4. Nearest neighbor searching
4.1 Foundation of nearest neighbors

At first, let us look at the theoretical background (in a simplified way) on the probabilistic basis.

Suppose we place a region of volume V and $\mathbf{x}$ is a vector representing the case for which k nearest neighbors are searched for. V is the volume of the variable space comprising all training cases.

Let us assume that searching for neighbors yields k cases (samples, examples or observations), k_i of which are from class c_i.

We may approximate the joint density for continuous variables (like probability in the space of discrete variables) for **x** and c_i by

$$p_n(\mathbf{x}, c_i) = \frac{k_i / n}{V}$$

as expected when the number of all cases is n. From Bayes rule we obtain

$$P_n(c_i|\mathbf{x})\, p_n(\mathbf{x}) = p_n(\mathbf{x}, c_i)$$

or

$$P_n(c_i \mid \mathbf{x}) = \frac{p_n(\mathbf{x}, c_i)}{\sum_{j=1}^{C} p_n(\mathbf{x}, c_j)} = \frac{(k_i / n) / V}{\sum_{j=1}^{C} (k_j / n) / V} = \frac{k_i}{k}$$

where *k* is the number of neighbors searched for. Thus,

$$P_n(c_i \mid \mathbf{x}) = \frac{k_i}{k}$$

which is simply the fraction of cases ”captured” in region *R* with class label c_i.

This gives a computationally simple direct classification strategy called *k*-nearest neighbor rule or searching.

Nearest neighbor search is a distance-based classification method. The familiar *distance metric* is L_2 or Euclidean distance as follows for p-dimensional vectors **x** and **y**.

$$D(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{j=1}^{p}\left(x_j - y_j\right)^2}$$

Cityblock, Manhattan or L_1 is

$$D(\mathbf{x},\mathbf{y}) = \sum_{j=1}^{p}\left|x_j - y_j\right|$$

and L_∞ is the following one.

$$D(\mathbf{x},\mathbf{y}) = \max_j \left| x_j - y_j \right|$$

Cosine

$$D(\mathbf{x},\mathbf{y}) = 1 - \frac{\mathbf{x}\cdot\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}$$

and correlation *distance measures* can sometimes be used (mean vector subtracted from **x** or **y**).

$$D(\mathbf{x},\mathbf{y}) = 1 - \frac{(\mathbf{x}-\bar{\mathbf{x}})\cdot(\mathbf{y}-\bar{\mathbf{y}})}{\|\mathbf{x}-\bar{\mathbf{x}}\|\|\mathbf{y}-\bar{\mathbf{y}}\|}$$

The norm equals L_2, Euclidean distance as usual.

$$\|\mathbf{x}\|_2 = \sqrt{x_1^2 + \ldots + x_p^2}$$

$$\|\mathbf{x} - \bar{\mathbf{x}}\|_2 = \sqrt{(x_1 - \bar{x}_1)^2 + \ldots + (x_1 - \bar{x}_p)^2}$$

Furthermore, Mahalonobis distance (p. 116) is another measure.

Algorithm: Nearest neighbor search

(1) Out of N training vectors, identify the k nearest neighbors, irrespective of class label. It is best to choose odd k, and in general not to be a multiple of the number of classes C.

(2) Out of these k cases, identify the number of vectors, k_i, that belong to class c_i, i=1,2,…,C. The next sum is gained.

$$\sum_{i=1}^{C} k_i = k$$

(3) Assign **x** to class c_i, with the maximum number of nearest neighbors.

When we use odd k values, a tie is not possible for binary classification. It is good always to use odd k's although this does not guarantee inexistence of ties for other than C equal to 2. A rule of thumb for the maximum of k is the square root of N.

The time complexity of one search is $O(kN)$ giving $O(N)$ for small k's. Searching for the nearest neighbors for all cases requires $O(N^2)$.

