Chapter 5
Self-Organizing Maps

In PCA, the most outlying data points determine the direction of the PCs—these are the ones contributing most to the variance. This often results in score plots showing a large group of points close to the center. As a result, any local structure is hard to recognize, even when zooming in: such points are not important in the determination of the PCs. One approach is to select the rows of the data matrix corresponding to these points, and to perform a separate PCA on them. Apart from the obvious difficulties in deciding which points to leave out and which to include, this leads to a cumbersome and hard to interpret two-step approach. It would be better if a projection can be found that *does* show structure, even within very similar groups of points.

Self-organizing maps (SOMs, Kohonen 2001), sometimes also referred to as Kohonen maps after their inventor, Teuvo Kohonen, offer such a view. Rather than providing a continuous projection into $\mathbb{R}^2$, SOMs map all data to a set of discrete locations, organized in a regular grid. Associated with every location is a prototypical object, called a *codebook vector*. This usually does not correspond to any particular object, but rather represents part of the space of the data. The complete set of codebook vectors therefore can be viewed as a concise summary of the original data. Individual objects from the data set can be mapped to the set of positions, by assigning them to the unit with the most similar codebook vectors.

The effect is shown in Fig. 5.1. A two-dimensional point cloud is simulated where most points are very close to the origin.[1] The codebook vectors of a 5-by-5 rectangular SOM are shown in black; neighboring units in the horizontal and vertical directions are connected by lines. Clearly, the density of the codebook vectors is greatest in areas where the density of points is greatest. When the codebook vectors are shown at their SOM positions the plot on the right in Fig. 5.1 emerges, where individual objects are shown at a random position close to "their" codebook vector. The codebook vectors in the middle of the map are the ones that cover the center of the data density, and

[1]The point cloud is a superposition of two bivariate normal distributions, centered at the origin and with diagonal covariance matrices. The first has unit variance and contains 100 points; the other, containing 500 points, has variances of 0.025.

© Springer-Verlag GmbH Germany, part of Springer Nature 2020

R. Wehrens, *Chemometrics with R*, Use R!,
https://doi.org/10.1007/978-3-662-62027-4_5

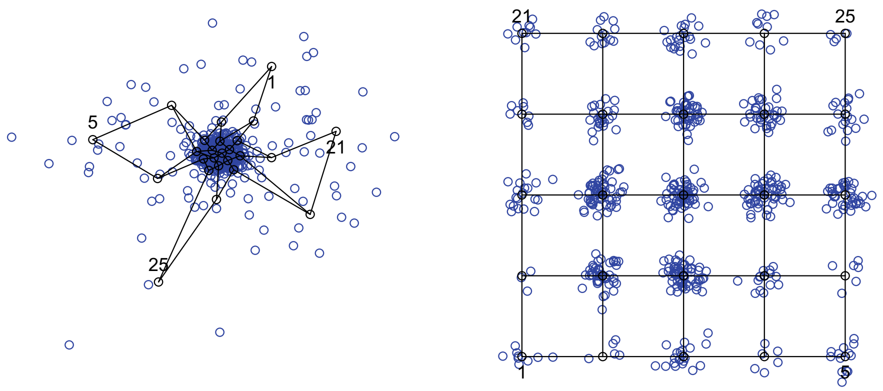

Fig. 5.1 Application of a 5-by-5 rectangular SOM to 600 bivariate normal data points. Left plot: location of codebook vectors in the original space. Right plot: location of data points in the SOM

one can see that these contain most data points. That is, relations within this densely populated area can be investigated in more detail.

5.1 Training SOMs

A SOM is trained by repeatedly presenting the individual samples to the map. At each iteration, the current sample is compared to the codebook vectors. The most similar codebook vector (the "winning unit") is then shifted slightly in the direction of the mapped object. This is achieved by replacing it with a weighted average of the old values of the codebook vector, cv_i, and the values of the new object obj:

$$cv_{i+1} = (1 - \alpha)\, cv_i + \alpha\, obj \tag{5.1}$$

The weight, also called the learning rate α, is a small value, typically in the order of 0.05, and decreases during training so that the final adjustments are very small.

