Machine learning

Non-parametric Methods II

k-nearest-neighbor

Mohammad-Reza A. Dehaqani

dehaqani@ut.ac.ir

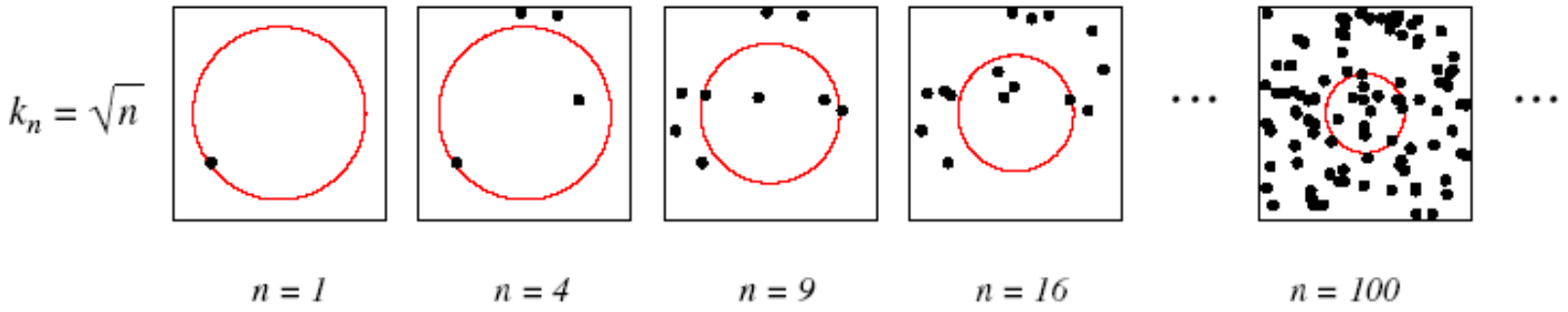
k_n-nearest-neighbor estimation

- The problem of the **unknown** "best" window function and **sparse** data set
- Fix k_n and allow V_n to vary:

$$p_n(\mathbf{x}) \cong \frac{k_n / n}{V_n}$$

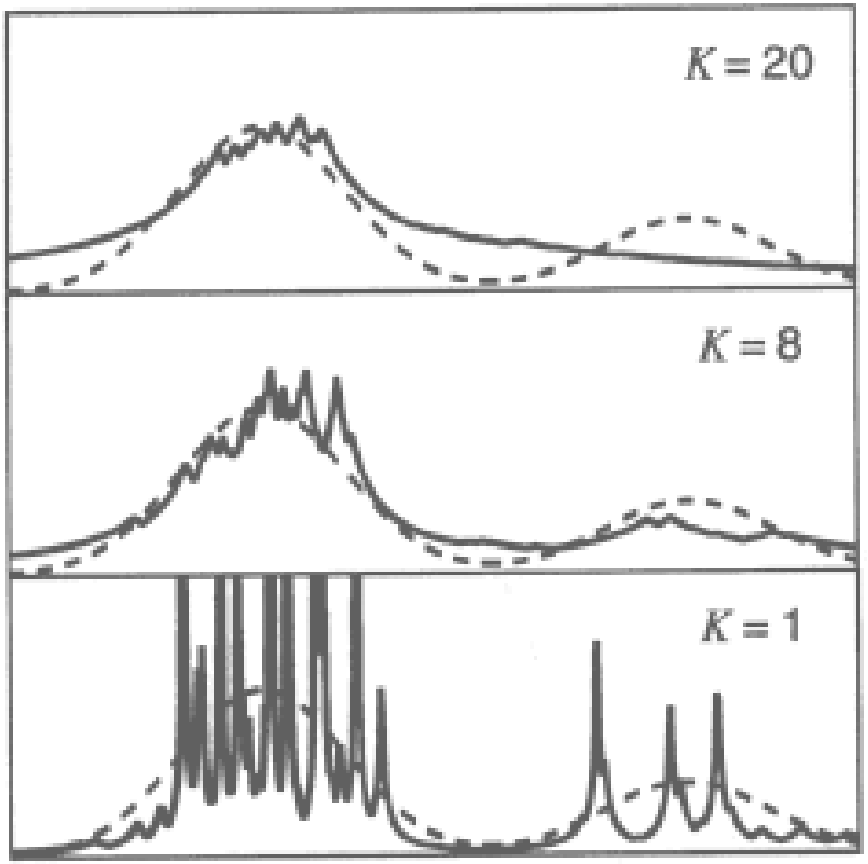
  - Consider a **hypersphere** around **x.**
  - Allow the radius of the hypersphere to **grow** until it contains k_n data points.
  - V_n is determined by the **volume** of the hypersphere.

size depends on density

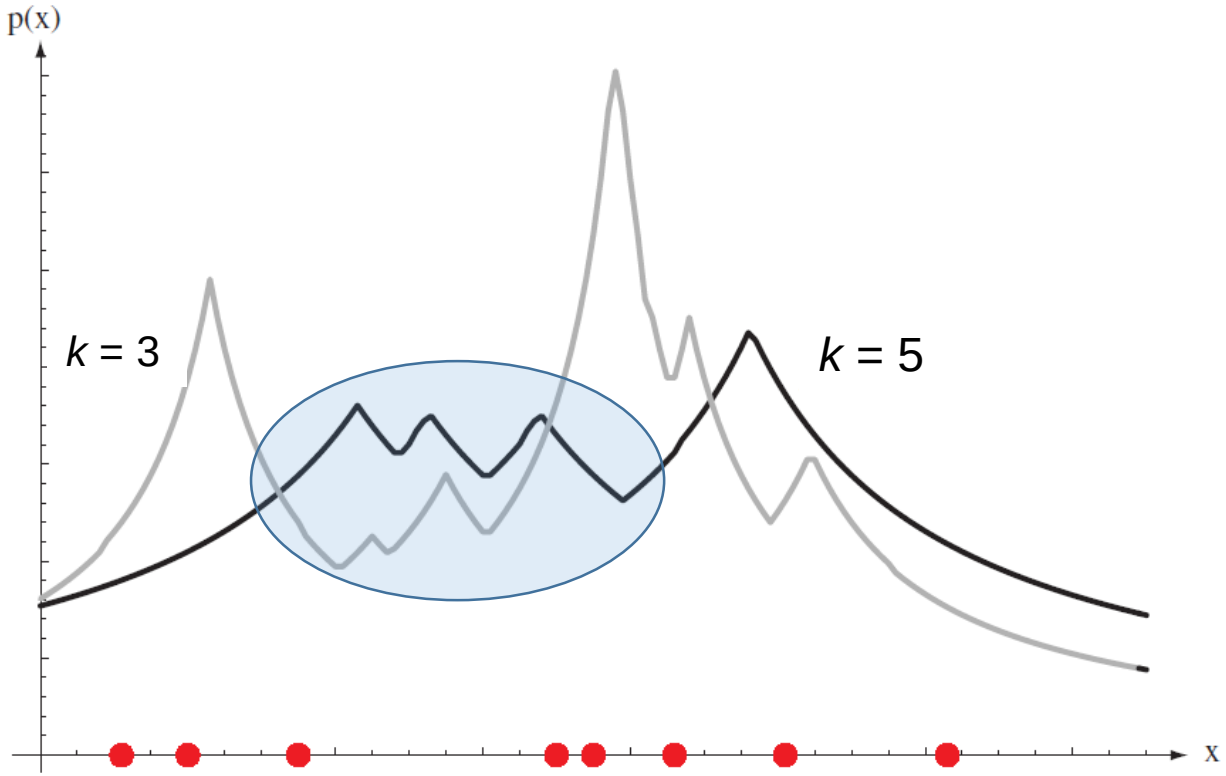


Parameter k_n

- The parameter k_n acts as a **smoothing** parameter and **needs to be optimized**

