Machine Learning

3. Nearest Neighbor and Kernel Methods

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.isml.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),

• ...

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 positive definite:** \( d(x, y) \geq 0 \) and \( d(x, y) = 0 \iff x = y \)

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 \( \mathcal{X} := \mathbb{R}^n \):

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

Minkowski Metric / \( L_p \) metric

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

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

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**):

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

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

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

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

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

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

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

Instead of a distance measure sometimes \textbf{similarity measures} are used, i.e.,

\[ \text{sim} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}_0^+ \]

with

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

Some similarity measures have stronger properties:

- sim is \textbf{discerning}: \( \text{sim}(x, y) \leq 1 \) and \( \text{sim}(x, y) = 1 \iff x = y \)

- \( \text{sim}(x, z) \geq \text{sim}(x, y) + \text{sim}(y, z) - 1 \).

Some similarity measures have values in \([-1, 1]\) or even \(\mathbb{R}\) where negative values denote “dissimilarity.”
A discerning similarity measure can be turned into a semi-metric (pos. def. & symmetric, but not necessarily subadditive) via

\[ d(x, y) := 1 - \text{sim}(x, y) \]

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

**Cosine Similarity**

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

\[ \text{sim}(x, y) := \arccos \left( \frac{\langle x, y \rangle}{||x||_2 ||y||_2} \right) \]

Example:

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

\[ \text{sim}(x, y) = \arccos \frac{1 \cdot 2 + 3 \cdot 4 + 4 \cdot 1}{\sqrt{1 + 9 + 16} \sqrt{4 + 16 + 1}} = \arccos \frac{18}{\sqrt{26} \sqrt{21}} \approx \arccos 0.77 \approx 0.69 \]

cosine similarity is not discerning as vectors with the same direction but of arbitrary length have angle 0 and thus similarity 1.
Distances for Nominal Variables

For binary variables there is only one reasonable distance measure:

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

This coincides with the \( L_\infty \) 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 \textbf{Hamming distance} often is used:

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

(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 \textbf{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:**

\[ d(x, y) := \text{minimal number of deletions, insertions or substitions to transform } x \text{ in } y \]

Examples:

\[ d(\text{man}, \text{men}) = 1 \]
\[ d(\text{house}, \text{spouse}) = 2 \]
\[ d(\text{order}, \text{express order}) = 8 \]

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

\[
c(x_{1:i}, y_{1:j}) := \min \left\{ 
  \begin{array}{l}
    c(x_{1:i-1}, y_{1:j}) + 1, \\
    c(x_{1:i}, y_{1:j-1}) + 1, \\
    c(x_{1:i-1}, y_{1:j-1}) + I(x_i \neq y_j)
  \end{array} \right\} \\
\]  

// delete \( x_i, x_{1:i-1} \mapsto y_{1:j} \)  
// \( x_{1:i} \mapsto y_{1:j-1} \), insert \( y_j \)  
// \( x_{1:i-1} \mapsto y_{1:j-1} \), substitute \( y_j \) for \( x_i \)

starting from

\[ c(x_{1:0}, y_{1:j}) = c(\emptyset, y_{1:j}) := j \]  
// insert \( y_1, \ldots, y_j \)
\[ c(x_{1:i}, y_{1:0}) = c(x_{1:i}, \emptyset) := i \]  
// delete \( x_1, \ldots, x_i \)

Such a recursive computing scheme is called **dynamic programming**.
Example: compute $d(\text{excused, exhausted})$.

$$
\begin{array}{|c|c|}
\hline
 d & 9 \\
 e & 8 \\
 t & 7 \\
 s & 6 \\
 u & 5 \\
 a & 4 \\
 h & 3 \\
 x & 2 \\
 e & 1 \\
 \hline
\end{array}

\begin{array}{|c|c|c|c|c|c|c|}
\hline
 y[j] / x[i] & e & x & c & u & s & e \\
\hline
\end{array}
$$
Example: compute $d(\text{excused}, \text{exhausted})$.

