Instance-based Learning or Nearest Neighbor

J. Savoy
Université de Neuchâtel

Ian H. Witten, Eibe Frank: Data Mining. Practical Machine Learning Tools and Techniques. Morgan Kaufmann, 2005.
D. Ullman: Course on Data Mining.

Overview

- **The Main Idea**
- Distance Measures
- Efficient Computation
- Nearest Shrunken Centroids

Instance-based Representation

- Simplest form of learning: *rote learning* (plain memorization)
  - Training instances are searched for existing instances that most closely *resembles* new example
  - The predicted class is given by the nearest training example
  - The instances themselves represent the knowledge
  - Similarity function defines what's "learned"

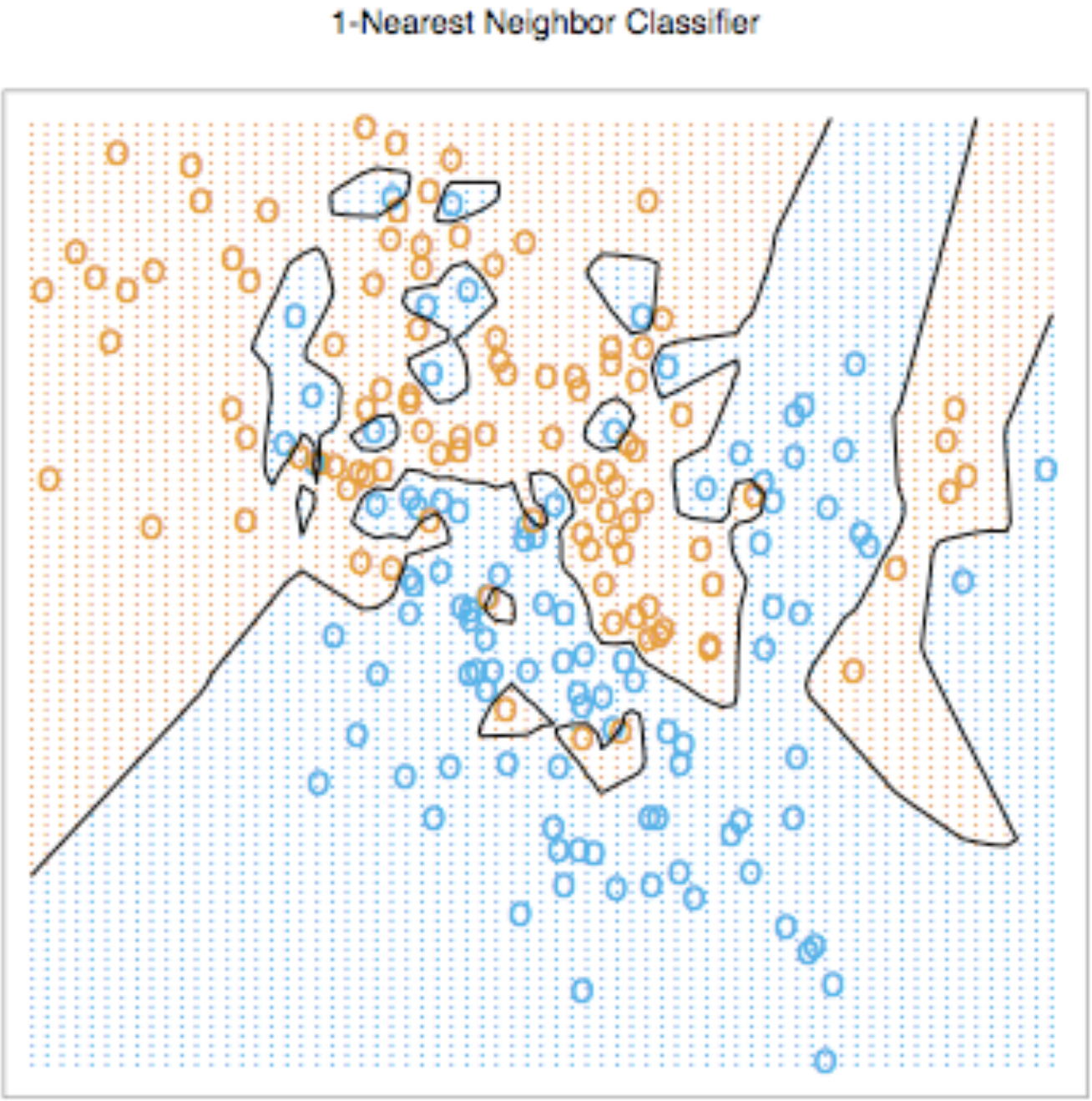
Instance-based Representation

- Instance-based learning is *lazy* learning
  (we don't spend time to learn)
- Methods: *nearest neighbor* (NN)*, k-nearest neighbor* (k-NN)
  with *k* = 1, 3, … odd number
  We suggest having a rather “large” value for *k*
  (when having a lot of training examples)
- Given a *k* value, we can determine regions belonging to the same class
- These regions are not rectangular (decision tree) but may have a general form (depending of the value of *k*)

