Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Flexible Self-Organising Maps in kohonen v3.0

Ron Wehrens
Wageningen University & Research

Johannes Kruisselbrink
Wageningen University & Research

Abstract

Self-organising maps (SOMs) are popular tools for grouping and visualizing data in many areas of science. This paper describes recent changes in **kohonen**, implementing several different forms of SOMs. These changes are primarily focused on making the package more useable for large data sets. Memory consumption has decreased dramatically, a.o., by replacing the old interface to the underlying compiled code by a new one relying on **Rcpp**. The batch SOM algorithm for training has been added in both sequential and parallel forms. A final important extension of the package's repertoire is the possibility to define and use data-dependent distance functions, extremely useful in cases where standard distances like the Euclidean distance are not appropriate. Several examples of possible applications are presented.

Keywords: self-organising maps, distance functions, parallellization, R.

1. Introduction

The **kohonen** package for R was published ten years ago (Wehrens and Buydens 2007) as a tool for self-organising maps (Kohonen 1995), providing simple yet effective visualization methods. A particular characteristic of the package was the possibility to provide several data layers that would contribute to the overall distances on which the mapping would be based. Since then, the package has been applied in different fields like environmental sciences (Vaclavik, Lautenbach, Kuemmerle, and Seppelt 2013), transcriptomics (Chitwood, Maloof, and Sinha 2013), and biomedical sciences (Wiwie, Baumbach, and Rottger 2015), to mention but a few.

Self-organising maps (SOMs) aim to project objects onto a two-dimensional plane in such a way that similar objects are close together and dissimilar objects are far away from each other. Obviously, such a visualization is extremely useful for data sets with large numbers of objects, but also in cases where the number of variables is large – everything is brought back to similarities between objects. What distinguishes SOMs from approaches like Principal Component Analysis (PCA) or Principal Coordinate Analysis (PCoA) is the discrete nature of the mapping: objects are mapped to specific positions in the 2D plane, called units. Each unit is associated with an object called a codebook vector, usually corresponding to the average of all objects mapped to that unit. The codebook vector can be seen as a "typical" object for that area of the map. Mapping data to a trained SOM is nothing more than calculating the distance of the new data points to the codebook vectors, and assigning each object to the unit with the most similar codebook vector (the best matching, or "winning", unit).

For training SOMs several different algorithms are used: the standard ones are the online and the batch algorithms (Kohonen 1995). In both cases, training objects are compared to the current set of codebook vectors. The codebook vector of the winning unit, as well as units within a certain radius of the winning unit, are changed to become more similar to the new object mapping to it. During training, the radius slowly decreases; in the end of the training only the winning unit is updated. The difference between the online and batch SOM algorithms is that the update of the winning unit(s) in the online algorithm is done after each individual object, whereas the batch algorithm does not change the codebook vectors until the whole data set has been presented. Then, the cumulative change based on all objects in the training set is applied in one go.

Several R packages implement more or less elaborate versions of the SOM: **class** (Venables and Ripley 2002), the basis of the original **kohonen** package, implements both the online and batch algorithms, using the `.C` interface to compiled code. The **som** package (Yan 2016) provides the basic online version of the SOM, with training done in C++ using the `.Call` interface. Package **SOMbrero** (Olteanu and Villa-Vialaneix 2015) provides methods for handling numerical data, contingency tables, and dissimilarity matrices. Different scaling methods for data of particular types are included. All functions are pure R – no compiled code is used, which leads to a relatively slow training phase. The package supports a **shiny** (Chang, Cheng, Allaire, Xie, and McPherson 2017) interface for dynamic visualization of results. The **RSNNS** package (Bergmeir and Benítez 2012) is an `R`-wrapper to the `SNNS` toolbox written in `C++`, including SOMs as one of many types of neural networks. Finally, a very recent publication introduced the **somoclu** package (Wittek, Gao, Lim, and Zhao 2017), providing a general toolbox for training SOMs with support for cluster and GPU computations, and interfaces to Python, MATLAB and R.

The **kohonen** package (Wehrens and Buydens 2007) provides extensive visualization possibilities[1] as well as fast training and mapping using compiled code. However, several limitations were becoming apparent (see, *e.g.*, Boyle, Araya *et al.* (2014)). Memory management was not very efficient, unnecessarily declaring potentially large memory chunks; the lack of parallellization support meant that even in simple desktop and laptop computers computing power was not used; and finally, fixed distance functions were implemented and no others could be added by the user without explicitly recompiling the package. In a major overhaul, these issues have been addressed, leading to version 3.0. Finally, the package has been extended with the batch algorithm, including support for parallel computing.

In the next section, we will discuss these and other changes in more detail. This is followed by three application examples, highlighting the capacity of the package to handle large data sets and the use of data-specific distance functions. Benchmarks are given, comparing the efficiency of the new version with the previous version of the package. The paper ends with an outlook to future developments and some conclusions.

[1] For visualization the **somoclu** package provides an interface to the **kohonen** package.

2. Changes in the kohonen package, version 3.0

Since the publication of the **kohonen** package in 2007 (Wehrens and Buydens 2007) a number of small improvements had been made prior to version 3.0, mostly affecting the plotting functions, and often in response to user requests or bug reports. At some point it became apparent, however, that in order to stay relevant the package needed a more fundamental overhaul, making it faster and, especially, more memory-efficient, in order to be able to tackle larger data sets. The changes in version 3.0 of the **kohonen** package are discussed below. They correspond to using a different interface for calling the underlying compiled code, the possibility to define and use problem-specific distance functions on the fly, the inclusion of the batch algorithm for training SOM maps, and smaller miscellaneous improvements.

2.1. Switching from `.C` to `Rcpp`

The most important change in the package is not visible to the user and consists of the replacement of the `.C` interface, inherited from the **class** package, by a more efficient `Rcpp` solution. The advantages of using the `Rcpp` interface to the underlying compiled code rather than the original `.C` interface are clear and well documented (Eddelbuettel and Francois 2011). Here, it is particularly important that `Rcpp` works directly on the R objects, passed as function arguments, instead of making local copies of these objects. This is much more memory-efficient and especially for large data sets can make a huge difference in performance.