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>e</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>s</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>u</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

$y[j]/x[i]$: e x c u s e d
Neighborhoods

Let $d$ be a distance measure. For a dataset

$$D \subseteq X \times Y$$

and $x \in X$ 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

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

and then predict the class with maximal predicted probability

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

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

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

These regions often are called **cells**, the whole partition a **Voronoi tessellation**.
To assess the quality of a classifier  \( \hat{y}(x) \), one can use the \textbf{expected error}, i.e., the probability to predict the wrong class if cases are picked at random:

\[
p(\text{error}) = E(I(y \neq \hat{y})) = \int_X I(y \neq \hat{y})dx = \int_X (1 - p(Y = \hat{y}(x) \mid x))p(x)dx
\]
Bayes Classifier

The minimal expected error can be achieved, if for each point \( x \) the class \( y \) with the largest conditional probability \( p(y | x) \) is predicted, i.e.,

\[
y^*(x) := \arg\max_{y \in Y} p(y | x)
\]

This classifier is called **Bayes classifier** \( y^* \), its error **Bayes error** \( p^*(\text{error}) \).

The Bayes classifier assumes the ideal case that the conditional class probabilities \( p(Y | X) \) are known.

In the case of a deterministic dependency of \( y \) on \( x \), i.e., for each \( x \) there is an \( y \) with \( p(y | x) = 1 \),
the Bayes error is

\[
p^*(\text{error}) = 0
\]

In the case that for each \( x \) there is a uniform distribution of the classes \( y \), i.e., for \( k \) classes \( p(y | x) = 1/k \) for all \( y \),
the Bayes error is maximal

\[
p^*(\text{error}) = \frac{k - 1}{k}
\]
If we have unlimited data, the error rate of the nearest neighbor classifier is bound as follows:

\[ p^\ast(\text{error}) \leq \lim_{n \to \infty} p_n(\text{error}) \leq p^\ast(\text{error})(2 - \frac{k}{k-1}p^\ast(\text{error})) \]

where \( p_n(\text{error}) \) denotes the error rate for the nearest neighbor classifier in a sample of \( n \) points.

Roughly spoken “at least half of the classification information in an infinite data set resides in the nearest neighbor” (Duda et al. 2001).
Now let \( y^*(x) := \arg\max_y p(y|\mathbf{x}) \) the Bayes classifier:

\[
\sum_y p(y_0 = y|\mathbf{x}_0)^2 = p(y_0 = y^*(\mathbf{x}_0)|\mathbf{x}_0)^2 + \sum_{y \neq y^*(\mathbf{x}_0)} p(y_0 = y|\mathbf{x}_0)^2 \\
\geq (1 - p^*(\text{error}|\mathbf{x}_0))^2 + \frac{1}{k-1} p^*(\text{error}|\mathbf{x}_0)^2 \\
= 1 - 2p^*(\text{error}|\mathbf{x}_0) + \frac{k}{k-1} p^*(\text{error}|\mathbf{x}_0)^2
\]

because the sum is minimal if all \( p(y_0 = y|\mathbf{x}_0) \) are equal, and thus

\[
p(y_0 = y|\mathbf{x}_0) = \frac{1}{k-1} (1 - p(y_0 = y^*(\mathbf{x}_0)|\mathbf{x}_0)) = \frac{1}{k-1} p^*(\text{error}|\mathbf{x}_0)
\]

Then we continue

\[
\lim_{n \to \infty} p_n(\text{error}|\mathbf{x}_0) = 1 - \sum_y p(y_0 = y|\mathbf{x}_0)^2 \leq 2p^*(\text{error}|\mathbf{x}_0) - \frac{k}{k-1} p^*(\text{error}|\mathbf{x}_0)^2
\]