Instance-ba

Binary Response

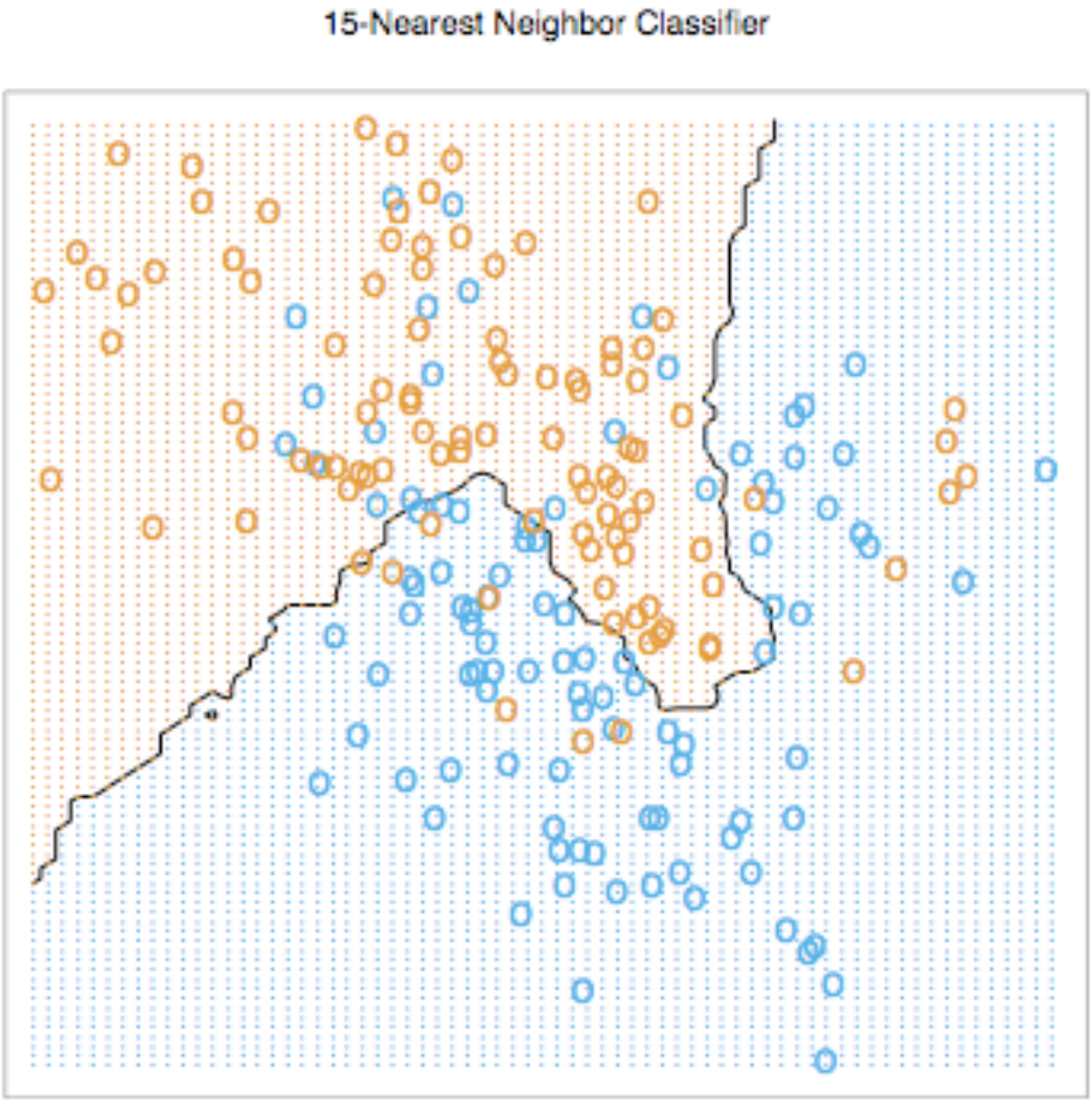
1-Nearest Neighbor



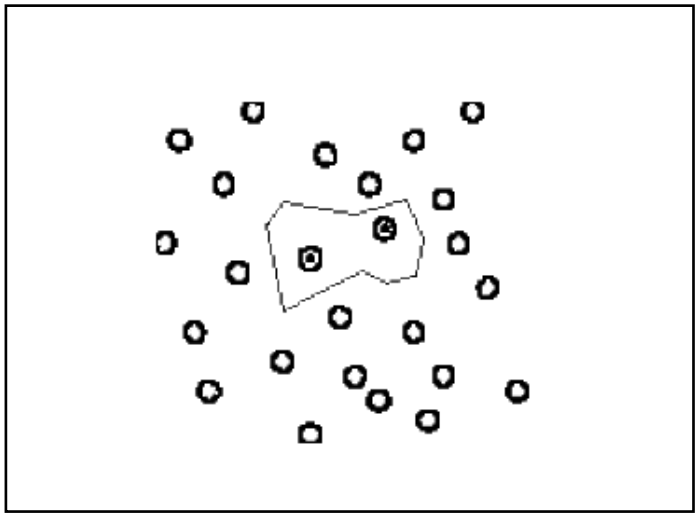
Instance-based Representation

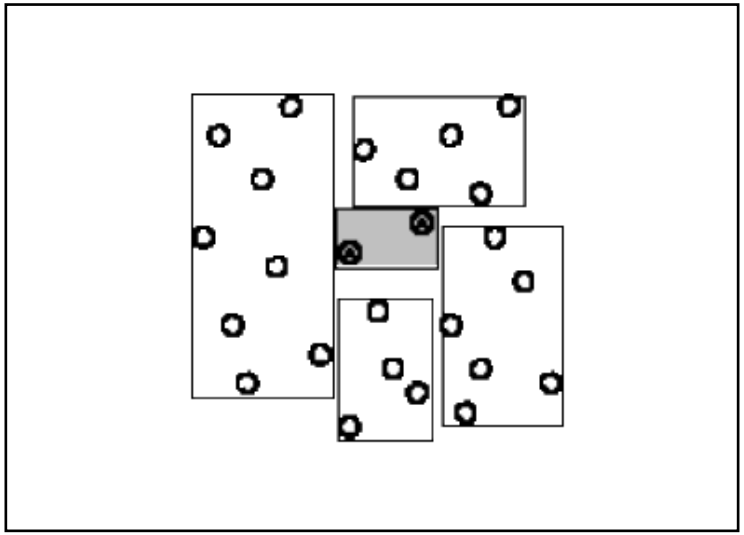
Binary Response

15-Nearest Neighbors



Learning prototypes





- Better learnt representation
  Inside boundaries, all instances belong to the same decision. Thus only those instances involved in a decision limit need to be stored (near of the boundaries)
- Noisy instances should be filtered out
- Idea: only use *prototypical* examples (3 times the same instances is not useful)
- But the main problem is to measure the distance …

Simple Distance Function

- Simplest case: one numeric attribute
  - Distance is the difference between the two attribute values involved (or a function thereof)

Feature \ Object	A	B	C	D
Feature 1	40	36	60	50

- Distance (A,B) = 40 - 36 = 4
  Distance (B,A) = 36 - 40 = -4
  Use the absolute value (L_1-norm)

Simple Distance Function

- Several numeric attributes: normally, Euclidean distance is used

Feature \ Objects	A	B	C	D
Feature 1	40	36	60	50
Feature 2	49	52	38	38

$$\begin{aligned} Distance(A, B) &= \sqrt{(40-36)^2 + (49-52)^2} \\ &= \sqrt{4^2 + 3^2} = \sqrt{16+9} = 5 \end{aligned}$$

- Computing the square root is not really necessary

Simple Distance Function

- Other example

Feature \ Objects	A	B	C	D
Weight	40	36	60	50
Value	4,900	5,200	3,800	3,700

$$\begin{aligned} Distance(A, B) &= (40 - 36)^2 + (4,900 - 5,200)^2 \\ &= 4^2 + 300^2 = 16 + 90,000 \end{aligned}$$

- Attribute values must be normalized!

Simple Distance Function

- To normalize a numeric attribute, we can normalize the values between 0 and 1 using the following rule. For an attribute with value v_k, we may compute its new normalizes value v'_k as:

$$v'_k = \frac{v_k - min\ (v_j)}{max\ (v_j) - min\ (v_j)}$$

- Nominal attributes (e.g., color): distance is set to 1 if values are different, 0 if they are equal

Simple Distance Function

- With missing values?
  Common policy for missing values: assumed to be maximally distant (given normalized attributes)
- Are all attributes equally important?
  - No. Weighting the attributes might be necessary

With more Attributes?

- Another example with different attribute types

Object \ Feature	A	B	C	D	E	F	Class
Object 1	yes	no	no	6.4	8.3	low	positive
Object 2	yes	yes	yes	18.2	4.5	high	negative

new example	yes	no	no	6.6	8.1	low	?

What should its classification be?

Overview

- The Main Idea
- **Distance Measures**
- Efficient Computation
- Nearest Shrunken Centroids

Distance Measures

- Each NN or clustering problem is based on some kind of "distance" between points.
- Not every measure is a distance
  A distance measure respects some properties
- Two major classes of distance measure:
  1. *Euclidean*
  2. *Non-Euclidean*

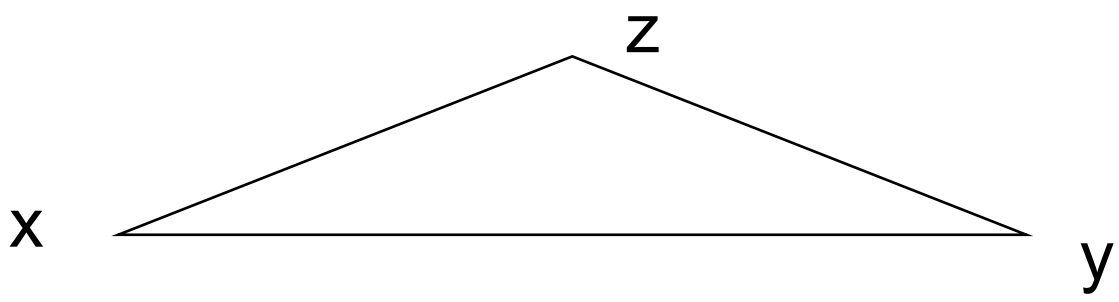
Euclidean vs. Non-Euclidean

- A *Euclidean space* has some number of real-valued dimensions
  - There is a notion of "average" of two points (gravity center)
  - A *Euclidean distance* is based on the locations of points in such a space

- A *Non-Euclidean distance* is based on properties of points, but not their "location" in a space

Axioms of a Distance Measure

- *d* is a *distance measure* if it is a function from pairs of points to real numbers such that:
  1. $d(x,y) \geq 0$.
  2. $d(x,y) = 0$ iff $x = y$.
  3. $d(x,y) = d(y,x)$.
  4. $d(x,y) \leq d(x,z) + d(z,y)$ (*triangle inequality*).



Some Euclidean Distances

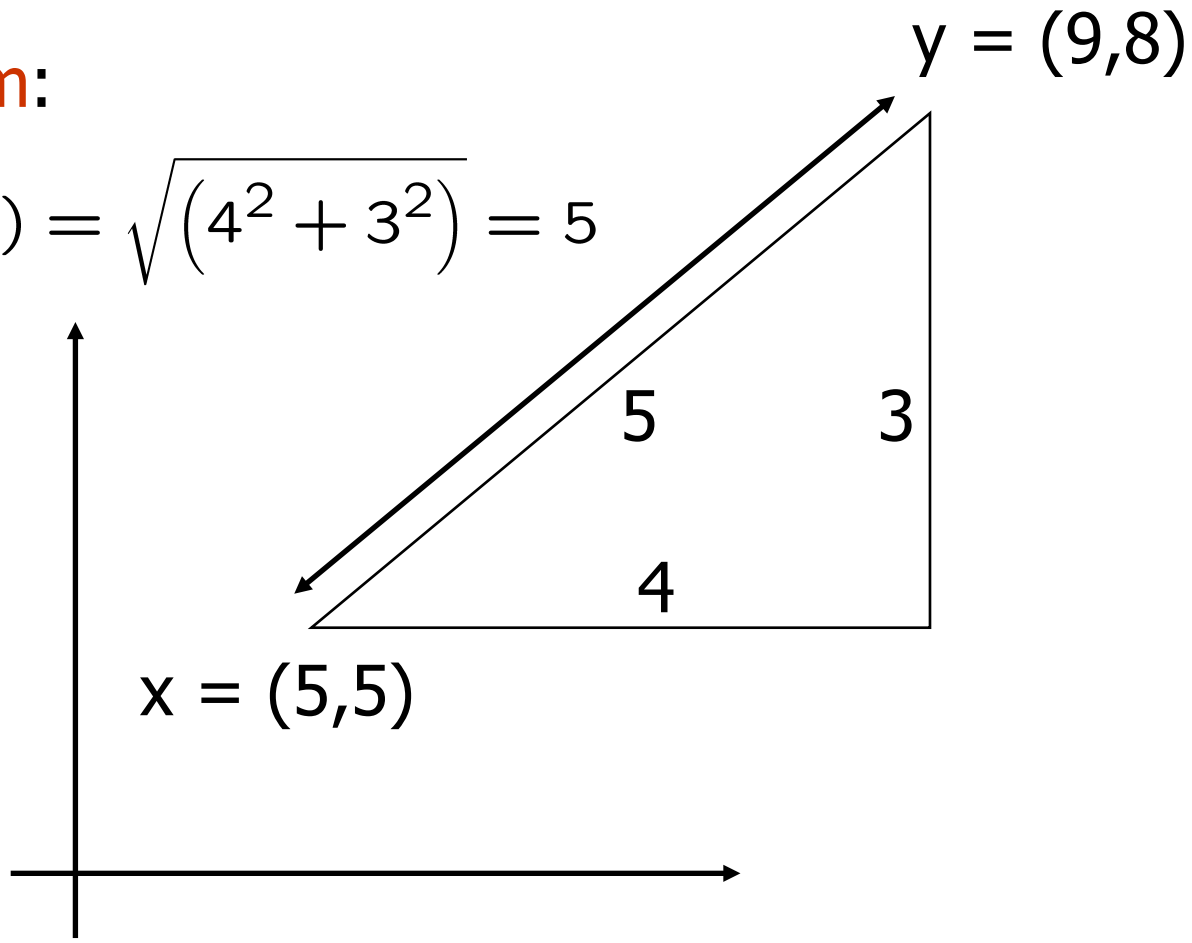
- L_2 *norm* : d(x,y) = square root of the sum of the squares of the differences between *x* and *y* in each dimension.
  - The most common notion of “distance.”
  - Distance "as the crow flies"
  - Defined as follows with two instances (x and y) having *n* attributes

$$d(x,y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Examples of Euclidean Distances

L_2-norm:

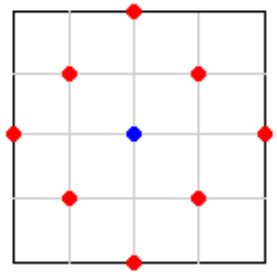
$$dist(x, y) = \sqrt{\left(4^2 + 3^2\right)} = 5$$



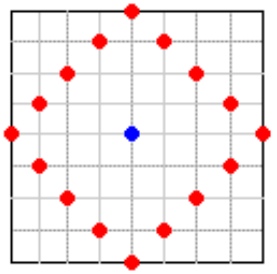
Some Euclidean Distances

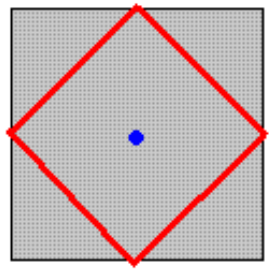
- L_1 *norm* : sum of the differences in each dimension.