In addition, several functions were redesigned and rewritten at the compiled-code level. The `som` and `xyf` functions now are wrappers for the `supersom` function, applicable in situations with one and two data layers, respectively. In versions before 3.0, each of these functions had its own C implementation, which made the code hard to maintain. As a beneficial side effect, `som` and `xyf` can now be used when there are NAs in the data, something that in earlier versions was only possible with the `supersom` function. If the argument `maxNA.fraction` is set to zero (the default) no checks for NAs are performed, which leads to a further speed increase. Function `bdk` has been deprecated, since it offered no added benefits over `xyf` and was roughly twice as slow. The main functions of the package, `supersom`, `predict.kohonen` and `map.kohonen` function have been rewritten completely. Especially the latter has become much more memory-efficient: for data sets with many records mapping could take as long as the training phase. Although we have tried to keep as close as possible to the original code, there is no complete backwards compatibility – when applying the same settings, the new **kohonen** version will give results that are slightly different from the old version.

2.2. Flexible definition of distance measures

In the previous version of the **kohonen** package basically two distance measures were available: the Euclidean distance for numeric variables, and Tanimoto distance for factors. The latter could only be used in `xyf` networks. Version 3.0 not only provides more predefined distance measures (the Euclidean distance, the Sum-of-squares distance, the Manhattan distance, and the Tanimoto distance), they now can be used in all SOM types, and can be defined for each layer separately[2]. By default, the Tanimoto distance is used for factors or class membership

[2] The sum-of-squares distance for single-layered maps is equivalent to the Euclidean distance, but faster to compute since no square root is taken; for maps with multiple layers the Euclidean and Sum-of-square distances lead to different overall distances.

matrices, and the Sum-of-squares distance in all other cases. Note that also the so-called U-matrix visualization, showing the average distance of each SOM unit to its neighbours and available through the `type = "dist.neighbours"` argument of the `plot.kohonen` function now is taking into account the specific distances used in training the map, as well as their corresponding weights. The same holds for methods used for grouping codebook vectors – up to now clustering (e.g., in the examples of the `plot.kohonen` function) was performed on Euclidean distances in one layer only, and did not take into account either different distance metrics nor weights. A new function has been included, codeobject.distances, to allow the user to calculate distances between data points (and also between codebook vectors), according to the weighted combination of distances employed in the SOM.

Specific distance functions appropriate for the data at hand can be defined by the user in C++, again for every data layer separately[3]. In ecologocial applications, for example, several specific different distance functions are routinely used, such as the Bray-Curtis and Jaccard dissimilarities (Goslee and Urban 2007). A user-provided function implementing such a distance should take two vectors of real numbers, corresponding to the data and the codebook vectors, the number of variables, and the number of NA values, and should return one number. An example will be given below.

2.3. The batch SOM algorithm

The batch SOM algorithm has been available for a long time in the **class** package. Amazingly, the central loop of the algorithm contains only four lines of pure R code. The main difference with the original SOM algorithm is that the map update is performed less frequently: only after the complete data set has been presented are the codebook vectors updated – in that sense, the algorithm is very similar to the k-means algorithm. The learning rate α therefore is no longer needed.

Already on the wishlist in Wehrens and Buydens (2007), this algorithm has been now included in the **kohonen** package. It is written in C++ for maximum performance, and is available through the `mode = "batch"` argument of the `som` functions. Whereas it is hard to define a useful parallel version of the online algorithm, the batch SOM algorithm does lend itself for parallellization: the task of finding the best matching units for all records in the data set is split up over different cores (Lawrence, Almasi, and Rushmeier 1999). This data partitioning parallel version is available using `mode = "pbatch"`. By default, all cores are used. This can be changed by the user through the `cores` argument of the `supersom` function.

2.4. Miscellaneous changes

Several smaller changes have been made to the package as well. These are outlined in brief below.

Changes in plotting functions. Several extensions of the plotting functions have been implemented: on the request of several users now hexagon- and square-shaped units can be used in the plot functions. This is particularly useful for larger SOMs, where the whitespace between circular units would be a distraction. If SOM units are grouped in clusters, the cluster

[3] User-defined distance functions in R, even though possible in principle, would lead to much lower speeds and are therefore not supported.

boundaries between units can be visualized. The validity of the clusters can be assessed by showing the average distance to neighbouring codebook vectors as unit background color, available as `type = "dist.neighbours"` in the `plot.kohonen` function. Examples of these new features can be seen in the manual pages of `plot.kohonen`.

Weighing different data layers. The default system of weighing different data layers has been changed. Since distances are crucially dependent on the units in which the data have been recorded, a set of internal `distance.weights` is calculated first, corresponding to the inverses of the median distances in the individual data layer. The application of these weights ensures an (approximately, at least) equal contribution of all layers to the final distance measure – the user-defined weights are then used to influence this further. A user who wants to bypass this behaviour is able to do so by setting `normalizeDataLayers` to `FALSE`.

Changes in SOM grid specification. An additional argument, `neighbourhood.fct` has been added to allow for a gaussian neighbourhood, that by definition always includes all units in the map in the update step. The `"bubble"` neighbourhood is still retained as the default, since it is much faster. Related to this, the default for the `radius` parameter has changed: for both the `"bubble"` and `"gaussian"` neighbourhoods the default is to decrease linearly to a value of zero, starting from a radius that includes two-thirds of the distances in the map. Note that in the `"bubble"` case only winning units will be updated when the radius is smaller than 1. Finally, the `toroidal` argument has been moved to a more appropriate location, the `somgrid` function (which is now no longer imported from the **class** package).

Growing larger SOMs. In some cases it may be cost-effective to train a small SOM first, to get the global structure of the data, and to extend the map with additional units in a later stage. Function `expandMap`, doing exactly this, has been introduced in this version of **kohonen** (taken from the soon to-be-phased out package **wccsom**, see below). This is one of two ways of obtaining a SOM with many units, potentially more than there are records (known as an emergent SOM); the other way (already available since the first version of the package) is to explicitly provide the initialization state through the `init` argument.

3. Applications

Some of the new features in **kohonen** will now be highlighted. First, we use an application in image segmentation, an example of SOMs in a big data context, to present some relevant benchmarks. Then, we show a couple of examples of the use of user-defined distance functions. Code to reproduce the examples in this paper is included as demos in the **kohonen** package:

```
R> demo("JSSdemo1", package = "kohonen")
```

This will run the first example from the paper on image segmentation – the other two demos are available as `JSSdemo2` and `JSSdemo3`, respectively. Note that especially the first example will take some time to complete (on an average PC from 2017 not more than a minute, though).

