Visualizing Common and Distinctive Components in Multi-block Data Using RegularizedSCA and qgraph in R

Josephina Cornelia Mathilda Arts
Data Science - Bachelor End Project
Tilburg University
Eindhoven University of Technology

Abstract

This paper introduces visualization methods for the **RegularizedSCA** package created by Gu and Van Deun (2019) for R (R Core Team 2013). **RegularizedSCA** performs an integrated component analysis of multi-block data that originates from different sources. **RegularizedSCA** outputs two matrices; one matrix with the loadings of the variables on the components (`P-matrix`) and one matrix with the scores of the individual cases on the components (`T-matrix`). Importantly, the components generated by **RegularizedSCA** account for the block structure and are either global, local, or distinctive components. Global components contain non-zero loadings from variables of all blocks, while local components contain non-zero loadings from variables of at least two, but not all blocks. Distinctive components only contain non-zero loadings from variables within one single block. Based on the output and the different types of components, three major categories of visualizations are developed. The data model visualizations create an overview of the structure of the variables, blocks, and components. The component loadings visualizations provide more detailed information on the loadings per block of variables for each component. Finally, the individual cases visualizations shows the position of each individual case against the different components and component types. The visualizations are captured within user-friendly code and can handle large data sets, making them a useful addition to **RegularizedSCA**.

Keywords: multi-block, complex data, visualization, common/distinctive components, component structure, **RegularizedSCA**, simultaneous component analysis, **qgraph**, R.

1. Introduction

The world is digitizing rapidly with data at the core of the transformation. Data has become an increasingly more important part of our lives, as the data-driven society is always, tracking, monitoring, listening, and watching (Reinsel, Gantz, and Rydning 2018). Recent technological developments have also caused an increase in the availability of complex data, by combining different sources about the same individual cases, creating multi-block data. This is often referred to as linked data, data fusion, or integrative data analysis (Van Mechelen and Smilde 2010). The analysis of such complex multi-block data is challenging, but very valuable, as many questions can be answered when analyzed properly. The introduction of new ways of sharing data on the World Wide Web has increased the amount of data available, and has enabled the combining of these data sets, generating new insights and developing patterns that were not detected before (Heath and Bizer 2011). Perceived randomness in data becomes more clear when analyzing these combined blocks of data simultaneously, but keeping their origin in mind.

In the medical field, multi-block data is generated by combining different so-called 'omics' data (genomics, proteomics, metabolomics, etc.), with multiple other parameters, such as clinical data, and food quality, providing cross-disciplinary insight into systems biology (Hassani, Martens, Qannari, Hanafi, Borge, and Kohler 2010). Notably, these data are of a high-dimensional nature, which adds to their complexity. Specifically, in the battle against Alzheimer's Disease (AD), it has become clear that extracting the most useful information from combined sources is critical. Multi-block data from imaging (MRI & PET), blood tests, and genetic data are combined to improve the diagnosis and understanding of AD (Xiang, Yuan, Fan, Wang, Thompson, and Ye 2014). The combination of data in health care, is not restricted to medical data only. Information coming directly from patient's wearables and user applications brings added value to the research of chronic obstructive pulmonary disease (COPD) and comorbidities supporting an integrated care system (Kilintzis, Chouvarda, Beredimas, Natsiavas, and Maglaveras 2019).

Not only the medical field benefits from multi-block data analysis, there are many more fields in which these methods have been successfully applied. Sánchez-Rada, Torres, Iglesias, Maestre, and Peinado (2014) used sentiment and emotion analysis of tweets combined with financial industry business ontology (FIBO) data to predict the Spanish stock market. Mavoa, Oliver, Witten, and Badland (2011) linked global positioning data (GPS) to self-reported diaries about trip purpose and perceptions, to get a better understanding of people's travel behavior. More examples can be found in the food industry, where multi-block data has found its way to determine the quality of edible oils (Rosa, de Figueiredo, Bonafé, Coqueiro, Visentainer, Março, Rutledge, and Valderrama 2017), increase the understanding of red winemaking (Aleixandre-Tudo and du Toit 2019), and reduce the level of NaCL (salt) in cheeses (Loudiyi, Rutledge, and Aït-Kaddour 2018).

Over the years many techniques have been developed that are equipped with the task of analyzing high-dimensional and multi-block data. Previously mentioned studies have used different methods such as, common dimension methods (ComDim) (Rosa *et al.* 2017; Loudiyi *et al.* 2018), consensus principal component analysis (CPCA) (Hassani *et al.* 2010), principal component analysis (PCA) (Aleixandre-Tudo and du Toit 2019), or simply adding all data together and using machine learning to find patterns (Sánchez-Rada *et al.* 2014; Mavoa *et al.* 2011). Other known methods are variations on the traditional PCA, such as groupwise-PCA (Camacho, Rodríguez-Gómez, and Saccenti 2017) and supervised PCA (Barshan, Ghodsi, Azimifar, and Jahromi 2011). There are many more techniques to be found, all of them are aimed at finding patterns within the huge amount of variables. In this paper, the focus is on regularized simultaneous component analysis, as created in the **RegularizedSCA** package by Gu and Van Deun (2019).

The **RegularizedSCA** method from Gu and Van Deun (2019) performs a joint analysis of multi-block data from different sources. The technique is an extension of simultaneous component analysis (SCA), which by itself is an extension of PCA. The advantage of SCA over PCA is the joint analysis of multiple blocks exposing the internal structure over the blocks and the shared information within them. **RegularizedSCA** has the additional benefit of not only identifying the shared information across all blocks (common components), but also the information that is unique to only certain blocks (distinct components). Although, **RegularizedSCA** is created with dimension reduction in mind and to make the understanding of high-dimensional data and the underlying associations between data blocks easier, its raw output is still somewhat challenging to interpret. It is not only relevant to get a correct statistical output, the real challenge lies in understanding these underlying patterns, making them visible, and extracting the uncovered information (Vellido, Martín, Rossi, and Lisboa 2010).

A picture is worth a thousand words - Is a very well-known saying in the English language and although its origin is debated, it still holds its value today. Especially in the topic of high-dimensional and multi-block data, this ancient saying says it all. Techniques for analysis are developing and we can learn more and more about the world using Big Data. However, these complicated analyses are often difficult to interpret. The human brain is highly evolved in detecting and recognizing patterns in visuals, which is believed to be the basis of our intelligence, imagination, and invention (Mattson 2014; Fang, Fuh, Yen, Cherng, and Chen 2004). However, it is not very strong computationally. Therefore, it is not only important to perform well-established analysis techniques, but also to visualize the output to make interpretation easier and to make the most use of the information that is extracted from the data.

This paper focuses on visualizing the output of the techniques developed in the **RegularizedSCA** package. The following sections provide more details on the output of **RegularizedSCA** and the visualization objectives. Next, two relatively small multi-block data sets are introduced that guide the user through the visualizations and the code. The visualizations are divided into three sections, which visualize the data model (component structure), the component loadings, and the individual cases (e.g., patients, subjects). Finally, three additional data sets, with increasing number of variables, show the developed visualizations to demonstrate the type of questions that can be answered.

2. RegularizedSCA package

RegularizedSCA is developed by Gu and Van Deun (2019) and focuses on the joint analysis of multi-block data. It regularizes the standard SCA method by using Lasso and Group Lasso to create sparse results and identify the component structure. Each component consists of loadings for variables that belong to a specific block. Blocks are the different sources of data about the same individual cases.

2.1. Model selection

RegularizedSCA roughly incorporates three stages of analysis. In the first stage the number of components in the data structure is determined, while in the second stage the structure of those components is established. The final stage evaluates the sparseness of the components. Gu and Van Deun (2019) provide a clear overview of the different stages and the methods that are used within each stage. They identify six different models from which the user can choose and provide a detailed description of these methods. Important for the visualization of the package is that all models output two matrices; one matrix with the loadings of the variables on the components and one matrix with the scores of the individual cases on the components. The former are usually named loadings and are stored in the `P-matrix`, while the latter are named scores and are stored in the `T-matrix`. The matrix with loadings (`P-matrix`) is going to be the basis for the data model (component structure) visualizations and component loadings visualizations. The `P-matrix` is similar to the matrix shown in Figure 1. The major difference between these two is that the `P-matrix` is not directly divided into blocks, rather all variables are sequentially shown with their original variable labels. An additional vector needs to identify the block structure. The matrix containing the component scores of the individual cases (`T-matrix`) will be used to identify clusters within the individual cases, based on the three levels of components (global, local, distinctive).

2.2. Component structure

The component structure identifies the location of zero-loadings in the data matrix, creating sparseness and the different types of components. According to the definition of Gu and Van Deun (2019) a common component has at least one loading within each block that is non-zero, explaining variation across all data blocks. Additionally, a distinctive component only has non-zero loadings in one single block and thus only reflects the variation in one specific block. A subset of the common components are those components that do not have non-zero loadings across all blocks, but do have non-zero loadings in at least two blocks. These are called local common components, while the first mentioned components are referred to as global common components. Figure 1 shows an example loadings matrix of possible output from **RegularizedSCA** in which the first component is identified as global common, as it has non-zero loadings in all three blocks. The second component is local common to block 2 and 3, while the third component is distinctive to block 1. In addition to identifying the different types of components, it is now also easy to see which variable contributes to which component. However, when the matrix grows very large, such as the case with many variables, a matrix alone does not provide sufficient insight into the complete structure. **RegularizedSCA** can be used both when the underlying component structure is known and when it is unknown by choosing among different models.

$$\begin{array}{l|ccc|}
 & \multicolumn{3}{c}{\textbf{Components}} \\
 & \textbf{1} & \textbf{2} & \textbf{3} \\
 & X & 0 & X \\
\textbf{Block 1} & X & 0 & X \\
 & 0 & 0 & X \\
 & X & X & 0 \\
\textbf{Block 2} & 0 & X & 0 \\
 & X & 0 & 0 \\
 & X & X & 0 \\
\textbf{Block 3} & 0 & X & 0 \\
 & 0 & X & 0
\end{array}$$

Figure 1: Example of components (columns) and variables (rows) in a loadings matrix, as generated by **RegularizedSCA** where X represent non-zero loadings and 0 are zero loadings.

2.3. Sign indeterminacy of loadings

The component loadings in the `P-matrix` generated by **RegularizedSCA** are either positive, negative, or zero. The signs of the loadings however are subjective. In case the entire matrix is multiplied with -1, the signs all change to their opposite. If a component exists entirely out of positive or negative loadings, it means that all variables are positively associated with each other. In case there is a mix of positive and negative loadings, there are also negative associations to be found. The notion of this interchangeability of the signs is important for the interpretation of the visualizations.

3. Visualization methodology

The output generated by **RegularizedSCA**, contains information on blocks, components, variables, and individual cases. These elements all need to be reflected within the visualizations. However, combining details on all of these elements in one graph is too much to handle as elements start to overlap even in the smallest data sets. The general goal of this paper is to develop visualizations for the **RegularizedSCA** output. Moreover, several additional subgoals for the visualizations development are set:

1. Show the common (global and local) and distinctive component structure of the data as revealed by **RegularizedSCA**.
2. Develop visualizations specifically for the **RegularizedSCA** output.
3. Develop visualizations that are appealing to look at.
4. Create static visualizations that can be used in future papers.
5. Develop user-friendly code (simple, clean, intuitive, and reliable).
6. Develop high-quality visualizations that can handle large data sets.

Several options are created to serve the different needs of the user and with the set goals in mind. These options are roughly divided into three different types. At first the model visualizations show an overview of the entire structure of blocks, components, variables, and loadings. Second, the component loadings visualizations, give a more detailed overview of the composition of each component and the loading values of each variable that contributes to each component. And finally the individual cases visualizations show the individual cases (e.g., patients, subjects) plotted against the established components and component types (global, local, distinctive).

One important assumption is, that all visualizations that are created with the code that is developed in this paper, are saved using the standard `pdf()` function in R to ensure maximum quality and resolution. Therefore, whenever code is stated in the main sections, the best way to render the visualization is to add the following code, where `mypath` needs to be set to the folder where the visualizations should be saved. Change the title to a suitable file name for the stored visualization.

```
pdf(file=file.path(mypath, "title.pdf"), width=10, height=8)
----- visualization code -----
dev.off()
```

The set width and height were chosen to suit the visualizations that are created in this paper. The advantage of using `pdf()` is that users are able to determine their own size of the plots.

4. Family and Herring data

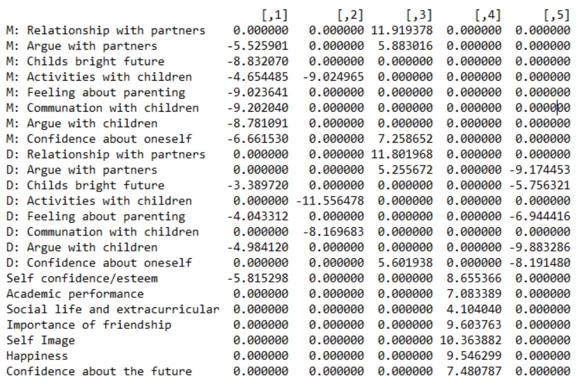
Throughout this paper two relatively small data sets are used to show the progress and results of the visualizations; these are the so-called family data and herring data. The family data set is based on a behavioural study about the parent-child relationship. It consists of survey data of 195 families in which parents and their child filled out questionnaires regarding different relationship topics. A joint analysis of these data blocks shows the association and influence of parents on their child. Gu and Van Deun (2019) have analyzed the family data according to model 4. This means that the VAF method is used to determine the number of components, which is found to be five. In the second step cross-validation is used to find the component structure and `sparseSCA()` is used to estimate the final model. Because of the use of Lasso in `structuredSCA()`, all non-zero loadings are shrunken towards zero, which is reversed with `undoShrinkage()`. To recreate the output from Gu and Van Deun (2019), which is used for the visualizations, the following R-code is used:

```
load(family_data.Rdata)
Mom <- pre_process(family_data$mom)
Dad <- pre_process(family_data$dad)
Child <- pre_process(family_data$child)
fam_data <- cbind(Mom, Dad, Child)
num_var_fam <- cbind(dim(Mom)[2], dim(Dad)[2], dim(Child)[2])
set.seed(115)
results_fam <- sparseSCA(fam_data, num_var_fam, R = 5, LASSO = 3.732885,
                         GROUPLASSO = 0.4278975, NRSTART = 20)
data_fam <- undoShrinkage(fam_data, R = 5, results_fam$Pmatrix)
```

The second data set, the herring data, consists of two blocks from 21 herring samples. The first block contains chemical compound variables of the herring samples, while the second block describes sensory variables on the same 21 herring samples. A joint analysis shows the association between changes in the chemical structure of herring and the way the herring feels and smells (sensory variables). The herring data set is analyzed according to model 5 as described by (Gu and Van Deun 2019). Gu and Van Deun (2019) have shown that the PCA-GCA method indicates that four components should be used and have indicated a target matrix to constrain the loadings for some of the data blocks to zero. The `structuredSCA()` function is used to estimate the final model. Even though the Lasso is set to zero in this case, and no shrinkage of non-zero loadings should have occurred, `undoShrinkage()` is applied to generate the same results as in Gu and Van Deun (2019). To recreate the output from Gu and Van Deun (2019), which is used for the visualizations the following R-code is used:

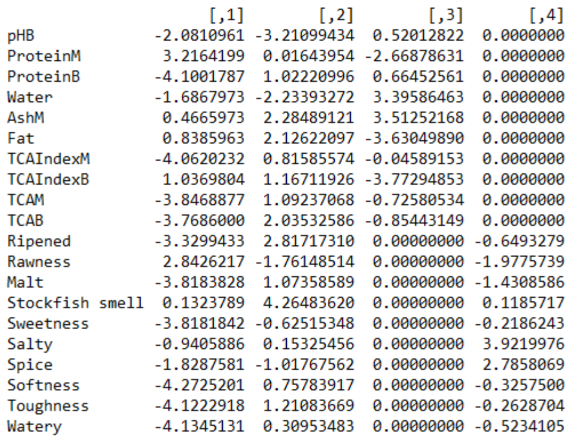
```
ChemPhy <- pre_process(Herring$Herring_ChemPhy)
Sensory <- pre_process(Herring$Herring_Sensory)
herring_data <- cbind(ChemPhy, Sensory)
num_var_her <- cbind(dim(ChemPhy)[2], dim(Sensory)[2])
targetmatrix <- matrix(c(1, 1, 1, 1, 1, 0, 0, 1), nrow = 2, ncol = 4)
set.seed(115)
results_her <- structuredSCA(DATA=herring_data, Jk=num_var_her, R=4,
                             Target=targetmatrix, LASSO=0)
data_her <- undoShrinkage(DATA=herring_data, R=4,Phat=results_her$Pmatrix)
```

The described process results in a list (`data_fam` and `data_her`) for each data set that contains three items; the component loadings matrix (`P-matrix`), the component scores matrix (`T-matrix`), and a vector containing the loss after each iteration. Figure 2 shows the component loadings matrices of the herring and family data analysis. The component scores matrices for the individual cases are not shown, as these have a similar structure (rows are individual cases instead of variables). Both the `P-matrix` and `T-matrix` form the basis for the visualizations.



```
                                  [,1]       [,2]      [,3]      [,4]      [,5]
M: Relationship with partners     0.000000   0.000000 11.919378  0.000000  0.000000
M: Argue with partners           -5.525901   0.000000  5.883016  0.000000  0.000000
M: Childs bright future          -8.832070   0.000000  0.000000  0.000000  0.000000
M: Activities with children      -4.654485  -9.024965  0.000000  0.000000  0.000000
M: Feeling about parenting       -9.023641   0.000000  0.000000  0.000000  0.000000
M: Communation with children     -9.202040   0.000000  0.000000  0.000000  0.000000
M: Argue with children           -8.781091   0.000000  0.000000  0.000000  0.000000
M: Confidence about oneself      -6.661530   0.000000  7.258652  0.000000  0.000000
D: Relationship with partners     0.000000   0.000000 11.801968  0.000000  0.000000
D: Argue with partners            0.000000   0.000000  5.255672  0.000000 -9.174453
D: Childs bright future          -3.389720   0.000000  0.000000  0.000000 -5.756321
D: Activities with children       0.000000 -11.556478  0.000000  0.000000  0.000000
D: Feeling about parenting       -4.043312   0.000000  0.000000  0.000000 -6.944416
D: Communation with children      0.000000  -8.169683  0.000000  0.000000  0.000000
D: Argue with children           -4.984120   0.000000  0.000000  0.000000 -9.883286
D: Confidence about oneself       0.000000   0.000000  5.601938  0.000000 -8.191480
Self confidence/esteem           -5.815298   0.000000  0.000000  8.655366  0.000000
Academic performance              0.000000   0.000000  0.000000  7.083389  0.000000
Social life and extracurricular   0.000000   0.000000  0.000000  4.104040  0.000000
Importance of friendship          0.000000   0.000000  0.000000  9.603763  0.000000
Self Image                        0.000000   0.000000  0.000000 10.363882  0.000000
Happiness                         0.000000   0.000000  0.000000  9.546299  0.000000
Confidence about the future       0.000000   0.000000  0.000000  7.480787  0.000000
```

(1)



```
                      [,1]        [,2]        [,3]       [,4]
pHB             -2.0810961 -3.21099434  0.52012822  0.0000000
ProteinM         3.2164199  0.01643954 -2.66878631  0.0000000
ProteinB        -4.1001787  1.02220996  0.66452561  0.0000000
Water           -1.6867973 -2.23393272  3.39586463  0.0000000
AshM             0.4665973  2.28489121  3.51252168  0.0000000
Fat              0.8385963  2.12622097 -3.63049890  0.0000000
TCAIndexM       -4.0620232  0.81585574 -0.04589153  0.0000000
TCAIndexB        1.0369804  1.16711926 -3.77294853  0.0000000
TCAM            -3.8468877  1.09237068 -0.72580534  0.0000000
TCAB            -3.7686000  2.03532586 -0.85443149  0.0000000
Ripened         -3.3299433  2.81717310  0.00000000 -0.6493279
Rawness          2.8426217 -1.76148514  0.00000000 -1.9775739
Malt            -3.8183828  1.07358589  0.00000000 -1.4308586
Stockfish smell  0.1323789  4.26483620  0.00000000  0.1185717
Sweetness       -3.8181842 -0.62515348  0.00000000 -0.2186243
Salty           -0.9405886  0.15325456  0.00000000  3.9219976
Spice           -1.8287581 -1.01767562  0.00000000  2.7858069
Softness        -4.2725201  0.75783917  0.00000000 -0.3257500
Toughness       -4.1222918  1.21083669  0.00000000 -0.2628704
Watery          -4.1345131  0.30953483  0.00000000 -0.5234105
```

(2)

Figure 2: Component loadings matrices (`P-matrix`) output from **RegularizedSCA** applied on family data (1) and herring data (2).