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$



- *Manhattan distance* = distance if you had to travel along coordinates only "walking round the blocks"
- The key idea for selecting a distance metric is to think of actual instances (what it means for them to be separated by a given distance, what would twice that distance mean)

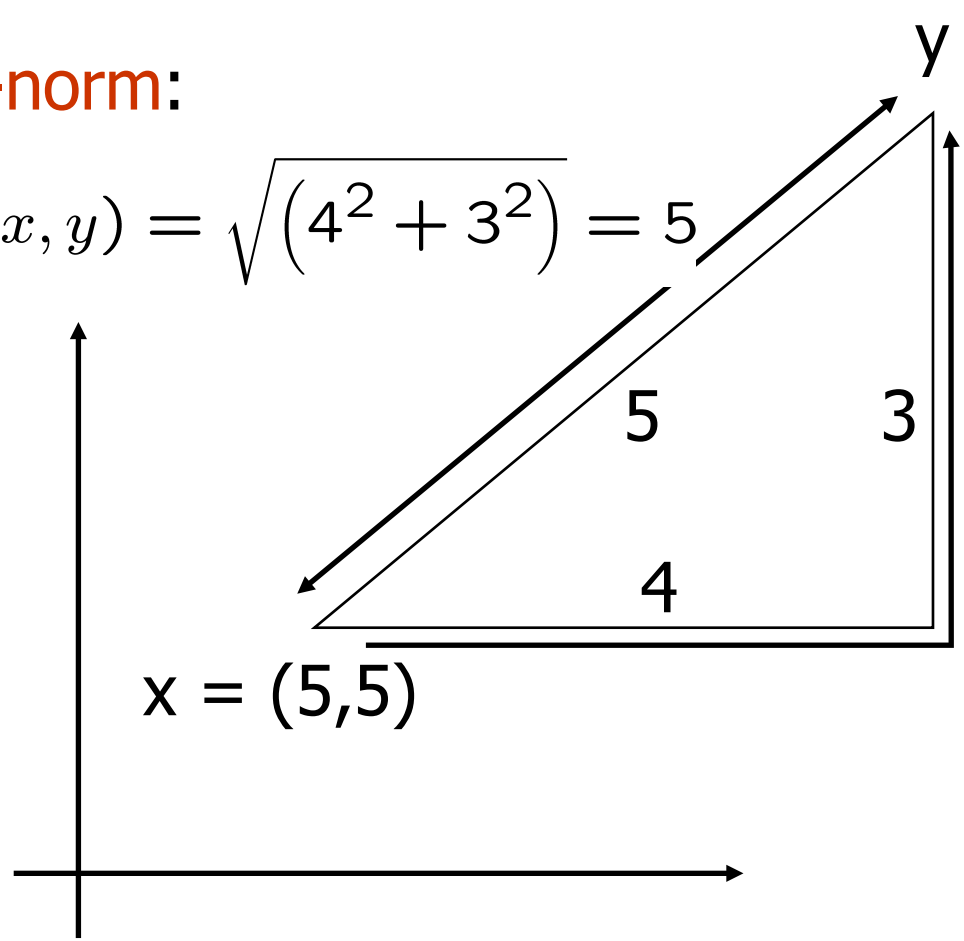




Examples of Euclidean Distances

L_2-norm:

$$dist(x,y) = \sqrt{\left(4^2 + 3^2\right)} = 5$$



L_1-norm:
dist(x,y) =
4 + 3 = 7

Another Euclidean Distance

- L_p *norm* : d(x,y) = generalization

  increases the influence of large differences

$$d(x,y) = \left[\sum_{i=1}^{n} |x_i - y_i|^p \right]^{1/p}$$

- L_∞ *norm* : d(x,y) = the maximum of the differences between *x* and *y* in any dimension or

$$d(x,y) = Max(|x_1 - y_1|, |x_2 - y_2|, \cdots, |x_n - y_n|)$$

- Note: the maximum is the limit as *p* goes to ∞ of what you get by taking the *p* th power of the differences, summing and taking the *p* th root.

Non-Euclidean Distances

- *Jaccard distance* for sets
  1 minus ratio of sizes of intersection and union
  (distance and similarity are opposed notion)
- *Cosine distance*
  angle between vectors from the origin to the points in question.
- *Edit distance*
  number of inserts and deletes to change one string into another.

Jaccard Distance for Bit-Vectors

- Definition $$Jaccard(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

  Values are between 0 and 1
- Example
  - Note by "1" the presence of the attribute
    p_1 = **1** 0 1 **1** **1**
    p_2 = **1** 0 0 **1** **1**.
  - Size of intersection = 3
    size of union = 4
    Jaccard similarity (not distance) = 3/4.
- d(x,y) = 1 – (Jaccard similarity)

  in our example 1 - 3/4 = 1/4.

Why Jaccard Distance is a Distance Measure

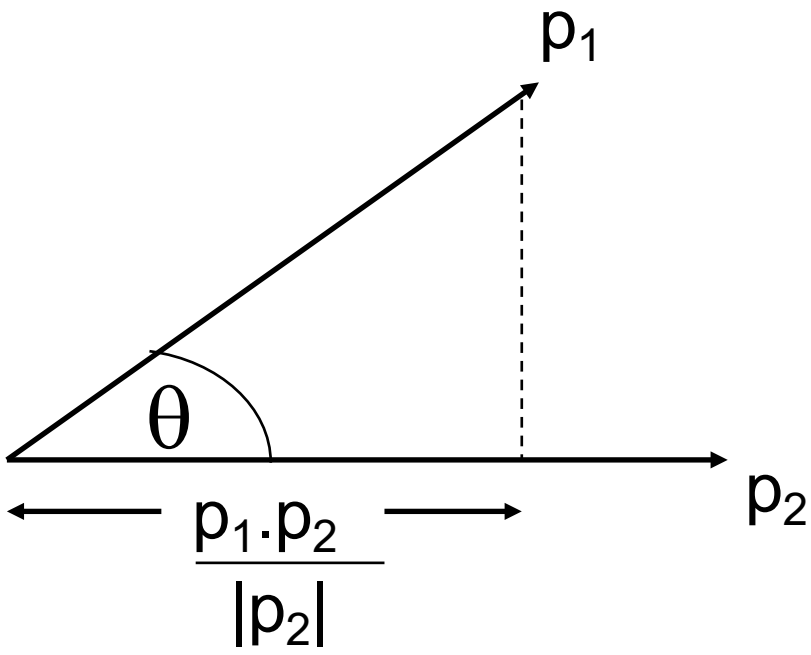
- Need to make a distance function satisfying triangle inequality and other laws.

- $d(x, x) = 0$ because $x \cap x = x \cup x$.
- $d(x, y) = d(y, x)$
  because union and intersection are symmetric.
- $d(x, y) \geq 0$ because $|x \cap y| \leq |x \cup y|$.
- $d(x, y) \leq d(x, z) + d(z, y)$ well …

Cosine Distance

- Think of a point as a vector from the origin (0, 0,…, 0) to its location.
- Two points' vectors make an angle, whose cosine is the normalized dot-product of the vectors: $p_1.p_2 / |p_2||p_1|$
  - Example
    p_1 = 00111; p_2 = 10011.
  - $p_1.p_2 = 2$; $|p_1| = |p_2| = \sqrt{3}$.
  - $\cos(\theta) = 2/3$; θ is about 48 degrees.

Cosine-Measure Diagram



$$\text{dist}(p_1, p_2) = \theta = \arccos(p_1.p_2/|p_2||p_1|)$$

Why Cosine Distance is a Distance Measure

- d(x,x) = 0 because arccos(1) = 0.
- d(x,y) = d(y,x) by symmetry.
- d(x,y) $\geq$ 0 because angles are chosen to be in the range 0 to 180 degrees.
- Triangle inequality: physical reasoning. If I rotate an angle from *x* to *z* and then from *z* to *y*, I can't rotate less than from *x* to *y*.

Edit Distance

- The edit distance of two strings is the number of inserts and deletes of characters needed to turn one into the other.
- Equivalently:
  $d(x,y) = |x| + |y| - 2|LCS(x,y)|$.
  - LCS = *longest common subsequence* = longest string obtained both by deleting from *x* and deleting from *y*.

Example

- $x = abcde$; $y = bcduve$.
- Turn x into y
  by deleting a,
  inserting u after d
  inserting v after u.
  - Edit-distance = 3.
- Or, LCS(x,y) = *bcde*.
- $|x| + |y| - 2|LCS(x,y)| = 5 + 6 - 2 \cdot 4 = 3$.