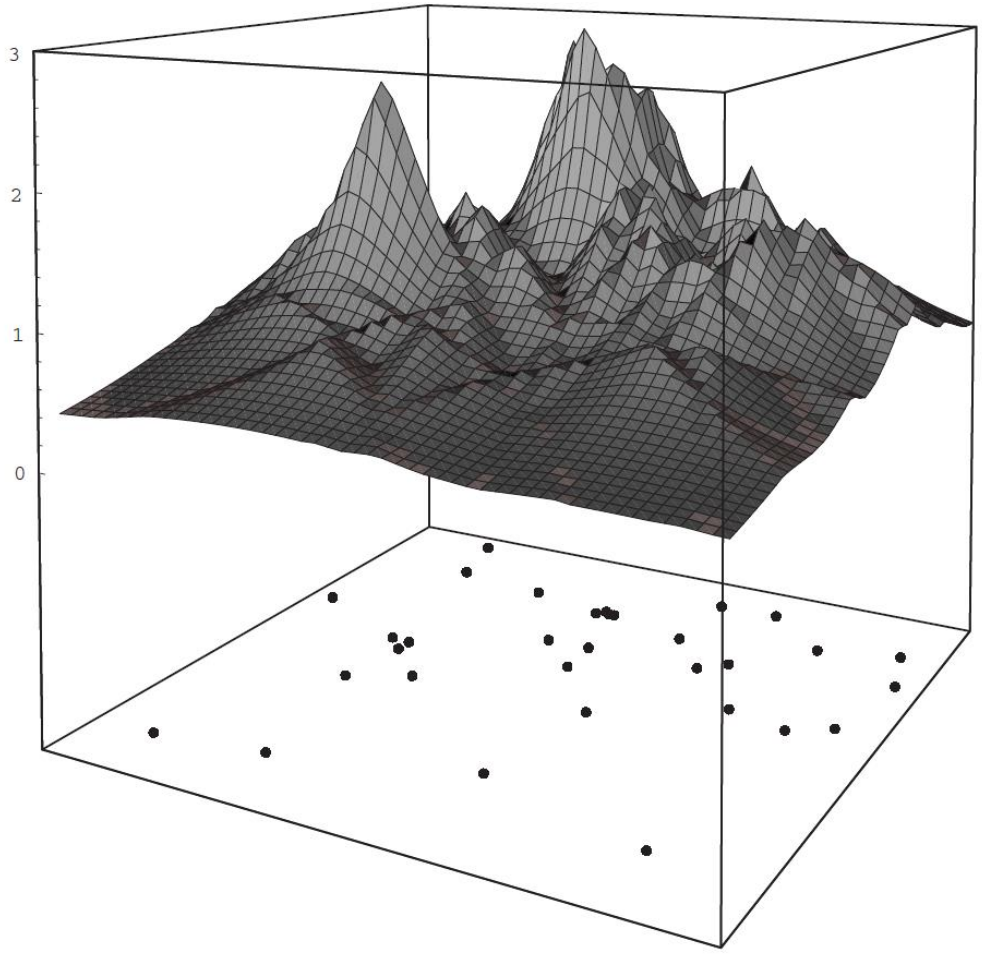


One dimension and the *k*-nearest-neighbor density estimates



Discontinuities in the slopes in the estimates generally occur ***away* from the** positions of the points themselves

k-nearest-neighbor estimate of a two-dimensional density for $k = 5$.

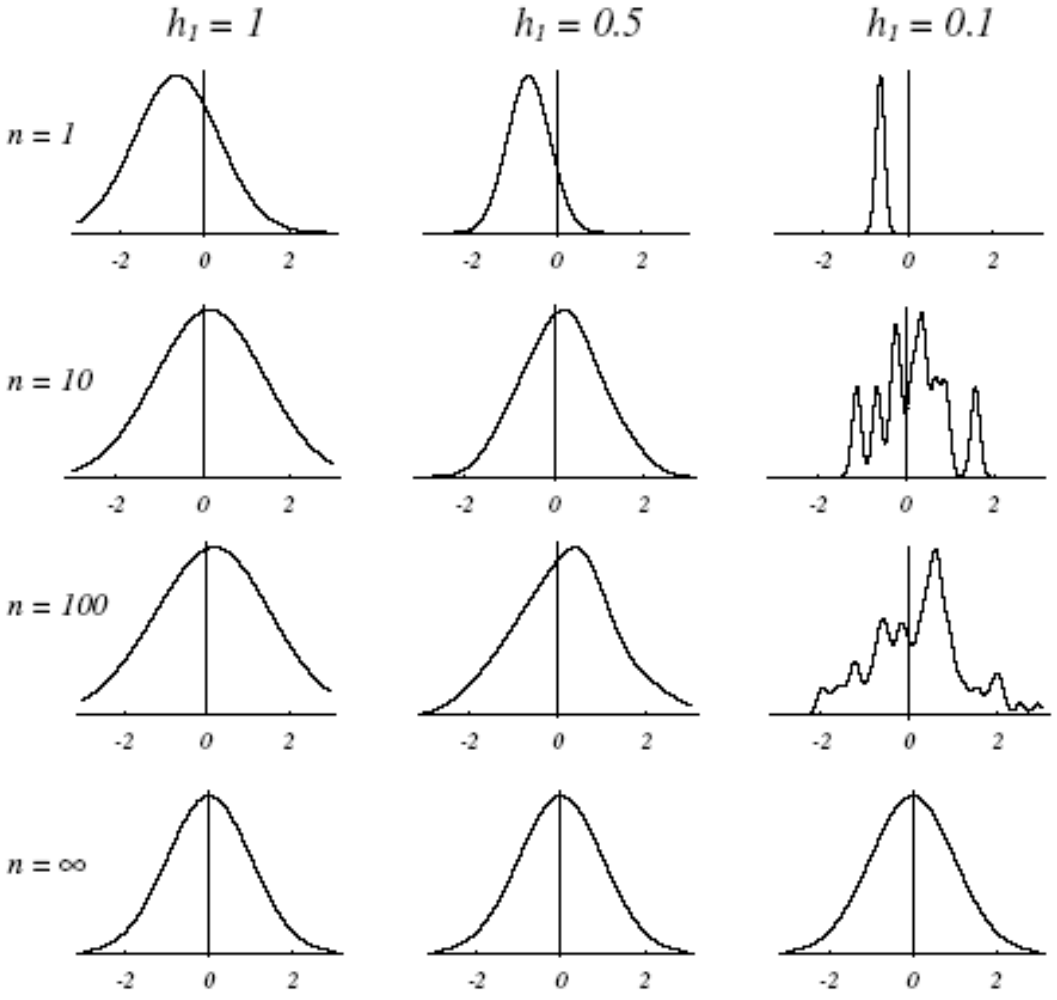


A finite n estimate can be quite "**jagged**," and that

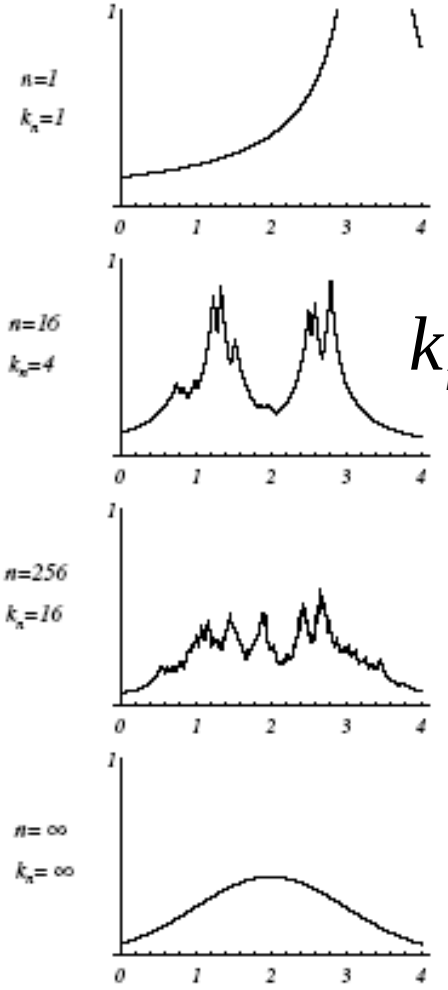
discontinuities in the slopes

Parzen windows vs k_n-nearest-neighbor estimation

Parzen windows



k_n-nearest-neighbor

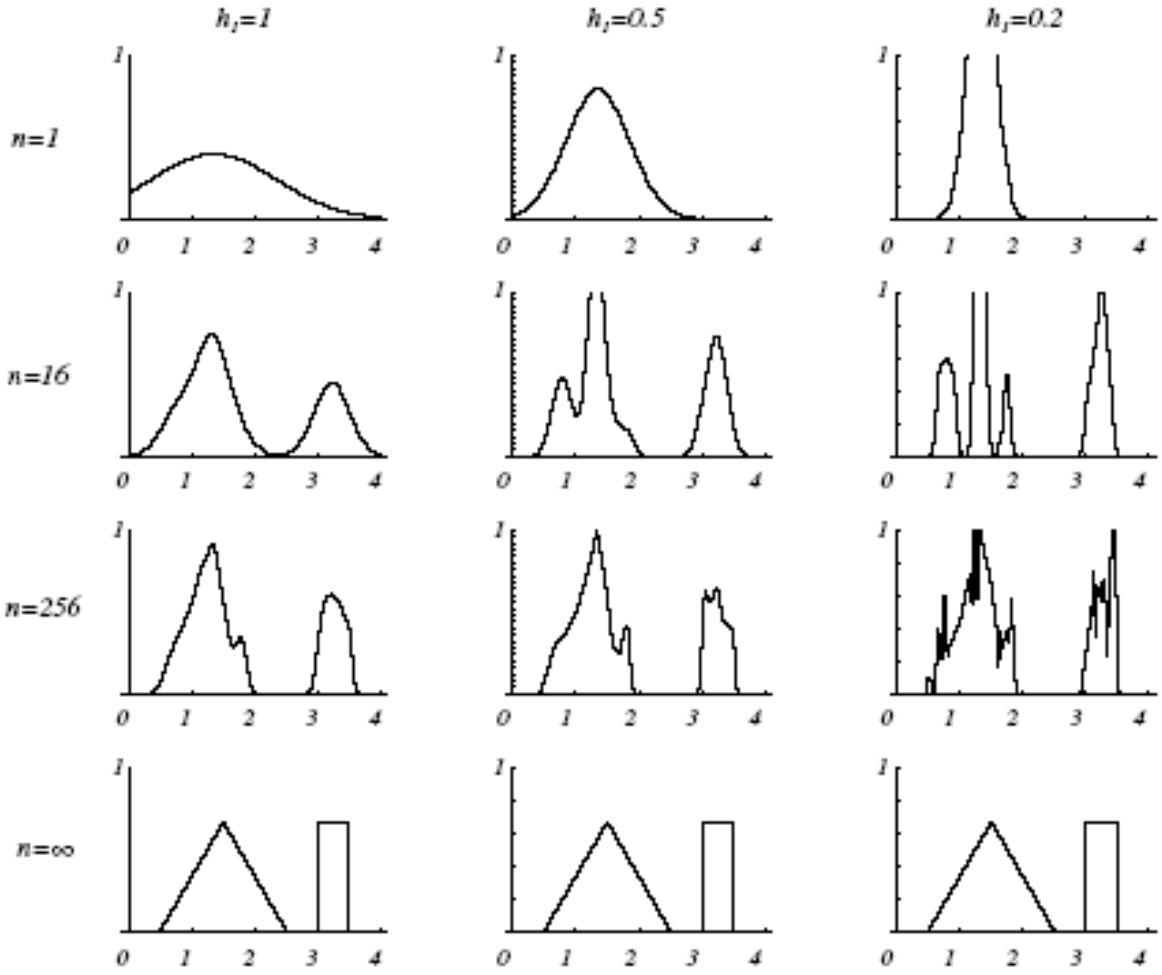


$$k_n = k_1\sqrt{n}$$

the finite n estimates can be quite "spiky.