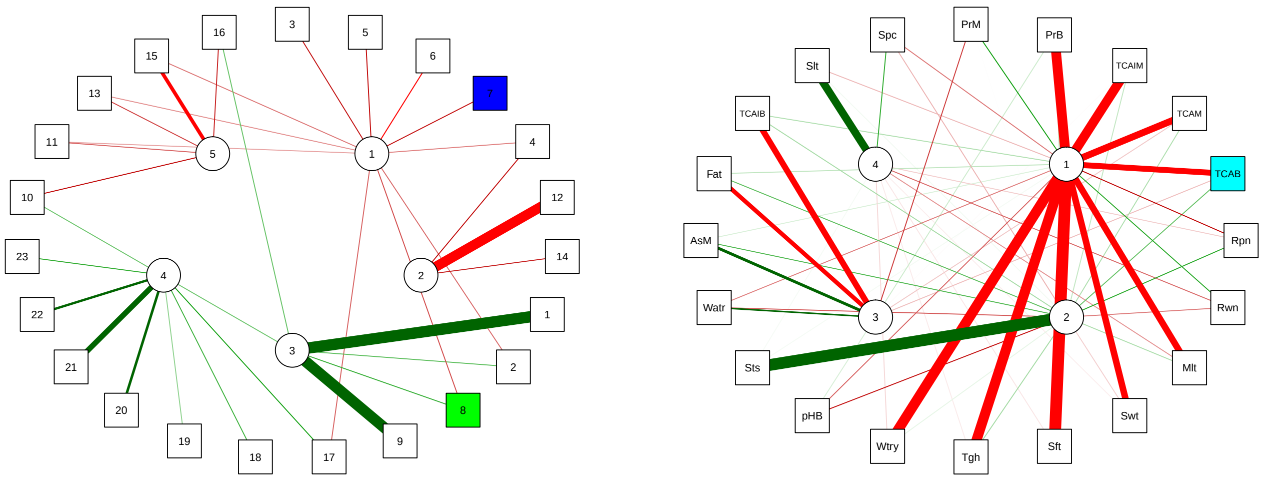
5. Data model visualizations

After completing the **RegularizedSCA** analysis, a first step is to get a global overview of the data structure, as revealed by the analysis. In these data structure visualizations there are two options; a tree model and a circle model. Both visualizations show the variables, components, blocks, and the presence or absence of connections between variables and components (loadings). The R-package **qgraph** entails a plotting function named `qgraph.loadings()`, which is already devoted to plotting a component loadings matrix, originally from either a PCA, exploratory factor analysis (EFA), or confirmatory factor analysis (CFA) (Epskamp, Schmittmann, and Borsboom 2012). The basic structure of the envisioned visualizations is already present within this function. However, adaptation to the specifics of multi-block data and the output of **RegularizedSCA** is necessary. The `qgraph.loadings()` function creates a `qgraph object`, which contains several lists with different parameters. This object lends itself perfectly for adjusting the visualization to the set goals from Section 3. Figure 3 shows the output of `qgraph.loadings()` before any changes are made, including the code that creates this output.

These graphs make a good starting point, but there is much left to improve. Some of the major improvement points are that the visualizations currently do not properly indicate to which block each variable belongs, the variable names in the family plot are replaced with numbers instead of recognizable abbreviations, and there is no indication of the type of components. Figure 4 shows the initial output of the tree layout of `qgraph.loadings()` before any changes are made, including the code that creates this output.

This graph shows an additional improvement point, which is fixing the overlapping variables. Although `qgraph.loadings()` has several additional parameters to improve these basic circle plots, there are also changes to be made to the input to help `qgraph.loadings()` function better. The parameters help to improve coloring, setting edge width, and adjusting component size, but they do not improve the graphs in such a way that they show the component structure correctly. This requires adaptation of the **RegularizedSCA** output to become the optimal input of `qgraph.loadings()`. Therefore, two new functions are created which directly take

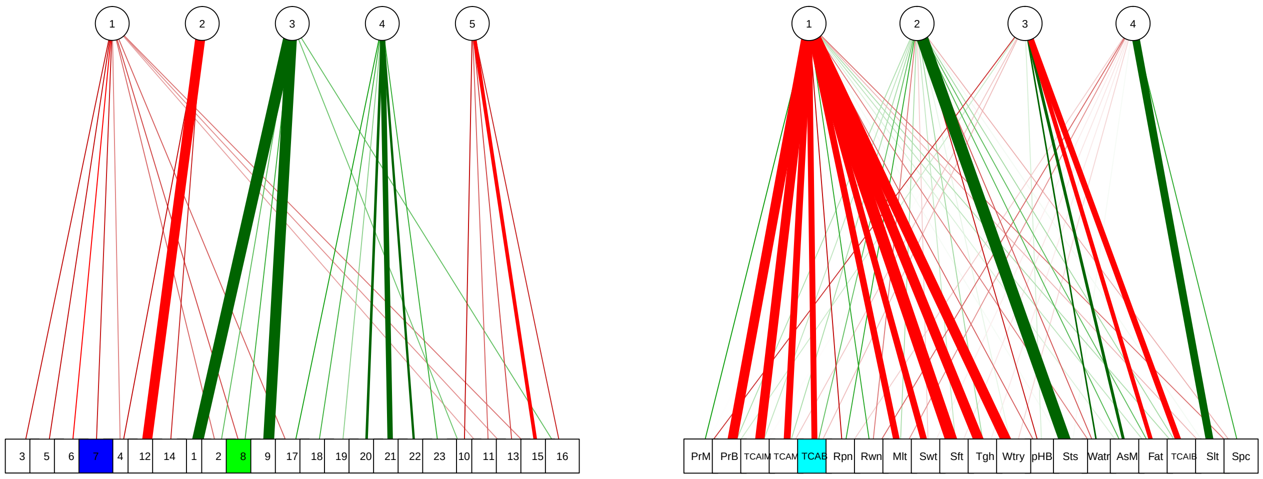
```
qgraph.loadings(data_fam$Pmatrix, groups=num_var_fam, layout="circle")
qgraph.loadings(data_her$Pmatrix, groups=num_var_her, layout="circle")
```



(1) (2)

Figure 3: Initial output of `qgraph.loadings()` for family data (1) and Herring data (2).

```
qgraph.loadings(data_fam$Pmatrix, groups=num_var_fam, layout="tree")
qgraph.loadings(data_her$Pmatrix, groups=num_var_her, layout="tree")
```



(1) (2)

Figure 4: Initial output of `qgraph.loadings()` for family data (1) and Herring data (2) with parameter layout set to “tree”.

the output of **RegularizedSCA** and create new plots with `qgraph.loadings()` as basis. These plotting functions are `plot.circleSCA()` and `plot.treeSCA()`.

The following sections explain how these new functions address adjusting the block names, prepare the data for an optimum input into `qgraph.loadings()`, identify the different component types, adjust the coloring of the graph to pastel colors, and set the circle and tree layout without any overlap. All of these elements are changed by separate supporting functions and come together in the supporting function `create.qgraph()`. The final plot functions, `plot.circleSCA()` and `plot.treeSCA()`, create the `qgraph.object` from all of these supporting functions, add a legend, and plot the final graphs. Thus, the user only needs to call `plot.circleSCA()` or `plot.treeSCA()` with the required parameters to create the visualizations. The final results are shown in Figure 6 of Section 5.8 and Figure 7 of Section 5.9. Sections 5.1 to 5.7 explain how the supporting functions, within `plot.circleSCA()` and `plot.treeSCA()`, change the `qgraph object`.

5.1. Block names

Before any of the models can be run in R, the block names of each of the blocks need to be identified and stored in a separate vector. In case of the family data, the block names are 'Mom', 'Dad', and 'Child', while for the herring data, these are 'ChemPhy' and 'Sensory'. Any name will do, but the user is advised to make them recognisable for easy interpretation and to keep them relatively short in length (e.g., 'Block 1', 'Block 2', etc.). This is the only vector the user needs to declare, before running any of the new plot functions, in addition to the code mentioned in Section 4 to generate the **RegularizedSCA** output.

5.2. Data preparation

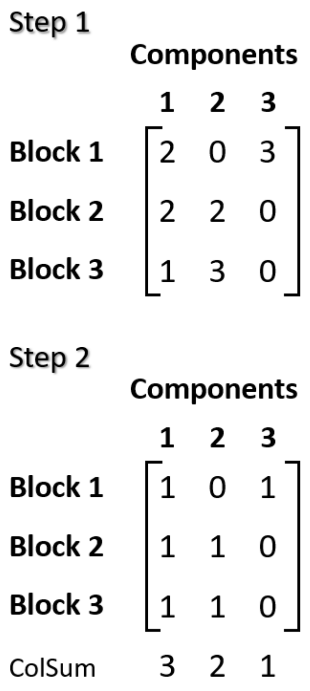
One of the major improvements to the plots in Figure 3 is the distinction between the different blocks amongst the variables. The `qgraph.loadings()` is able to make this distinction if the correct blocks are assigned to the groups parameter. In Figure 3 the `num_var` is assigned to this parameter. However, this is a vector containing the number of variables in each block. In order to assign the variables to the correct blocks, `qgraph.loadings()` needs a list of vectors per block. Each vector contains the variables that belongs to that block. The supporting function `prep.data()` prepares the raw output from **RegularizedSCA**. It extracts the `P-matrix` from the **RegularizedSCA** output and uses `num_var` and `blocknames` to create the list of vectors that is needed as input in `qgraph.loadings()`. In addition, the `prep.data()` names the different components, which are the columns in the `P-matrix` with increasing numbers, starting at number 1.

Finally, the variable names in both the family and herring data set are too large to display in the graph, therefore they need to be abbreviated to three characters. This is also clearly visible in Figure 4, which shows the raw tree output from **qgraph**. The default option to create the abbreviations in `prep.data()` is to use the first letters of the main words of the variable. The main words are all words of three or more characters. However, within each block the abbreviations need to be unique, which means that in some occasions the last letters of the main words are used. For instance the variable `M: Activities with children` is abbreviated, using the first letter of each main word, to `Awc`, while `M: Argue with children` is abbreviated, using the last letter of each main word, to `ehn`, as taking the first letters would also result in `Awc`.

The `prep.data()` function returns a list of the extracted `P-matrix` (`data_p`), `blocks`, and `nodelabels`. The last variable is a vector containing the abbreviated variable names followed by the created component names. The complete code is listed as a general supporting function in Appendix A.1, as it is used to prepare the data in all of the plotting functions.

5.3. Identify components

As explained in Section 2.2, **RegularizedSCA** reveals a certain component structure within the data in which three different component types are identified; global, local, and distinctive components. The second general supporting function `create.components()` extracts this structure from the prepared matrix `data_p`, created `blocks`, and declared `blocknames`. The complete code is listed in Appendix A.2. This function creates a small empty matrix with the number of components as columns and the number of blocks as rows. It then checks for each variable within each block within a component of the `P-matrix`, if there are only zero loadings or if there is at least one non-zero loading. The result of this check is shown in Figure 5, which is based on the same component structure from Figure 1. The matrix shows for each block within a component the number of variables that have a non-zero loading (indicated in Figure 1 by an X). In the next step, the numbers larger than or equal to 1 are changed to a 1, while the zeros remain 0. Now, the sum of the columns starts to show the component structure. One additional logical reasoning step needs to be made in order to correctly identify the types. If `colSum` is equal to the number of blocks, the component is identified as global, as it has non-zero loadings occurring in each block. If `colSum` is between 2 and the number of blocks - 1, the component is identified as local. Lastly, if `colSum` is equal to 1, the component is identified as distinctive. Note that in case there are only two blocks, as with the herring data, it is not possible to have a local component. Finally, `create.components()` creates a vector for the component colors, based on the type. This ensures the use of the same colors throughout the plotting functions. `create.components()` returns a list of the component type vector and the component color vector.



$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline \text{Block 1} & 2 & 0 & 3 \\ \text{Block 2} & 2 & 2 & 0 \\ \text{Block 3} & 1 & 3 & 0 \end{array}$$

$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline \text{Block 1} & 1 & 0 & 1 \\ \text{Block 2} & 1 & 1 & 0 \\ \text{Block 3} & 1 & 1 & 0 \\ \text{ColSum} & 3 & 2 & 1 \end{array}$$

Figure 5: Steps and calculations in determining component type based on the component structure from Figure 1.

5.4. Adjust coloring

The raw **qgraph** plots in Figures 3 and 4 show coloring of the edges and some coloring of the variables. The reason why only some variables are colored is because the groups are not properly identified with the raw output from **RegularizedSCA**, which is corrected with `prep.data()`. Nevertheless, the colors are very bright and make the variable names difficult to read, especially in case of the dark blue color. To adjust the readability and improve the overall look of the graph, all colors are transformed with `color.pastel()`. It uses the basic graphics functions from R to extract the 'red, green, blue' from a color and convert them to a lighter shade, and thus creates a pastel version of the color. As this pastel color conversion is also used in the loadings graph, to ensure similarity in colors between the different blocks across different graphs, it is listed as a general supporting function in Appendix A.3. Finally, to ensure the correct coloring of the edges, `color.edges()` creates a vector of edge colors, in which edges with a positive variable loading are green and edges with a negative variable loading are red.

5.5. Create layout

The intention of the model plot is to show the overall structure of the variables, components, and blocks. There are two versions of this overview, a circle model plot and a tree model plot. Even though, `qgraph.loadings()` already has a setting for creating a circle and tree model, two additional supporting functions are created to have full control over the circle and tree layouts. These functions are listed in Appendix B.2 (`layout.circle()`) and Appendix B.3 (`layout.tree()`). Figure 4 shows that the original `qgraph.loadings()` function does not distribute the variables evenly across the graph and creates a different box size for each of the variables. Nevertheless, both layout functions are similar to the code developed in **qgraph** (Epskamp *et al.* 2012). They are recreated as an additional function, because of the follow-up steps that need to take place in the supporting function `create.qgraph()` to reduce the overlapping variables within the tree layout. The `layout.tree()` function creates a matrix with two columns and the same number of rows as the combined number of rows and columns in the `P-matrix`. In the second column it adds the y-coordinates, which is -1 for the variables and 1 for the components. The first column is intended for the x-coordinates, which are based on an equally spaced sequence from -1 to 1 and with length equal to the number of variables. The x-coordinates for the components is also based on a sequence from -1 to 1, but for a length of the amount of components + 2. The first and last coordinate are discarded, which ensures a nice centered position for the components. The `layout.circle()` function creates a similar matrix, although the calculations are based on an inner circle for the components and an outer circle for the variables. Again, the x- and y-coordinates are between 1 and -1. Both functions return the matrix, which is further processed in `create.qgraph()`.

5.6. Adjust qgraph object

Previous functions all made small adjustments to the **RegularizedSCA** output. It is now time to incorporate them all to create a new `qgraph object`, containing all changes. Appendix B.4 lists the full version of `create.qgraph()`. This section explains how each of the previous supporting functions is used to make the changes to the `qgraph object` within the supporting `create.qgraph()` function.

First, the lists from `prep.data()` and `create.components` are created and all of the individual elements are extracted. Next, the `qgraph object` is created with the extracted `P-matrix` and the created `blocks` from `prep.data()` and the user assigned `blocknames`.

Now, the colors that are attributed to the variables are changed to their pastel version with `color.pastel()` and attributed to the correct position in the `qgraph object`. The component colors are assigned to the `qgraph object` separately, as these are identified with `create.components()`. In addition, the size of the components is altered to depend on the amount of variance the component accounts for, using the variance accounted for

(VAF) method (Gu and Van Deun 2019). This method is already incorporated in **RegularizedSCA**, and is invoked with the following function: `VAF(DATA=orig_data, Jk=num_var, R=nr_comps)`. The VAF function returns the amount of variance explained per component per block. Per component the variance for all blocks is summed up to retrieve the total amount of variance explained per component. The `orig_data` refers to the data before analysis, which are `fam_data` and `herring_data` in the examples. Up until this point, both plotting functions only required the **RegularizedSCA** output as parameter input. To use the VAF-method as parameter for component size, the original data (`fam_data` and `herring_data`) now also needs to be entered as input.

In addition, the original graphs from Figures 3 and 4 show different widths for each of the edges, which represent the loadings of a variable within a component. As these plots are intended to form an overview, the width is set equal for each of the variables. Moreover, when the graphs become denser, the difference in widths make the graph less readable. This is already visible when comparing the family and herring model plots (both tree and circle). The structure of the entire system becomes less visible, as the attention is drawn towards the heavy edges and taken away from the overall view. Nevertheless, in some cases, showing the width is relevant. Therefore, there is an option embedded to use the loadings as weight of the edges by setting the parameter `width=TRUE`. The default position of this parameter is set to `width=FALSE`, which means it does not need to be declared in case equal widths is indeed the preferred option. The green and red coloring is ensured by using `color.edges()` and assigning the resulting vector to the `qgraph object`.

