Journal of Statistical Software

October 2007, Volume 21, Issue 5. http://www.jstatsoft.org/

Self- and Super-organizing Maps in R: The kohonen Package

Ron Wehrens
Radboud University Nijmegen

Lutgarde M. C. Buydens
Radboud University Nijmegen

Abstract

In this age of ever-increasing data set sizes, especially in the natural sciences, visualisation becomes more and more important. Self-organizing maps have many features that make them attractive in this respect: they do not rely on distributional assumptions, can handle huge data sets with ease, and have shown their worth in a large number of applications. In this paper, we highlight the **kohonen** package for R, which implements self-organizing maps as well as some extensions for supervised pattern recognition and data fusion.

Keywords: self-organizing maps, visualisation, classification, clustering.

1. Introduction

Due to advancements in computer hardware and software, as well as in measurement instrumentation, researchers in many scientific fields are able to collect and analyze datasets of steadily increasing size and dimensionality. Exploratory analysis, where visualization plays a very important role, has become more and more difficult given the increasing dimensionality of the data. There is a real need for methods that provide meaningful mappings into two dimensions, so that we can fully utilize the pattern recognition capabilities of our own brains. There are many approaches to mapping a high dimensional data set into two dimensions, of which principal component analyis (PCA, Jackson 1991) is probably the most used. However, in many cases more than two dimensions are needed to provide a reasonably useful picture of the data so that visualisation remains a problem. Moreover, PCA in its pure form does not incorporate information on how objects should be compared; the standard Euclidean distance measure is not always the optimal dissimilarity measure. Methods starting from distance or similarity matrices, in this respect, may prove more useful. First of all, such methods scale well to large numbers of variables, and second, by choosing an appropriate distance function

for the data at hand, one concentrates on those aspects of the data that are most informative.

One approach to the visualization of a distance matrix in two dimensions is multi-dimensional scaling (MDS) and its many variants (Cox and Cox 2001). This technique aims to find a configuration in two-dimensional space whose distance matrix in some sense approaches the original distance matrix, calculated from the high-dimensional data. The procedure may end up in a local optimum and may have to be performed several times, which for larger numbers of objects can be quite tedious. Moreover, there is no simple way to project new objects into the same space.

Self-organizing maps (SOMs, Kohonen 2001) tackle the problem in a way similar to MDS, but instead of trying to reproduce distances they aim at reproducing topology, or in other words, they try to keep the same neighbours. So if two high-dimensional objects are very similar, then their position in a two-dimensional plane should be very similar as well. Rather than mapping objects in a continuous space, SOMs use a regular grid of "units" onto which objects are mapped. The differences with MDS can be seen as both strengths and weaknesses: where in a 2D MDS plot a distance – also a large distance – can be directly interpreted as an "estimate" of the true distance, in a SOM plot this is not the case: one can only say that objects mapped to the same, or neighbouring, units are very similar. In other words, SOMs concentrate on the largest *similarities*, whereas MDS concentrates on the largest *dissimilarities*. Which of these is more useful depends on the application.

SOMs have seen many diverse applications in a broad range of fields, such as medicine, biology, chemistry, image analysis, speech recognition, engineering, computer science and many more; a bibliography of more than 7,700 papers is available at the website of the university where Teuvo Kohonen initiated the SOM research (Helsinki University of Technology – CIS Laboratory 2007). From the same research group one can obtain C source code for SOMs and a MATLAB-based package (Helsinki University of Technology – CIS Laboratory 2006). For R (R Development Core Team 2007), two packages are available from the Comprehensive R Archive Network (CRAN, `http://CRAN.R-project.org/`) implementing standard SOMs: the **som** package, by (Yan 2004), and the **class** package, part of the **VR** package bundle (Venables and Ripley 2002).

The latter provided the basis for the **kohonen** package, described in this paper and also available from CRAN. The **kohonen** package features standard SOMs and two extensions that make it possible to use SOMs for classification and regression purposes, and for data mining. It provides extensive graphics to enable what is for us the most basic function of SOMs, namely visualization and information packaging. A MATLAB implementation of several of the tools discussed in this paper, written by Willem Melssen, is available from `http://www.cac.science.ru.nl/software/`.

Section 2 gives a very brief description on the theory of SOMs. In Section 3 and following sections, the **kohonen** package is described in some detail. Section 7 contains some plans for the near future.

2. Theory

In a sense, SOMs can be thought of as a spatially constrained form of k-means clustering (Ripley 1996). In that analogy, every unit corresponds to a "cluster", and the number of clusters is defined by the size of the grid, which typically is arranged in a rectangular or

hexagonal fashion. Indeed, one of the training strategies for SOMs (the "batch" algorithm, an implementation of which is found in the **class** package as the function `batchSOM`) is very similar to k-means.

One starts by assigning a so-called *codebook vector* to every unit, that will play the role of a typical pattern, a prototype, associated with that unit. Usually, one randomly assigns a subset of the data to the units. During training, objects are repeatedly presented – in random order – to the map. The "winning unit", i.e., the one most similar to the current training object, will be updated to become even more similar; a weighted average is used, where the weight of the new object is one of the training parameters of the SOM. Also referred to as the learning rate α, it typically is a small value in the order of 0.05. During training, this value decreases so that the map converges. The spatial constraint mentioned before lies in the fact that SOMs require neighbouring units to have similar codebook vectors. This is achieved by not only updating the winning unit, but also the units in the immediate neighbourhood of the winning unit, in the same way. The size of the neighbourhood decreases during training as well, so that eventually (in our implementation after one-third of the iterations) only the winning units are adapted. At that stage, the procedure is exactly equal to k-means. The algorithm terminates after a predefined number of iterations. More information can be found in the book of Kohonen (2001).

The algorithm is very simple and allows for many subtle adaptations. One can experiment with different distance measures, different map topologies, different training parameters (learning rate and neighbourhood), etcetera. Even with identical settings, repeated training of a SOM will lead to sometimes even quite different mappings, because of the random initialisation. However, in our experience the conclusions drawn from the map remain remarkably consistent, which makes it a very useful tool in many different circumstances. Nevertheless, it is always wise to train several maps before jumping to conclusions.

