Machine Learning

3. Nearest Neighbor and Kernel Methods

Steffen Rendle

(Slides mainly by Lars Schmidt-Thieme)

Information Systems and Machine Learning Lab (ISMLL)
Institute for Business Economics and Information Systems
& Institute for Computer Science
University of Hildesheim
http://www.ismll.uni-hildesheim.de

1. Distance Measures

2. k-Nearest Neighbor Method

3. Parzen Windows

Motivation

So far, regression and classification methods covered in the lecture can be used for

- numerical variables,
- binary variables (re-interpreted as numerical), and
- nominal variables (coded as set of binary indicator variables).

Often one is also interested in more complex variables such as

- set-valued variables,
- sequence-valued variables (e.g., strings),
- ...

Motivation

There are two kinds of approaches to deal with such variables:

feature extraction:
try to derive binary or numerical variables,
then use standard methods on the feature vectors.

kernel methods:
try to establish a distance measure between two variables,
then use methods that use only distances between objects
(but no feature vectors).

Distance measures

Let d be a **distance measure** (also called **metric**) on a set $\mathcal{X}$, i.e.,

$$d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_0^+$$

with

1. d is **positiv definite**: $d(x, y) \geq 0$ and $d(x, y) = 0 \Leftrightarrow x = y$

2. d is **symmetric**: $d(x, y) = d(y, x)$

3. d is **subadditive**: $d(x, z) \leq d(x, y) + d(y, z)$
   (triangle inequality)

(for all $x, y, z \in \mathcal{X}$.)

Example: **Euclidean metric** on $\mathcal{X} := \mathbb{R}^n$:

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

Minkowski Metric / L_p metric

Minkowski Metric / L_p metric on $\mathcal{X} := \mathbb{R}^n$:

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

with $p \in \mathbb{R}, p \geq 1$.

$p = 1$ **(taxicab distance; Manhattan distance):**

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

$p = 2$ **(euclidean distance):**

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

$p = \infty$ **(maximum distance; Chebyshev distance):**

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

Minkowski Metric / L_p metric / Example

Example:

$$x := \begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix}, \quad y := \begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix}$$

$$d_{L_1}(x,y) = |1-2| + |3-4| + |4-1| = 1+1+3 = 5$$

$$d_{L_2}(x,y) = \sqrt{(1-2)^2 + (3-4)^2 + (4-1)^2} = \sqrt{1+1+9} = \sqrt{11} \approx 3.32$$

$$d_{L_\infty}(x,y) = \max\{|1-2|, |3-4|, |4-1|\} = \max\{1,1,3\} = 3$$

Similarity measures

Instead of a distance measure sometimes **similarity measures** are used, i.e.,

$$\text{sim} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_0^+$$

with

- sim is symmetric: $\text{sim}(x, y) = \text{sim}(y, x)$.

Some similarity measures have stronger properties:

- sim is **discerning**: $\text{sim}(x, y) \leq 1$ and $\text{sim}(x, y) = 1 \Leftrightarrow x = y$
- $\text{sim}(x, z) \geq \text{sim}(x, y) + \text{sim}(y, z) - 1$.

Some similarity measures have values in $[-1, 1]$ or even $\mathbb{R}$ where negative values denote “dissimilarity”.

Distance vs. Similarity measures

A discerning similarity measure can be turned into a semi-metric (pos. def. & symmetric, but not necessarily subadditive) via

$$d(x, y) := 1 - \mathsf{sim}(x, y)$$

In the same way, a metric can be turned into a discerning similarity measure
(with values eventually in $]-\infty, 1]$).

Cosine Similarity

The cosine of the angle between two vectors in $\mathbb{R}^n$ is used as similarity measure: **cosine similarity**:

$$\text{sim}(x, y) := \cos(\theta) = \frac{\langle x, y \rangle}{||x||_2 \, ||y||_2}$$

Example:

$$x := \begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix}, \quad y := \begin{pmatrix} 2 \\ 4 \\ 1 \end{pmatrix}$$

$$\text{sim}(x, y) = \frac{1 \cdot 2 + 3 \cdot 4 + 4 \cdot 1}{\sqrt{1 + 9 + 16}\sqrt{4 + 16 + 1}} = \frac{18}{\sqrt{26}\sqrt{21}} \approx 0.77$$

- Vectors with similarity 1 point in the same direction.
- Vectors with similarity -1 point in the opposite direction.
- Vectors with similarity 0 are orthogonal.

Distances for Nominal Variables

For binary variables there is only one reasonable distance measure:

$$d(x, y) := 1 - I(x = y) \quad \text{with } I(x = y) := \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

This coincides with the L_∞ distance for the indicator/dummy variables.

The same distance measure is useful for nominal variables with more than two possible values.

For hierarchical variables, i.e., a nominal variable with levels arranged in a hierarchy, there are more advanced distance measures (not covered here).

Distances for Set-valued Variables

For set-valued variables (which values are subsets of a set A) the **Hamming distance** often is used:

$$d(x,y) := |(x \setminus y) \cup (y \setminus x)| = |\{a \in A \,|\, I(a \in x) \neq I(a \in y)\}|$$

(the number of elements contained in only one of the two sets).

Example:

$$d(\{a,e,p,l\},\{a,b,n\}) = 5, \quad d(\{a,e,p,l\},\{a,e,g,n,o,r\}) = 6$$

Also often used is the similarity measure **Jaccard coefficient**:

$$\text{sim}(x,y) := \frac{|x \cap y|}{|x \cup y|}$$

Example:

$$\text{sim}(\{a,e,p,l\},\{a,b,n\}) = \frac{1}{6}, \quad \text{sim}(\{a,e,p,l\},\{a,e,g,n,o,r\}) = \frac{2}{8}$$

Distances for Strings / Sequences

edit distance / **Levenshtein distance**:

$d(x, y) :=$ minimal number of deletions, insertions or substitions to transform x in y

Examples:

$$d(\text{man}, \text{men}) = 1$$
$$d(\text{house}, \text{spouse}) = 2$$

$$d(\text{order}, \text{express order}) = 8$$

Distances for Strings / Sequences

The edit distance is computed recursively. With

$$x_{1:i} := (x_{i'})_{i'=1,\ldots,i} = (x_1, x_2, \ldots, x_i), \quad i \in \mathbb{N}$$

we compute the number of operations to transform $x_{1:i}$ into $y_{1:j}$ as

$$\begin{aligned} c(x_{1:i}, y_{1:j}) := \min\{ & c(x_{1:i-1}, y_{1:j}) + 1, && \text{// delete } x_i, x_{1:i-1} \rightsquigarrow y_{1:j} \\ & c(x_{1:i}, y_{1:j-1}) + 1, && \text{// } x_{1:i} \rightsquigarrow y_{1:j-1}, \text{insert } y_j \\ & c(x_{1:i-1}, y_{1:j-1}) + I(x_i \neq y_j)\} && \text{// } x_{1:i-1} \rightsquigarrow y_{1:j-1}, \text{substitute } y_j \text{ for } x_i \end{aligned}$$

starting from

$$\begin{aligned} c(x_{1:0}, y_{1:j}) = \; c(\emptyset, y_{1:j}) := \; j \quad & \text{// insert } y_1, \ldots, y_j \\ c(x_{1:i}, y_{1:0}) = \; c(x_{1:i}, \emptyset) := \; i \quad & \text{// delete } x_1, \ldots, x_i \end{aligned}$$

Such a recursive computing scheme is called **dynamic programming**.