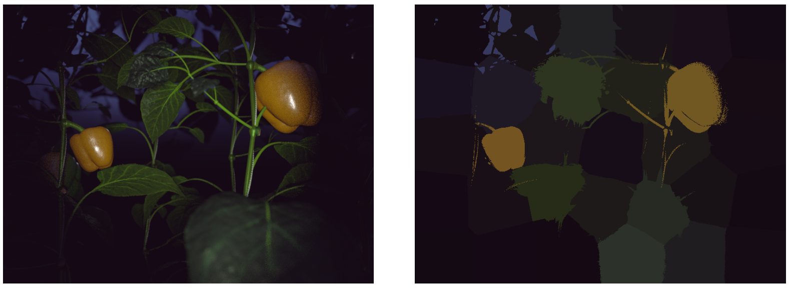

Figure 1: Original image of pepper plants (left) and the segmented image using 32 colors (right).

3.1. Image segmentation

SOMs are particularly useful for grouping large numbers of objects. An area where this is relevant is pixel-based image segmentation: even a relatively small image already can contain thousands of pixels, and databases of thousands and thousands of images are becoming increasingly common. The aim then is to obtain an accurate description of the original images using a limited set of "colors", where colors may be either true colors (RGB values, as in this paper), or artificial colors obtained from spectroscopic measurements (Franceschi and Wehrens 2014). In many applications, specific colors will be associated with objects in the image. Here, we use one image from a set of artificial images from pepper plants, where the objective is to separate the peppers from the plants and the background (Barth, IJsselmuiden, Hemming, and van Henten 2017)[4].

In principle, segmentation of an image using self-organising maps is easy enough – one can present the color values of the individual pixels to the SOM and observe where they will end up in the map. The image that we will use as an example, a synthetic (but very realistic) image of a pepper plant is shown in the left plot of Figure 1. Image segmentation can help in getting a simple and quick idea of the number and the size of the peppers on a large number of plants, a process known as high-throughput phenotyping.

The segmented image is shown in the right panel of Figure 1. It consists of a total of 32 different colors, each corresponding to one SOM unit. The cartoon-like representation still shows the most important features. In this case we have performed using two layers of information, the RGB values of the individual pixels in the first SOM layer, and the pixel positions in the image in the second. The idea is that pixels mapping to the same unit in the SOM will not only have the same color, but will also be close together in the image. The tessellation-like pattern in the segmented image that is the result from this is clearly visible. To achieve this, we give quite a high weight to the pixel coordinate layer:

```
R> mygrid <- somgrid(4, 8, "hexagonal")
R> set.seed(15)
R> somsegm1 <- supersom(imdata, whatmap = c("rgb", "coords"),
+                         user.weights = c(1, 9), grid = mygrid)
```

[4] The complete data, containing 10,500 images of 600x800 pixels, is available from http://dx.doi.org/10.4121/uuid:884958f5-b868-46e1-b3d8-a0b5d91b02c0.

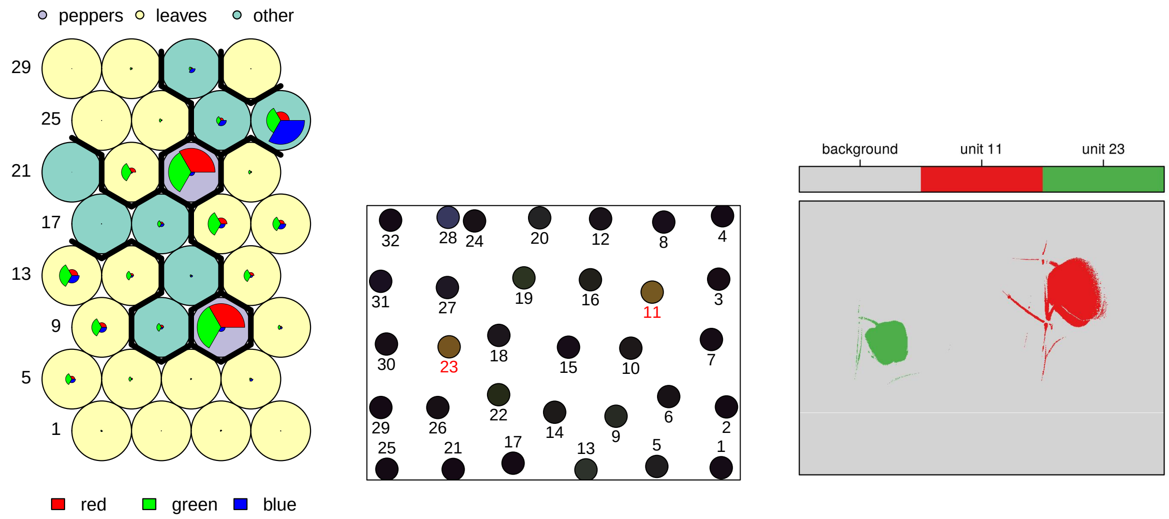

Figure 2: Left plot: codebook vectors for the RGB layer of the SOM; background colors indicate the dominant class of pixels mapping to the individual units. Middle plot: codebook vectors for the coordinate layer of the SOM, plotted in image coordinates. Fill colors correspond to the RGB colors of the corresponding units. Right plot: segmentation image, highlighting only the pixels from the two pepper units (purple background in the plot on the left).

The RGB codebook vectors are shown in the left plot of Figure 2. The background colors are associated with the dominant pixel classes mapping to the individual units (where only the two most important classes are shown explicitly and the other six are merged as "other" – note that since this is a synthetic image, the true pixel classes are known). One can see that the brightest colors (the largest R, G and B segments) correspond to peppers (the pastel purple background color), as one might expect. Note that there are two separate units where pepper pixels tend to be projected, because of the second information layer, the X-Y coordinates of the pixels (middle panel in Figure 2). Each circle depicts the position of the unit in the original image; the fill color corresponds to the RGB color of the units. The numbers correspond to the SOM units; these are covering the image in a regular fashion. The numbers of the pepper units are indicated in red. Unit 23 corresponds to the left pepper in the image, and unit 11 to the right pepper. Finally, the right plot in Figure 2 shows the location of the pixels mapping to the the pepper units in the original image. Although these units contain some non-pepper pixels the overall result is quite satisfying.