The classical description of SOMs given above focusses on unsupervised exploratory analysis. However, SOMs can be used as supervised pattern recognizers, too. This means that additional information, e.g., class information, is available that can be modelled as a dependent variable for which predictions can be obtained. The original data are often indicated with X; the additional information with Y. The oldest and simplest approach is to assess the extra information only after the training phase; if a continuous variable is being predicted, the mean of all objects mapped to a specific unit can be seen as the estimate for that unit. In case of a class variable, a winner-takes-all strategy is often used. Note that in this approach – in chemistry applications often called a Counter-Propagation Network (Zupan and Gasteiger 1999) – the dependent variable Y does not influence the mapping.

Another strategy, already suggested by Kohonen (2001), is to perform SOM training on the concatenation of the X and Y matrices. Although this works in the more simple cases, it can be hard to find a suitable scaling so that X and Y both contribute to the similarities that are calculated. Melssen *et al.* (2006) proposed a more flexible approach where distances in X- and Y-space are calculated separately. Both are scaled so that the maximal distance equals 1, and the overall distance is a weighted sum of both:

$$D(o, u) = \alpha D_x(o, u) + (1 - \alpha) D_y(o, u)$$

where $D(o, u)$ indicates the combined distance of an object o to unit u, and D_x and D_y indicate the distances in the individual spaces. Choosing $\alpha = 0.5$ leads to equal weights for both X- and Y-spaces. Scaling so that the maximum distances in X- and Y-spaces equal

one takes care of possible differences in units between X and Y. Training the map is done as usual; the winning unit and its neighbourhood are updated, and during training the learning rate and the size of the neighbourhood are decreased. The final result consists of two maps: one map for the X variables, and one for the Y variables. For supervised SOMs, one extra parameter, the weight for the X (or Y) space needs to be defined by the user.

This principle can be extended to more layers as well; in that case we refer to the result as a super-organized map. For every layer a similarity value is calculated, and all individual similarities then are combined into one value that is used to determine the winning unit:

$$D(o, u) = \sum_i \alpha_i D_i(o, u)$$

where the weights α_i are scaled to unit sum. These weights are the only extra parameters (compared to classical SOMs) that need to be defined by the user.

3. The kohonen package for R

The R package **kohonen** aims to provide simple-to-use functions for self-organizing maps and the above-mentioned extensions, with specific emphasis on visualisation. The basic functions are `som`, for the usual form of self-organizing maps; `xyf`, for supervised self-organizing maps, or X-Y fused maps, which are useful when additional information in the form of, e.g., a class variable is available for all objects; `bdk`, an alternative formulation called bi-directional Kohonen maps; and finally, from version 2.0.0 on, the generalisation of the `xyf` maps to more than two layers of information, in the function `supersom`. These functions can be used to define the mapping of the objects in the training set to the units of the map.

After the training phase, one can use several plotting functions for the visualisation; the package can show where objects are mapped, has several options for visualizing the codebook vectors of the map units, and provides means to assess the training progress. Summary functions exist for all SOM types. Furthermore, one can easily project new data into the trained map; this provides possibilities for property estimation.

Several data sets are included in the **kohonen** package: the wine data from the UCI Machine Learning repository (Asuncion and Newman 2007), near-infrared spectra from ternary mixtures of ethanol, water and iso-propanol, measured at different temperatures described by Wülfert *et al.* (1998), and finally a set of microarray data, the well-known yeast data from Spellman *et al.* (1998). The wine data set contains information on a set of 177 Italian wine samples from three different grape cultivars; thirteen variables (such as concentrations of alcohol and flavonoids, but also color hue) have been measured. The yeast data are a subset of the original set containing 6178 genes, that are assumed to be related to the yeast cell cycle. The set contains 800 genes for which, using six different synchronisation methods, time-dependent expressions have been measured. A summary of the functions and data sets available in the package is given in Table 1.

Below, we will elaborate on the different stages in an exploratory analysis using self-organizing maps. We will use the data sets available in the package, so that the reader can easily reproduce and extend these examples. The all-important visualisation possibilities of the package are introduced along the way.

Function name	Short description
`som`	standard SOM
`xyf`	supervised SOM: two parallel maps
`bdk`	supervised SOM: two parallel maps (alternative formulation)
`supersom`	SOM with multiple parallel maps
`plot.kohonen`	generic plotting function
`summary.kohonen`	generic summary function
`map.kohonen`	map data to the most similar unit
`predict.kohonen`	generic function to predict properties
`wines`	wine data: a 177-by-13 matrix
`nir`	NIR spectra of 95 ternary mixtures
`yeast`	microarray data of the yeast cell cycle

Table 1: The main functions and data sets in the **kohonen** package, version 2.0.0.

4. Creating the maps

The different types of self-organizing maps can be obtained by calling the functions `som`, `xyf` (or `bdk`), or `supersom`, with the appropriate data representation as the first argument(s). Several other arguments provide additional parameters, such as the map size, the number of iterations, etcetera. The object that is returned can then be used for inspection, plotting, mapping, and prediction.

4.1. Self-organizing maps: Function `som`

The standard form of self-organizing maps is implemented in function `som`. To map the 177-sample wine data set to a map of five-by-four hexagonally oriented units, the following code can be used. First, we load the package (from now on, we assume the package is loaded), and then the data, which are subsequently autoscaled because of the widely different ranges (especially the proline concentration, variable 13, deviates). The fourteenth variable is a class variable and is not used in the mapping; it will be used later for visualisation purposes. To allow readers to exactly reproduce the figures in this paper, we set a random seed, and then train the network.

```
R> library("kohonen")
Loading required package: class
R> data("wines")
R> wines.sc <- scale(wines)
R> set.seed(7)
R> wine.som <- som(data = wines.sc, grid = somgrid(5, 4, "hexagonal"))
R> plot(wine.som, main = "Wine data")
```

The result is shown in Figure 1. The codebook vectors are visualized in a segments plot, which is the default plotting type. High alcohol levels, for example, are associated with wine samples projected in the bottom right corner of the map, while color intensity is largest in the bottom left corner. More plotting types will be discussed below.

