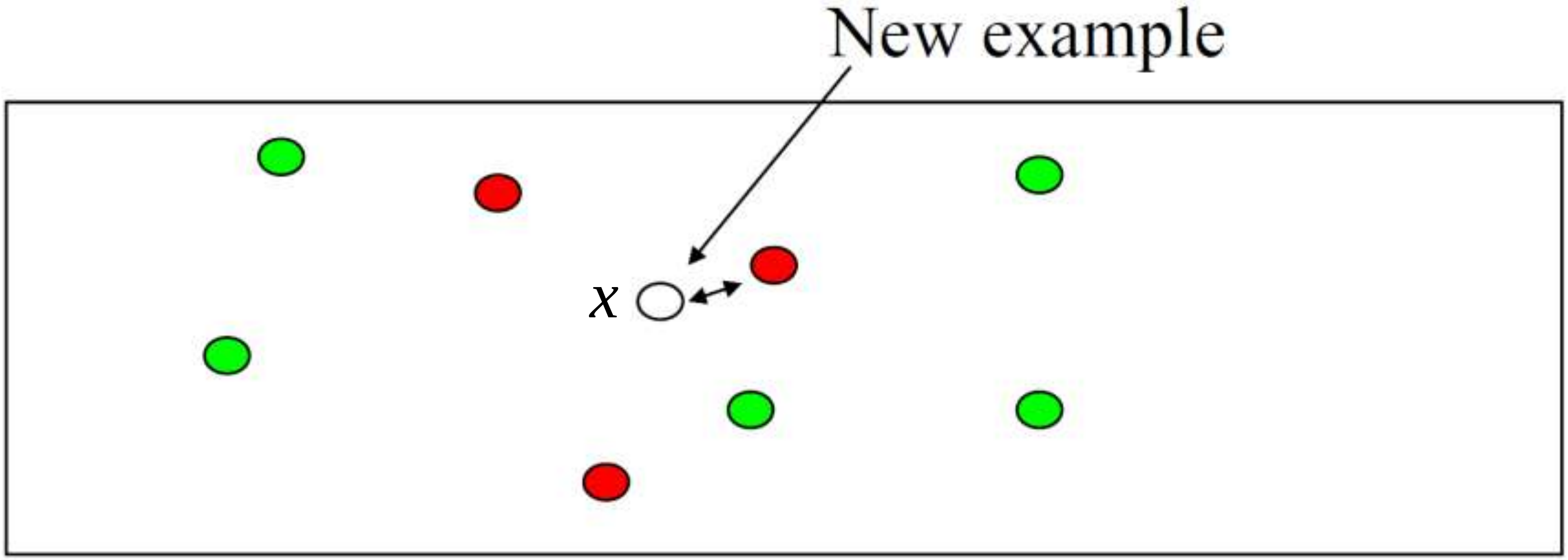
k Nearest Neighbors Algorithm

Ringo

2017.2.4

The Nearest Neighbor Algorithm

- Remember all training examples
- Given a new example x, find it's closest training example $< x^i, \ y^i >$ and predict y^i



- How to measure distance – Euclidean (squared):

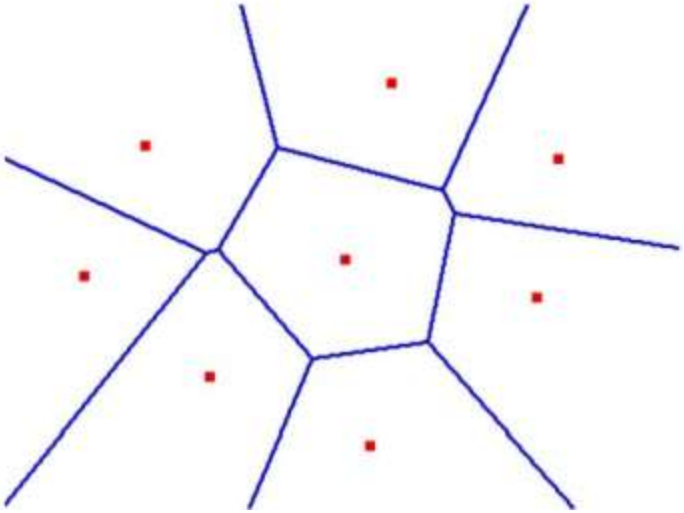
$$\| x - x^i \|^2 = \sum_j (x_j - x_j^i)^2$$

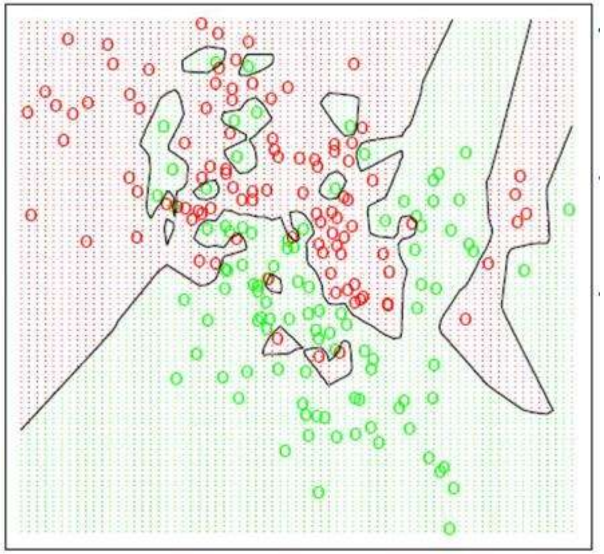
The Nearest Neighbor Algorithm

- A **lazy learning** algorithm
  - the "learning" does not occur until the test example is given
  - in contrast to so called "eager learning" algorithms (which carries out learning without knowing the test example, and after learning training examples can be discarded)

Decision Boundaries

- Given a set of points, a **Voronoi diagram** describes the areas that are nearest to any given point.
- With large number of examples and possible noise in the labels, the decision boundary can become nasty!