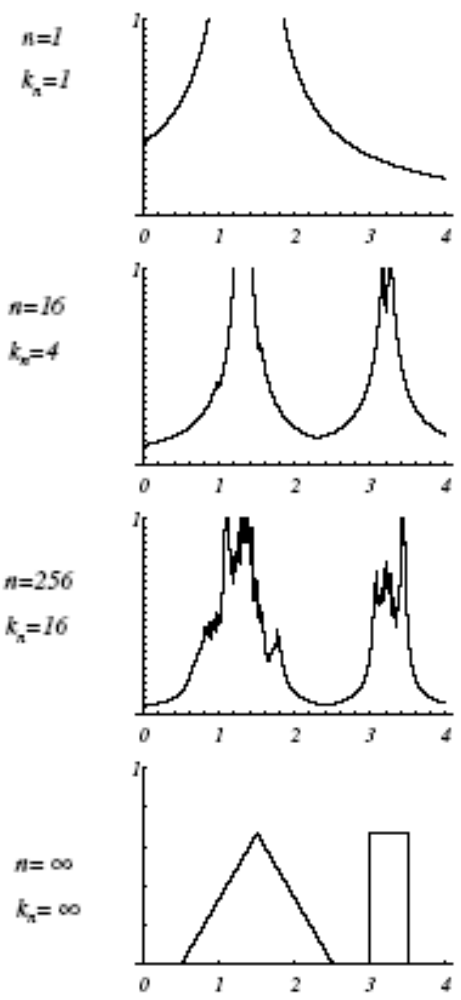
Parzen windows vs k_n-nearest-neighbor estimation

Parzen windows



k_n-nearest-neighbor

$$k_n = k_1\sqrt{n}$$



Estimation of a posteriori probabilities

- Posterior probabilities can be estimated from a set of n **labeled samples** and can be used with the **Bayesian decision rule** for classification.
- Suppose that a **volume** V around x **includes** k **samples**, k_i of which **are labeled** as belonging to class ω_i.
- Then the obvious estimate for the joint probability $p(\mathbf{x}, \omega_i)$ is

$$P(\omega_i) = \frac{n_i}{n} \quad P_n(\boldsymbol{x}|\omega_i) = \frac{k_i}{n_i V} \qquad p_n(\mathbf{x}, \omega_i) = \frac{k_i/n}{V},$$

- a reasonable estimate for $P(\omega_i|\mathbf{x})$ is

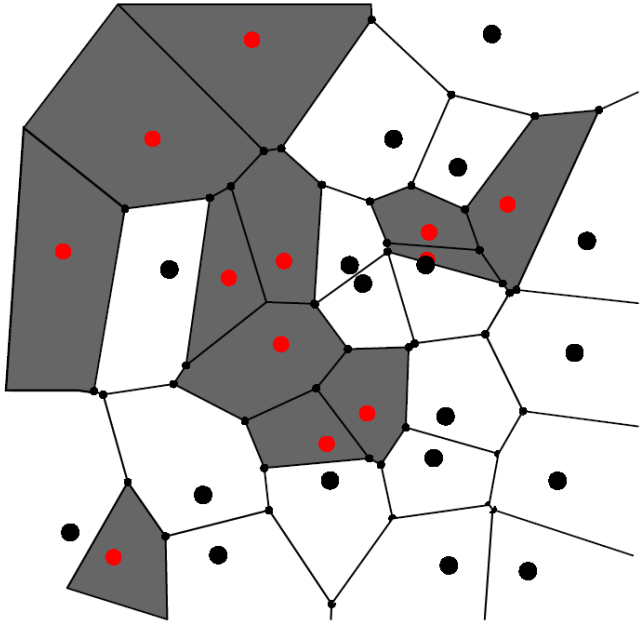
$$P_n(\omega_i|\mathbf{x}) = \frac{p_n(\mathbf{x}, \omega_i)}{\sum_{j=1}^{c} p_n(\mathbf{x}, \omega_j)} = \frac{k_i}{k}.$$

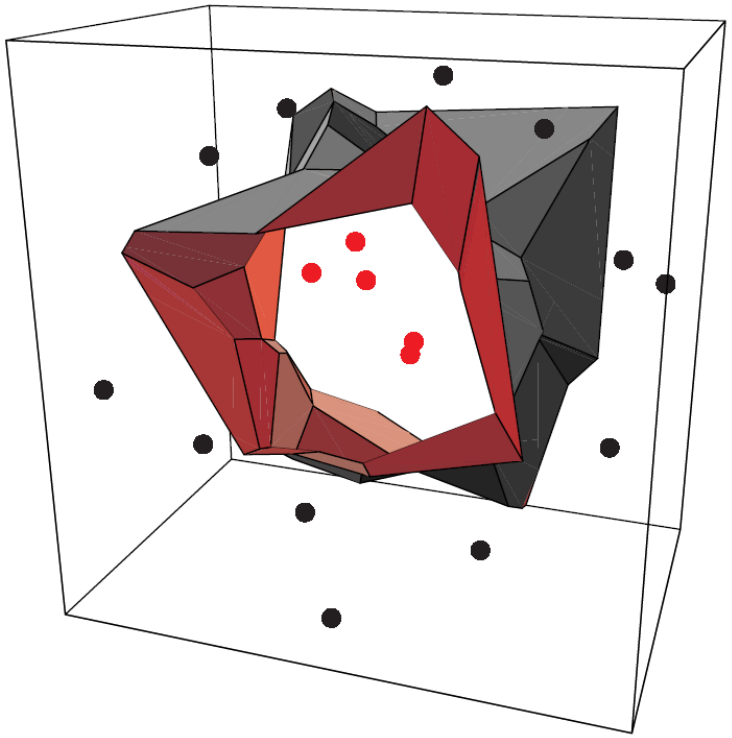
Nearest-neighbor rule

- Given a data point **x**, find a **hypersphere** around it that **contains k points** and assign **x** to the class having the largest number of representatives inside the hypersphere.
- When k=1, we get the **nearest-neighbor** rule.
  - Suppose we have $D^n=\{x_1, ..., x_n\}$ labeled training samples (i.e., known classes).
  - Let x' in D^n be the **closest** point to x, which needs to be classified.
  - The nearest neighbor rule is to assign x the **class associated with** x'.

Partition the feature space; Voronoi tesselation

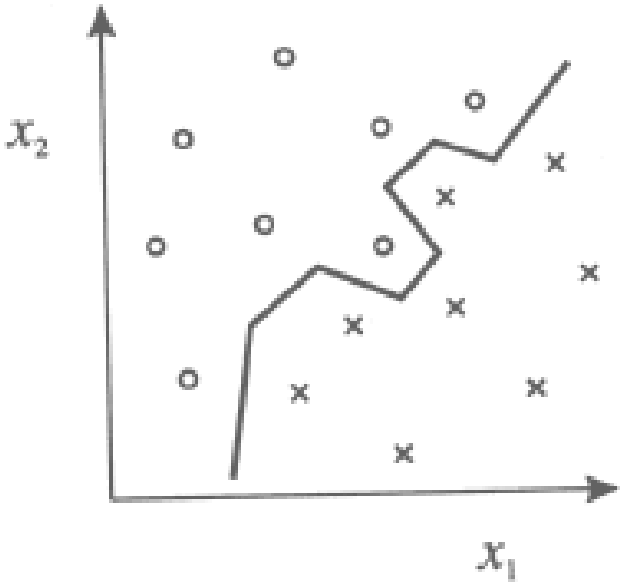
- This rule allows us to **partition the feature space** into cells consisting of all points closer to a given training point **x** than to any other training points.
- All points in such a cell are thus **labelled by** the **category of the training point** — a so-called ***Voronoi tesselation*** of the space





Decision boundary for nearest-neighbor rule

- The decision boundary is **piece-wise linear**.
- Each line segment corresponds to the **perpendicular bisector** of two points belonging to **different classes**.



Error bounds (nearest-neighbor rule)

- The *nearest-neighbor rule* for classifying **x** is to assign it the label **associated with x'**
- The label θ associated with the nearest neighbor is **a random variable**, and the probability that $\theta = \omega_i$ **is merely the a posteriori probability** $P(\omega_i|\mathbf{x'})$.
- When **the number of samples is very large**, it is reasonable to assume that **x is sufficiently close to x** that $P(\omega_i|\mathbf{x}) \approx P(\omega_i|\mathbf{x'})$.
- We define $\omega_m(\mathbf{x})$ by

$$P(\omega_m|\mathbf{x}) = \max_i P(\omega_i|\mathbf{x}')$$

- The **Bayes decision** rule **always** selects ω_m.