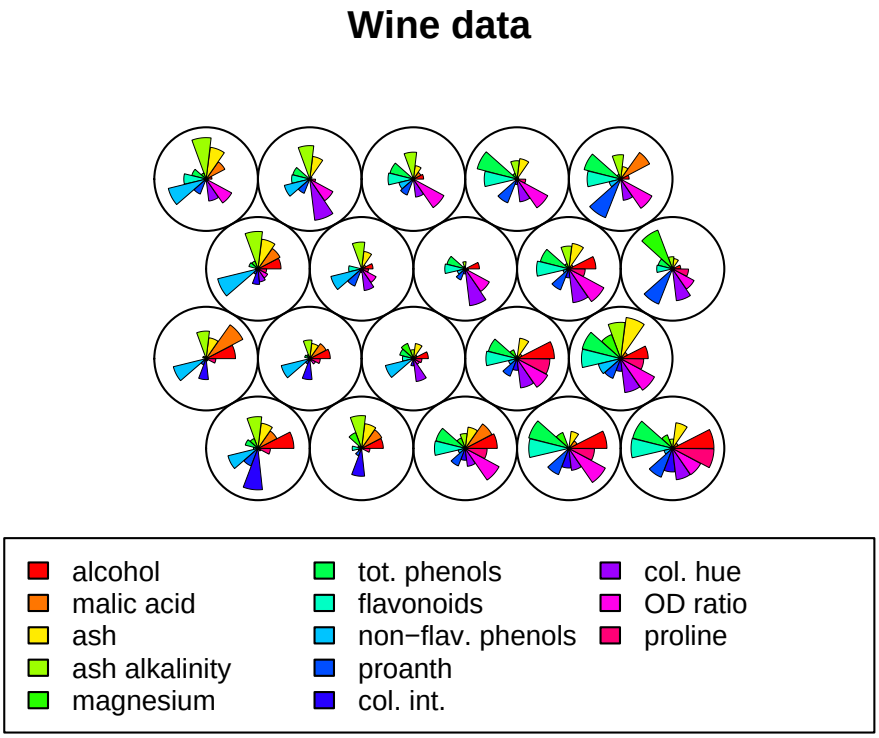

Figure 1: A plot of the codebook vectors of the 5-by-4 mapping of the wine data.

The `som` function has several parameters. Default values are available for all of them, except the first, the data. Because the training parameters appear in the other SOM functions as well, we mention them briefly below.

`grid`: the rectangular or hexagonal grid of units. The format is the one returned by the function `somgrid` from the **class** package.

`rlen`: the numer of iterations, i.e. the number of times the data set will be presented to the map. The default is 100;

`alpha`: the learning rate, determining the size of the adjustments during training. The decrease is linear, and default values are to start from 0.05 and to stop at 0.01;

`radius`: the initial size of the neighbourhood, by default chosen in such a way that two-thirds of all distances of the map units fall inside this number. The size of the neighbourhood decreases linearly during training; after one-third of the iterations only the winning unit is being adapted and the algorithm corresponds to k-means.

`init`: optional matrix of codebook vectors. If it is not given, randomly selected objects from the data are used. This feature can be useful when re-training a map with new data.

`toroidal`: by default, `FALSE`. If `TRUE`, the edges of the map are not real edges, and data are actually mapped to a torus. Put differently: opposite map edges are joined together.

`KeepData`: default value equals `TRUE`. However, for large data sets it may be too expensive to keep the data in the `som` object, and one may set this parameter to `FALSE`.

The result of the training, the `wine.som` object, is a list. The most important element is the `codes` element, which contains the codebook vectors as rows. Another element worth inspecting is `changes`, a vector indicating the size of the adaptions to the codebook vectors during training. This can be used to assess whether the number of iterations is sufficient.

4.2. Supervised mapping: The `xyf` function

Supervised mapping, where a dependent variable (categorical or continuous) is available, is implemented in the `xyf` function of the **kohonen** package. An example using the NIR data included in the package is shown below: for every ternary mixture, we have a near-infrared spectrum, as well as concentrations of the three chemical compounds (summing to 1). Moreover, every sample is measured at five different temperatures. The aim in the example below is to model the water content (the second of the three concentrations). Of the three chemicals, water has the largest effect on the NIR spectra. We start by loading the data and attaching the data frame so that objects `spectra`, `composition` and `temperature` become directly available. Parameter `xweight` indicates how much importance is given to X; here it is set to 0.5 (X and Y are equally important), also the default value in `xyf`.

```
R> data("nir")
R> attach(nir)
R> set.seed(13)
R> nir.xyf <- xyf(data = spectra,
+                 Y = composition[,2],
+                 xweight = 0.5,
+                 grid = somgrid(6, 6, "hexagonal"))
R> par(mfrow = c(1, 2))
R> plot(nir.xyf, type = "counts", main = "NIR data: counts")
R> plot(nir.xyf, type = "quality", main = "NIR data: mapping quality")
```

This leads to the output shown in Figure 2. In the left plot, the background color of a unit corresponds to the number of samples mapped to that particular unit; they are reasonably spread out over the map. Four of the units are empty: no samples have been mapped to them. The right plot shows the mean distance of objects, mapped to a particular unit, to the codebook vector of that unit. A good mapping should show small distances everywhere in the map.

An object generated by the `xyf` function has a few elements not found in the unsupervised case. The most important difference is the `codes` element, which itself now is a list containing the codebook vectors for both the X and Y maps. In the example above, the codebook matrix for Y only has one column (corresponding to the concentration of water). In addition, the `changes` element is now a matrix rather than a vector, with a column for both X and Y.

An alternative method, called Bi-Directional Kohonen mapping (Melssen *et al.* 2006) has been implemented in the `bdk` function; there, the meaning of the `xweight` parameter is slightly different – consult the manual page for details. Since the results are usually very similar to the ones obtained with the `xyf` implementation we will focus for the remainder of the paper on `xyf`.