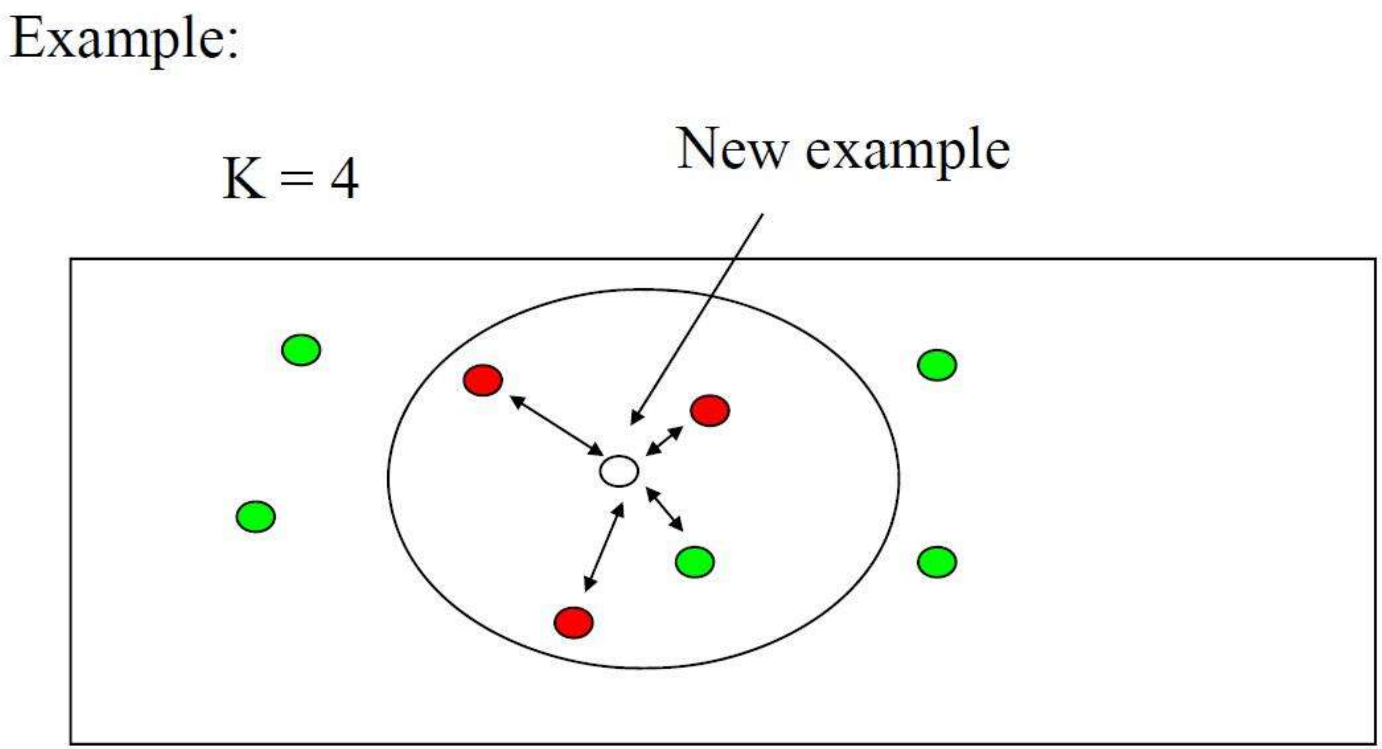


k Nearest Neighbor: Algorithm

- Remember all training samples $\{(x_1, y_1), (x_2, y_2)...(x_n, y_n)\}$
  x_i is the vector of a sample, y_i is the class of x_i
- Given an unlabeled example x, find ***k*** most similar labeled examples (closest neighbors among sample points)
- Assign the most frequent class among those neighbors to x (***Majority voting***)

$$y = \arg\max \sum_{x \in N_k(x)} I(y_i = c_j)$$

k Nearest Neighbor: Example



Find the ***k*** nearest neighbors and have them vote. Has a smoothing effect. This is especially good when there is noise in the class labels.