Distances for Strings / Sequences

Example: compute $d(\text{excused}, \text{exhausted})$.

d	9							
e	8							
t	7							
s	6							
u	5							
a	4							
h	3							
x	2							
e	1							
	0	1	2	3	4	5	6	7
$y[j]/x[i]$		e	x	c	u	s	e	d

Distances for Strings / Sequences

Example: compute $d(\text{excused}, \text{exhausted})$.

d	9	8	7	7	6	5	4	3
e	8	7	6	6	5	4	3	4
t	7	6	5	5	4	3	3	4
s	6	5	4	4	3	2	3	4
u	5	4	3	3	2	3	4	5
a	4	3	2	2	2	3	4	5
h	3	2	1	1	2	3	4	5
x	2	1	0	1	2	3	4	5
e	1	0	1	2	3	4	5	6
	0	1	2	3	4	5	6	7
$y[j]/x[i]$		e	x	c	u	s	e	d

Distances for Strings / Sequences

Example: compute $d(\text{excused}, \text{exhausted})$.

d	9	8	7	7	6	5	4	**3**
e	8	7	6	6	5	4	**3**	4
t	7	6	5	5	4	**3**	3	4
s	6	5	4	4	3	**2**	3	4
u	5	4	3	3	**2**	3	4	5
a	4	3	2	**2**	2	3	4	5
h	3	2	1	**1**	2	3	4	5
x	2	1	**0**	1	2	3	4	5
e	1	**0**	1	2	3	4	5	6
	0	1	2	3	4	5	6	7
$y[j]/x[i]$		e	x	c	u	s	e	d

1. Distance Measures

2. k-Nearest Neighbor Method

3. Parzen Windows

Neighborhoods

Let d be a distance measure.

For a dataset

$$D \subseteq X \times Y$$

and $x \in \mathcal{X}$ let

$$D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

be an enumeration with increasing distance to x, i.e., $d(x, x_i) \leq d(x, x_{i+1})$ (ties broken arbitrarily).

The first $k \in \mathbb{N}$ points of such an enumeration, i.e.,

$$N_k(x) := \{(x_1, y_1), (x_2, y_2), \ldots (x_k, y_k)\}$$

are called a k-**neighborhood** of x (in D).

Nearest Neighbor Regression

The k**-nearest neighbor regressor**

$$\hat{Y}(x) := \frac{1}{k} \sum_{(x',y') \in N_k(x)} y'$$

The k**-nearest neighbor classifier**

$$\hat{p}(Y = y \,|\, x) := \frac{1}{k} \sum_{(x',y') \in N_k(x)} I(y = y')$$

and then predict the class with maximal predicted probability

$$\hat{Y}(x) := \text{argmax}_{y \in \mathcal{Y}} \hat{p}(Y = y \,|\, x)$$

i.e., the majority class w.r.t. the classes of the neighbors.

Decision Boundaries

For 1-nearest neighbor, the predictor space is partitioned in regions of points that are closest to a given data point:

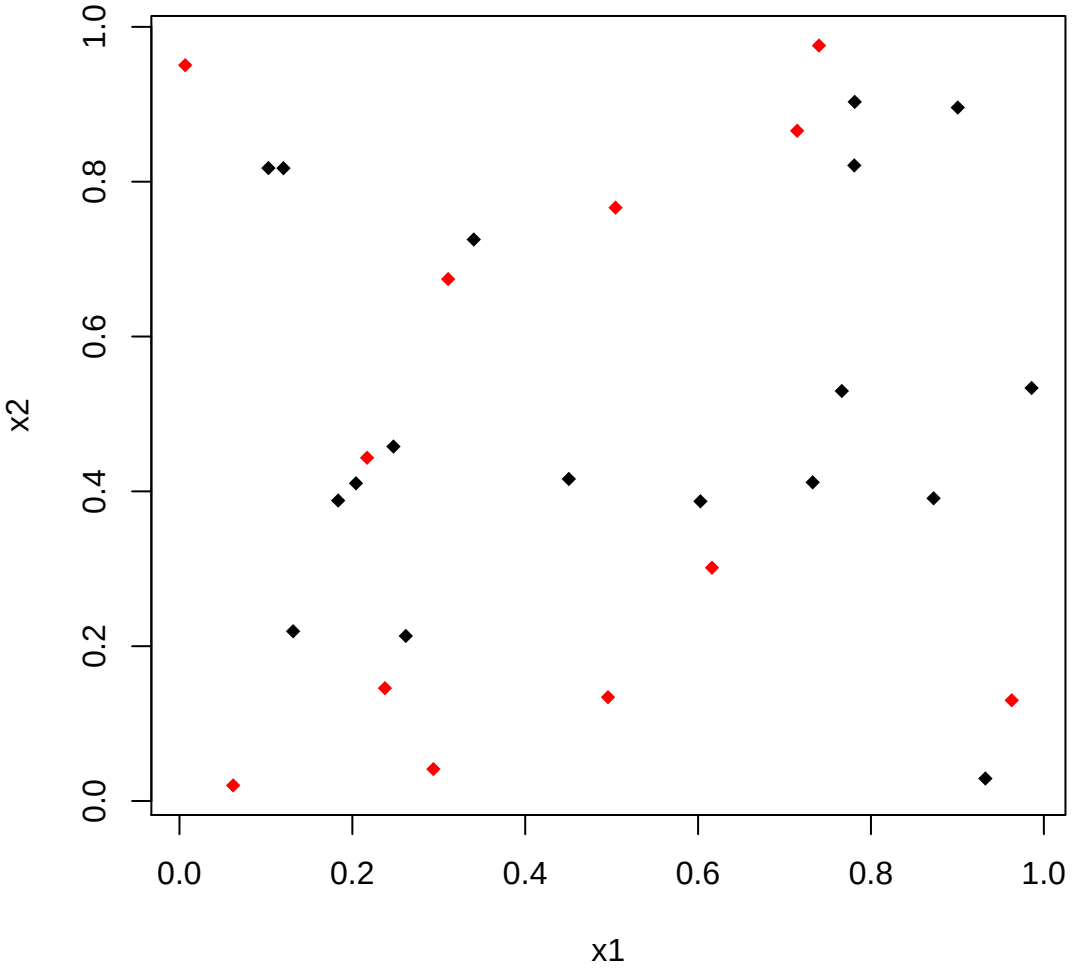
$$\text{region}_D(x_1), \text{region}_D(x_2), \ldots, \text{region}_D(x_n)$$

