardization (values distributed according to standard normal)
- many other possible distance functions we could use ...

Standardizing numeric features

- given the training set D, determine the mean and stddev for feature x_i

$$\mu_i = \frac{1}{|D|} \sum_{d=1}^{|D|} x_i^{(d)} \qquad \sigma_i = \sqrt{\frac{1}{|D|} \sum_{d=1}^{|D|} \left(x_i^{(d)} - \mu_i \right)^2}$$

- standardize each value of feature x_i as follows

$$\hat{x}_i^{(d)} = \frac{x_i^{(d)} - \mu_i}{\sigma_i}$$

- do the same for test instances, using the same μ and σ derived from the *training* data

Speeding up k-NN

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.

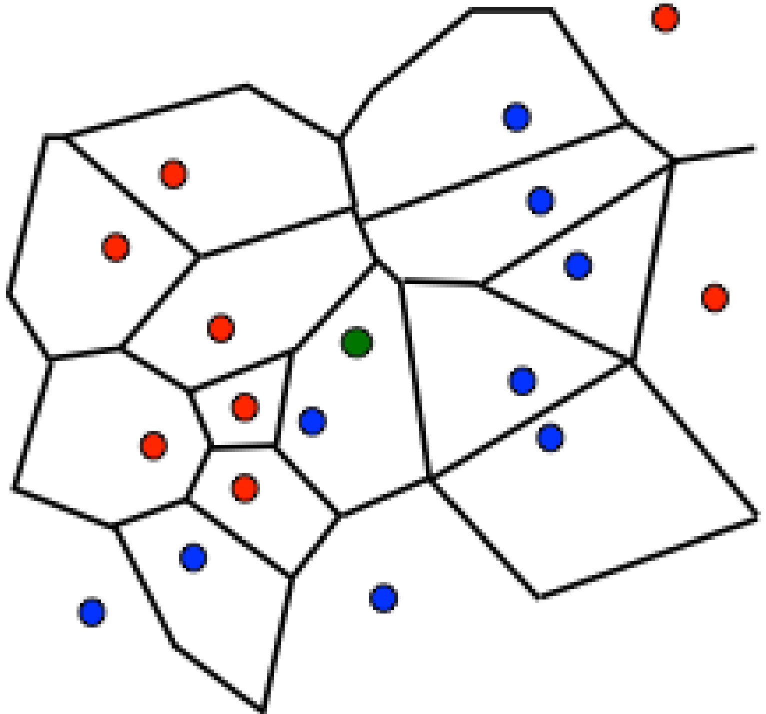
Nearest neighbour problem

- 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 $(x^{(i)}, y^{(i)})$
        if other training instances provide correct classification for $(x^{(i)}, y^{(i)})$
            delete it from the memory

- *incremental growth*

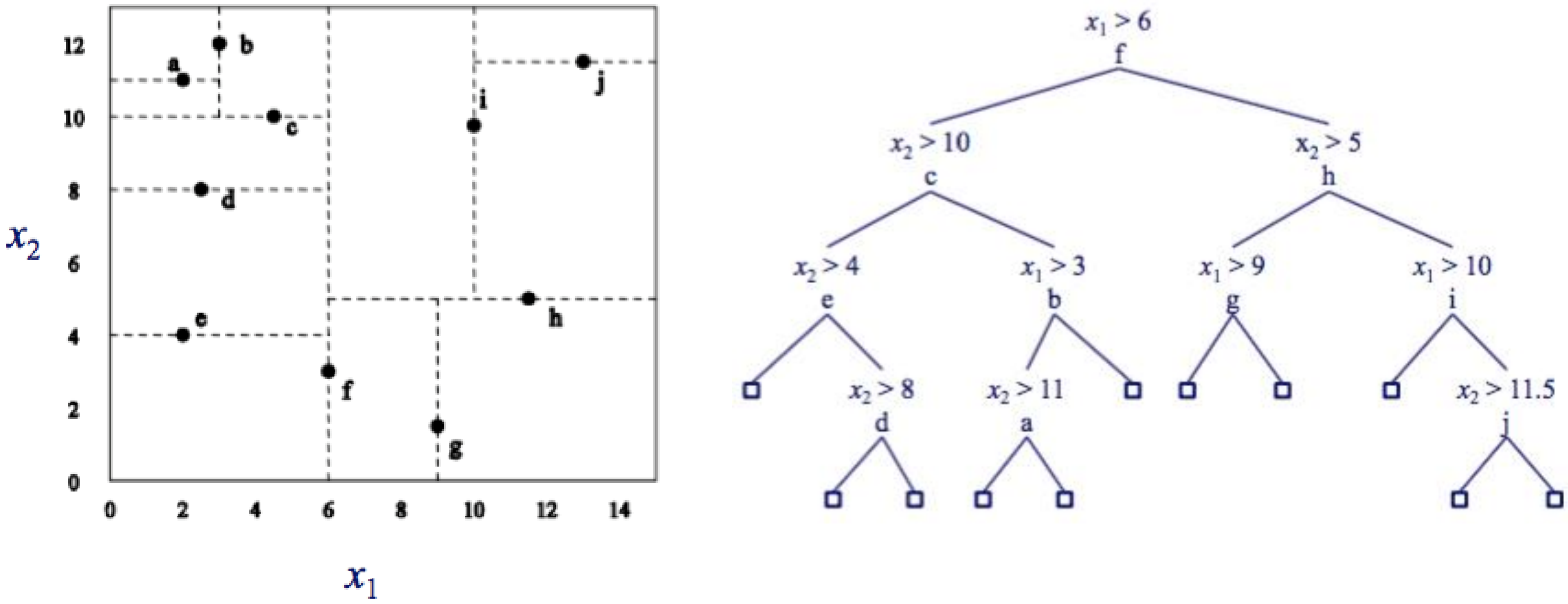
    start with an empty memory
    for each training instance $(x^{(i)}, y^{(i)})$
        if other training instances in memory **don't** correctly classify $(x^{(i)}, y^{(i)})$
            add it to the memory

Q1: Does ordering matter?

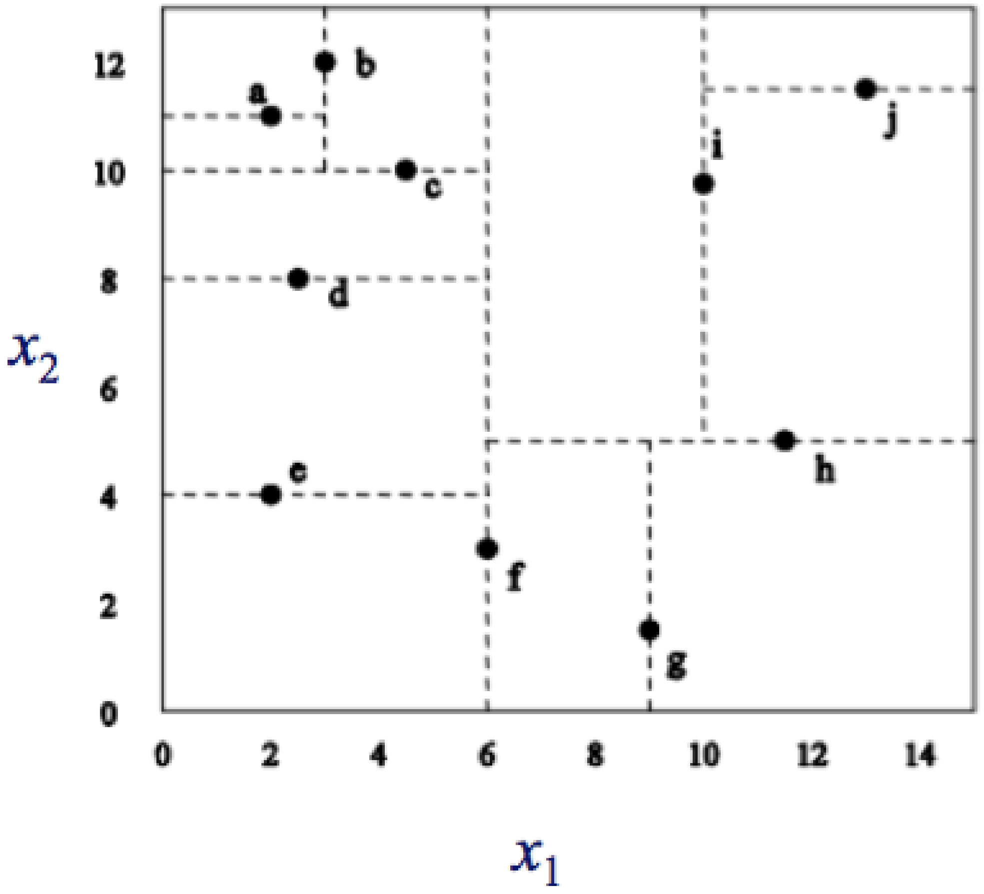
Q2: If following the optimal ordering, do the two approaches produce the same subset of instances?