Why Edit Distance is a Distance Measure

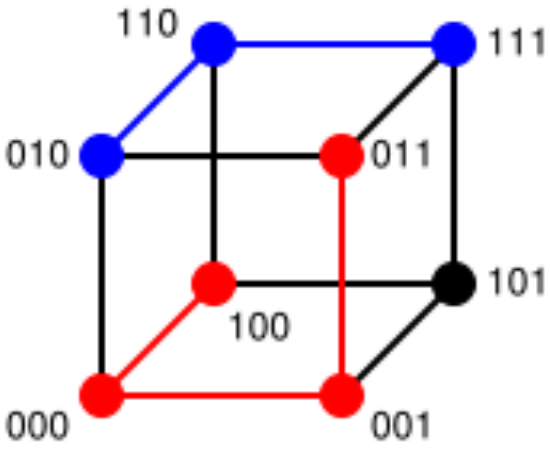
- d(x,x) = 0 because 0 edits suffice.
- d(x,y) = d(y,x) because insert/delete are inverses of each other.
- d(x,y) $\geq$ 0: no notion of negative edits.
- Triangle inequality: changing *x* to *z* and then to *y* is one way to change *x* to *y*.

Correlation-Based Distance

- You may compute the Pearson correlation distance based on Pearson correlation coefficient r
- r varies between -1 and +1, with 0 indicating no correlation between the two variables
- d = 1 - r and varies from 0 to 2
  - with 0 when r = 1, the two variables are the most similar
  - 2 when r = -1, the two variables are negatively correlated (most dissimilar)
- Variant: Absolute Pearson correlation distance
- d = 1 - |r| and varies from 0 to 1

Other Distances

- **Hamming distance** between two strings of *equal length* is the number of positions for which the corresponding symbols are different.
  It measures the minimum number of *substitutions* required to change one into the other, or the number of *errors* that transformed one string into the other



Overview

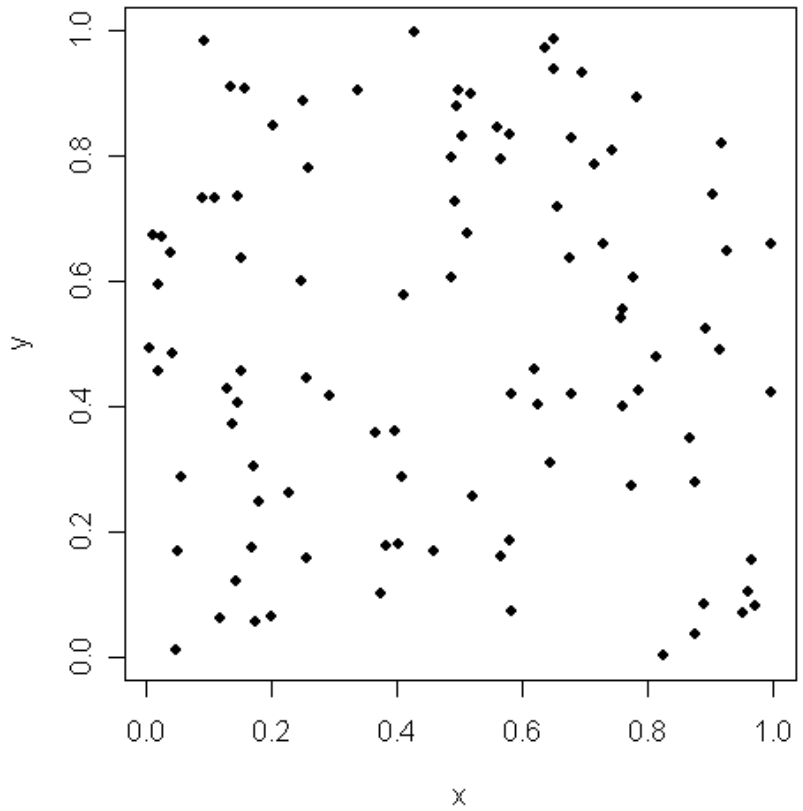
- The Main Idea
- Distance Measures
- **Efficient Computation**
- Nearest Shrunken Centroids

Computation

- Simplest way of finding nearest neighbor:
  linear scan of the data (brute force approach) O(n)
  - Classification takes time proportional to the product of the number of instances in training and test sets thus we have O($n_1 \cdot n_2$)

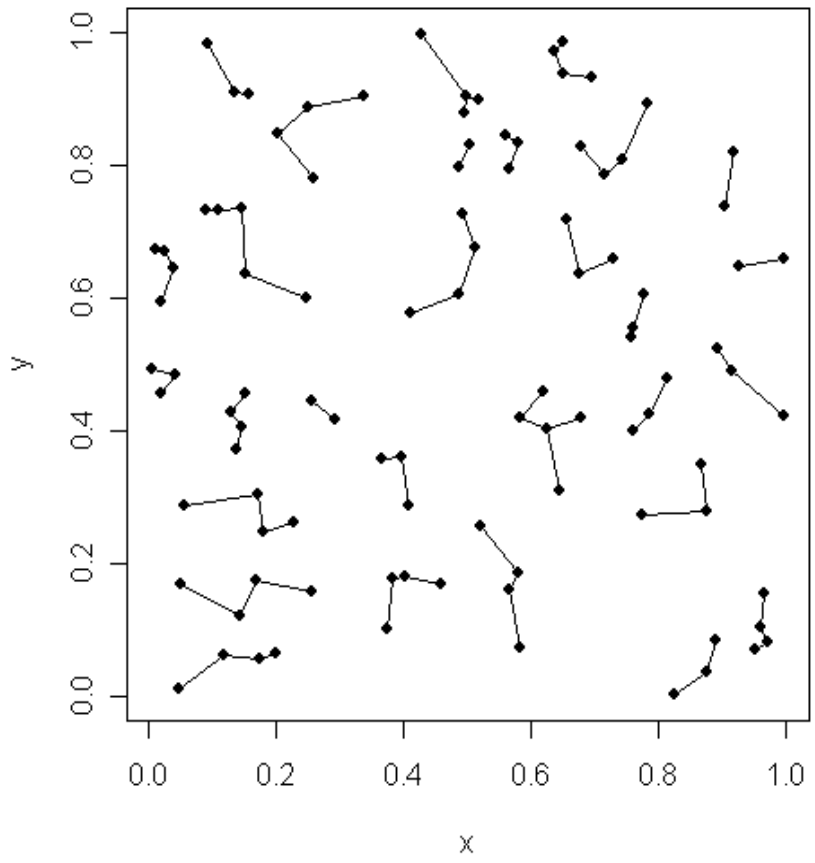
Computation

```
# generate random coordinate
x <- runif(100)
y <- runif(100)

# view the generate points
plot(x,y,pch=16)

# define the distance funct
distance <-
function(x1, y1, x2, y2)
  sqrt((x2-x1)^2 +
       (y2 - y1)^2)
```



Computation

```
r <- numeric(100)
nn <- numeric(100)
d <- numeric(100)
# compute the all the distance fr
   x[i], y[i]
for (i in 1:100)  {
   for (k in 1:100) {
         d[k] <- distance(x[i],
                     y[i],x[k],y[k]
      }
# select the min without the ith
   r[i] <- min(d[-i]);
   nn[i] <- which(d==min(d[-i]))
}
for (i in 1:100) lines(c(x[i],
   x[nn[i]]), c(y[i], y[nn[i]]))
```



Computation

- Simplest way of finding nearest neighbor:
  linear scan of the data (brute force approach) $O(n^2)$
- We can do better by having a *better* representation of our training set
- Nearest neighbor search can be done more efficiently using appropriate data structures
- We will discuss two methods that represent training data in a tree structure: *k*D-tree and ball tree (but shortly)

*k*D-Tree

- The *k*D-tree is a binary tree (to store a set of points in a *k*-dimensional space)
  - Each intern node may have up to two children
  - Divide the hyperspace by an hyperplane at each level (a line if we are in a 2D space)
  - Do it recursively
- All splits are made parallel to one of the axes
- The hyperplanes are not decision boundaries
- See computer graphics textbooks