As we shall see in Sect. 6.2.1, the algorithm is very similar in spirit to the one used in k-means clustering, where cluster centers and memberships are alternatingly estimated in an iterative fashion. The crucial difference is that not only the winning unit is updated, but also the other units in the "neighborhood" of the winning unit. Initially, the neighborhood is fairly large, but during training it decreases so that finally only the winning unit is updated. The effect is that neighboring units in general are more similar than units far away. Or, to put it differently, moving through the map by jumping from one unit to its neighbor would see gradual and more or less smooth transitions in the values of the codebook vectors. This is clearly visible in the mapping of the autoscaled wine data to a 5-by-4 SOM, using the **kohonen** package:

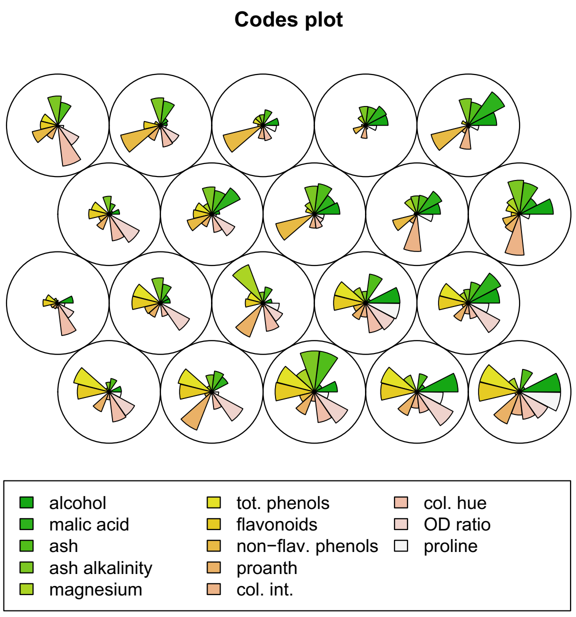

Fig. 5.2 Codebook vectors for a SOM mapping of the autoscaled wine data. The thirteen variables are shown counterclockwise, beginning in the first quadrant

```
> wines.som <- som(wines.sc, somgrid(5, 4, "hexagonal"))
> plot(wines.som, type = "codes")
```

The result is shown in Fig. 5.2. Units in this example are arranged in a hexagonal fashion and are numbered row-wise from left to right, starting from the bottom left. The first unit at the bottom left for instance, is characterized by relatively large values of `alcohol`, `flavonoids` and `proanth`; the second unit, to the right of the first, has lower values for these variables, but still is quite similar to unit number one.

The codebook vectors are usually initialized by a random set of objects from the data, but also random values in the range of the data can be employed. Sometimes a grid is used, based on the plane formed by the first two PCs. In practice, the initialization method will hardly ever matter; however, starting from other random initial values will lead to different maps. The conclusions drawn from the different maps, however, tend to be very similar.

The training algorithm for SOMs can be tweaked in many different ways. One can, e.g., update units using smaller changes for units that are further away from the winning unit, rather than using a constant learning rate within the neighborhood. One can experiment with different rates of decreasing values for learning rate and neighborhood size. One can use different distance measures. Regarding topology, hexagonal or rectangular ordering of the units is usually applied; in the first case, each unit has six equivalent neighbors, unless it is at the border of the map, in the second case, depending on the implementation, there are four or eight equivalent neighbors. The most important parameter, however, is the size of the map. Larger maps allow for more detail, but may contain more empty units as well. In addition, they take more time to be trained. Smaller maps are more easy to interpret; groups of units with similar characteristics are more easily identified. However, they may lack the flexibility to show specific groupings or structure in the data. Some experimentation usually is needed. As a rule of thumb, one can consider the object-to-unit ratio, which can lead to useful starting points. In image segmentation applications, for instance, where hundreds of thousands of (multivariate) pixels need to be mapped, one can choose a map size corresponding to an average of several hundreds of pixels per unit; in other applications where the number of samples is much lower, a useful object-to-unit ratio may be five. One more consideration may be the presence of class structure: for every class, several units should be allocated. This allows intra-class structure to be taken into account, and will lead to a better mapping.