When the k=1 nearest neighbor rule is applied, the training vectors $\mathbf{x}_i$, i=1,2,…,N, define a partition of p-dimensional space into N regions, R_i. Each of these is defined as follows (Fig. 4.1).

$$R_i = \left\{ \mathbf{x} \middle| D(\mathbf{x}, \mathbf{x}_i) < D(\mathbf{x}, \mathbf{x}_j), i \neq j \right\}$$

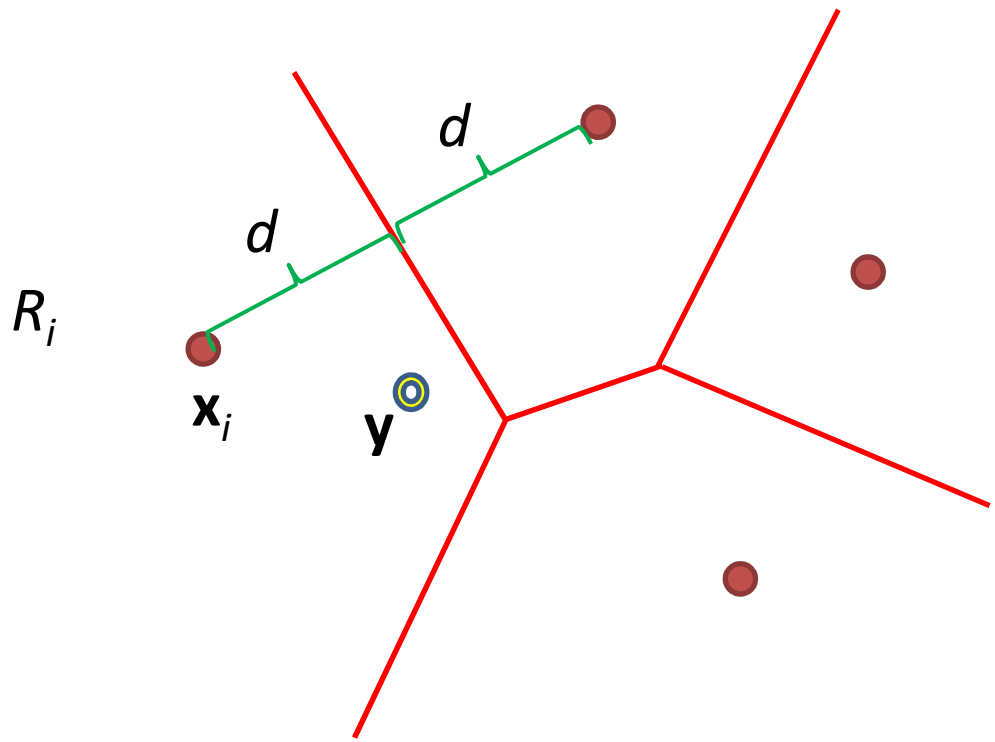

Fig. 4.1 An example of Voronoi tesselation in the two-dimensional space and for Euclidean distance. R_i contains all points or vectors (cases) **y** in the space that are closer to $\mathbf{x}_i$ than any other points of the training set.

Fig. 4.2 shows decision boundaries for k equal to 1 and 2. As mentioned, the latter is no good choice, but is here merely as a simplified example.

The complete search of nearest neighbors requires much computation, $O(N^2)$. The practical computational costs get higher when the number p of dimensions grows, i.e., the method suffers from the curse of dimensionality. However, this can be relieved with such methods as *KD*-trees that compute this in $O(N \log N)$.