*k*D-Tree

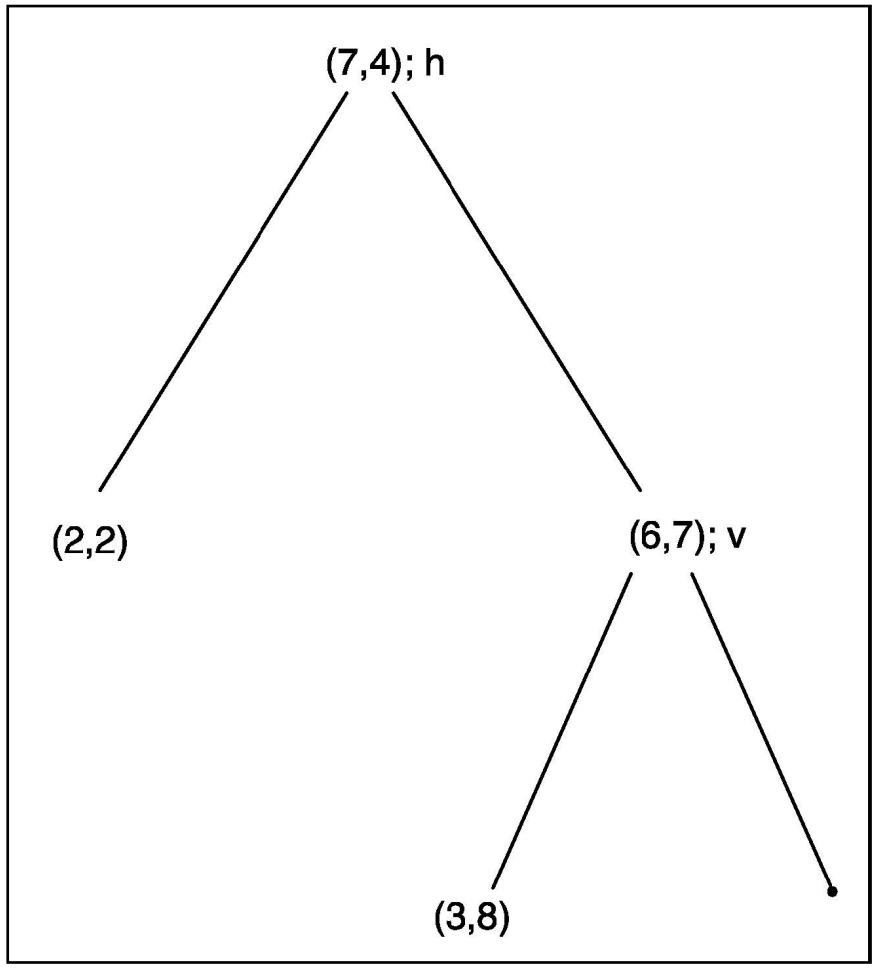
- Example

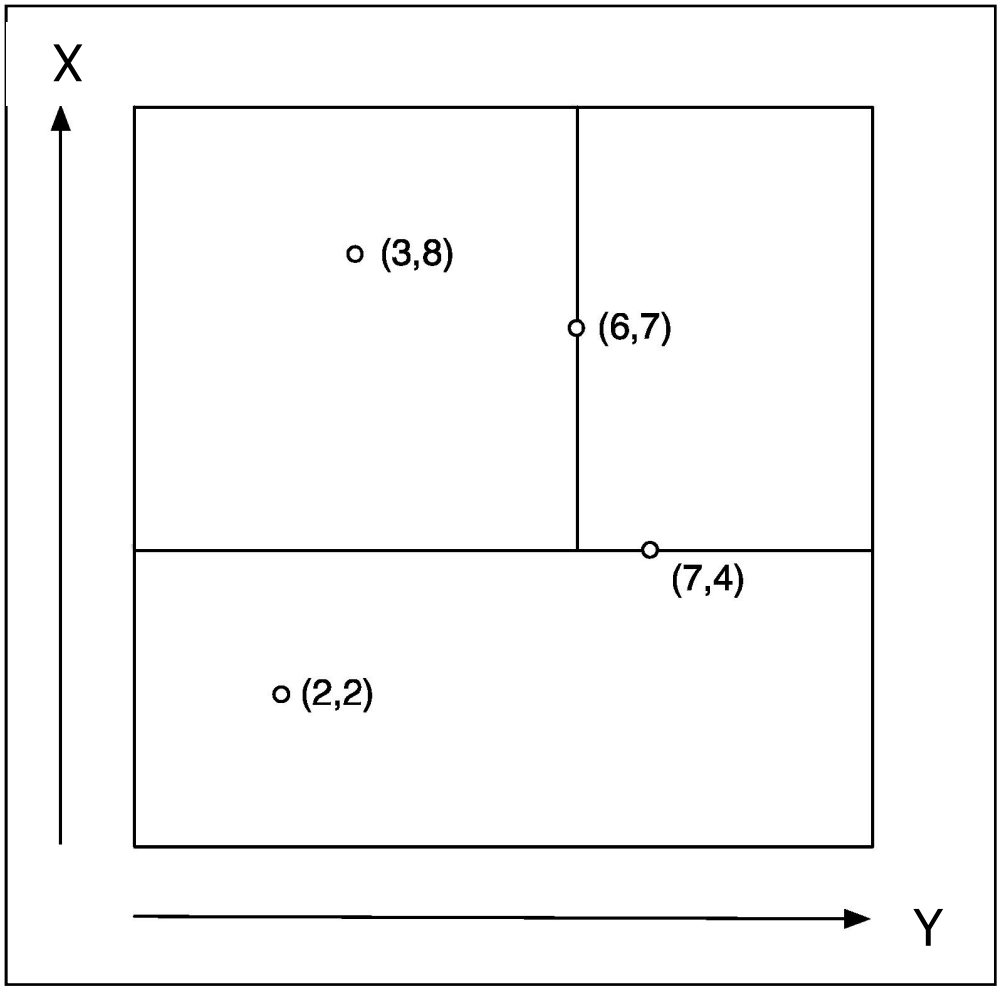
Objects	X	Y
a	2	2
b	3	8
v	6	7
h	7	4

- We select the a1 attribute to draw the first line (hyperplane).
- We select the instance *h* (7,4) as root.
  Smaller than Y=4 on the left, larger on the right
  (view as arbitrary decision for the moment)

*k*D-Tree

*k*D-tree (*k*=2) example with 4 examples



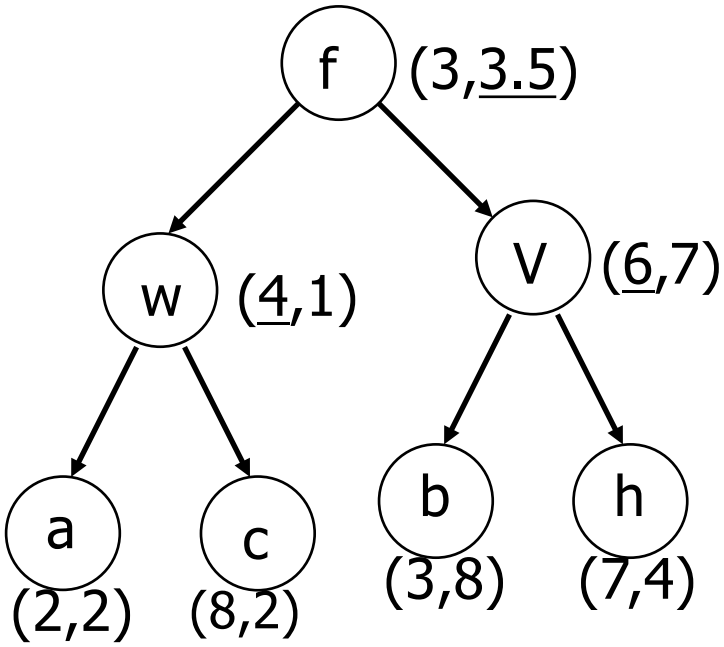


*k*D-Tree

- Then on example *v* (6,7)
  - each leaf contains either one point or none
  - each sibling branches are not necessarily developed at the same depth
- How can we search into such a tree when having a new instance?
- At each node, we must know the split direction (“X" or “Y" in our example) and the branches with upper or lower values.
- First we add three examples in our *k*D-tree

*k*D-Tree

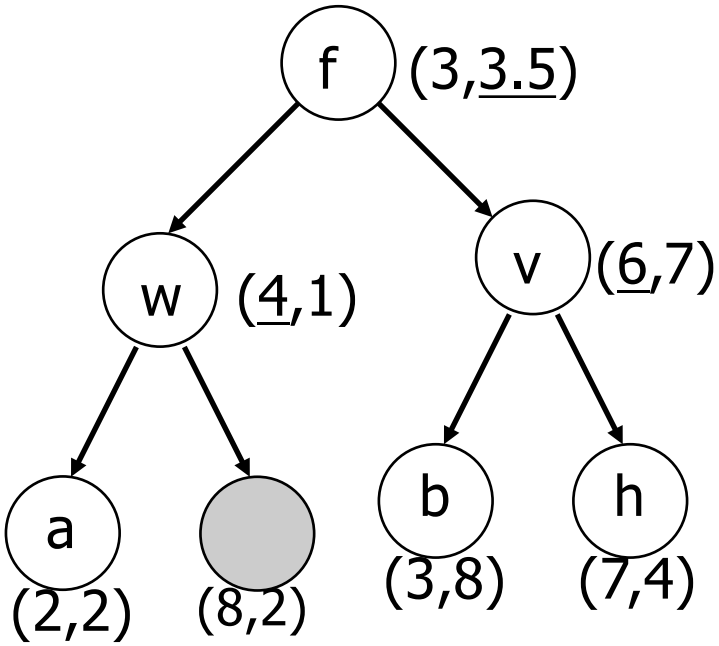
- We add new instances (*w*, *c*, *f*), we rebuild the tree.
- A new example (star)



Objects	X	Y
a	2	2
b	3	8
v	6	7
h	7	4
w	4	1
c	8	2
f	3	3.5
☼ (new)	8	3

*k*D-Tree

- How can we search into such a tree when having a new instance (e.g., ☼(8,3))?
- The star is the new instance. The leaf node of the region containing the target is colored grey. This is a good first approximation.