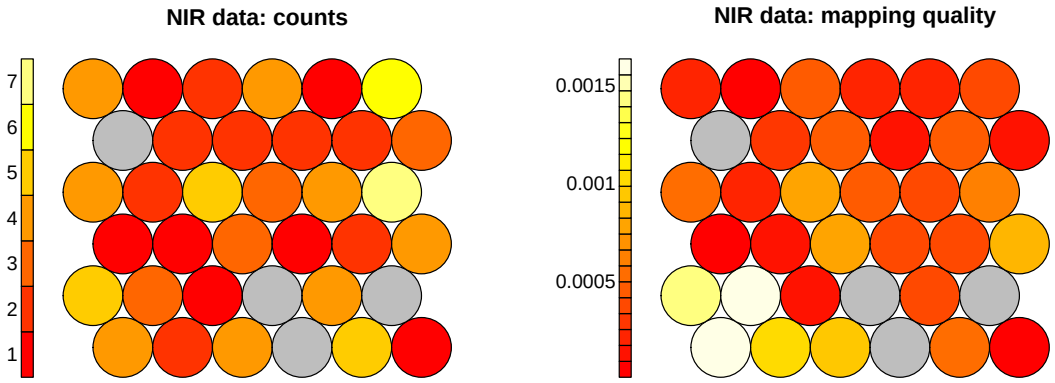

Figure 2: Counts plot of the map obtained from the NIR data using `xyf` (left plot). Empty units are depicted in gray. The right plot shows the quality of the mapping; the biggest distances between objects and codebook vectors are found in the bottom left of the map.

In case the dependent variable contains class information, we can present that to the `xyf` function in the form of a class matrix, where every column corresponds to a class, and where every object is represented by a row of zeros and one "1". The distance employed in such a case is the Tanimoto distance rather than the Euclidean distance. To convert between a vector of class names and a class matrix, the functions `classvec2classmat` and `classmat2classvec` are available. We could, e.g., train a map using the NIR spectra with temperature as a class variable:

```
R> set.seed(13)
R> nir.xyf2 <- xyf(data = spectra,
+                  Y = classvec2classmat(temperature),
+                  xweight = .2, grid = somgrid(6, 6, "hexagonal"))
```

Note that in this case we put more emphasis on the Y variable to enforce a spatial grouping of the different temperatures; for this data set it is necessary to do that since the influence of temperature on the spectra is only small.

Whether the dependent data should be seen as categorical or continuous is governed by the input parameter `contin`. By default, this is `FALSE` (indicating a categorical variable) when all row sums of Y equal 1; Y is then seen as a classification matrix. Note that when we want to model the concentrations of the three chemicals simultaneously, the Y matrix also sums to 1. Obviously, this is not a classification problem, and we can prevent the function from thinking it is by providing `contin = TRUE`.

We can add information to the plot by showing where every object is mapped, e.g., by plotting a symbol or a label. For the NIR data, we can make the symbol size dependent on the temperature at which the sample has been measured. In the left plot of Figure 3, the locations

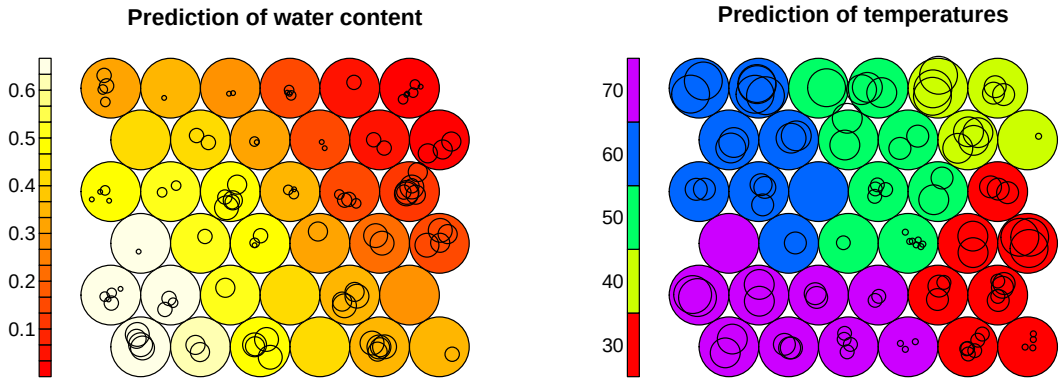

Figure 3: Supervised SOMs for the NIR data. The left plot shows the mapping of spectra when the XYF map has been trained with the water content as the y-variable; the radius of the circles indicate the temperature at which spectra have been measured. The right plot does the opposite: temperature is used as y-variable and is visualized using the background colors, and the radii of the circles indicate water content.

of the circles indicate the units onto which samples have been mapped – thus indicating an estimation of the water content – and the circle radii indicate measurement temperatures. The right plot shows the reverse situation: there, the map has been trained to predict temperature. Thus, the position of a circle says something about the expected temperature at which that sample has been measured, and the water concentration is indicated by the circle radius. This plot is obtained using the following code (adapted from the manual page of the `xyf` function):

```
R> water.predict <- predict(nir.xyf)$unit.prediction
R> temp.xyf <- predict(nir.xyf2)$unit.prediction
R> temp.predict <- as.numeric(classmat2classvec(temp.xyf))

R> par(mfrow = c(1, 2))
R> plot(nir.xyf, type = "property", property = water.predict,
+       main = "Prediction of water content", keepMargins = TRUE)
R> scatter <- matrix(rnorm(length(temperature)*2, sd=.1), ncol=2)
R> radii <- (temperature - 20)/250
R> symbols(nir.xyf$grid$pts[nir.xyf$unit.classif,] + scatter,
+          circles = radii, inches = FALSE, add = TRUE)

R> plot(nir.xyf2, type = "property", property = temp.predict,
+       palette.name = rainbow, main = "Prediction of temperatures")
R> scatter <- matrix(rnorm(nrow(composition)*2, sd=.1), ncol=2)
```