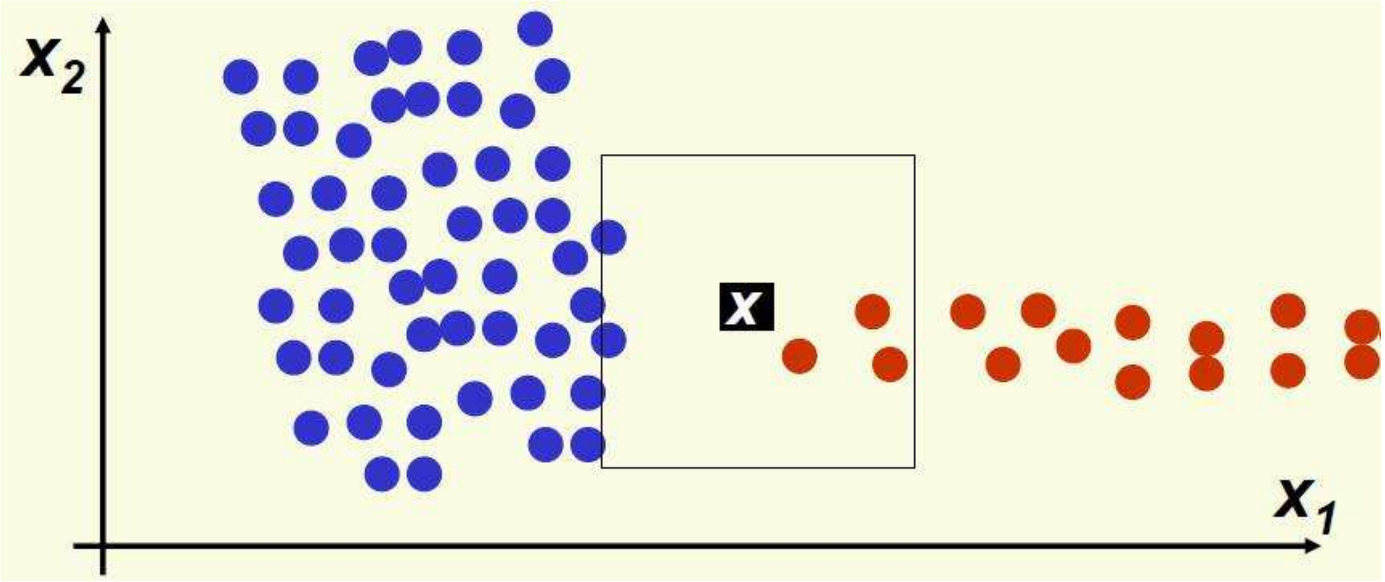
k Nearest Neighbor: 3 elements

- How to choose ***k***?

- Classification decision rule?

- How to measure distance?

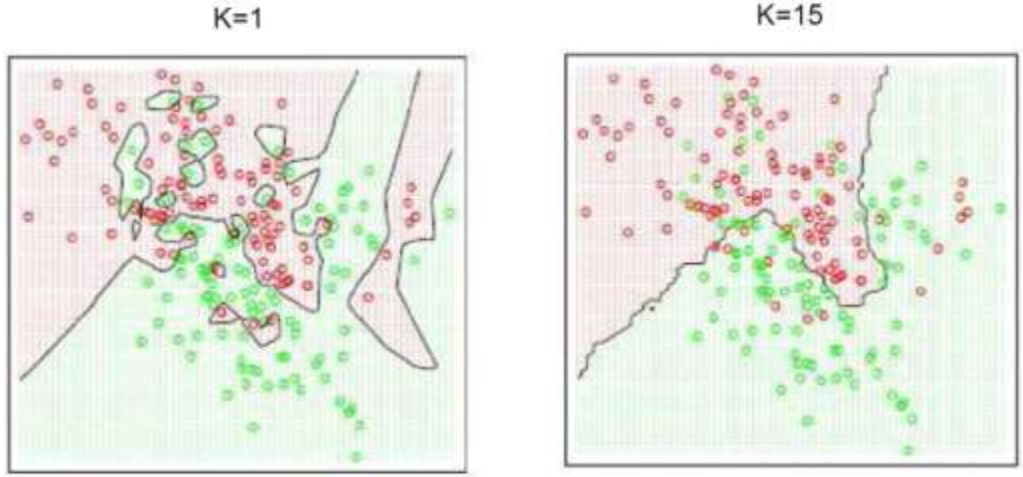
How to choose *k*?



- For ***k*** = 1,…5 point x gets classified correctly
  — red class
- For larger ***k*** classification of x is wrong
  — blue class

How to choose *k*?

- If ***k*** is too small
  - The result can be sensitive to noise points. larger ***k*** produces smoother boundary effect and can reduce the impact of class label noise.
- If ***k*** is too large
  - The neighborhood may include too many points from other classes, especially when ***k*** = N, we always predict the majority class.



How to choose *k*?

- "rule of thumb": $k = \sqrt{n}$
  - — can prove convergence if n goes to infinity
  - — not too useful in practice, however
- K-fold cross validation
  - — the meaning of "K" in K-fold is not the same with ***k***
  - — it is particularly suited for:
    - typically have only a few possible candidates for ***k*** (e.g. in order 3-10 or 50-100)
    - performance is rather monotone on the number of neighbors.

Classification decision rule?

Majority Voting Rule:

— Take a majority vote class label for the new sample

Definition:

Input: D, the set of training objects, and test object

$z = (X', y')$

Process:

1. Compute $d(X', X)$, the distance between z and every object, $(X, y) \in D$.
2. Select $D_z \in D$, the set of ***k*** closest training objects to z

Output: $$y' = \arg\max_{v} \sum_{(X_i, y_i) \in D_z} I(v = y_i)$$

Majority Voting: Interpretation

Assumption:

0-1 loss function: $I(Y, f(X)) = \begin{cases} 1, Y \neq f(X) \\ 0, Y = f(X) \end{cases}$

classification function $f : \mathrm{R}^{\mathrm{n}} \rightarrow \{c_1, c_2, \ldots, c_m\}$

So:

Misclassification rate for $z = (X, Y)$:

$$P(Y \neq f(X)) = 1 - P(Y = f(X)) = 1 - I(Y, f(X))$$

$N_k(z)$ is the set of ***k*** closest training objects to z, if the class of z is c_j, then the misclassification rate is:

$$P(Y \neq f(X)) = 1 - I(Y, f(X)) = 1 - \frac{1}{k} \sum_{x_i \in N_k(x)} I(y_i, c_j)$$