It can be a definite advantage that the two peppers in the image are projected in different areas of the map, illustrating that image segmentation on the pixel level can sometimes profit from additional information – in this case pixel coordinates. Other types of information that can be useful include information of neighbouring pixels, and depth information. While these could conceptually also be included as extra columns in the original data, leading to one SOM layer only, keeping this information in additional layers provides more flexibility: one can use different weights for different layers (as was done here), or even use different distance functions, appropriate for particular layers.

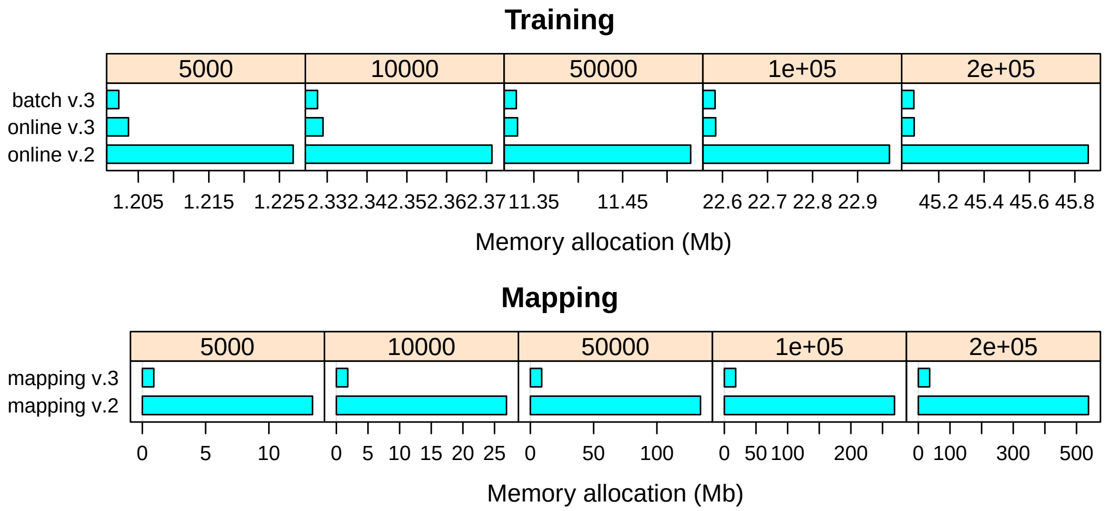

Figure 3: Memory allocation for training SOMs and mapping data to a trained SOM for different numbers of pixels.

One of the main reasons for redesigning the **kohonen** package was the difficulty of analysing larger data sets, in particular those with many records. Especially memory usage was not optimized. The improvement in the new package version is illustrated here by some simple benchmarks. These benchmarking experiments consist of image segmentation of parts of the pepper plant image, RGB information only (5.5 Mb), on hexagonal grids of eight by four units, the same as used in the example. Five different pixel subsets were used, ranging in size from 5,000 pixels to 200,000 pixels. The assessment of memory usage[5] was performed using the `profmem` function from the package with the same name (Bengtsson 2016), keeping track of memory allocation during execution of the training or mapping phases. Speed benchmarking was performed using the **microbenchmark** package (Mersmann 2015), and was done on a HPC cluster node with an Intel(R) Xeon(R) CPU E5-2660 processor having eight physical cores and two logical cores per physical core. One hundred repetitions are used for speed benchmarking (the default `microbenchmark`).

Memory allocation during training with several forms of the `supersom` function as well as during mapping the data to the trained SOM is shown in Figure 3. The old version of the package is always shown as the bottom line in the plots. Training in the new version is more efficient than in the old version, with small differences between batch and online algorithms. Parallel batch algorithms show the same memory allocation as the batch algorithm, independent of the number of nodes used in the parallel version. In the mapping phase, where data are projected to a fully trained SOM, a huge improvement in memory usage is visible. Memory profiling results are obtained on two computers, one running Linux and one running Windows, and are virtually identical.

The improvement in speed is shown in Figure 4, showing the time needed for training the SOM, and for mapping data, respectively. Training is considerably faster in the new version, here showing an almost two-fold speed improvement. In the new version, differences between

[5] Note that memory profiling of R functions is not easy, a.o., because of the effects of garbage collection – these numbers should be taken as ballpark figures.

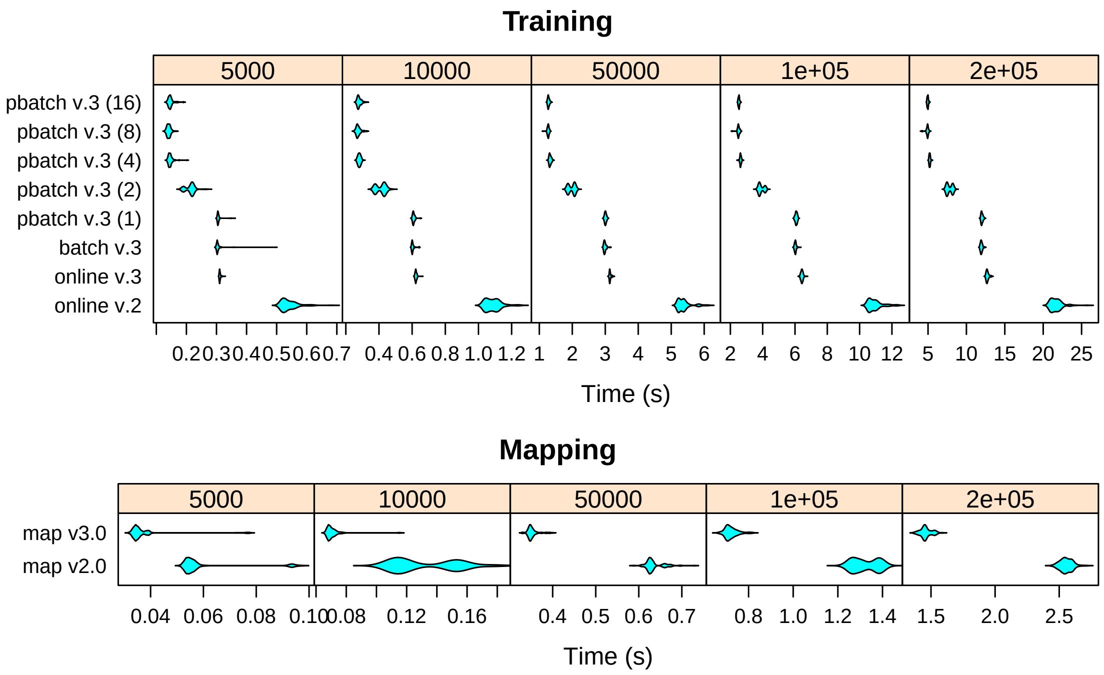

Figure 4: Comparison of running times of different SOM implementations, again for different numbers of pixels. The number of logical cores employed with the training using the parallel batch algorithm is given in brackets.

the online and batch training are small; also the parallel batch algorithm using one core only gives comparable timings (as expected). Using the parallel batch algorithm using more than one core provides further considerable speed improvements. Also mapping is much faster in the new version. Finally, the new version presents the opportunity to avoid checks for NA values by providing `maxNA.fraction = 0L` (obviously for data without NAs), which would lead to another speed-up.

