CSCI 347, Data Mining

Instance Based Learning

Instance-Based Learning

- No structure is learned
- Given an instance to predict, simply predict the class of its nearest neighbor
- Alternatively, predict the class which appears most frequently for the nearest k neighbors

Example

Relation: weather.symbolic

No.	outlook Nominal	temperature Nominal	humidity Nominal	windy Nominal	play Nominal
1	sunny	hot	high	FALSE	no
2	sunny	hot	high	TRUE	no
3	overcast	hot	high	FALSE	yes
4	rainy	mild	high	FALSE	yes
5	rainy	cool	normal	FALSE	yes
6	rainy	cool	normal	TRUE	no
7	overcast	cool	normal	TRUE	yes
8	sunny	mild	high	FALSE	no
9	sunny	cool	normal	FALSE	yes
10	rainy	mild	normal	FALSE	yes
11	sunny	mild	normal	TRUE	yes
12	overcast	mild	high	TRUE	yes
13	overcast	hot	normal	FALSE	yes
14	rainy	mild	high	TRUE	no

Predict the class value of the following:

Outlook	Temp	Humidity	Windy	Play
rainy	hot	normal	false	?

Example

Relation: weather

No.	outlook Nominal	temperature Numeric	humidity Numeric	windy Nominal	play Nominal
1	sunny	85.0	85.0	FALSE	no
2	sunny	80.0	90.0	TRUE	no
3	overcast	83.0	86.0	FALSE	yes
4	rainy	70.0	96.0	FALSE	yes
5	rainy	68.0	80.0	FALSE	yes
6	rainy	65.0	70.0	TRUE	no
7	overcast	64.0	65.0	TRUE	yes
8	sunny	72.0	95.0	FALSE	no
9	sunny	69.0	70.0	FALSE	yes
10	rainy	75.0	80.0	FALSE	yes
11	sunny	75.0	70.0	TRUE	yes
12	overcast	72.0	90.0	TRUE	yes
13	overcast	81.0	75.0	FALSE	yes
14	rainy	71.0	91.0	TRUE	no

Predict the class value of the following:

Outlook	Temp	Humidity	Windy	Play
rainy	89.0	65.0	false	?

Recall Linear Regression

Goal: Choose the weights w_0, …, w_n to minimize the sum of the squares of the differences between the actual and predicted class values. That is, minimize:

$$\sum_{j=1}^{m}\left(x^{(j)} - \sum_{i=0}^{n} w_i * a_i^{(j)}\right)^2$$

where m is the number of instances in the dataset, n is the number of attributes, $x^{(j)}$ is the actual value of the jth instance, w_i is the weight of the ith attribute (except w_0 is the bias) and $a_i^{(j)}$ is the value of the ith attribute in the jth instance.

Euclidean Distance

Ordinary distance which one would measure with a ruler

In two dimensions:

For distance between

$p=(p_1, p_2)$ and $q=(q_1,q_2)$

$$\sqrt{(p_1 - p_2)^2 + (q_1 - q_2)^2}$$

Uses the Pythagorean Theorem

Euclidean Distance

In n dimensions:

For distance between

$p=(p_1, p_2, \ldots, p_n)$ and $q=(q_1,q_2, \ldots, q_n)$

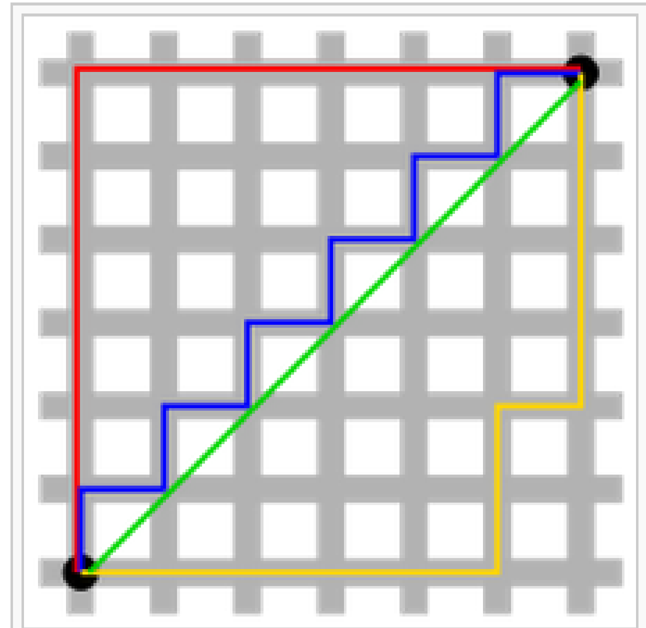
$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}$$

Manhattan Distance

In two dimensions:

if $p=(p_1, p_2)$ and $q=(q_1,q_2)$

$$|p_1 - q_1| + |p_2 - q_2|$$



Taxicab geometry versus Euclidean distance: In taxicab geometry all three pictured lines (red, yellow, and blue) have the same length (12) for the same route. In Euclidean geometry, the green line has length $6\sqrt{2} \approx 8.49$, and is the unique shortest path.