Majority Voting: Interpretation

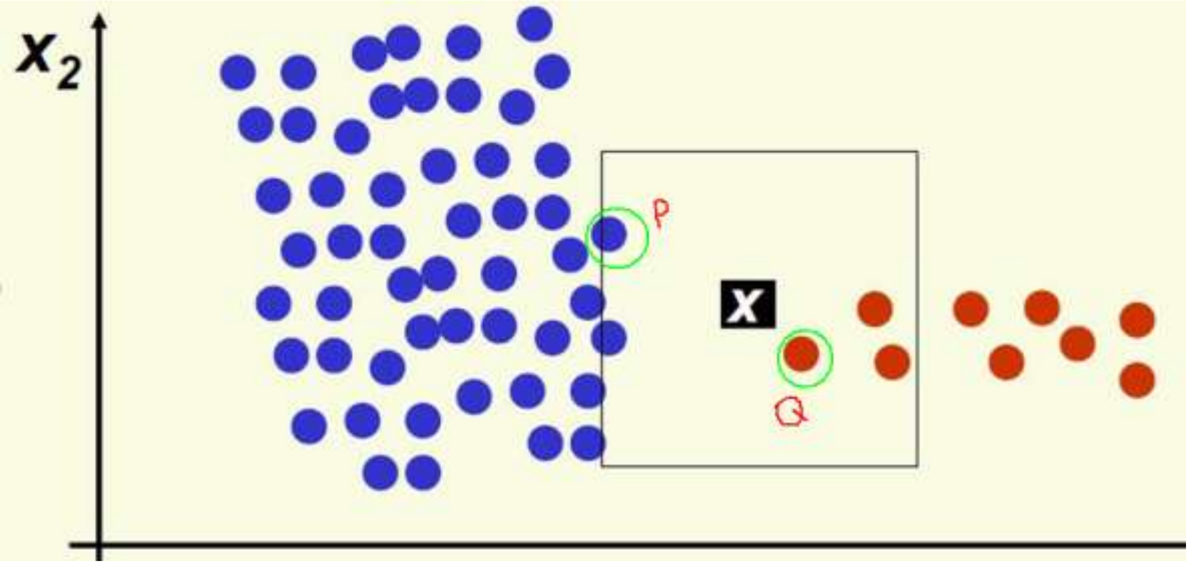
$$P(Y \neq f(X)) = 1 - I(Y, f(X)) = 1 - \frac{1}{k} \sum_{x_i \in N_k(x)} I(y_i, c_j)$$

$$\min P(Y \neq f(X)) \Rightarrow \max \frac{1}{k} \sum_{x_i \in N_k(x)} I(y_i, c_j)$$

$$\Rightarrow c_j = \arg \max_{p \in \{1,..m\}} \sum_{i=1}^{k} I(y_i, c_p)$$

So majority voting is equivalent to empirical risk minimization.

Majority Voting: Go farther



Can P and Q be treated equally?

Improvement: weights each object's vote by its distance.

Suppose:

$$\omega_i = \frac{1}{d(X', X_i)^2}$$

Distance-Weighted Voting: $y' = \arg\max_{v} \sum_{(X_i, y_i) \in D_z} \omega_i \times I(v = y_i)$

Majority Voting: Go farther

Distance-Weighted Voting: $y' = \arg\max_{v} \sum_{(X_i, y_i) \in D_z} \omega_i \times I(v = y_i)$

Advantage:

— The closer neighbors more reliably indicate the class of the object

— Much less sensitive to the choice of ***k***

How to measure distance?

Aim: a smaller distance between two objects implies a greater likelihood of having the same class.

Choice:

- Euclidean Distance

$$L(x_i, x_j) = (\sum_{l=1}^{n} | x_i^{(l)} - x_j^{(l)} |^2)^{\frac{1}{2}}$$

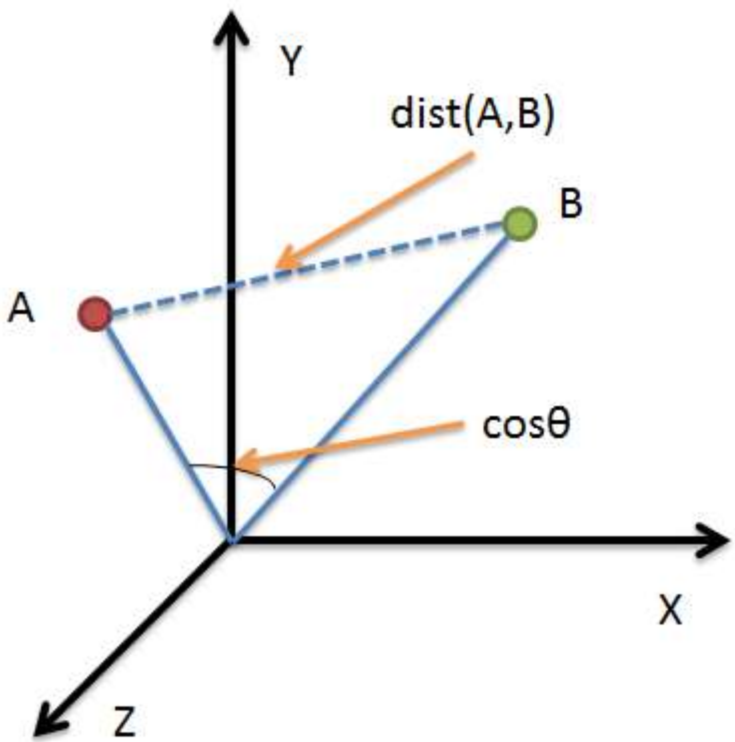
- Cosine Similarity

$$sim(X, Y) = \cos\theta = \frac{\vec{x} \cdot \vec{y}}{\| x \| \cdot \| y \|}$$

- ……

How to measure distance?