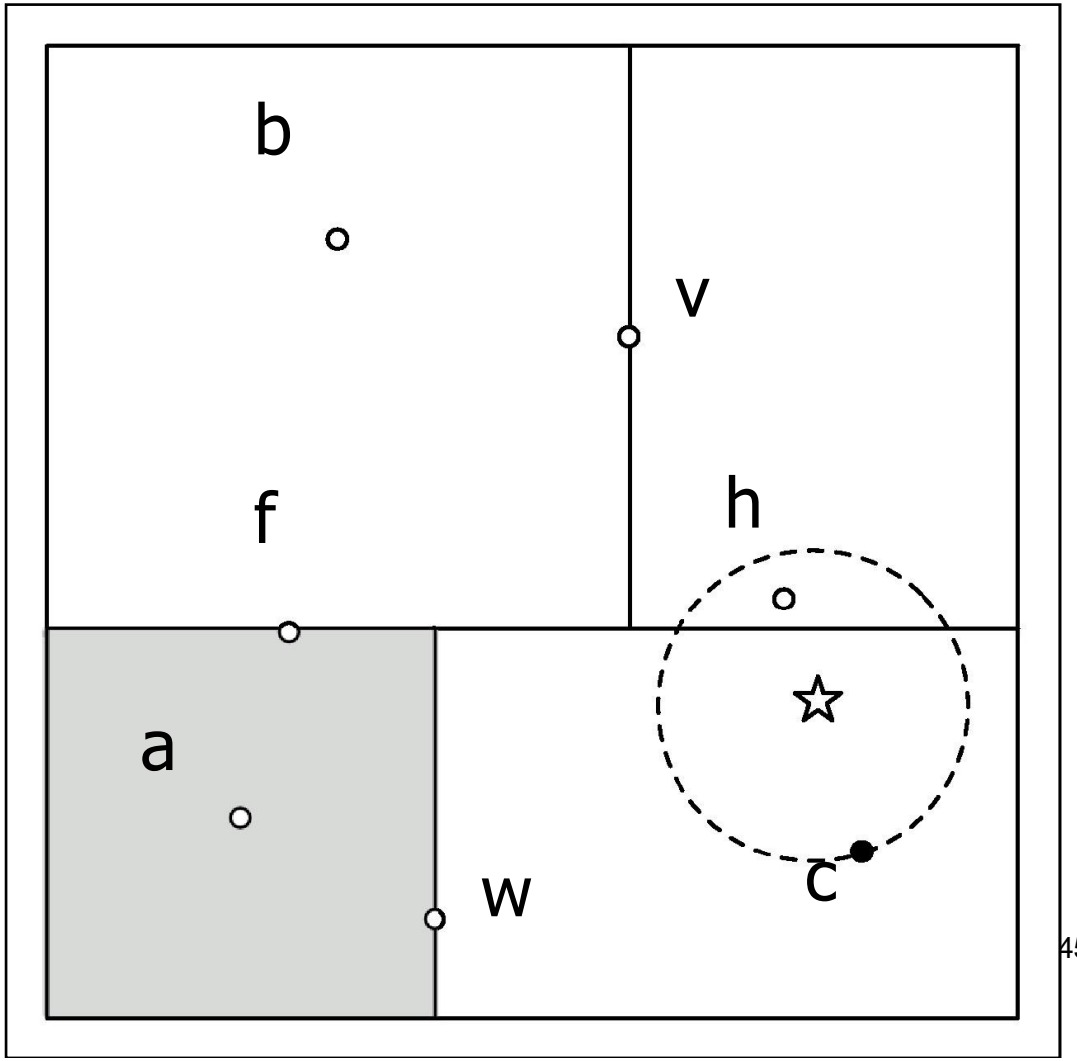


*k*D-Tree

$$Dist(c, \star) = \sqrt{(8-8)^2 + (3-2)^2} = 1$$

We have a potential nearest neighbor (with point *c*) and a potential shortest distance (distance(c,star)=1)

But the whole circle in not inside the rectangle!

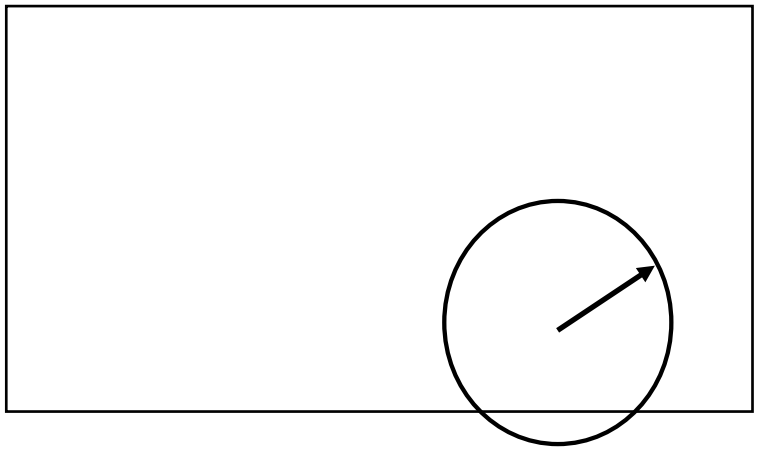


*k*D-Tree

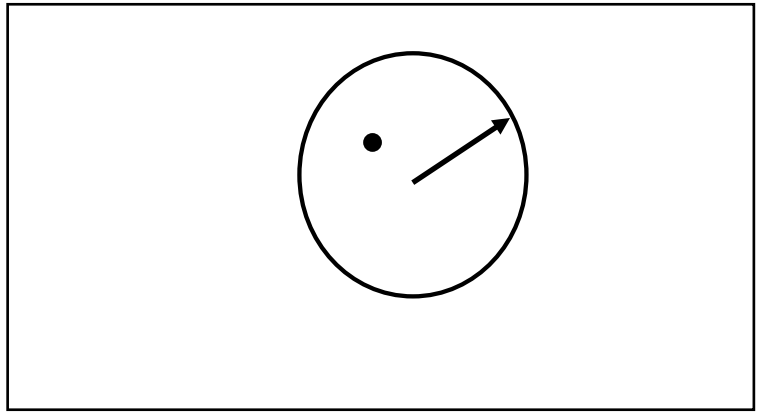
- Where to find the NN?
- Define the radius (distance between the new instance and the point in the leaf node, or between the star and the black point).
- Then you need some graphical testing functions

*k*D-Tree

Can you determine (fast) if the circle is inside a rectangle?

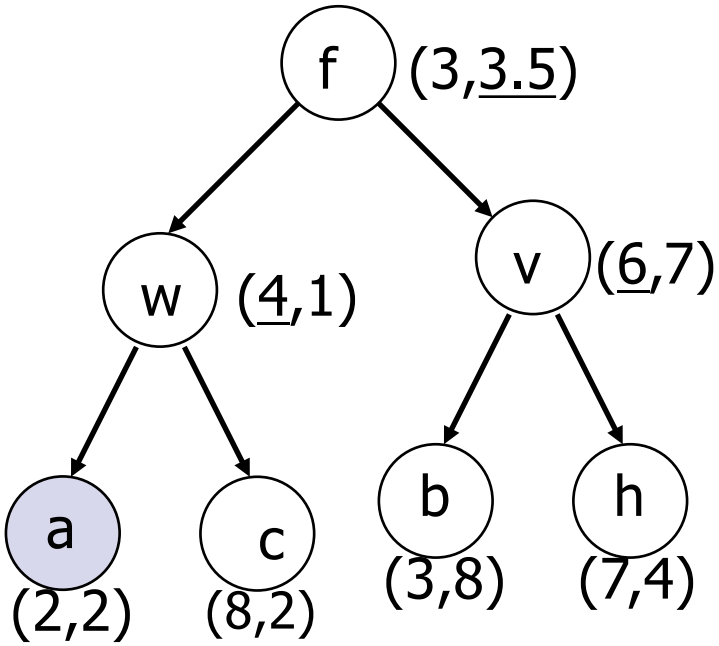


Can you determine (fast) if a point is inside a circle?



*k*D-Tree

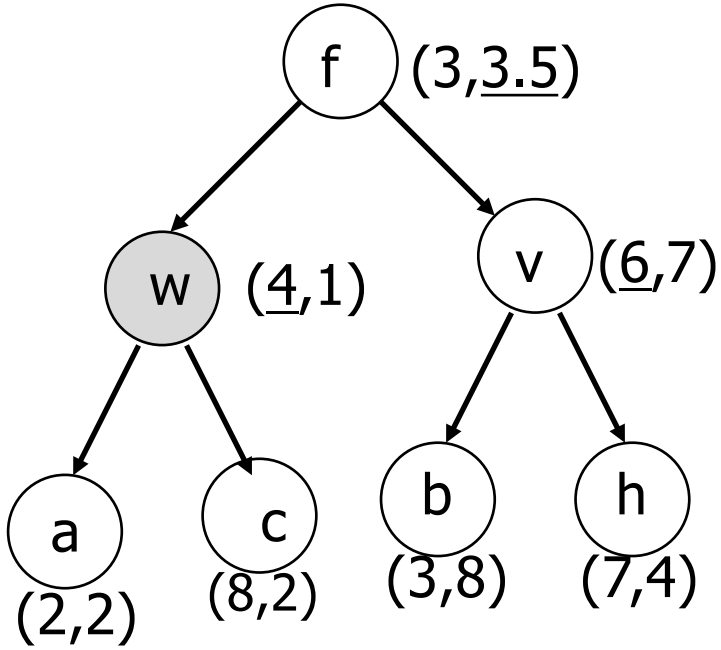
- From this candidate (c) with a potential shortest distance distance (1) we must backtrack to find the rectangle(s) that intersect with the circle.
- Backtracking procedure might go back to the root!
- Here, we visit rectangle “a”



*k*D-Tree

- Determine if it is possible for a closer neighbor to lie within the node's sibling (shaded).
  In our example, we have no intersection with the circle.

- If not, back up to the parent node (w, then f) and check *the* other siblings. (In our case, we must do it). Check if the aunt (v) intersects the circle and if yes, check for a closer point (its daughters, *b* and *h*) (it does in our example).



*k*D-Tree

- Complexity depends on depth of tree, given by logarithm of number of nodes $O(\log_2(n))$
- Amount of backtracking required depends on quality of tree ("square" (better) vs. "skinny" nodes (large rectangles)). But usually $O(\log_2(n))$
- If … if the tree is reasonable (well built)

*k*D-Tree

- How to build a good tree? Need to find good split point and split direction. After that apply the same strategy recursively.
- A splitting strategy
  - Split direction: direction with greatest variance
  - Split point: median value along that direction (the coordinates are easy to find).
- Using value closest to mean (rather than median) can better if data is skewed (but you need to find the point closer to it)
- Can apply this recursively

*k*D-Tree

- Big advantage of instance-based learning: classifier can be updated incrementally
  - Just add new training instances!
- 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 rebuilt occasionally (i.e. if depth grows to twice the optimum depth) to allow a better balance