Error bounds (nearest-neighbor rule)

- When $P(\omega_m|\mathbf{x})$ is **close to unity**, the nearest-neighbor selection is almost always the **same as the Bayes selection**.
- When $P(\omega_m/\mathbf{x})$ is **close to $1/c$**, so that all classes are essentially equally likely, the probability of error is approximately $1 - 1/c$ for both NN and Bayse.
- The average probability of error

$$P(e) = \int P(e|\mathbf{x})p(\mathbf{x})\ d\mathbf{x}.$$

- If we let $P^*(e|\mathbf{x})$ be the **minimum possible value** of $P(e|\mathbf{x})$, and P^* be the minimum possible value of $P(e)$

$$P^*(e|\mathbf{x}) = 1 - P(\omega_m|\mathbf{x})$$

$$P^* = \int P^*(e|\mathbf{x})p(\mathbf{x})\ d\mathbf{x}.$$

Error bounds (nearest-neighbor rule)

- The decision rule **depends** on the nearest-neighbor

$$P(e|\mathbf{x}) = \int P(e|\mathbf{x}, \mathbf{x}')p(\mathbf{x}'|\mathbf{x})\, d\mathbf{x}'.$$

- Suppose that selected pair is (**x**,θ), and that **x'**$_j$, labelled θ'$_j$, is the training sample nearest **x**. We commit an error whenever $\theta \neq \theta'_j$.

$$P_n(e|\mathbf{x}, \mathbf{x}'_j) = 1 - \sum_{i=1}^{c} P(\theta = \omega_i, \theta' = \omega_i|\mathbf{x}, \mathbf{x}'_j)$$

x and **x'**$_j$ are **independent**

$$= 1 - \sum_{i=1}^{c} P(\omega_i|\mathbf{x})P(\omega_i|\mathbf{x}'_j).$$

- n goes to **infinity** and p(**x'**|**x**) approaches a **delta function**.

$$\lim_{n\to\infty} P_n(e|\mathbf{x}) = \int \left[1 - \sum_{i=1}^{c} P(\omega_i|\mathbf{x})P(\omega_i|\mathbf{x}')\right]\delta(\mathbf{x}' - \mathbf{x})\, d\mathbf{x}'$$

$$= 1 - \sum_{i=1}^{c} P^2(\omega_i|\mathbf{x}).$$

$$
\begin{aligned}
P &= \lim_{n\to\infty} P_n(e) \\
&= \lim_{n\to\infty} \int P_n(e|\mathbf{x})p(\mathbf{x})\, d\mathbf{x} \\
&= \int \left[1 - \sum_{i=1}^{c} P^2(\omega_i|\mathbf{x})\right] p(\mathbf{x})\, d\mathbf{x}.
\end{aligned}
$$

- Let P* be the **minimum possible error**, which is given by the minimum error rate classifier.
- An obvious lower bound **on P is P*** itself
- If the Bayes errore rate is low, **$P(\omega_i|\mathbf{x})$ is near 1.0 for** some *i*, say $i = m$.

$$
1 - P^2(\omega_m|\mathbf{x}) \simeq 2(1 - P(\omega_m|\mathbf{x}))
$$

$$
P^*(e|\mathbf{x}) = 1 - P(\omega_m|\mathbf{x}),
$$

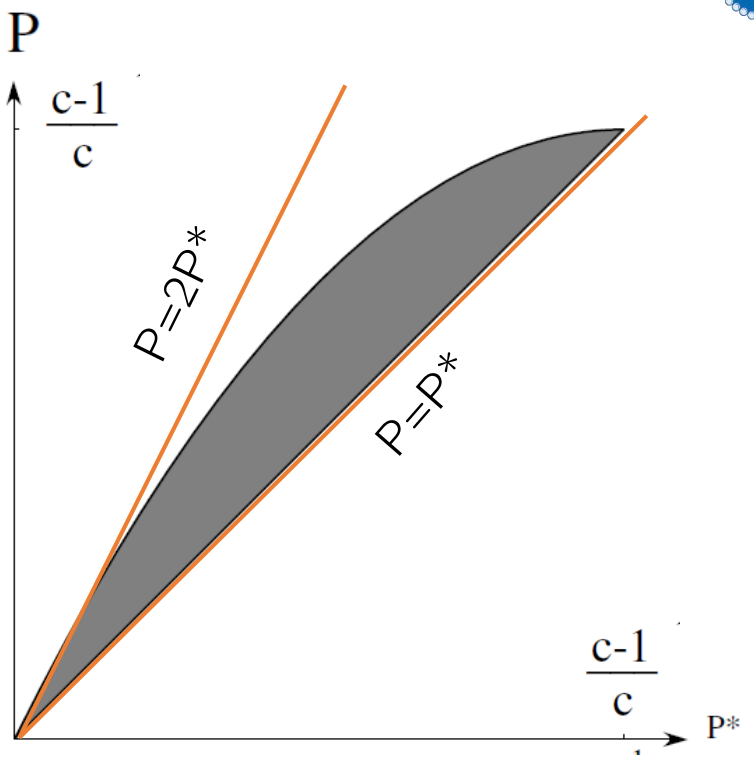
$$
f(x) \approx f(a) + f'(a)(x-a), \qquad \text{for } x \approx a\,.
$$

- **Upper bound**: integration over **x** might **yield about twice** the Bayes rate

$$
P(e) = \int P(e|\mathbf{x})p(\mathbf{x})\, d\mathbf{x}.
$$

Exact upper bound

- Let P^* be **the minimum possible error**, which is given by the minimum error rate classifier.
- Let P be the error given by the nearest neighbor rule.
- Given unlimited number of training data and ***c*-category problem**, it can be shown that:

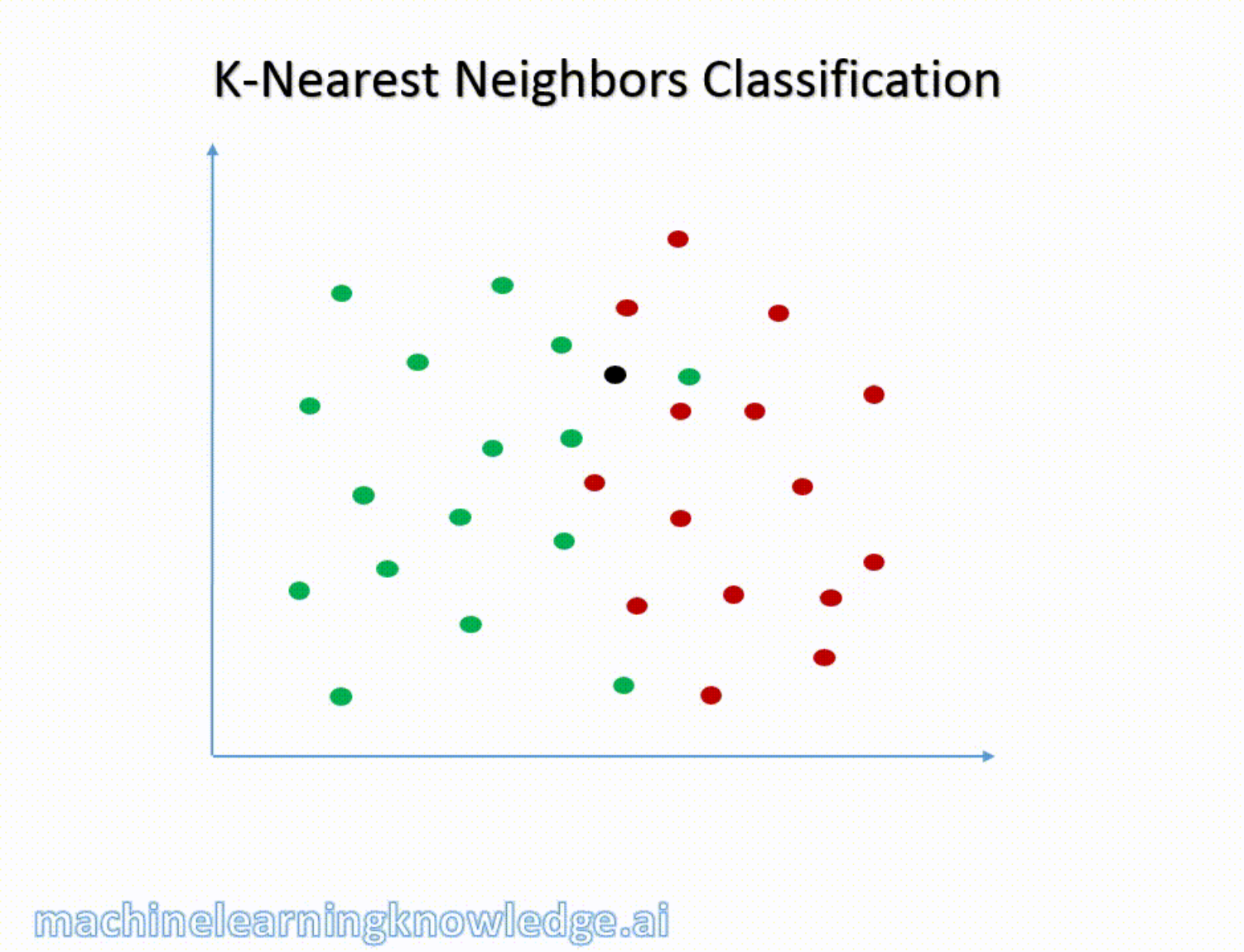


Maximum error:

1 – 1/c = (c-1)/c

$$P^* \leq P \leq P^*\left(2 - \frac{c}{c-1}P^*\right) \leq 2P^*$$

The *k*-Nearest-Neighbor Rule

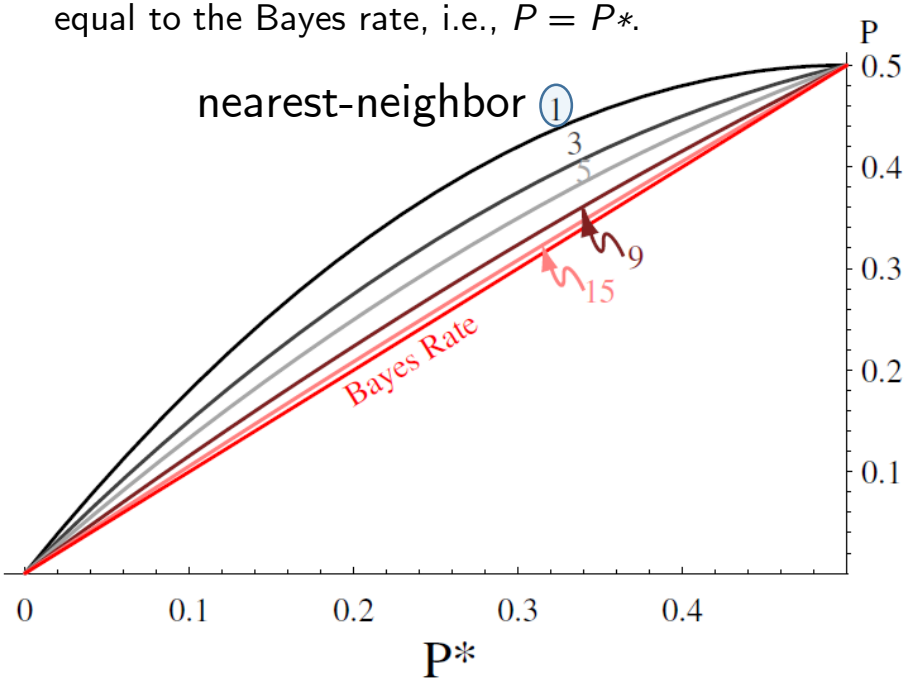


Error rate bound

- The error-rate for the k-nearest-neighbor rule for a **two-category** problem is bounded **above** by $C_k(P*)$
- Where $C_k(P*)$ is defined to be the smallest **concave** function of $P*$ greater than

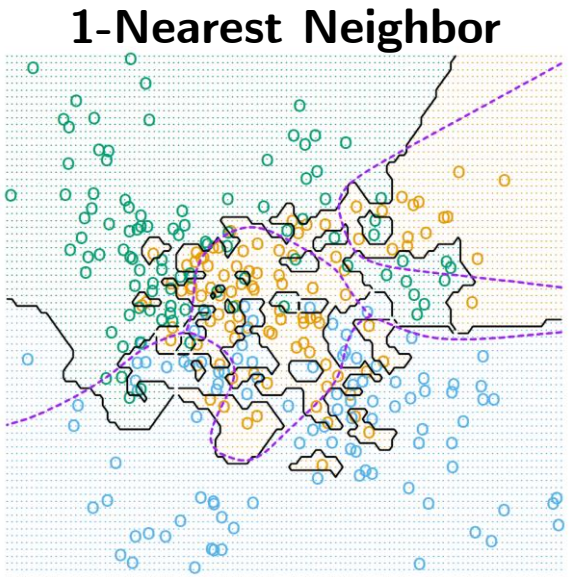
Each curve is **labelled** by k;

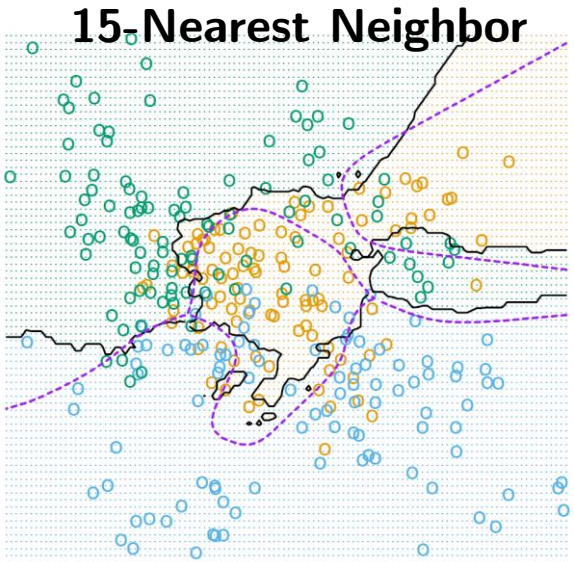
When $k = \infty$, the **estimated probabilities match the true probabilities** and thus the error rate is equal to the Bayes rate, i.e., $P = P*$.

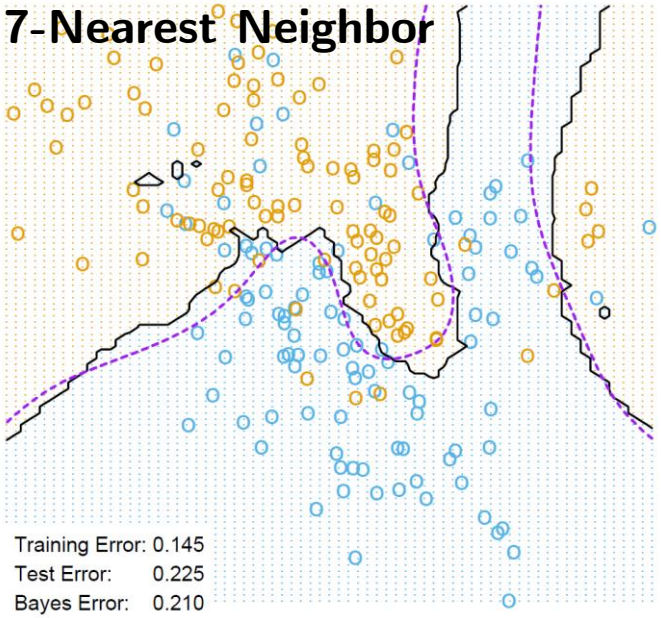


$$\sum_{i=0}^{(k-1)/2} \binom{k}{i} \left[(P^*)^{i+1}(1-P^*)^{k-i} + (P^*)^{k-i}(1-P^*)^{i+1} \right].$$

Decision boundary (bias variance trade off)







Computational Complexity of the *k*–Nearest-Neighbor Rule

- Assuming n training examples in d dimensions, a straightforward implementation would take $O(dn^2)$
  - distance calculation is $O(d)$,
- A **parallel** implementation would take O(1)
- Three generic approaches for **reducing** computational complexity:
  - Computing **partial** distances
  - **Pre-structuring** (e.g., search tree)
  - **Editing** the stored prototypes

Partial distances

- Compute distance using **first *r* dimensions** only:

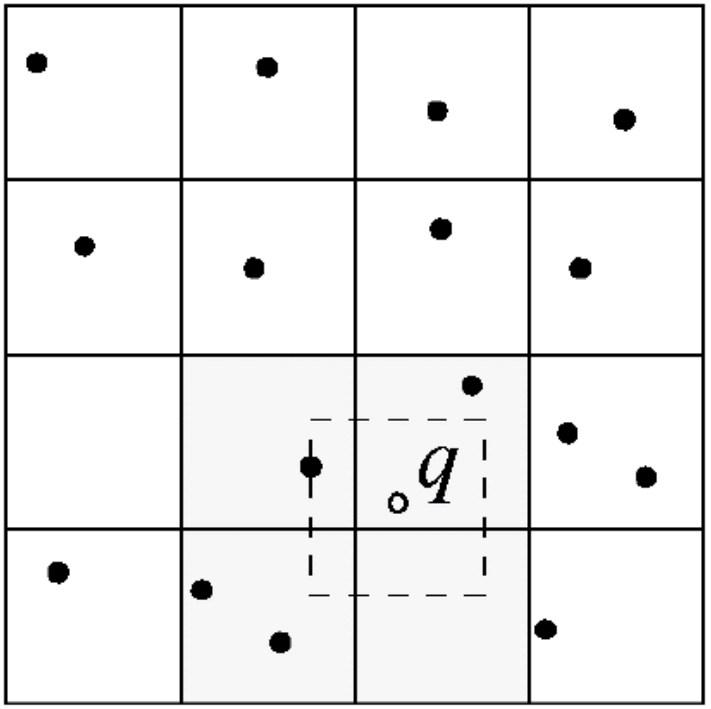
$$D_r(\mathbf{x}, \mathbf{x}') = \left(\sum_{k=1}^{r} (x_k - x'_k)^2\right)^{1/2}$$

where $r < d$.

- If the partial distance is **too great** (i.e., **greater than the distance of x to current closest prototype**), there is **no reason** to compute additional terms.