Moreover, the layout that is created with either `layout.circle()` or `layout.tree()` is assigned to the `qgraph object` along with several additional steps to change the sizing of the variables. Especially in Figure 4, there is some overlap in the variable boxes that needs to be prevented. Therefore, the height and width of each of the variables is set to divide the total length and width of the figure evenly. This ensures that the size of the variable is adjusted according to the number of variables that need to be shown. The larger the data set, the smaller the variables become. This means there is a limit to the number of variables that can be shown and when the data set grows too large, this graph option might become less useful.

Finally, the new labels that are created with `prep.data()` are assigned to the `qgraph object` and the label size proportion is set to 1. Using the proportion function in the `qgraph object` ensures that the label size increases and decreases with changes made in the size of the variables.

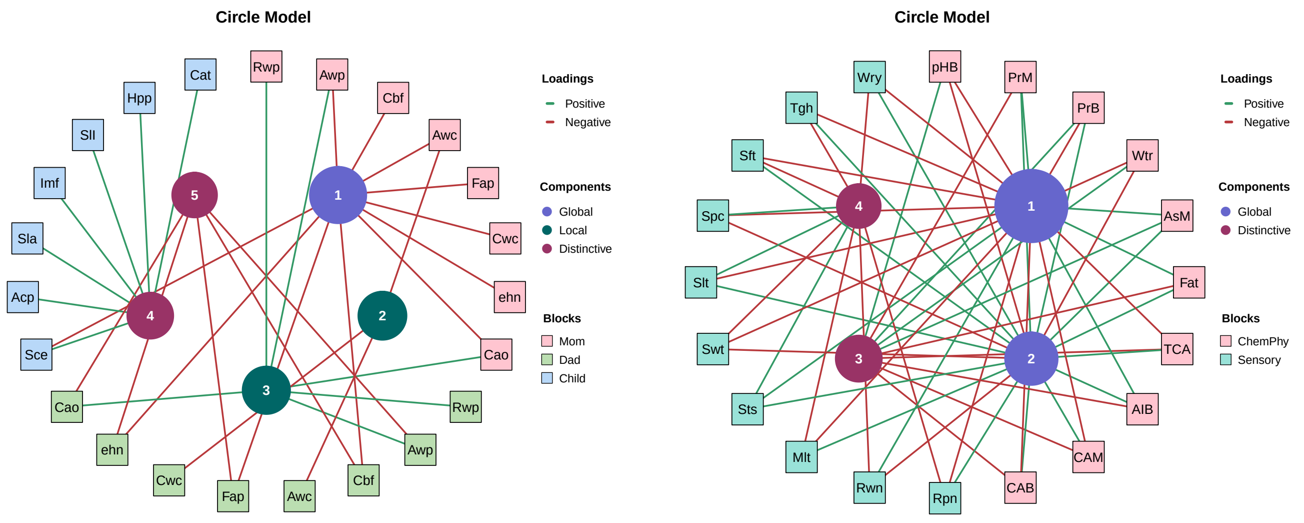
5.7. Create legend

To make the graphs complete, there is an option to add a legend. The default option is set to showing a legend, however in some cases a legend is not preferred. This will become clear in Section 8. The legend function `create.legend()` is listed in Appendix B.5 and creates three separate legends. These legends explain the details and coloring of the loadings, components, and blocks. All legends are always placed on the right side of the plot. The loadings' legend is static, as each graph contains positive and negative loadings. The legends of the components and blocks change depending on the data. The components' legend shows the coloring for the different types of components. Only those components that occur in the plot are mentioned. The same principle holds for the blocks. The function returns a list of the different legend objects, which is used when calling the final plot functions.

5.8. Final Circle model

The newly developed `plot.circleSCA()` function incorporates all previously described supporting functions and creates a dynamic overview of the data in a circular model. The circle model requires the matrix created from the original data set as input (`fam_data` and `herring_data`), as well as the matrix created from the **RegularizedSCA** results (`data_fam` and `data_her`), the amount of variables in each block (`num_var`), and the vector with block names. There are three additional parameters in which the user can adjust the title of the graph, choose to switch off the legend (add `legend=FALSE`), and choose to set the width of the edges to the weight of the loadings. In the default code, as stated below, the parameter `title` is set to either "Circle Model" or "Tree Model" by default. The parameter `legend` is set to `TRUE` by default and the parameter `width` is set to `FALSE` by default too. As these parameters all have a default value, they do not need to be declared separately in the code below. Figure 6 shows the result of the code, while Appendix E shows the result of the graphs when `width=TRUE`.

```
block_names_fam <- cbind("Mom", "Dad", "Child")
block_names_her <- cbind("ChemPhy", "Sensory")
plot.circleSCA(fam_data, data_fam, num_var_fam, block_names_fam)
plot.circleSCA(herring_data, data_her, num_var_her, block_names_her)
```



(1) (2)

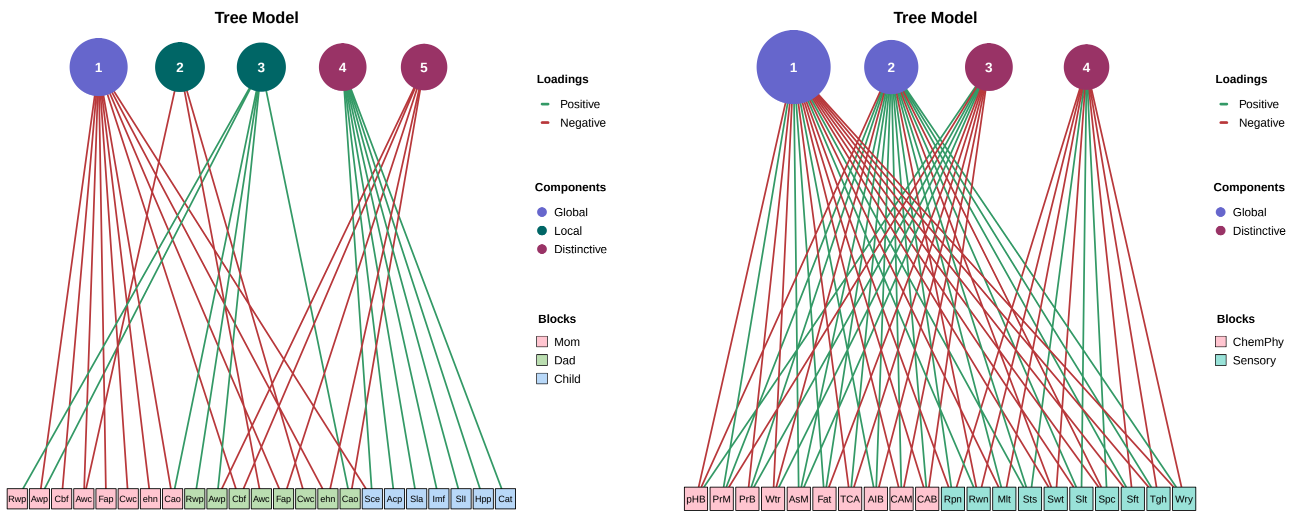
Figure 6: Circle model of family data (1) and herring data (2).

Figure 6 shows a clear improvement over Figure 4. There is a clear distinction between the different blocks, the variable names are readable and easily traceable to their original name, and the components types are distinguishable. In addition, the amount of variance accounted for by the components is now clearly visible as well. Moreover, the amount of variance accounted for by the components can also be compared across data sets. The first component of the herring circle model plot is by far the biggest component, which means it explains the most variance. The first component from the family data set is only slightly larger than the rest of the components and definitely smaller than the first component of the herring data set. In this case comparing the two components from different data sets cannot lead to any conclusion, but there might be future cases in which this could be relevant and valuable.

5.9. Final Tree model

The newly developed `plot.treeSCA()` function provides the user with a traditional, tree graph-style model and incorporates all previously described supporting functions. The tree model plot requires the exact same input as `plot.circleSCA()`, which is the matrix derived from the original data set (`fam_data` and `herring_data`), the results from **RegularizedSCA** (`data_fam` and `data_her`), the number of variables in each block (`num_var`), and the vector containing the block names. The parameters `title`, `legend`, and `width` are all set to their default settings. Figure 7 shows the result of the code, while Appendix E shows the result of the graphs when `width=TRUE`.

```
block_names_fam <- cbind("Mom", "Dad", "Child")
block_names_her <- cbind("ChemPhy", "Sensory")
plot.treeSCA(fam_data, data_fam, num_var_fam, block_names_fam)
plot.treeSCA(herring_data, data_her, num_var_her, block_names_her)
```



(1) (2)

Figure 7: Tree model of family data (1) and herring data (2).

There is a clear difference between the plots in Figure 7 and 4. The variables are no longer overlapping, the different blocks are clearly indicated,the components have a prominent position, and the component type is clearly indicated. Both the circle model plots and tree model plots have an overall improved look, as they have less bright colors, which are still very distinguishable. The spacing of the variables is improved, and in general more information is added, which can be extended by changing the parameter `width=TRUE` when required.

5.10. Component highlights

In addition to the two new plotting functions, `plot.treeSCA()` and `plot.circleSCA()`, two variations on `plot.circleSCA()` are created for additional insight in the data structure. These functions either highlight each component one by one, `plot.comp.indiv()`, or highlight all types of components type by type, `plot.comp.type()`. These two new functions each require an additional supporting function and have somewhat more elaborated plotting functions. The plotting functions are listed in Appendix B.10 for `plot.comp.indiv()` and B.11 for `plot.comp.type()`. Note that the code is created in a such a way that the user only needs to call `plot.comp.indiv()` and `plot.comp.type()` to create the visualizations. All supporting functions are incorporated in these plotting functions. The following sections explain how these functions work.

Highlight per component

The function `plot.comp.indiv()` highlights a specific component. It therefore requires an additional parameter named `number` to identify which component needs to be highlighted. Note that `prep.data()` transforms the names of all components to increasing numbers starting at 1. In addition, the `number` parameter can also take the string `"all"` as parameter, in which case all of the components are highlighted one by one and are plotted in a two-column frame. Figure 8 show the first two components of the family and herring data. The additional components are plotted in Appendix F, which also includes the graphs where `number="all"`.

```
plot.comp.indiv(fam_data, data_fam, num_var_fam, block_names_fam, 1)
plot.comp.indiv(herring_data, data_her, num_var_her, block_names_her, 1)
```

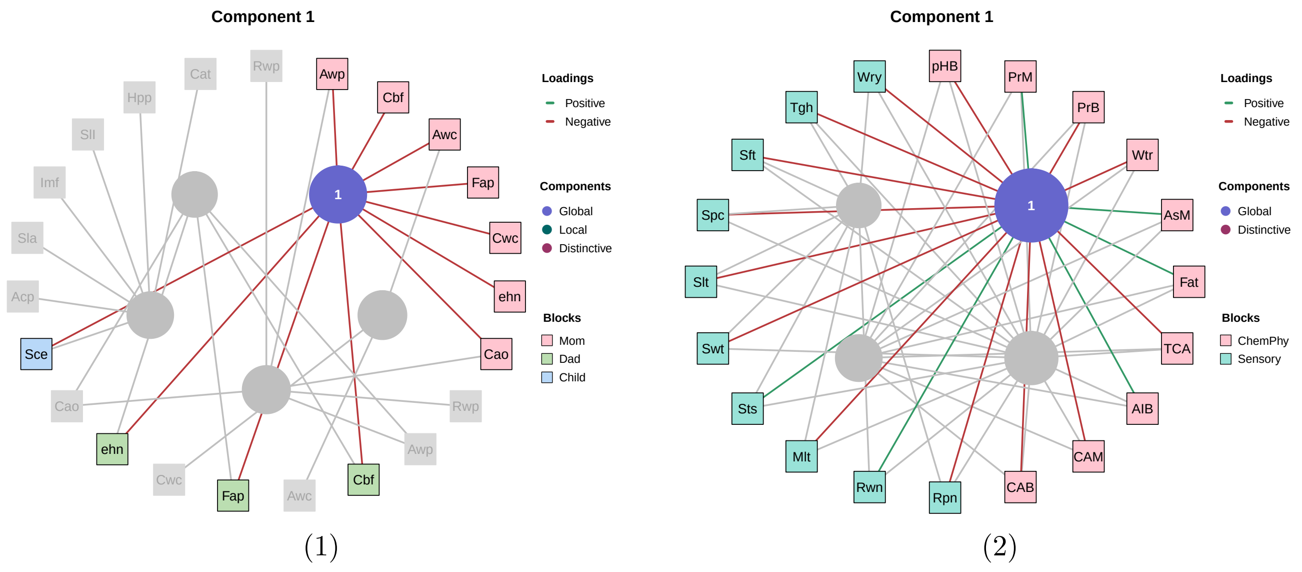


Figure 8: Circle model plot with highlighted first component of family data (1) and herring data (2).

The three additional parameters for `title`, `legend`, and `width` can also be used in this function. The default option for `title` in this function is the component name that is highlighted.

To highlight the specific component, the additional supporting function `grey.indiv()` is created, which is listed in Appendix B.6. This function requires the `qgraph object`, the list of component types (`comp_type`), and the number of the component that needs to be highlighted as input. To highlight this specific component, the remaining components, the edges to those components, and the variables that are not part of the highlighted component all need to be colored grey. All of the colors are stored in lists within the `qgraph object`. Therefore, for each of the remaining components, unused edges, and unused variables, `grey.indiv()` first finds the position of the elements that belong to the highlighted component and colors all other elements grey and replaces the old list in the `qgraph object` with the new list of colors.

Within the plotting function `plot.comp.indiv()`, two cases are distinguished after the data is processed with `create.qgraph()`. In case `number="all"` all of the components are highlighted one by one in a two-column frame. The number of rows is based upon the number of components that need to be visualized, where the number of components is divided by two and the result is rounded up. In case a specific number is entered in `number`, only one graph is generated where that specific component is highlighted.

Highlight per component type

The second highlighting function highlights components of a specific type. It therefore requires the additional parameter `type` to indicate which type of components need to be highlighted. Note that not all data sets obtain all three different types of components. Similar to `plot.comp.indiv()`, `plot.comp.type()` can also take `"all"` as input for parameter `type`. A difference is that the highlighted plots are always visualized in a single row, because the maximum number of columns it can assume is only three. Figure 9 shows all of the highlights for the family and herring data set. Each visualization is individually created by calling each type of component separately as `type`, like in the code stated below. A similar result is achieved when setting `type="all"`. These visualizations are shown in Appendix F.

```
plot.comp.type(fam_data, data_fam, num_var_fam, block_names_fam, "global")
plot.comp.type(fam_data, data_fam, num_var_fam, block_names_fam, "local")
plot.comp.type(fam_data, data_fam, num_var_fam, block_names_fam, "distinctive")
plot.comp.type(herring_data, data_her, num_var_her, block_names_her, "global")
plot.comp.type(herring_data, data_her, num_var_her, block_names_her, "local")
```

As with `plot.comp.indiv()`, the three additional parameters for `title`, `legend`, and `width` can also be used in this function. The default option for `title` in this function is the component type that is highlighted. Using the help of supporting function `grey.type()`, which is listed in Appendix B.7, the unused edges, components, and variables are colored grey according to a similar principle as in `grey.indiv()`. The main difference is an additional line of code that converts the type of components to the component numbers. For each of the component numbers the same process as in `grey.indiv()` needs to be repeated. The function makes use of the fact that the same component types always succeed each other. Finally, the input for the `type` parameter is not case sensitive, all input is converted to lower case upon input. In case the default title is used, the first letter is restored to an uppercase.

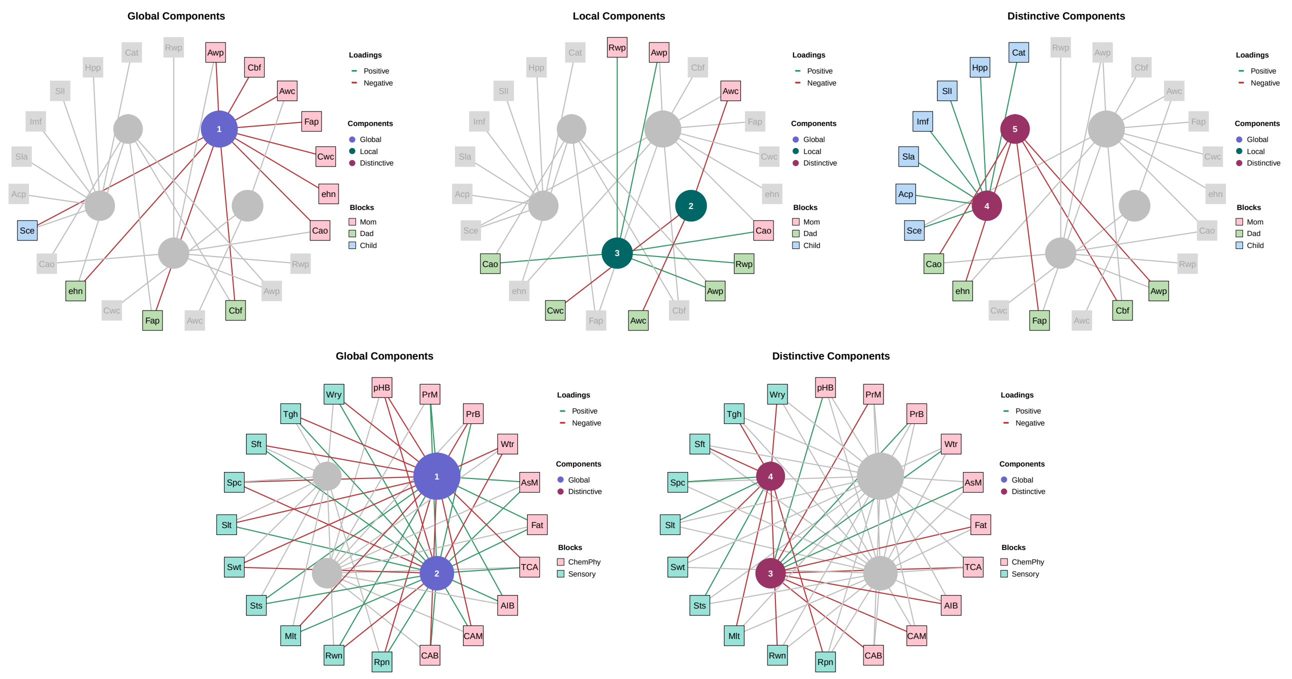


Figure 9: Circle model plot where each type of component is highlighted for family data (row 1) and herring data (row 2).

6. Visualizing component loadings

In the data model visualizations, the default option does not show the size of the loadings as weight of the edges, because it potentially makes these plots too complex. Therefore, two separate but similar functions are created to have a closer look at the component loadings specifically; `plot.loadings()` (Appendix C.2) and `plot.loadings2()` (Appendix C.3). They are two versions of the same bar plot that show the value of the loadings for each block and per component. These functions both use an additional supporting function that wraps up the variable labels. Like previous plotting functions, they incorporate all necessary supporting functions. Thus, the user only needs to call `plot.loadings()` and `plot.loadings2()` with their preferred parameters to create the visualizations.