Finally, there is the option to close the map, i.e., to connect the left and right sides of the map, as well as the bottom and top sides. This leads to a toroidal map, resembling the surface of a closed tube. In such a map, all differences between units have been eliminated: there are no more edge units, and they all have the same number of neighbors. Whereas this may seem a desirable property, there are a number of disadvantages. First, it will almost certainly be depicted as a regular map with edges, and when looking at the map one has to remember that the edges in reality do not exist. In such a case, similar objects may be found in seemingly different parts of the map that are, in fact, close together. Another pressing argument against toroidal maps is that in many cases the edges serve a useful purpose: they provide refuge for objects that are quite different from the others. Indeed, the corners of non-toroidal maps often contain the most distinct classes.

5.2 Visualization

Several different visualization methods are provided in the **kohonen** package: one can look at the codebook vectors, the mapping of the samples, and one can also use SOMs for prediction. Here, only a few examples are shown. For more information, consult the manual pages of the `plot.kohonen` function, or the software description (Wehrens and Buydens 2007; Wehrens and Kruisselbrink 2018).

For multivariate data, the locations of the codebook vectors can not be visualized as was done for the two-dimensional data in Fig. 5.1. In the **kohonen** package, the default is to show `segment` plots, such as in Fig. 5.2 if the number of variables is smaller than 15, and a line plot otherwise. One can also zoom in and concentrate on the values of just one of the variables:

```
> for (i in c(1, 8, 11, 13))
+   plot(wines.som, "property",
+        property = getCodes(wines.som, 1)[, i],
+        main = colnames(wines)[i])
```

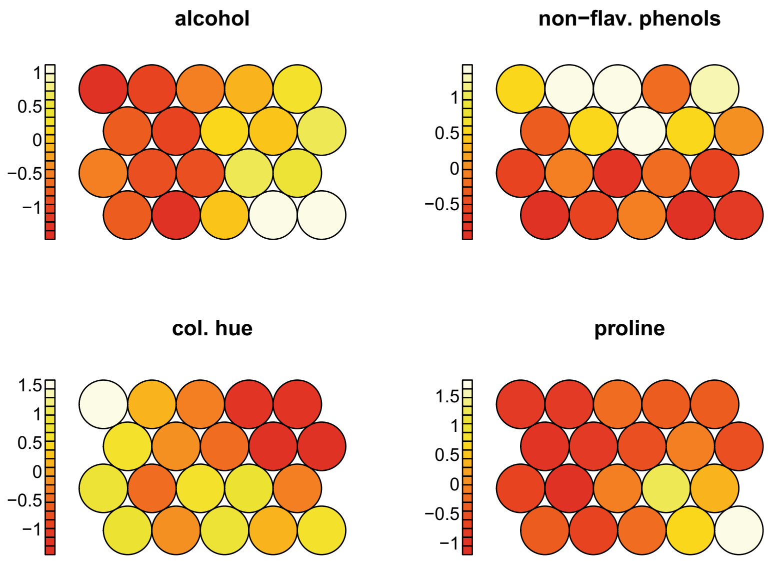

Fig. 5.3 Separate maps for the contributions of individual variables to the codebook vectors of the SOM shown in Fig. 5.2

Clearly, in these plots, shown in Fig. 5.3, there are regions in the map where specific variables have high values, and other regions where they are low. Areas of high values and low values are much more easily recognized than in Fig. 5.2. Note the use of the accessor function `getCodes` here.

Perhaps the most important visualization is to show which objects map in which units. In the **kohonen** package, this is achieved by supplying the the `type = "mapping"` argument to the plotting function. It allows for using different plotting characters and colors (see Fig. 5.4):

```
> plot(wines.som, type = "mapping",
+      col = as.integer(vintages), pch = as.integer(vintages))
```

Again, one can see that the wines are well separated. Some class overlap remains, especially for the Grignolinos (pluses in the figure). These plots can be used to make predictions for new data points: when the majority of the objects in a unit are, e.g., of the Barbera class, one can hypothesize that this is also the most probably class for future wines that end up in that unit. Such predictions can play a role in determining authenticity, an economically very important application.