Pre-structuring: Bucketing

- In the Bucketing algorithm, the space is **divided** into **identical cells**.
  - For each cell the data points inside it are **stored in a list**.
  - Given a test point **x**, find the **cell that contains** it.
  - **Search** only the points **inside that cell**!
  - Does **not guarantee** to find the true nearest neighbor(s)
  - Speed vs accuracy **tradeoff**



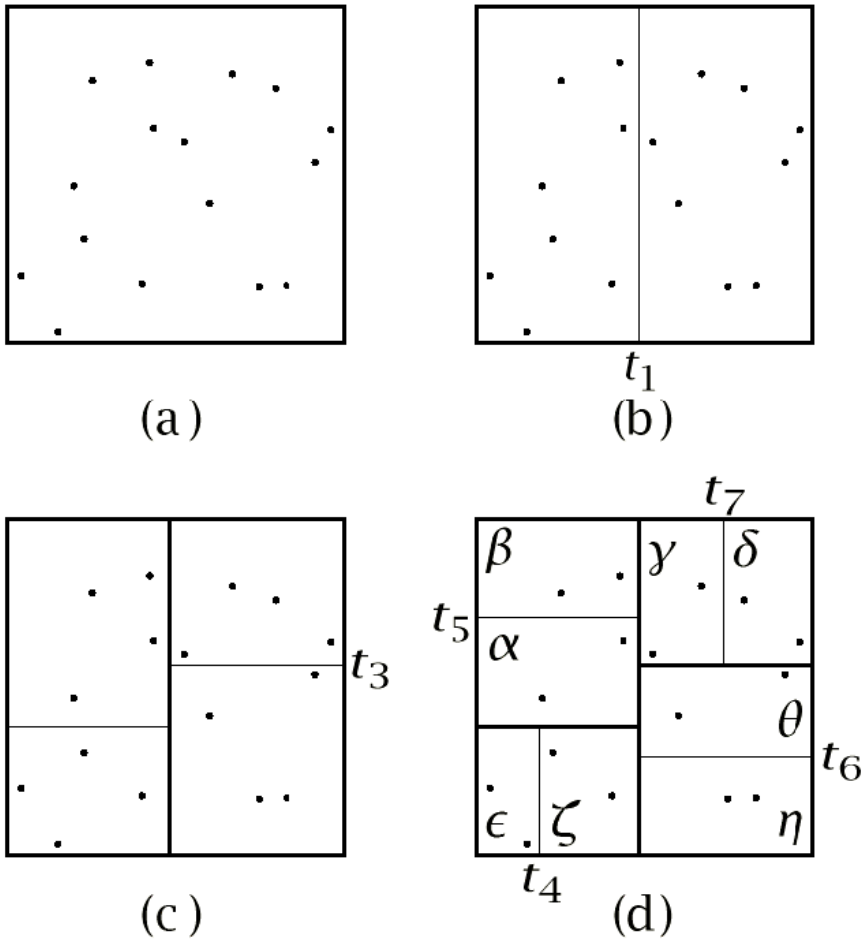
Pre-structuring: Search Trees (k-d tree: short for k-dimensional **tree**)

- A k-d tree is a **data structure** for **storing** a finite set of points from a k-dimensional space.
- Generalization of **binary search** ...
- **Goal: hierarchically decompose** space into a relatively small number of cells such that no **cell contains too many points**.

How to build it

- Each internal node in a k-d tree is associated with a hyper-rectangle and a hyper-plane orthogonal to one of the coordinate axis.
  - The hyper-plane splits the hyper-rectangle into two parts, which are **associated with the child** nodes.
  - The **partitioning process goes** on until the number of data points in the hyper-rectangle falls **below** some given **threshold**

k-d tree

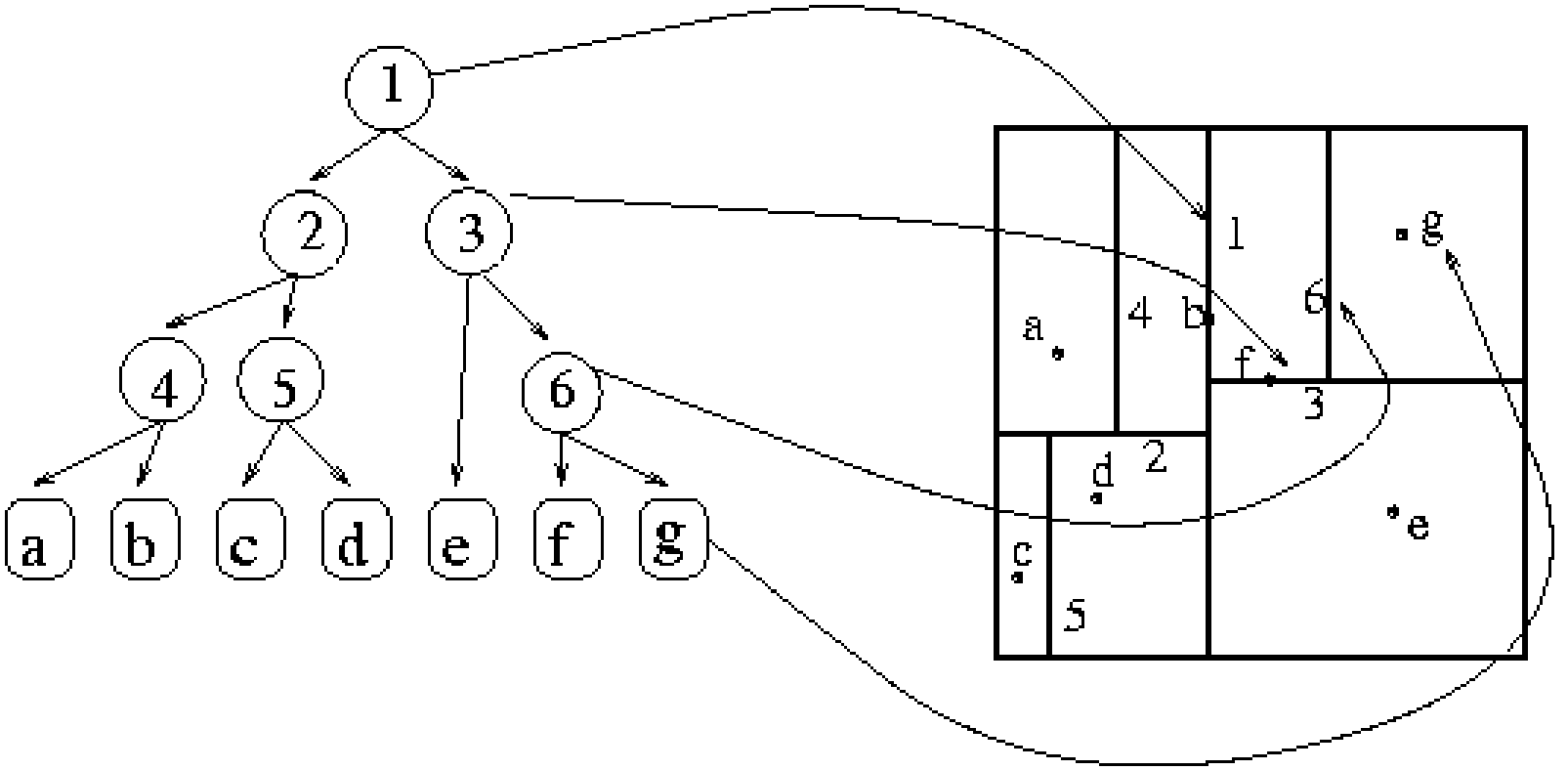

(a)
t_1
(b)
t_2
t_3
(c)
t_7
t_5
β
γ
δ
α
θ
t_6
ϵ
ζ
η
t_4
(d)
$x_1 < t_1$
$x_1 \geqslant t_1$
$x_2 < t_2$
$x_2 \geqslant t_2$
$x_2 < t_3$
$x_2 \geqslant t_3$
$x_1 < t_4$
$x_1 \geqslant t_4$
$x_2 < t_5$
$x_2 \geqslant t_5$
$x_2 < t_6$
$x_2 \geqslant t_6$
$x_1 < t_7$
$x_1 \geqslant t_7$
α β ϵ ζ γ δ η θ
(e)

k-d tree;

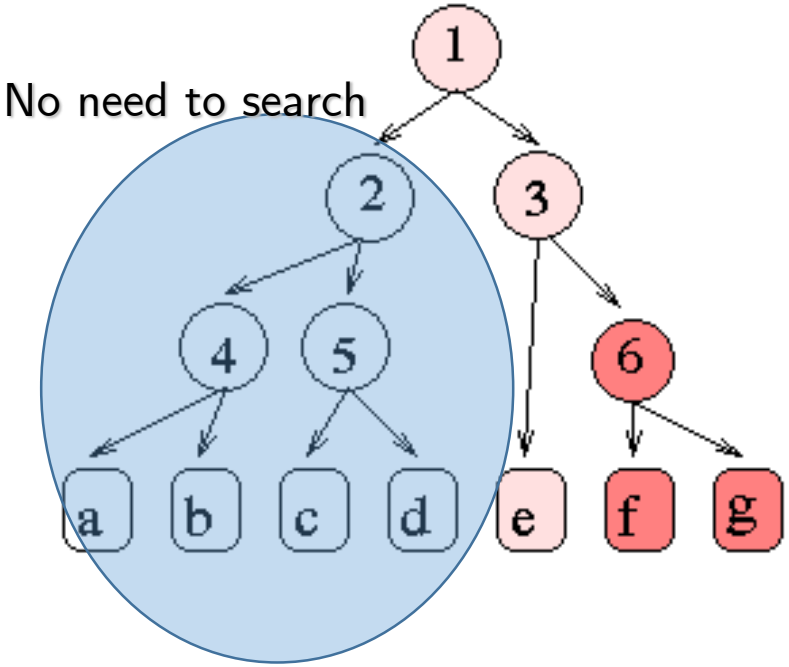
O(n) storage and O(n log n) construction time;