with

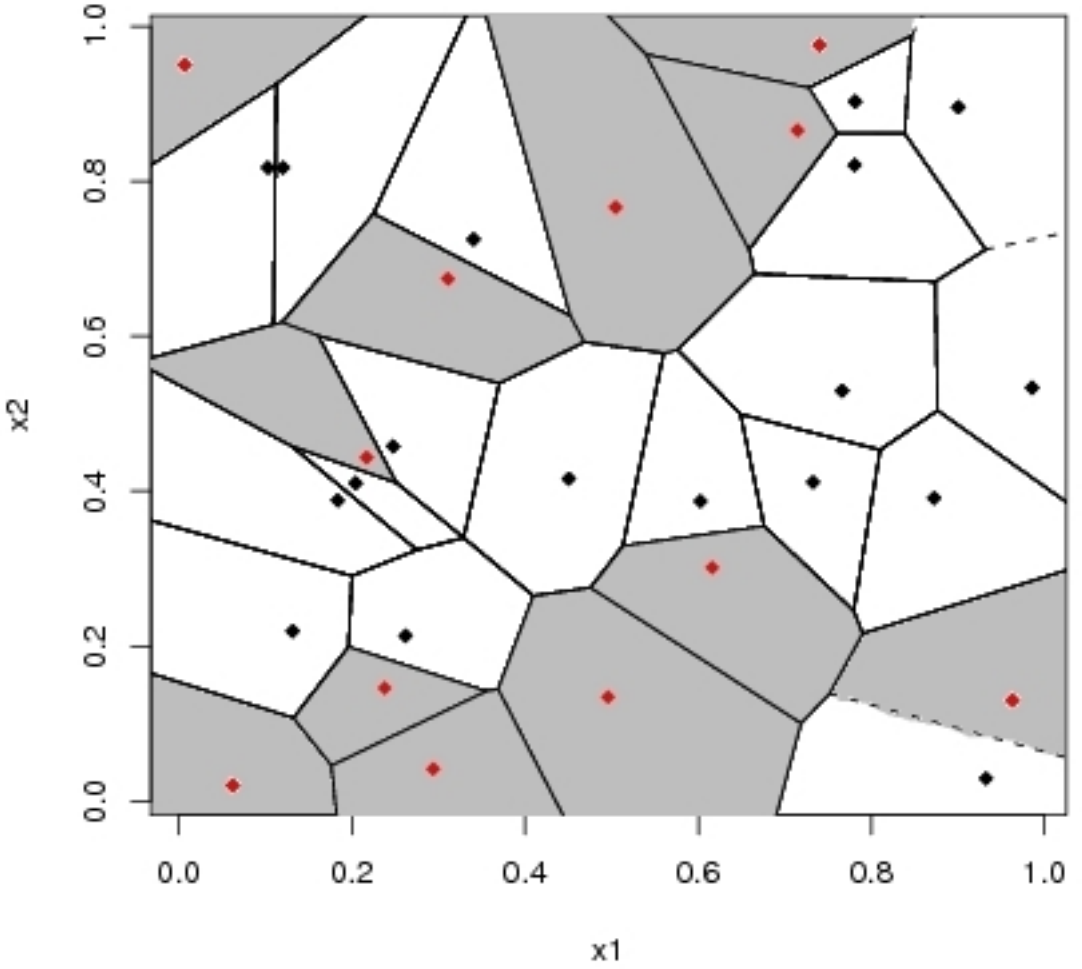
$$\text{region}_D(x) := \{x' \in \mathcal{X} \mid d(x', x) \leq d(x', x'') \quad \forall (x'', y'') \in D\}$$

These regions often are called **cells**, the whole partition a **Voronoi tesselation**.

Decision Boundaries



Decision Boundaries



Complexity of k-Nearest Neighbor Classifier

The k-Nearest Neighbor classifier does not need any learning algorithm as it just stores all the training examples.

On the other hand, predicting using a k-nearest neighbor classifier is slow:

- To predict the class of a new point x, the distance $d(x, x_i)$ from x to each of the n training examples $(x_1, y_1), \ldots, (x_n, y_n)$ has to be computed.
- If the predictor space is $\mathcal{X} := \mathbb{R}^p$, for one such computation we need $O(p)$ operations.
- We then keep track of the k points with the smallest distance.

So in total one needs $O(npk)$ operations.

Accelerations: partial distances

In practice, nearest neighbor classifiers often can be accelerated by several methods.

Partial distances:
Compute the distance to each training point x' only partially, e.g.,

$$d_r(x, x') := (\sum_{i=1}^{r}(x_i - x'_i)^2)^{\frac{1}{2}}, \quad r \leq p$$

As d_r is non-decreasing in r, once $d_r(x, x')$ exceeds the k-th smallest distance computed so far, the training point x' can be dropped.

This is a heuristic:
it may accelerate computations, but it also may slow it down
(as there are additional comparisions of the partial distances with the k smallest distance).

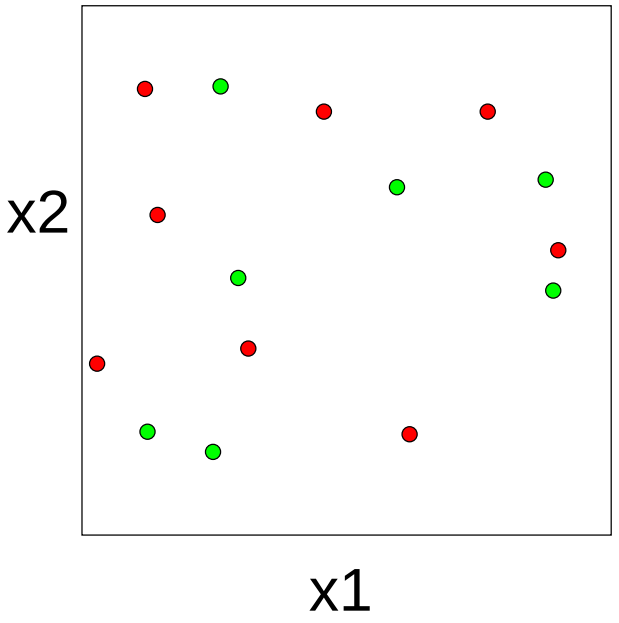
Accelerations: search trees

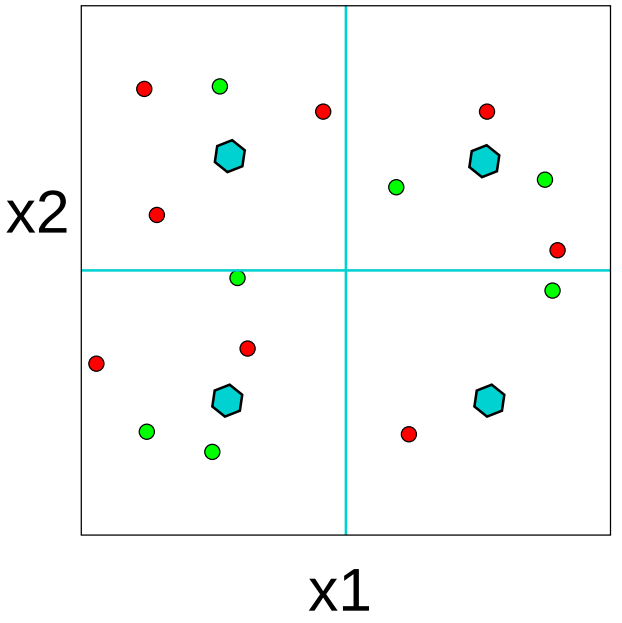
Search trees:
Do not compute the distance of a new point x to **all** training examples, but

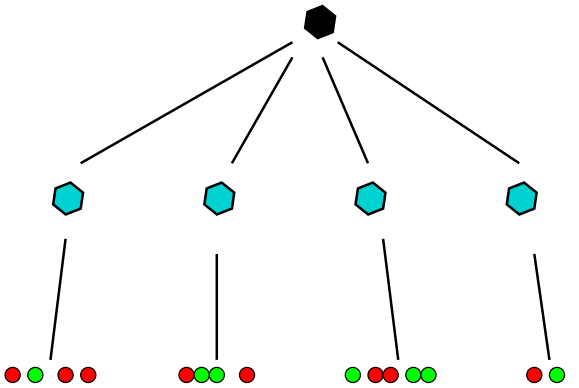
1. organize the training examples as a tree (or a DAG) with
   - sets of training examples at the leaves and
   - a prototype (e.g., the mean of the training examples at all descendent leaves) at each intermediate node.
2. starting at the root, recursively
   - compute the distance to all children of the actual node and
   - branch to the child with the smallest distance,
3. compute distances only to training examples in the leaf finally found.

This is an approximation.