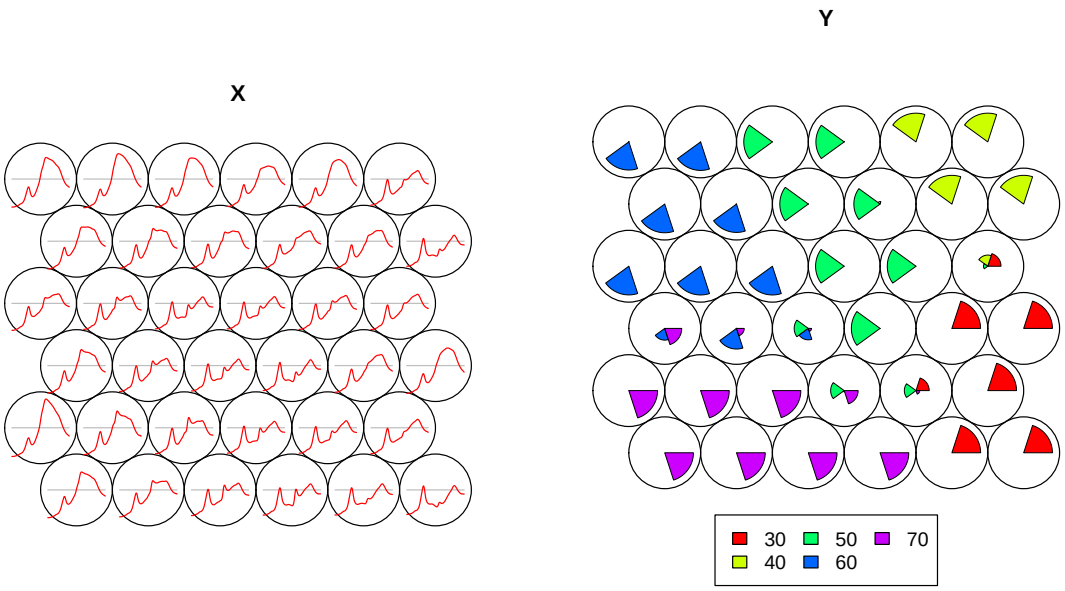

Figure 4: Plots of codebook vectors for the `xyf` mapping of IR spectra (X) and temperature (Y).

```
R> radii <- 0.05 + 0.4 * composition[,2]
R> symbols(nir.xyf2$grid$pts[nir.xyf2$unit.classif,] + scatter,
+          circles = radii, inches = FALSE, add = TRUE)
```

The `plot` functions themselves use the `property` argument to show the unit predictions in a background color; we have chosen a different palette in the second plot to distinguish also graphically between the prediction of a continuous variable in the left plot and a categorical variable in the right plot. Circles are added using the `symbols` function. Their location contains two components: first, the position of the unit onto which a sample is mapped (contained in the `unit.classif` list element), and, second, a random component within that unit. The `keepMargins = TRUE` argument, which prevents the original graphical parameters from being restored is needed here (the plotting functions may change margins, for example, to accomodate all units in the figure and allow for a title or a legend), since the circles need to be added to the correct position in the figure. Another application of setting `keepMargins = TRUE` is to find out the unit number by the combination of the `identify` function and clicking the map.

The plots in Figure 3 show that the modelled parameter, indicated with the background colors, indeed has a spatially smooth (or coherent) distribution. Moreover, in the prediction of the water content (the left plot in the figure), the samples are ordered in such a way that the low temperatures (the small circles) are located close together, as are the samples measured at high temperatures. In the right plot, this is even more clear. Within one color (one measurement temperature), there is a gradient of water concentrations.

In Figure 4 another use of the codebook vectors is shown. The R code to generate these

figures is quite simple:

```
R> par(mfrow = c(1, 2))
R> plot(nir.xyf2, "codes")
```

For large numbers of variables, the default behaviour is to make a line plot rather than a segment plot, which leads to the spectra-like patterns to the left. By comparing the codebook vectors with Figure 3 we can immediately associate spectral features with water content. In the right plot, the codebook vectors of the dependent variable (in this case the categorical temperature variable) are shown. These can be directly interpreted as an indication of how likely a given class is at a certain unit. Note that in some cases, such as at the boundaries between the higher temperatures, the classification of a unit is pretty uncertain.

4.3. Super-organized maps: Data fusion with `supersom`

Instead of one set of independent variables and one set of dependent variables, one may have several data types for every object. The super-organized map introduced here accounts for individual data types by using a separate layer for every type. Thus, when three types of spectroscopy have been performed on a set of samples for which class information is present as well, we could train a map using four layers. The first three would be continuous, and the codebook vectors for these maps would resemble spectra, and the fourth would be discrete where the codebook vectors can be interpreted as class memberships. A weight is associated to every layer to be able to define an overall distance of an object to a unit. Again, this allows much more flexibility than just concatenating the individual vectors corresponding to the different data entitites for every sample. Obvious choices for the weights are to set them all equal (the default in the **kohonen** package), or to set them according to the number of variables in the indicidual data sets.

We show an example, based on the well-known yeast cell-cycle data of Spellman *et al.* (1998). The mapping of these data, based on the four synchronisation methods with the largest numbers of time points, is shown in Figure 5. The `yeast` data set, included in the **kohonen** package, are already in the correct form: a list where each element is a data matrix with one row per gene. Note that the numbers of variables in the matrices need not be equal. The R code to generate the plot is as follows:

```
R> data(yeast)
R> set.seed(7)
R> yeast.supersom <- supersom(yeast, somgrid(8, 8, "hexagonal"),
+                             whatmap = 3:6)
Warning message:
removing 45 NA objects from the training data
 in: supersom(yeast, somgrid(8, 8, "hexagonal"), whatmap = 3:6)
R> classes <- levels(yeast$class)
R> colors <- c("yellow", "green", "blue", "red", "orange")
R> par(mfrow = c(3, 2))
R> plot(yeast.supersom, type = "mapping",
+       pch = 1, main = "All", keepMargins = TRUE)
R> for (i in seq(along = classes)) {
```