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



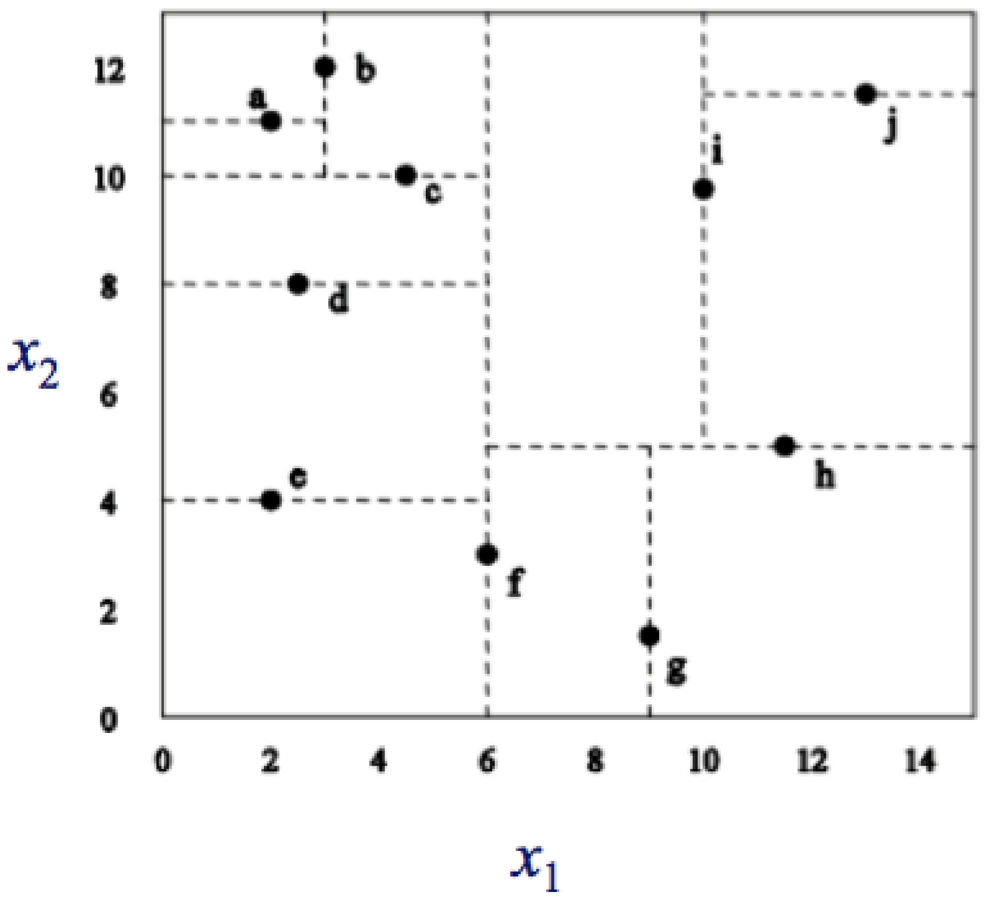
median value of the feature having the highest variance?

-- point f, $x_1 = 6$

$x_1>6$

f

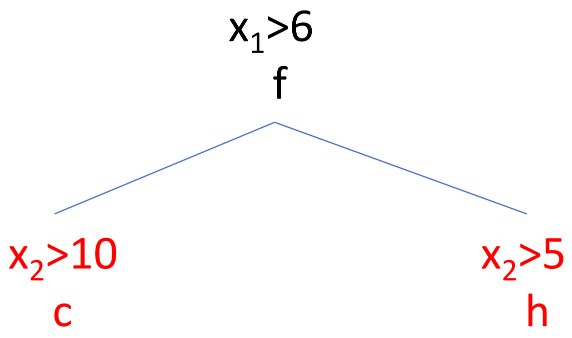
Construction of k-d tree



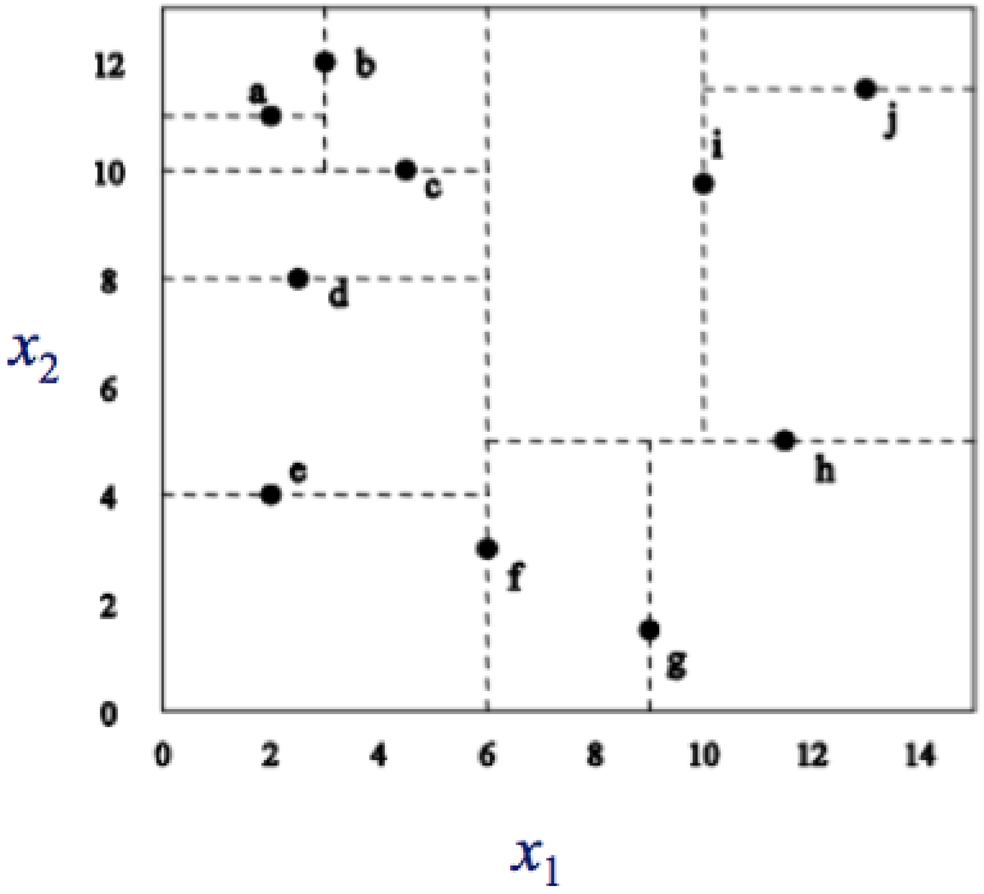
median value of the feature having the highest variance?

