Self- and Super-organising Maps in R: the Kohonen Package

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

September 7, 2007

Abstract

In this age of ever-increasing data set sizes, especially in the natural sciences, visualisation becomes more and more important. Self-organising 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-organising maps as well as some extensions for supervised pattern recognition and data fusion. It is available from CRAN.

keywords: self-organising maps, visualisation, classification, clustering

1 Introduction

In many areas in science, the past decades have seen a shift towards high-dimensional data. Exploratory analysis, where visualization plays a very important role, has become more and more difficult, and 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, of which Principal Component Analyis (PCA) [1] is probably the most-used. However, even PCA in many cases would need more than two dimensions, and in its pure form does not incorporate information on how objects should be compared. Methods starting from distance or similarity matrices, in this respect, may prove more useful. First of all, they are not bothered by large numbers of

variables, and second, by choosing the appropriate distance function, one can concentrate on those aspects of the data that are most informative.

One approach to visualize a distance matrix in two dimensions is Multi-Dimensional Scaling (MDS) and its many variants [2]. 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-Organising Maps (SOMs) [3] aim at something similar as MDS, but instead of trying to reproduce distances they aim at reproducing topology, or in other words, 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 (see [3] for some examples). This paper presents the `kohonen` package for R, initially based on the `class` library by B.D. Ripley. It features SOMs and two extensions that make it possible to use SOMs for classification and regression purposes, and for data mining. The package features extensive graphics to provide what is for us the most basic function of SOMs, namely visualization and information packaging. The next section gives some background on SOMs; next, the `kohonen` package is described in some detail. We conclude with 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 [4]. The number of clusters is defined by the size of the grid, that typically is arranged in a rectangular or hexagonal fashion. Indeed, one of the training strategies for SOMs (the "batch" algorithm) 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 associated with that unit. Usually, one randomly assigns a subset of the data to the map 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 clusters that are close to be similar. This is achieved by not only updating the winning unit, but also the units in the immediate neighbourhood of the winning unit. 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 [3].

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 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 obtained. Note that in this approach – sometimes called a Couter-Propagation Network – the dependent variable does not influence the mapping.

Another strategy, already suggested in [3], 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. In [5], we 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. The scaling takes care of possible differences in units between X and Y. Training is performed 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 it as super-organised maps. 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. The only extra parameters that need to be defined by the user (compared to classical SOMs) are the weights for the individual maps.

Table 1: Functions and data sets in the `kohonen` package, version 2.0.0

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

3 The `kohonen` package for R

The R package `kohonen` aims to provide simple-to-use functions for self-organising maps and the above-mentioned extensions, with specific emphasis on visualisation. The basic functions are `som`, for the usual form of self-organising maps; `xyf`, for supervised self-organising maps, or X-Y fused maps, useful when additional information in the form of, e.g., a class variable is available for all objects – and an alternative formulation called bi-directional Kohonen maps (function `bdk`); 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 visualising 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. A summary of the functions and data sets available in the package is given in Table 1.

Several data sets are included in the `kohonen` package: the wine data from the UCI Machine Learning repository (`http://kdd.ics.uci.edu`), near-infrared spectra from ternary mixtures of ethanol, water and isopropanol, measured at different temperatures described in [6], and finally a set of microarray data, the well-known yeast data from [7]. 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 colour 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.

Below, we will elaborate on the different stages in an exploratory analysis using self-organising 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.

4 Creating the maps

The different types of self-organising 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-organising maps: function `som`

The standard form of self-organising 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.

```
> library(kohonen)
Loading required package: class
> data(wines)
> wines.sc <- scale(wines[, -14])
> set.seed(7)
> wine.som <- som(data = wines.sc, grid = somgrid(5, 4, "hexagonal"))
> plot(wine.som, main="Wine data")     # default plot: codebook vectors
```

Finally, the result is shown in Figure 1: the codebook vectors are visualised in a segments plot (the default plotting type). High alcohol levels, for example, are associated with wine samples projected in the bottom right corner of the map, while colour intensity is largest in the bottom left corner. More plotting possibilities will be discussed below.

