Data Mining

K Nearest Neighbor Classification

Instance-based learning

- Distance function defines what's learned
- Most instance-based schemes use *Euclidean distance*:

$$\sqrt{(a_1^{(1)} - a_1^{(2)})^2 + (a_2^{(1)} - a_2^{(2)})^2 + \ldots (a_k^{(1)} - a_k^{(2)})^2}$$

  $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(2)}$: two instances with *k* attributes
- Taking the square root is not required when comparing distances
- Other popular metric: *city-block metric*
  - Adds differences without squaring them

Normalization and other issues

- Different attributes are measured on different scales ⇒ need to be *normalized*:

$$a_i = \frac{v_i - \min v_i}{\max v_i - \min v_i} \qquad a_i = \frac{v_i - u_i}{\sigma_i}$$

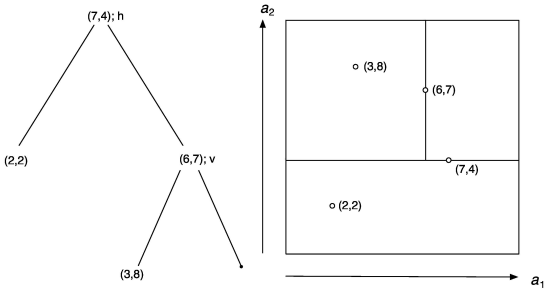
  v_i: the actual value of attribute *i*, μ_i and σ_i are the mean and standard deviation along attribute i
- Nominal attributes: distance either 0 or 1
- Common policy for missing values: assumed to be maximally distant (given normalized attributes)

Finding nearest neighbors efficiently

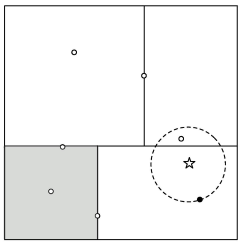
- Simplest way of finding nearest neighbor: linear scan of the data
  - Classification takes time proportional to the product of the number of instances in training and test sets
- Nearest-neighbor search can be done more efficiently using appropriate data structures
- Two methods that represent training data in a tree structure:

kD-trees and *ball trees*

*k*D-tree example



Using kD-trees: example



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)
- How to build a good tree? Need to find good split point and split direction
  - Split direction: direction with greatest variance
  - Split point: median value along that direction
- Using value closest to mean (rather than median) can be better if data is skewed
- Can apply this recursively

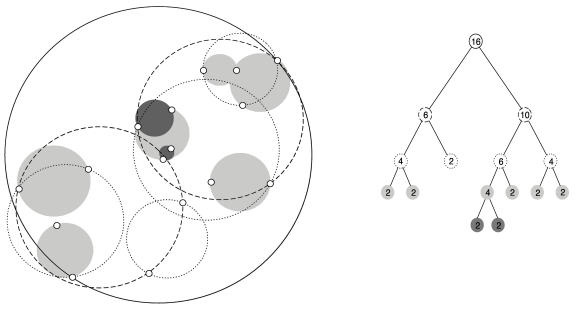
Building trees incrementally

- Big advantage of instance-based learning: classifier can be updated incrementally
  - Just add new training instance!
- Can we do the same with *k*D-trees?
- Heuristic strategy:
  - Find leaf node containing new instance
  - Place instance into leaf if leaf is empty
  - Otherwise, split leaf according to the longest dimension (to preserve squareness)
- Tree should be re-built occasionally (i.e. if depth grows to twice the optimum depth)

Ball trees

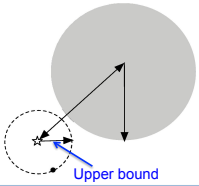
- Problem in *k*D-trees: corners
- Observation: no need to make sure that regions don't overlap
- Can use balls (hyperspheres) instead of hyperrectangles
  - A *ball tree* organizes the data into a tree of *k*-dimensional hyperspheres
  - Normally allows for a better fit to the data and thus more efficient search

Ball tree example



Using ball trees

- Nearest-neighbor search is done using the same backtracking strategy as in *k*D-trees
- Ball can be ruled out from consideration if: distance from target to ball's center exceeds ball's radius plus current upper bound



Building ball trees

- Ball trees are built top down (like *k*D-trees)
- Don't have to continue until leaf balls contain just two points: can enforce minimum occupancy (same in *k*D-trees)
- Basic problem: splitting a ball into two
- Simple (linear-time) split selection strategy:
  - Choose point farthest from ball's center
  - Choose second point farthest from first one
  - Assign each point to these two points
  - Compute cluster centers and radii based on the two subsets to get two balls

Discussion of nearest-neighbor learning

- Often very accurate
- Assumes all attributes are equally important
  - Remedy: attribute selection or weights
- Possible remedies against noisy instances:
  - Take a majority vote over the *k* nearest neighbors
  - Removing noisy instances from dataset (difficult!)
- Statisticians have used *k*-NN since early 1950s
  - If $n \rightarrow \infty$ and $k/n \rightarrow 0$, error approaches minimum
- *k*D-trees become inefficient when number of attributes is too large (approximately > 10)
- Ball trees (which are instances of *metric trees*) work well in higher-dimensional spaces

More discussion

- Instead of storing all training instances, compress them into regions
- One simple technique (Voting Feature Intervals):
  - Construct intervals for each attribute
    - Discretize numeric attributes
    - Treat each value of a nominal attribute as an "interval"
  - Count number of times class occurs in interval
  - Prediction is generated by letting intervals vote (those that contain the test instance)