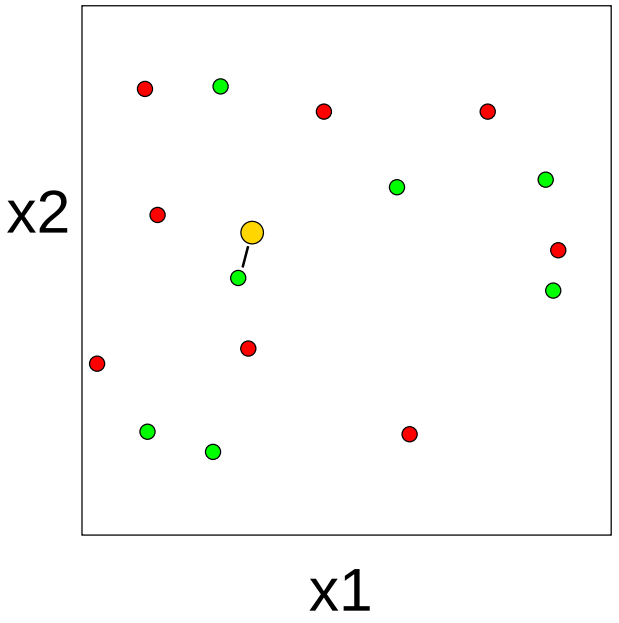
Accelerations: search trees

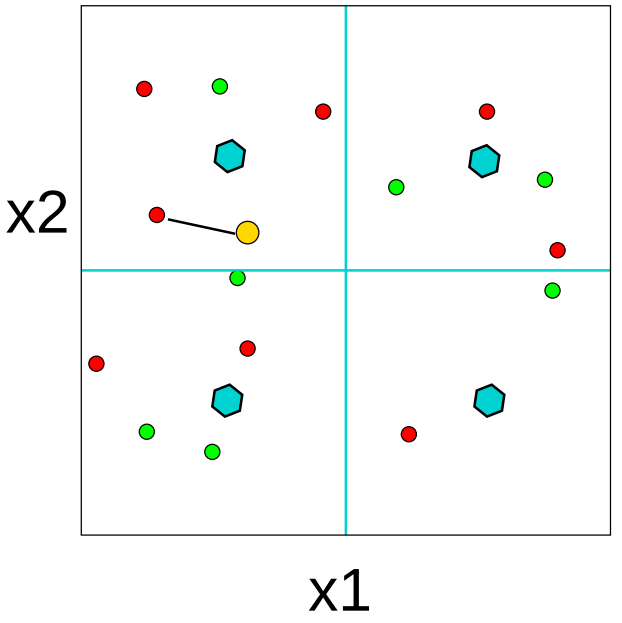


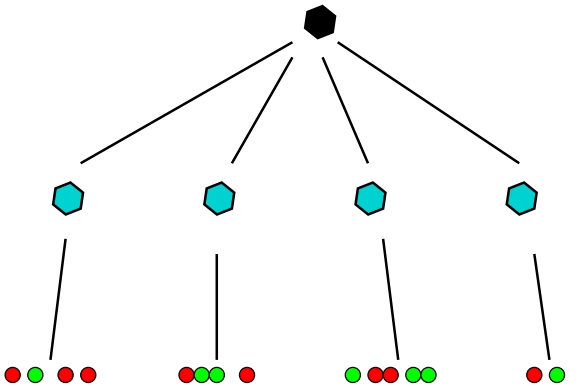




Accelerations: search trees







Accelerations: editing

Editing / Pruning / Condensing:
shrink the set of training data points,
e.g., select a subset of the original training data points.

Example: remove all points with cells that are surrounded by cells of points of the same class.

$$X_{\text{edited}} := \{(x,y) \in X \,|\, \exists (x',y') \in X, R(x') \cap R(x) \neq \emptyset \text{ and } y' \neq y\}$$

This basic editing algorithm

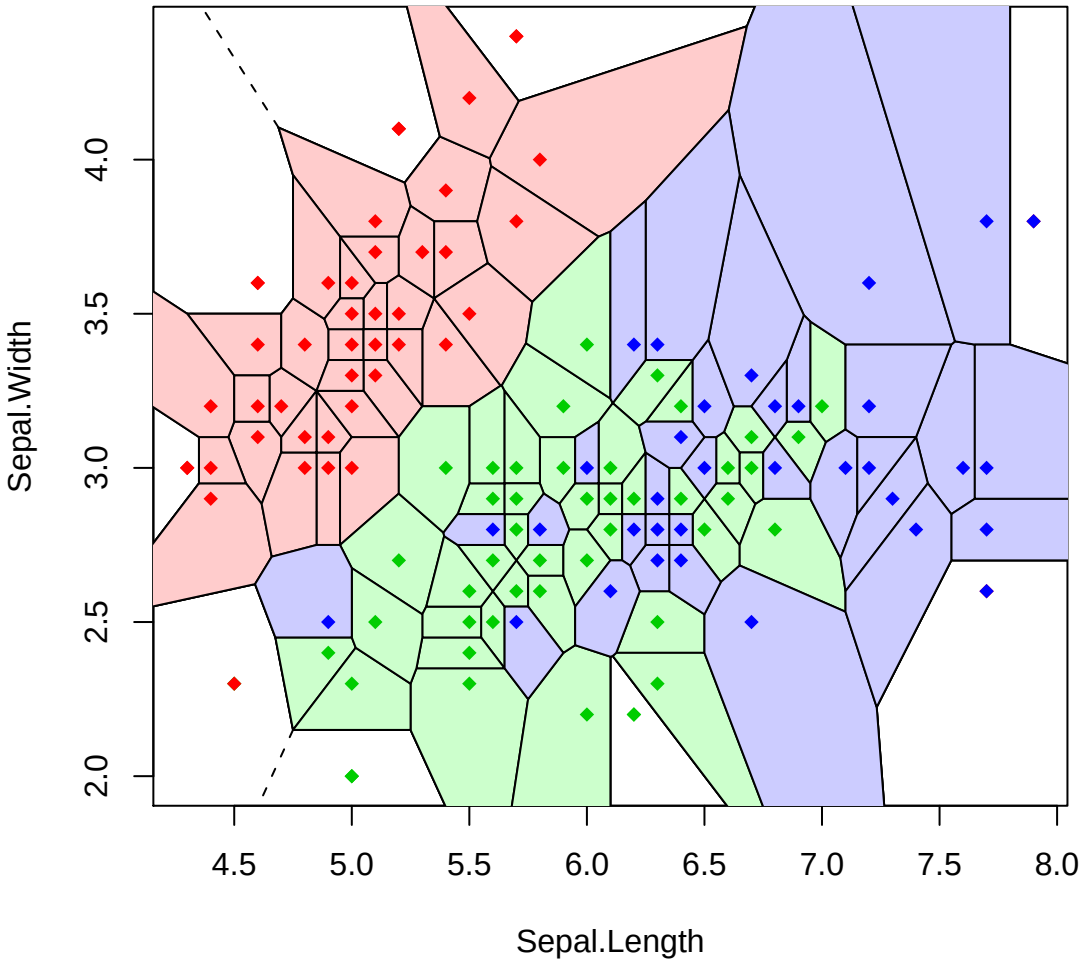
- retains the decision function,
- has complexity $O(d^3 n^{\lfloor \frac{d}{2} \rfloor} \log n)$
  (with $\lfloor x \rfloor := \max\{n \in \mathbb{N} \,|\, n \leq x\}$; Duda et al. 2001, p. 186).

See e.g., Ottmann/Widmayer 2002, p. 501–515 for computing Voronoi diagrams in two dimensions.