6.1. Wrap labels

The variable labels from the family data set, are almost complete sentences and are therefore too long to show in any graph. In the model visualizations, the variables are abbreviated because of the limited space to place the variable names. In the bar plots, however, there is more space available, although this depends on the number of variables that need to be shown. The wrapping function `wraplabels()` is based on a development of Schwartz (2013) and altered to fit the needs of this visualization (Appendix C.1). The wrapping function takes the vector containing the original variable names and the desired cut-off length as input. It removes the one- and two-letter words and then places an enter at the cut-off point. It returns the vector of adjusted variable names. In both loading plots the cut-off point is set at ten characters. Figures 10 and 11 show the results of this wrapping function.

6.2. Two plot functions

Both loadings plots require the output created by **RegularizedSCA** (`data_fam` and `data_her`), the number of variables in each block (`num_var`), the vector with block names and an additional parameter `nr`, that indicates which component is visualized. Both functions also prepare the data with `prep.data()` and create the components with `create.components()`. Next, the variable names are extracted from the data, stripped of punctuation and words consisting out of one or two letters, and processed with `wraplabels()`. The difference between the two loading plots is made at the next step. Where `plot.loadings()` will always generate plots for all blocks, even if they are empty, `plot.loadings2()` will only generate a plot for the contributing blocks. In both plots, the colors to distinguish the different blocks are the same as the colors that are used to distinguish the blocks in the model visualizations. All bar plots within one visualization have the same minimum and maximum value on the x-axis to optimize readability.

In `plot.loadings()` data per block is extracted from the `P-matrix` and plotted in a horizontal bar plot. Each individual bar plot receives a title with the name of that block. The set of bar plots receives a title that contains the component number and the component type. These are static and can not be changed by the user. Figure 10 shows the visualizations created with the following code:

```
for (i in ncol(data_fam$Pmatrix){
    plot.loadings(data_fam, num_var_fam, block_names_fam, i)
    }

for (i in ncol(data_her$Pmatrix){
    plot.loadings(data_her, num_var_her, block_names_her, i)
    }
```

The for-loop can also be replaced by repeating the plot function for each component number. By placing the graphs underneath each other, one can clearly see how each block contributes to each component. The blocks that do not contribute have empty bar plots.

In `plot.loadings2()` only the blocks that contribute to the component are visualized. Therefore, the blocks that do not contribute are removed from the data first. The color that belongs to the removed block is removed from the graph colors as well. The remaining process is similar to that of `plot.loadings()`. Figure 11 shows component 3 of both the family and herring data. Note that the block “Child” and “Sensory” are now missing from the bar plots. Appendix G shows all of the `plot.loadings2()` plots that can be generated for the family and herring data.

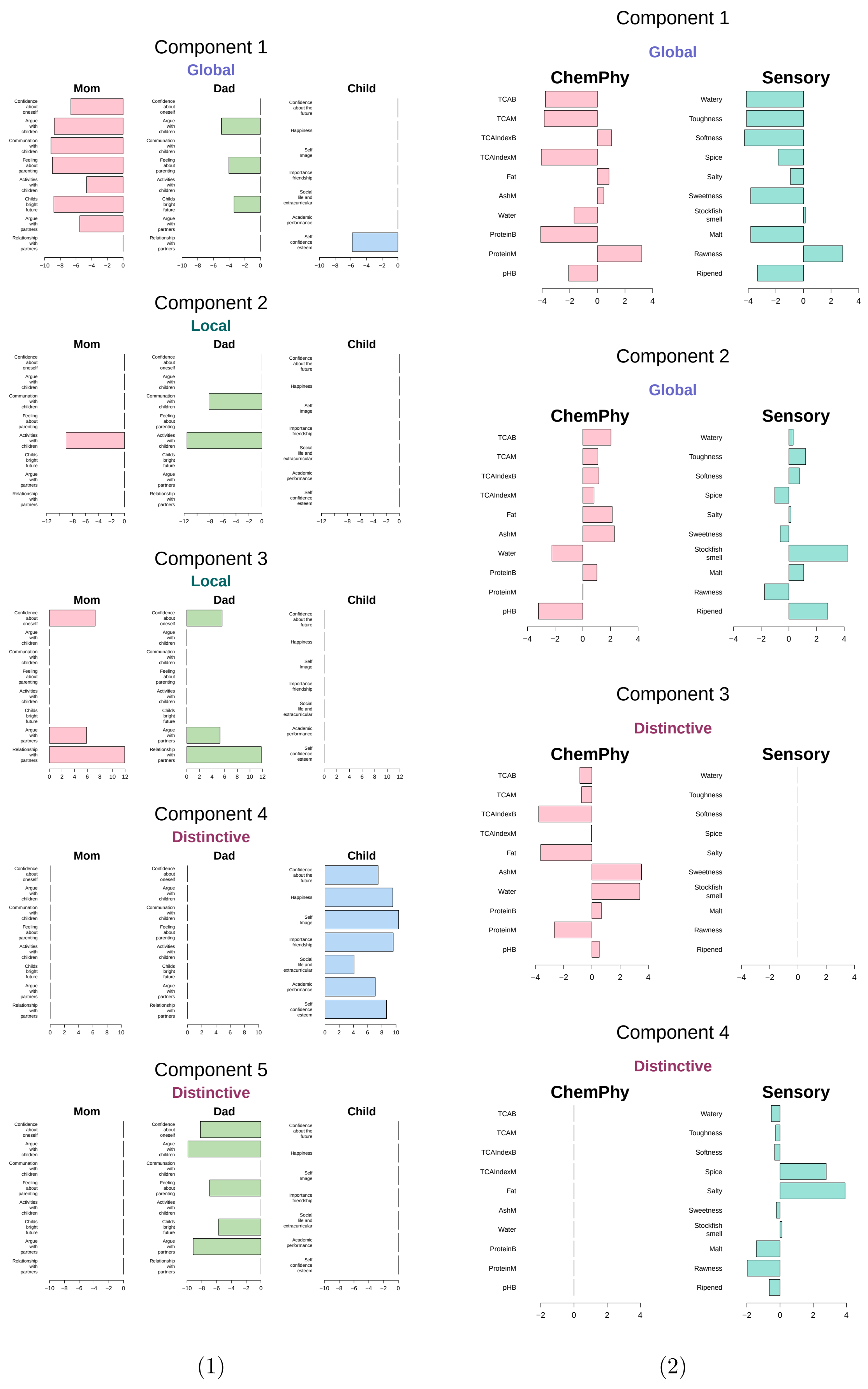
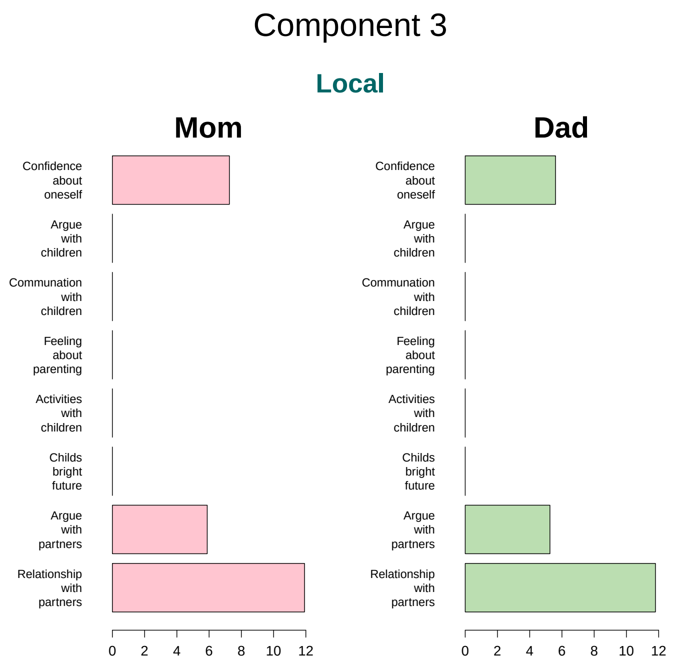
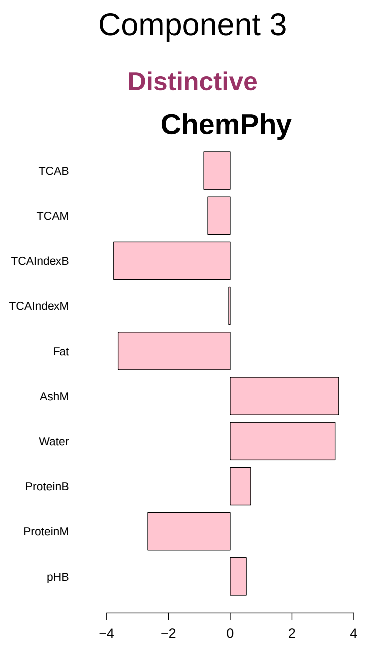


Figure 10: Component loadings visualization using `plot.loadings()` of family data (1) and herring data (2) plotted by increasing component number for all available components.

```
plot.loadings2(data_fam, num_var_fam, block_names_fam, 3)
plot.loadings2(data_her, num_var_her, block_names_her, 3)
```



(1)



(2)

Figure 11: Component loadings visualization using `plot.loadings2()` for component 3 of family data (1) and herring data (2).

7. Visualizing individual cases

The final type of visualization focuses on the individual cases within the data set, such as families and herrings in the family data and herring data. This requires a different part of the **RegularizedSCA** results, which is the `T-matrix`. This matrix contains the scores of the individual cases on each of the components; the component scores. These component scores are often used to cluster the individual cases according to a specific factor variable such as gender, role, diet, etc. As these are missing in the family and herring data sets, the first visualization option uses the different components types (global, local, or distinctive) to try to identify patterns in the data. This visualization is listed in Appendix D.1 and is named `plot.individuals()`. A simple `k-means` clustering algorithm can add clusters in the visualization to identify more patterns. The second visualization option creates pairwise plots per component number and is listed in Appendix D.2 with the name `plot.pairwise()`.

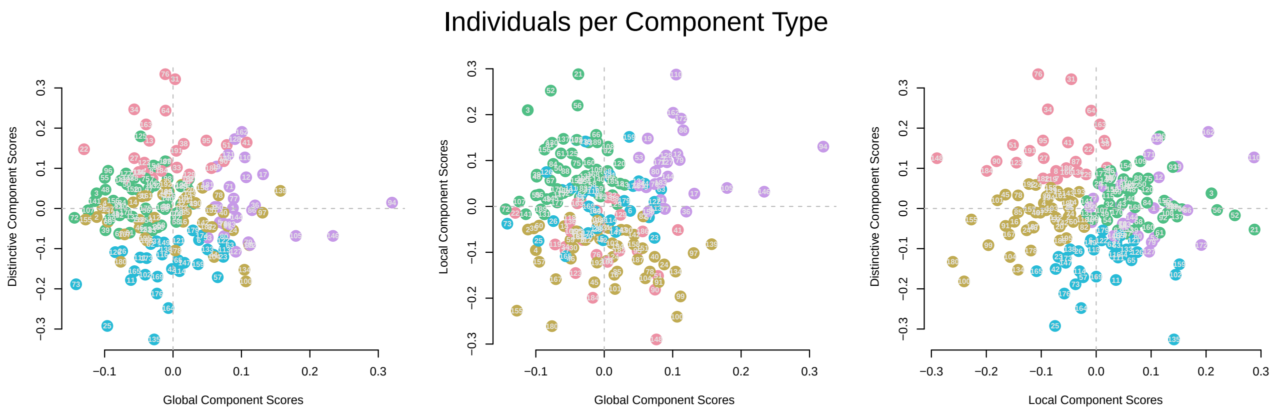
7.1. Individual cases per component type

The function `plot.individuals()` combines the component scores for each of the different types of components and creates a pairwise plot. It requires the output from **RegularizedSCA** (`data_fam` and `data_her`), the number of variables in each block (`num_var`), and the vector with block names as input. There are two additional optional parameters, which are `title` to adjust the title and `clust` to adjust the clustering pattern. The function starts by preparing the data with `prep.data()` and creating the components with `create.components()`. Next, it extracts the `T-matrix` from the processed data and sums up the component scores per individual case per type of component to a new, separate data frame. By summing up the component scores, the visualizations can identify if any of the different types of components is dominant. Since the scores can either be positive, negative, or zero, adding scores from multiple components (of the same type) can cause the scores to cancel each other out towards zero (when an individual cases scores negatively on one component and positive on another component of the same type), or it can enhance them (when the scores have the same sign), showing dominance. Next, the function determines whether the default option for `clust=NULL` is used, or if a different clustering vector is added. In case of the default option, a simple `k-means` algorithm determines the clustering colors for each of the individual cases based on all of the components. To determine the optimum number of clusters (`k`), `fviz_nbclust()` from **factoextra** (Kassambara and Mundt 2020) is used. The maximum number of clusters is set to six to make sure that the clusters are still readable from the graph. In case `clust` receives a vector of factors, these are used to color the individual cases. Now, note that it is possible to have one, two, or three different types of components. In case there is only one type of component, this has to be a global component. In this case the individual cases are plotted with the sum of the component scores on the y-axis and the order of appearance of the individual cases in the data set are taken as x-coordinates. When there are two types of components, these have to be global and distinctive, as is the case in the herring data set. Then the sum of global component scores is plotted on the y-axis and the sum of the distinctive component scores is plotted on the x-axis. Lastly, when there are three types of components, as is the case in the family data set, three plots are generated; one for each combination of component types. Finally, if `clust` is chosen by the user, a legend is also added to the plot that explains the cluster coloring. In addition, the user may choose a new title for the graph with `title`. The default title is set to “Individuals per Component Type”. The following code shows the default versions of `plot.individuals()` for family and herring data. The generated visualizations are shown in Figure 12.

7.2. Pairwise per component

The final visualization creates a pairwise comparison of the individual cases across the different components with `plot.pairwise()`. It requires the matrix derived from the original data set (`fam_data` and `herring_data`), the results from **RegularizedSCA** (`data_fam` and `data_her`), the number of variables in each block (`num_var`), and the vector containing the block names. In addition, it also has the two additional parameters `title` and `clust`, which function exactly the same as in `plot.individuals()`. Similar to `plot.individuals()`, `plot.pairwise()` prepares the data with `prep.data`, creates the components with `create.components()` and extracts the `T-matrix` from the processed data. In the default option of `clust=NULL`, **factoextra** (Kassambara and Mundt 2020) determines the optimum number of clusters (`k`) and a simple `k-means` algorithm clusters all of the variables. Using the standard `pairs` function the pairwise plots are created for each combination of components. Only the upper triangle is shown for optimum reading quality. In the diagonal squares, the name of the component, type of component, and the amount of variance it explains is stated. The latter is determined

```
plot.individuals(data_fam, num_var_fam, block_names_fam)
```



```
plot.individuals(data_her, num_var_her, block_names_her)
```

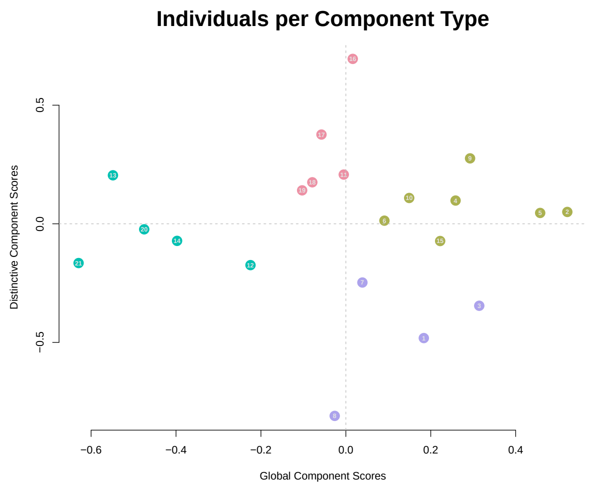


Figure 12: Individual cases visualization using `plot.individuals()` for family data (1st row) and herring data (2nd row)

by the VAF method from Gu and Van Deun (2019) and requires the matrix derived from the original data set, as this is the required input for the VAF-method. Figure 13 shows the result for the family data, which is created using the following code. The pairwise plot for herring data can be found in Appendix H.

```
plot.pairwise(fam_data, data_fam, num_var_fam, block_names_fam)
```

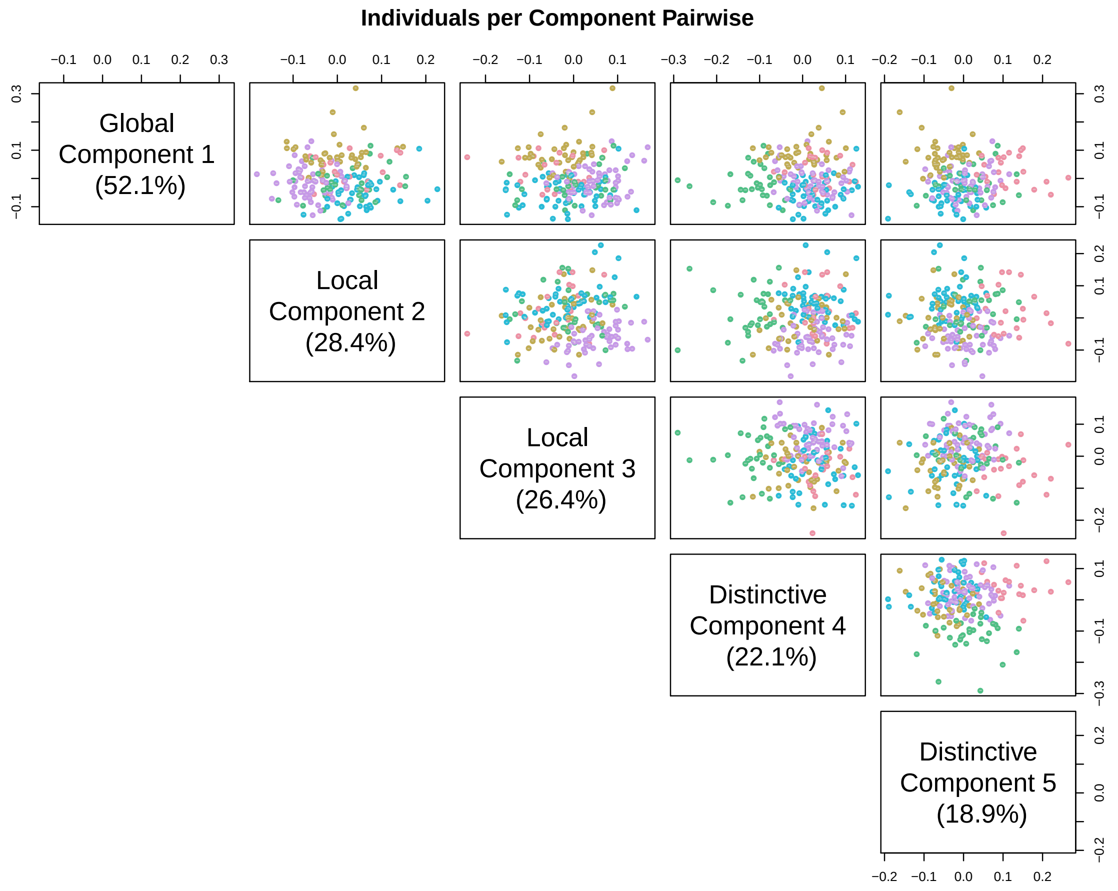


Figure 13: Pairwise individual cases plot per component using `plot.pairwise()` with default options for family data.