3.2. User-defined distance functions

In some cases, the usual Euclidean or related distance functions are simply not optimal, and one would rather use specific measures appropriate for the data at hand. From version 3.0 onwards, such functions can be written and compiled in C++, and a pointer to the function can be provided to the SOM functions in **kohonen**. We give two examples, one from ecology, comparing sites according to the number of different plant species found in each site, and a crystallographic one comparing spectra where peaks are shifted in different samples.

Ecological applications. Package **vegan** (Oksanen, Blanchet, Friendly, Kindt, Legendre, McGlinn, Minchin, O'Hara, Simpson, Solymos, Stevens, Szoecs, and Wagner 2016) contains a pair of data sets containing information on 24 sites, one set concentrating on cover values of 44 plant species at the sites, the other on soil characteristics (Väre, Ohtonen, and Oksanen 1995). Even though the data set too small for a realistic application of SOMs, it does provide a good illustration of the possibilities of using several data layers associated with specific distance functions. First, the data are loaded and converted to matrices:

```
R> data("varespec", package = "vegan")
R> data("varechem", package = "vegan")
R> varechem <- as.matrix(varechem)
R> varespec <- as.matrix(varespec)
```

We train the SOM using the majority of the data, the training set, and assess its value using the rest, the test set, here consisting of the last five records:

```
R> n <- nrow(varechem)
R> tr.idx <- 1:(n-5)
R> tst.idx <- (n-4):n
R> chemmat.tr <- scale(varechem[tr.idx,])
R> chemmat.tst <- scale(varechem[tst.idx,],
+                       center = colMeans(varechem[tr.idx,]),
+                       scale = apply(varechem[tr.idx,], 2, sd))
R> specmat.tr <- varespec[tr.idx,]
R> specmat.tst <- varespec[tst.idx,]
```

The aim is to group sites in such a way that sites in the same SOM unit have both a similar species profile and similar soil characteristics. For comparing species counts over different sites, often a Bray-Curtis dissimilarity is used, defined by

$$d_{jk} = \sum_i |x_{ij} - x_{ik}| / \sum_i (x_{ij} + x_{ik})$$

This is implemented in `C++` as follows:

```
R> BCcode <- '
+    #include <Rcpp.h>
+    typedef double (*DistanceFunctionPtr)(double *, double *, int, int);
+
+    double brayCurtisDissim(double *data, double *codes, int n, int nNA) {
+      if (nNA > 0) return NA_REAL;
+
+      double num = 0.0, denom = 0.0;
+      for (int i = 0; i < n; i++) {
+          num += std::abs(data[i] - codes[i]);
+          denom += data[i] + codes[i];
+      }
+
+      return num/denom;
+    }
+
+    // [[Rcpp::export]]
+    Rcpp::XPtr<DistanceFunctionPtr> BrayCurtis() {
+      return Rcpp::XPtr<DistanceFunctionPtr>(new DistanceFunctionPtr(&brayCurtisDissim));
+    } '
```

In this piece of code, the Bray-Curtis dissimilarity is the function `brayCurtisDissim`. A valid dissimilarity function in this context has four arguments: the first two are the vectors that will be compared, the third is the length of the vectors (they should be equally long), and the final argument is the number of NA values in the first of the two vectors. It is assumed that the second vector (which corresponds to the codebook vectors) never contains NA values. Note that in this example we do not allow NAs at all: any input NA value will immediately lead to an NA value for the distance. The last two lines define the function `createBrayCurtisDistPtr` that creates a pointer to the distance function. The name of this pointer-generating function is to be passed as argument to the SOM training and mapping functions. The `C++` code can be compiled using the **Rcpp** package:

```
R> library("Rcpp")
R> sourceCpp(code = BCcode)
```

Training the map now is easy: we simply provide the names of the appropriate distance functions corresponding to the data matrices:

```
R> set.seed(101)
R> varesom <- supersom(list(species = specmat.tr, soil = chemmat.tr),
+                       somgrid(4, 3, "hexagonal"),
+                       dist.fcts = c("BrayCurtis", "sumofsquares"))
```

Note that saving such a map will save the name of the distance functions employed. Loading it again in a new session will work as expected provided that the corresponding functions are available in the workspace, which can require recompiling the corresponding `C++` code.

The trained map now can be used for making predictions as well: if a new object is mapped to a particular unit (using the same distance functions as when training the map), then it can be assumed that it is very close to the other objects mapping to that unit. In this way, predictions can be made, e.g., for data layers that are absent. Without the `newdata` argument, the `predict.kohonen` function returns for each record the averages of the training objects mapping to the same winning unit. The `predict` function allows one to provide the `unit.predictions` values explicitly: an alternative would be to use codebook vectors rather than averages of objects. In most cases, this would lead to very similar predictions.

```
R> trainingPred <- predict(varesom)
R> names(trainingPred)
[1] "predictions"      "unit.classif"     "unit.predictions" "whatmap"
```

The result is a list, where the `unit.predictions` slot contains the average values for individual map units, the `unit.classif` slot contains the winning units for data records, and `predictions` is the combination of the two. Both `unit.predictions` and `predictions` are lists with, in this case, two elements: `soil` and `species`. We can visualize the expected values for soil parameters using a colour scale, as is shown in Figure 5. The code to generate the first panel is shown here:

```
R> plot(varesom, type = "property",
+       main = "Nitrogen (standardized)",
+       property = trainingPred$unit.predictions$soil[,"N"])
```

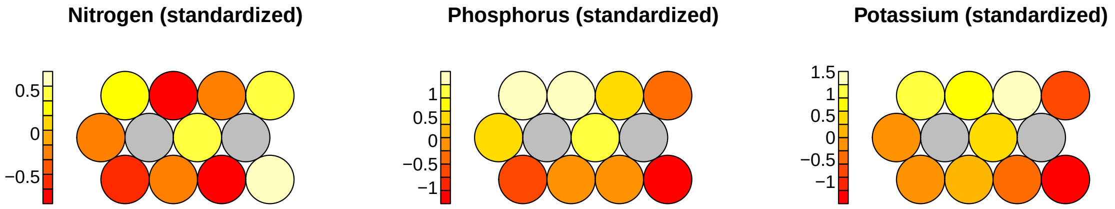

Figure 5: Expected values for the first three soil parameters (standardized units) based on mapping the training data to the SOM.