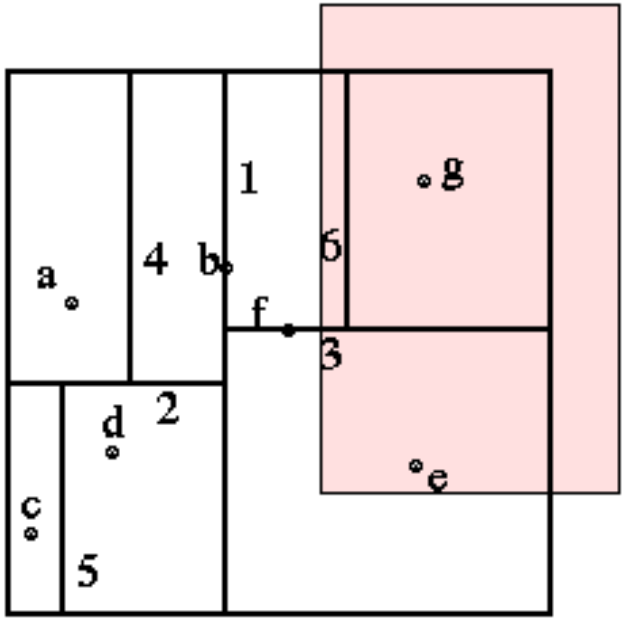
Good for massive low dimensional data:

KNN with k-d tree is o(d logn)



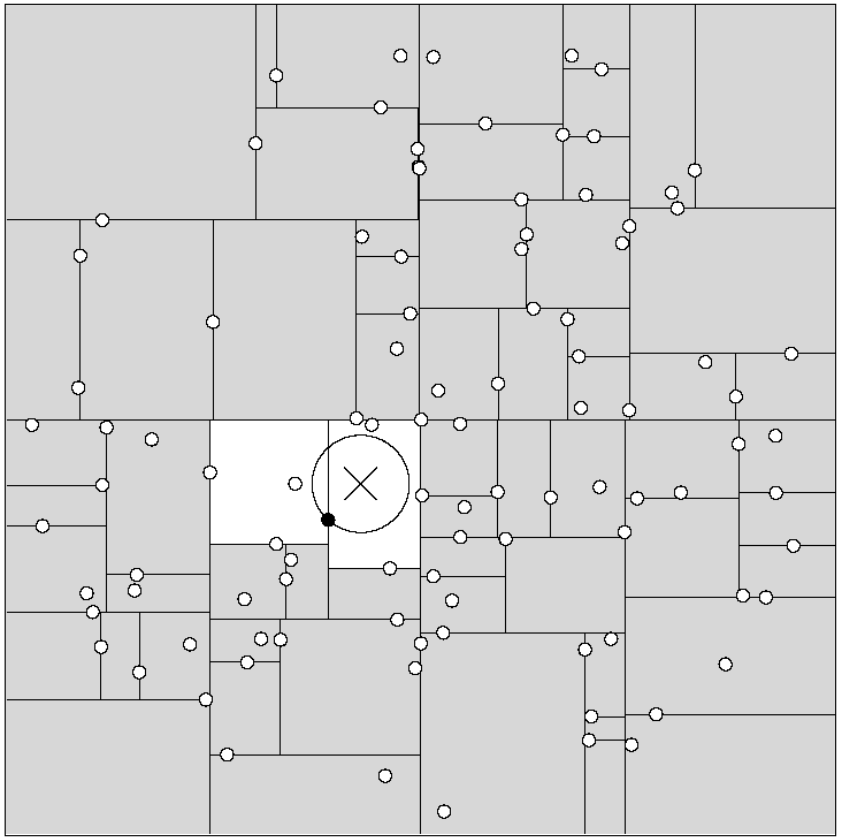
k-d tree; Finding the set of points within a window



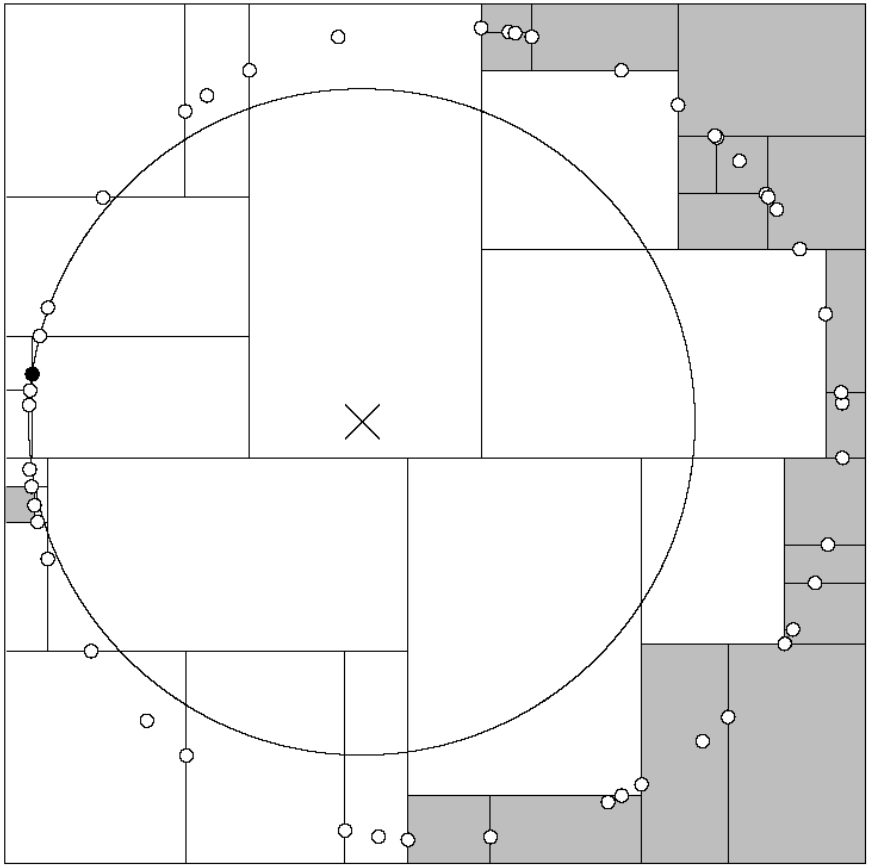


Nearest neighbor search in Kd tree

Usually a few leaf node



Bad distribution, force to search almost all nodes

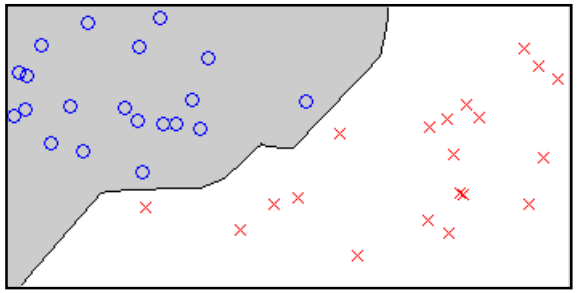


Editing

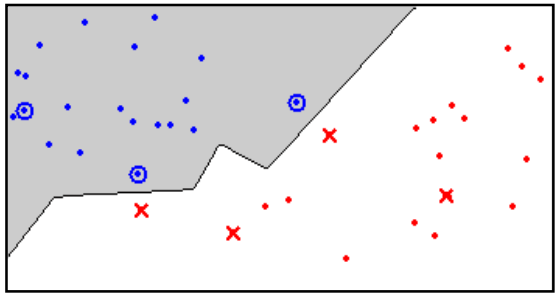
- **Goal: reduce** the number of training samples.
- Two main approaches:
  - **Condensing: preserve** decision boundaries.
  - **Pruning: eliminate** noisy examples to produce **smoother** boundaries and improve accuracy.

Editing using condensing

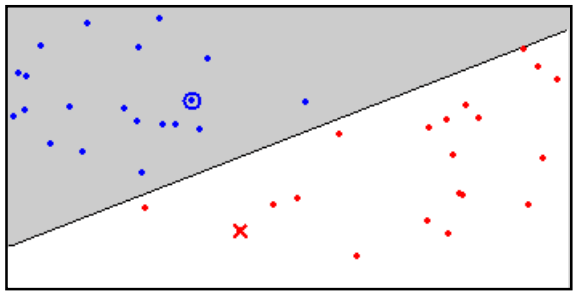
- **Retain** only the samples that are **needed** to define the decision boundary.
- **Decision Boundary Consistent** – a subset whose nearest neighbour decision boundary is **close to the boundary** of the entire training set.
- **Minimum Consistent Set** – the **smallest subset** of the training data that **correctly classifies all** of the original training data.