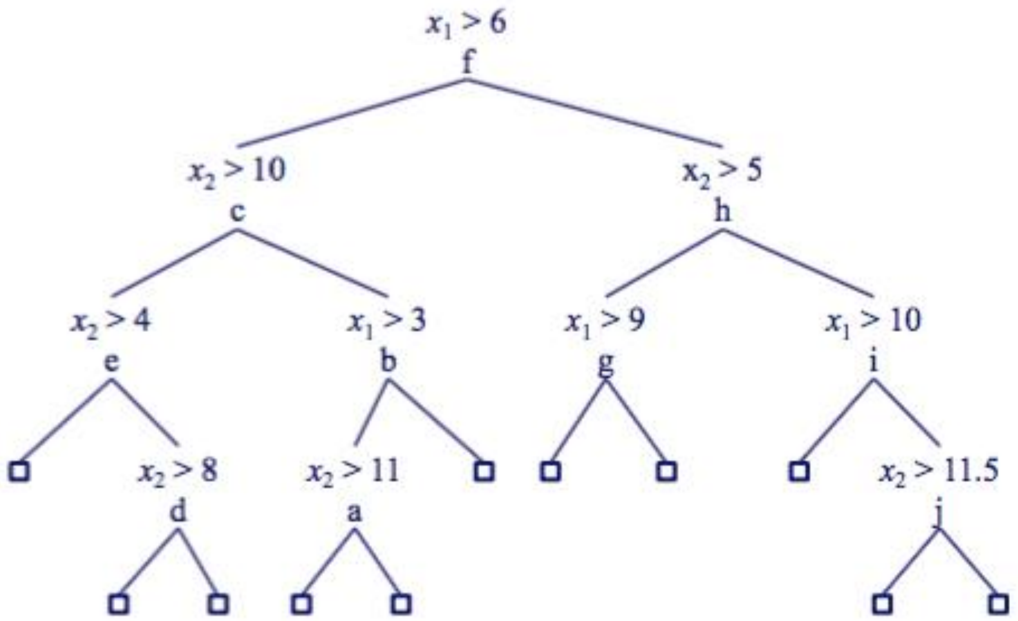
-- point f, x_1 = 6

-- 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

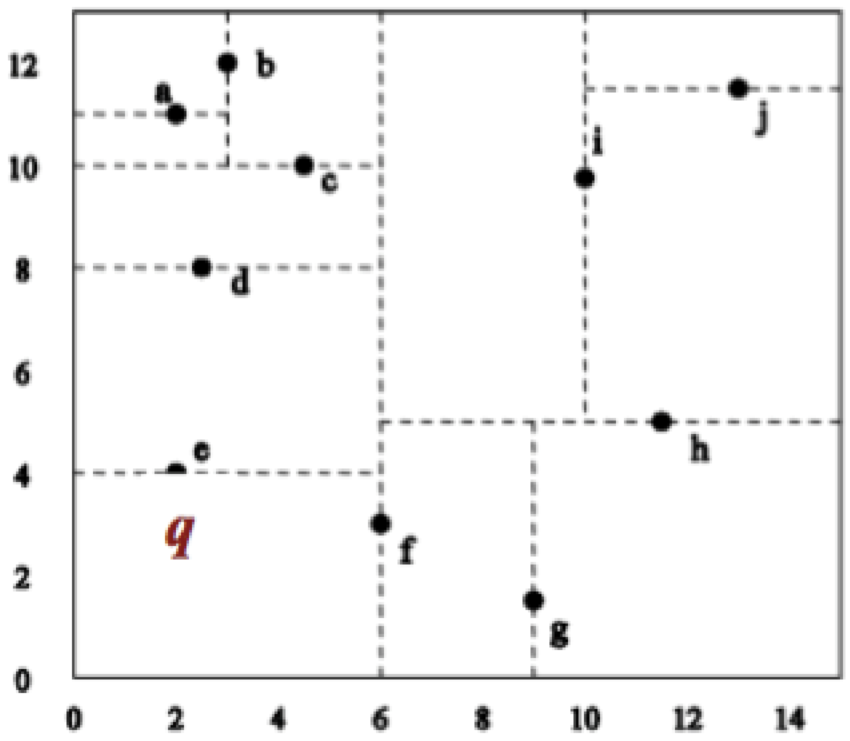
```
NearestNeighbor(instance x^(q))
    PQ = { }                                          // minimizing priority queue
    best_dist = ∞                                     // smallest distance seen so far
    PQ.push(root, 0)
    while PQ is not empty
        (node, bound) = PQ.pop();
        if (bound ≥ 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
```

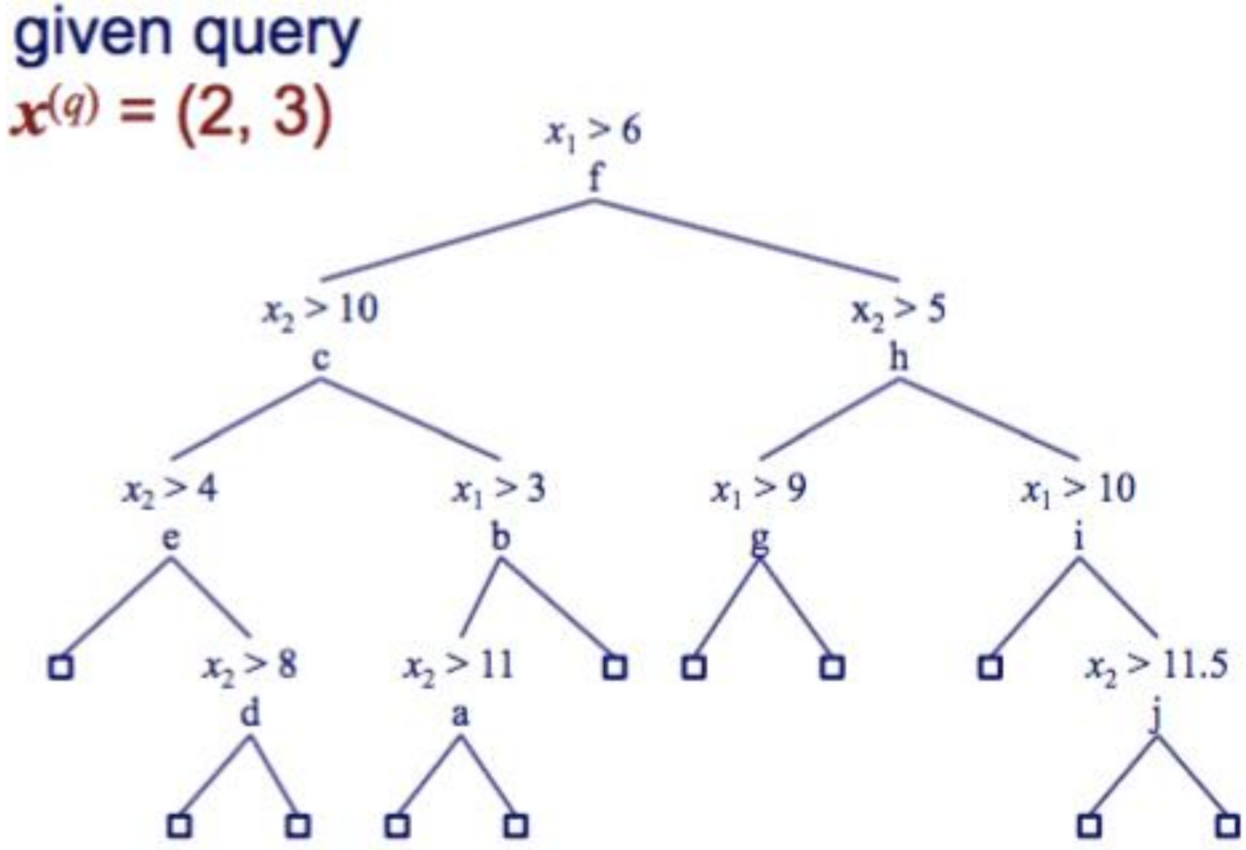
Intuitively, for a pair (*node*,*value*), *value* represents the smallest guaranteed distance, i.e., greatest lower bound up to now, from the instance $x^{(q)}$ to the set of instances over which *node* is the selected one to split