8. Application examples

The data sets, family and herring, that are used throughout this paper have shown what the newly developed visualizations look like on a relatively small data set. One of the goals of this paper is to develop visualizations that can be used on large data sets. Therefore, three additional data sets are processed using **RegularizedSCA** to demonstrate how the visualizations scale with larger data sets. The first data set is nutrimouse, which is included in the **MixOmics** package (Rohart, Gautier, Singh, and Lê Cao 2017) and contains information on liver genes that might be involved in the concentration of hepatic fatty acids and the nutritional problems found in mice. This data set contains measurements for 120 genes available from liver cells and a measure of the concentration of 21 fatty acids. These data were collected on a set of 40 mice. In addition, there are two background variables available that contain the mice's genotypes and the type of diet each mouse is on. The second data set, liver toxicity also originates from **MixOmics** (Rohart *et al.* 2017) and contains information on genes and clinical measurements on 64 rats that have been exposed to non-toxic, moderately toxic, or severely toxic doses of acetaminophen. The liver toxicity data set contains information on

3116 genes, 10 clinical measurements, and four treatments. The final data set, BRCA, is derived from the former **r.jive** package that is based on research from the Cancer Genome Atlas Network (2012) and contains the gene expression, DNA methylation, and miRNA expression of 348 tumor samples to determine the origin of breast cancer. There are 654 measurements on genes, 574 entries for DNA methylation and 423 miRNA expressions available. Table 1 shows an overview of the sizes of the data sets. Elements `orig_data` refers to the data before **RegularizedSCA** is applied.

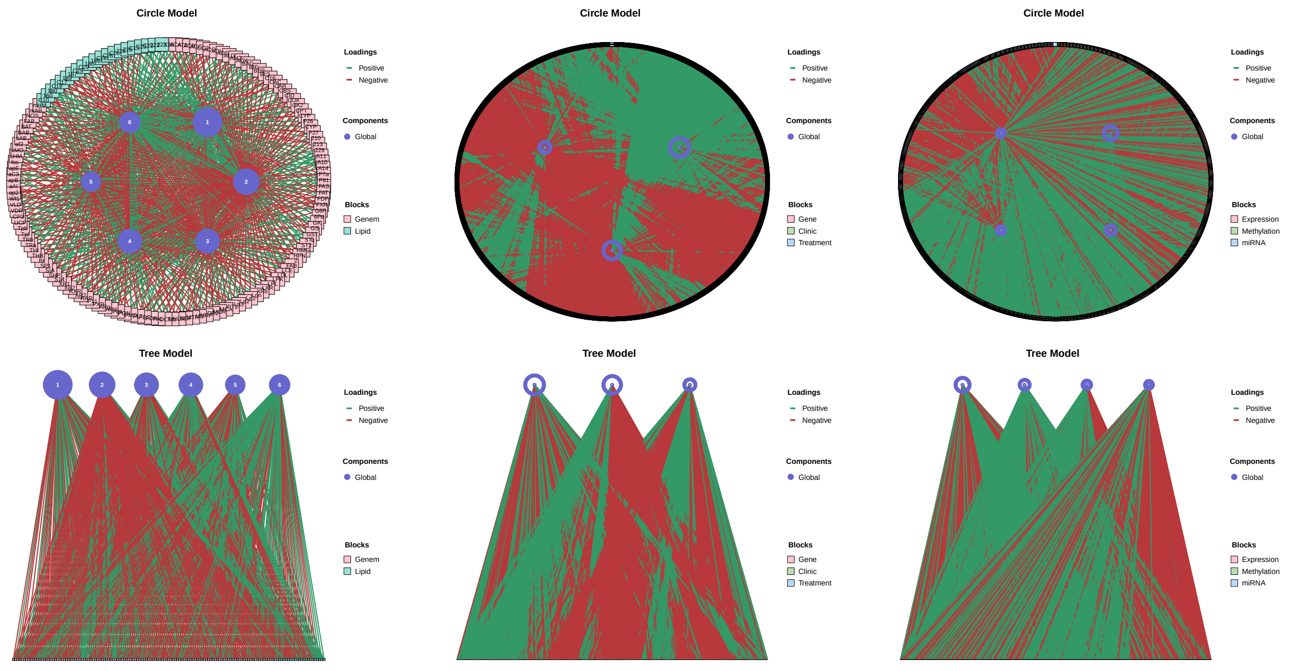
Data set	Variables	Individuals	Blocks	Elements `orig_data`	Components	Elements `P-matrix`	Elements `T-matrix`
Family	23	195	3	4 485	5	115	975
Herring	20	21	2	420	4	80	84
Nutrimouse	141	40	2	5 640	6	846	240
Liver toxicity	3130	64	3	200 320	3	9 390	192
BRCA	1642	348	3	571 416	4	6 568	1 392

Table 1: Short overview of the size of the three additional data sets in comparison to the family and herring data.

Each of the three new data sets is processed with **RegularizedSCA**. As the component structure is unknown in all of the data sets, model 4, as described by Gu and Van Deun (2019), is used. At first the variance is calculated using the VAF method, after which there were six components retained for the nutrimouse data, three components for the liver toxicity data, and four components for the BRCA data. Table 1 shows the implications for the size of the `T-matrix` and `P-matrix` with these numbers of components. Using `cv_sparseSCA` the optimum values for lasso and grouplasso are determined and entered in `sparseSCA`. Finally, `undoShrinkage` is used to reverse the lasso and grouplasso effects and to achieve the final results for each of the data sets. Note that **RegularizedSCA** already reduces the size of the data sets when comparing the elements in `orig_data` to the elements in the `P-matrix` and `T-matrix`.

8.1. Data model visualizations

Each of the data sets is visualized using `plot.circleSCA()` and `plot.treeSCA()` and shown in Figure 14. While the visualizations are aesthetically pleasing, they do not provide much information anymore. The variables from the nutrimouse data set are still somewhat distinguishable, while the variables from the other data sets are simply too small. However, as mentioned in Section 3, all visualizations are initially rendered and saved using `pdf()`. It is also an option to run the code in R and use the zoom function to show a larger view of the visualization. For both the liver toxicity data and BRCA data, this does not improve much, but for the nutrimouse data this does the trick as is shown in Figure 15. It is important in this case to switch off the legend, as its does not properly adjust to the increased plot size. Note that in all of these data sets there are only global components, which makes the highlight option by type not useful. Highlighting per component might be useful only in case of nutrimouse, as is setting `width=TRUE` to identify any variable that might pop out. Unfortunately, Figure 16 shows that this is not the case. Although, some of the negative variables from component two are more highlighted, which means they have a higher contribution to that component.



(1) (2) (3)

Figure 14: Circle model visualizations using `plot.circleSCA()` of nutrimouse data (1), liver toxicity data (2), BRCA data (3).

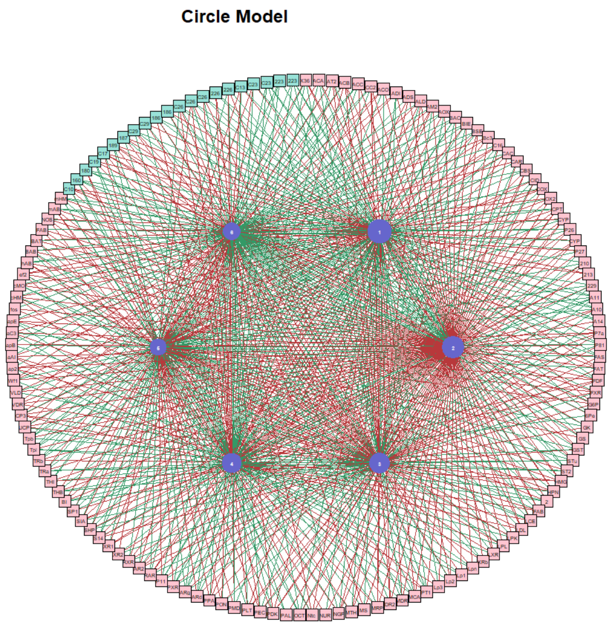
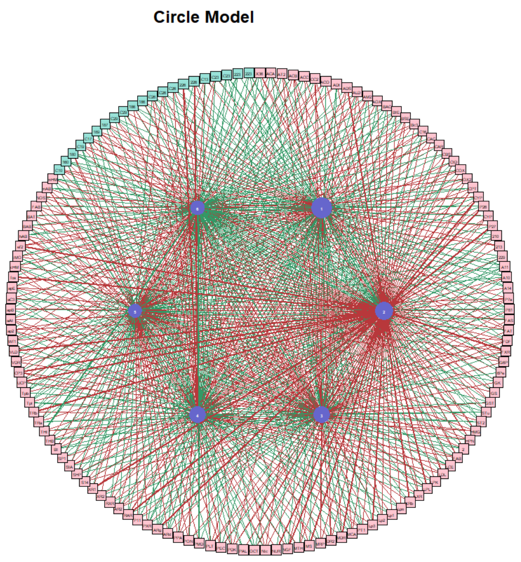


Figure 15: Circle model visualizations using `plot.circleSCA()` of nutrimouse data with `legend=FALSE` and without using `pdf()` to save the plot.



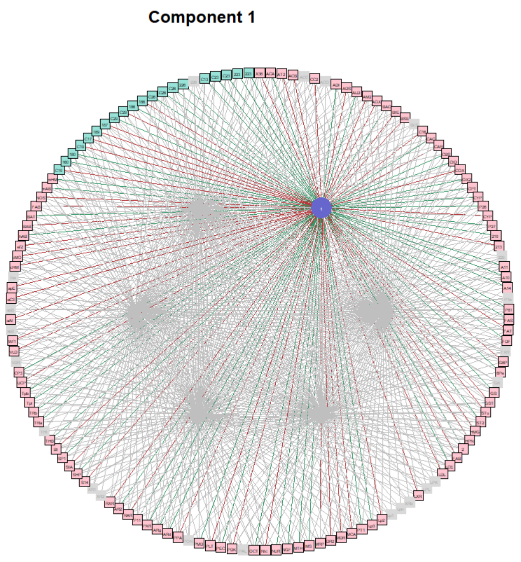


Figure 16: Circle model visualizations using `plot.circleSCA()` of nutrimouse data with `legend=FALSE` and `width=TRUE` (1) and highlighted component with `legend=FALSE` (2), both without using `pdf()` to save the plot.

From a developers point of view there is one more remark to make, which is that the highlighted edges are not on top of the grey edges. This was not as apparent in the family data and herring data visualizations, but becomes more disturbing when the number of variables has increased. It reduces the highlight effect that the visualization is trying to achieve.

8.2. Component loadings visualizations

In this section two different component loadings visualizations are created, differing in whether or not they show the blocks that are not contributing to a component. However, in all three example data sets, there are only global components identified, which means that both functions will yield the same result. Note that there is a large number of variables in the liver toxicity data and BRCA data. Both functions work well with this large number of variables, although the user has to adjust some of the `pdf()` settings to properly show all of the variables and their names. The downside of these adjustments is that the size of the visualization grows very large. Appendix I shows the first component loadings of the liver toxicity data and BRCA data, however the visualizations are too large to properly display on one page. Figure 17 shows the first component of the nutrimouse data. The 21 variables of the fatty acids (lipid) are clearly shown, but the 120 genes are less distinguishable. This is due to the amount of space available on the page, as is the case with the other two data sets. When all three visualizations are opened in the pdf viewer on a computer, all of the genes are visible for inspection.

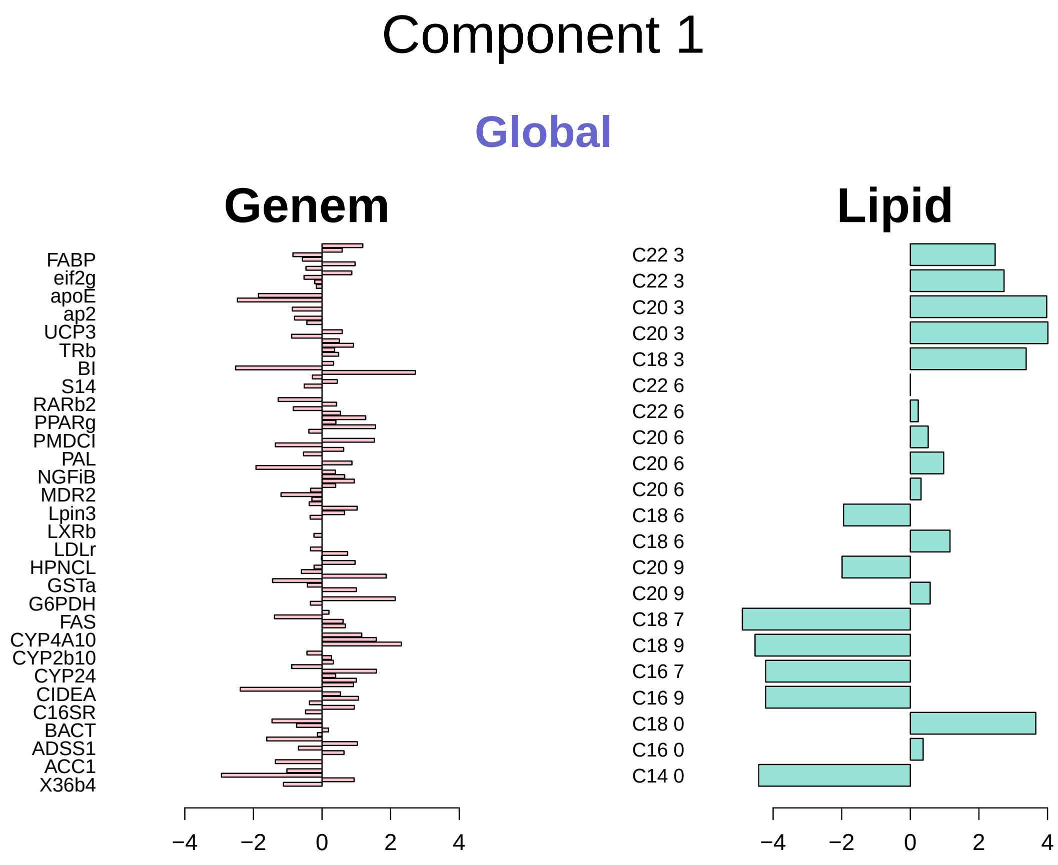
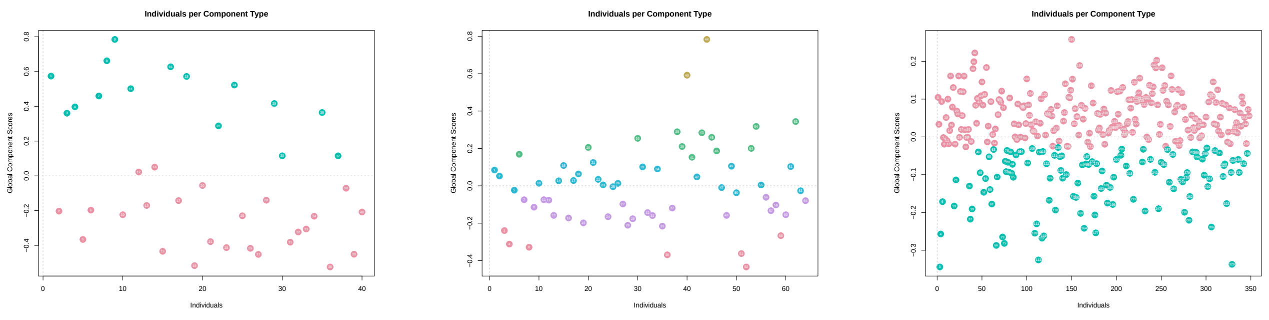


Figure 17: Component loadings visualizations using `plot.loadings()` of nutrimouse data (1), liver toxicity data (2), and BRCA data (3) showing the first component

8.3. Individual cases visualizations

The final visualizations show the individual cases of each of the example data sets. Table 1 states the number of individual cases in each of the data sets. Note that the number of elements in the `T-matrix` is much smaller than the number of elements in the `P-matrix` of each data set and that a large `P-matrix` does not always mean a large `T-matrix`. Since there are only global components in all of the example data sets, `plot.individuals()` will only show one plot per data set and use the ordering of the individual cases as coordinates for the x-axis.

Figure 18 shows the individual cases of each data set with the default clustering algorithm. All three plots clearly identify different clusters, mainly whether individual cases score positively or negatively on the summed global components. Note that the signs of these scores depends on the signs of the component loadings in the `P-matrix` and could also all be reversed as explained in Section 2.3. On the other hand, by combining the components the signs for the variables within the different components could have counteracted each other, leaving an individual component score of close to 0. To draw any conclusions on the clustering pattern,

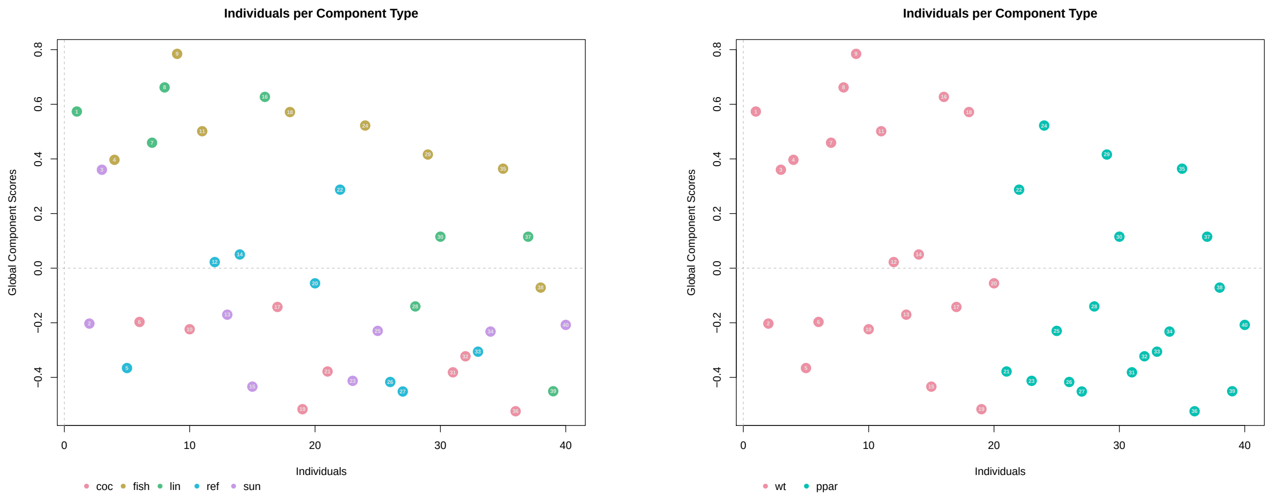


(1) (2) (3)

Figure 18: Individual cases visualizations using `plot.individuals()` for nutrimouse data (1), liver toxicity data (2), and BRCA data (3) with `clust=NULL`.

more insight information on the data sets is needed, which is left to the experts.