Original data



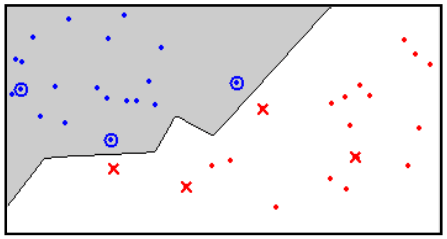
Condensed data

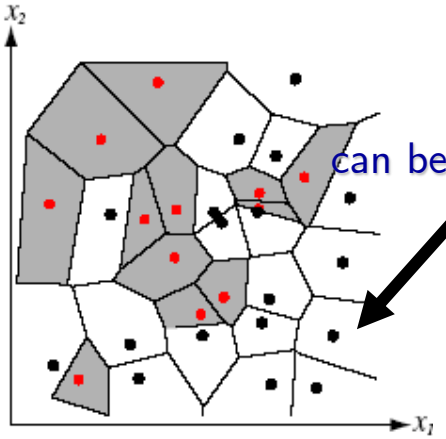


Minimum Consistent Set

Editing; Keep points contributing to the boundary

- i.e.,
  - At least **one neighbor** belongs to a **different** category.
  - Eliminate prototypes that are **surrounded by samples of the same category.**



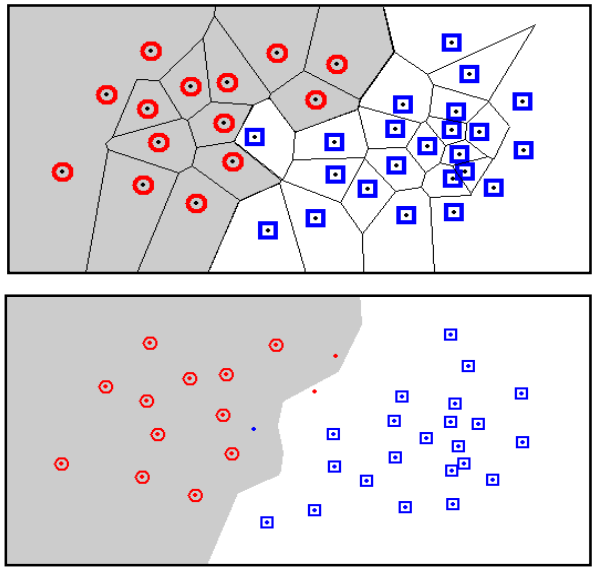



```
1 begin initialize j = 0, D = data set, n = #prototypes
2         construct the full Voronoi diagram of D
3         do j ← j + 1;  for each prototype x'_j
4             Find the Voronoi neighbors of x'_j
5             if any neighbor is not from the same class as x'_j then mark x'_j
6         until j = n
7   Discard all points that are not marked
8   Construct the Voronoi diagram of the remaining (marked) prototypes
9 end
```

Editing using pruning

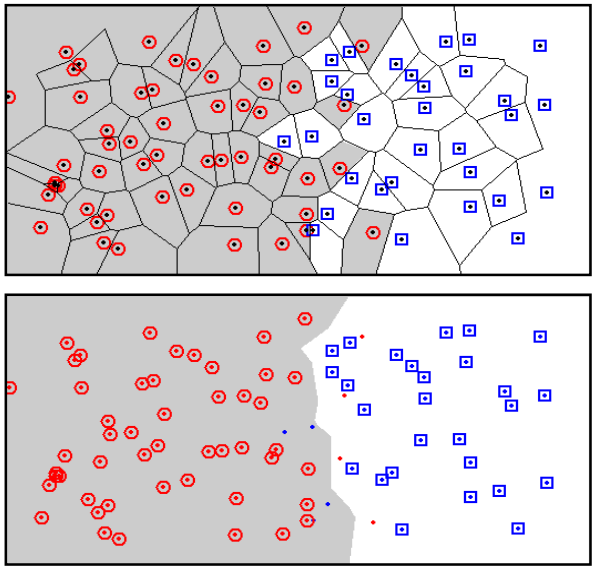
- Pruning seeks to **remove "noisy" points** and produces **smooth** decision boundaries.
- Often, it **retains points far from** the decision boundaries.
- **Wilson pruning**: remove points that **do not agree** with the **majority of** their *k*-nearest-neighbours.

Original data



Wilson editing with k=7

Original data



Wilson editing with k=7

Nearest Neighbor Embedding

- Map the training examples to a **low dimensional** space such that **distances between training examples are preserved as much as possible**.
  - i.e., **reduce *d*** and at the same time **keep all the nearest neighbors** in the original space.

General comments (nearest-neighbor classifier)

- The nearest neighbor classifier provides a **powerful** tool.
- Its **error is bounded** to be at most **two times** of the Bayes error (in the limiting case).
- It is **easy** to implement and understand. No optimization or training required.
- It can be implemented **efficiently**.
- Its performance, however, relies on the **metric** used to compute distances!

Properties of distance metrics

non-negativity: $D(\mathbf{a}, \mathbf{b}) \geq 0$

reflexivity: $D(\mathbf{a}, \mathbf{b}) = 0$ if and only if $\mathbf{a} = \mathbf{b}$

symmetry: $D(\mathbf{a}, \mathbf{b}) = D(\mathbf{b}, \mathbf{a})$

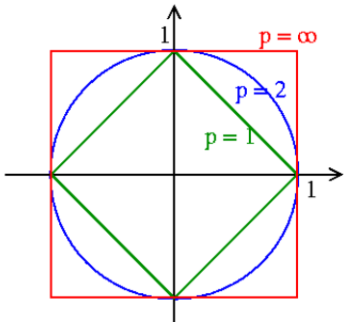
triangle inequality: $D(\mathbf{a}, \mathbf{b}) + D(\mathbf{b}, \mathbf{c}) \geq D(\mathbf{a}, \mathbf{c})$.

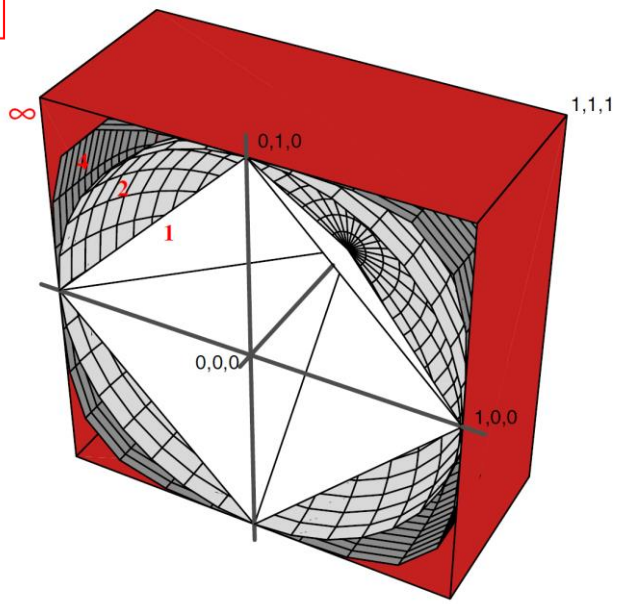
- Euclidean formula for distance in *d* dimensions,

$$D(\mathbf{a}, \mathbf{b}) = \left(\sum_{k=1}^{d} (a_k - b_k)^2 \right)^{1/2},$$

- Results depending upon **rescaling (the importance of invariant metrics)**

Minkowski metric





- Minkowski metric

(L_k norm)

$$L_k(\mathbf{a}, \mathbf{b}) = \left(\sum_{i=1}^{d} |a_i - b_i|^k \right)^{1/k},$$

- L_2 Euclidean distance is the
- L_1 *(Manhattan or city block)*
- L_∞ *(max distance among dimensions)*

$$L_\infty(\mathbf{x}, \mathbf{y}) = \max_{i=1}^{d} |\mathbf{x}_i - \mathbf{y}_i|.$$

Each colored surface consists of **points a distance 1.0 from the origin**, measured using different values for *k* in the Minkowski metric (*k* is printed in red).

Feature Normalization

- When there is **great difference** in the **range** of the data along different axes in a **multidimensional** space, these metrics **implicitly** assign **more weighting** to **features with large ranges** than those with small ranges
- Feature normalization can be used to approximately **equalize ranges of the features** and make them have approximately the **same effect in the distance** computation.

Feature Normalization

- Linear scaling to **unit range**:
  - Given a lower bound l and an upper bound u for a feature **x**∈R,

$$\tilde{x} = \frac{x - l}{u - l}$$

- Linear scaling to **unit variance**:
  - A feature **x**∈R can be transformed to a random variable with zero mean and unit variance as

$$\tilde{x} = \frac{x - \mu}{\sigma}$$

where μ and σ are the sample mean and the sample standard deviation of that feature, respectively

Feature Normalization

- Normalization using the **cumulative distribution function**:
  - Given a random variable **x**∈R with cumulative distribution function $F_x(x)$, the random variable $\tilde{x}$ resulting from the transformation $\tilde{x} = F_X(x)$ will be uniformly distributed in [0, 1].
- **Rank normalization**:
  - Given the sample for a feature as $x_1, \ldots, x_n \in R$, first we find the **order statistics** $x^{(1)}, \ldots, x^{(n)}$ and then replace each pattern's feature value by its corresponding normalized rank as

$$\tilde{x}_i = \frac{\text{rank}_{x_1,\ldots,x_n}(x_i) - 1}{n-1}$$

- where x_i is the feature value for the i'th pattern. This procedure uniformly maps all feature values to the [0, 1] range

General comments on parametric and nonparametric

- Parametric: moments, ML, MAP, HMM, BN, EM
  - Reducing the dimension to a few number of parameter
  - Global view
- Non-parametric (histogram, Parzen,KNN)
  - Local view
  - Need great memory (convey all data)
- **EM is among the best**