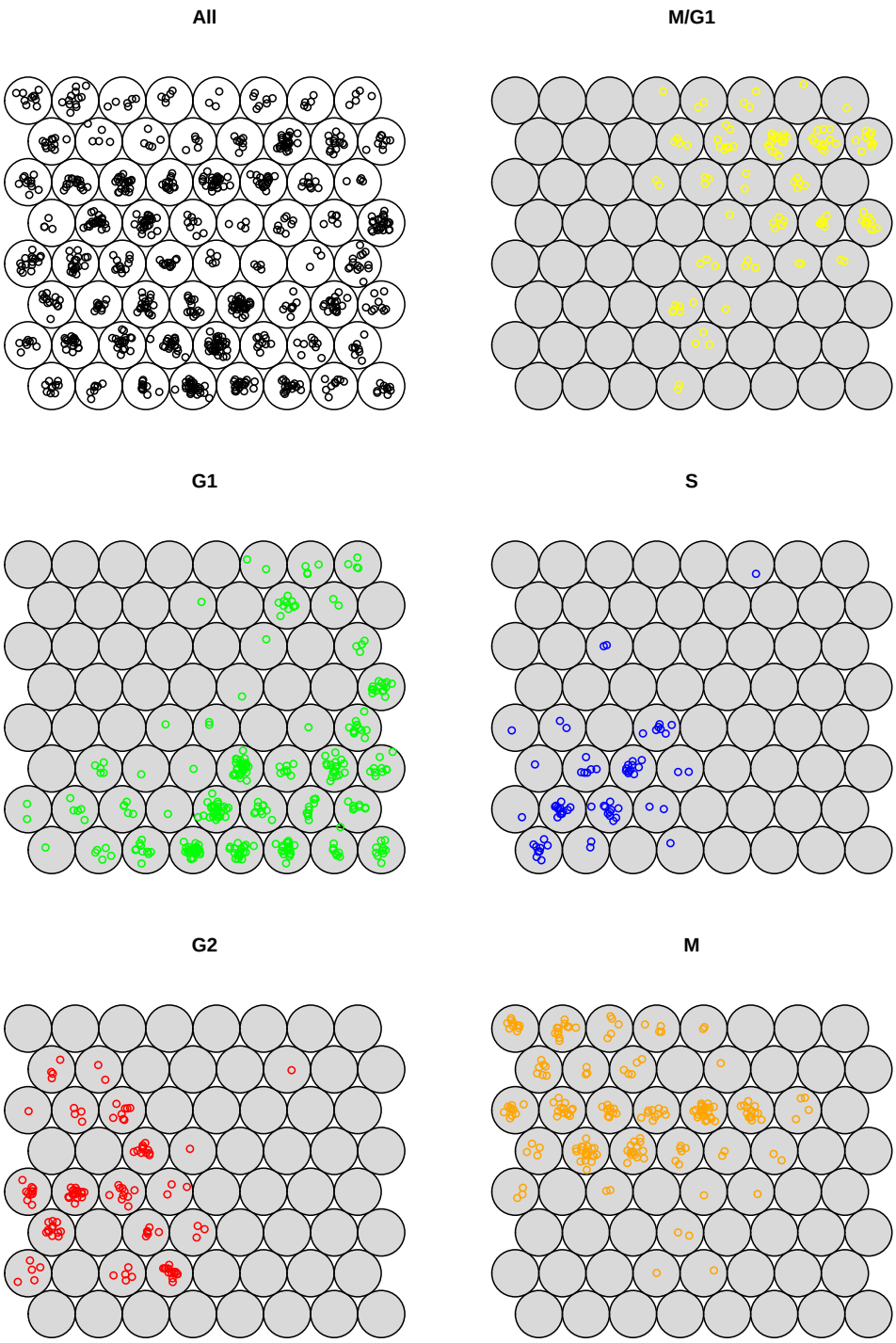

Figure 5: Mapping of the 800 genes in a eight-by-eight superSOM, based on the `alpha`, `cdc15`, `cdc28` and `elu` data (with equal weights for each of the four layers). The top left plot shows the position of all 800 genes; the other plots show the same mapping split up over the five cell-cycle stages.

```
+    X.class <- lapply(yeast,
+                      function(x) subset(x, yeast$class == classes[i]))
+    X.map <- map(yeast.supersom, X.class)
+    plot(yeast.supersom, type = "mapping", classif = X.map,
+         col = colors[i], pch = 1, main = classes[i], keepMargins = TRUE,
+         bgcol = gray(0.85))
+  }
```

The first and second elements of the yeast data, containing only two variables each, are not used in this mapping. This is indicated using the `whatmaps` argument of the `supersom` function; an alternative to achieve this is to explicitly define the weights for these entities as zero. The warning message indicates that for 45 genes, at least one of the elements in the `yeast` list contains only missing values; these genes have been removed from the data prior to training the superSOM. They are retained in the data, however, and may be mapped later, where the missing information will simply be ignored. The class labels, corresponding to five stages in the cell cycle, and coloring in the figure are chosen to match that of Spellman *et al.* (1998). The five classes are concentrated at specific areas in the map, and the order in the map (starting at the top right and traversing the map clockwise) corresponds with the cell cycle.

The `keepMargins` argument in the code generating Figure 5 is used to decide whether the original set of graphical parameters, in particular figure margins, must be restored or not. If `keepMargins` is `FALSE` (the default), a new figure will be plotted in the same way as before plotting the SOM. If, on the other hand, `keepMargins` is set to `TRUE`, the coordinate system of the map is retained. In that case, one can add symbols to the map, or use the `locator` function in R to find the number of a particular unit, for example. The easiest way to subsequently generate a new plot with the default settings is to open a new graphics window.

5. Inspecting the maps

5.1. Queries and summaries

Generic `print` and `summary` methods are available: the `print.kohonen` function state the size of the map, the training method and whether or not the training data have been included in the map object. If the data are included, the `summary.kohonen` method provides information on the data, as well as an indication of the mapping quality, as estimated by the average distance of an object to its corresponding codebook vector. Of course, the individual elements of the map objects can be inspected to obtain more detailed information.

5.2. Plotting

Several plotting functions have already been shown in the examples above; in particular, plotting types `codes`, `counts`, `property`, `quality` and `mapping`. For more examples and possibilities with these plotting types, consult the manual pages of the package.