Euclidean Distance vs Cosine Similarity :



Euclidean Distance	Cosine Similarity
focus on numerical differences	focus on dimensional differences
values vary widely	bound on the interval of [-1,1]
suffer from: 1. curse of dimensionality 2. noisy/Irrelevant features	not influenced, but suffer the loss of numerical difference

Tips for using Euclidean Distance:

- feature normalization(e.g $x_1 = [1,100], x_2 = [2,150]$)
- feature weighting
- not a good distance in high dimensions

kNN: Computational Complexity

Basic ***kNN*** algorithm stores all examples. Suppose we have n examples each of dimension d, requiring $O(nd)$ memory

- $O(d)$ to compute distance to one training example
- $O(nd)$ to compute distances to all examples, storage takes $O(n)$ memory
- For $i=1:k$, loop through all training set, selecting the smallest $dist_i$ that has not been selected before, $O(nk)$

Time complexity: $O(nd+nk)$ Space complexity: $O(nd+n)$

Paradox :

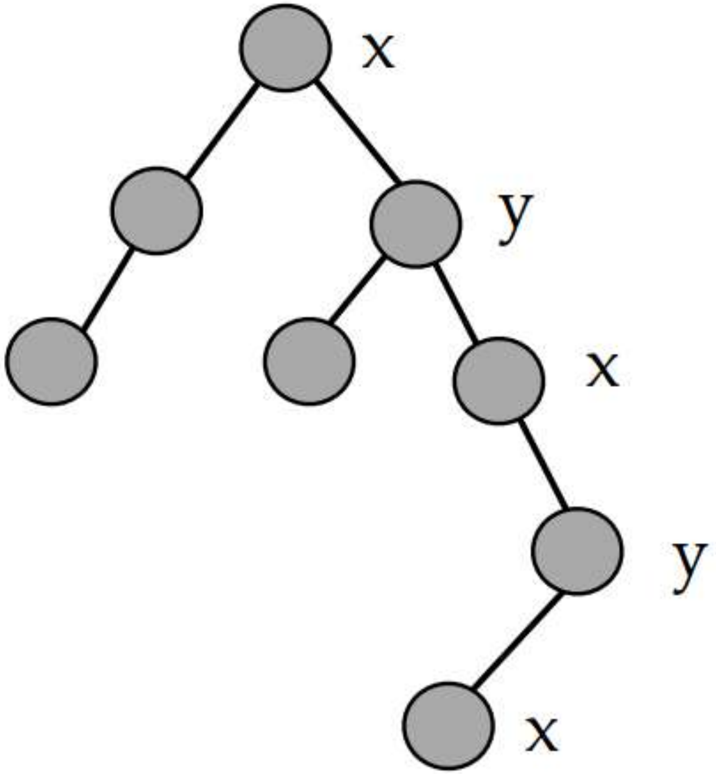
— it's expensive for large number of samples

— but we need large number of samples for ***kNN*** to work well !