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

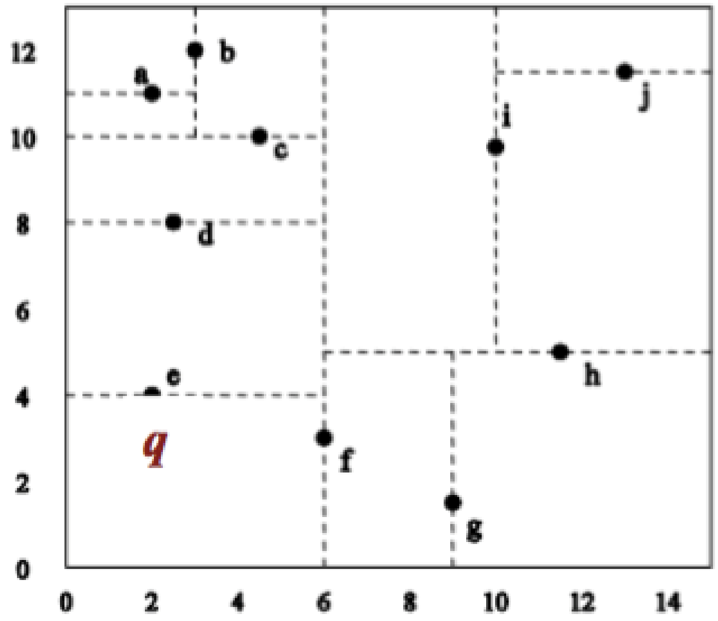
(root,0) means that at the beginning, the guaranteed smallest distance to the training set is 0

k-d tree example (Manhattan distance)



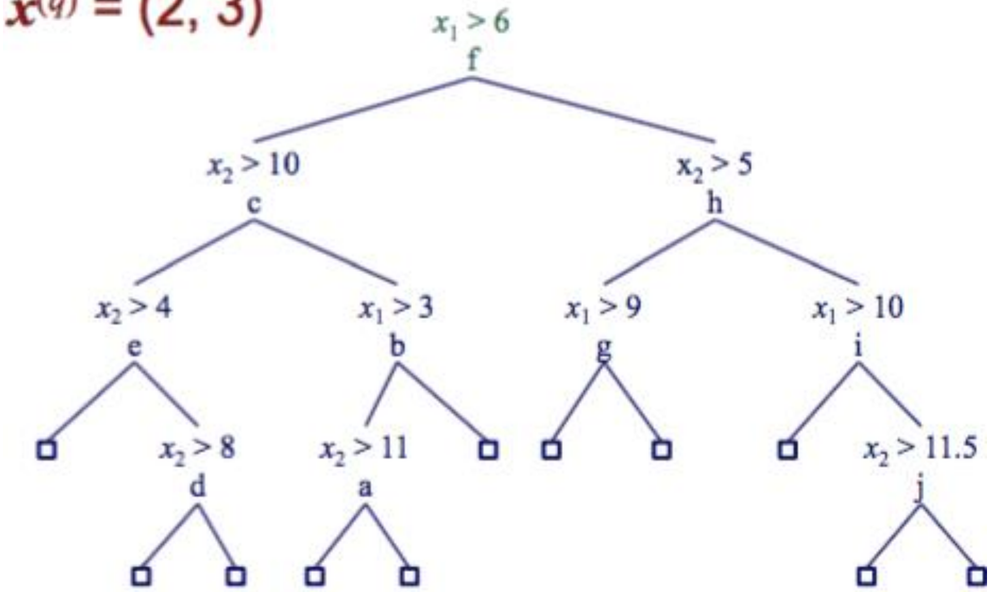


k-d tree example (Manhattan distance)



given query

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



(node, bound) = PQ.pop();

if (bound ≥ best_dist)

    return best_node.instance

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.featu

    PQ.push(node.right, 0)

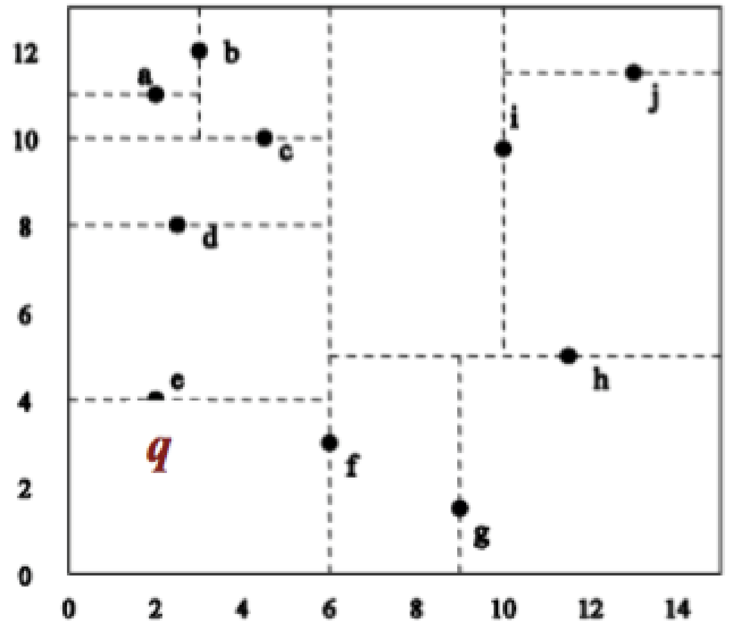
else

    PQ.push(node.left, 0)

    PQ.push(node.right, node. thresh

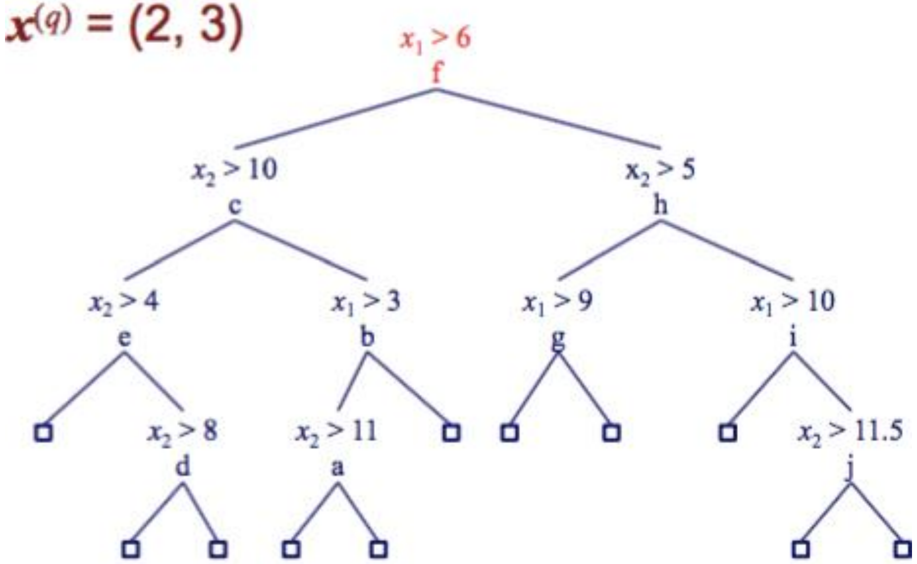
distance	best distance	best node	priority queue
	∞		(f, 0)

k-d tree example (Manhattan distance)