The other panels are obtained by substituting the appropriate column names (and plot titles). The figure shows two units in gray – these are examples of units that do have codebook vectors, but are never winning units when mapping the training data to the SOM. It is interesting to see that the four soil parameters do not vary equally smoothly over the map.

Predictions for test data can be made using the `newdata` argument. The new data should contain at least some data layers that are also present in the training data (the agreement is checked comparing the names and dimensions of the data layers; if no names are given, it is assumed that the data layers are presented in the same order). Layers may be selected or weighted by using the `whatmap` and `user.weights` arguments, providing a simple way to test different scenarios. If these arguments are not provided, the values used in training the map are taken, usually also the most meaningful option. As an example, predictions for the test set based on the mapping according to the soil data only are obtained as follows:

```
R> soilBasedPred <- predict(varesom,
+                          newdata = list(soil = chemmat.tst))
R> lapply(soilBasedPred$predictions, function(x) signif(head(x[,1:4]), 3))
$species
   Callvulg Empenigr Rhodtome Vaccmyrt
9      0.00     8.38     1.38   12.100
12     1.25     9.08     0.00    0.833
10     0.10     3.73     0.00    0.000
11     0.00     8.38     1.38   12.100
21     0.10     3.73     0.00    0.000

$soil
        N       P       K     Ca
9   0.255  1.4500  1.1900  1.080
12 -0.218 -0.0351  0.0209  0.506
10  0.721 -1.3400 -1.5000 -1.370
11  0.255  1.4500  1.1900  1.080
21  0.721 -1.3400 -1.5000 -1.370
```

Again we see the effect of a unit to which no training data are mapped, leading to `NA` values in the predictions. In some cases, it could be useful to impute these by interpolation from neighbouring units.

Classification of powder patterns. In Wehrens, Melssen, Buydens, and de Gelder (2005); Willighagen, Wehrens, Melssen, de Gelder, and Buydens (2007), a SOM package using a specialized distance function for comparing X-ray powder diffractograms was presented, called **wccsom**. These powder patterns provide information on the crystal cell structure (de Gelder, Wehrens, and Hageman 2001) which is important in 3D structure elucidation. Structures represented by similar unit cell parameters give rise to very similar patterns; however, patterns may be stretched or compressed, based on the unit cell size. In many cases, one is not interested in the unit cell size, and would like to group patterns on the basis of the peaks in the powder pattern, irrespective of the stretching of compression that has taken place. To do this, a distance (WCCd) based on the weighted crosscorrelation (WCC) was used in **wccsom**, given by

$$\text{WCCd} = 1 - \text{WCC} = 1 - \frac{\mathbf{f}^\top \mathbf{W} \mathbf{g}}{\sqrt{\mathbf{f}^\top \mathbf{W} \mathbf{f}} \sqrt{\mathbf{g}^\top \mathbf{W} \mathbf{g}}}$$

Here, $\mathbf{f}$ and $\mathbf{g}$ are vectors of observations (here along measurement angles, but in other applications this could be a time axis, for example). $\mathbf{W}$ is a banded matrix with ones on the diagonal and values progressively smaller further away from the diagonal. After a certain number of rows (or columns), only zeros are present; this number, the width of the band, is indicated with a parameter θ. If $\theta = 0$ (and $\mathbf{W}$ becomes a diagonal matrix) the second term on the right hand side in the equation reduces to the Pearson correlation coefficient. The WCC has also been used in other applications, such as aligning chromatographic profiles (Wehrens, Carvalho, and Fraser 2015b) using the **ptw** packge (Bloemberg, Gerretzen, Wouters, Gloerich, Wessels, van Dael, van den Heuvel, Eilers, Buydens, and Wehrens 2010; Wehrens, Bloemberg, and Eilers 2015a).

The purpose of this example is to show how to emulate the **wccsom** package with **kohonen** – the **wccsom** package itself will soon be phased out. More information on the background and the interpretation of the results can be found in the original **wccsom** publications (Wehrens *et al.* 2005; Willighagen *et al.* 2007).

The code for the WCCd dissimilarity is saved in a file called `wcc.cpp` (in the `inst/Distances` subdirectory of the package), structured in much the same way as previously the Bray-Curtis distance (which is available in the same directory). Parameter θ is hard-coded in this distance function (here it is set to a value of 20); if one wants to try different values of θ it is easy to define several distance functions. The data consist of 131 structures, a subset of a set of 205 used earlier in Wehrens *et al.* (2005); Willighagen *et al.* (2007).

```
R> data("degelder")
R> mydata <- list(patterns = degelder$patterns,
+                 CellVol = log(degelder$properties[,"cell.vol"]))
```

Figure 6 shows the powder patterns of four of the structures from two different space groups. Studying the variability in such a data set may show interesting groupings and may help in the structure elucidation of unknown compounds (Wehrens *et al.* 2005).