One general plot showing the training progress has not yet been mentioned. During training, the codebook vectors are becoming more and more similar to the closest objects in the data set. Visualisation of this process can be used to optimize training parameters; for instance, it

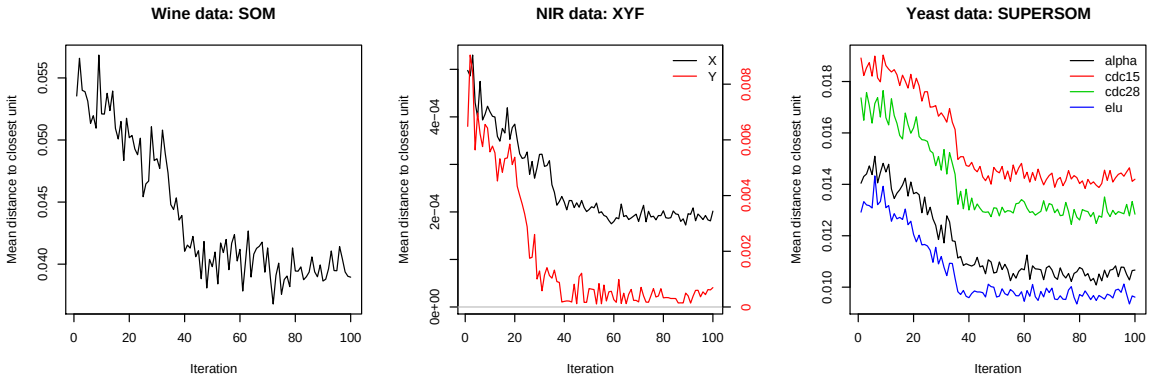

Figure 6: Training progress, as measured by the average distance of an object with the closest codebook vector unit. Left plot: unsupervised mapping of the wine data using `som`; middle plot: supervised mapping of the NIR data using `xyf`; and right plot: super-organized mapping of the yeast data using `supersom`.

may appear that more iterations are needed to obtain convergence. For every layer, one curve is shown: unsupervised `som` objects lead to one curve, supervised `xyf` and `bdk` objects yield two curves, and `supersom` objects can yield multiple curves. In the case of more than two curves, these are scaled individually so that the whole y-range of the plot is used; a separate axis is shown on the right. Three examples, based on the maps created above are shown in Figure 6. In all three cases, one can see the effect of the neighbourhood shrinking to include only the winning unit; this is the case after one-third of the iterations. After that stage, the training is merely fine-tuning. The plots are obtained with the following code:

```
R> graphics.off()
R> par(mfrow = c(1, 3))
R> plot(wine.som, type = "changes", main = "Wine data: SOM")
R> plot(nir.xyf, type = "changes", main = "NIR data: XYF")
R> plot(yeast.supersom, type = "changes", main = "Yeast data: SUPERSOM")
```

Many other plots can be made using these basic functions. Especially the `property` plots are very versatile. Indeed, the `counts` and `quality` plotting types internally calculate the number of mapped objects and the mean similarity per unit, and use these as property argument in a `property` plot.

6. Extra topics

6.1. Mapping

Once a map is trained, we can use it to project new data using the `map.kohonen` function. This function calculates (perhaps weighted) distances of the new data points to the codebook vectors and assigns the data to the closest unit. It is used internally in the training phase when

the data are stored in the map object: the `unit.classif` list element mentioned in relation to Figure 3. The function returns a list. The most important elements are the `unit.classif` and `distances` elements; these contain the indices of the winning units, and the distances to these winning units, respectively.

The function provides an extra layer of flexibility: even when `som` and `xyf` maps cannot be trained when missing values are present, they may be present when mapping new data. In the case of more than one layer (`xyf` and `supersom`) one can even tinker with the weights and the maps involved; by default, these are equal to the training parameters, but one can explicitly provide other values for the mapping phase. This makes it possible, e.g., to assess which of the layers has the biggest influence on the position of objects, or to try out "what if" scenarios quickly, without having to repeat the training phase.

6.2. Prediction

The `predict.kohonen` function takes a trained map and a set of new objects, maps them to the corresponding winning units, and returns the dependent variable associated with these units. For supervised maps (`xyf` and `bdk`), this information is already stored in the map. For SOMs and superSOMs, the user should provide data that can be used for calculating the predictions per unit. If the training data are stored in the map, the user only needs to provide the corresponding data for the dependent variable (using the `trainY` argument); if no training data are stored, the user should provide both `trainX` and `trainY` data. Examples are shown below for the yeast data. The first mapping uses only the `alpha` synchronisation element to predict cell cycle phase class.

```
R> set.seed(7)
R> training.indices <- sample(800, 400)
R> training <- rep(FALSE, 800)
R> training[training.indices] <- TRUE
R> yeast.ssom1 <- supersom(lapply(yeast, function(x) subset(x, training)),
+                          somgrid(4, 6, "hexagonal"),
+                          whatmap = 3)
Warning message:
removing 6 NA objects from the training data
 in: ...
R> ssom1.predict <- predict(yeast.ssom1,
+                           newdata = lapply(yeast,
+                             function(x) subset(x, !training)),
+                           trainY = subset(classvec2classmat(yeast[[7]]),
+                             training))
R> prediction1 <- factor(classmat2classvec(ssom1.predict$prediction),
+                         levels=levels(yeast$class))
R> confus1 <- table(subset(yeast$class, !training), prediction1)
R> confus1
      prediction1
       M/G1  G1   S  G2   M
  M/G1   22  25   0  10   6
  G1     10 127   1   6   0
```

```
  S       0  10   5  11   1
  G2      1   4   6  41  10
  M      16   5   1  36  44
R> sum(diag(confus1))
239
```

In total, 239 of the 400 genes in the test set have been classified correctly on the basis of their `alpha`-synchronized expression profile. If we include more information, we would expect this number to go up. Therefore, we train a second map using three other synchronisation methods, `cdc15`, `cdc28` and `elu`, as well.

```
R> yeast.ssom2 <- supersom(lapply(yeast, function(x) subset(x, training)),
+                          somgrid(4, 6, "hexagonal"),
+                          whatmap = 3:6)
Warning message:
removing 21 NA objects from the training data
 in: ...
R> ssom2.predict <- predict(yeast.ssom2,
+                           newdata = lapply(yeast,
+                             function(x) subset(x, !training)),
+                           trainY = subset(classvec2classmat(yeast[[7]]),
+                             training))
R> prediction2 <- factor(classmat2classvec(ssom2.predict$prediction),
+                        levels=levels(yeast$class))
R> confus2 <- table(subset(yeast$class, !training), prediction2)
R> confus2
      prediction2
       M/G1  G1   S  G2   M
  M/G1   24  22   0   0  17
  G1      7 130   5   3   0
  S       0   4  14  10   0
  G2      0   3   7  38  14
  M       1   1   0   9  91
R> sum(diag(confus2))
297
```

Adding the extra three layers in the second mapping leads to 58 more correctly classified genes, an improvement of fifteen percent points in error rate. Especially the predictions for the classes M and S have improved.