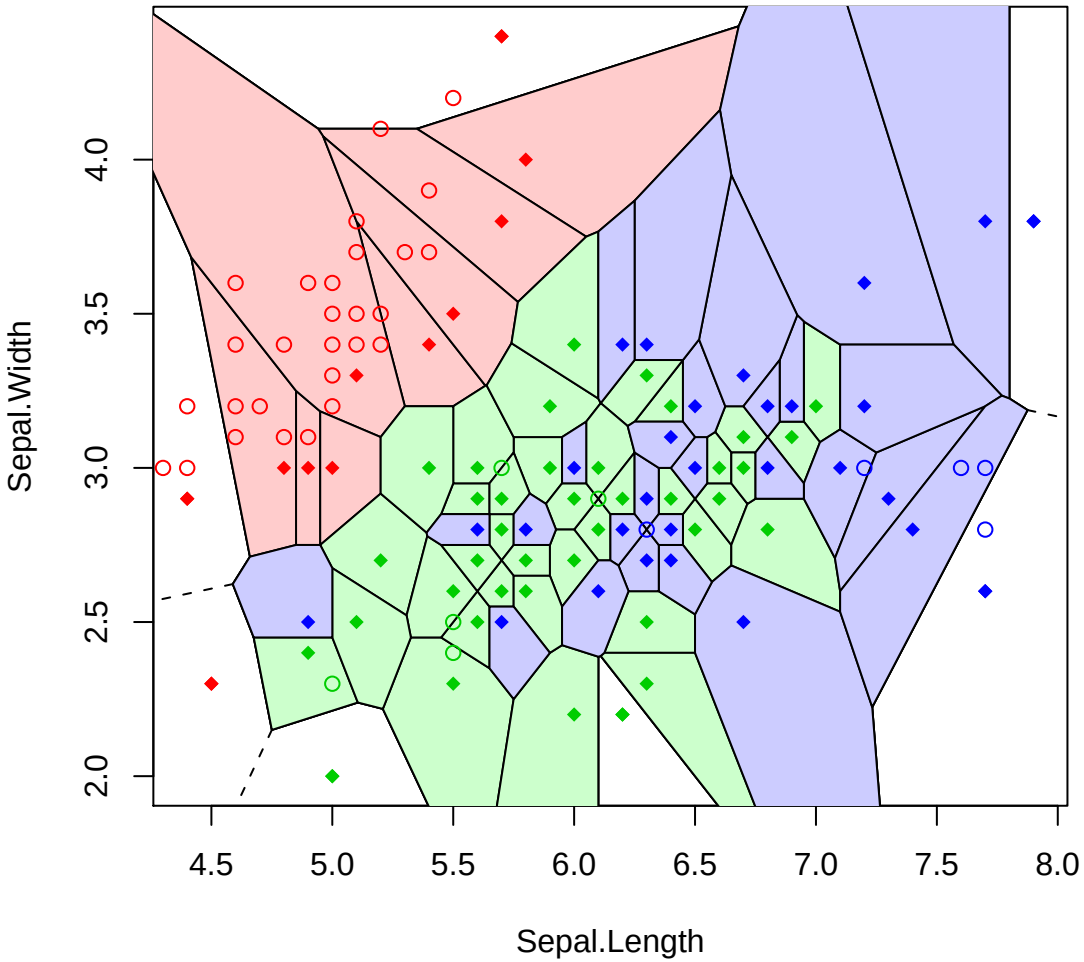
Accelerations: editing

```
1  knn-edit-training-data(training data X) :
2  compute Voronoi cells R(x)    ∀(x, y) ∈ X,
3    esp. Voronoi neighbors N(x) := {(x', y') ∈ X || R(x') ∩ R(x) ≠ ∅}
4  E := ∅
5  for (x, y) ∈ X do
6      hasNeighborOfOtherClass := false
7      for (x', y') ∈ N(x) do
8          if y ≠ y'
9              hasNeighborOfOtherClass := true
10         fi
11     od
12     if not hasNeighborOfOtherClass
13         E := E ∪ {(x, y)}
14     fi
15 od
16 for (x, y) ∈ E do
17     X := X \ {(x, y)}
18 od
```

Accelerations: editing



Accelerations: editing



1. Distance Measures

2. k-Nearest Neighbor Method

3. Parzen Windows

Example

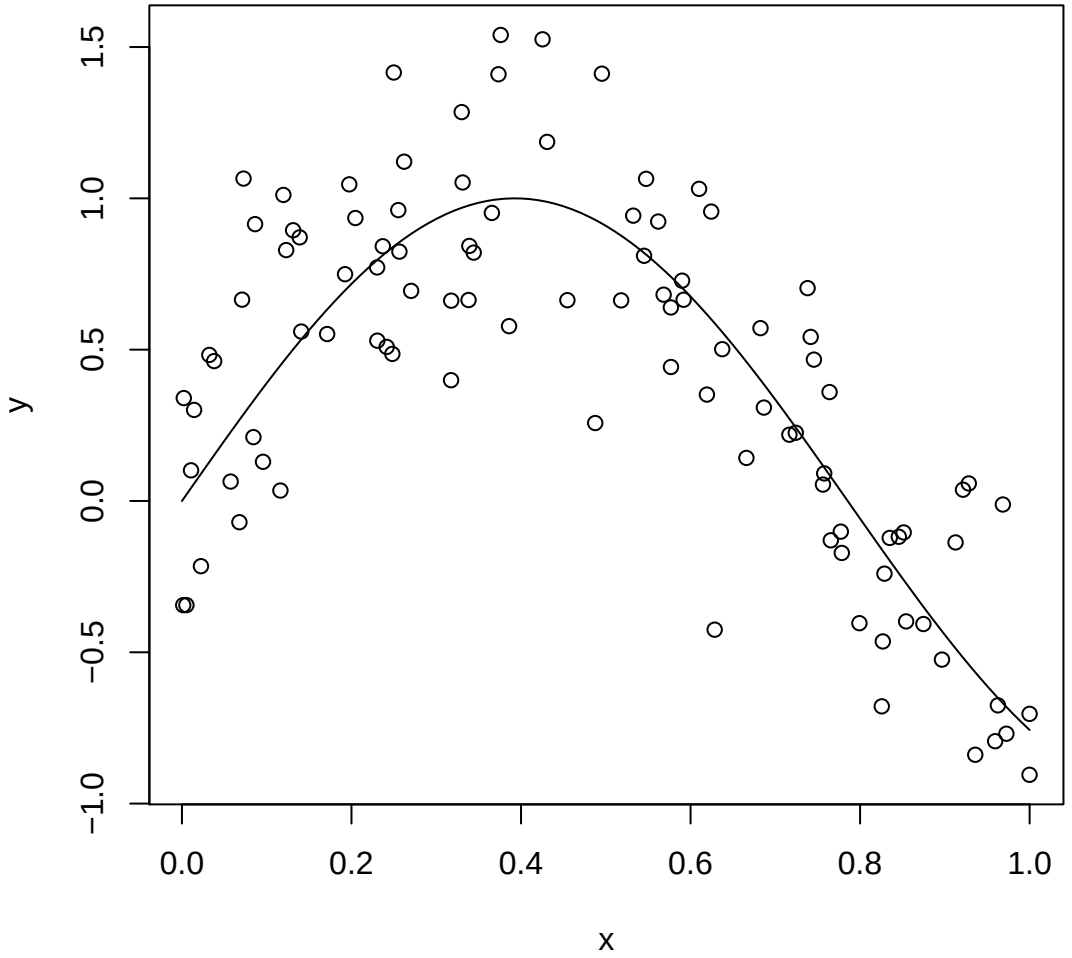


Figure 8: Points generated by the model $y = \sin(4x) + \mathcal{N}(0, 1/3)$ with $x \sim \text{unif}(0, 1)$.