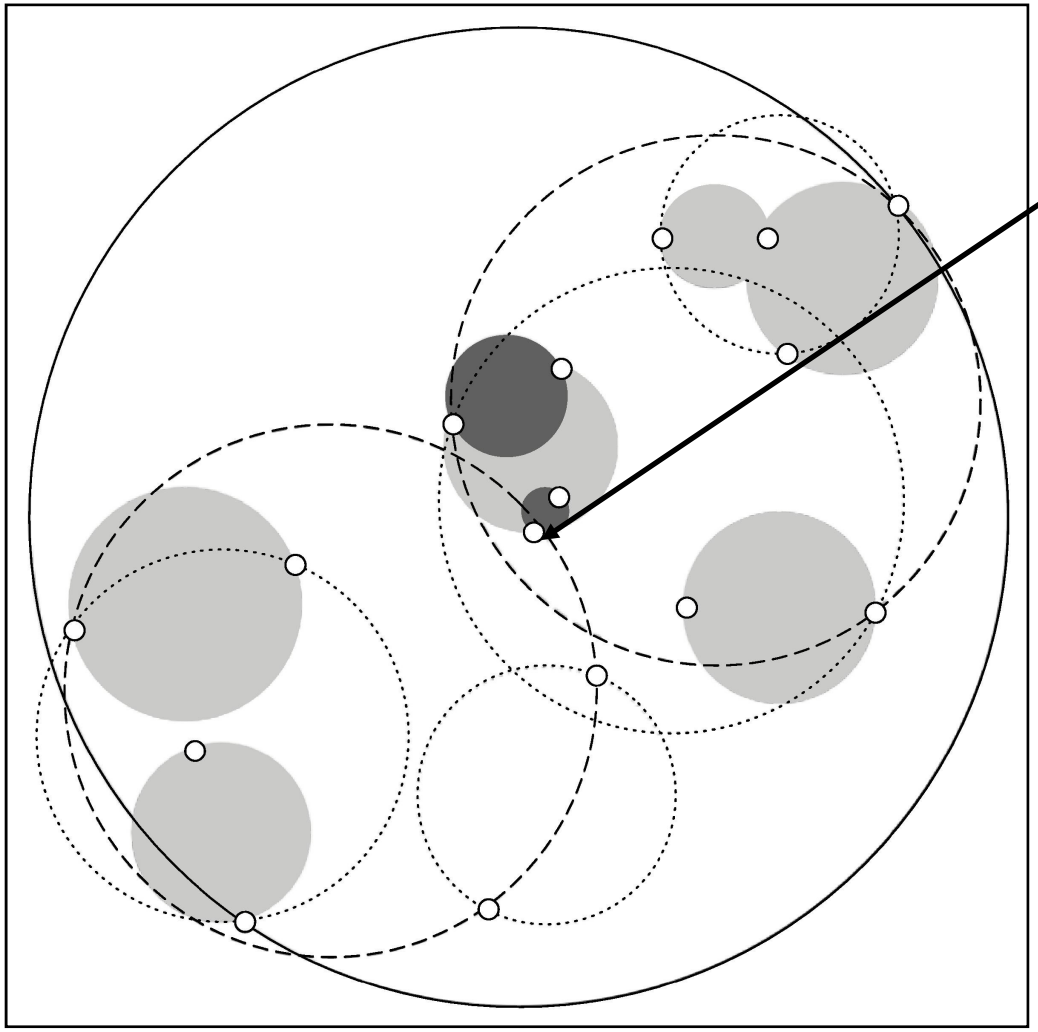
*k*D-Tree & Ball Tree

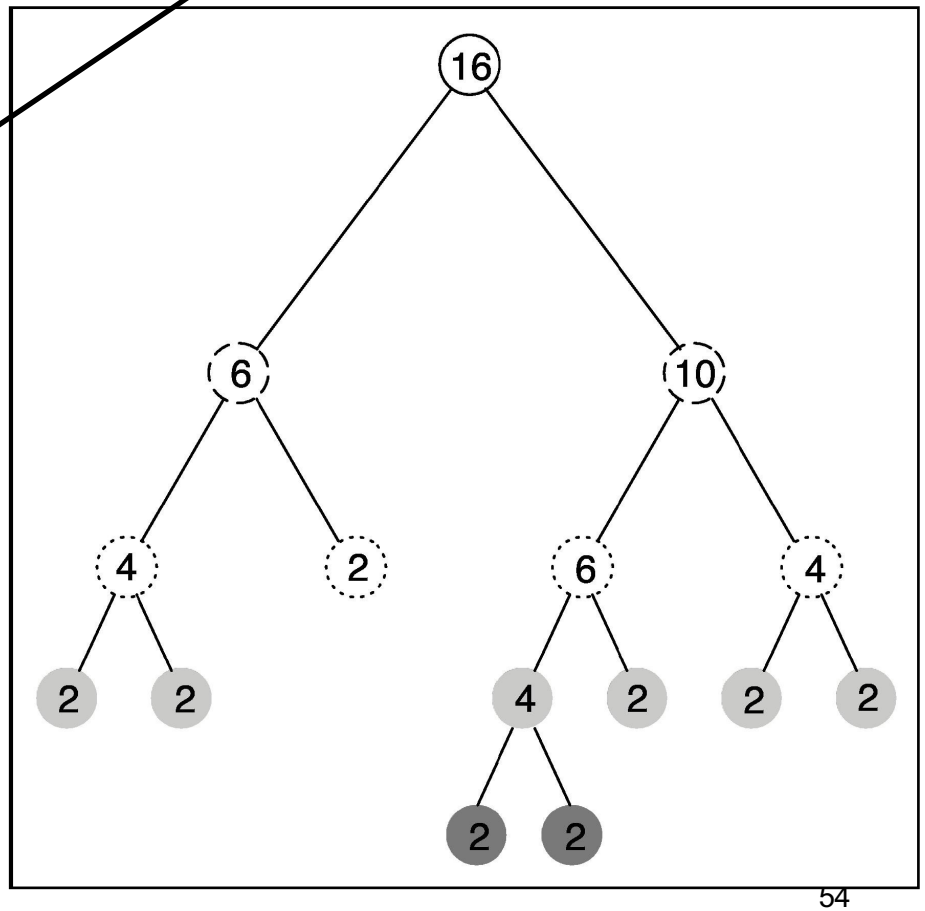
- Problem in *k*D-trees: corners with a new instance close to one of them
- Observation: no need to make sure that regions don't overlap
- Can use balls (hyperspheres) instead of hyperrectangles
- A *ball tree* organizes the data into a tree of *k*-dimensional hyperspheres
- Normally allows for a better fit to the data and thus more efficient search
- Point can belong to two circles but assigned to only one

Ball Tree

Example with 16 instances

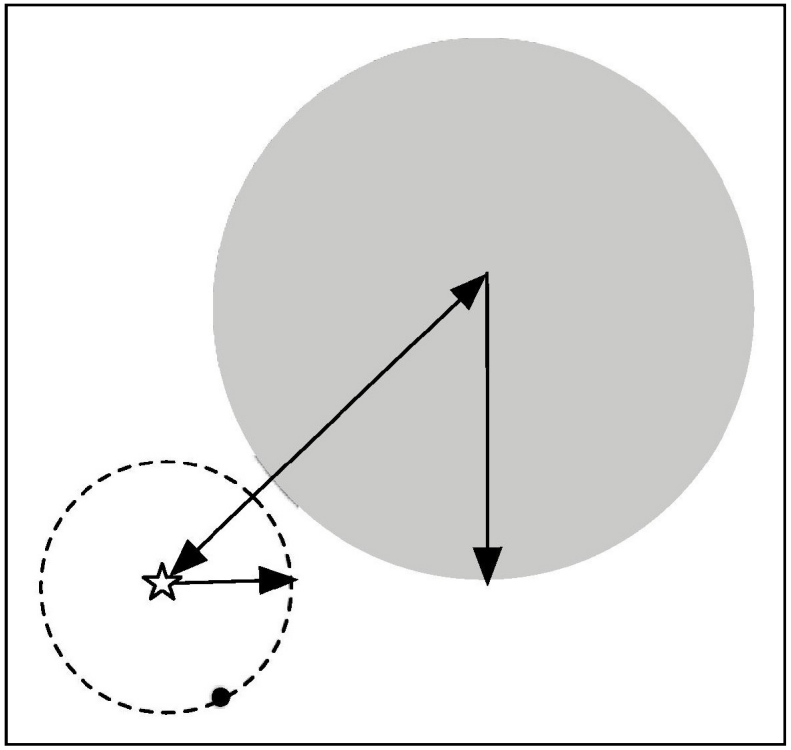
This point belongs to the other circle





Ball Tree

- Nearest neighbor search is done using the same backtracking strategy as in kD-trees (top-down approach to define the leaf)
- Ball can be ruled out from consideration
  if distance from target point (star) to ball's center exceeds ball's radius plus current upper bound (its nearest neighbor)



Ball Tree

- Ball trees are built top down (like *k*D-tree)
- Don't have to continue until leaf balls contain just two points: can enforce minimum occupancy (same in *k*D-tree)
- 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

Conclusion

- Simple and often very accurate
- Assumes all attributes are equally important (like the Naïve Bayes approach)
  - Remedy: attribute selection or weights
- Possible remedies against noisy instances:
  - Take a majority vote over the *k*-nearest neighbors (*k* odd)
  - Removing noisy instances from dataset (difficult!)
- Statisticians have used *k*-NN since early 1950s

Conclusion

- Analysis of *k*-NN approach
- If *n* grows, it is better to have $k > 1$
- If $n \rightarrow \infty$ and $k/n \rightarrow 0$, error approaches minimum
- Not really useful before *k*D-trees (early 90)
- *k*D-trees become inefficient when number of attributes is too large (approximately > 40)
- Ball trees (which are instances of *metric trees*) work well in higher dimensional spaces

Overview

- The Main Idea
- Distance Measures
- Efficient Computation
- **Nearest Shrunken Centroids**

Nearest Shrunken Centroids

- Mainly based on *k*-nn approach
- Represent each class by a centroid (profile)
  nearest centroid classification
- Ignore small discriminant values and thus
  remove irrelevant features (for a given class)
- Effective in different applications

Nearest Shrunken Centroids

- Given a set of features i = 1, 2, …, *p*
- Given a set of observations j = 1, 2, …, *n*
- Given a set of classes (labels) k = 1, 2, … *K*
- Let C_k indicates the n_k observations with the label *k*

- We can define the prior of each class by n_k / n

- Gravity center G = $[\bar{x}_1, \bar{x}_2, \ldots \bar{x}_p]$

  with the mean for the *i*th feature $\bar{x}_i = \frac{1}{n} \cdot \sum_{j=1}^{n} x_{ij}$

Nearest Shrunken Centroids

- Normalize within each class with

$$d_{ik} = \frac{\bar{x}_{ik} - \bar{x}_i}{m_k \cdot s_i}$$

the difference between the mean of the class and the *i*th feature of the gravity center.
d_{ik} close to 0, for the *k*th class, the *i*th feature not useful

- For the *k*th class and the *i*th feature, we compute

$$\bar{x}_{ik} = \frac{1}{n_k} \cdot \sum_{j \in C_k} x_{ij}$$

An Example

- Three target classes with 8 features

Feature	Red (50)	Green (75)	Blue (100)
color	50	75	100
red	48	2	4
green	2	73	3
blue	1	3	97
test	20	20	20
yellow	10	15	30
brown	25	2	50
sky	2	52	87

Nearest Shrunken Centroids

- And the denominator is the estimation of the standard error of the numerator.

$$d_{ik} = \frac{\bar{x}_{ik} - \bar{x}_i}{m_k \cdot s_i}$$

For the *i*th feature, estimated variance s_i^2

$$\hat{\sigma}_i^2 = s_i^2 = \frac{1}{n-K} \cdot \sum_{k=1}^{K} \sum_{j \in C_k} (x_{ij} - \bar{x}_{ik})^2$$

With $m_k = \sqrt{1/n_k - 1/n}$

d_{ik} is a *t*-statistic for the *i*th feature comparing class *k* to the average.