Since SOMs are often used to detect grouping in the data, it makes sense to look at the codebook vectors more closely, and investigate if there are obvious class boundaries in the map—areas where the differences between neighboring units are relatively large. Using a color code based on the average distance to neighbors one can get a quick and simple idea of where the class boundaries can be found. This

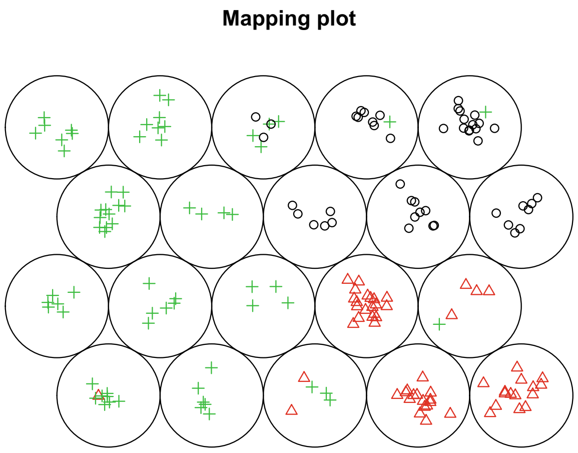

Fig. 5.4 Mapping of the 177 wine samples to the SOM from Fig. 5.2. Circles correspond to Barbera, triangles to Barolo, and pluses to Grignolino wines

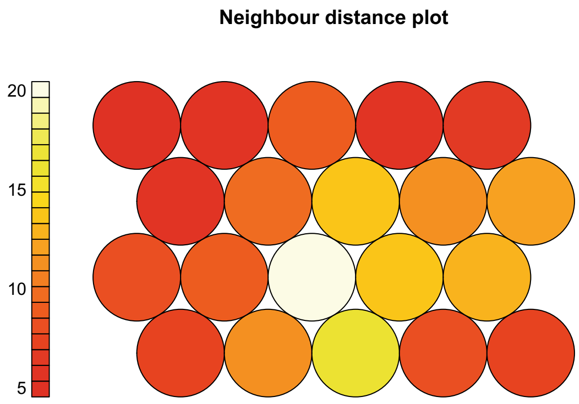

Fig. 5.5 Summed distances to direct neighbors: the U-matrix plot for the mapping of the wine data

idea is often referred to as the "U-matrix" (Ultsch 1993), and can be employed by issuing:

```
> plot(wines.som, type = "dist.neighb")
```

The resulting plot is shown in Fig. 5.5. The map is too small to really be able to see class boundaries, but one can see that the centers of the classes (the bottom left corner for Barbera, the bottom right corner for Barolo, and the top row for the Grignolino variety) correspond to areas of relatively small distances, i.e., high homogeneity.

Training progress, and an indication of the quality of the mapping, can be obtained using the following plotting commands:

```
> par(mfrow = c(1, 2))
> plot(wines.som, "changes")
> plot(wines.som, "quality")
```

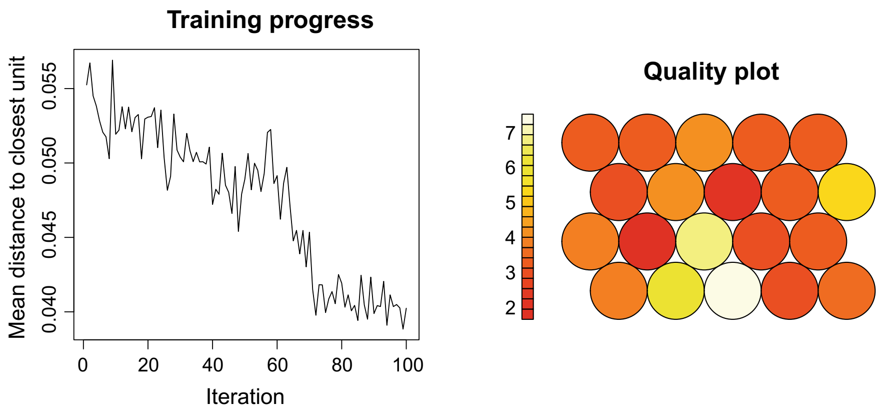

Fig. 5.6 Quality parameters for SOMs: the plot on the left shows the decrease in distance between objects and their closest codebook vectors during training. The plot on the right shows the mean distances between objects and codebook vectors per unit

This leads to the plots in Fig. 5.6. The left plot shows the average distance (expressed per variable) to the winning unit during the training iterations, and the right plot shows the average distance of the samples and their corresponding codebook vectors after training. Note that the latter plot concentrates on distances *within* the unit whereas the U-matrix plot in Fig. 5.5 visualizes average distances *between* neighboring units.