When mapping the patterns to a SOM, the X-ray powder patterns are used as the first layer, and the cell volume as the second layer. This will bring crystal structures together that have similar diffraction patterns as well as similar cell volumes. Since the cell volume has a very large range, logarithms are used. The distance function for the cell volume is the standard sum-of-squares distance, and the WCCd distance is defined in `wcc.cpp`:

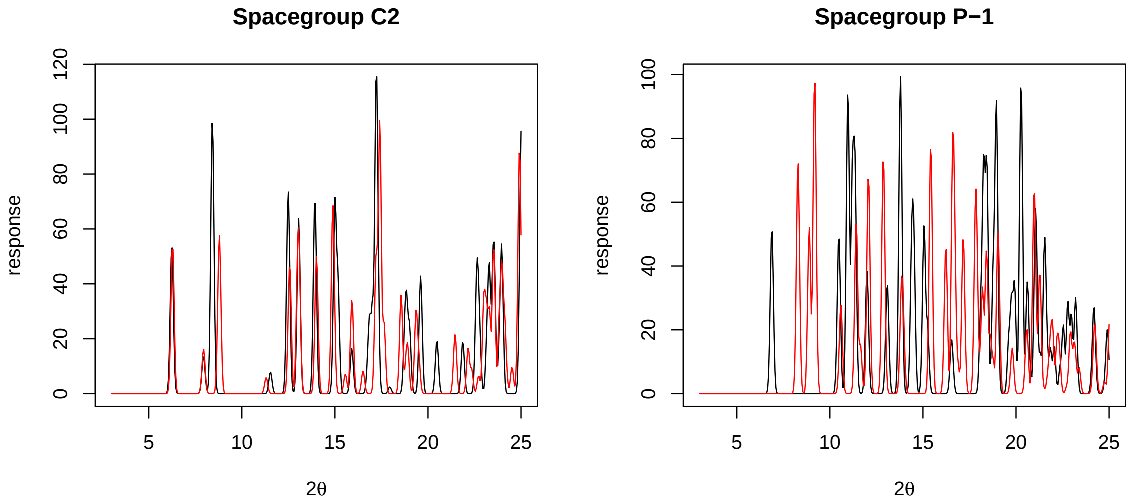

Figure 6: Examples of powder patterns from two different space groups: the left panel shows very similar patterns (a WCC distance of 0.08) whereas the right panel shows a more different couple (0.39).

```
R> sourceCpp(paste(path.package("kohonen"), "Distances/wcc.cpp", sep = "/"))
R> set.seed(7)
R> powsom <- supersom(data = mydata, grid = somgrid(6, 4, "hexagonal"),
+                        dist.fcts = c("WCCd", "sumofsquares"), keep.data = TRUE)
```

The codebook vectors of the trained map are shown in Figure 7. Clearly, the crystal structures with the largest cell volumes are located in the bottom right corner of the map. We could use such a map to estimate the cell volume for new diffraction patterns, simply by projecting the powder patterns into the map on the basis of the WCCd dissimilarity. Then, the cell volume associated with the winning unit will be used as the prediction:

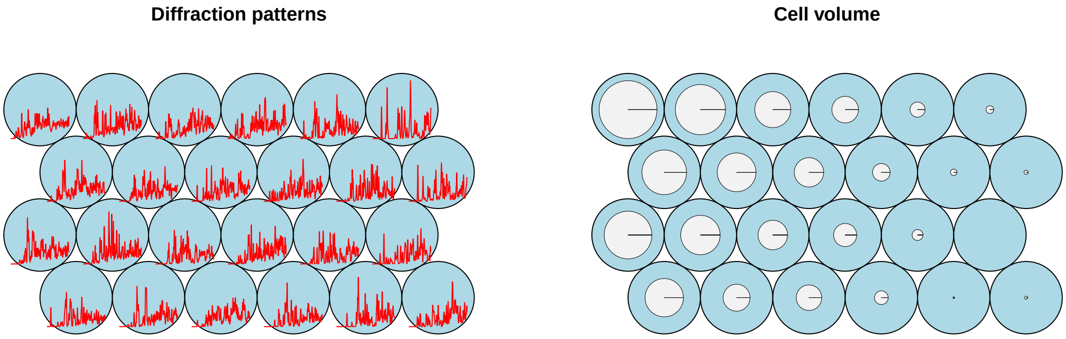

Figure 7: Codebook vectors for the two data layers in the powder diffraction map.

```
R> cellPreds <- predict(powsom, newdata = mydata, whatmap = "patterns")
R> names(cellPreds)
[1] "predictions"      "unit.classif"     "unit.predictions" "whatmap"
R> cellPreds$predictions$CellVol[1:5,]
  asageg   asuyes   axerie   axerok   azuyoj
8.091085 6.852079 8.093932 7.121513 7.290206
```

Again, the `predict` function makes it possible to combine data in a number of different ways. Here we present the complete data including all layers, and use the `whatmap` argument to determine which layer is to be used for mapping the objects to the map. In many cases this is easier than subsetting the data or creating specific new data objects, and allows for an easy investigation of different scenarios and the relation between the individual data layers.

4. Conclusions and outlook

In this Big Data era there is more and more need for methods that group large numbers of records, and fuse different data layers associated with these record. SOMs, and in particular the `supersom` variant implemented in the **kohonen** package, are very useful tools in this respect. They can be used to assess groupings in the data, but also to see structure within groups. The package enables users to investigate their data in a quick and easy way, and allows for what-if scenarios by switching data layers in or out, or by changing weights for individual layers.

This paper concentrates on several improvements of the package. In particular the possibility of including data-specific distance or dissimilarity functions is an important feature that should increase the applicability of SOM methodology in a number of fields. It should be noted that the update of the codebook vectors during training still is done through (weighted) averaging; conceivably, for some types of data and distance functions this is not the optimal route. In the same way as used-defined distance functions, user-defined update functions are a real possibility. Significant performance improvements have been obtained, also by the inclusion of the batch algorithm and, in particular, its parallel version.

As it is now, the package is a reasonably rounded and complete set of functions. One obvious extension is to make the plotting functions more interactive, so that it would be more easy to switch between different visualizations. Another major improvement in the context of big data would be if objects could be sampled from external databases without explicitly having to import them in the form of one self-contained datasets, or to allow streaming data to come in for permanent updates of the codebook vectors. The latter aspect would almost automatically also require aspects of the interactive graphics mentioned earlier. For these and other improvements, user contributions are warmly welcomed. We hope in the mean time that the package will continue to find use in many diverse fields of science.

Acknowledgements

The members of the `Rcpp` discussion list are acknowledged for quick and pertinent responses to our questions about the `Rcpp` interface. The authors thank Ruud Barth for the pepper plant images and for useful discussions.

References

Barth R, IJsselmuiden J, Hemming J, van Henten E (2017). "Data Synthesis Methods for Semantic Segmentation in Agriculture. A Capsicum annuum Dataset." *Submitted.*

Bengtsson H (2016). *profmem: Simple Memory Profiling for R.* R package version 0.4.0, URL `https://CRAN.R-project.org/package=profmem`.