given query

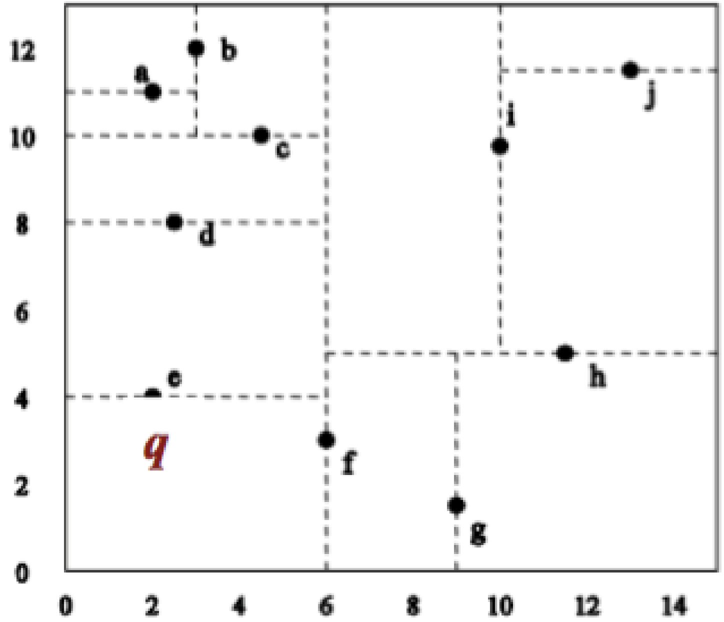
$x^{(q)} = (2, 3)$



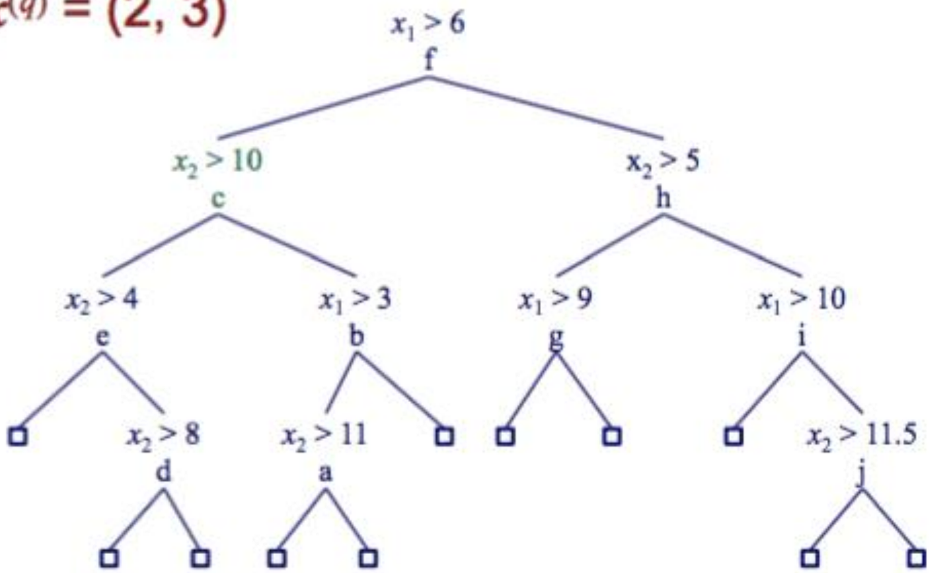

```
(node, bound) = PQ.pop();
if (bound ≥ best_dist)
        return best_node.instance
dist = distance(x(q), node. instance)
if (dist < best_dist)
        best_dist = dist
        best_node = node
if (q[node.feature] – node.threshold > 0)
        PQ.push(node.left, x(q)[node.featu
        PQ.push(node.right, 0)
else
        PQ.push(node.left, 0)
        PQ.push(node.right, node. thresh
```

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

k-d tree example (Manhattan distance)



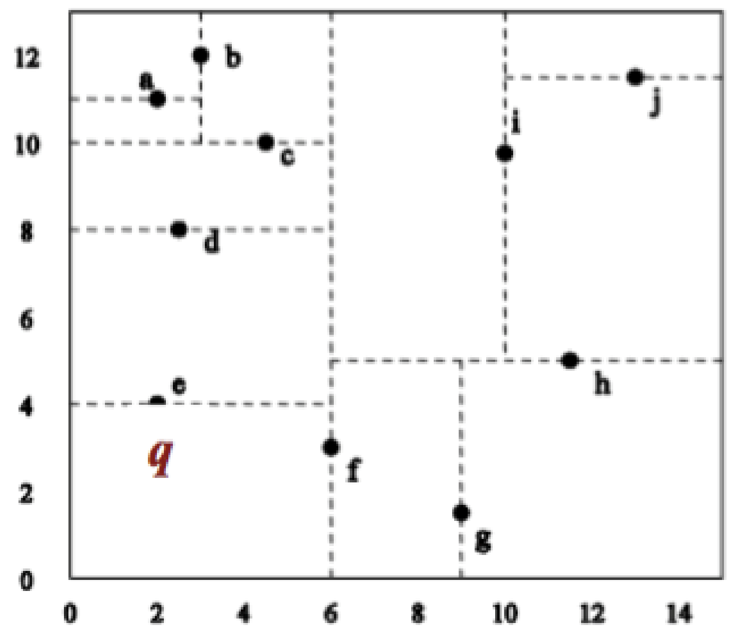
given query
$x^{(q)} = (2, 3)$



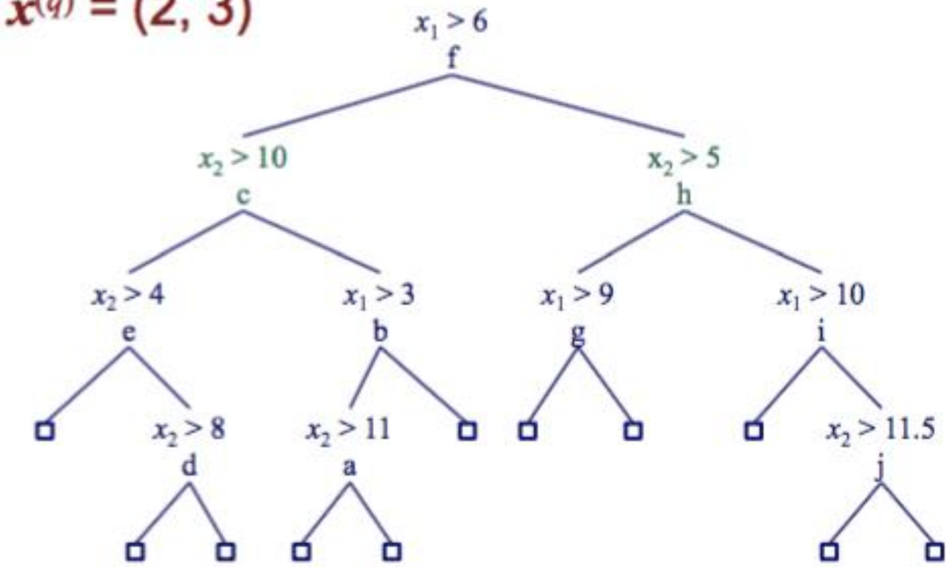
(node, bound) = PQ.pop();
if (bound ≥ best_dist)
    return best_node.instance
dist = distance($x^{(q)}$, node. instance)
if (dist < best_dist)
    best_dist = dist
    best_node = node
if (q[node.feature] – node.threshold > 0)
    PQ.push(node.left, $x^{(q)}$[node.featu
    PQ.push(node.right, 0)
else
    PQ.push(node.left, 0)
    PQ.push(node.right, node. thresh

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

k-d tree example (Manhattan distance)