Normalization

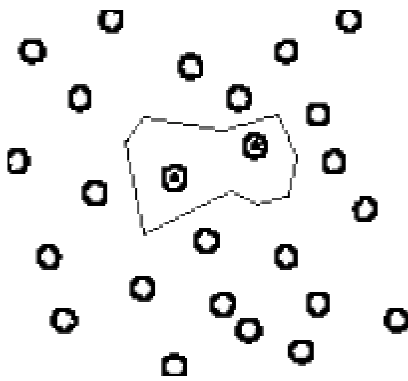
Attribute values are likely to have different ranges causing the difference for an attribute with a large range to dominate the difference for an attribute with a small range.

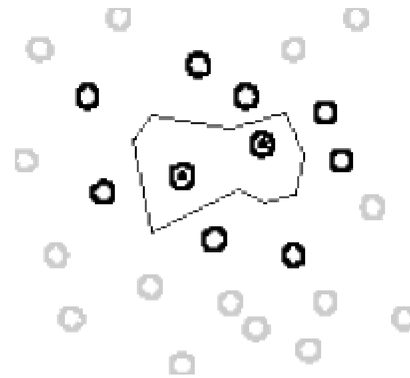
Solution: work with normalized values rather than actual value

$$\text{Normalized value} = \frac{\text{value} - \text{min_attribute_value}}{\text{max_attribute_value} - \text{min_attribute_value}}$$

Learning prototypes

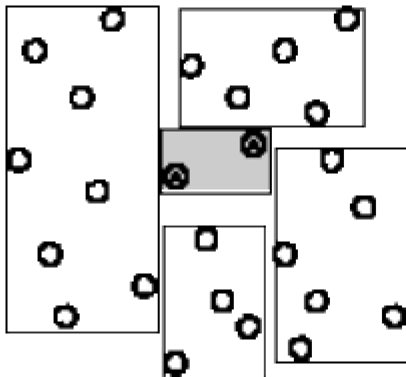
- Only those instances involved in a decision need to be stored
- Noisy instances should be filtered out
- Idea: only use *prototypical* examples

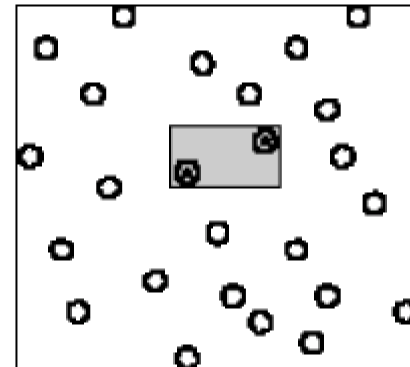




Rectangular Generalizations

- Nearest-neighbor rule is used outside rectangles
- Rectangles are rules! (But they can be more conservative than “normal” rules.)
- Nested rectangles are rules with exceptions





Finding Nearest neighbors Efficiently

Nearest neighbor algorithm can be done more efficiently using appropriate data structures.

Two methods that represent training data in a tree structure:

- kD-trees
- ball trees

*k*D-Tree Example

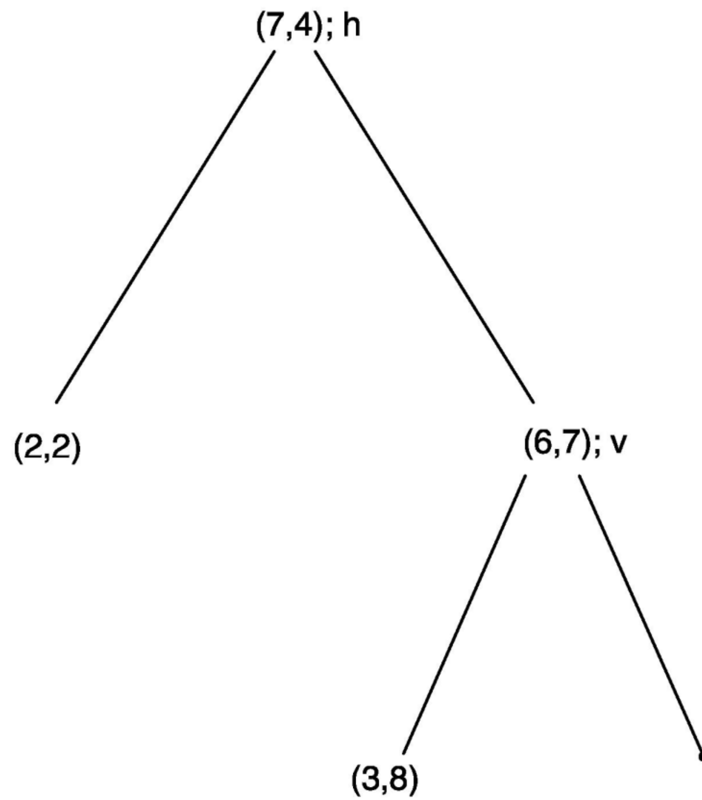
k – number of attributes (don't include class value)