The individual cases visualizations are all created while `clust=NULL`. In case of the nutrimouse data, two additional background factors are provided that can be used to cluster the individual cases accordingly. Figure 19 shows the results of the clustering. Note that in these visualizations a legend is added to the graph to explain the clusters. At first sight the genotype clustering seems to make a perfect split, but that is because of the placing of the variables on the x-axis. The first variables all belong to genotype `wt` and the second part to `ppar`. In case of the diet, there seems to be no pattern at all. Although, all of the mice with a `coc` diet and the majority of the mice with a `sun` diet appear in the lower quadrant, while the majority of the mice with a `fish` diet appear in the upper quadrant. Again, any real conclusions need to be left up to the experts to be drawn.



(1) (2)

Figure 19: Individual cases visualizations using `plot.individuals()` for nutrimouse data with `clust=Diet` (1) and `clust=genotype` (2).

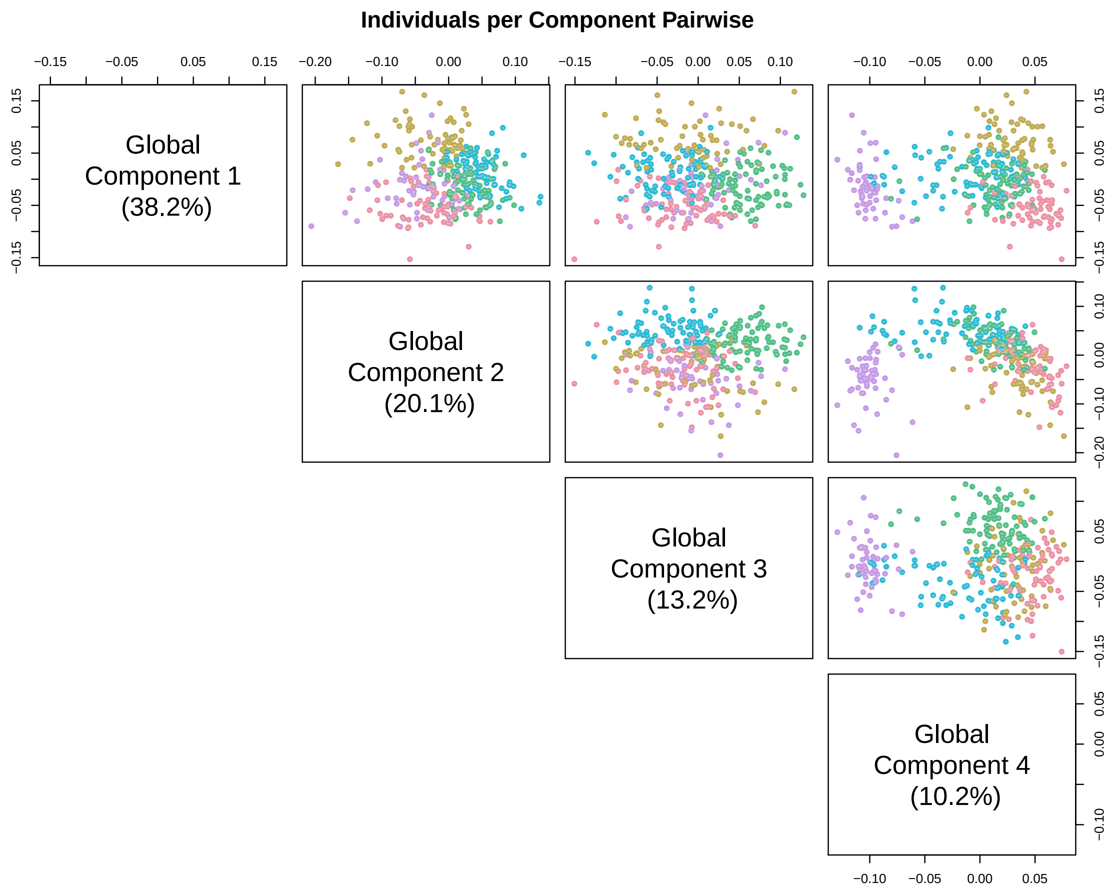


Figure 20: Pairwise individual cases visualizations using `plot.pairwise()` for BRCA data with `clust=NULL`.

The second type of individual cases visualization, `plot.pairwise()` might give some more insight into the data sets. It plots the component scores pairwise for each component combination. The pairwise plot for BRCA data is shown in Figure 20, the other two are shown in Appendix I, because of the size of the graphs. It is interesting to see that Figure 20 shows a clear separation of the purple individual cases when looking at component 4. Any additional conclusions are again left up to the experts.

As for the previous individual cases plot, the additional background factors of the nutrimouse data can also be used to cluster the individual cases in the pairwise plot. Figure 21 shows the results when diet is used as clustering, while the genotype clustering is found in Appendix I. In the case of diet clustering component 4 seems to separate the mice with a `sun` diet from the other mice, while component 1 seems to do the same thing for mice with the `coc` diet.

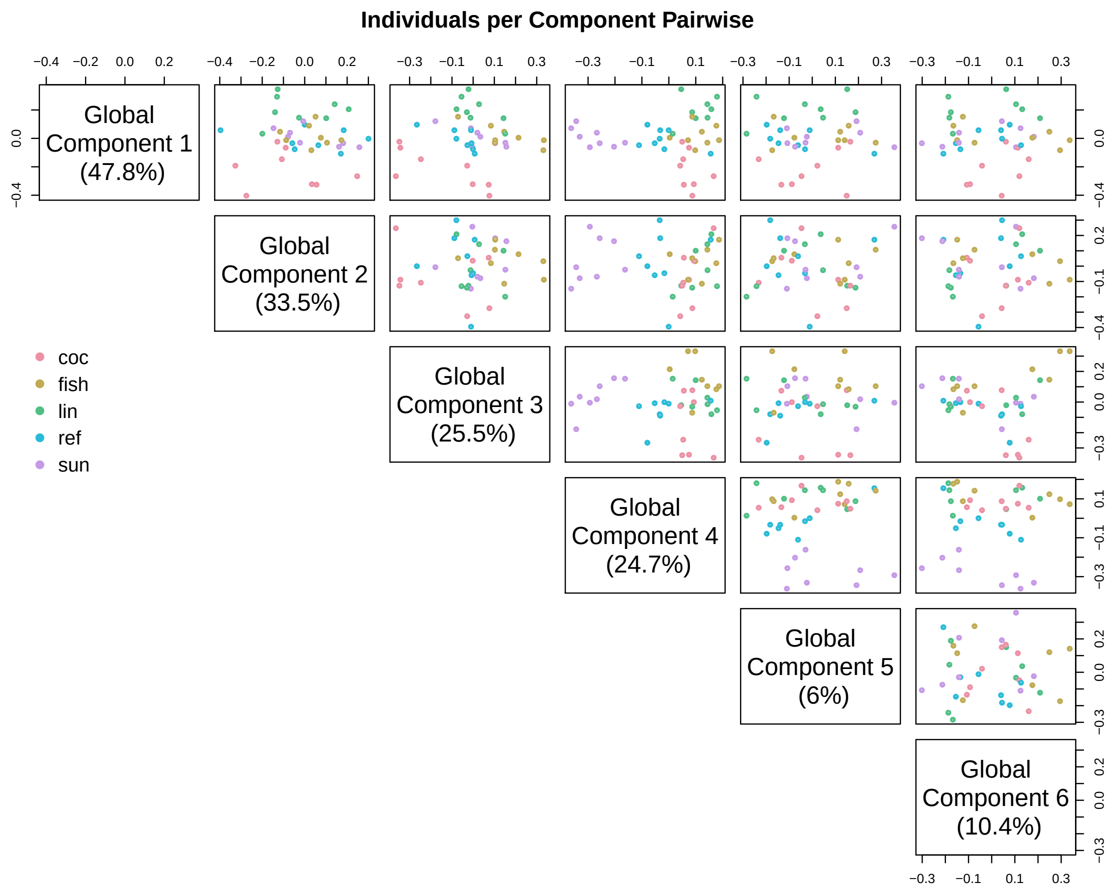


Figure 21: Individual cases visualizations using `plot.pairwise()` for nutrimouse data with `clust=Diet`

8.4. Running times

One of the goals set in Section 3 is that the visualizations can handle large data sets. Therefore, the running times of these functions is analysed using the **microbenchmark** package (Mersmann 2019). For each of the data sets that is used in this paper, the running time for all eight different plotting functions is calculated based on 50 runs per function per data set. The mean of these 50 runs is shown in Figure 22. The running time is only calculated for the plotting functions with their default parameter settings.

All plotting functions run within one second for all data sets, except for the plotting functions that visualize the data model (`plot.treeSCA()`, `plot.circleSCA()`, `plot.comp.indiv()`, `plot.comp.type()`). In those cases the running time increases to a range of five to eight seconds for the liver toxicity and BRCA data sets. The increase of running time does not come as a surprise, as these data sets contain respectively 9 300 and 6 500 data points, which is up to 80 times more than the other data sets. Considering these plots only need to be rendered once or at maximum to the extend of the number of components, the running time is still considered good enough to conclude the visualizations can handle large data sets.

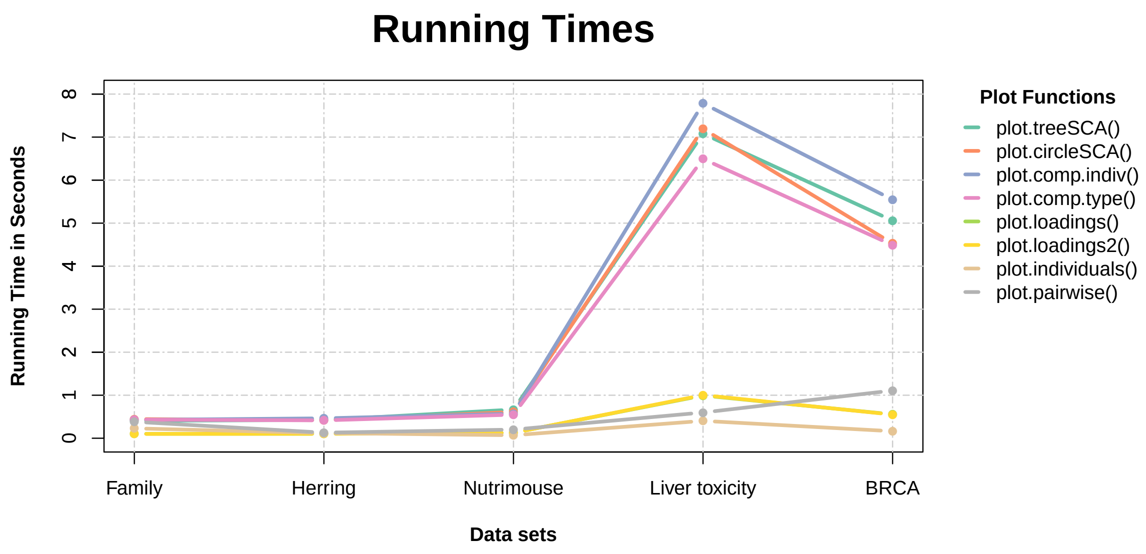


Figure 22: Running times for each plotting function per data set. Each plotting function is run 50 times per data set. The mean running time is shown.

9. Conclusion

With the idea of *A picture is worth a thousand words* in mind, the main purpose of this paper was to create functions for the visualization of the **RegularizedSCA** output. In Section 3 several additional subgoals were formulated to further support the development of the visualizations. The data model visualizations (`plot.treeSCA()`, `plot.circleSCA()`, `plot.comp.indiv()`, `plot.comp.type()`) were developed in the fulfillment of the first subgoal, as they clearly show the structure of the data in terms of the contribution of the variables to each component. A more detailed overview of the component loadings per block of variables is generated with the two loadings functions (`plot.loadings()` and `plot.loadings2()`). Finally, the individual cases are plotted against the different components and component types in the `plot.individuals()` and `plot.pairwise()` visualizations. Together, all eight functions give a complete picture of the data structure within each data set, where each function focuses on a different aspect. These visualizations were created with **RegularizedSCA** in mind, and therefore no effort was made into generalizing the functions input, as in agreement with the second subgoal. The use of colors in the visualizations make the graphs appealing to look at, while the optional parameters allow for specific user preference settings. In addition, by incorporating all support functions into the plotting functions, the code is very user-friendly. The user only needs to process their data according to the principles of **RegularizedSCA**, declare the `blocknames` vector and run each of the visualizations with the correct parameters. Moreover, Section 8.4 has shown that the visualizations are able to handle large data sets, even though running time does somewhat increase in case of larger data sets. Finally, all visualizations are static, and with the use of the `pdf()` function easily exportable to be used in a paper.

10. Future work

Despite the fact that the visualizations have reached each subgoal and the main objective, there are improvements to be made. The data sets in which many variables are present, are less suitable for the data model visualizations in terms of overlapping variables and edges. They also increase the size of the visualizations a lot when it comes to the component loadings plots. This often happens in case a data set contains molecular profiles (e.g., gene expression measurements), as these generally come in large sizes. Therefore, future improvements could be made to reduce the number of genes (or other variables) by clustering them, either before or during the Regularized SCA process. This would reduce the total number of elements in the `P-matrix`. Another approach could be to forcefully set some, or many, of the variables to zero-loadings. This does mean that the components lose some of their amount of variance explained, but it would reduce the number of edges in the data model visualizations, while still retaining all variables. This would especially be an option for data sets with a size similar to the liver toxicity data set, in which the variables are still distinguishable, but the number of edges make the visualization overcrowded. It is less effective for the component loadings visualizations. Although, some alterations could be made to only show the contributing variables and discard those with zero-loadings. Finally, the plotting order of the highlighted data model visualizations (`plot.comp.indiv()`, `plot.comp.type()`) could be improved by first plotting the grey variables, components, and edges and lay the highlighted variables, components, and edges on top. This way the highlighted sections are even more emphasized.

All of these additions might be altered before the functions are fully incorporated in **RegularizedSCA**. Nevertheless, the functions are fully operational in their current form and have proven to be very valuable in regularized simultaneous component analyses.

11. Acknowledgements

This paper is written as part of the Bachelor End Project and is submitted in partial fulfillment of the requirements of the degree of Bachelor of Science to Eindhoven University of Technology and Tilburg University under the supervision of dr. K. van Deun in June 2020.

I would like to express my sincere appreciation towards Katrijn, for her enthusiasm, support, and feedback, throughout the entire process. She has inspired me to get the most out of the project and was always excited when I proposed new ideas. Our collaboration was inspirational and has resulted into great additions for **ReglarizedSCA**.

References

Aleixandre-Tudo JL, du Toit W (2019). "Understanding cold maceration in red winemaking: A batch processing and multi-block data analysis approach." *Lwt*, **111**, 147–157. ISSN 00236438. doi:10.1016/j.lwt.2019.05.020. URL https://doi.org/10.1016/j.lwt.2019.05.020.

Barshan E, Ghodsi A, Azimifar Z, Jahromi MZ (2011). "Supervised principal component analysis: Visualization, classification and regression on subspaces and submanifolds." *Pattern Recognition*, **44**(7), 1357–1371. ISSN 00313203. doi:10.1016/j.patcog.2010.12.015. URL http://dx.doi.org/10.1016/j.patcog.2010.12.015.

Camacho J, Rodríguez-Gómez RA, Saccenti E (2017). "Group-Wise Principal Component Analysis for Exploratory Data Analysis." *Journal of Computational and Graphical Statistics*, **26**(3), 501–512. ISSN 15372715. doi:10.1080/10618600.2016.1265527.

Cancer Genome Atlas Network (2012). "Comprehensive Molecular Portraits of Human Breast Tumours." *Nature*, **490**(7418), 61–70.

Epskamp S, Schmittmann VD, Borsboom D (2012). "qgraph : Network Visualizations of Relationships in Psychometric Data." *Journal of Statistical Software*, **48**(4), 1–18.

Fang CY, Fuh CS, Yen PS, Cherng S, Chen SW (2004). "An automatic road sign recognition system based on a computational model of human recognition processing." *Computer Vision and Image Understanding*, **96**(2), 237–268. ISSN 10773142. doi:10.1016/j.cviu.2004.02.007.

Gu Z, Van Deun K (2019). "RegularizedSCA: Regularized simultaneous component analysis of multiblock data in R." *Behavior Research Methods*, **51**(5), 2268–2289. ISSN 15543528. doi:10.3758/s13428-018-1163-z.

Hassani S, Martens H, Qannari EM, Hanafi M, Borge GI, Kohler A (2010). "Analysis of -omics data: Graphical interpretation- and validation tools in multi-block methods." *Chemometrics and Intelligent Laboratory Systems*, **104**(1), 140–153. ISSN 01697439. doi:10.1016/j.chemolab.2010.08.008.

Heath T, Bizer C (2011). "Linked Data: Evolving the Web into a Global Data Space." *Technical report*, Morgan & Claypool Publishers.

Kassambara A, Mundt F (2020). "factoextra: Extract and Visualize the Results of Multivariate Data Analyses." URL https://cran.r-project.org/package=factoextra.

Kilintzis V, Chouvarda I, Beredimas N, Natsiavas P, Maglaveras N (2019). "Supporting integrated care with a flexible data management framework built upon Linked Data, HL7 FHIR and ontologies." *Journal of Biomedical Informatics*, **94**. ISSN 15320464. doi:10.1016/j.jbi.2019.103179. URL https://doi.org/10.1016/j.jbi.2019.103179.

Loudiyi M, Rutledge DN, Aït-Kaddour A (2018). "ComDim for explorative multi-block data analysis of Cantal-type cheeses: Effects of salts, gentle heating and ripening." *Food Chemistry*, **264**, 401–410. ISSN 18737072. doi:10.1016/j.foodchem.2018.05.039. URL https://doi.org/10.1016/j.foodchem.2018.05.039.

Mattson MP (2014). “Superior pattern processing is the essence of the evolved human brain.” *Frontiers in Neuroscience*, **8**, 1–17. ISSN 1662453X. doi:10.3389/fnins.2014.00265.