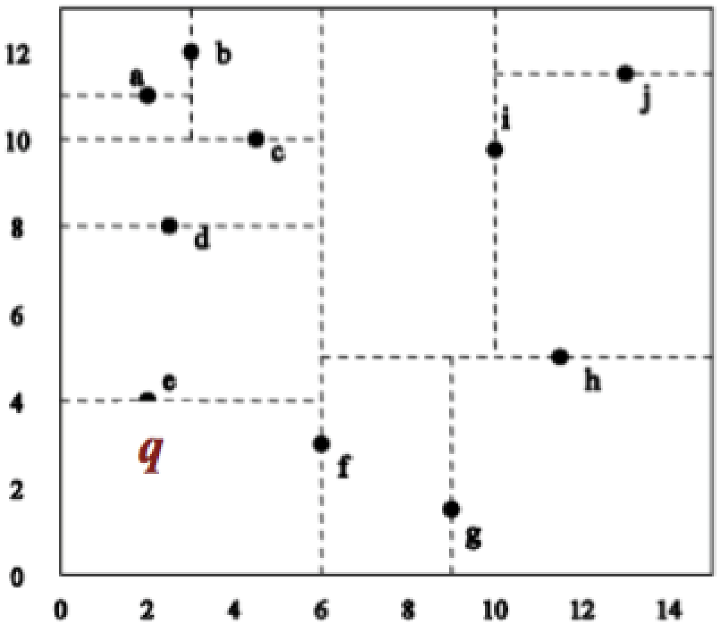
given query
$x^{(q)} = (2, 3)$




```
(node, bound) = PQ.pop();
if (bound ≥ best_dist)
        return best_node.instance
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.featu
        PQ.push(node.right, 0)
else
        PQ.push(node.left, 0)
    ➡   PQ.push(node.right, node. thresh
```

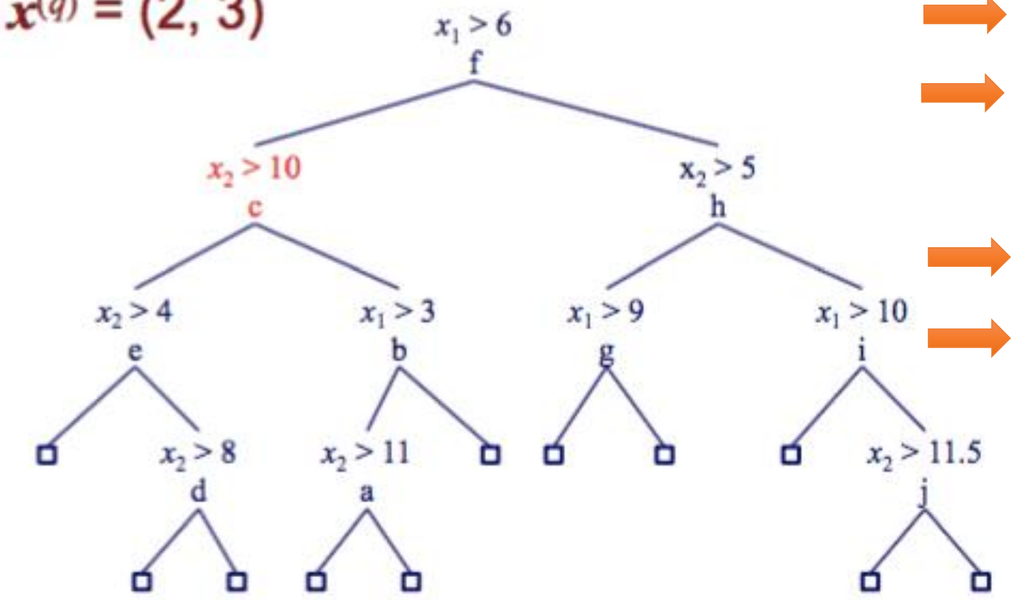
	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	(c, 0) (h, 4)

k-d tree example (Manhattan distance)



given query

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



(node, bound) = PQ.pop();

if (bound ≥ best_dist)

    return best_node.instance

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.featu

    PQ.push(node.right, 0)

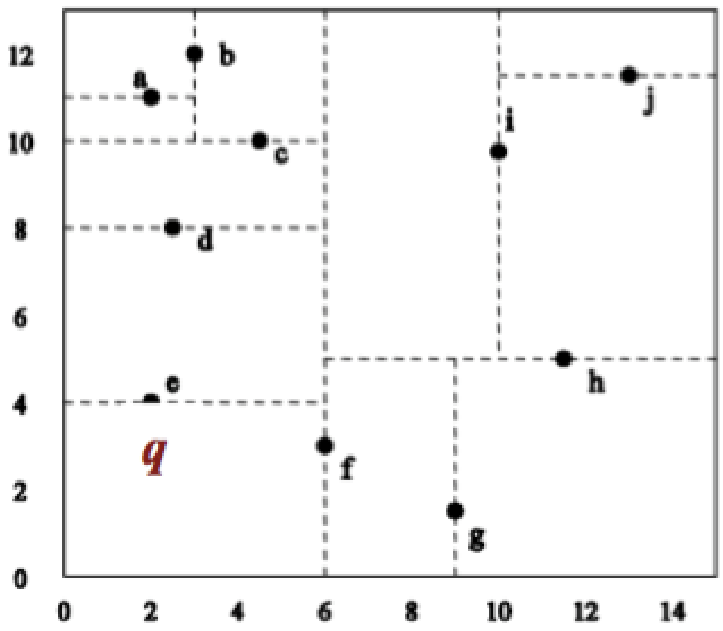
else

    PQ.push(node.left, 0)

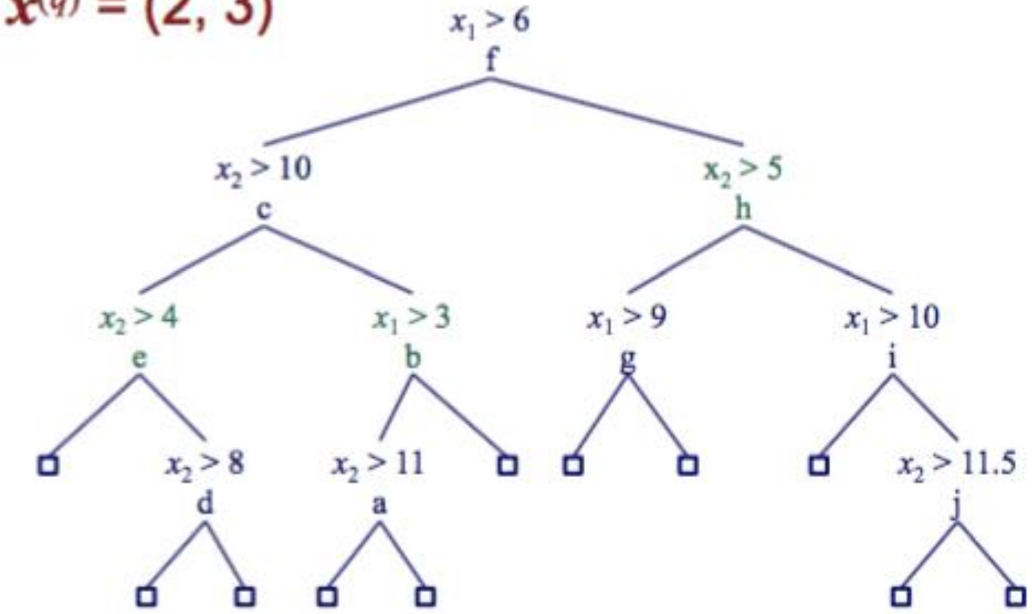
    PQ.push(node.right, node. thresh

	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)