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;

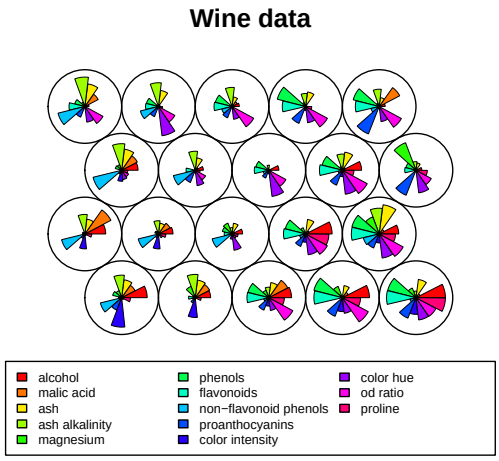

Figure 1: A plot of the codebook vectors of the 5-by-4 mapping of the wine data.

`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.

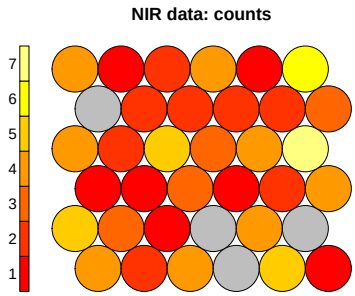

Figure 2: Counts plot of the map obtained from the NIR data using `xyf`. Empty units are depicted in gray.

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 (the same as Y), also the default value in `xyf`.

```
> data(nir); attach(nir)
> set.seed(13)
> nir.xyf <- xyf(data = spectra,
+                Y = composition[,2],
+                xweight = 0.5,
+                grid = somgrid(6, 6, "hexagonal"))
> plot(nir.xyf, type = "counts", main = "NIR data: water content")
```

This leads to the output shown in Figure 2. 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.

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 [5] 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`.

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:

```
> set.seed(13)
> 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 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):

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

> par(mfrow = c(1,2))
> plot(nir.xyf, type = "property", property = water.predict,
+      main="Prediction of water content", keepMargins = TRUE)
```

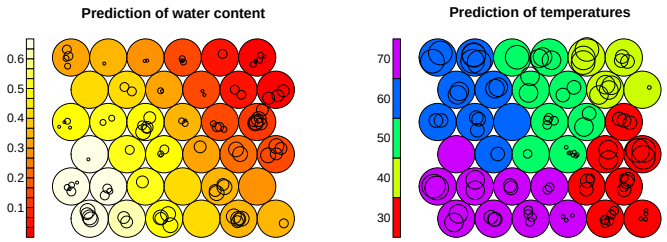

Figure 3: Supervised SOMs for the NIR data. The left plot shows the mapping of spectra when the XYF map has been trained with the water content as the y-variable (colors of the units are associated with the color key); the radius of the circles indicate the temperature at which spectra have been measured.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

```
> scatter <- matrix(rnorm(length(temperature)*2, sd=.1), ncol=2)
> radii <- (temperature - 20)/250
> symbols(nir.xyfnet$grid$pts[nir.xyfnet$unit.classif,] + scatter,
+         circles = radii, inches = FALSE, add = TRUE)

> plot(nir.xyf2, type = "property", property = temp.predict,
+      palette.name = rainbow, main="Prediction of temperatures")
> scatter <- matrix(rnorm(nrow(composition)*2, sd=.1), ncol=2)
> radii <- 0.05 + 0.4 * composition[,2]
> 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 give show the unit predictions in a background colour; 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.

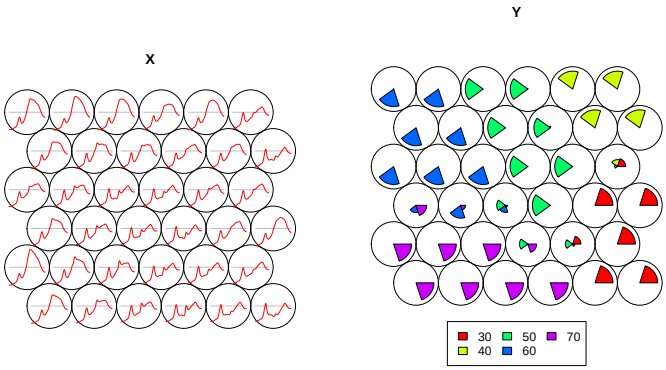

Figure 4: Plots of codebook vectors for the `xyf` mapping of IR spectra (X) and temperature (Y).

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:

```
> par(mfrow=c(1,2))
> 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-Organised 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-organised 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.

We show an example, based on the well-known yeast cell-cycle data by [7]. 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:

```
> data(yeast)
> set.seed(7)
> 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)
> classes <- levels(yeast$class)
> colors <- c("yellow", "green", "blue", "red", "orange")
> par(mfrow=c(3,2))
> plot(yeast.supersom, type = "mapping",
+      pch = 1, main = "All", keepMargins = TRUE)
> for (i in seq(along=classes)) {
+   X.class <- lapply(yeast,
+                     function(x) subset(x, yeast$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)
+ }
```

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 [7]. 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.

Note that graphical parameters, notably the margin settings, particular for this figure, are still in effect. When making new plots, it is advisable

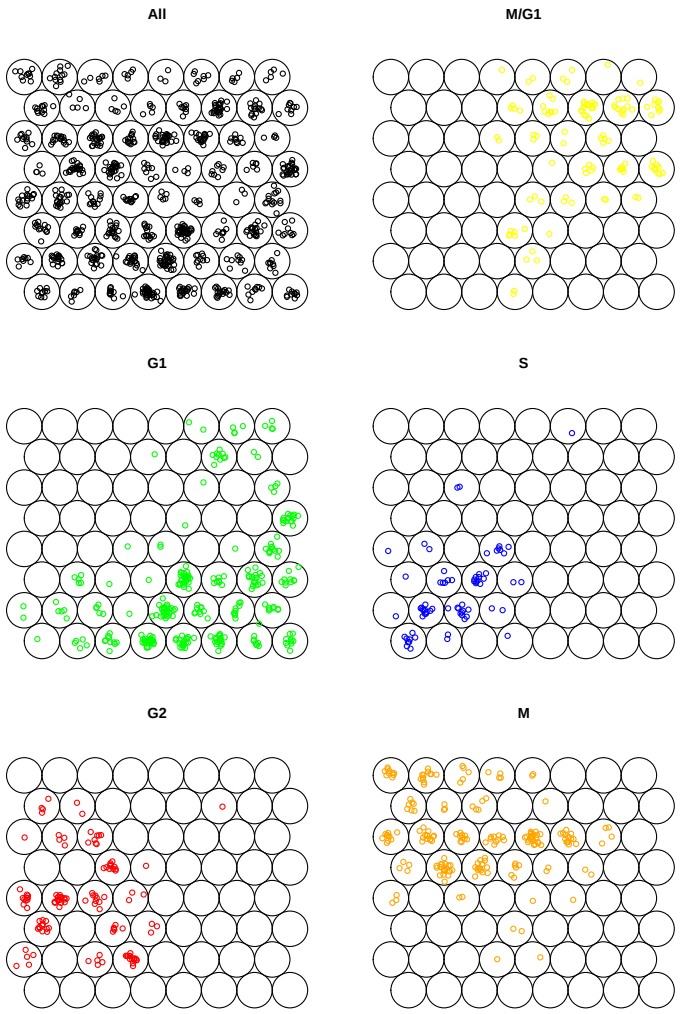

Figure 5: Mapping of the 800 genes in a eight-by-eight superSOM, based on the `alpha`, `cdc15`, `cdc28` and `elu` data (with equal weights for each of the four layers). The top left plot shows the position of all 800 genes; the other plots show the same mapping split up over the five cell-cycle stages.