Finally, an indication of the quality of the map is given by the mean distances of objects to their units:

```
> summary(wines.som)
SOM of size 5x4 with a hexagonal topology
  and a bubble neighbourhood function.
The number of data layers is 1.
Distance measure(s) used: sumofsquares.
Training data included: 177 objects.
Mean distance to the closest unit in the map: 3.646.
```

The `summary` function indicates that an object, on average, has a distance of 3.6 units to its closest codebook vector. The plot on the left in Fig. 5.6 shows that the average distance drops during training: codebook vectors become more similar to the units that are mapped to them. The plot on the right, finally, shows that the distances within units can be quite different. Interestingly, some of the units with the largest spread only contain Grignolinos (units 2 and 8), so the variation can not be attributed to class overlap alone.

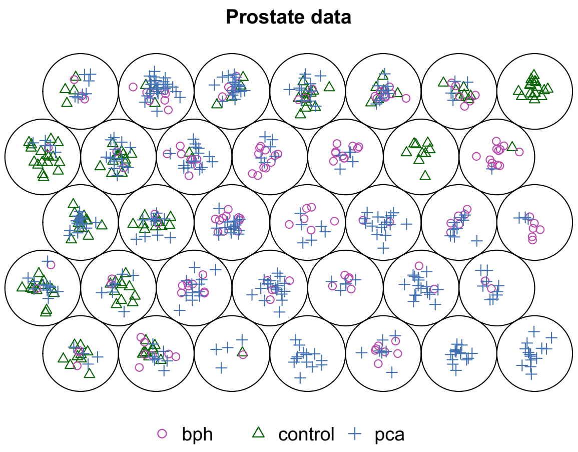

Fig. 5.7 Mapping of the prostate data. The cancer samples (`pca`) lie in a broad band from the bottom right to the top of the map. Control samples are split in two groups on either side of the `pca` samples. There is considerable class overlap

5.3 Application

The main attraction of SOMs lies in the applicability to large data sets; even if the data are too large to be loaded in memory in one go, one can train the map sequentially on (random) subsets of the data. It is also possible to update the map when new data points become available. In this way, SOMs provide a intuitive and simple visualization of large data sets in a way that is complementary to PCA. An especially interesting feature is that these maps can show grouping of the data without explicitly performing a clustering. In large maps, sudden transitions between units, as visualized by, e.g., a U-matrix plot, enable one to view the major structure at a glance. In smaller maps, this often does not show clear differences between groups—see Fig. 5.5 for an example. One way to find groups is to perform a clustering of the individual codebook vectors. The advantage of clustering the codebook vectors rather than the original data is that the number of units is usually orders of magnitude smaller than the number of objects.

As a practical example, consider the mapping of the 654 samples from the `prostate` data using the complete, 10,523-dimensional mass spectra in a 7-by-5 map. This would on average lead to almost twenty samples per unit and, given the fact that there are three classes, leave enough flexibility to show within-class structure as well:

```
> X <- t(Prostate2000Raw$intensity)
> prostate.som <- som(X, somgrid(7, 5, "hexagonal"))
```

The plot in Fig. 5.7 is produced with the following code:

```
> types <- as.integer(Prostate2000Raw$type)
> trellis.cols <- trellis.par.get("superpose.symbol")$col[c(2, 3, 1)]
> plot(prostate.som, "mapping", col = trellis.cols[types],
+      pch =  types, main = "Prostate data")
```

```
> legend("bottom", legend = levels(Prostate2000Raw$type),
+        col = trellis.cols, pch = 1:3, ncol = 3, bty = "n")
```