Nearest Shrunken Centroids

- We can rewrite the previous equation as:

$$\bar{x}_{ik} = \bar{x}_i + m_k \cdot s_i \cdot d_{ik}$$

- But the values d_{ik} must be shrunken towards zero, generating new d'_{ik} values

$$\bar{x}'_{ik} = \bar{x}_i + m_k \cdot s_i \cdot d'_{ik}$$

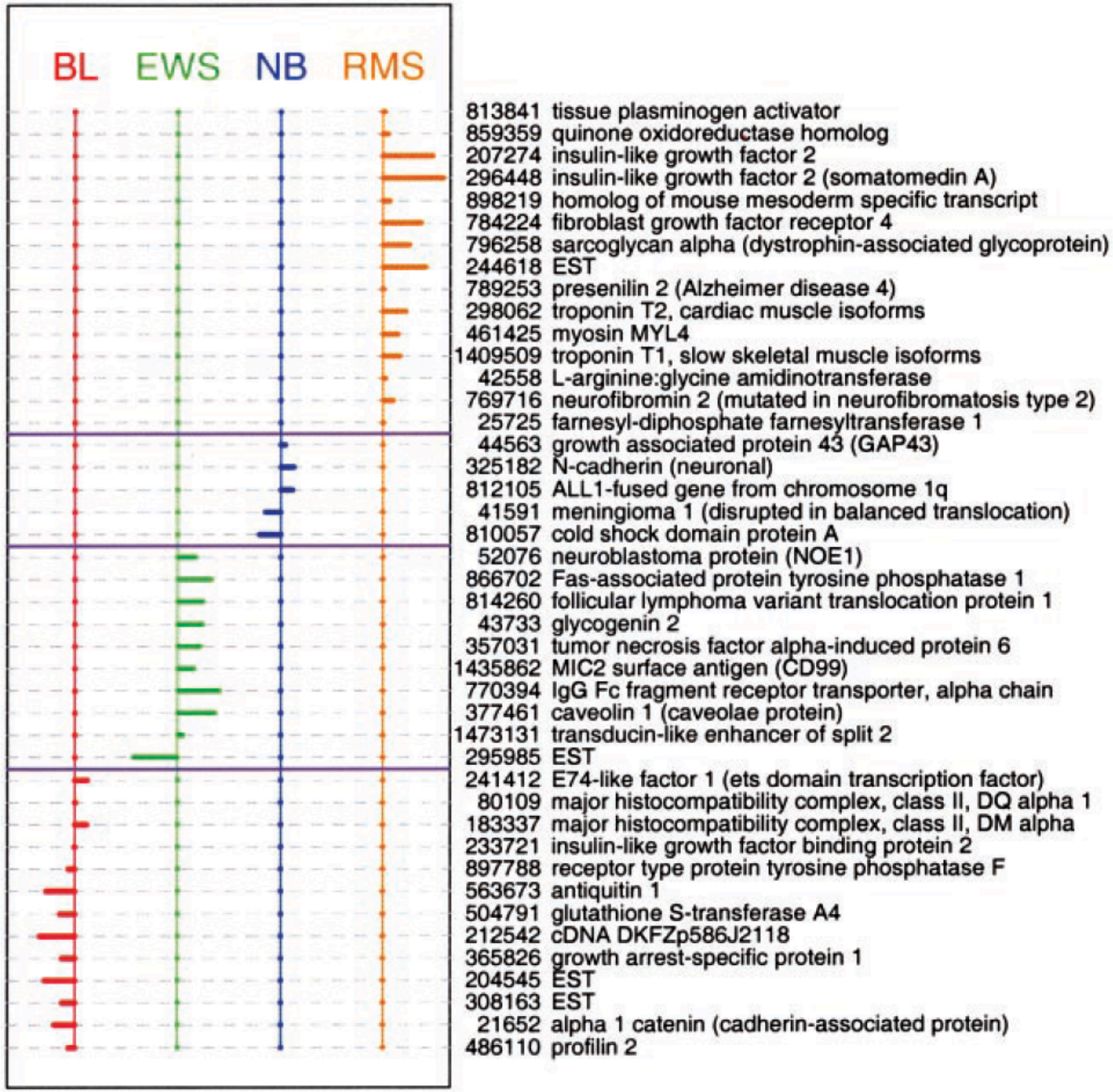
with $$d'_{ik} = sign(d_{ik}) \cdot (|d_{ik}| - \Delta)_+$$

and $(x)_+$ is the positive part if $x > 0$, 0 otherwise
Small values of d_{ik} will be set to 0.

Nearest S

Ignore some of the features (gene) for each class

d'_{ik} values for the four possible diseases



Nearest S

Varying the amount of shrinkage and the effect on the number of features (genes)

cv in red
test in blue
(training in green

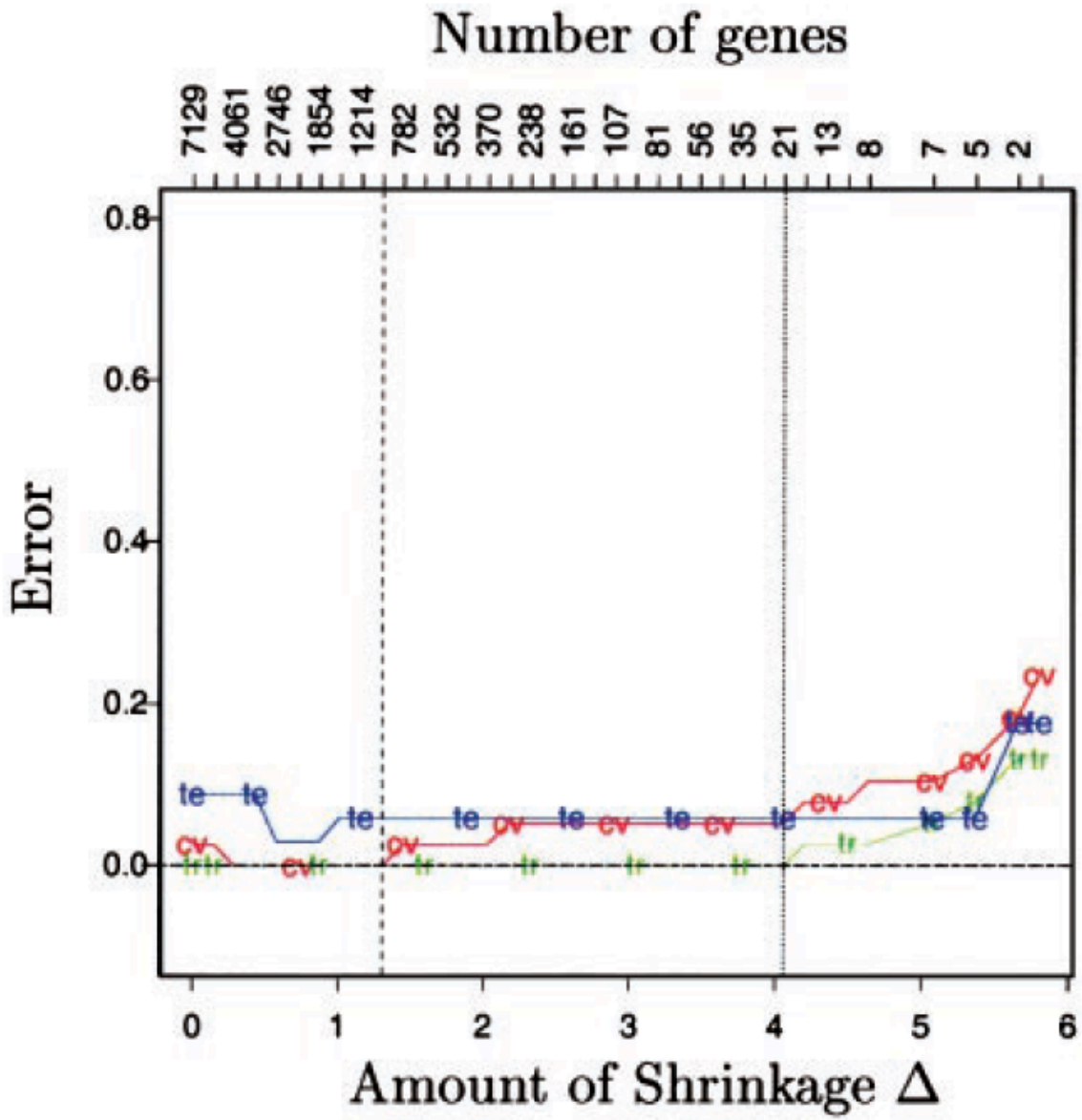


Fig. 6. Leukemia classification: training (tr, green), cross-validation (cv, red), and test (te, blue) errors. The value $\Delta = 4.06$ yields a subset of 21 genes.

Conclusion

- Instead of storing all training instances, compress them into regions
- Another 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)

Conclusion

- Variant of *k*-NN approach
- Selecting the features inside the classifier
- NSC tends to produce effective solution
- But the shrinkage level in snot specify by a theory (determine by CV)