Now

\[
\lim_{n \to \infty} p_n(\text{error}) = \lim_{n \to \infty} \int p_n(\text{error}|\mathbf{x}_0)p(\mathbf{x}_0)d\mathbf{x}_0 \\
\leq \int (2p^*(\text{error}|\mathbf{x}_0) - \frac{k}{k-1} p^*(\text{error}|\mathbf{x}_0)^2)p(\mathbf{x}_0)d\mathbf{x}_0 \\
= 2p^*(\text{error}) - \frac{k}{k-1} \int p^*(\text{error}|\mathbf{x}_0)^2p(\mathbf{x}_0)d\mathbf{x}_0
\]
And finally as

$$V(p^*(\text{error})) = \int (p^*(\text{error}|x_0) - p^*(\text{error}))^2 p(x_0) dx_0$$

$$= \int p^*(\text{error}|x_0)^2 p(x_0) dx_0 - p^*(\text{error})^2 \geq 0$$

$$\Rightarrow \int p^*(\text{error}|x_0)^2 p(x_0) dx_0 \geq p^*(\text{error})^2$$

we get

$$\lim_{n \to \infty} p_n(\text{error}) \leq 2p^*(\text{error}) - \frac{k}{k-1} \int p^*(\text{error}|x_0)^2 p(x_0) dx_0$$

$$\leq 2p^*(\text{error}) - \frac{k}{k-1} p^*(\text{error})^2$$

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 $X := \mathbb{R}^p$, 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.,

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

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 comparisons 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.
2. \( k \)-Nearest Neighbor Method

Accelerations: search trees
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 algorithms
• retains the decision function,
• has complexity \( O(d^3n^\frac{d}{2} \log n) \)
  (with \( |x| := \max\{n \in \mathbb{N} | n < x\} \); Duda et al. 2001, p. 186).

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

```plaintext
1 knn-edit-training-data(training data X) :
2 compute Voronoi cells R(x) \forall (x, y) \in X,
3 esp. Voronoi neighbors N(x) := \{(x', y') \in X \parallel R(x') \cap R(x) \neq \emptyset\}
4 E := \emptyset
5 for (x, y) \in X do
6    hasNeighborOfOtherClass := false
7    for (x', y') \in N(x) do
8        if y \neq y'
9            hasNeighborOfOtherClass := true
10       fi
11    od
12    if not hasNeighborOfOtherClass
13       E := E \cup \{(x, y)\}
14    fi
15 od
16 for (x, y) \in E do
17    X := X \ \{(x, y)\}
18 od
```
Machine Learning / 2. \( k \)-Nearest Neighbor Method

Accelerations: editing

Lars Schmidt-Thieme, Information Systems and Machine Learning Lab (ISMLL), Institute BW/WI & Institute for Computer Science, University of Hildesheim
Course on Machine Learning, winter term 2007
33/48
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 \( x \sim \text{unif}(0, 1) \).
**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,

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

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

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

**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

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 $\lambda$, e.g.,

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

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**:

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

**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, \( \lambda \) acts as standard deviance.
Example / Epanechnikov Kernel, $\lambda = 0.2$
Choosing the Kernelwidth

If the kernelwidth $\lambda$ is small
larger variance – as averaged over fewer points
smaller bias – as closer instances are used
$\Rightarrow$ risks to be too bumpy

If the kernelwidth $\lambda$ 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 kernelwidth $\lambda$ is a parameter (sometimes called a hyperparameter) of the model that needs to be optimized / estimated by data.
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 $R$:

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

then we can estimate

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

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 $\text{vol}(R)$ 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:

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

It must be assured that

\[
V_n \to 0 \quad \quad k_n \to \infty \quad \quad k_n/n \to 0
\]

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:
   \[
   V_n := \frac{1}{\sqrt{n}}, \quad k_n \text{ is set adaptive to the data}
   \]
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,
  ⇒ the model is called nearest neighbor,
  if points are weighted with some similarity measure
    (called kernel or Parzen window),
  ⇒ the model is called kernel regression and kernel classification.

• There is 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).

• The error rate of the 1-nearest-neighbor classifier is bound by twice
  the Bayes error rate.