Bergmeir C, Benítez J (2012). "Neural Networks in R Using the Stuttgart Neural Network Simulator: RSNNS." *Journal of Statistical Software, Articles*, **46**(7), 1–26. ISSN 1548-7660. `doi:10.18637/jss.v046.i07`. URL `https://www.jstatsoft.org/v046/i07`.

Bloemberg T, Gerretzen J, Wouters H, Gloerich J, Wessels H, van Dael M, van den Heuvel L, Eilers P, Buydens L, Wehrens R (2010). "Improved Parametric Time Warping for Proteomics." *Chemom. Intell. Lab. Syst.*, **104**, 65–74. `doi:10.1016/j.chemolab.2010.04.008`.

Boyle AP, Araya CL, *et al.* (2014). "Comparative Analysis of Regulatory Information and Circuits across Distant Species." *Nature*, **512**(7515), 453–456. `doi:10.1038/nature13668`.

Chang W, Cheng J, Allaire J, Xie Y, McPherson J (2017). *shiny: Web Application Framework for R.* R package version 1.0.2, URL `https://CRAN.R-project.org/package=shiny`.

Chitwood DH, Maloof JN, Sinha NR (2013). "Dynamic Transcriptomic Profiles between Tomato and a Wild Relative Reflect Distinct Developmental Architectures." *Plant Physiology*, **162**(2), 537–552. `doi:10.1104/pp.112.213546`.

de Gelder R, Wehrens R, Hageman J (2001). "A Generalized Expression for the Similarity of Spectra: Application to Powder Diffraction Pattern Classification." *J. Comput. Chem.*, **22**(3), 273–289. `doi:10.1002/1096-987X(200102)22:3<273::AID-JCC1001>3.0.CO;2-0`.

Eddelbuettel D, Francois R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software, Articles*, **40**(8), 1–18. ISSN 1548-7660. `doi:10.18637/jss.v040.i08`. URL `https://www.jstatsoft.org/v040/i08`.

Franceschi P, Wehrens R (2014). "Self-organising Maps: a Versatile Tool for the Automatic Analysis of Untargeted Metabolomic Imaging Datasets." *Proteomics*, **14**, 853–861. `doi:10.1002/pmic.201300308`.

Goslee S, Urban D (2007). "The ecodist Package for Dissimilarity-based Analysis of Ecological Data." *J. Stat. Softw.*, **22**(7), 1–19. `doi:10.18637/jss.v022.i07`.

Kohonen T (1995). *Self-Organizing Maps.* Springer-Verlag, Berlin.

Lawrence R, Almasi G, Rushmeier H (1999). "A Scalable Parallel Algorithm for Self-Organizing Maps with Applications to Sparse Data Mining Problems." *Data Mining and Knowledge Discovery*, **3**(2), 171–195. ISSN 1573-756X. `doi:10.1023/A:1009817804059`. URL `http://dx.doi.org/10.1023/A:1009817804059`.

Mersmann O (2015). *microbenchmark: Accurate Timing Functions.* R package version 1.4-2.1, URL `https://CRAN.R-project.org/package=microbenchmark`.

Oksanen J, Blanchet FG, Friendly M, Kindt R, Legendre P, McGlinn D, Minchin PR, O'Hara RB, Simpson GL, Solymos P, Stevens MHH, Szoecs E, Wagner H (2016). *vegan: Community Ecology Package.* R package version 2.4-1, URL `https://CRAN.R-project.org/package=vegan`.

Olteanu M, Villa-Vialaneix N (2015). "On-line Relational and Multiple Relational SOM." *Neurocomputing*, **147**, 15–30. `doi:10.1016/j.neucom.2013.11.047`.

Vaclavik T, Lautenbach S, Kuemmerle T, Seppelt R (2013). "Mapping Global Land System Archetypes." *Global Environmental Change*, **23**(6), 1637–1647. `doi:10.1016/j.gloenvcha.2013.09.004`.

Väre H, Ohtonen R, Oksanen J (1995). "Effects of reindeer grazing on understorey vegetation in dry *Pinus sylvestris* forests." *J. Veg. Sci.*, **6**, 523–530. `doi:10.2307/3236351`.

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

Wehrens R, Bloemberg T, Eilers P (2015a). "Fast Parametric Time Warping of Peak Lists." *Bioinformatics*, **31**, 3063–3065. `doi:10.1093/bioinformatics/btv299`.

Wehrens R, Buydens L (2007). "Self- and Super-organizing Maps in R: The kohonen Package." *Journal of Statistical Software, Articles*, **21**(5), 1–19. ISSN 1548-7660. `doi:10.18637/jss.v021.i05`. URL `https://www.jstatsoft.org/v021/i05`.

Wehrens R, Carvalho E, Fraser P (2015b). "Metabolite Profiling in LC-DAD Using Multivariate Curve Resolution: the alsace Package for R." *Metabolomics*, **11**, 143–154. `doi:10.1007/s11306-014-0683-5`.

Wehrens R, Melssen W, Buydens L, de Gelder R (2005). "Representing Structural Databases in a Self-organizing Map." *Acta Cryst.*, **B61**, 548–557. `doi:10.1107/S0108768105020331`.

Willighagen E, Wehrens R, Melssen W, de Gelder R, Buydens L (2007). "Supervised Self-organising Maps in Crystal Structure Prediction." *Cryst. Growth & Des.*, **7**, 1738–1745. `doi:10.1021/cg060872y`.

Wittek P, Gao S, Lim I, Zhao L (2017). "somoclu: An Efficient Parallel Library for Self-Organizing Maps." *Journal of Statistical Software, Articles*, **78**(9), 1–21. ISSN 1548-7660. `doi:10.18637/jss.v078.i09`. URL `https://www.jstatsoft.org/v078/i09`.

Wiwie C, Baumbach J, Rottger R (2015). "Comparing the Performance of Biomedical Clustering Methods." *Nature Methods*, **12**(11), 1033–1038. `doi:10.1038/nmeth.3583`.

Yan J (2016). *som: Self-Organizing Map.* R package version 0.3-5.1, URL `https://CRAN.R-project.org/package=som`.

Affiliation:

Ron Wehrens, Johannes Kruisselbrink
Biometris
Wageningen University & Research
Wageningen, The Netherlands
E-mail: ron.wehrens@wur.nl, johannes.kruisselbrink@wur.nl
URL: http://www.wur.nl/biometris/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd