Software Manual
Institute of Bioinformatics, Johannes Kepler University Linz

APCluster

An R Package for Affinity Propagation Clustering

Ulrich Bodenhofer, Johannes Palme, Chrats Melkonian, and Andreas Kothmeier

Institute of Bioinformatics, Johannes Kepler University Linz
Altenberger Str. 69, 4040 Linz, Austria
apcluster@bioinf.jku.at

Version 1.4.10, May 31, 2022

Institute of Bioinformatics
Johannes Kepler University Linz
A-4040 Linz, Austria

Tel. +43 732 2468 4520
Fax +43 732 2468 4539
http://www.bioinf.jku.at

Scope and Purpose of this Document

This document is a user manual for the R package `apcluster` [2]. It is only meant as a gentle introduction into how to use the basic functions implemented in this package. Not all features of the R package are described in full detail. Such details can be obtained from the documentation enclosed in the R package. Further note the following: (1) this is neither an introduction to affinity propagation nor to clustering in general; (2) this is not an introduction to R. If you lack the background for understanding this manual, you first have to read introductory literature on these subjects.

Contents

1 Introduction

Affinity propagation (AP) is a relatively new clustering algorithm that has been introduced by Brendan J. Frey and Delbert Dueck [6].[1] The authors themselves describe affinity propagation as follows:

> "*An algorithm that identifies exemplars among data points and forms clusters of data points around these exemplars. It operates by simultaneously considering all data point as potential exemplars and exchanging messages between data points until a good set of exemplars and clusters emerges.*"

AP has been applied in various fields recently, among which bioinformatics is becoming increasingly important. Frey and Dueck have made their algorithm available as Matlab code.[1] Matlab, however, is relatively uncommon in bioinformatics. Instead, the statistical computing platform R has become a widely accepted standard in this field. In order to leverage affinity propagation for bioinformatics applications, we have implemented affinity propagation as an R package. Note, however, that the given package is in no way restricted to bioinformatics applications. It is as generally applicable as Frey's and Dueck's original Matlab code.[1]

Starting with Version 1.1.0, the `apcluster` package also features *exemplar-based agglomerative clustering* which can be used as a clustering method on its own or for creating a hierarchy of clusters that have been computed previously by affinity propagation. *Leveraged Affinity Propagation*, a variant of AP especially geared to applications involving large data sets, has first been included in Version 1.3.0.

2 Installation

2.1 Installation via CRAN

The R package `apcluster` (current version: 1.4.10) is part of the *Comprehensive R Archive Network (CRAN)*[2]. The simplest way to install the package, therefore, is to enter the following command into your R session:

```
install.packages("apcluster")
```

If you use R on Windows or Mac OS, you can also conveniently use the package installation menu of your R GUI.

2.2 Manual installation from source

Under special circumstances, e.g. if you want to compile the C++ code included in the package with some custom options, you may prefer to install the package manually from source. To this end, open the package's page at CRAN[3] and then proceed as follows:

[1] `https://psi.toronto.edu/research/affinity-propagation-clustering-by-message-passing/`
[2] `http://cran.r-project.org/`
[3] `https://CRAN.R-project.org/package=apcluster`

1. Download `apcluster_1.4.10.tar.gz` and save it to your harddisk.
2. Open a shell/terminal/command prompt window and change to the directory where you put `apcluster_1.4.10.tar.gz`. Enter

   ```
   R CMD INSTALL apcluster_1.4.10.tar.gz
   ```

   to install the package.

Note that this might require additional software on some platforms. Windows requires Rtools[4] to be installed and to be available in the default search path (environment variable `PATH`). Mac OS X requires Xcode developer tools[5] (make sure that you have the command line tools installed with Xcode).

2.3 Compatibility issues

All versions downloadable from CRAN have been built using the latest version, R 4.1.1. However, the package should work without severe problems on R versions $\geq$3.0.0.

3 Getting Started

To load the package, enter the following in your R session:

```
library(apcluster)
```

If this command terminates without any error message or warning, you can be sure that the package has been installed successfully. If so, the package is ready for use now and you can start clustering your data with affinity propagation.

The package includes both a user manual (this document) and a reference manual (help pages for each function). To view the user manual, enter

```
vignette("apcluster")
```

Help pages can be viewed using the `help` command. It is recommended to start with

```
help(apcluster)
```

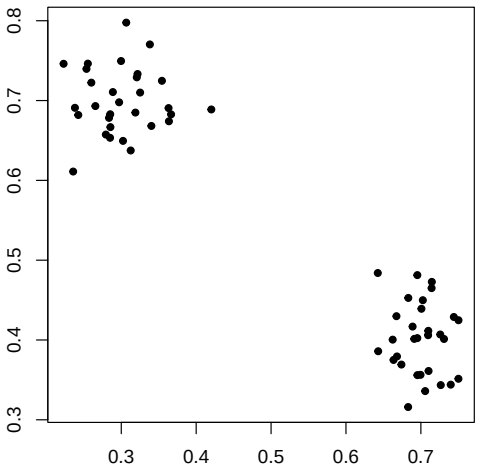
Affinity propagation does not require the data samples to be of any specific kind or structure. AP only requires a *similarity matrix*, i.e., given l data samples, this is an $l \times l$ real-valued matrix $\mathbf{S}$, in which an entry S_{ij} corresponds to a value measuring how similar sample i is to sample j. AP does not require these values to be in a specific range. Values can be positive or negative. AP does not even require the similarity matrix to be symmetric (although, in most applications, it will

[4] `http://cran.r-project.org/bin/windows/Rtools/`
[5] `https://developer.apple.com/technologies/tools/`

be symmetric anyway). A value of $-\infty$ is interpreted as "absolute dissimilarity". The higher a value, the more similar two samples are considered.

To get a first impression, let us create a random data set in $\mathbb{R}^2$ as the union of two "Gaussian clouds":

```
cl1 <- cbind(rnorm(30, 0.3, 0.05), rnorm(30, 0.7, 0.04))
cl2 <- cbind(rnorm(30, 0.7, 0.04), rnorm(30, 0.4, .05))
x1 <- rbind(cl1, cl2)
plot(x1, xlab="", ylab="", pch=19, cex=0.8)
```



The package `apcluster` offers several different ways for clustering data. The simplest way is the following:

```
apres1a <- apcluster(negDistMat(r=2), x1)
```

In this example, the function `apcluster()` first computes a similarity matrix for the input data `x1` using the *similarity function* passed as first argument. The choice `negDistMat(r=2)` is the standard similarity measure used in the papers of Frey and Dueck — negative squared distances.

Alternatively, one can compute the similarity matrix beforehand and call `apcluster()` for the similarity matrix (for a more detailed description of the differences, see 10.1):

```
s1 <- negDistMat(x1, r=2)
apres1b <- apcluster(s1)
```

The function `apcluster()` creates an object belonging to the S4 class `APResult` which is defined by the present package. To get detailed information on which data are stored in such objects, enter

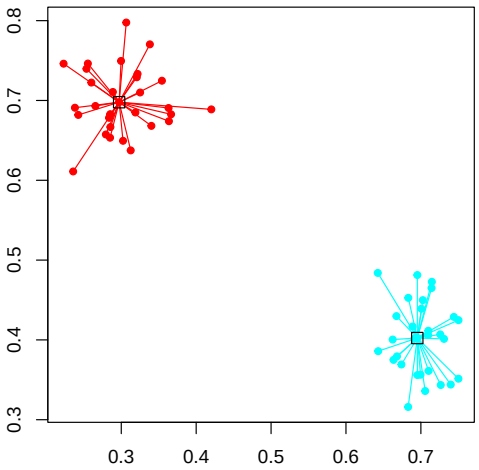
```
help(APResult)
```

The simplest thing we can do is to enter the name of the object (which implicitly calls `show()`) to get a summary of the clustering result:

```
apres1a

##
## APResult object
##
## Number of samples     =  60
## Number of iterations  =  131
## Input preference      =  -0.1416022
## Sum of similarities   =  -0.1955119
## Sum of preferences    =  -0.2832044
## Net similarity        =  -0.4787163
## Number of clusters    =  2
##
## Exemplars:
##    25 44
## Clusters:
##    Cluster 1, exemplar 25:
##       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
##       26 27 28 29 30
##    Cluster 2, exemplar 44:
##       31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
##       53 54 55 56 57 58 59 60
```

The `apcluster` package allows for plotting the original data set along with a clustering result:

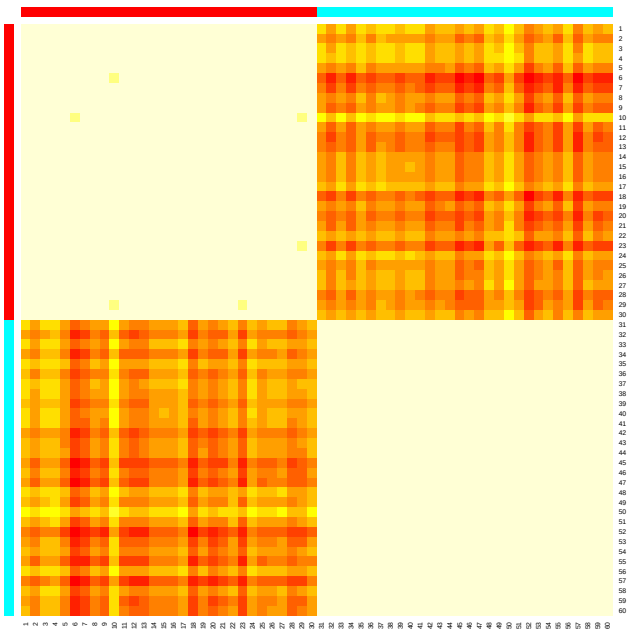
```
plot(apres1a, x1)
```



In this plot, each color corresponds to one cluster. The exemplar of each cluster is marked by a box and all cluster members are connected to their exemplars with lines.

A heatmap is plotted with `heatmap()`:

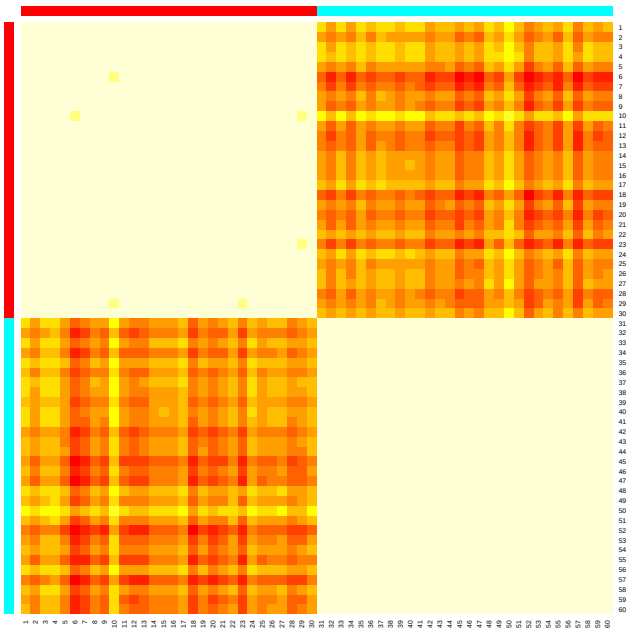
```
heatmap(apres1a)
```



In the heatmap, the samples are grouped according to clusters. The above heatmap confirms again that there are two main clusters in the data. A heatmap can be plotted for the object `apres1a` be-

cause `apcluster()`, if called for data and a similarity function, by default includes the similarity matrix in the output object (unless it was called with the switch `includeSim=FALSE`). If the similarity matrix is not included (which is the default if `apcluster()` has been called on a similarity matrix directly), `heatmap()` must be called with the similarity matrix as second argument:

```
heatmap(apres1b, s1)
```

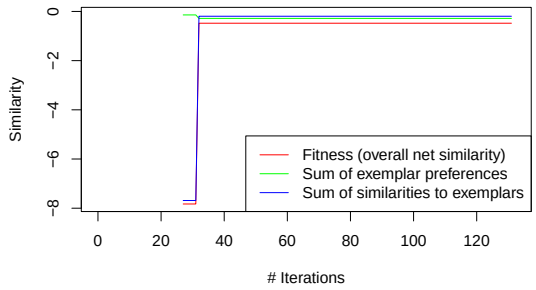


Suppose we want to have better insight into what the algorithm did in each iteration. For this purpose, we can supply the option `details=TRUE` to `apcluster()`:

```
apres1c <- apcluster(s1, details=TRUE)
```

This option tells the algorithm to keep a detailed log about its progress. For example, this allows for plotting the three performance measures that AP uses internally for each iteration:

```
plot(apres1c)
```



These performance measures are:

1. Sum of exemplar preferences
2. Sum of similarities of exemplars to their cluster members
3. Net fitness: sum of the two former

For details, the user is referred to the original affinity propagation paper [6] and the supplementary material published on the affinity propagation Web page.[1] We see from the above plot that the algorithm has not made any change for the last 100 iterations. AP, through its parameter `convits`, allows to control for how long AP waits for a change until it terminates (the default is `convits=100`). If the user has the feeling that AP will probably converge quicker on his/her data set, a lower value can be used:

```
apres1c <- apcluster(s1, convits=15, details=TRUE)
apres1c

##
## APResult object
##
## Number of samples     =  60
## Number of iterations  =  46
## Input preference      =  -0.1416022
## Sum of similarities   =  -0.1955119
## Sum of preferences    =  -0.2832044
## Net similarity        =  -0.4787163
## Number of clusters    =  2
##
## Exemplars:
##    25 44
## Clusters:
##    Cluster 1, exemplar 25:
##       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
##       26 27 28 29 30
##    Cluster 2, exemplar 44:
##       31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
##       53 54 55 56 57 58 59 60
```

4 Adjusting Input Preferences

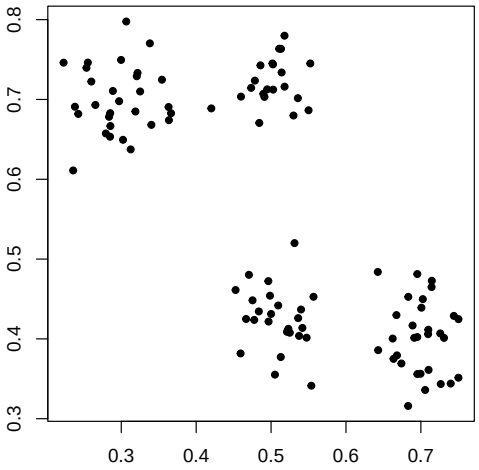
Apart from the similarity matrix itself, the most important input parameter of AP is the so-called *input preference* which can be interpreted as the tendency of a data sample to become an exemplar (see [6] and supplementary material on the AP homepage[1] for a more detailed explanation). This input preference can either be chosen individually for each data sample or it can be a single value

shared among all data samples. Input preferences largely determine the number of clusters, in other words, how fine- or coarse-grained the clustering result will be.

The input preferences one can specify for AP are roughly in the same range as the similarity values, but they do not have a straightforward interpretation. Frey and Dueck have introduced the following rule of thumb: "*The shared value could be the median of the input similarities (resulting in a moderate number of clusters) or their minimum (resulting in a small number of clusters).*" [6]

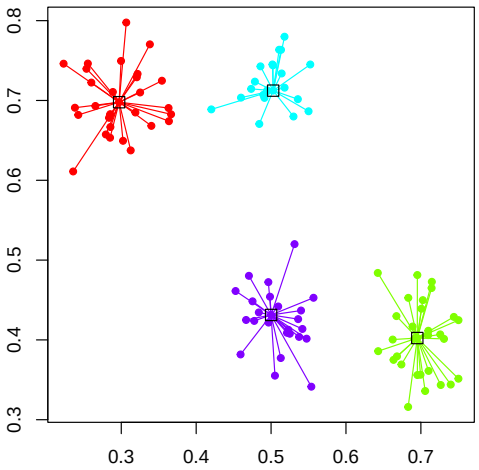
Our AP implementation uses the median rule by default if the user does not supply a custom value for the input preferences. In order to provide the user with a knob that is — at least to some extent — interpretable, the function `apcluster()` provides an argument `q` that allows to set the input preference to a certain quantile of the input similarities: resulting in the median for `q=0.5` and in the minimum for `q=0`. As an example, let us add two more "clouds" to the data set from above:

```
cl3 <- cbind(rnorm(20, 0.50, 0.03), rnorm(20, 0.72, 0.03))
cl4 <- cbind(rnorm(25, 0.50, 0.03), rnorm(25, 0.42, 0.04))
x2 <- rbind(x1, cl3, cl4)
plot(x2, xlab="", ylab="", pch=19, cex=0.8)
```



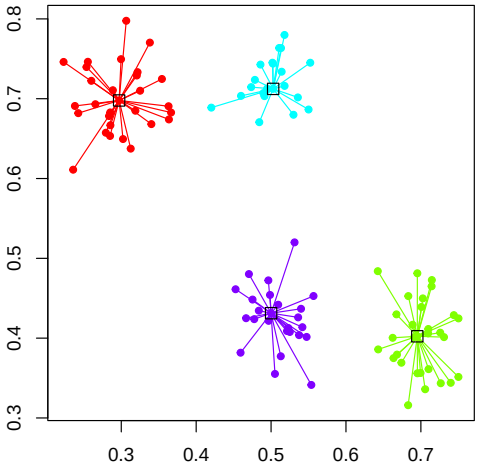
For the default setting, we obtain the following result:

```
apres2a <- apcluster(negDistMat(r=2), x2)
plot(apres2a, x2)
```



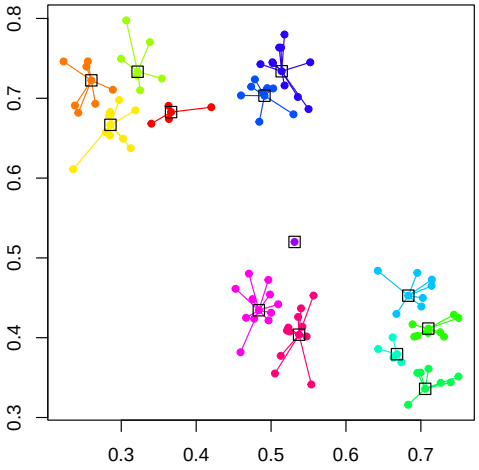
For the minimum of input similarities, we obtain the following result:

```
apres2b <- apcluster(negDistMat(r=2), x2, q=0)
plot(apres2b, x2)
```



So we see that AP is quite robust against a reduction of input preferences in this example which may be caused by the clear separation of the four clusters. If we increase input preferences, however, we can force AP to split the four clusters into smaller sub-clusters:

```
apres2c <- apcluster(negDistMat(r=2), x2, q=0.8)
plot(apres2c, x2)
```

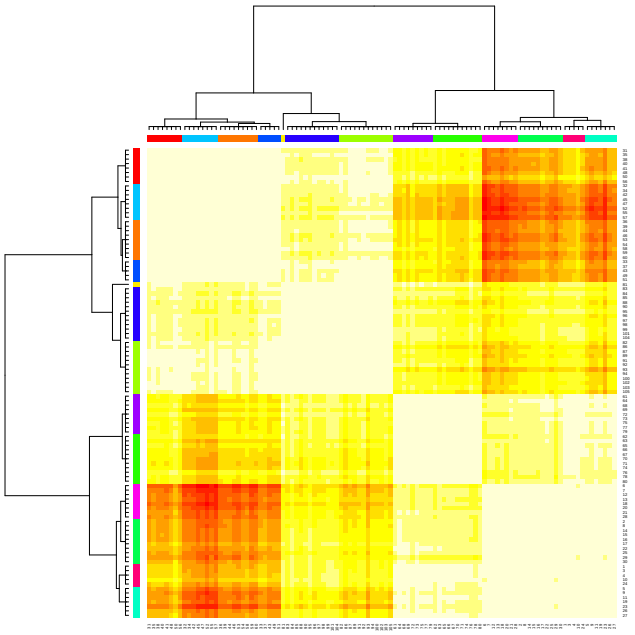


Note that the input preference used by AP can be recovered from the output object (no matter which method to adjust input preferences has been used). On the one hand, the value is printed if the object is displayed (by `show` or by entering the output object's name). On the other hand, the value can be accessed directly via the slot `p`:

```
apres2c@p

## [1] -0.009144609
```

As noted above already, we can produce a heatmap by calling `heatmap()` for an `APResult` object:

```
heatmap(apres2c)
```



The order in which the clusters are arranged in the heatmap is determined by means of joining the cluster agglomeratively (see Section 5 below). Although the affinity propagation result contains 13 clusters, the heatmap indicates that there are actually four clusters which can be seen as very brightly colored squares along the diagonal. We also see that there seem to be two pairs of adjacent clusters, which can be seen from the fact that there are two relatively light-colored blocks along the diagonal encompassing two of the four clusters in each case. If we look back at how the data have been created (see also plots above), this is exactly what is to be expected.

The above example with `q=0` demonstrates that setting input preferences to the minimum of input similarities does not necessarily result in a very small number of clusters (like one or two). This is due to the fact that input preferences need not necessarily be exactly in the range of the similarities. To determine a meaningful range, an auxiliary function is available which, in line with Frey's and Dueck's Matlab code,[1] allows to compute a minimum value (for which one or at most two clusters would be obtained) and a maximum value (for which as many clusters as data samples would be obtained):

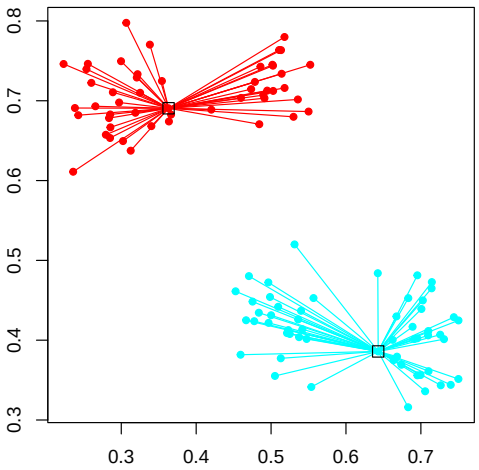
```
preferenceRange(apres2b@sim)

## [1] -5.136255e+00 -1.818538e-06
```

The function returns a two-element vector with the minimum value as first and the maximum value as second entry. The computations are done approximately by default. If one is interested in exact bounds, supply `exact=TRUE` (resulting in longer computation times).

Many clustering algorithms need to know a pre-defined number of clusters. This is often a major nuisance, since the exact number of clusters is hard to know for non-trivial (in particular, high-dimensional) data sets. AP avoids this problem. If, however, one still wants to require a fixed

number of clusters, this has to be accomplished by a search algorithm that adjusts input preferences in order to produce the desired number of clusters in the end. For convenience, this search algorithm is available as a function `apclusterK()` (analogous to Frey's and Dueck's Matlab implementation[1]). We can use this function to force AP to produce only two clusters (merging the two pairs of adjacent clouds into one cluster each). Analogously to `apcluster()`, `apclusterK()` supports two variants — it can either be called for a similarity measure and data or on a similarity matrix directly.

```
apres2d <- apclusterK(negDistMat(r=2), x2, K=2, verbose=TRUE)

## Trying p = -0.005138071
##    Number of clusters: 16
## Trying p = -0.05136435
##    Number of clusters: 4
## Trying p = -0.5136271
##    Number of clusters: 4
## Trying p = -2.568128 (bisection step no. 1 )
##    Number of clusters: 2
##
## Number of clusters: 2 for p = -2.568128

plot(apres2d, x2)
```

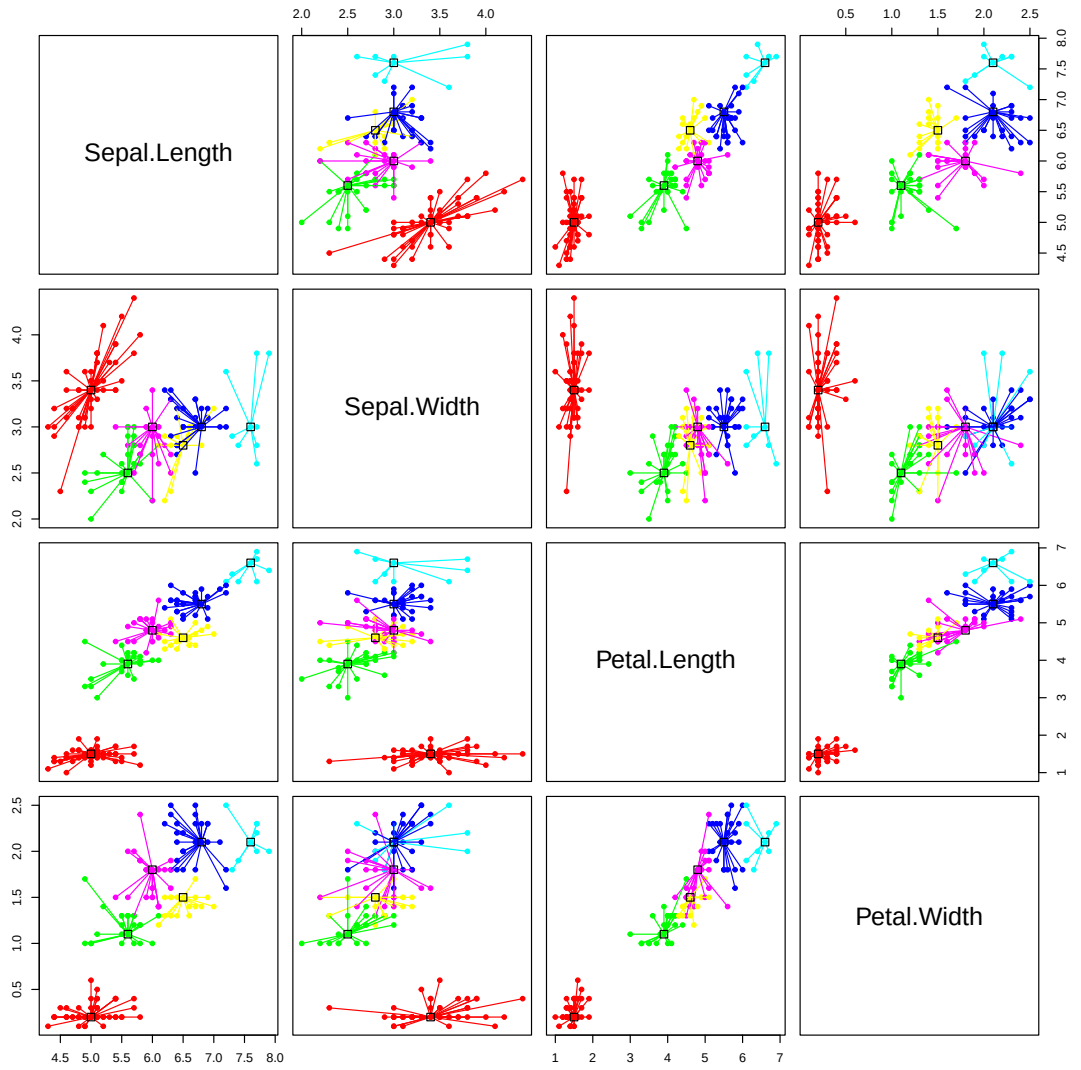


Now let us quickly consider a simple data set with more than two features. The notorious example is Fisher's iris data set:

```
data(iris)
apIris1 <- apcluster(negDistMat(r=2), iris)
apIris1

##
## APResult object
##
## Number of samples     =  150
## Number of iterations  =  162
## Input preference      =  -5.57
## Sum of similarities   =  -45.96
## Sum of preferences    =  -33.42
## Net similarity        =  -79.38
## Number of clusters    =  6
##
## Exemplars:
##    8 55 70 106 113 139
## Clusters:
##    Cluster 1, exemplar 8:
##       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
##       26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
##       48 49 50
##    Cluster 2, exemplar 55:
##       51 52 53 55 59 66 69 73 74 75 76 77 78 87 88 98 134
##    Cluster 3, exemplar 70:
##       54 58 60 61 63 65 68 70 72 80 81 82 83 89 90 91 93 94 95 96 97 99
##       100 107
##    Cluster 4, exemplar 106:
##       106 108 110 118 119 123 131 132 136
##    Cluster 5, exemplar 113:
##       101 103 104 105 109 111 112 113 116 117 121 125 126 129 130 133
##       137 138 140 141 142 144 145 146 148 149
##    Cluster 6, exemplar 139:
##       56 57 62 64 67 71 79 84 85 86 92 102 114 115 120 122 124 127 128
##       135 139 143 147 150
```

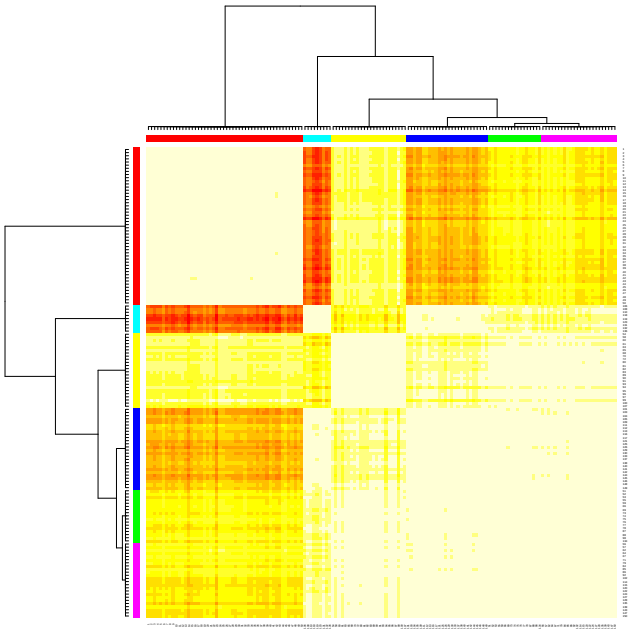
AP has identified 6 clusters. Since Version 1.3.2, the package also allows for superimposing clustering results in scatter plot matrices:

```
plot(apIris1, iris)
```



The heatmap looks as follows:

```
heatmap(apIris1)
```



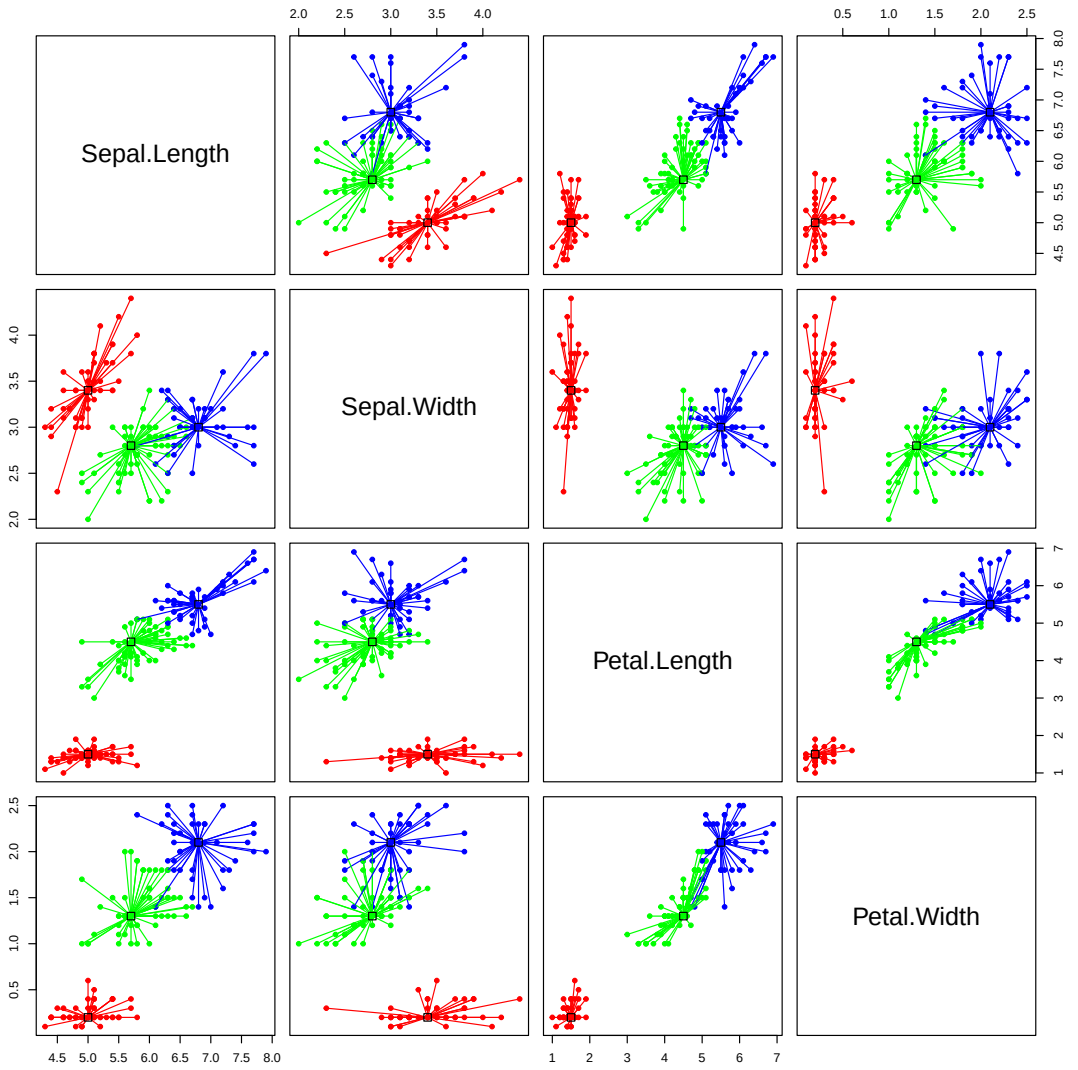
Now let us try to obtain fewer clusters by using the minimum of off-diagonal similarities:

```
data(iris)
apIris2 <- apcluster(negDistMat(r=2), iris, q=0)
apIris2

##
## APResult object
##
## Number of samples     =  150
## Number of iterations  =  126
## Input preference      =  -50.2
## Sum of similarities   =  -84.44
## Sum of preferences    =  -150.6
## Net similarity        =  -235.04
## Number of clusters    =  3
##
## Exemplars:
##    8 56 113
## Clusters:
##    Cluster 1, exemplar 8:
##       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
##       26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
##       48 49 50
##    Cluster 2, exemplar 56:
##       52 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
```

```
##       75 76 79 80 81 82 83 84 85 86 88 89 90 91 92 93 94 95 96 97 98 99
##       100 102 107 114 120 122 124 127 128 134 139 143 150
##    Cluster 3, exemplar 113:
##       51 53 77 78 87 101 103 104 105 106 108 109 110 111 112 113 115
##       116 117 118 119 121 123 125 126 129 130 131 132 133 135 136 137
##       138 140 141 142 144 145 146 147 148 149
```

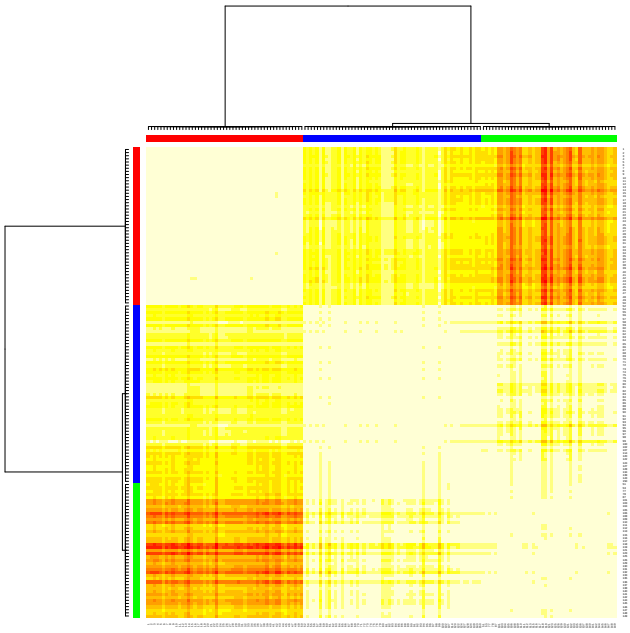
AP has identified 3 clusters. If we again superimpose them in the scatter plot matrix, we obtain the following:

```
plot(apIris2, iris)
```



Finally, the heatmap looks as follows:

```
heatmap(apIris2)
```



So, looking at the heatmap, the 3 clusters seem quite reasonable, at least in the light of the fact that there are three species in the data set, *Iris setosa*, *Iris versicolor*, and *Iris virginica*, where *Iris setosa* is very clearly separated from each other (first cluster in the heatmap) and the two others are partly overlapping.

5 Exemplar-based Agglomerative Clustering

The function `aggExCluster()` realizes what can best be described as "exemplar-based agglomerative clustering", i.e. agglomerative clustering whose merging objective is geared towards the identification of meaningful exemplars. Analogously to `apcluster()`, `aggExCluster()` supports two variants — it can either be called for a similarity measure and data or on matrix of pairwise similarities.

5.1 Getting started

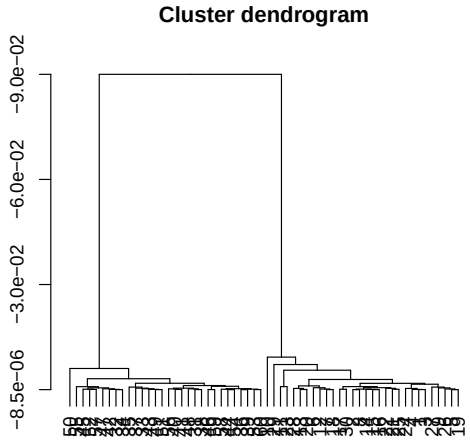
Let us start with a simple example:

```
aggres1a <- aggExCluster(negDistMat(r=2), x1)
aggres1a
```

```
##
## AggExResult object
##
## Number of samples          =  60
## Maximum number of clusters =  60
```

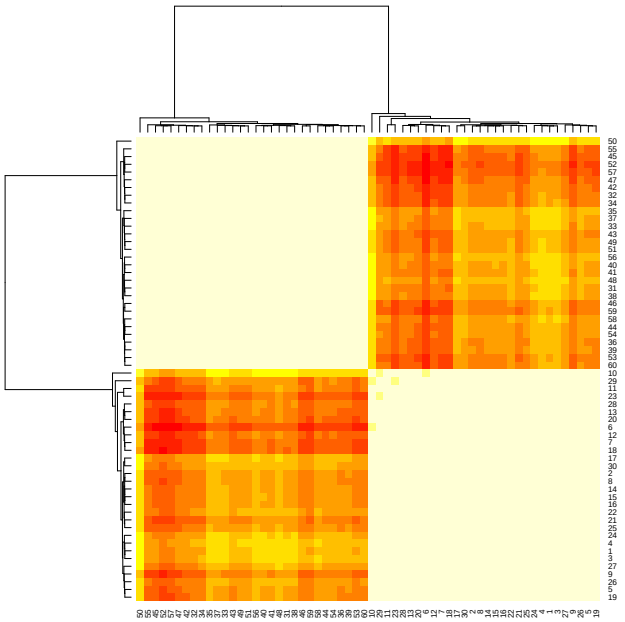
The output object `aggres1a` contains the complete cluster hierarchy. As obvious from the above example, the `show()` method only displays the most basic information. Calling `plot()` on an object that was the result of `aggExCluster()` (an object of class `AggExResult`), a dendrogram is plotted:

```
plot(aggres1a)
```



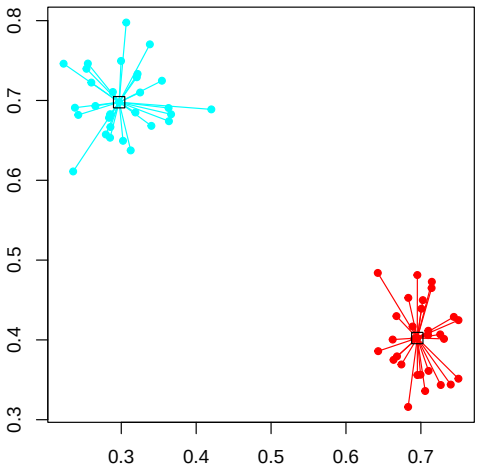
The heights of the merges in the dendrogram correspond to the merging objective: the higher the vertical bar of a merge, the less similar the two clusters have been. The dendrogram, therefore, clearly indicates two clusters. Heatmaps can be produced analogously as for `APResult` objects with the additional property that dendrograms are displayed on the top and on the left:

```
heatmap(aggres1a, s1)
```



Once we have confirmed the number of clusters, which is clearly 2 according to the dendrogram and the heatmap above, we can extract the level with two clusters from the cluster hierarchy. In concordance with standard R terminology, the function for doing this is called `cutree()`:

```
cl1a <- cutree(aggres1a, k=2)
cl1a

##
## ExClust object
##
## Number of samples   =  60
## Number of clusters  =  2
##
## Exemplars:
##    44 25
## Clusters:
##    Cluster 1, exemplar 44:
##       50 55 45 52 57 47 42 32 34 35 37 33 43 49 51 56 40 41 48 31 38 46
##       59 58 44 54 36 39 53 60
##    Cluster 2, exemplar 25:
##       10 29 11 23 28 13 20 6 12 7 18 17 30 2 8 14 15 16 22 21 25 24 4 1
##       3 27 9 26 5 19

plot(cl1a, x1)
```



5.2 Merging clusters obtained from affinity propagation

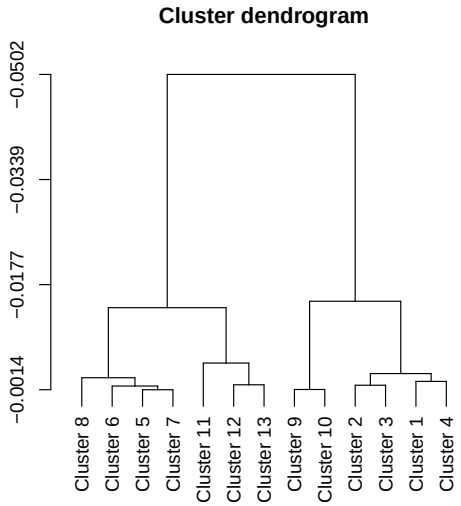
The most important application of `aggExCluster()` (and the reason why it is part of the `apcluster` package) is that it can be used for creating a hierarchy of clusters starting from a set of clusters previously computed by affinity propagation. The examples in Section 4 indicate that it may sometimes be tricky to define the right input preference. Exemplar-based agglomerative clustering on affinity propagation results provides an additional tool for finding the right number of clusters.

Let us revisit the four-cluster example from Section 4. We can apply `aggExCluster()` to an affinity propagation result if we run it on the affinity propagation result supplied as second argument `x`:

```
aggres2a <- aggExCluster(x=apres2c)
aggres2a

##
## AggExResult object
##
## Number of samples          =  105
## Maximum number of clusters =  13
```

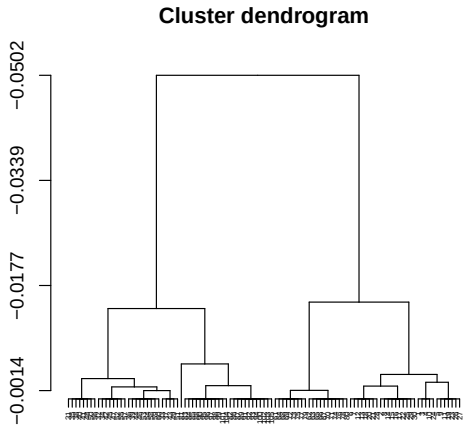
The result `apres2c` had 13 clusters. `aggExCluster()` successively joins these clusters until only one cluster is left. The dendrogram of this cluster hierarchy is given as follows:

```
plot(aggres2a)
```



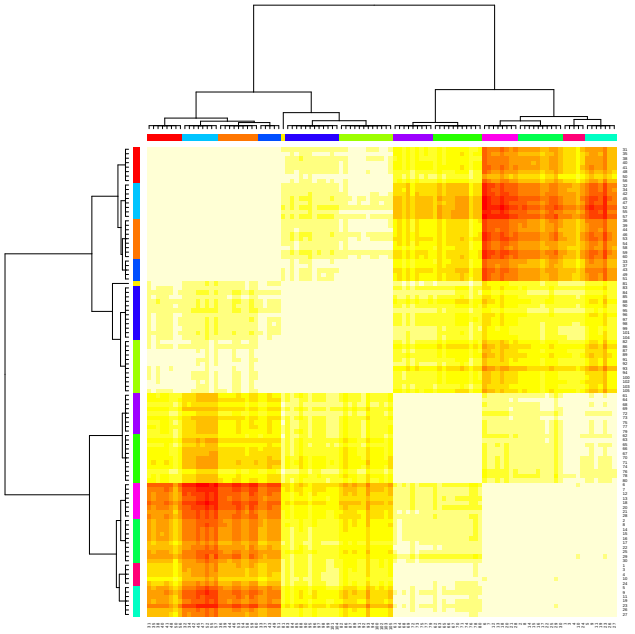
If one wants to see the original samples in the dendrogram of the cluster hierarchy, the `showSamples=TRUE` option can be used. In this case, it is recommended to reduce the font size of the labels via the `nodePar` parameter (see `?plot.dendrogram` and the examples therein):

```
plot(aggres2a, showSamples=TRUE, nodePar=list(pch=NA, lab.cex=0.4))
```



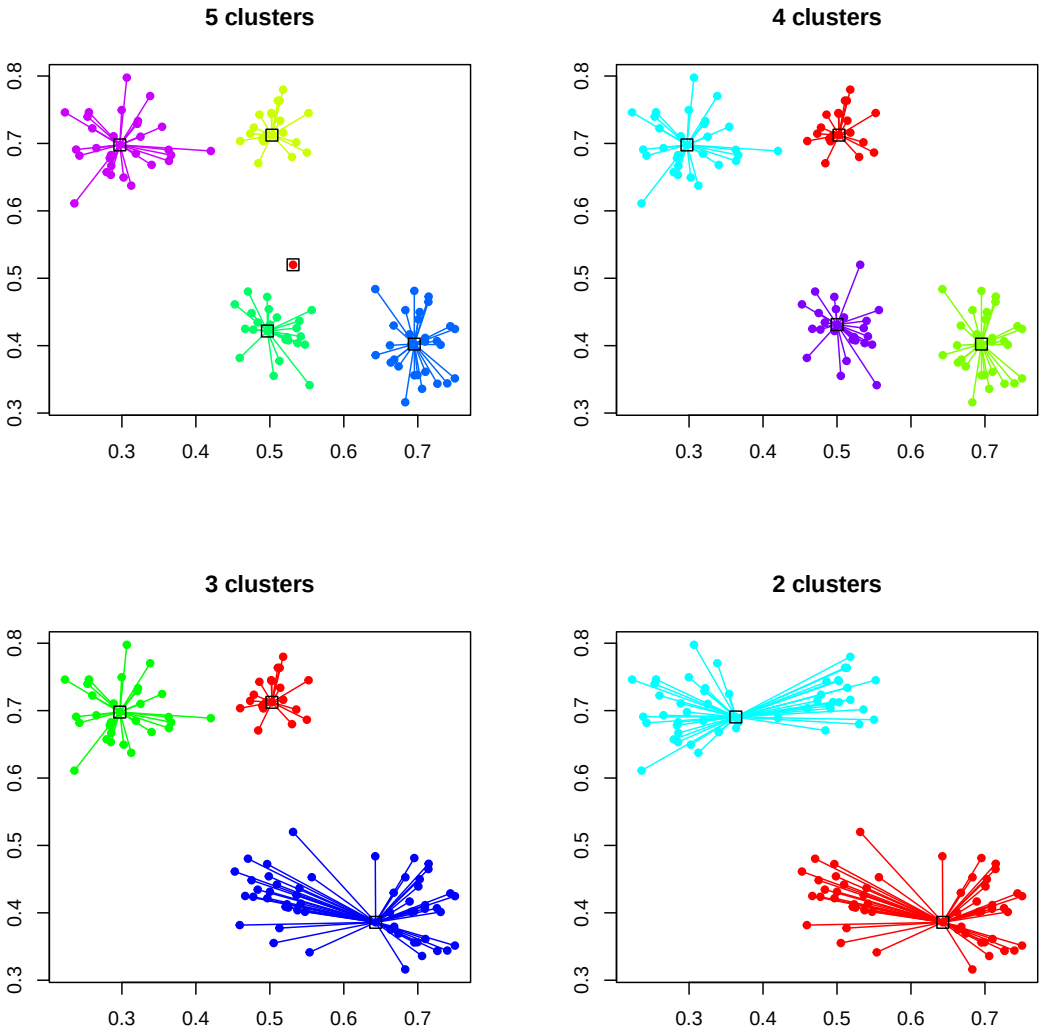
The following heatmap coincides with the one shown in Section 4 above. This is not surprising, since the heatmap plot for an affinity propagation result uses `aggExCluster()` internally to arrange the clusters:

```
heatmap(aggres2a)
```



Once we are more or less sure about the number of clusters, we extract the right clustering level from the hierarchy. For demonstation purposes, we do this for $k = 5, \ldots, 2$ in the following plots:

```
par(mfrow=c(2,2))
for (k in 5:2)
    plot(aggres2a, x2, k=k, main=paste(k, "clusters"))
```



There is one obvious, but important, condition: applying `aggExCluster()` to an affinity propagation result only makes sense if the number of clusters to start from is at least as large as the number of true clusters in the data set. Clearly, if the number of clusters is already too small, then merging will make the situation only worse.

5.3 Details on the merging objective

Like any other agglomerative clustering method (see, e.g., [7, 10, 13]), `aggExCluster()` merges clusters until only one cluster containing all samples is obtained. In each step, two clusters are merged into one, i.e. the number of clusters is reduced by one. The only aspect in which `aggExCluster()` differs from other methods is the merging objective.