to do this in another plot window, or to close this plot first by issuing

```
> graphics.off()
```

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` 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 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 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:

```
> par(mfrow=c(1,3))
> plot(wine.som, type = "changes", main="Wine data: SOM")
> plot(nir.xyfnet, type = "changes", main = "NIR data: XYF")
> 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. One can think, e.g., of a "quality" plot showing the average similarity of objects mapped to individual units, for example.[1]

[1] In a previous version of the package, this indeed was a separate function.

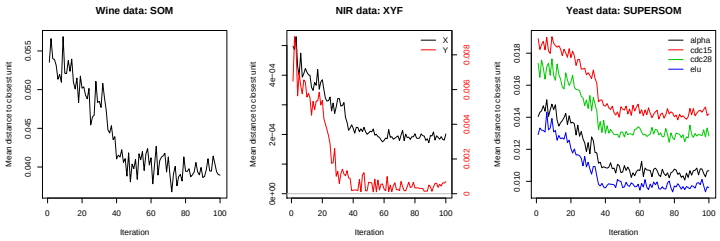

Figure 6: Training progress, as measured by the average distance of an object with the closest codebook vector unit. Left plot: unsupervised mapping of the wine data using `som`; middle plot: supervised mapping of the NIR data using `xyf`; and right plot: super-organised mapping of the yeast data using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

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.

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

```
> 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: ...
> ssom2.predict <- predict(yeast.ssom2,
+                          newdata = lapply(yeast,
+                            function(x) subset(x, !training)),
+                          trainY = subset(classvec2classmat(yeast[[7]]),
+                            training))
> prediction2 <- factor(classmat2classvec(ssom2.predict$prediction),
+                       levels=levels(yeast$class))
> confus2 <- table(subset(yeast$class, !training), prediction2)
> 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
> 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 specialised 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` [8], 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

[9]. 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: the algorithm involved is known as the batch version. In the `class` package, a batch version of SOMs has been implemented that could be extended for the other SOM variants as well. Not only does it require one fewer parameter, it is usually quicker, too.

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.

Acknowledgements

The `kohonen` package started as an extension of the som-related functions in the `class` package by B.D. Ripley, and also depends on the `MASS` package by the same author.

Willem Melssen (Radboud University Nijmegen) is acknowledged for invaluable discussions; his Matlab implementation of several of the tools discussed in this paper is available from `http://www.cac.science.ru.nl/software`.

References

[1] J.E. Jackson. *A user's guide to principal components*. Wiley, New York, 1991.

[2] T.F. Cox and M.A.A. Cox. *Multidimensional Scaling*. Chapman and Hall, 2001.

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

[4] B.D. Ripley. *Pattern recognition and neural networks*. Cambridge University Press, 1996.

[5] W.J. Melssen, R. Wehrens, and L.M.C. Buydens. Supervised Kohonen networks for classification problems. *Chemom. Intell. Lab. Syst.*, 83:99–113, 2006.

[6] F. Wülfert, W.Th. Kok, and A.K. Smilde. Influence of temperature on vibration spectra and consequences for multivariate models. *Anal. Chem.*, 70:1761–1767, 1998.

[7] P.T. Spellman, G. Sherlock, M.Q. Zhang, V.R. Iyer, K. Anders, M.B. Eisen, P.O. Brown, D. Botstein, and B. Futcher. Comprehensive identification of cell cycle-regulated genes of the yeast *saccharomyces cerevisiae* by microarray hybridization. *Mol. Biol. Cell.*, 9:3273–3297, 1998.

[8] R. Wehrens, W.J. Melssen, L.M.C. Buydens, and R. de Gelder. Representing structural databases in a self-organizing map. *Acta Cryst.*, B61:548–557, 2005.

[9] Ron Wehrens and Egon Willighagen. Mapping databases of x-ray powder patterns. *R News*, 6(3):24–28, August 2006.