Clearly, there is considerable class overlap, as may be expected when calculating distances over more than 10,000 variables. Some separation can be observed, however, especially between the cancer and control samples. To investigate differences between the individual units, one can plot the codebook vectors of some of the units containing (predominantly) objects from one class only, corresponding to the three right-most units in the plot in Fig. 5.7:

```
> units <- c(7, 21, 35)
> unitfs <- paste("Unit", units)
> prost.plotdf <-
+   data.frame(mz = Prostate2000Raw$mz,
+              intensity = c(t(getCodes(prostate.som, 1)[units, ])),
+              unit = rep(factor(unitfs, levels = unitfs),
+                         each = length(Prostate2000Raw$mz)))
> xyplot(intensity ~ mz | unit, data = prost.plotdf, type = "l",
+        scale = list(y = "free"), as.table = TRUE,
+        xlab = bquote(italic(.("m/z"))~.("(Da)")),
+        groups = unit, layout = c(1, 3),
+        panel = function(...) {
+          panel.abline(v = c(3300, 4000, 6000, 6200),
+                       col = "gray", lty = 2)
+          panel.xyplot(...)
+        })
```

These codebook vectors, shown in Fig. 5.8, display appreciable differences. The cancer samples from Unit 7, for instance, are missing the large peaks at 3,100 and 4,000 Da that are present in the other two units but contain very clear signals around 6,100 and 6,200 Da, where the others have nothing.

5.4 R Packages for SOMs

The **kohonen** package used in this chapter, originally based on the **class** package (Venables and Ripley 2002), has several noteworthy features not discussed yet (Wehrens and Kruisselbrink 2018). It can use distance functions other than the usual Euclidean distance, which might be extremely useful for some data sets, often avoiding the need for prior data transformations. One example is the WCC function mentioned earlier: this can be used to group sets of X-ray powder diffractograms where the position rather than the position of peaks contains the primary information (Wehrens and Willighagen 2006; Wehrens and Kruisselbrink 2018). For numerical variables, the sum-of-squares distance is the default (slightly faster than the Euclidean distance); for factors, the Tanimoto distance. In the **kohonen** package

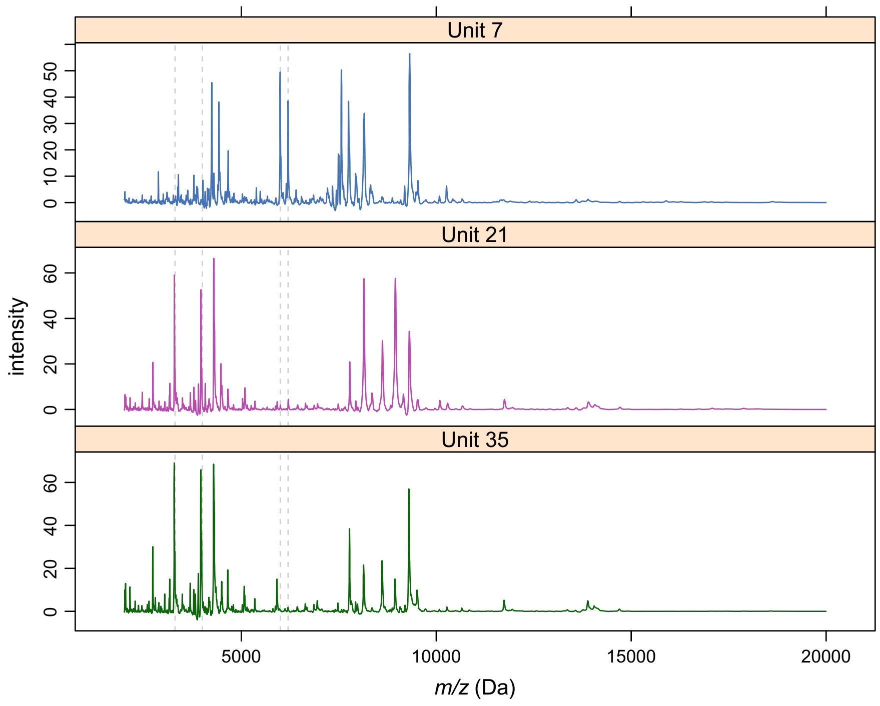

Fig. 5.8 Codebook vectors for three units from the far-right side of the map in Fig. 5.7, containing only samples from one class: unit 7 contains `pca` samples, unit 21 mostly `bph` samples and unit 35 `control` samples. Vertical gray lines indicate mass-to-charge ratios mentioned in the text