Example / k-Nearest-Neighbor

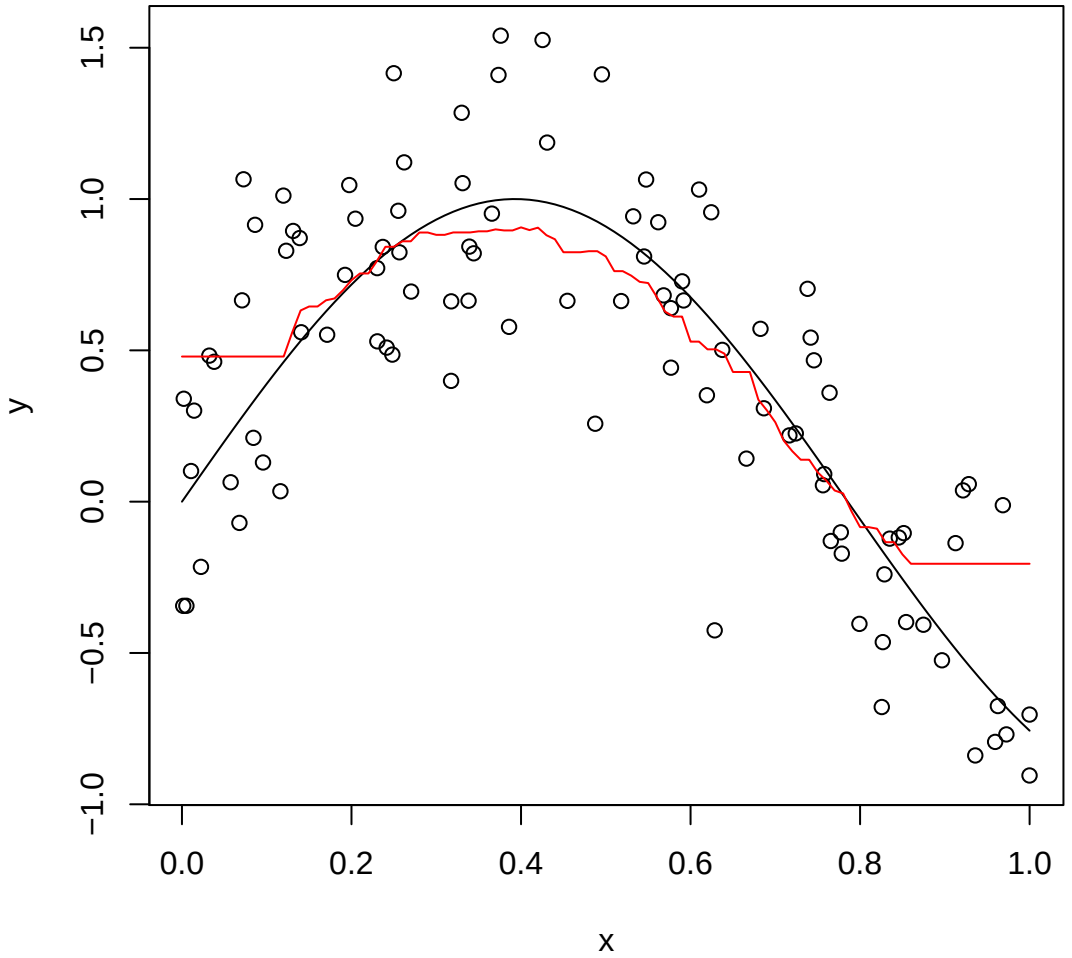


Figure 9: Points generated by the model $y = \sin(4x) + \mathcal{N}(0, 1/3)$ with $x \sim \text{unif}(0, 1)$. 30-nearest-neighbor regressor.

k-Nearest Neighbor is locally constant

k-nearest neighbor models are

- based on discrete decisions if a point is a k-nearest neighbor or not,
- in effect, locally constant,
- and thus not continuous.

Discrete decisions can be captured by binary window functions,

i.e.,

$$K(x, x_0) := \begin{cases} 1, & \text{if } (x, y) \in N_k(x_0) \\ 0, & \text{otherwise} \end{cases}$$

$$\hat{y}(x_0) = \frac{\sum_{(x,y)\in X} K(x, x_0) y}{\sum_{(x,y)\in X} K(x, x_0)}$$

instead of

$$\hat{y}(x_0) = \frac{\sum_{(x,y)\in N_k(x_0)} y}{k}$$

k-Nearest Neighbor is locally constant

In k-nearest neighbor the size of the window varies from point to point: it depends on the density of the data:

in dense parts
the effective window size is small,

in sparse parts
the effective window size is large.

Alternatively, it is also possible to set the size of the windows to a constant λ, e.g.,

$$K_\lambda(x, x_0) := \begin{cases} 1, & \text{if } |x - x_0| \leq \lambda \\ 0, & \text{otherwise} \end{cases}$$

Kernel Regression

Instead of discrete windows, one typically uses continuous windows, i.e., continuous weights

$$K(x, x_0)$$

that reflect the distance of a training point x to a prediction point x_0, called **kernel** or **Parzen window**, e.g.,

$$K(x, x_0) := \begin{cases} 1 - \frac{|x-x_0|}{\lambda}, & \text{if } |x - x_0| \leq \lambda \\ 0, & \text{otherwise} \end{cases}$$

Instead of a binary neighbor/not-neighbor decision, a continuous kernel captures a "degree of neighborship".

Kernels can be used for prediction via **kernel regression**, esp. **Nadaraya-Watson kernel-weighted average**:

$$\hat{y}(x_0) := \frac{\sum_{(x,y)\in X} K(x, x_0)y}{\sum_{(x,y)\in X} K(x, x_0)}$$

Epanechnikov Kernel

Kernels are similarity measures:
the closer two points, the larger the kernel value.

Epanechnikov kernel

$$K_\lambda(x,y) := D\left(\frac{|x-y|}{\lambda}\right)$$

with

$$D(t) := \begin{cases} \frac{3}{4}(1-t^2), & t < 1 \\ 0, & \text{otherwise} \end{cases}$$

The constant $\lambda \in \mathbb{R}^+$ is called **bandwidth**.

More kernels

Tri-cube kernel

$$D(t) := \begin{cases} (1-t^3)^3, & t < 1 \\ 0, & \text{otherwise} \end{cases}$$