6.3. Relations between different methods

The functions `som` and `xyf` can be seen as special versions of `supersom`, applied with a list containing one and two data matrices, respectively. The `som` function, in particular, is mainly retained to keep compatibility with previous versions of the package; it has the disadvantage that it can not handle missing values, whereas `supersom` can. The `xyf` function, on the other hand, has extra functionality compared to `supersom`: if the Y variable is a class matrix (i.e., represents a categorical rather than continuous variable), the Tanimoto distance is used

instead of the Euclidean distance. In our experience, this gives slightly better classification results. Again, the `xyf` function has the disadvantage that it does not handle missing values. Both `xyf` and `som` are, because of their greater simplicitly, slightly faster than `supersom` although this will hardly be relevant in practical applications.

7. To do

The package can (and hopefully will) be extended in several directions. First of all, different similarity measures should be implemented for all mapping types, such as the Tanimoto distance, correlation, or even specialized measures. An example is the weighted cross-correlation (WCC), useful in cases where spectral features show random shifts. The latter measure, in combination with SOMs, has been implemented in a separate R package called **wccsom** (Wehrens *et al.* 2005), and has proved useful in mapping X-ray powder diffractograms. For `supersom`, it should eventually be possible to assign a useful distance measure to every layer in a map; thus, the need for a separate `xyf` function would disappear as well.

When data sets and the corresponding maps are large, or the similarity measure takes a lot of computation time (as in the WCC case), training can be a time-consuming process. In such cases, it is worthwhile to start training with a small map that increases in size, rather than a large map where the size of the neighbourhood is decreased. After adding the extra units, the codebook vectors for the new units are initialized through interpolation, and a new round of training begins. An example has been implemented in function `expand.som` from the `wccsom` package (Wehrens and Willighagen 2006). Eventually, the `wccsom` package should be merged into the `kohonen` package.

One major snag in using SOMs and analogues lies in the number of parameters influencing the mapping: learning rate, map size and topology, similarity functions, etcetera. Although the conclusions drawn from the mapping in many cases are fairly robust for different settings, one would like to have as few parameters as possible. One parameter that can be dispensed with by using another algorithm is the learning rate, by using the already mentioned batch algorithm. This, in principle, could be extended to the other SOM variants as well. Not only is there one fewer parameter to be set, it is usually quicker as well.

Several ways to enhance the graphical capabilities of the package can be explored as well. One example that would be potentially useful is to have an arbitarary center unit in the case of toroidal maps. Interpretation of these maps can be much easier when the focus of attention can be placed in the middle of the map, so that apparent but in fact non-existent edges can be ignored. Other plans include plots showing the smoothness of the map – i.e. the similarity between neighbouring units.

Acknowledgments

The **kohonen** package started as an extension of the SOM-related functions in the **class** package by Bill Venables and Brian D. Ripley, and also depends on the **MASS** package by the same authors. Willem Melssen (Radboud University Nijmegen) is acknowledged for invaluable discussions. The referees are acknowledged for suggestions and remarks.

References

Asuncion A, Newman D (2007). "UCI Machine Learning Repository." URL http://www.ics.uci.edu/~mlearn/MLRepository.html.

Cox TF, Cox MAA (2001). *Multidimensional Scaling*. Chapman & Hall/CRC, Boca Raton, Florida.

Helsinki University of Technology – CIS Laboratory (2006). "CIS Software Packages." URL http://www.cis.hut.fi/research/software/.

Helsinki University of Technology – CIS Laboratory (2007). "Bibliography of SOM Papers." URL http://www.cis.hut.fi/research/som-bibl/.

Jackson JE (1991). *A User's Guide to Principal Components*. Wiley, New York.

Kohonen T (2001). *Self-Organizing Maps*. Number 30 in Springer Series in Information Sciences. Springer-Verlag, Berlin, 3 edition.

Melssen WJ, Wehrens R, Buydens LMC (2006). "Supervised Kohonen Networks for Classification Problems." *Chemometrics and Intelligent Laboratory Systems*, **83**, 99–113.

R Development Core Team (2007). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Ripley BD (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge.

Spellman PT, Sherlock G, Zhang MQ, Iyer VR, Anders K, Eisen MB, Brown PO, Botstein D, Futcher B (1998). "Comprehensive Identification of Cell Cycle-regulated Genes of the Yeast *Saccharomyces Cerevisiae* by Microarray Hybridization." *Molecular Biology of the Cell*, **9**, 3273–3297.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S*. Springer-Verlag, New York, fourth edition. ISBN 0-387-95457-0, URL http://www.stats.ox.ac.uk/pub/MASS4/.

Wehrens R, Melssen WJ, Buydens LMC, de Gelder R (2005). "Representing Structural Databases in a Self-organizing Map." *Acta Crystallographica*, **B61**, 548–557.

Wehrens R, Willighagen E (2006). "Mapping Databases of X-ray Powder Patterns." *R News*, **6**(3), 24–28. URL http://CRAN.R-project.org/doc/Rnews/.

Wülfert F, Kok WT, Smilde AK (1998). "Influence of Temperature on Vibration Spectra and Consequences for Multivariate Models." *Analytical Chemistry*, **70**, 1761–1767.

Yan J (2004). **som**: *Self-Organizing Map*. R package version 0.3-4, URL http://CRAN.R-project.org/.

Zupan J, Gasteiger J (1999). *Neural Networks in Chemistry and Drug Design*. Wiley, New York, second edition.

Affiliation:

Ron Wehrens, Lutgarde Buydens
Institut for Molecules and Materials
Analytical Chemistry
Radboud University Nijmegen
Toernooiveld 1
6525 ED Nijmegen, The Netherlands
E-mail: r.wehrens@science.ru.nl, l.buydens@science.ru.nl
URL: http://www.cac.science.ru.nl/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 21, Issue 5 *Submitted:* 2007-05-04
October 2007 *Accepted:* 2007-09-10