Solution: kd-Tree

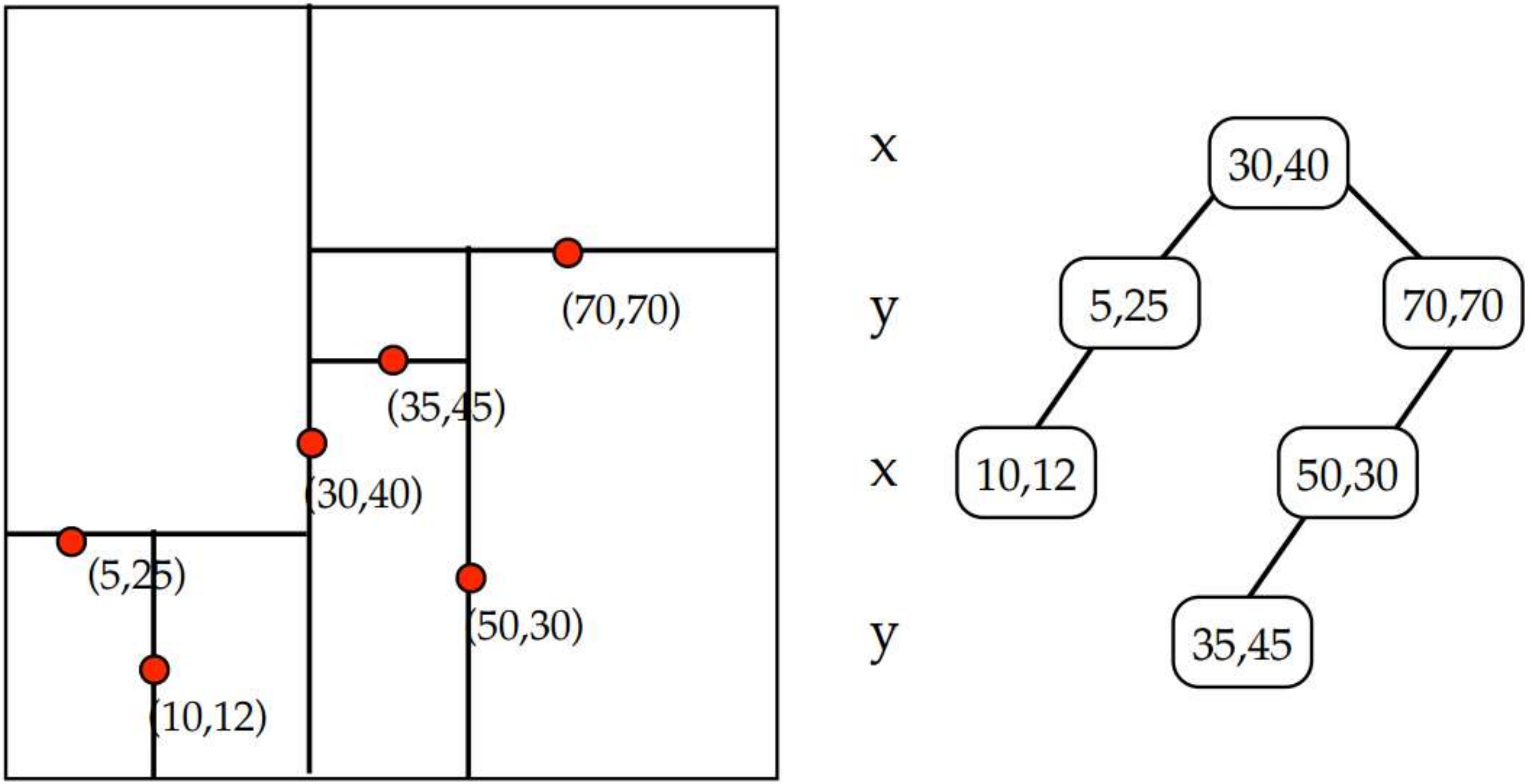
kd-Tree

- Each level has a “cutting dimension”
- Cycle through the dimensions as you walk down the tree
- Each node contains a point $P = (x', y')$
- To find (x', y') you only compare coordinate from the cutting dimension
  - e.g. if cutting dimension is x , then you ask: is $x' < x$?



kd-Tree: Example

Insert: (30,40), (5,25), (10,12), (70,70), (50,30), (35,45)



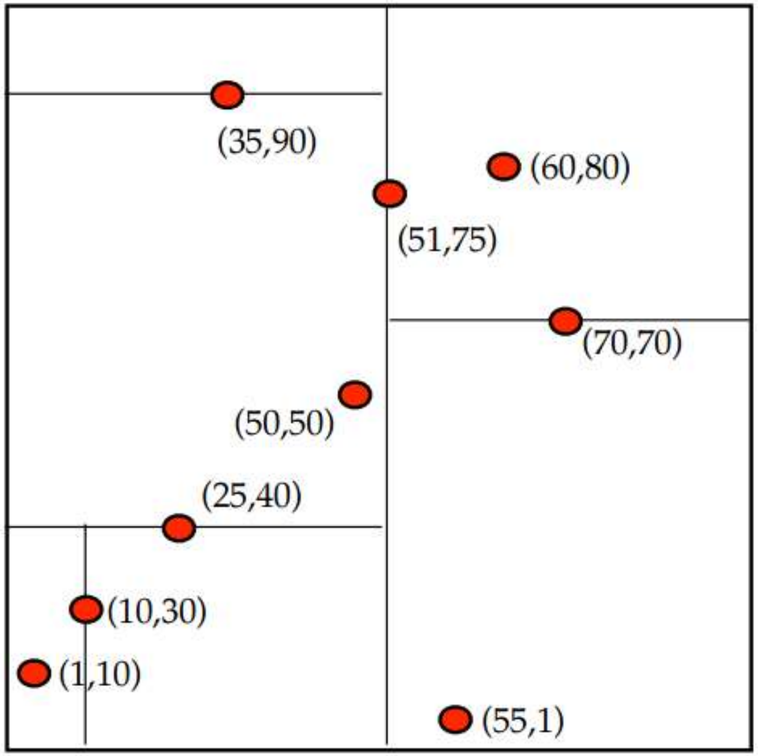
kd-Tree: FindMin

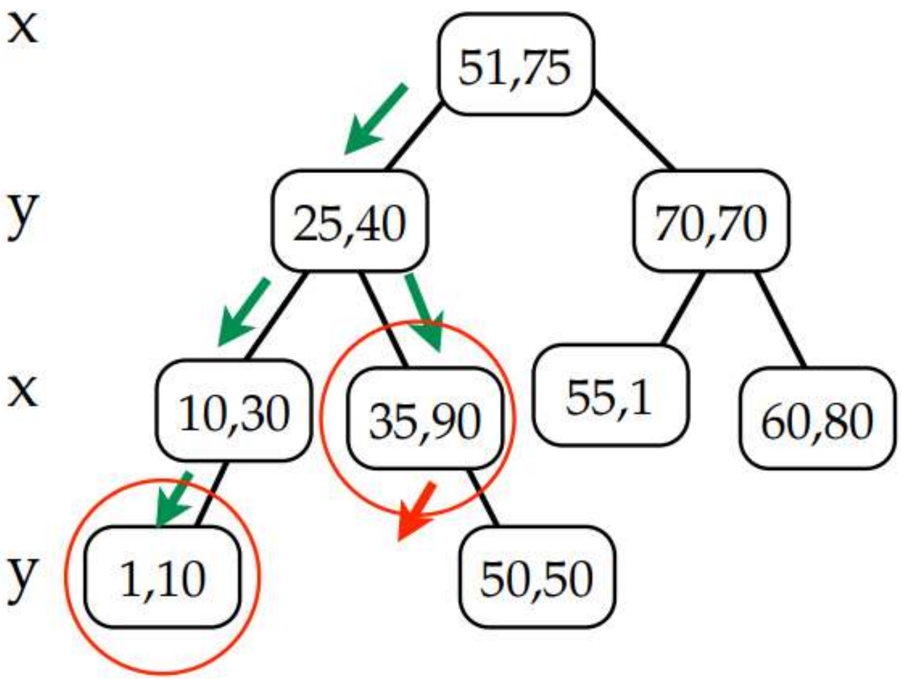
- FindMin(d): find the point with the smallest value in the d th dimension