it is possible to supply several different data layers, where the rows in each layer correspond to different bits of information on the same objects. Separate distance functions can be defined for each single layer, which are then combined into one overall distance measure using weights that can be defined by the user. Apart from the usual "online" training algorithm described in this chapter, a "batch" algorithm is implemented as well, where codebook vectors are not updated until all records have been presented to the map. One advantage of the batch algorithm is that it dispenses with one of the parameters of the SOM: the learning rate α is no longer needed. The main disadvantage is that it is sometimes less stable and more likely to end up in a local optimum. The batch algorithm also allows for parallel execution by distributing the comparisons of objects to all codebook vectors over several cores (Lawrence et al. 1999) which may lead to considerable savings with larger data sets (Wehrens and Kruisselbrink 2018).

Several other packages are available from repositories like CRAN. One example is the **som** package (Yan 2016). This package implements the online and batch algorithms and provides great flexibility in setting training parameters. The **somoclu** package implements a general SOM toolbox supporting parallel computation, also on GPUs (Wittek et al. 2017). It uses the **kohonen** plotting functions for visualization. A package providing a **shiny** (Chang et al. 2018) web interface is **SOMbrero** (Olteanu and Villa-Vialaneix 2015). Here, one can use numerical data, contingency tables as well as distance matrices as primary input data. Finally, the Stuttgart Neural Network Simulator (**SNNS**) provides SOMs as one of many types of neural networks (Bergmeir and Benítez 2012).

5.5 Discussion

Conceptually, the SOM is most related to MDS, seen in Sect. 4.6.1. Both, in a way, aim to find a configuration in two-dimensional space that represents the distances between the samples in the data. Whereas metric forms of MDS focus on the preservation of the actual distances, SOMs provide a topological mapping, preserving the *order* of the distances, at least for the smallest ones. Because of this, an MDS mapping is often dominated by the larger distances, even when using methods like Sammon mapping, and the configuration of the finer structure in the data may not be well preserved. In SOMs, on the other hand, a big distance between the positions of two samples in the map does not mean that they are very dissimilar: if the map is too large, and not well trained, two regions in the map that are far apart may very well have quite similar codebook vectors. What one *can* say is that objects mapped to the same or to neighboring units are likely to be similar.

Both MDS and SOMs operate using distances rather than the original data to determine the position in the low-dimensional representation of the data. This can be a considerable advantage when working with high-dimensional data: even when the number of variables is in the tens or hundreds of thousands, the distances between objects can be calculated fairly quickly. Obviously, MDS, in particular, runs into trouble when the number of samples gets large—SOMs can handle that more easily because of the iterative training procedure employed. It is not even necessary to have all the data in memory simultaneously.

Using SOMs is doubtful when the number of samples is low, although applications have been published with fewer than fifty objects. If the number of units in the map is much smaller than the number of objects in such cases, one loses the advantage of the spatial smoothness in the map, and one could just as well perform a clustering; if the number of units approaches the number of objects, it is more likely than not that the majority of the objects will occupy a unit by itself, which is not very informative either.

One should realize that in the case of correlated variables the distances that are calculated may be a bit biased: a group of highly correlated variables will have a major influence on the distance between objects. In areas like, e.g., quantitative structureactivity relationships (QSAR), it is usual to calculate as many chemical structure descriptors as possible in order to define the two- or three-dimensional structure of a set of compounds. Many of these descriptors are variations on a theme: some groups measure properties related to dipole, polarizability, surface area, etcetera. The influence of one single descriptor capturing information that is unrelated to the hundreds of other descriptors can easily be lost when calculating distances. For SOMs, one simple solution is to decorrelate the (scaled) data, e.g., using PCA, and to calculate distances using the scores.