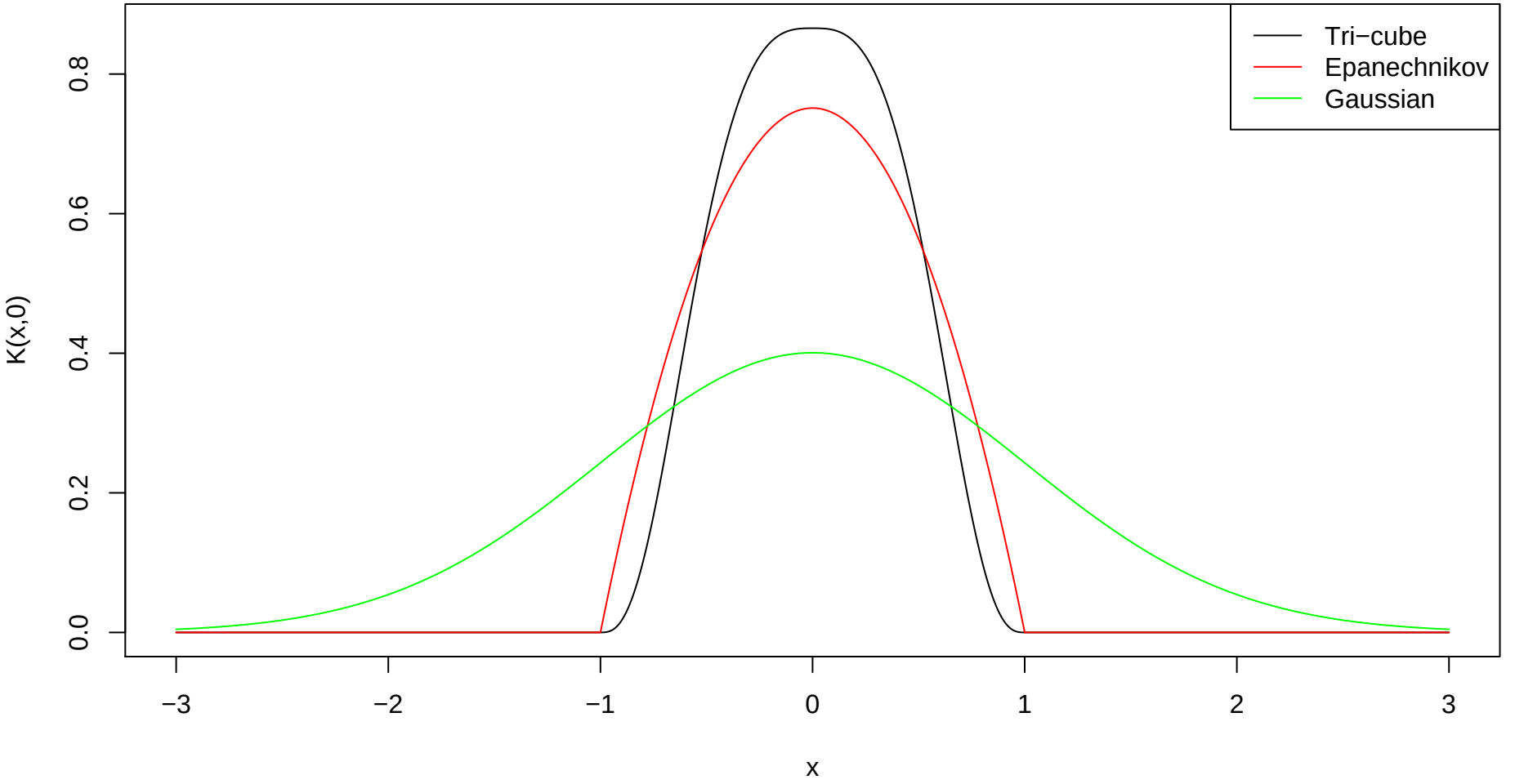
Gaussian kernel

$$D(t) := \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}t^2}$$

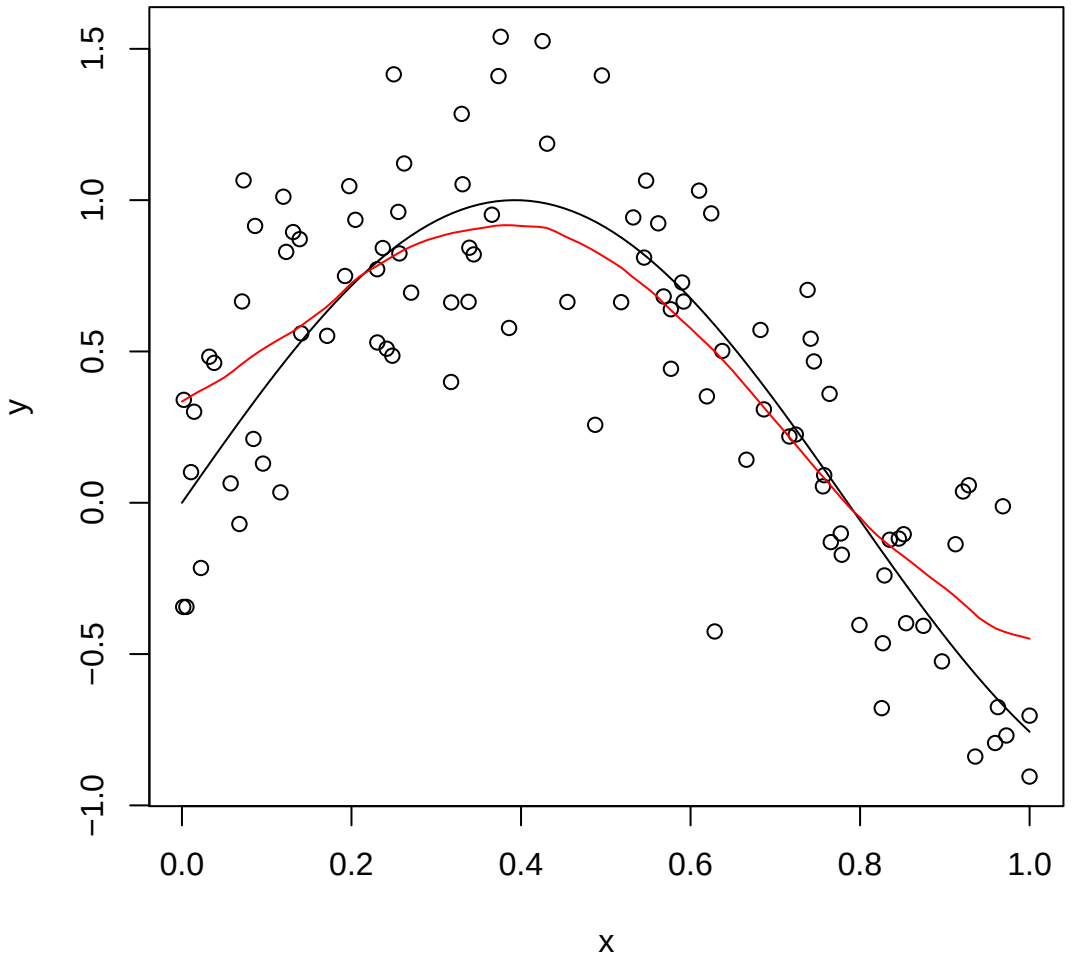
The Epanechnikov and Tri-cube kernel have compact support $[x_0 - \lambda, x_0 + \lambda]$.

The Gaussian kernel has noncompact support, λ acts as standard deviation.