Mavoa S, Oliver M, Witten K, Badland HM (2011). “Linking GPS and travel diary data using sequence alignment in a study of children’s independent mobility.” *International Journal of Health Geographics*, **10**(64), 1–10. ISSN 1476072X. doi:10.1186/1476-072X-10-64.

Mersmann O (2019). “microbenchmark: Accurate Timing Functions.” URL https://cran.r-project.org/package=microbenchmark.

R Core Team (2013). “R: A language and envrionment for statistical computing.” URL http://www.r-project.org/.

Reinsel D, Gantz J, Rydning J (2018). “The Digitization of the World - From Edge to Core.” *Technical report*, IDC White Paper. URL https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

Rohart F, Gautier B, Singh A, Lê Cao Ka (2017). “mixOmics : an R package for ‘ omics feature selection and multiple data integration.” *PLoS Computational Biology,*, **13**(11), 1–21.

Rosa LN, de Figueiredo LC, Bonafé EG, Coqueiro A, Visentainer JV, Març̧o PH, Rutledge DN, Valderrama P (2017). “Multi-block data analysis using ComDim for the evaluation of complex samples: Characterization of edible oils.” *Analytica Chimica Acta*, **961**, 42–48. ISSN 18734324. doi:10.1016/j.aca.2017.01.019.

Sánchez-Rada JF, Torres M, Iglesias CA, Maestre R, Peinado E (2014). “A Linked Data Approach to Sentiment and Emotion Analysis of Twitter in the Financial Domain.” *CEUR Workshop Proceedings*, **1240**, 51–62.

Schwartz M (2013). “R barplot: wrapping long text labels?” URL https://stackoverflow.com/questions/20241065/r-barplot-wrapping-long-text-labels.

Van Mechelen I, Smilde AK (2010). “A generic linked-mode decomposition model for data fusion.” *Chemometrics and Intelligent Laboratory Systems*, **104**(1), 83–94. ISSN 01697439. doi:10.1016/j.chemolab.2010.04.012. URL http://dx.doi.org/10.1016/j.chemolab.2010.04.012.

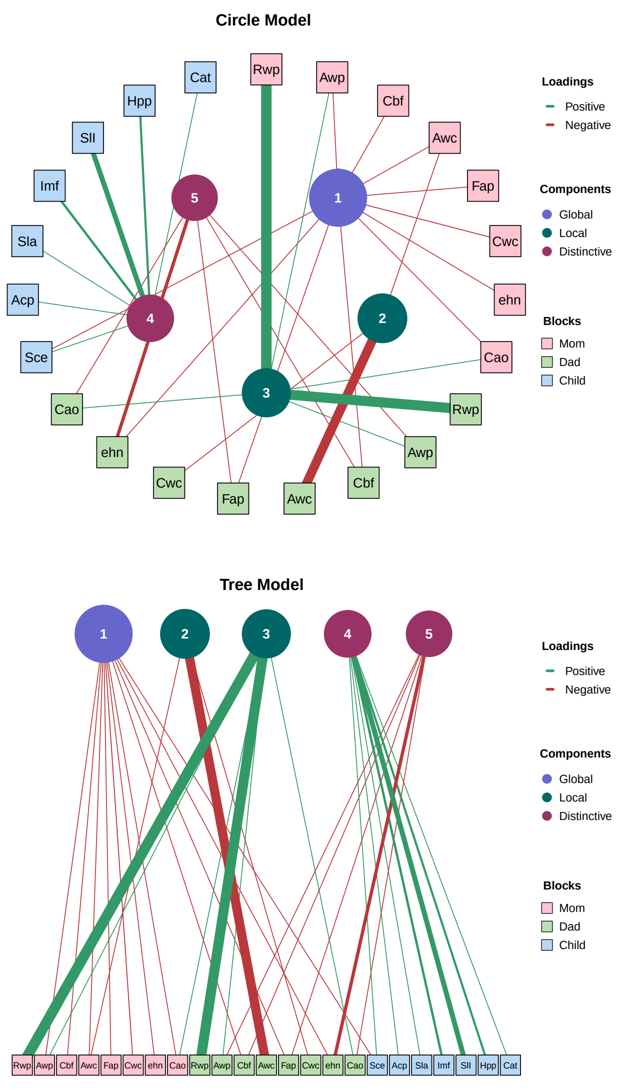
Vellido A, Martín JD, Rossi F, Lisboa PJ (2010). “Seeing is believing: The importance of visualization in real-world machine learning applications.” In *ESANN 2011 proceedings, 19th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pp. 219–226. ISBN 9782874190445.

Xiang S, Yuan L, Fan W, Wang Y, Thompson PM, Ye J (2014). “Bi-level multi-source learning for heterogeneous block-wise missing data.” *NeuroImage*, **102**(P1), 192–206. ISSN 10959572. doi:10.1016/j.neuroimage.2013.08.015. URL http://dx.doi.org/10.1016/j.neuroimage.2013.08.015.

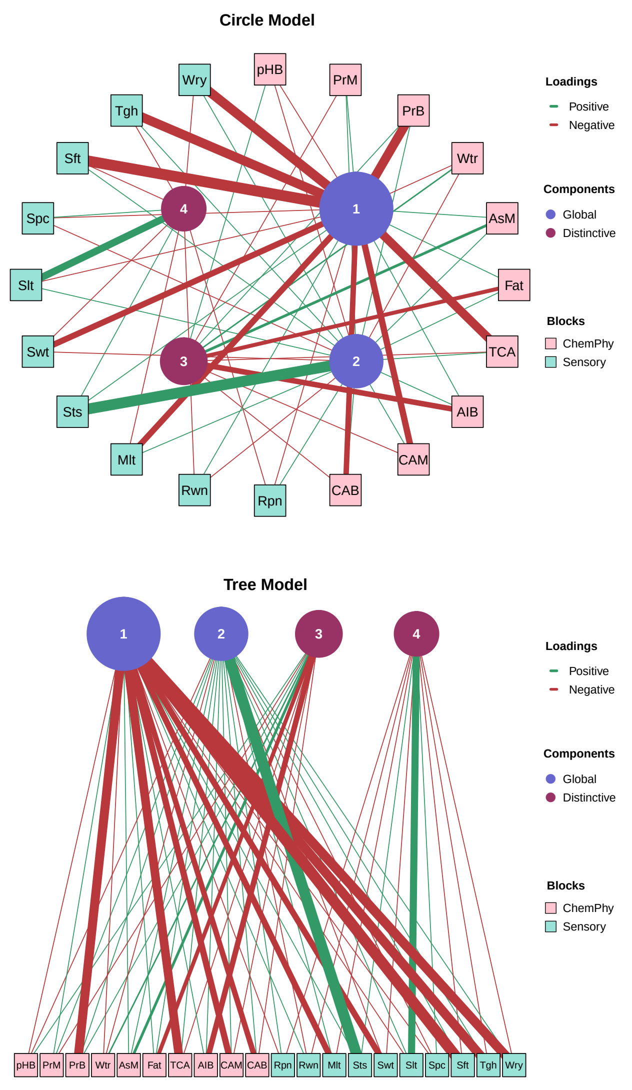
E. Alternative model plots

This section contains the circle and tree model plots while the parameter `width` is set to `TRUE`.

```
plot.circleSCA(fam_data, data_fam, num_var_fam, block_names_fam, width=TRUE)
plot.circleSCA(herring_data, data_her, num_var_her, block_names_her, width=TRUE)
plot.treeSCA(fam_data, data_fam, num_var_fam, block_names_fam, width=TRUE)
plot.treeSCA(herring_data, data_her, num_var_her, block_names_her, width=TRUE)
```



(1)



(2)

Figure 23: Circle model plots and tree model plots of family (1) and herring data (2) where `width=TRUE`.

F. Additional highlighted model plots

This section includes the additional highlights options from `plot.comp.indiv()` and `plot.comp.type()`.

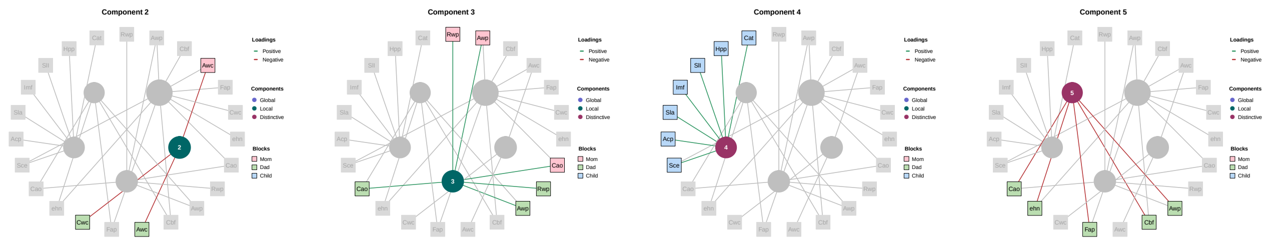


Figure 24: Highlighted plots of family data using `plot.comp.indiv()` for component 2 to 5.

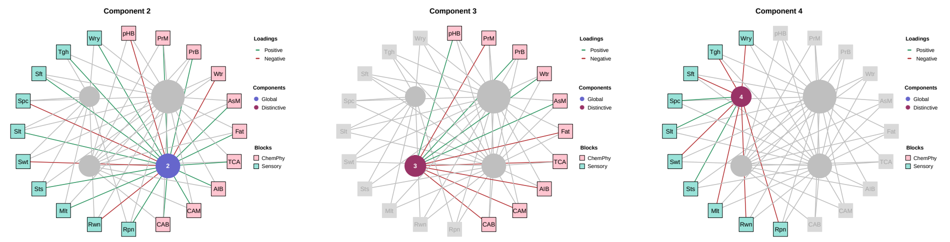


Figure 25: Highlighted plots of herring data using `plot.comp.indiv()` for component 2 to 4.

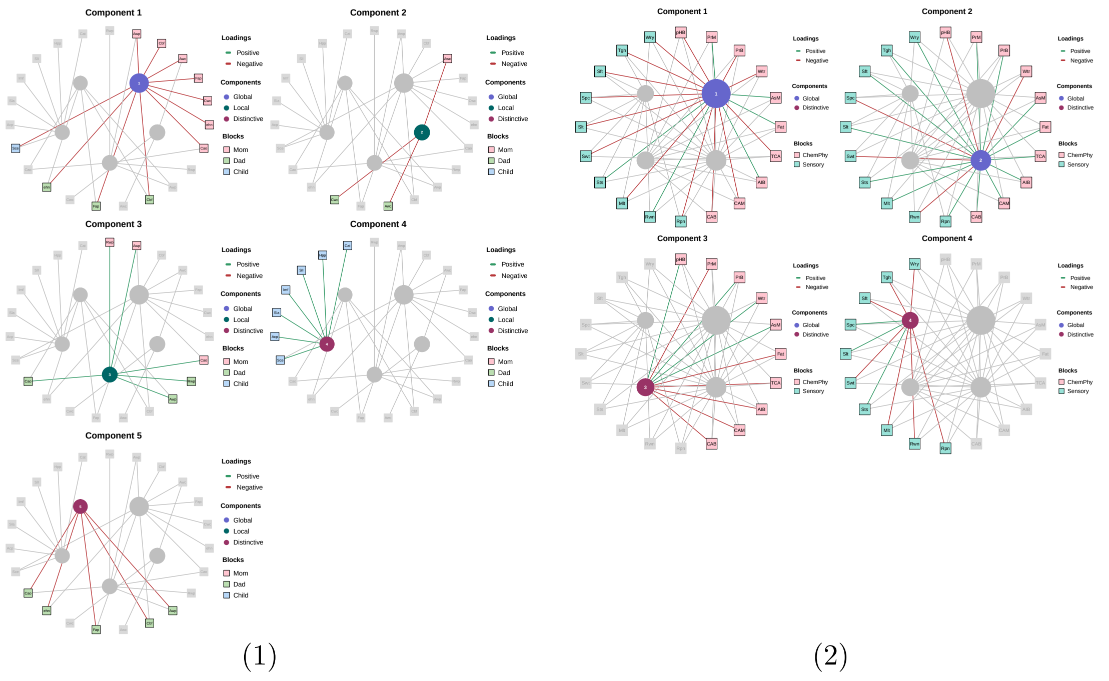


Figure 26: Highlighted plots of family data (1) and herring data (2) of each component using `plot.comp.indiv()` with `number="all"`.

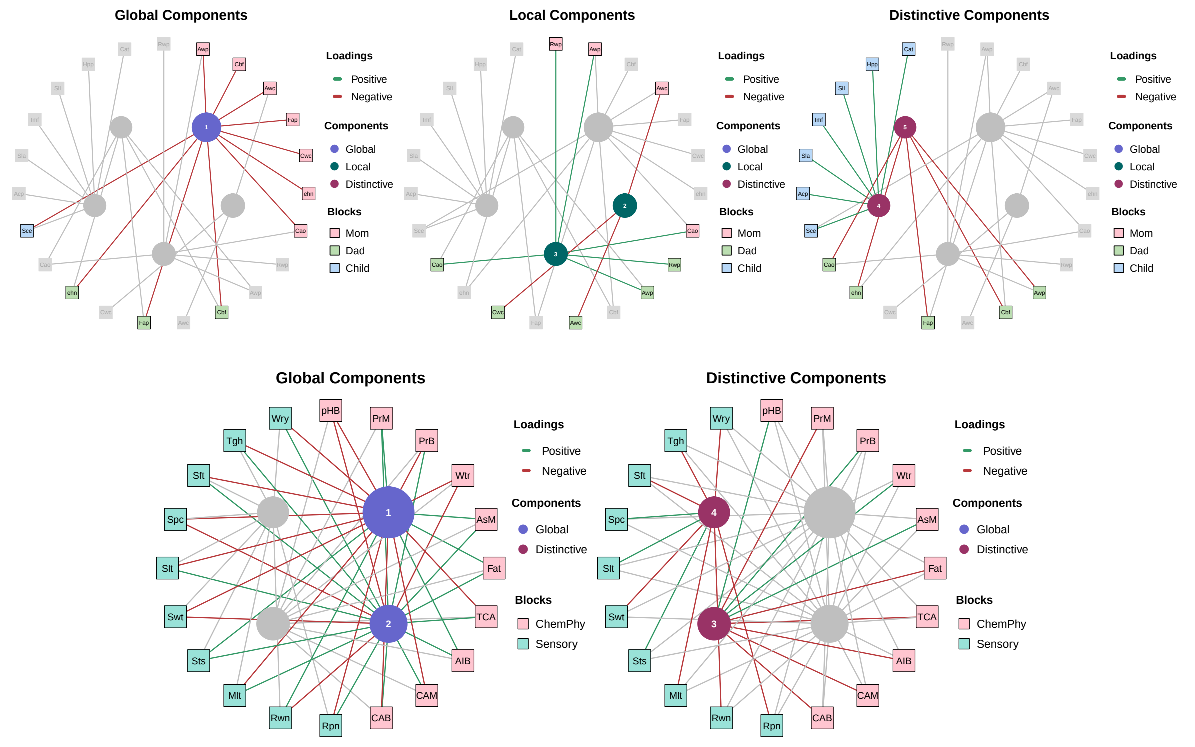


Figure 27: Highlighted plots of family data (row 1) and herring data (row 2) of each component type using `plot.comp.type()` with `type="all"`.

G. Additional component loadings plots

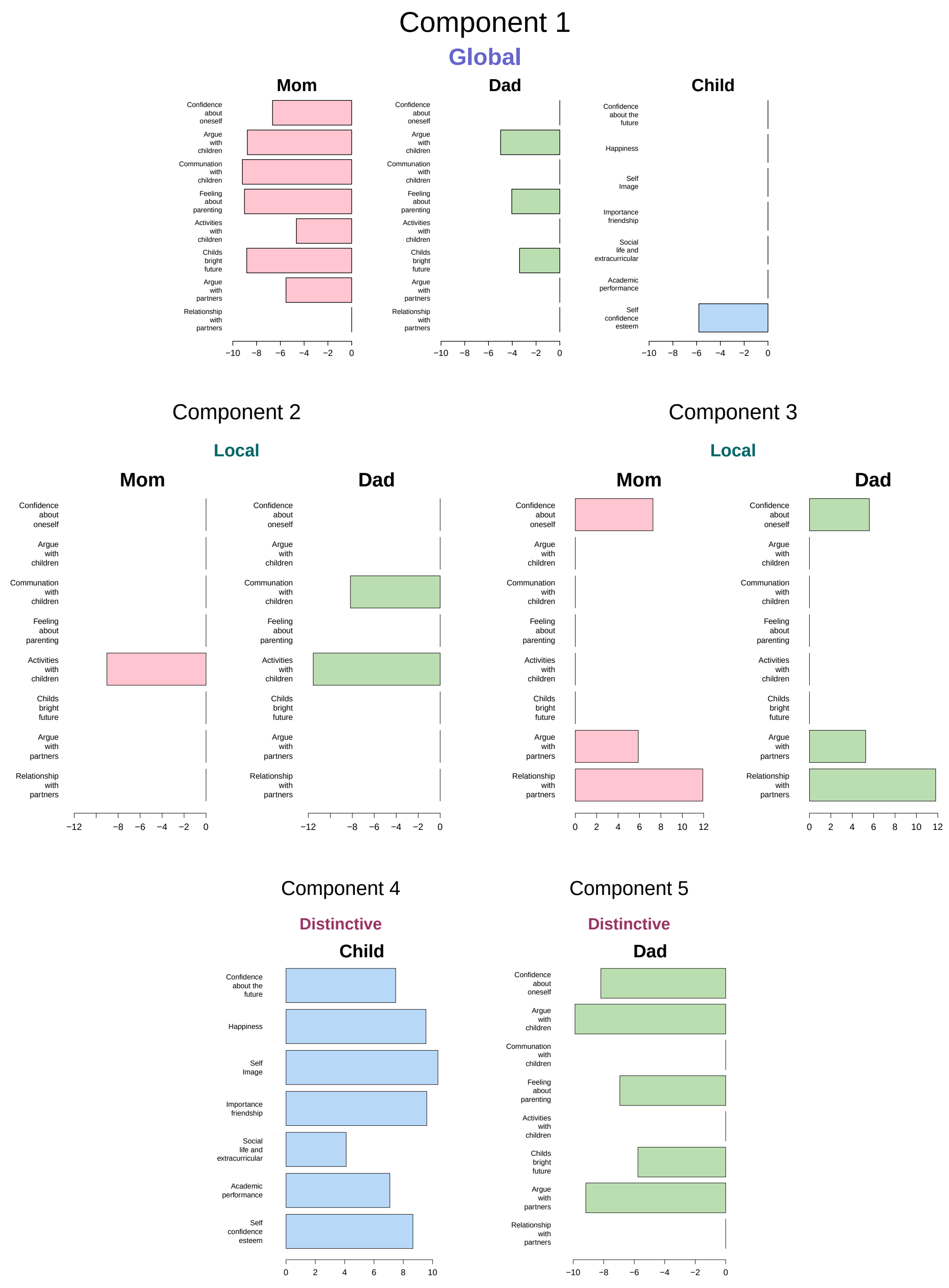


Figure 28: Loadings visualizations of family data using `plot.loadings2()`.