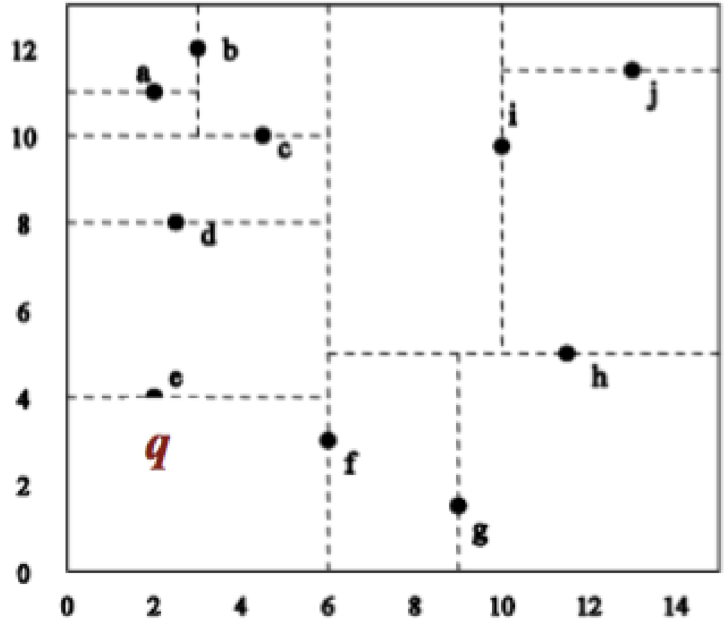
given query
$x^{(q)} = (2, 3)$



(node, bound) = PQ.pop();
if (bound ≥ best_dist)
    return best_node.instance
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.featu
    PQ.push(node.right, 0)
else
    PQ.push(node.left, 0)
    PQ.push(node.right, node. thresh

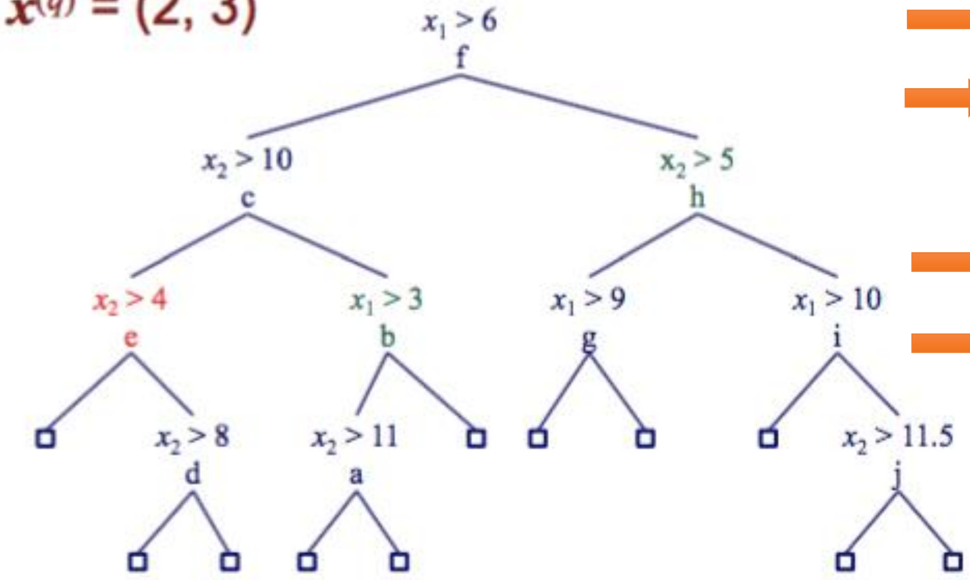
	distance	best distance	best node	priority queue
		∞		(f, 0)
pop f	4.0	4.0	f	(c, 0) (h, 4)
pop c	10.0	4.0	f	(e, 0) (h, 4) (b, 7)

k-d tree example (Manhattan distance)



given query

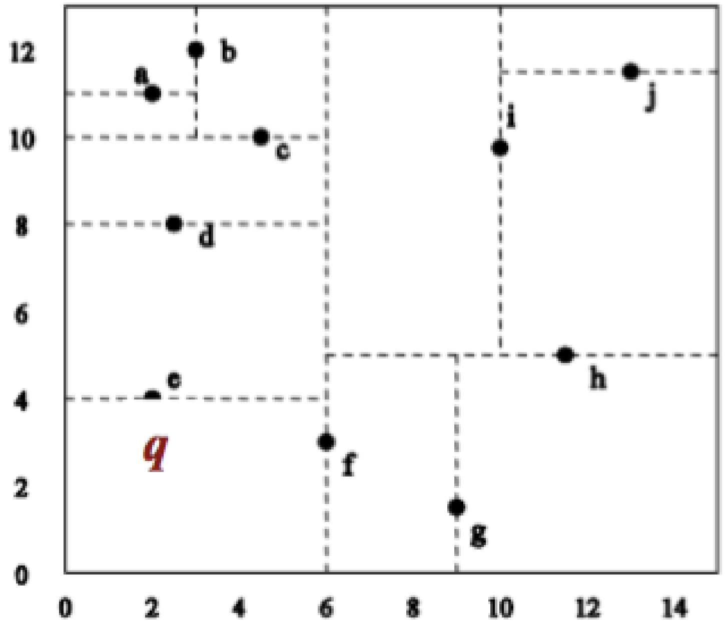
$x^{(q)} = (2, 3)$



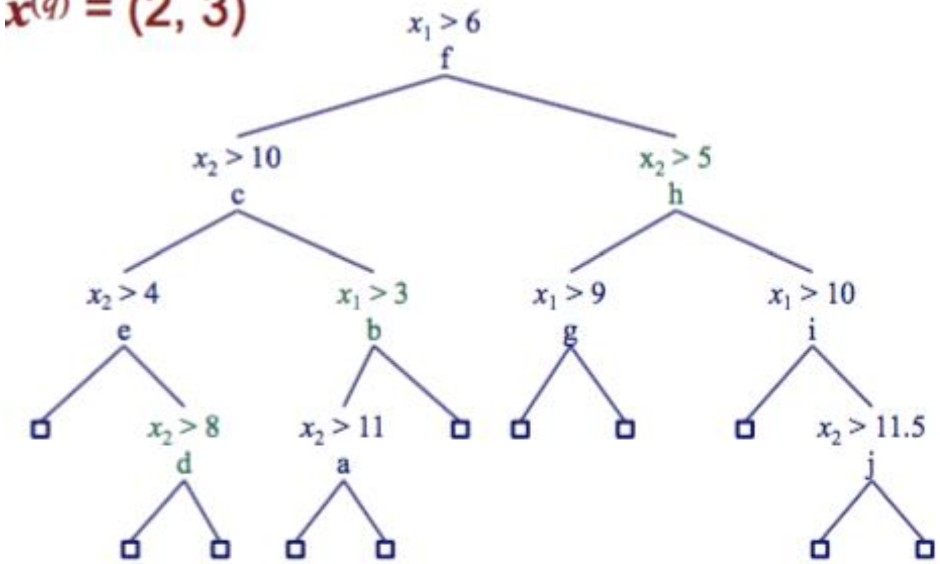

```
(node, bound) = PQ.pop();
if (bound ≥ best_dist)
        return best_node.instance
dist = distance(x^(q), node. instance)
if (dist < best_dist)
        best_dist = dist
        best_node = node
if (q[node.feature] – node.threshold > 0)
        PQ.push(node.left, x^(q)[node.featu
        PQ.push(node.right, 0)
else
        PQ.push(node.left, 0)
        PQ.push(node.right, node. thresh
```

	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)