- Recursively traverse the tree
  - If curr_dim = d, then the minimum can't be in right subtree, so recurse on just the left subtree
  - If curr_dim ≠ d, then the minimum could be in either subtree, so recurse on both subtree.

kd-Tree: FindMin

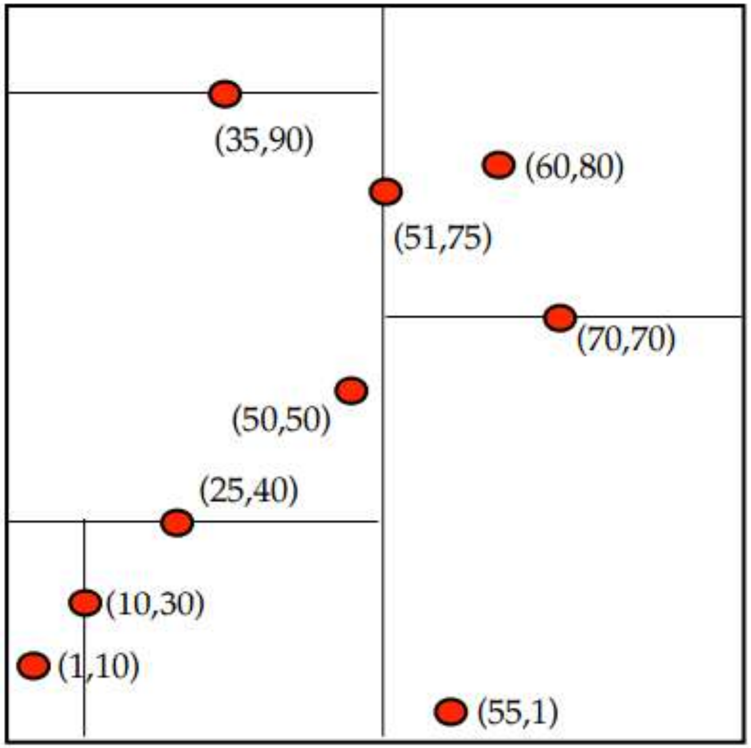
FindMin(x-dimension):

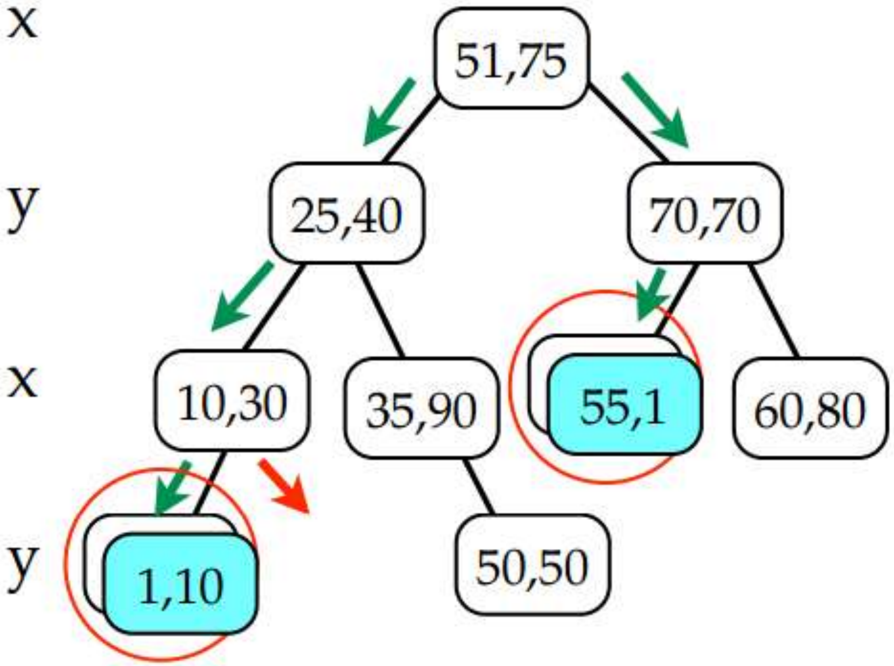




kd-Tree: FindMin

FindMin(y-dimension):





Nearest Neighbor Searching in kd-trees

- Nearest Neighbor Queries: given a point Q find the point P in the data set that is closest to Q

Algorithm:

Input: kd-tree of training data set, point x

Process:

1. Recursive search:

$$Value_{curr\_node}(curr\_\dim) > x(curr\_\dim)\,?\,TurnLeft : TurnRights$$

2. Take final node P as "current nearest neighbor"
3. Back to the upper nodes recursively:

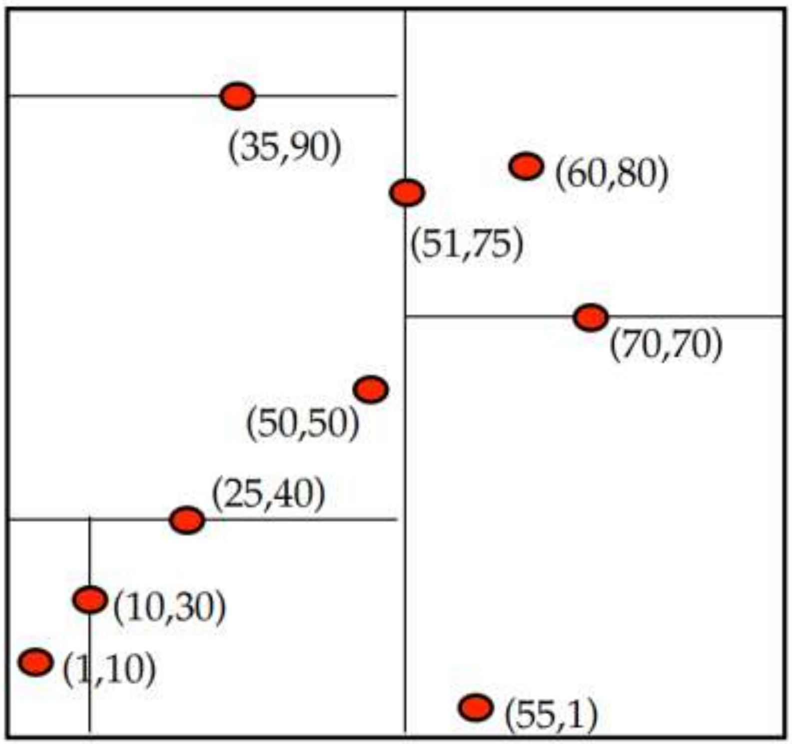
— if brother neighbor of P is closer to x, move to that node, repeat 1,2, 3

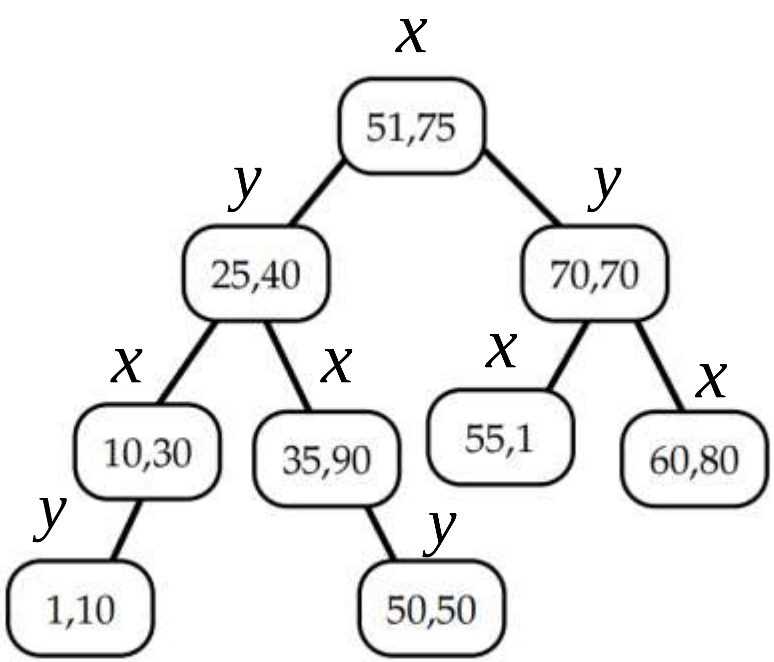
— else, back to father node, till to root node

Output: nearest neighbor of x

Nearest Neighbor Searching in kd-trees

E.g. NN(52, 52):





Space Complexity: $O(n)$ Search Complexity: $O(\log n)$

When

Computation Complexity with kd-trees

Space Complexity: $O(n)$ Search Complexity: $O(\log n)$ (avg)

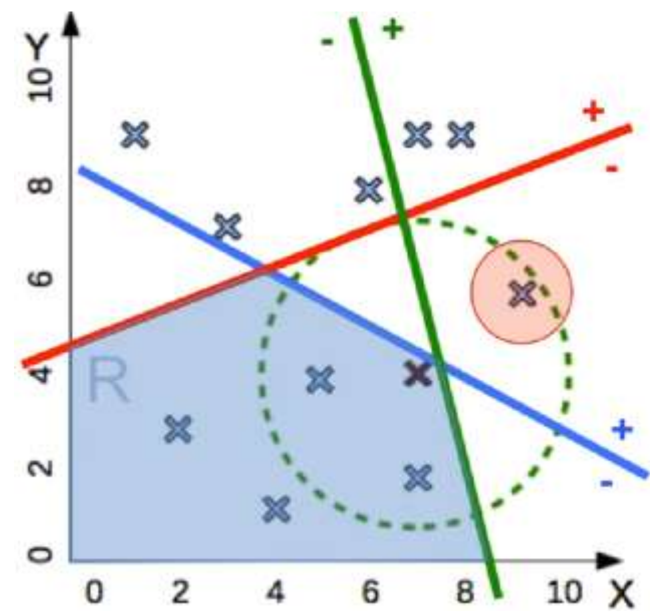
Limitation:

- In high-dimensional spaces, the curse of dimensionality causes the algorithm to need to visit more branches
- In particular, when the number of points is only slightly higher than the number of dimensions, the algorithm is only slightly better than linear search.

Solution: Locality-Sensitive Hashing (LSH)

Locality-Sensitive Hashing (LSH)

- Random hyper-planes $h_1 \dots h_k$
  - space sliced into 2^k regions
  - compare x only to training points in the same region
- Complexity: $O(kd + dn / 2^k) \approx O(d \log n)$
  - $O(kd)$ to find region R, $k \ll n$
    - dot-product x with $h_1 \dots h_k$
  - compare to $n / 2^k$ points in R
    - dot-product x with each point
- Inexact: missed neighbors
  - repeat with different $h_1 \dots h_k$



kNN Summary

- Advantages
  - Very simple and intuitive
  - Good classification if the number of samples is large enough

- Disadvantages
  - choosing best k may be difficult
  - computationally heavy, but improvements possible
  - need large number of samples for accuracy

Thanks!