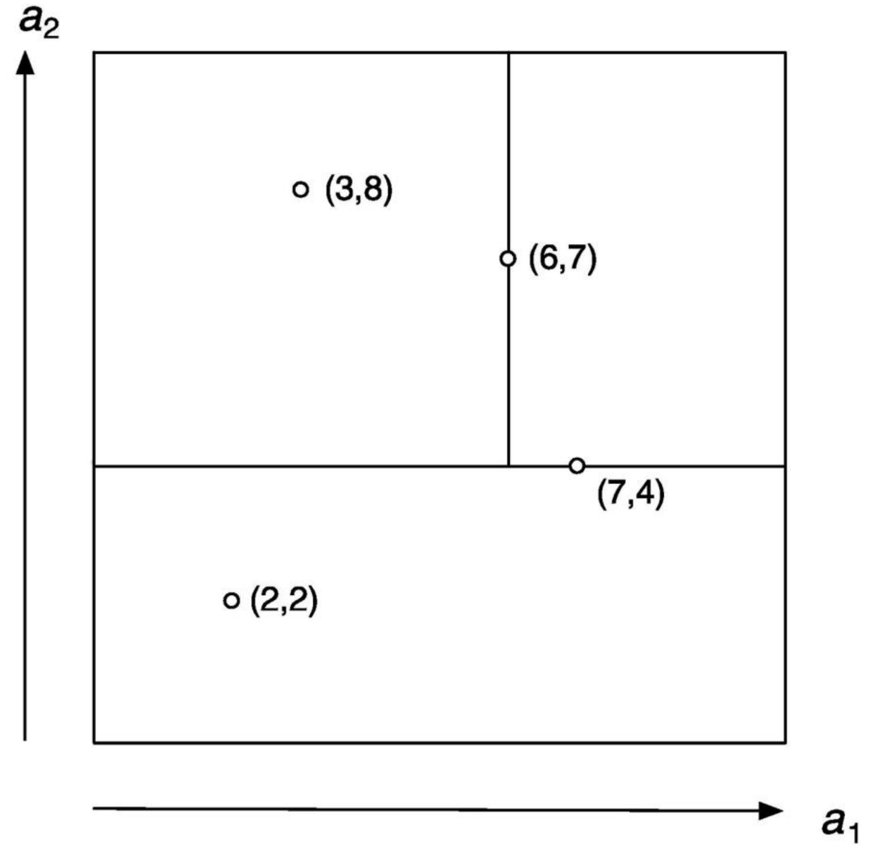
Build a tree containing all instances, to reduce the search space.

*k*D-Tree Algorithm

1. Find an attribute with the largest variance.
2. Split on an instance with the median value for that attribute. This is the root of the tree.
3. Repeat process recursively. Hope to split in different directions to make the tree "square-ish", not a skinny rectangle.

*k*D-Tree Example





Find Nearest Neighbor using a kD Tree, Step 1

Given a new instance (star in the figure):

1. Start at root, branching according to attribute values to locate the region containing the target (black instance in figure).

This is not necessarily the closest neighbor, but is a good first approximation. Any nearest neighbor must lie within the dashed circle.

Example – Find Nearest Neighbor using kD Tree

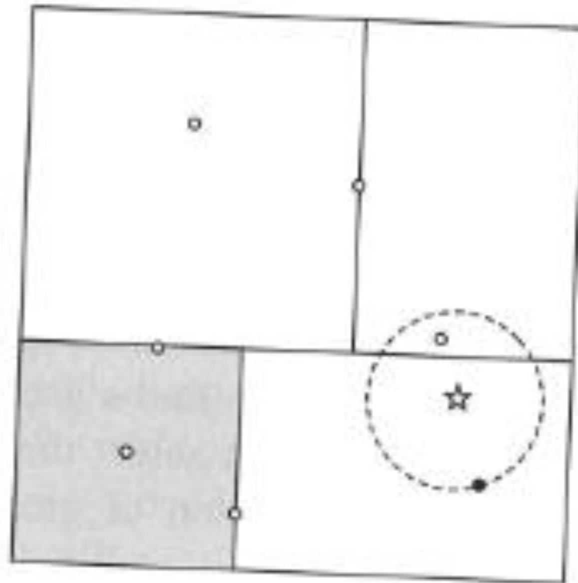


FIGURE 4.14

Using a *k*D-tree to find the nearest neighbor of the star.

Find Nearest Neighbor using a kD Tree, Step 2

2. To determine if a nearer neighbor exists, check the note’s siblings (shaded figure), back up to the parent node and check it’s siblings. May need to descend it to see if a closer neighbor exists (as it does in the figure).

More on *k*D-Trees

Complexity depends on depth of tree, given by logarithm of number of nodes

Amount of backtracking required depends on quality of tree (“square” vs. “skinny” nodes)

More on *k*D-Trees

If data is skewed, might be better to use the value closest to the mean, instead of the median

Ball Trees

kD-trees have many corners. Avoid corners by using hyperspheres, balls, rather than hyperrectangles.

Overlapping regions are not a problem

Ball Tree Example