Suppose we consider two clusters for possible merging, each of which is given by an index

set:

$$I = \{i_1, \ldots, i_{n_I}\} \text{ and } J = \{j_1, \ldots, j_{n_J}\}$$

Then we first determine the potential *joint exemplar* $\mathrm{ex}(I, J)$ as the sample that maximizes the average similarity to all samples in the joint cluster $I \cup J$:

$$\mathrm{ex}(I, J) = \underset{i \in I \cup J}{\operatorname{argmax}} \frac{1}{n_I + n_J} \cdot \sum_{j \in I \cup J} S_{ij}$$

Recall that $\mathbf{S}$ denotes the similarity matrix and S_{ij} corresponds to the similarity of the i-th and the j-th sample. Then the merging objective is computed as

$$\mathrm{obj}(I, J) = \frac{1}{2} \cdot \Big(\frac{1}{n_I} \cdot \sum_{j \in I} S_{\mathrm{ex}(I,J)j} + \frac{1}{n_J} \cdot \sum_{k \in J} S_{\mathrm{ex}(I,J)k} \Big),$$

which can be best described as "*balanced average similarity to the joint exemplar*". In each step, `aggExCluster()` considers all pairs of clusters in the current cluster set and joins that pair of clusters whose merging objective is maximal. The rationale behind the merging objective is that those two clusters should be joined that are best described by a joint exemplar.

6 Leveraged Affinity Propagation

Leveraged affinity propagation is based on the idea that, for large data sets with many samples, the cluster structure is already visible on a subset of the samples. Instead of evaluating the similarity matrix for all sample pairs, the similarities of all samples to a subset of samples are computed — resulting in a non-square similarity matrix. Clustering is performed on this reduced similarity matrix allowing for clustering large data sets more efficiently.

In this form of clustering, several rounds of affinity propagation are executed with different sample subsets — iteratively improving the clustering result. The implementation is based on the Matlab code of Frey and Dueck provided on the AP Web page[1]. Apart from dynamic improvements through reduced amount of distance calculations and faster clustering, the memory consumption is also reduced not only in terms of the memory used for storing the similarity matrix, but also in terms of memory used by the clustering algorithm internally.

The two main parameters controlling leveraged AP clustering are the fraction of data points that should be selected for clustering (parameter `frac`) and the number of sweeps or repetitions of individual clustering runs (parameter `sweeps`). Initially, a sample subset is selected randomly. For the subsequent repetitions, the exemplars of the previous run are kept in the sample subset and the other samples in the subset are chosen randomly again. The best result of all sweeps with the highest net similarity is kept as final clustering result.

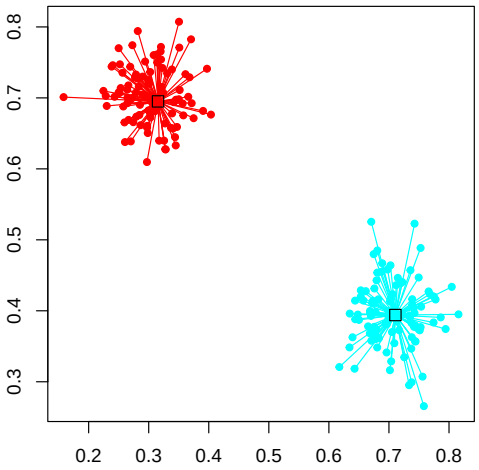
When called with a similarity measure and a dataset the function `apclusterL()` performs both the calculation of similarities and leveraged affinity propagation. In the example below, we use 10% of the samples and run 5 repetitions. The function implementing the similarity measure can either be passed as a function or as a function name (which must of course be resolvable in the current environment). Additional parameters for the distance calculation can be passed to `apclusterL()` which passes them on to the function implementing the similarity measure via the `...` argument list. In any case, this function must be implemented such that it expects the data in

its first argument `x` (a subsettable data structure, such as, a vector, matrix, data frame, or list) and that it takes the selection of "column objects" as a second argument `sel` which must be a set of column indices. The functions `negDistMat()`, `expSimMat()`, `linSimMat()`, `corSimMat()`, and `linKernel()` provided by the `apcluster` package also support the easy creation of parameter-free similarity measures (in R terminology called "closures"). We recommend this variant, as it is safer in terms of possible name conflicts between arguments of `apclusterL()` and arguments of the similarity function.

Here is an example that makes use of a closure for defining the similarity measure:

```
cl5 <- cbind(rnorm(100, 0.3, 0.05), rnorm(100, 0.7, 0.04))
cl6 <- cbind(rnorm(100, 0.70, 0.04), rnorm(100, 0.4, 0.05))
x3 <- rbind(cl5, cl6)
apres3 <- apclusterL(s=negDistMat(r=2), x=x3, frac=0.1, sweeps=5, p=-0.2)
apres3

##
## APResult object
##
## Number of samples     =  200
## Number of sel samples =  20    (10%)
## Number of sweeps      =  5
## Number of iterations  =  127
## Input preference      =  -0.2
## Sum of similarities   =  -0.7386853
## Sum of preferences    =  -0.4
## Net similarity        =  -1.138685
## Number of clusters    =  2
##
## Exemplars:
##    24 197
## Clusters:
##    Cluster 1, exemplar 24:
##       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
##       26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
##       48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
##       70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
##       92 93 94 95 96 97 98 99 100
##    Cluster 2, exemplar 197:
##       101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
##       117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132
##       133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148
##       149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164
##       165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
##       181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196
##       197 198 199 200
```

```
plot(apres3, x3)
```



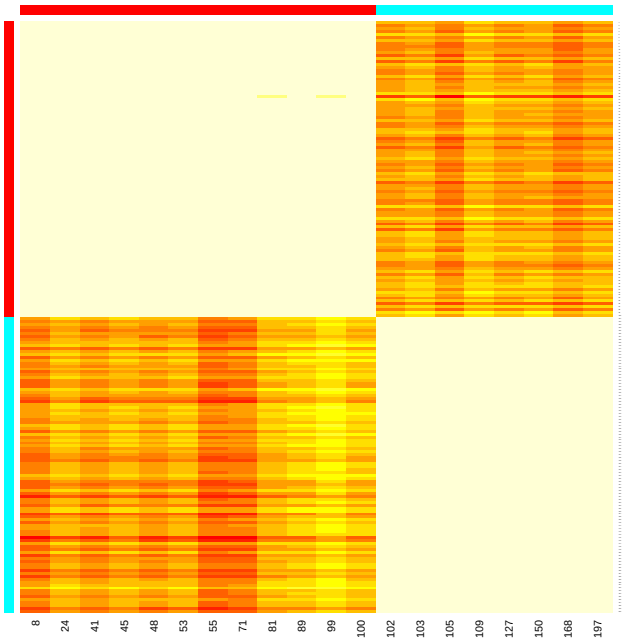
The function `apclusterL()` creates a result object of S4 class `APResult` that contains the same information as for standard AP. Additionally, the selected sample subset, the associated rectangular similarity matrix for the best sweep (provided that `includeSim=TRUE`) and the net similarities of all sweeps are returned in this object.

```
dim(apres3@sim)

## [1] 200  20

apres3@sel

##   8  24  41  45  48  53  55  71  81  89  99 100 102 103 105 109 127 150
##   8  24  41  45  48  53  55  71  81  89  99 100 102 103 105 109 127 150
## 168 197
## 168 197

apres3@netsimLev

## [1] -1.247198 -1.145307 -1.141661 -1.141661 -1.138685
```

The result returned by leveraged affinity propagation can be used for further processing in the same way as a result object returned from `apcluster()`, e.g., merging of clusters with agglomerative clustering can be performed.

For heatmap plotting either the parameter `includeSim=TRUE` must be set in `apcluster()` or `apclusterL()` to make the similarity matrix available in the result object or the similarity matrix must be passed as second parameter to `heatmap()` explicitly. The heatmap for leveraged AP looks slightly different compared to the heatmap for affinity propagation because the number of samples is different in both dimensions.

```
heatmap(apres3)
```



Often selected samples will be chosen as exemplars because, only for them, the full similarity information is available. This means that the fraction of samples should be selected in a way such that a considerable number of samples is available for each expected cluster. Please also note that a data set of the size used in this example can easily be clustered with regular affinity propagation. The data set was kept small to keep the package build time short and the amount of data output in the manual reasonable.

For users requiring a higher degree of flexibility, e.g., for a customized selection of the sample subset, `apclusterL()` called with a rectangular similiarity matrix performs affinity propagation on a rectangular similarity matrix. See the source code of `apclusterL()` with signature `s=function` and `x=ANY` for an example how to embed `apclusterL()` into a complete loop performing leveraged AP. The package-provided functions for distance calculation support the generation of rectangular similarity matrices (see Chapter 9).

7 Sparse Affinity Propagation

Starting with Version 1.4.0 of the `apcluster` package, the functions `apcluster()`, `apclusterK()`, and `preferenceRange()` can also handle similarity matrices as defined by the `Matrix` package.

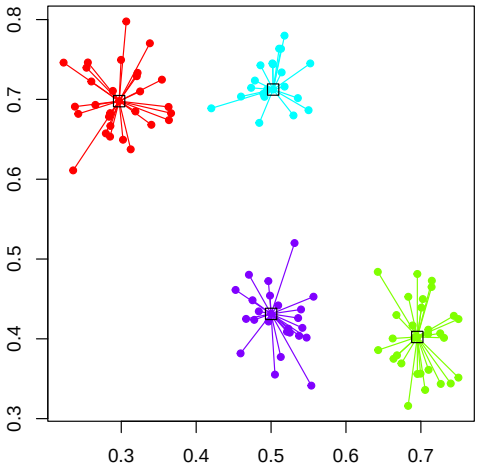
While all dense matrix formats are converted to standard R matrices, sparse matrices are converted internally to `dgTMatrix` objects. For these sparse matrices, special implementations of the `apcluster()`, `apclusterK()`, and `preferenceRange()` are available that fully exploit the sparseness of the matrices and may require much less operations if the matrix is sufficiently sparse.

In order to demonstrate that, consider the following example:

```
dsim <- negDistMat(x2, r=2)
ssim <- as.SparseSimilarityMatrix(dsim, lower=-0.2)
str(ssim)

## Formal class 'dgTMatrix' [package "Matrix"] with 6 slots
##   ..@ i       : int [1:9244] 1 2 3 4 5 6 7 8 9 10 ...
##   ..@ j       : int [1:9244] 0 0 0 0 0 0 0 0 0 0 ...
##   ..@ Dim     : int [1:2] 105 105
##   ..@ Dimnames:List of 2
##   .. ..$ : NULL
##   .. ..$ : NULL
##   ..@ x       : num [1:9244] -6.46e-03 -7.36e-05 -2.74e-04 -3.28e-03 -2.27e-02 ...
##   ..@ factors : list()
```

The function `as.SparseSimilarityMatrix()` converts the dense similarity matrix `dsim` into a sparse similarity matrix by removing all pairwise similarities that are -0.2 or lower. Note that this is only for demonstration purposes. If the size of data permits that, it is advisable to use the entire dense similarity matrix. Anyway, let us run sparse AP on this similarity matrix:

```
sapres <- apcluster(ssim, q=0)
plot(sapres, x2)
```

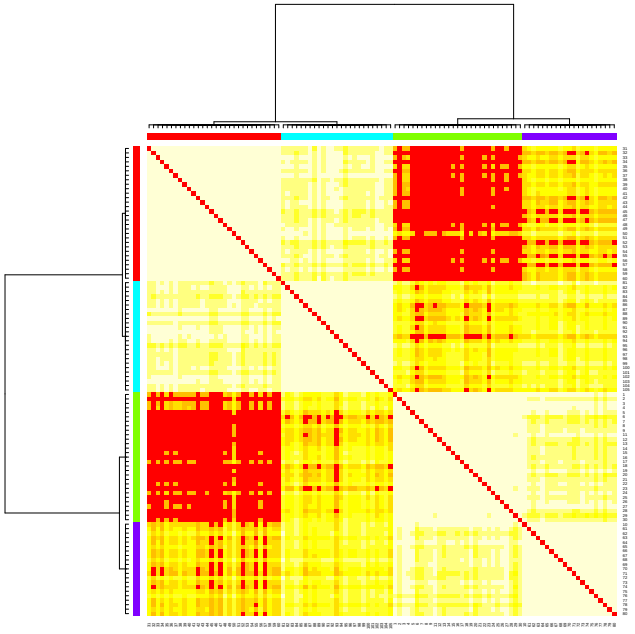


The functions `preferenceRange()` and `apclusterK()` work in the same way as for dense similarity matrices:

```
preferenceRange(ssim)

## [1] -3.186784e+00 -1.818538e-06

apclusterK(ssim, K=2)

## Trying p = -0.0031886
##    Number of clusters: 23
## Trying p = -0.03186964
##    Number of clusters: 5
## Trying p = -0.31868
##    Number of clusters: 4
## Trying p = -1.593393 (bisection step no. 1 )
##    Number of clusters: 2
##
## Number of clusters: 2 for p = -1.593393
##
## APResult object
##
## Number of samples     =  105
## Number of iterations  =  129
## Input preference      =  -1.593393
## Sum of similarities   =  -1.416411
## Sum of preferences    =  -3.186785
## Net similarity        =  -4.603197
## Number of clusters    =  2
##
## Exemplars:
##    1 37
## Clusters:
##    Cluster 1, exemplar 1:
##       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
##       26 27 28 29 30 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
##       78 79 80
##    Cluster 2, exemplar 37:
##       31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
##       53 54 55 56 57 58 59 60 81 82 83 84 85 86 87 88 89 90 91 92 93 94
##       95 96 97 98 99 100 101 102 103 104 105
```

The functions `aggExCluster()` and `heatmap()` have been extended to be able to handle sparse matrices. Note, however, that these functions are not yet exploiting sparsity properly. Instead, they convert all inputs to dense matrices before processing them, which may lead to memory and/or performance issues for large data sets.

```
heatmap(sapres, ssim)
```



The above heatmap illustrates that values that are not stored in the sparse similarity matrix are filled up with low values (see the red areas between some pairs of samples that belong to different clusters). Actually, each missing value is replaced with

$$\min(s) - (\max(s) - \min(s)) = 2 \cdot \min(s) - \max(s),$$

where $\min(s)$ and $\max(s)$ denote the smallest and the largest similarity value specified in the sparse similarity matrix s, respectively. The same replacement takes place when `aggExCluster()` converts sparse similarity matrices to dense ones.

8 Processing Biological Sequences

As noted in the introduction above, one of the goals of this package is to leverage affinity propagation in bioinformatics applications. Previous versions of this document showed a toy example of using affinity propagation on a set of biological sequences that computed a similarity matrix using the simple *spectrum kernel* [8] as implemented in the `kebabs` package [11]. This example has been removed in version 1.4.9 in order to avoid dependencies to a non-CRAN package. Instead, readers are now refered to the vignette of the `kebabs` package [11], which also includes an example how to use affinity propagation clustering on a set of biological sequences.

9 Similarity Matrices

Apart from the obvious monotonicity "the higher the value, the more similar two samples", affinity propagation does not make any specific assumption about the similarity measure. Negative

squared distances must be used if one wants to minimize squared errors [6]. Apart from that, the choice and implementation of the similarity measure is left to the user.

Our package offers a few more methods to obtain similarity matrices. The choice of the right one (and, consequently, the objective function the algorithm optimizes) still has to be made by the user.

All functions described in this section assume the input data matrix to be organized such that each row corresponds to one sample and each column corresponds to one feature (in line with the standard function `dist`). If a vector is supplied instead of a matrix, each single entry is interpreted as a (one-dimensional) sample.

9.1 The function `negDistMat()`

The function `negDistMat()`, in line with Frey and Dueck, allows for computing negative distances for a given set of real-valued data samples. If called with the first argument `x`, a similarity matrix with pairwise negative distances is returned:

```
s <- negDistMat(x2)
```

The function `negDistMat()` provides the same set of distance measures and parameters as the standard function `dist()` (except for `method="binary"` which makes little sense for real-valued data). Presently, `negDistMat()` provides the following variants of computing the distance $d(\mathbf{x}, \mathbf{y})$ of two data samples $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$:

Euclidean:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

use `method="euclidean"` or do not specify argument `method` (since this is the default);

Maximum:

$$d(\mathbf{x}, \mathbf{y}) = \max_{i=1}^{n} |x_i - y_i|$$

use `method="maximum"`;

Sum of absolute distances / Manhattan:

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} |x_i - y_i|$$

use `method="manhattan"`;

Canberra:

$$d(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i + y_i|}$$

summands with zero denominators are not taken into account; use `method="canberra"`;

Minkowski:

$$d(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{n} (x_i - y_i)^p \right)^{\frac{1}{p}}$$

use `method="minkowski"` and specify p using the additional argument `p` (default is `p=2`, resulting in the standard Euclidean distance);

Discrepancy:

$$d(\mathbf{x}, \mathbf{y}) = \max_{1 \le \alpha \le \beta \le n} \left| \sum_{i=\alpha}^{\beta} (y_i - x_i) \right|$$

use `method="discrepancy"` [14].

The function `negDistMat()` then takes the distances computed with one of the variants listed above and returns -1 times the r-th power of it, i.e.,

$$s(\mathbf{x}, \mathbf{y}) = -d(\mathbf{x}, \mathbf{y})^r. \tag{1}$$

The exponent r can be adjusted with the argument `r`. The default is `r=1`, hence, one has to supply `r=2` to obtain negative squared distances as in the examples in previous sections.

Here are some examples:

```
ex <- matrix(c(0, 0.5, 0.8, 1, 0, 0.2, 0.5, 0.7,
               0.1, 0, 1, 0.3, 1, 0.8, 0.2), 5, 3, byrow=TRUE)
ex
```

```
##      [,1] [,2] [,3]
## [1,]  0.0  0.5  0.8
## [2,]  1.0  0.0  0.2
## [3,]  0.5  0.7  0.1
## [4,]  0.0  1.0  0.3
## [5,]  1.0  0.8  0.2
```

Standard Euclidean distance:

```
negDistMat(ex)
```

```
##            1          2          3          4          5
## 1  0.0000000 -1.2688578 -0.8831761 -0.7071068 -1.2041595
## 2 -1.2688578  0.0000000 -0.8660254 -1.4177447 -0.8000000
## 3 -0.8831761 -0.8660254  0.0000000 -0.6164414 -0.5196152
## 4 -0.7071068 -1.4177447 -0.6164414  0.0000000 -1.0246951
## 5 -1.2041595 -0.8000000 -0.5196152 -1.0246951  0.0000000
```

Squared Euclidean distance:

```
negDistMat(ex, r=2)
```

```
##       1     2     3     4     5
## 1  0.00 -1.61 -0.78 -0.50 -1.45
## 2 -1.61  0.00 -0.75 -2.01 -0.64
## 3 -0.78 -0.75  0.00 -0.38 -0.27
## 4 -0.50 -2.01 -0.38  0.00 -1.05
## 5 -1.45 -0.64 -0.27 -1.05  0.00
```

Maximum norm-based distance:

```
negDistMat(ex, method="maximum")
```

```
##      1    2    3    4    5
## 1  0.0 -1.0 -0.7 -0.5 -1.0
## 2 -1.0  0.0 -0.7 -1.0 -0.8
## 3 -0.7 -0.7  0.0 -0.5 -0.5
## 4 -0.5 -1.0 -0.5  0.0 -1.0
## 5 -1.0 -0.8 -0.5 -1.0  0.0
```

Sum of absolute distances (aka Manhattan distance):

```
negDistMat(ex, method="manhattan")
```

```
##      1    2    3    4    5
## 1  0.0 -2.1 -1.4 -1.0 -1.9
## 2 -2.1  0.0 -1.3 -2.1 -0.8
## 3 -1.4 -1.3  0.0 -1.0 -0.7
## 4 -1.0 -2.1 -1.0  0.0 -1.3
## 5 -1.9 -0.8 -0.7 -1.3  0.0
```

Canberra distance:

```
negDistMat(ex, method="canberra")
```

```
##           1         2          3         4          5
## 1  0.000000 -2.600000 -1.9444444 -1.181818 -1.8307692
## 2 -2.600000  0.000000 -1.6666667 -2.200000 -1.0000000
## 3 -1.944444 -1.666667  0.0000000 -1.676471 -0.7333333
## 4 -1.181818 -2.200000 -1.6764706  0.000000 -1.3111111
## 5 -1.830769 -1.000000 -0.7333333 -1.311111  0.0000000
```

Minkowski distance for $p = 3$ (3-norm):

```
negDistMat(ex, method="minkowski", p=3)

##            1          2          3          4          5
## 1  0.0000000 -1.1027480 -0.7807925 -0.6299605 -1.0752028
## 2 -1.1027480  0.0000000 -0.7769462 -1.2601310 -0.8000000
## 3 -0.7807925 -0.7769462  0.0000000 -0.5428835 -0.5026526
## 4 -0.6299605 -1.2601310 -0.5428835  0.0000000 -1.0029910
## 5 -1.0752028 -0.8000000 -0.5026526 -1.0029910  0.0000000
```

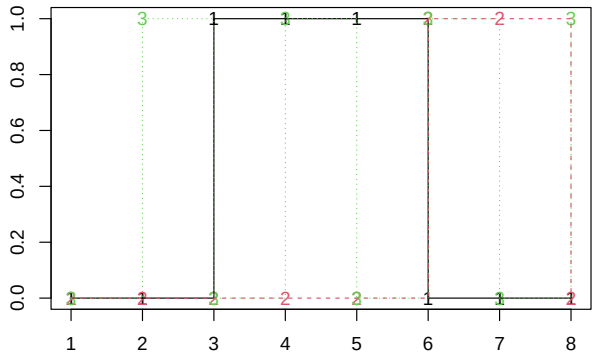
If called without the data argument `x`, a function object is returned that can be supplied to clustering functions — as in the majority of the above examples:

```
sim <- negDistMat(r=2)
is.function(sim)

## [1] TRUE

apcluster(sim, x1)

##
## APResult object
##
## Number of samples     =  60
## Number of iterations  =  131
## Input preference      =  -0.1416022
## Sum of similarities   =  -0.1955119
## Sum of preferences    =  -0.2832044
## Net similarity        =  -0.4787163
## Number of clusters    =  2
##
## Exemplars:
##    25 44
## Clusters:
##    Cluster 1, exemplar 25:
##       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
##       26 27 28 29 30
##    Cluster 2, exemplar 44:
##       31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
##       53 54 55 56 57 58 59 60
```

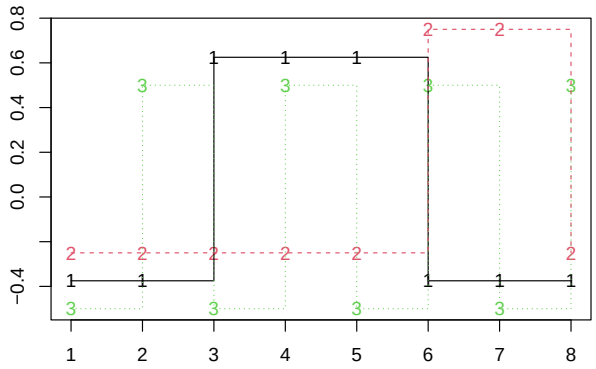
Depending on the application, it might be advisable to center and/or scale the data in order to equalize the influence of all features/columns. This makes sense for standard vector space distances like the Euclidean distance and can easily be accomplished by the `scale()` method. The discrepancy distance, in contrast, is strongly dependent on the order to feature/columns and

is rather aimed at comparing signals. For this measure, therefore, row-wise centering can be advisable [1]. This is easily done with the `sweep()` function:

```
ex2 <- matrix(c(0, 0, 1, 1, 1, 0, 0, 0,
                0, 0, 0, 0, 0, 1, 1, 0,
                0, 1, 0, 1, 0, 1, 0, 1), 3, 8, byrow=TRUE)

matplot(t(ex2), ylab="")
matlines(t(ex2), type="s")
```




```
negDistMat(ex2, method="discrepancy")

##    1  2  3
## 1  0 -3 -2
## 2 -3  0 -1
## 3 -2 -1  0

ex2Scaled <- sweep(ex2, 1, rowMeans(ex2))
ex2Scaled

##        [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]
## [1,] -0.375 -0.375  0.625  0.625  0.625 -0.375 -0.375 -0.375
## [2,] -0.250 -0.250 -0.250 -0.250 -0.250  0.750  0.750 -0.250
## [3,] -0.500  0.500 -0.500  0.500 -0.500  0.500 -0.500  0.500

matplot(t(ex2Scaled), ylab="")
matlines(t(ex2Scaled), type="s")
```




```
negDistMat(ex2Scaled, method="discrepancy")

##        1      2      3
## 1  0.000 -2.625 -2.375
## 2 -2.625  0.000 -1.750
## 3 -2.375 -1.750  0.000
```

9.2 Other similarity measures

The package `apcluster` offers four more functions for creating similarity matrices for real-valued data:

Exponential transformation of distances: the function `expSimMat()` works in the same way as the function `negDistMat()`. The difference is that, instead of the transformation (1), it uses the following transformation:

$$s(\mathbf{x},\mathbf{y}) = \exp\left(-\left(\frac{d(\mathbf{x},\mathbf{y})}{w}\right)^r\right)$$

Here the default is `r=2`. It is clear that `r=2` in conjunction with `method="euclidean"` results in the well-known *Gaussian kernel / RBF kernel* [5, 9, 12], whereas `r=1` in conjunction with `method="euclidean"` results in the similarity measure that is sometimes called *Laplace kernel* [5, 9]. Both variants (for non-Euclidean distances as well) can also be interpreted as *fuzzy equality/similarity relations* [3].

Linear scaling of distances with truncation: the function `linSimMat()` uses the transformation

$$s(\mathbf{x},\mathbf{y}) = \max\left(1 - \frac{d(\mathbf{x},\mathbf{y})}{w}, 0\right)$$

which is also often interpreted as a *fuzzy equality/similarity relation* [3].

Correlation: the function `corSimMat()` interprets the rows of its argument `x` (matrix or data frame) as multivariate observations and computes similarities as pairwise correlations. The function `corSimMat()` is actually a wrapper around the standard function `cor()`. Consequently, the `method` argument allows for selecting the type of correlation to compute (Pearson, Spearman, or Kendall).

Linear kernel: scalar products can also be interpreted as similarity measures, a view that is often adopted by kernel methods in machine learning. In order to provide the user with this option as well, the function `linKernel()` is available. For two data samples $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$, it computes the similarity as

$$s(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} x_i \cdot y_i.$$

The function has one additional argument, `normalize` (default: `FALSE`). If `normalize` is set to `TRUE`, values are normalized to the range $[-1, +1]$ in the following way:

$$s(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{n} x_i \cdot y_i}{\sqrt{\left(\sum_{i=1}^{n} x_i^2\right) \cdot \left(\sum_{i=1}^{n} y_i^2\right)}}$$

Entries for which at least one of the two factors in the denominator is zero are set to zero (however, the user should be aware that this should be avoided anyway).

For the same example data as above, we obtain the following for the RBF kernel:

```
expSimMat(ex)
```

```
##           1         2         3         4         5
## 1 1.0000000 0.1998876 0.4584060 0.6065307 0.2345703
## 2 0.1998876 1.0000000 0.4723666 0.1339887 0.5272924
## 3 0.4584060 0.4723666 1.0000000 0.6838614 0.7633795
## 4 0.6065307 0.1339887 0.6838614 1.0000000 0.3499377
## 5 0.2345703 0.5272924 0.7633795 0.3499377 1.0000000
```

Laplace kernel:

```
expSimMat(ex, r=1)
```

```
##           1         2         3         4         5
## 1 1.0000000 0.2811526 0.4134676 0.4930687 0.2999440
## 2 0.2811526 1.0000000 0.4206200 0.2422598 0.4493290
## 3 0.4134676 0.4206200 1.0000000 0.5398622 0.5947493
## 4 0.4930687 0.2422598 0.5398622 1.0000000 0.3589059
## 5 0.2999440 0.4493290 0.5947493 0.3589059 1.0000000
```

Pearson correlation coefficient:

```
corSimMat(ex, method="pearson")
```

```
##            1             2             3           4           5
## 1  1.0000000 -8.416976e-01 -5.399492e-01  0.42592613 -0.91129318
## 2 -0.8416976  1.000000e+00  4.007290e-17 -0.84702436  0.54470478
## 3 -0.5399492  4.007290e-17  1.000000e+00  0.53155407  0.83862787
## 4  0.4259261 -8.470244e-01  5.315541e-01  1.00000000 -0.01560216
## 5 -0.9112932  5.447048e-01  8.386279e-01 -0.01560216  1.00000000
```

Spearman rank correlation coefficient:

```
corSimMat(ex, method="spearman")
```

```
##      1    2    3    4    5
## 1  1.0 -0.5 -0.5  0.5 -1.0
## 2 -0.5  1.0 -0.5 -1.0  0.5
## 3 -0.5 -0.5  1.0  0.5  0.5
## 4  0.5 -1.0  0.5  1.0 -0.5
## 5 -1.0  0.5  0.5 -0.5  1.0
```

Linear scaling of distances with truncation:

```
linSimMat(ex, w=1.2)
```

```
##           1         2         3         4         5
## 1 1.0000000 0.0000000 0.2640199 0.4107443 0.0000000
## 2 0.0000000 1.0000000 0.2783122 0.0000000 0.3333333
## 3 0.2640199 0.2783122 1.0000000 0.4862988 0.5669873
## 4 0.4107443 0.0000000 0.4862988 1.0000000 0.1460874
## 5 0.0000000 0.3333333 0.5669873 0.1460874 1.0000000
```

Linear kernel:

```
linKernel(ex[2:5,])
```

```
##      1    2    3    4
## 1 1.04 0.52 0.06 1.04
## 2 0.52 0.75 0.73 1.08
## 3 0.06 0.73 1.09 0.86
## 4 1.04 1.08 0.86 1.68
```

Normalized linear kernel:

```
linKernel(ex[2:5,], normalize=TRUE)
```

```
##           1         2          3         4
## 1 1.00000000 0.5887841 0.05635356 0.7867958
## 2 0.58878406 1.0000000 0.80738184 0.9621405
## 3 0.05635356 0.8073818 1.00000000 0.6355220
## 4 0.78679579 0.9621405 0.63552196 1.0000000
```

All of these functions work in the same way as `negDistMat()`: if called with argument `x`, a similarity matrix is returned, otherwise a function is returned.

9.3 Rectangular similarity matrices

With the introduction of leveraged affinity propagation, distance calculations are entirely performed within the `apcluster` package. The code is based on a customized version of the `dist()` function from the `stats` package. In the following example, a rectangular similarity matrix of all samples against a subset of the samples is computed:

```
sel <- sort(sample(1:nrow(x1), ceiling(0.08 * nrow(x1))))
sel
```

```
## [1] 11 12 24 43 59
```

```
s1r <- negDistMat(x1, sel, r=2)
dim(s1r)
```

```
## [1] 60  5
```

```
s1r[1:7,]
```

```
##             11             12             24         43         59
## 1 -0.006982570 -0.0116441729 -0.0010272646 -0.1999332 -0.2205076
## 2 -0.011439132 -0.0025076066 -0.0032966100 -0.2479070 -0.2819561
## 3 -0.008483738 -0.0129106012 -0.0009003678 -0.1928484 -0.2137914
## 4 -0.009920010 -0.0130705475 -0.0005821650 -0.1892750 -0.2116157
## 5 -0.002005659 -0.0037264620 -0.0040861813 -0.2543323 -0.2770946
## 6 -0.013853342 -0.0019316461 -0.0198043331 -0.3454700 -0.3812374
## 7 -0.008096697 -0.0003397523 -0.0126183601 -0.3140531 -0.3458716
```

The rows correspond to all samples, the columns to the sample subset. The `sel` parameter specifies the sample indices of the selected samples in increasing order. Rectangular similarity calculation is provided in all distance functions of the package. If the parameter `sel` is not specified, the quadratic similarity matrix of all sample pairs is computed.

9.4 Defining a custom similarity measure for leveraged affinity propagation

As mentioned in Section 6 above, leveraged affinity propagation requires the definition of a similarity measure that is supplied as a function or function name to `apclusterL()`. For vectorial data, the similarity measures supplied with the package (see above) may be sufficient. If other similarity measures are necessary or if the data are not vectorial, the user must supply his/her own similarity measure. The user can supply any function as argument `s` to `apcluster()`, `apclusterK()`, or `apclusterL()`, but the following rules must be obeyed in order to avoid errors and to ensure meaningful results:

1. The data must be supplied as first argument, which must be named `x`.
2. The second argument must be named `sel` and must be interpreted as a vector of indices that select a subset of data items in `x`.
3. The function must return a numeric matrix with similarities. If `sel=NA`, the format of the matrix must be `length(x)`×`length(x)`. If `sel` is not `NA`, but contains indices selecting a subset, the format of the returned similarity matrix must be `length(x)`×`length(sel)`.
4. Although this is not a must, it is recommended to properly set row and column names in the returned similarity matrix.

9.5 Defining a custom similarity measure that creates a sparse similarity matrix

Since Version 1.4.0, similarity matrices may also be sparse (cf. Section 7). Correspondingly, the similarity measures passed to `apcluster()` and `apclusterK()` may also return sparse similarity matrices:

```
sparseSim <- function(x)
{
    as.SparseSimilarityMatrix(negDistMat(x, r=2), lower=-0.2)
}

sapres2 <- apcluster(sparseSim, x2, q=0)
sapres2


##
## APResult object
##
## Number of samples     =  105
## Number of iterations  =  132
## Input preference      =  -0.1999997
## Sum of similarities   =  -0.2791395
## Sum of preferences    =  -0.799999
## Net similarity        =  -1.079138
## Number of clusters    =  4
##
```

```
## Exemplars:
##    25 44 79 83
## Clusters:
##    Cluster 1, exemplar 25:
##       1 2 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
##       27 28 29 30
##    Cluster 2, exemplar 44:
##       31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
##       53 54 55 56 57 58 59 60
##    Cluster 3, exemplar 79:
##       10 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
##    Cluster 4, exemplar 83:
##       81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
##       102 103 104 105

str(similarity(sapres2))

## Formal class 'dgTMatrix' [package "Matrix"] with 6 slots
##   ..@ i       : int [1:9244] 1 2 3 4 5 6 7 8 9 10 ...
##   ..@ j       : int [1:9244] 0 0 0 0 0 0 0 0 0 0 ...
##   ..@ Dim     : int [1:2] 105 105
##   ..@ Dimnames:List of 2
##   .. ..$ : NULL
##   .. ..$ : NULL
##   ..@ x       : num [1:9244] -6.46e-03 -7.36e-05 -2.74e-04 -3.28e-03 -2.27e-02 ...
##   ..@ factors : list()
```

Note that similarity measures passed to `apclusterL()` may not return sparse matrices. Instead, they must accept a `sel` argument and return a rectangular dense matrix (see Subsection 9.4 above).

10 Miscellaneous

10.1 Convenience vs. efficiency

In most of the above examples, we called a clustering method by supplying it with a similarity function and the data to be clustered. This is undoubtedly a convenient approach. Since the resulting output objects (unless the option `includeSim=FALSE` is supplied) even includes the similarity matrix, we can plot heatmaps and produce a cluster hierarchy on the basis of the clustering result without the need to supply the similarity matrix explicitly.

For large data sets, however, this convenient approach has some disadvantages:

- If the clustering algorithm is run several times on the same data set (e.g., for different parameters), the similarity matrix is recomputed every time.

- Every clustering result (depending on the option `includeSim`) usually includes a copy of the similarity matrix.

For these reasons, depending on the actual application scenario, users should consider computing the similarity matrix beforehand. This strategy, however, requires some extra effort for subsequent processing, i.e. the similarity must be supplied as an extra argument in subsequent processing.

10.2 Clustering named objects

The function `apcluster()` and all functions for computing distance matrices are implemented to recognize names of data objects and to correctly pass them through computations. The mechanism is best described with a simple example:

```
x3 <- c(1, 2, 3, 7, 8, 9)
names(x3) <- c("a", "b", "c", "d", "e", "f")
s3 <- negDistMat(x3, r=2)
```

So we see that the `names` attribute must be used if a vector of named one-dimensional samples is to be clustered. If the data are not one-dimensional (a matrix or data frame), object names must be stored in the row names of the data matrix.

All functions for computing similarity matrices recognize the object names. The resulting similarity matrix has the list of names both as row and column names.

```
s3
```

```
##     a   b   c   d   e   f
## a   0  -1  -4 -36 -49 -64
## b  -1   0  -1 -25 -36 -49
## c  -4  -1   0 -16 -25 -36
## d -36 -25 -16   0  -1  -4
## e -49 -36 -25  -1   0  -1
## f -64 -49 -36  -4  -1   0
```

```
colnames(s3)
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

The function `apcluster()` and all related functions use column names of similarity matrices as object names. If object names are available, clustering results are by default shown by names.

```
apres3a <-apcluster(s3)
apres3a
```

```
##
## APResult object
##
## Number of samples     =  6
## Number of iterations  =  124
## Input preference      =  -25
## Sum of similarities   =  -4
## Sum of preferences    =  -50
## Net similarity        =  -54
## Number of clusters    =  2
##
## Exemplars:
##    b e
## Clusters:
##    Cluster 1, exemplar b:
##       a b c
##    Cluster 2, exemplar e:
##       d e f
```

```
apres3a@exemplars
```

```
## b e
## 2 5
```

```
apres3a@clusters
```

```
## [[1]]
## a b c
## 1 2 3
##
## [[2]]
## d e f
## 4 5 6
```

10.3 Computing a label vector from a clustering result

For later classification or comparisons with other clustering methods, it may be useful to compute a label vector from a clustering result. Our package provides an instance of the generic function `labels()` for this task. As obvious from the following example, the argument `type` can be used to determine how to compute the label vector.

```
apres3a@exemplars
```

```
## b e
```

```
## 2 5

labels(apres3a, type="names")

## [1] "b" "b" "b" "e" "e" "e"

labels(apres3a, type="exemplars")

## [1] 2 2 2 5 5 5

labels(apres3a, type="enum")

## [1] 1 1 1 2 2 2
```

The first choice, `"names"` (default), uses names of exemplars as labels (if names are available, otherwise an error message is displayed). The second choice, `"exemplars"`, uses indices of exemplars (enumerated as in the original data set). The third choice, `"enum"`, uses indices of clusters (consecutively numbered as stored in the slot `clusters` — analogous to the standard implementation of `cutree()` or the `clusters` field of the list returned by the standard function `kmeans()`).

10.4 Customizing heatmaps

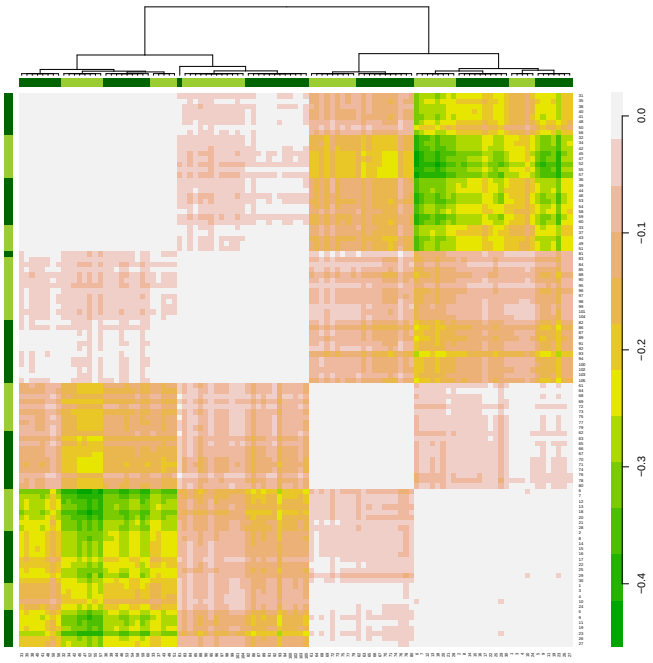
With Version 1.3.1, the implementation of heatmap plotting has changed significantly. The method now allows for many more customizations than before. Apart from changes in the argument list (see `?heatmap`), the behavior of the method has changed as follows:

- Dendrograms are always plotted if possible. To switch off plotting of dendrograms, set `Rowv` and `Colv` to `FALSE` or `NA`. If a dendrogram should only appear to the left of the heatmap, set `Colv` to `FALSE` or `NA`. Analogously, set `Rowv` to `FALSE` or `NA` if a dendrogram should only be plotted on top of the plot (not possible if the similarity matrix is non-quadratic).
- Previously, `rainbow()` was used internally to determine how the bars illustrating the clusters are colored. Now users can determine the coloring of the color bars using the `sideColors` argument. For `sideColors=NULL`, a meaningful color coding is determined automatically which still uses `rainbow()`, but ensures that no similar colors are placed next to each other in the bar.
- The default font sizes for displaying row/column labels have been changed to make sure that they do not overlap. This can result in quite small labels if the number of samples is larger. In any case, the user can override the sizes by making custom settings of the parameters `cexRow` and `cexCol`. Row and column labels can even be switched off entirely by setting `cexRow` and `cexCol` to 0, respectively.

Moreover, with Version 1.4.3, the possibility to add a color legend has been integrated.

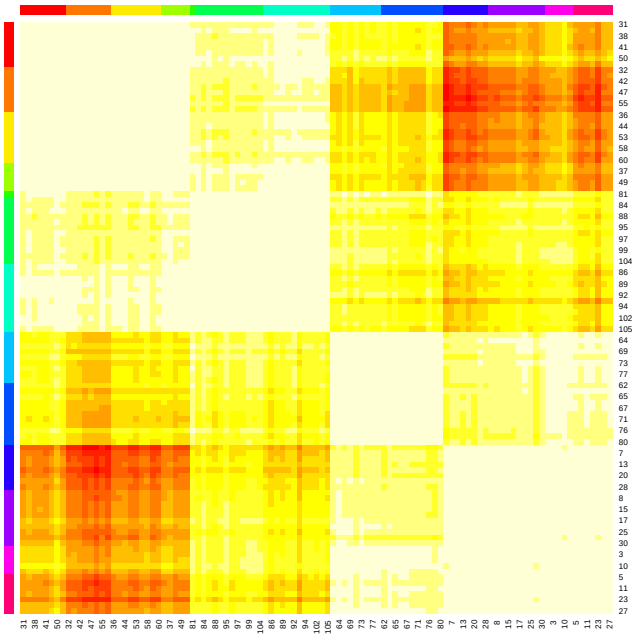
Here is an example with the vertical dendrogram switched off, an alternate color scheme, custom margins, and a color legend:

```
heatmap(apres2c, sideColors=c("darkgreen", "yellowgreen"),
        col=terrain.colors(12), Rowv=FALSE, dendScale=0.5,
        margins=c(3, 3, 2), legend="col")
```



The following example reverts to the default behavior prior to Version 1.3.1: consecutive rainbow colors, no dendrograms, and traditional sizing of row/column labels:

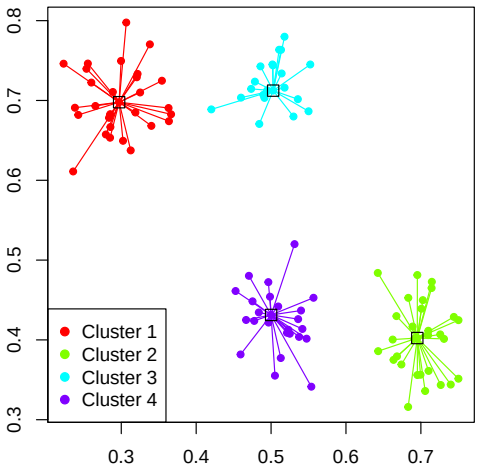
```
heatmap(apres2c, sideColors=rainbow(length(apres2c)), Rowv=FALSE, Colv=FALSE,
        cexRow=(0.2 + 1 / log10(nrow(apres2c@sim))),
        cexCol=(0.2 + 1 / log10(nrow(apres2c@sim))))
```



10.5 Adding a legend to plots of clustering results

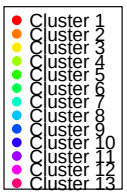
As shown above, `plot()` called for an `APResult` object as first and a matrix or data frame as second argument plots the clustering result superimposed on a scatter plot (or a scatter plot matrix if the number of columns in the second argument exceeds 2). The clusters are shown in different colors, but it may not be clear which cluster is shown in which color. Therefore, it may be useful to show a legend along with the plot. The current implementation of `plot()` does not show a legend, since it is hard to determine where to actually place the legend such that no important cluster information gets occluded by the legend. Therefore, the user has to add legends manually. Actually, colors are always chosen according to a simple rule: `plot()` uses `rainbow()` to create a vector of colors that is exactly as long as the number of clusters in the `APResult` object. The following example shows how to plot a legend manually (with the clusters enumerated in the same way as in the `APResult` object):

```
plot(apres2a, x2)
legend("bottomleft", legend=paste("Cluster", 1:length(apres2a)),
       col=rainbow(length(apres2a)), pch=19)
```



Note that this method is only meaningful for plotting clustering results superimposed on a 2D data set. For scatter plot matrices, this does not work in a meaningful way. In such a case, the user is rather recommended to create a legend separately (in a separate graphics device/file) and to display it along with the scatter plot matrix. To create only the legend, code like the following could be used:

```
plot.new()
par(oma=rep(0, 4), mar=rep(0, 4))
legend("center", legend=paste("Cluster", 1:length(apres2c)),
       col=rainbow(length(apres2c)), pch=19)
```



It still may be necessary to strip off white margins for further usage of the legend.

10.6 Implementation and performance issues

Prior to Version 1.2.0, `apcluster()` was implemented in R. Starting with version 1.2.0, the main iteration loop of `apcluster()` has been implemented in C++ using the Rcpp package [4], which has led to a speedup in the range of a factor or 9–10.

Note that `details=TRUE` requires quite an amount of additional memory. If possible, avoid this for larger data sets.

The asymptotic computational complexity of `aggExCluster()` is $\mathcal{O}(l^3)$ (where l is the number of samples or clusters from which the clustering starts). This may result in excessively long computation times if `aggExCluster()` is used for larger data sets without using affinity propagation first. For real-world data sets, in particular, if they are large, we recommend to use affinity propagation first and then, if necessary, to use `aggExCluster()` to create a cluster hierarchy.

11 Special Notes for Users Upgrading from Previous Versions

11.1 Upgrading from a version older than 1.3.0

Version 1.3.0 has brought several fundamental changes to the architecture of the package. We tried to ensure backward compatibility with previous versions where possible. However, there are still some caveats the users should take into account:

- The functions `apcluster()`, `apclusterK()`, and `aggExCluster()` have been re-implemented as S4 generics, therefore, they do not have a fixed list of arguments anymore. For this reason, users are recommended to name all optional parameters.
- Heatmap plotting has been shifted to the function `heatmap()` which has now been defined as an S4 generic method. Previous methods for plotting heatmaps using `plot()` have been partly available in Versions 1.3.0 and 1.3.1. Since Version 1.3.2, they are no longer available.

11.2 Upgrading to Version 1.3.3 or newer

Users who upgrade to Version 1.3.3 (or newer) from an older version should be aware that the package now requires a newer version of Rcpp. This issue can simply be solved by re-installing Rcpp from CRAN using `install.packages("Rcpp")`.

11.3 Upgrading to Version 1.4.0

The function `sparseToFull()` has been deprecated. A fully compatible function `as.DenseSimilarityMatrix()` is available that replaces and extends `sparseToFull()`.

11.4 Upgrading to Version 1.4.9

The function `sparseToFull()` that has been deprecated since version 1.4.0 has finally been removed completely. From now on, you really must use the function `as.DenseSimilarityMatrix()` that replaces and extends `sparseToFull()`.

Since the dependency to the `kebabs` package has been removed, the example file `inst/examples/ch22Promoter` has been removed, too.

12 How to Cite This Package

If you use this package for research that is published later, you are kindly asked to cite it as follows:

> U. Bodenhofer, A. Kothmeier, and S. Hochreiter (2011). APCluster: an R package for affinity propagation clustering. *Bioinformatics* **27**(17):2463–2464. DOI: 10.1093/bioinformatics/btr406.

Moreover, we insist that, any time you cite the package, you also cite the original paper in which affinity propagation has been introduced [6].

To obtain BibTEX entries of the two references, you can enter the following into your R session:

```
toBibtex(citation("apcluster"))
```

References

[1] P. Bauer, U. Bodenhofer, and E. P. Klement. A fuzzy algorithm for pixel classification based on the discrepancy norm. In *Proc. 5th IEEE Int. Conf. on Fuzzy Systems*, volume III, pages 2007–2012, New Orleans, LA, September 1996.

[2] U. Bodenhofer, A. Kothmeier, and S. Hochreiter. APCluster: an R package for affinity propagation clustering. *Bioinformatics*, 27(17):2463–2464, 2011.

[3] B. De Baets and R. Mesiar. Metrics and T-equalities. *J. Math. Anal. Appl.*, 267:531–547, 2002.

[4] D. Eddelbuettel and R. François. Rcpp: seamless R and C++ integration. *J. Stat. Softw.*, 40(8):1–18, 2011.

[5] C. H. FitzGerald, C. A. Micchelli, and A. Pinkus. Functions that preserve families of positive semidefinite matrices. *Linear Alg. Appl.*, 221:83–102, 1995.

[6] B. J. Frey and D. Dueck. Clustering by passing messages between data points. *Science*, 315(5814):972–976, 2007.

[7] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[8] C. Leslie, E. Eskin, and W. S. Noble. The spectrum kernel: a string kernel for SVM protein classification. In R. B. Altman, A. K. Dunker, L. Hunter, K. Lauderdale, and T. E. D. Klein, editors, *Pacific Symposium on Biocomputing 2002*, pages 566–575. World Scientific, 2002.

[9] C. A. Micchelli. Interpolation of scattered data: Distance matrices and conditionally positive definite functions. *Constr. Approx.*, 2:11–22, 1986.

[10] R. S. Michalski and R. E. Stepp. Clustering. In S. C. Shapiro, editor, *Encyclopedia of artificial intelligence*, pages 168–176. John Wiley & Sons, Chichester, 1992.

[11] J. Palme, S. Hochreiter, and U. Bodenhofer. KeBABS: an R package for kernel-based analysis of biological sequences. *Bioinformatics*, 31(15):2574–2576, 2015.

[12] B. Schölkopf and A. J. Smola. *Learning with Kernels*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2002.

[13] J. H. Ward Jr. Hierarchical grouping to optimize an objective function. *J. Amer. Statist. Assoc.*, 58:236–244, 1963.

[14] H. Weyl. Über die Gleichverteilung von Zahlen mod. Eins. *Math. Ann.*, 77:313–352, 1916.