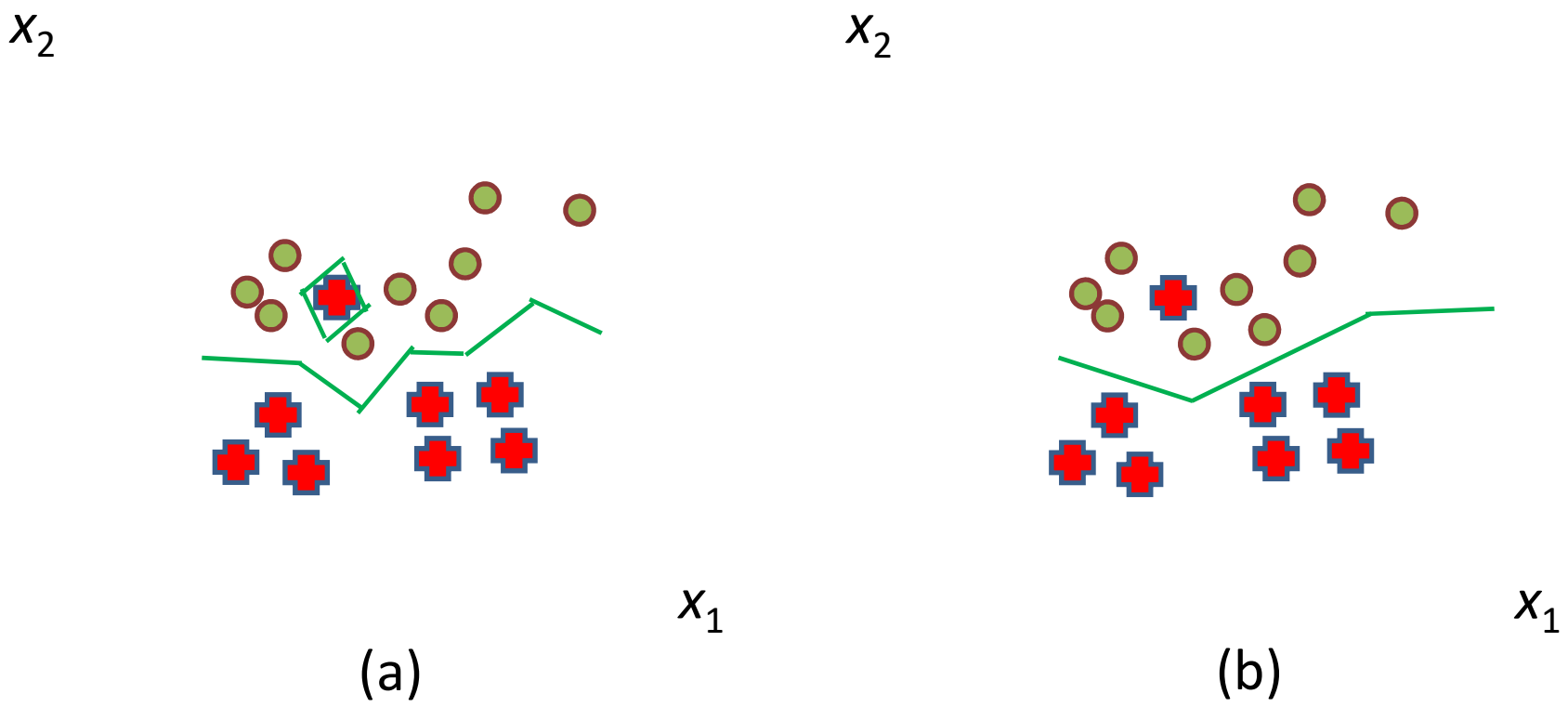

Fig. 4.2 The decision boundary of nearest neighbors searching for (a) k=1 and (b) 2.

More importantly, as the number of dimensions *p* increases, so the distance to other cases tends to increase. In addition, they can be far away in a variety of directions - there might be cases that are relatively close in some dimensions (or for some variables), but a long way in others.

4.2. *KD*-trees

KD-trees as a data structure reduce the cost of finding a nearest neighbor to $O(\log N)$. The construction of a tree is $O(N \log^2 N)$, with much of the computational cost being in the computation of the median, which with a naive algorithm requires sorting and is therefore $O(N \log N)$. Still, this can be computed efficiently with a randomised algorithm in $O(N)$ time.

All in all, *KD*-trees are no powerful remedy, because they are often reasonable to use for low dimensions. In high-dimensional spaces, the curse of dimensionality causes the algorithm to require to visit many more branches than in low-dimensional spaces. In particular, when the number of cases is only slightly higher than the number of dimensions, the algorithm is only slightly better than a linear search of all of the cases.

The idea behind the *KD*-tree is simple. We create a binary tree by selecting one dimension at a time to split into two, and placing the line through the median of the point coordinates of that dimension. This resembles a decision tree, quite similar to constructing a binary tree: we identify a place to split into two choices, left and right, and then carry on down the tree. Therefore, this is natural to make recursively.

The choice of what to split and where is what makes *KD*-tree special. Just one dimension is split in each step.

The position of the split is found by computing the median of the points that are to be split in that one dimension and putting the line there. The choice of which dimension to split alternates through the different choices, or it can be made randomly. In the following example (Fig. 4.3), the procedure cycles through the possible dimensions based on the depth of the tree so far, so that in two dimensions it alternates horizontal and vertical splits.

In Fig. 4.3 there are seven points: (5,4), (1,6), (6,1), (7,5), (2,7), (2,2), (5,8).

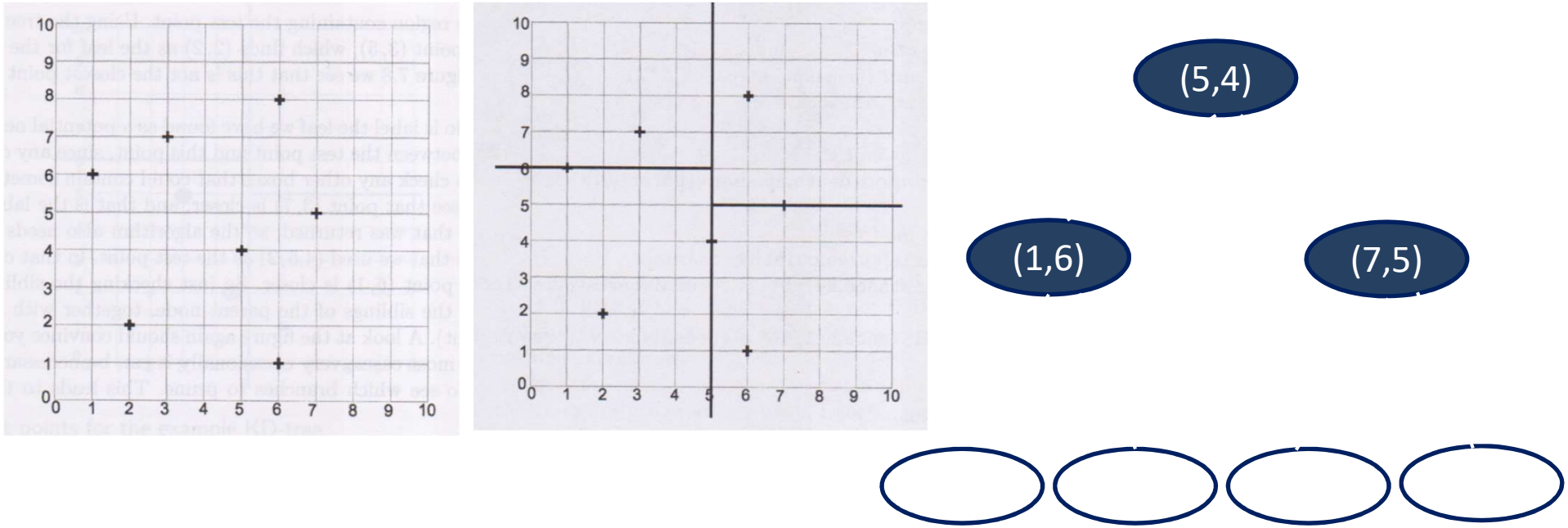

Fig. 4.3 (a) The initial set of cases, (b) the splits and leaf cases found by the *KD*-tree, and (c) the *KD*-tree that made the splits.

Searching for a nearest neighbour in a *KD*-tree proceeds this way:

(1) Starting with the root node, move down the tree recursively, in the same way that it would be if the search point were being inserted, i.e. go left or right depending on whether the point is less or greater than the current node in the split dimension.

(2) Once a leaf is found, save that node point as the "current best“.

(3) Unwind the recursion of the tree, performing the following steps at each node:

(3.1) If the current node is closer to the test case than the current best, then change it to be the current best.

(3.2) Check whether there could be any points on the other side of the splitting plane that are closer to the search point than the current best. This is done by intersecting the splitting hyperplane with a hypersphere around the search point that has a radius equal to the current nearest distance. Since the hyperplanes are all axis-aligned this is implemented as a simple comparison to see whether the distance between the splitting coordinate of the search point and current node is less than the distance (overall coordinates) from the search point to the current best.

(3.2.1) If the hypersphere crosses the plane, there could be nearer points on the other side of the plane, so move down the other branch of the tree from the current node looking for nearer points, following the same recursive process as the entire search.

(3.2.2) If the hypersphere does not intersect the splitting plane, then continue walking up the tree, and the entire branch on the other side of that node is eliminated.

(4) When finishing the process for the root node, stop.

Searching the tree is like that of a binary tree. In addition, we are interested in detecting the nearest neighbors of a test case as follows. We start at the root, recurse down through the tree comparing just one dimension at a time until we find a leaf being in the region with the test case (point).

Let us look at the tree in Fig. 4.4. The test point is (3,5), which detects (2,2) as the leaf for the box that (3,5) is in. Nevertheless, this is not the nearest. Thus, more computation is needed.

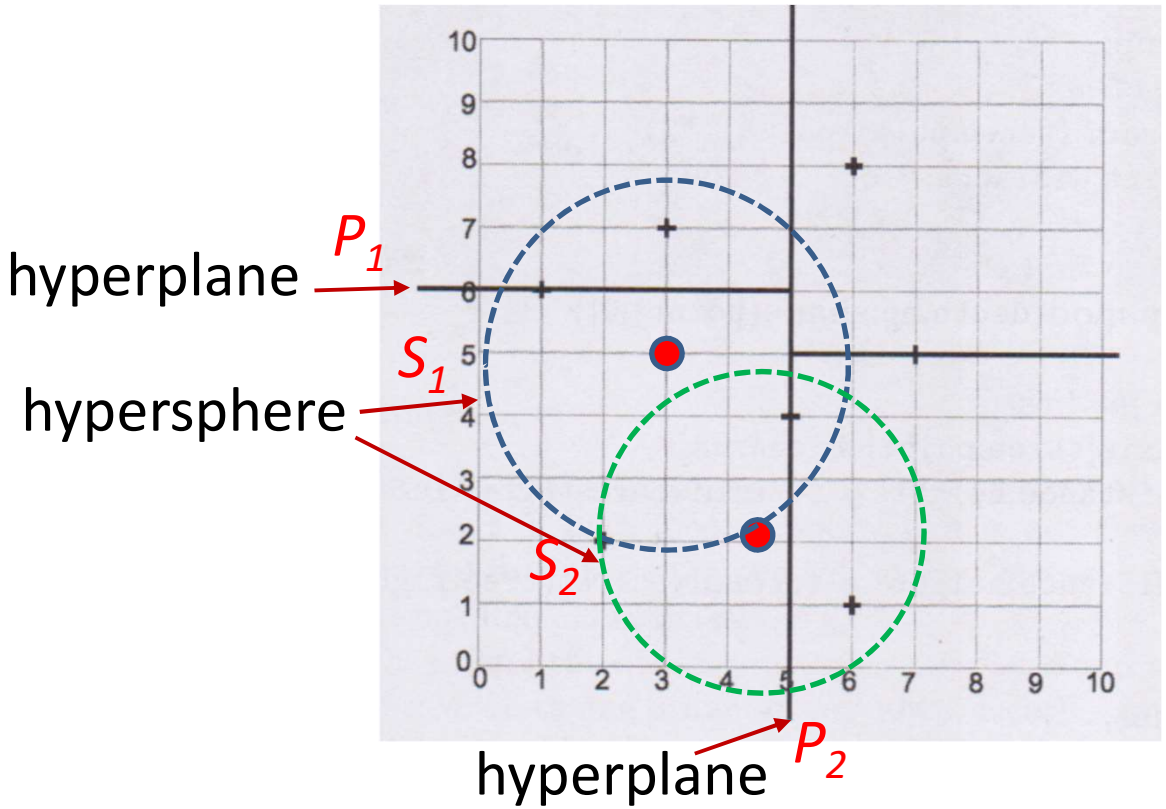

Fig. 4.4 Two test points of the *KD*-tree: (3,5) and (4.5,2).

At first, the leaf (2,2) found is labeled as the current best. Then searching is continued inside the hypersphere S_1 intersecting, among others, hyperplane P_1 and containing three cases (1,6), (5,4) and (3,7) of which the last is nearest the test case and also nearer than the current best (leaf). Thus, (3,7) is the new current best. Although S_1 also intersects P_2, this does not bring new candidates.

For the other test case (4.5,2) hypersphere S_2 intersects hyperplane P_2. Thus, it is necessary to go via the root down to leaf including leaf (6,1) that is nearer than (2,2) or (5,4).

4.3 Weighted nearest neighbor searching

When in nearest neighbor classification majority votes or winner-takes-all technique between classes are employed, ties are possible even if rather infrequent for most data sets. We have to notice that in the nearest neighbor searching all *k* ”votes” were ”democratic” or equal independent of the distances of the neighbors. Thus, the entire process deals with only sorting the nearest neighbors according to their class labels, but not using the actual distance information after having decided which the nearest neighbors are.

We can present the final decision process of nearest neighbor classification, after having found the nearest neighbors, with function

$$\hat{h}(\mathbf{x}_q) = \arg\max_{c \in L} \sum_{j=1}^{k} \delta(c, f(\mathbf{x}_j))$$

where $\delta(a,b)=1$ if $a=b$ and $\delta(a,b)=0$ otherwise, and L is the set of class labels, c represent any class label, $\mathbf{x}_q$ is a test case, $\mathbf{x}_j$ is a nearest neighbor and $f(\cdot)$ is a mapping from data cases (their variables) onto L (its value is some class label). The result or prediction $\hat{h}(\cdot)$ will be the label of the majority class among the nearest neighbors.

We can also take more advantage of actual distance information, as distance values. For example, to alleviate the possible problem of ties we can calculate weight value w_j for every nearest neighbor as follows.

$$\hat{h}(\mathbf{x}_q) = \arg\max_{c \in L} \sum_{j=1}^{k} w_j \delta(c, f(\mathbf{x}_j))$$

where

$$w_j = \frac{1}{D(\mathbf{x}_q, \mathbf{x}_j)^2}$$

Here D is some distance measure, say, Euclidean. If $\mathbf{x}_q$ were equal to $\mathbf{x}_j$, in other words, these were two identical cases, we would assign $\hat{h}(\mathbf{x}_q)$ to be equal to $f(\mathbf{x}_j)$. (The denominator cannot become 0.)

This rule does not give votes (integers 1), but real values that are summed up class by class and, finally, the maximum of those sums is taken which determines the class predicted.

In a way, we now measure the importance of a nearest neighbors: the nearer, the more important.

Example 1: Demographic vs. crime variables

Table 4.1 Classification accuracies (%) of *k*-nearest neighbor searching computed after scaling or without it.

Method	*k*	Not scaled	Scaled into [0,1]	Standardized
Nearest neighbor searching	1	58.9	73.2	73.2
	2	50	89.3	89.3
	3	51.8	89.3	89.3

Let us return to the data [3] of Example 3 in Section 3 for a while. Table 4.1 shows the results of nearest neighbor classification. Euclidean distance metric was applied. Exceptionally, *k* equal to 2 was used, because the smallest class contained 3 cases only.

[3] X. Li, H. Joutsijoki, J. Laurikkala and M. Juhola: Crime vs. demographic factors: application of data mining methods, Webology, Article 132, 12(1), 1-19, 2015.

For the non-scaled data, *k* equal to 2 was the poorest choice. Otherwise, *k* equal to 1 was the poorest.

The influence of normalization (scaling or standardization) was notable producing essentially better classification accuracies. The best of all, 89.3%, is higher than that of naïve Bayes with 80.4%.

Example 2: Vertigo data

The class sizes (numbers of cases) were 130 (16%), 146 (18%), 313 (38%), 41 (5%), 65 (8%), and 120 (15%) cases in six classes.

Let us view Vertigo data[4] from Section 3 (p. 124-125). We obtained high true positive rates shown in Table 4.2.

[4] M. Juhola: Data Classification, Encyclopedia of Computer Science and Engineering, ed. B. Wah, John Wiley & Sons, 2008 (print version, 2009, Hoboken, NJ), 759-767.

Table 4.2 Average true positive rates and accuracies [%] of vertigo data (815 cases and 6 classes) given by nearest neighbor classification when 10 times 10-fold cross validation are computed.

k	Vestibular schwannoma	Benign positional vertigo	Meniere's disease	Sudden deafness	Traumatic vertigo	Vestibular neuritis	Accuracy
1	88.5	80.7	87.6	92.5	82.6	86.7	86.2
3	89.2	77.3	87.9	92.5	78.8	84.2	85.2
5	79.2	83.3	89.5	90.5	80.2	90.0	86.1