Kernels



Example / Epanechnikov Kernel, $\lambda = 0.2$



Choosing the Bandwidth

If the bandwidth λ is small

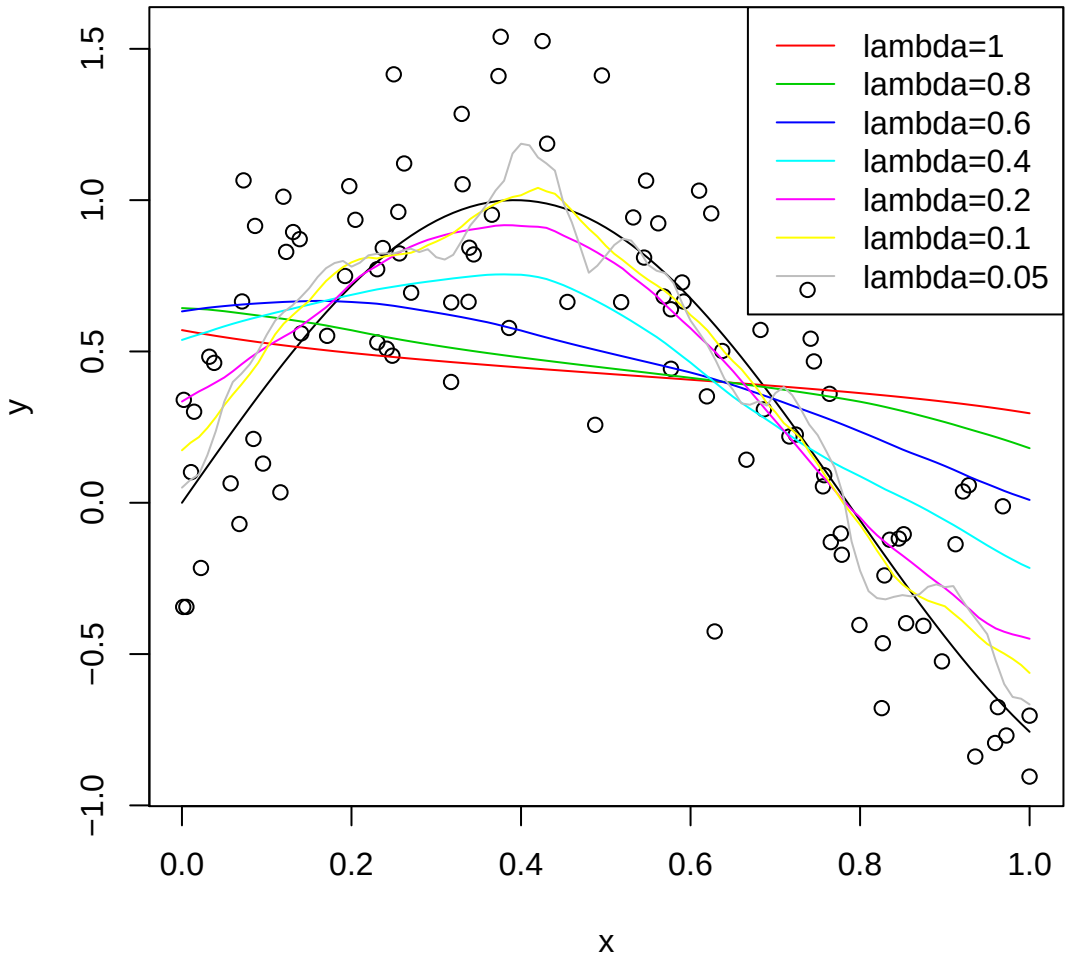
larger variance – as averaged over fewer points
smaller bias – as closer instances are used
$\Rightarrow$ risks to be too bumpy

If the bandwidth λ is large

smaller variance – as averaged over more points
larger bias – as instances further apart are used
$\Rightarrow$ risks to be too rigid / over-smoothed

The bandwidth λ is a parameter (sometimes called a **hyperparameter**) of the model that needs to be optimized / estimated by data.

Example / Epanechnikov Kernel, various bandwidths



Space-averaged Estimates

The probability that an instance x is within a given region $R \subseteq \mathcal{X}$:

$$p(x \in R) = \int_R p(x)dx$$

For a sample

$$x_1, x_2, \ldots, x_n \sim p$$

it is

$$(x_i \in P) \sim \text{binom}(p(x \in R))$$

Let k be the number of x_i that are in region R:

$$k := |\{x_i \,|\, x_i \in R, i = 1, \ldots, n\}|$$

then we can estimate

$$\hat{p}(x \in R) := \frac{k}{n}$$

Space-averaged Estimates

If p is continuous and R is very small, $p(x)$ is almost constant in R:

$$p(x \in R) = \int_R p(x)dx \approx p(x)\,\text{vol}(R), \quad \text{for any } x \in R$$

where vol(R) denotes the volume of region R.

$$p(x) \approx \frac{k/n}{\text{vol}(R)}$$

Space-averaged Estimates

For unlimited data, i.e., $n \to \infty$, we can estimate p more and more accurately:

$$\hat{p}_n(x) = \frac{k_n/n}{V_n}, \qquad \text{with } V_n := \text{vol}(R_n).$$

It must be assured that

$$V_n \to 0$$
$$k_n \to \infty$$
$$k_n/n \to 0$$

There are two methods to accomplish this:

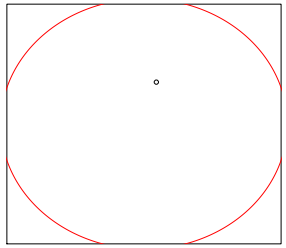
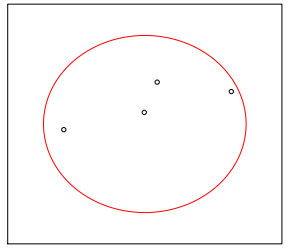
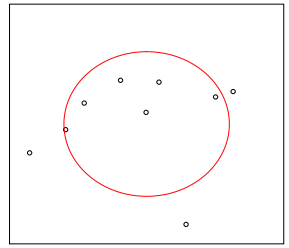
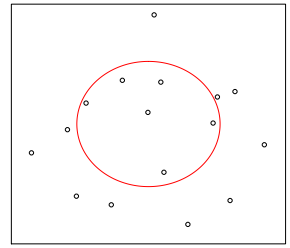
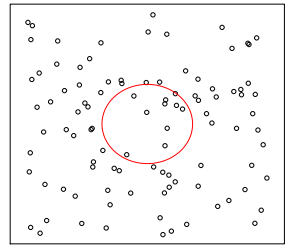
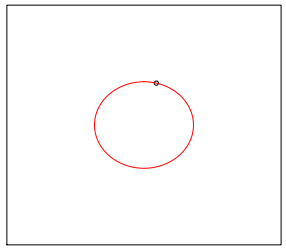
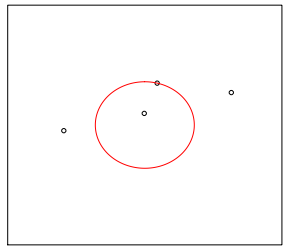
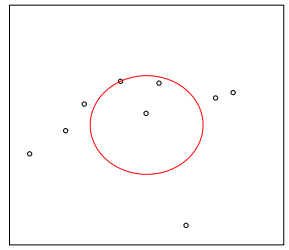
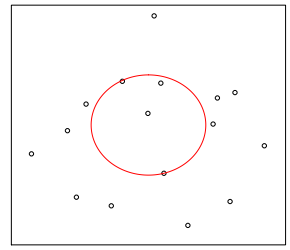
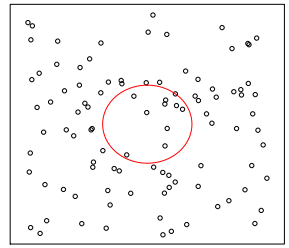
1. nearest-neighbor method:

$$k_n := \sqrt{n}, \quad V_n \text{ is set adaptive to the data}$$

2. Parzen windows:

$$V_n := \frac{1}{\sqrt{n}}, \quad k_n \text{ is set adaptive to the data}$$

Space-averaged Estimates

	$n = 1$	$n = 4$	$n = 9$	$n = 16$	$n = 100$
$V_n = 1/\sqrt{n}$					
$k_n = \sqrt{n}$					

Summary

- Simple classification and regression models can be built by
  - averaging over target values (regression)
  - counting the occurrences of the target class (classification)

  of training instances close by (measured in some **distance measure**).

- If always a fixed number of nearest points is taken into account,
  $\Rightarrow$ the model is called **nearest neighbor**,
  if points are weighted with some similarity measure
  (called **kernel** or **Parzen window**),
  $\Rightarrow$ the model is called **kernel regression** and **kernel classification**.

- There are no learning tasks for these models, as simply all training instances are stored ("memory-based methods").

- Therefore, to compute predictions is more costly than for say linear models. — There are several acceleration techniques (partial distances, search trees, editing).