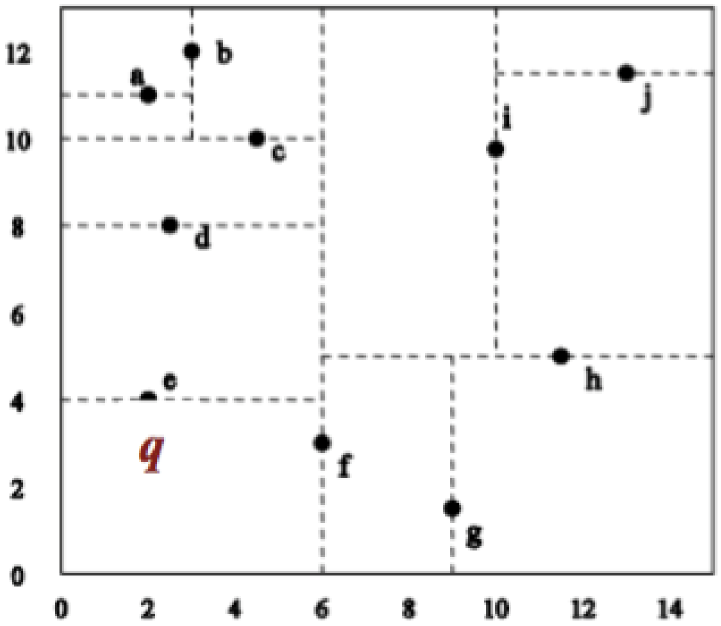
given query
$x^{(q)} = (2, 3)$



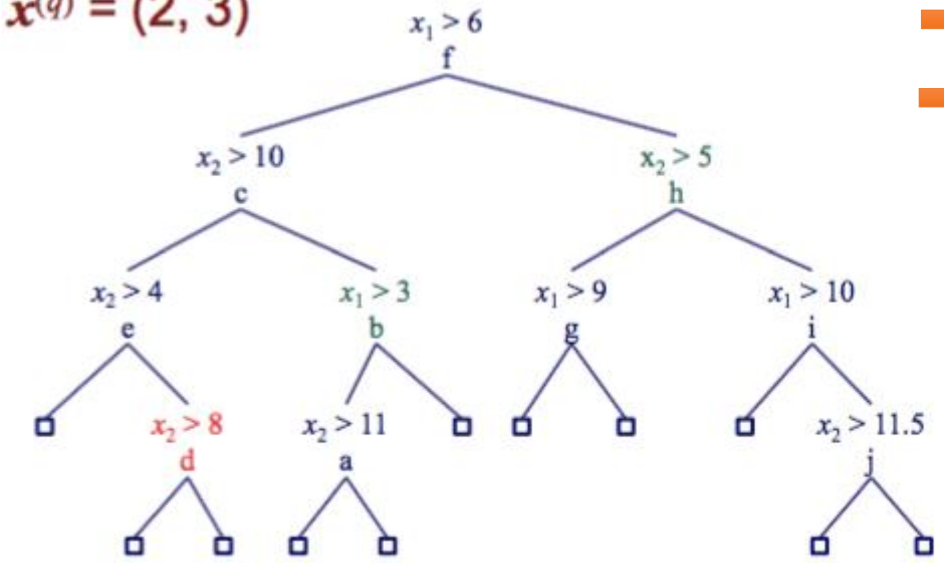
(node, bound) = PQ.pop();
if (bound ≥ best_dist)
    return best_node.instance
dist = distance($x^{(q)}$, node. instance)
if (dist < best_dist)
    best_dist = dist
    best_node = node
if (q[node.feature] – node.threshold > 0)
    PQ.push(node.left, $x^{(q)}$[node.featu
    PQ.push(node.right, 0)
else
    PQ.push(node.left, 0)
    PQ.push(node.right, node. thresh

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



(node, bound) = PQ.pop();
if (bound ≥ best_dist)
    return best_node.instance
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.featu
    PQ.push(node.right, 0)
else
    PQ.push(node.left, 0)
    PQ.push(node.right, node. thresh

	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

Variants of k-NN

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:

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

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

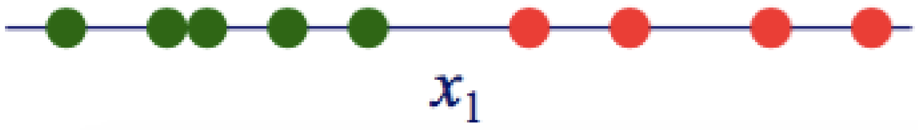
- regression:

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

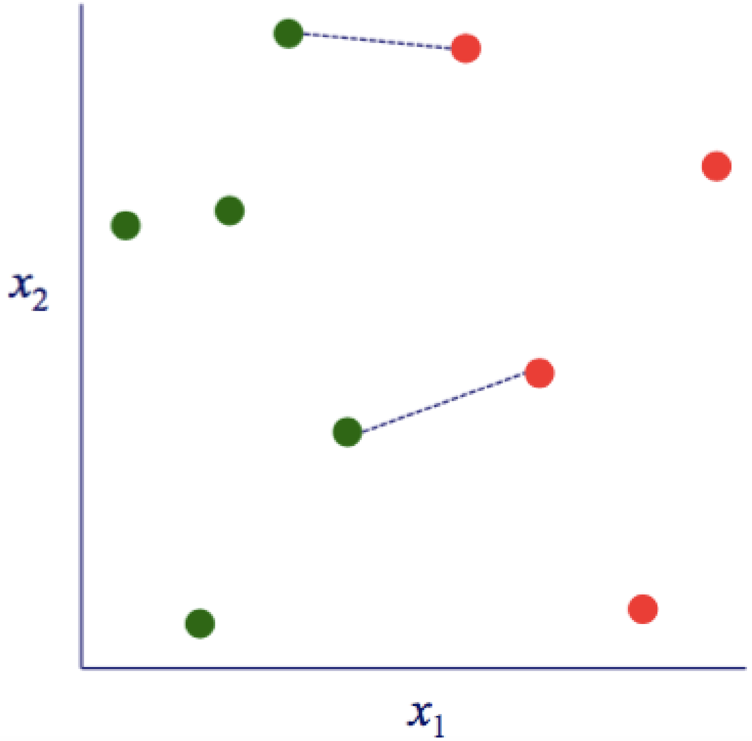
reciprocal of the distance

Irrelevant features in instance-based learning

here's a case in which there is one relevant feature x_1 and a 1-NN rule classifies each instance correctly



consider the effect of an irrelevant feature x_2 on distances and nearest neighbors



Can you find a point (a,b) which is red, if classified only according to feature x1, but is green, if classified according to both features?

Locally weighted regression

- one way around this limitation is to weight features differently
- *locally weighted regression* is one nearest-neighbor variant that does this

- prediction task
  - **given**: an instance $x^{(q)}$ to make a prediction for
  - 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 $f(x^{(q)})$

What's function f ?

Locally weighted regression

- Determining function f
  - Assume that f is a linear function over the features, i.e.,

$$f(x^{(i)}) = w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + \ldots + w_n x_n^{(i)}$$

  - find the weights w_i for each $x^{(q)}$ by

$$\arg \min_{w_0, w_1, \ldots, x_n} \sum_{i=1}^{k} (f(x^{(i)}) - y^{(i)})^2$$

can do this using gradient descent (to be covered soon)

  - After obtaining weights, for $x^{(q)}$, we have $f(\mathbf{x}^{(q)}) = w_0 + w_1 x_1^{(q)} + w_2 x_2^{(q)} + \ldots + w_n x_n^{(q)}$

Discussions

Strengths of instance-based learning

- simple to implement
- “training” is very efficient
- adapts well to on-line learning
- robust to noisy training data (when $k > 1$)
- often works well in practice

Limitations of instance-based learning

- sensitive to range of feature values
- sensitive to irrelevant and correlated features, although ...
  - there are variants (such as locally weighted regression) that learn weights for different features
- classification/prediction can be inefficient, although …
  - edited methods and k-d trees can help alleviate this weakness
- doesn’t provide much insight into problem domain because there is no explicit model

Inductive bias

- *inductive bias* is the set of assumptions a learner uses to be able to predict y_i for a previously unseen instance x_i
- two components
    - *hypothesis space bias*: determines the models that can be represented
    - *preference bias*: specifies a preference ordering within the space of models
- in order to *generalize* (i.e. make predictions for previously unseen instances) a learning algorithm must have an inductive bias

Consider the inductive bias of DT and k-NN learners

learner	hypothesis space bias	preference bias
ID3 decision tree	trees with single-feature, axis-parallel splits	small trees identified by greedy search
k-NN	Voronoi decomposition determined by nearest neighbors	instances in neighborhood belong to same class