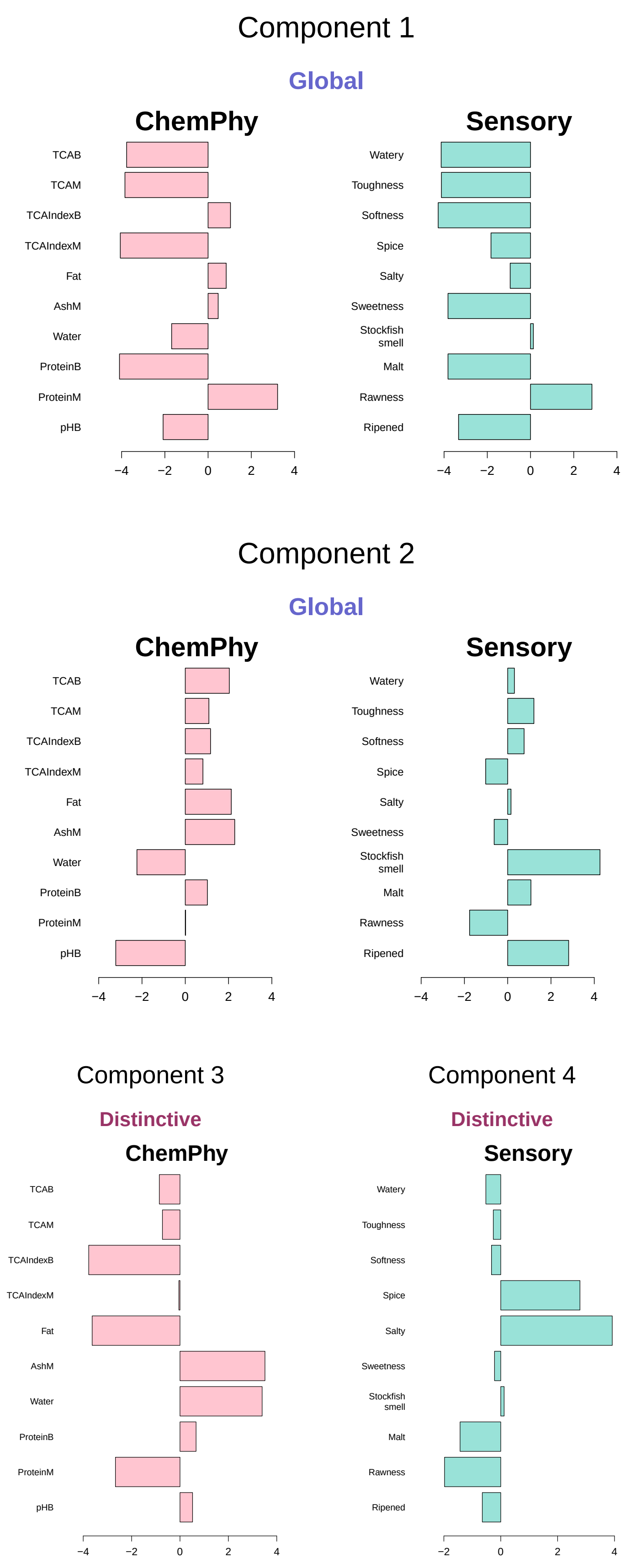


Figure 29: Loadings visualizations of herring data using `plot.loadings2()`.

H. Pairwise plot herring data

```
plot.pairwise(herring_data, data_her, num_var_her, block_names_her)
```

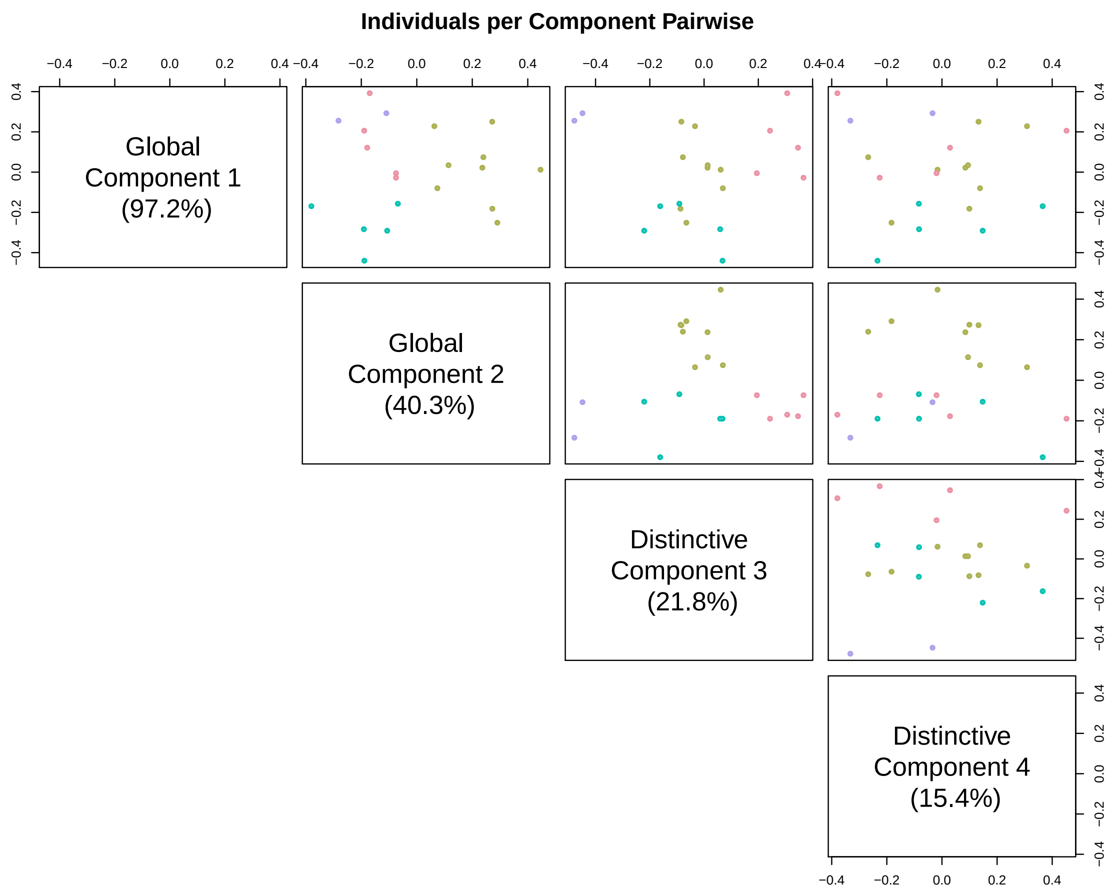
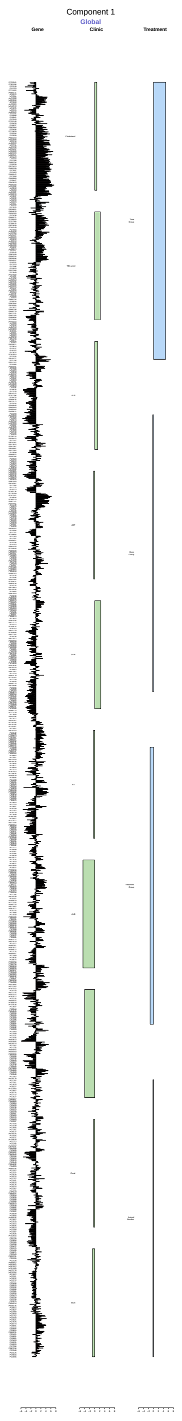
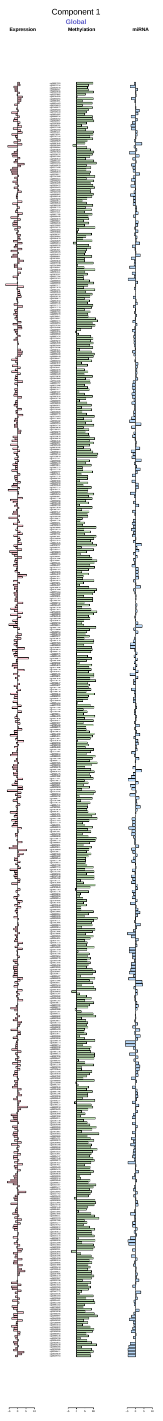


Figure 30: Pairwise individual cases visualization per component using `plot.pairwise()` with default options for family data.

I. Additional application examples plots



(1)



(2)

Figure 31: Component loadings visualizations using `plot.loadings()` of liver toxicity data (1), and BRCA data (2) showing the first component

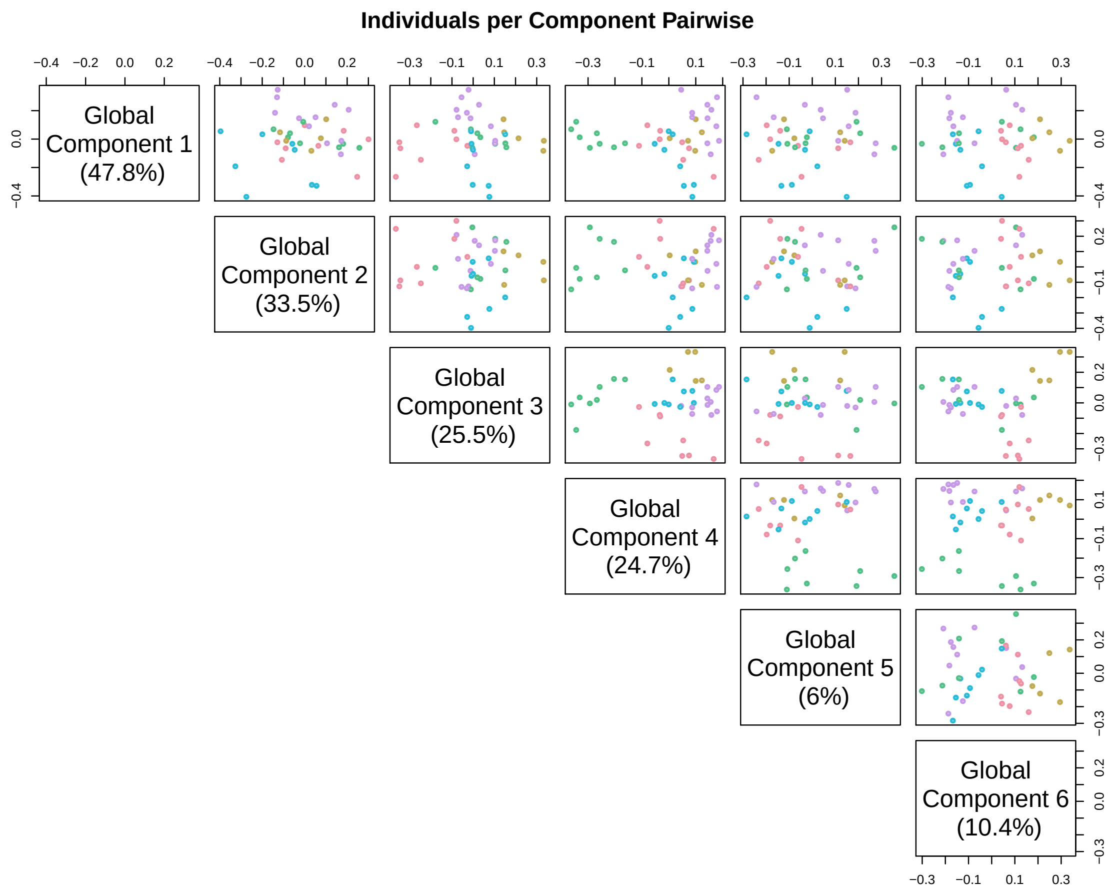


Figure 32: Pairwise individual cases visualization using `plot.pairwise()` for nutrimouse data with `clust=NULL`.

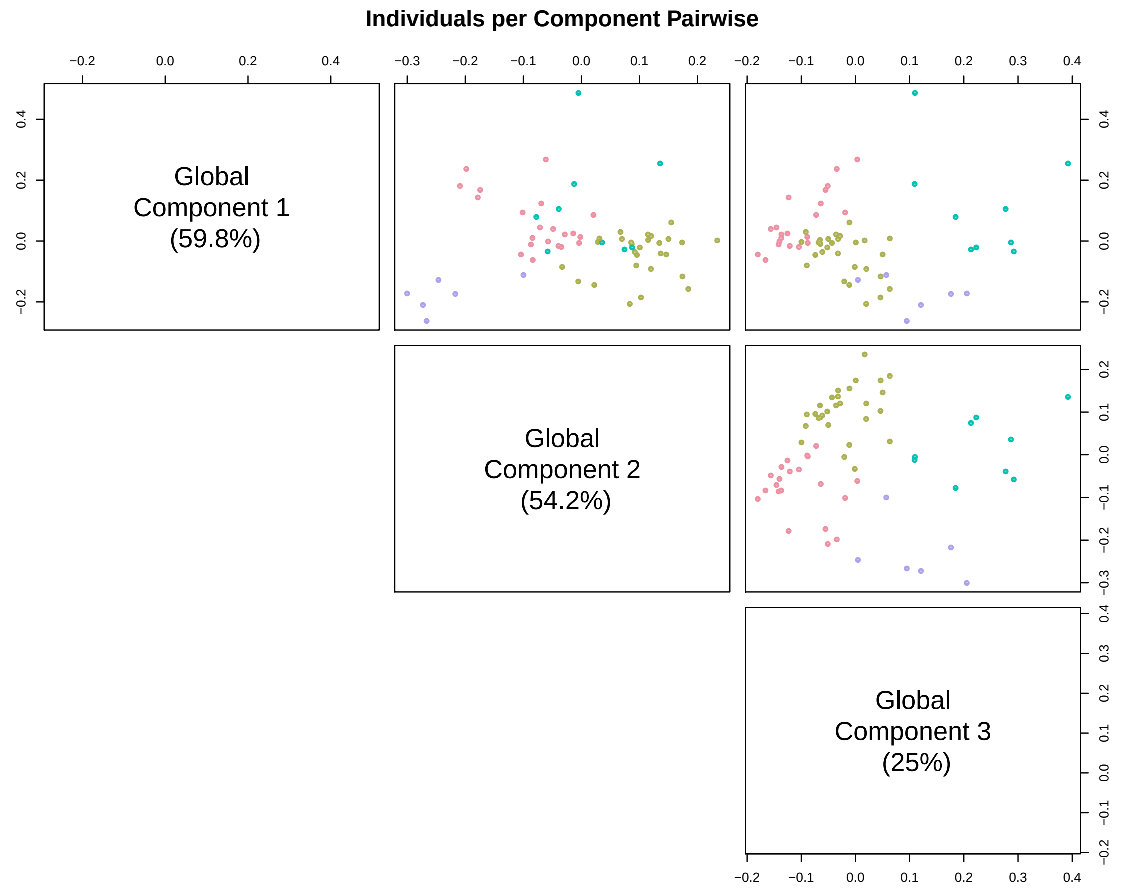


Figure 33: Pairwise individual cases visualization using `plot.pairwise()` for liver toxicity data with `clust=NULL`.

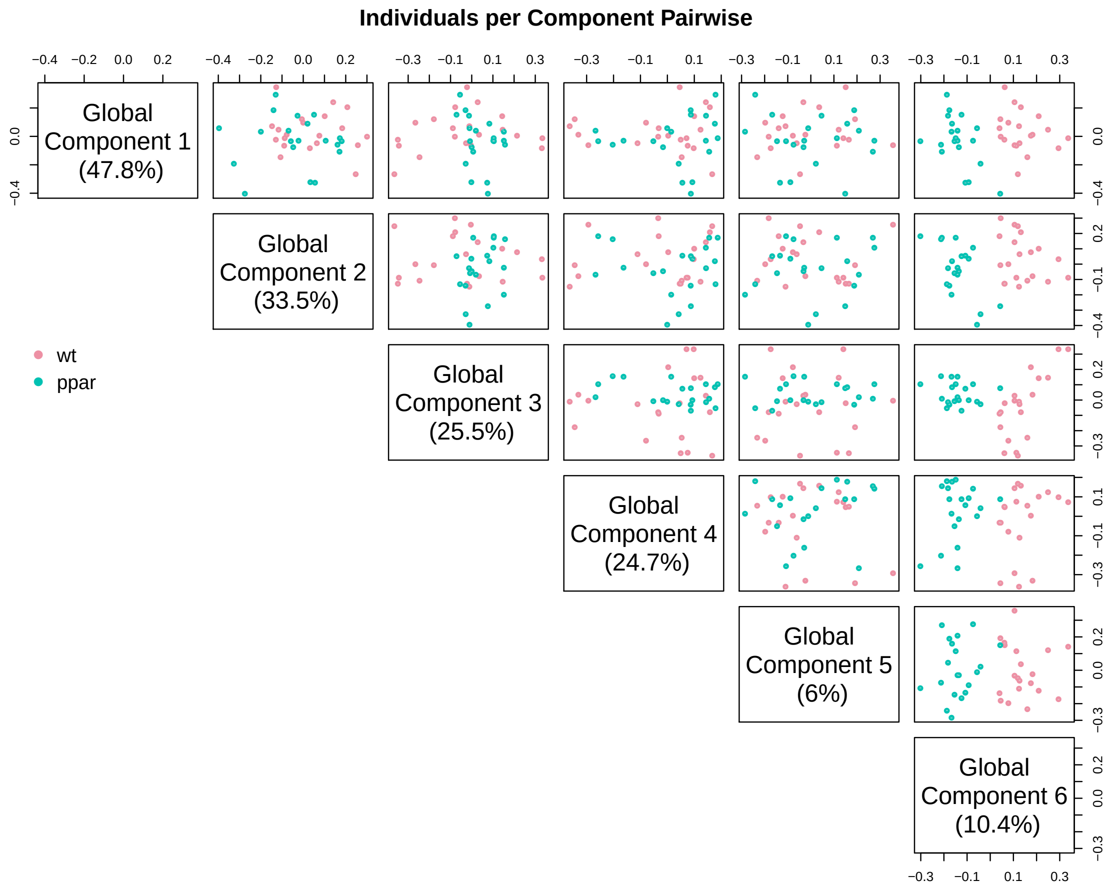


Figure 34: Individual cases visualization using `plot.pairwise()` for nutrimouse data with `clust=Genotype`.

Affiliation:

Josephina Cornelia Mathilda Arts
Bachelor End Project (Bachelor Thesis)
Submitted In Partial Fulfillment of the Requirements
For the Degree of Bachelor of Science - Data Science (Joint Degree)
Tilburg University - Eindhoven University of Technology
E-mail: `j.c.m.arts@student.tue.nl`
Under supervision of dr. Katrijn van Deun - Tilburg University
Spring 2020