R.ComDim (a tutorial)

Francesc Puig Castellví

2022-10-23

Contents

Chapter 1

About

This is the documentation of the **R.ComDim** R-package.

Chapter 2

Introduction

2.1 The ComDim method

ComDim (also known as CCSWA) is an unsupervised multi-block method that aims to simultaneously consider multiple data tables to find the latent components that are common to all the tables as well as those that are specific to each data table, along with the contribution of each of the tables to each of these components. ComDim determines a common space describing the dispersion of the samples in all the blocks, each block having a specific weight (`salience`) associated with each dimension in this common space. Significant differences in the saliences for a given dimension reflect the fact that the dimension contains different amounts of information coming from each block. In addition to the saliences, `Local loadings` for each analyzed block and two different sets of scores are obtained. The first set corresponds to the `Local scores` for each analyzed block while the second set is composed of the `Global scores`, common to all the blocks.

2.2 Why should I use ComDim?

- To analyze **different types of data** (ex. multi-omics) and see how they are untangled.
- To extract the common profiles of **related variables** (ex. metabolites detected in the same pathway).
- To deal with **unbalanced multi-block** datasets (ex. different number of sample replicates in the blocks). However, ComDim can also deal with **balanced multi-block** datasets.
- Within the data from the same analytical platform, to evaluate **inter-sample variability** and **batch effects** related to the analytical platform.

- To investigate **cross-platform variability**, which is useful to detect errors in the sample preparation.

2.3 Functions

To successfully extract all the potential of the ComDim method, several functions coded in R are proposed. Some of them are listed below:

- `MultiBlock()`: To initialize a `MultiBlock` object with the first data-block(s).
- `BuildMultiBlock()`: To combine several single data-blocks into a `MultiBlock` object, containing some of them metadata information.
- `ExpandMultiBlock()`: To combine several single data-blocks into a `MultiBlock` object, containing some of them metadata information.
- `NormalizeMultiBlock()`: To normalize (some or all) the data-blocks of the `MultiBlock` object.
- `ProcessMultiBlock()`: To apply customized data transformation to (some or all) the data-blocks of the `MultiBlock` object.
- `ComDim_PCA_MB()`: This function applies the ComDim-PCA algorithm on the `MultiBlock` object resulting from `BuildMultiBlock()` or `ExpandMultiBlock()`.

For more information on the usage of these functions, please consult the next chapters and the help (`?`).

2.4 Install and load R.ComDim package

```
if (!require("devtools")) install.packages("devtools")
library("devtools")
install_github("f-puig/R.ComDim")

# Load R.ComDim
library("R.ComDim")
```

2.5 References

- Puig-Castellví, F.; Jouan-Rimbaud Bouveresse, D.; Mazéas, L.; Chapleur, O.; Rutledge, D. N. Rearrangement of incomplete multi-omics datasets combined with ComDim for evaluating replicate cross-platform variability

and batch influence. Chemom. Intell. Lab. Syst. 2021, 18 (104422). **(DOI)**

- Qannari, E. M.; Courcoux, P.; Vigneau, E. Common Components and Specific Weights Analysis Performed on Preference Data. Food Qual. Prefer. 2001, 12 (5–7), 365–368.**(DOI)**
- Mazerolles, G.; Hanafi, M.; Dufour, E.; Bertrand, D.; Qannari, E. M. Common Components and Specific Weights Analysis: A Chemometric Method for Dealing with Complexity of Food Products. Chemom. Intell. Lab. Syst. 2006, 81 (1), 41–49. **(DOI)**
- Claeys-Bruno, M.; Béal, A.; Rutledge, D. N.; Sergent, M. Use of the Common Components and Specific Weights Analysis to Interpret Supersaturated Designs. Chemom. Intell. Lab. Syst. 2016, 152, 97–106. **(DOI)**

Chapter 3

Create a MultiBlock

3.1 Option 1: for small MultiBlocks

In order to use the ComDim algorithm, the data blocks need to be combined into a `MultiBlock` object.

`MultiBlock` has 5 fields: `Samples`, `Data`, `Variables`, `Metadata` and `Batch`. The easiest way to create a `MultiBlock` is by using the function `MultiBlock()` as below:

```
b1 = matrix(rnorm(500),10,50) # 10 rows and 50 columns
b2 = as.data.frame(matrix(rnorm(800),10,80)) # 10 rows and 80 columns
b2[c(2,3,4),c(5,7,8)] <- NA # Making some data missing, just for fun.
rownames(b2) <- LETTERS[5:14] # Adding some sample names
b3 = MultiBlock(Samples = 1:10,
                Data = list(s1 = b1, s2 = b2),
                Variables = list(s1 = 1:ncol(b1),
                                 s2 = 1:ncol(b2)))
```

With the code above, a `MultiBlock` containing 2 blocks was built. As shown, the provided data blocks support the format `matrix` and `data.frame`.

When building a `MultiBlock`, only the fields `Samples`, `Data`, and `Variables` are mandatory.

```
b4 = MultiBlock(Samples = 1:10,
                Data = list(s1 = b1, s2 = b2),
                Variables = list(s1 = 1:ncol(b1),
                                 s2 = 1:ncol(b2)),
                Batch = list(s1 = rep('Batch1',10),
                             s2 = c(rep('Batch2',5),rep('Batch3',5))))
```

And to create a `MultiBlock` with sample metadata, the easiest way is with the function `AddMetadata`:

```
b4 <- AddMetadata(newBlock = matrix(rnorm(100),10,10),
                  metadata = data.frame(x1 = c(rep(0,5),rep(1,5)),
                                        x2 = c(rep(1,3),rep(2,4),rep(3,3))))
```

In `AddMetadata`, newBlock can be a matrix, a data.frame, or a MultiBlock object. In the giving example, the metadata contains two variables (`x1` and `x2`).

3.2 Option 2: for large MultiBlocks

Then, it is possible to create a `MultiBlock` from another preexisting `MultiBlock`. In this case, the function `BuildMultiBlock()` comes in handy.

```
MB <- BuildMultiBlock(b1,b2,b3,ignore.names = TRUE)
getBlockNames(MB)
```

3.3 Option 3: from SummarizedExperiment or MultiAssayExperiment objects.

`MultiBlocks` can also be created from `SummarizedExperiment` or `MultiAssayExperiment` objects, which are pretty common format nowadays in multi-omics studies. In the same way, `MultiBlocks` can be converted back to `MultiAssayExperiments`.

```
library(MultiAssayExperiment)
data(miniACC)
library(SummarizedExperiment)
data(airway, package="airway")

MB1 <- SummarizedExperiment2MultiBlock(airway,
                                       colData_samplenames = 'Run',
                                       Batch = NULL)
MB2 <- MultiAssayExperiment2MultiBlock(miniACC,
                                       colData_samplenames = 'patientID',
                                       Batch = NULL)
se2 <-  MultiBlock2MultiAssayExperiment(MB2, MSEmetadata = NULL)
```

3.4 Sample correspondence across blocks

We define as **sample correspondence across blocks** to the fact that the sample order is maintained across blocks. That is, the first sample from the first block has a correspondence sample in the second block at the same position, and so on.

To run ComDim **and most multi-omics analysis** in general, sample correspondence across blocks is needed. This is normally verified by the user. For `R.ComDim`, we can use `BuildMultiBlock()` to check for the same correspondence and resort samples (and discard) if needed. This process is achieved with the option `ignore.names = FALSE` (the default).

```
c1 = matrix(1:500,10,50) # 10 rows and 50 columns
c2 = matrix(500:1,10,50) # 10 rows and 50 columns
c3 = matrix(501:1000,10,50) # 10 rows and 50 columns
c4 = matrix(1:1000,20,50) # 20 rows and 50 columns
rownames(c1) <- paste0('c',6:15)
rownames(c2) <- paste0('c',1:10)
rownames(c3) <- paste0('c',10:1)
rownames(c4) <- paste0('c',1:20)
# With ignore.names = FALSE, only common samples across blocks are kept.
# Samples will be resorted if needed.
MB12 <- BuildMultiBlock(c1,c2,ignore.names = FALSE) # 10 samples in common
MB13 <- BuildMultiBlock(c1,c3,ignore.names = FALSE) # 5 samples in common
MB13b <- BuildMultiBlock(c1,c3,ignore.names = TRUE) # Blocks were appended
                                                    # regardless of their sample
                                                    # names. Sample names were
                                                    # replaced by integers.
# (Not run) The following code does not work because block sizes are different
# and ignore.names = TRUE.
#MB14 <- BuildMultiBlock(c1,c4,ignore.names = TRUE)
```

If we don't need to verify the sample correspondence across blocks, we can use `ignore.names = TRUE`. However, in case the sample size is different across blocks, the `MultiBlock` will not be built (as the sample correspondence does not actually exist). It is possible to overrid this situation with `ignore.size = TRUE`, but the resulting `MultiBlock` is **not compatible** for `ComDim` analyses.

```
# Option A (ignore.names = FALSE) : Only common samples across blocks are kept.
MB14c <- BuildMultiBlock(c1,c4,ignore.names = FALSE)
# Option B (ignore.names = TRUE, ignore.size = TRUE):
#    Blocks were appended regardless of their sample names and sizes.
#    All samples were kept.
MB14b <- BuildMultiBlock(c1,c4,ignore.names = TRUE, ignore.size = TRUE)
```

```
# MB14b is not compatible with ComDim
getSampleNames(MB14b)
# A particularity of this MB14b is that it has one 'Samples' vector per block.
```

3.5 Splitting blocks by the Batch criterion.

We can consider split one or more blocks according to the Batch criterion, in order to examine whether this factor contains is a relevant source of information in our dataset. This is performed with the `SplitRW()` function. We call this split strategy **Replicate-Wise** (RW).

```
# Build the MultiBlock
x1 = MultiBlock(Samples = 1:10,
                Data = list(x1 = matrix(rnorm(500),10,50)),
                Variables = list(x1 = 1:50),
                Batch = list(x1 = rep('Batch1',10)))
x2 = MultiBlock(Samples = c(1:10,1:10,1:10),
                Data = list(x2 = matrix(rnorm(2400),30,80)),
                Variables = list(x2 = 1:80),
                Batch = list(x2 = c(rep('Batch1',10),
                                    rep('Batch2',10),
                                    rep('Batch3',10)))
                )
x1x2 <- BuildMultiBlock(x1,x2, ignore.names = TRUE, ignore.size = TRUE)
# Proceed with the split.
rw <- SplitRW(x1x2) # SplitRW looks specifically to the Batch information.
```

More information about the reasoning behind this analytical strategy can be consulted here: [**https://doi.org/10.1016/j.chemolab.2021.104422**] (https://doi.org/10.1016/j.chemolab.2021.104422).

In case there exist sample correspondence although the sample names do not match across blocks, we can force the MultiBlock split with the argument `checkSampleCorrespondence = FALSE`.

```
# Build the MultiBlock
x4 = MultiBlock(Samples = c(1:30),
                Data = list(x4 = matrix(rnorm(2400),30,80)),
                Variables = list(x4 = 1:80),
                Batch = list(x4 = c(rep('Batch1',10),
                                    rep('Batch2',10),
                                    rep('Batch3',10)))
)
```

```
x1x4 <- BuildMultiBlock(x1,x4, ignore.names = TRUE, ignore.size = TRUE)
  # Sample names in block x1 go from 1 to 10.
  # Sample names in block x4 go from 1 to 10 (batch1), 11 to 20 (batch2),
  # and 21 to 30 (batch3). Despite the sample names does not batch, samples
  # have correspondence across blocks. Thus, we need to impose **not to**
  # check the sample names. This is done with checkSampleCorrespondence = FALSE.
# Proceed with the split.
rw2 <- SplitRW(x1x4, checkSampleCorrespondence = FALSE)
```

With `checkSampleCorrespondence = TRUE`, only the common samples across blocks are kept.

```
# Build the MultiBlock
x5 = MultiBlock(Samples = c(1:10, 5:14, 1:10),
                Data = list(x5 = matrix(rnorm(2400),30,80)),
                Variables = list(x5 = 1:80),
                Batch = list(x5 = c(rep('Batch1',10),
                                    rep('Batch2',10),
                                    rep('Batch3',10)))
)
x1x5 <- BuildMultiBlock(x1,x5, ignore.size = TRUE, ignore.names = TRUE)
# Proceed with the split.
rw3 <- SplitRW(x1x5, checkSampleCorrespondence = TRUE)
```

Chapter 4

MultiBlock data handling

4.1 Inspecting the data

We can easily read the content of a MultiBlock object with the following functions:

```
getSampleNames(MB)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
getVariableNames(MB)
```

```
## $b1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
##
## $b2
##  [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10" "V11" "V12"
## [13] "V13" "V14" "V15" "V16" "V17" "V18" "V19" "V20" "V21" "V22" "V23" "V24"
## [25] "V25" "V26" "V27" "V28" "V29" "V30" "V31" "V32" "V33" "V34" "V35" "V36"
## [37] "V37" "V38" "V39" "V40" "V41" "V42" "V43" "V44" "V45" "V46" "V47" "V48"
## [49] "V49" "V50" "V51" "V52" "V53" "V54" "V55" "V56" "V57" "V58" "V59" "V60"
## [61] "V61" "V62" "V63" "V64" "V65" "V66" "V67" "V68" "V69" "V70" "V71" "V72"
## [73] "V73" "V74" "V75" "V76" "V77" "V78" "V79" "V80"
##
## $s1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

```
##
## $s2
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
## [51] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
## [76] 76 77 78 79 80
```

```
getVariableNames(MB, block = 2)
```

```
##  [1] "V1"  "V2"  "V3"  "V4"  "V5"  "V6"  "V7"  "V8"  "V9"  "V10" "V11" "V12"
## [13] "V13" "V14" "V15" "V16" "V17" "V18" "V19" "V20" "V21" "V22" "V23" "V24"
## [25] "V25" "V26" "V27" "V28" "V29" "V30" "V31" "V32" "V33" "V34" "V35" "V36"
## [37] "V37" "V38" "V39" "V40" "V41" "V42" "V43" "V44" "V45" "V46" "V47" "V48"
## [49] "V49" "V50" "V51" "V52" "V53" "V54" "V55" "V56" "V57" "V58" "V59" "V60"
## [61] "V61" "V62" "V63" "V64" "V65" "V66" "V67" "V68" "V69" "V70" "V71" "V72"
## [73] "V73" "V74" "V75" "V76" "V77" "V78" "V79" "V80"
```

```
getBlockNames(MB)
```

```
## [1] "b1" "b2" "s1" "s2"
```

```
getBlockNames(MB, "Batch")
```

```
## NULL
```

```
getBlockNames(MB, "Metadata")
```

```
## NULL
```

```
ncolMultiBlock(MB)
```

```
## [1] 50 80 50 80
```

```
nrowMultiBlock(MB)
```

```
## [1] 10 10 10 10
```

```
MB2<- FilterSamplesMultiBlock(MB, c(1:9)) # To create a MultiBlock with
 # a sample subset. In this case, we selected samples 1-9.
```

Or we can simply see all the content with `str()` function.

If needed, everything in the MB can be renamed.

```
MB <- setBlockNames(MB, paste("X", 1:4, sep = '')) # The blocks
getBlockNames(MB)
MB <- setSampleNames(MB, LETTERS[1:10]) # The samples
getSampleNames(MB)
MB <- setVariableNames(MB, paste("vars", 1:50, sep = ''), 1)
getVariableNames(MB, 1)
```

4.2 Data pre-processing

The `R.ComDim` package contains functions to easily apply some data transformations. This can result in handy since we there is no need to build the `MultiBlock` every time a new data transformation is tested.

```
## NA removal
# We first add some NAs to the MultiBlock.
MB@Data$X2[c(2,3,5),c(1,2,3)] <- NA # Add some NAs
allMB <- NARemoveMultiBlock(MB, method = 'none', minfrac = 0.2)
# Variables containing more than 20% of NAs will be discarded

## Data normalization
allMB <- NormalizeMultiBlock(allMB, method = 'norm')
# MB is normalized (mean-center and divided by the block norm)
```

The `R.ComDim` package has been writted with the idea to result very flexible and, as such, it allows the user to apply custom data transformations. For instance, in the code below, all variables with values lower than the 5% of the most intense value are discarded.

```
# MB is converted to matrix to calculate the max value.
maxMB <- max(MultiBlock2matrix(allMB), na.rm = TRUE)
# Variables are filtered with the ProcessMultiBlock() function.
allMB <- ProcessMultiBlock(allMB,
  FUN.SelectVars = function(x) {apply(x,2,max) > maxMB * 0.05})
```

Chapter 5

Multi-omics amd Multi-blocks

This section will be explained with a multi-omics dataset from an exposition experiment on normal and tumoral cells. The dataset contains 4 different types of data (RNAseq, lipidomics, intracellular and extracellular metabolites). The studied samples can be classified into 3 groups:

- NI - non-induced (normal cells)
- DOX - doxycycline induced (tumoral cells),
- OFF - residual cells (treatment-resistant tumoral cells)

More information regarding the experimental protocol can be consulted **here**. The metabolomics data was downloaded from Metabolights **MTBLS1507** while the RNAseq data was obtained from ARRAYEXPRESS **E-MTAB-8834**.

```
data(mouse_ds)

allMB <- BuildMultiBlock(t(RNAseq3[,1:12]), t(lipids), t(intra), t(extra))
allMB <- setBlockNames(allMB, c('RNAseq', 'lipids','intra','extra'))
```

5.1 Data processing

This is an example of a possible data processing:

```
# 1) Exclude normalized variables with max intensity reported below 0.1% of the
#    max from all RNAseq blocks.
```

```
allMB <- ProcessMultiBlock(
  allMB,
  blocks = 'RNAseq',
  FUN.SelectVars = function(x) {apply(x,2,max) > max(x, na.rm = TRUE) * 0.001})
allMB <- NARemoveMultiBlock(allMB, blocks = 'RNAseq',
                            method = 'fixed.value.all', constant = 1)

# 2) Do rlog transform of the RNAseq data (rlog assumes samples in columns)
#    The rlog transform function is obtained from the DESeq2 R-package.
library(DESeq2)
allMB <- ProcessMultiBlock(allMB, blocks = 'RNAseq',
                           FUN = function(x){t(DESeq2::rlog(t(x)))})
# 3) Replace NAs by random noise
allMB <- NARemoveMultiBlock(allMB, method = 'random.noise')

# 4) Normalize (mean-center and divided by each block-norm)
allMB <- NormalizeMultiBlock(allMB, method = 'norm')
```

5.2 Data analysis

We use in this example ComDim-PCA, which is intended for exploratory purposes. The ComDim analysis is run with 2 components.

```
resultsPCA <- ComDim_PCA_MB(allMB, ndim = 2) # 2 Components.
```

`resultsPCA` is an object of class `ComDim`. Let's proceed now to inspect some of the results from this analysis.

5.2.1 Saliences

We can start by looking at the saliences, which show the contribution of each block for every component.

```
saliences <- resultsPCA@Saliences %>%
  as.data.frame() %>%
  mutate(dataset = rownames(.)) %>%
  pivot_longer(cols = c('CC1','CC2'),
               names_to = 'CC',
               values_to = 'Salience')

saliences # The salience values.
```

```
ggplot(data = saliences,
       aes(x = CC, y = Salience, group = dataset )) +
  geom_bar(stat = 'identity', position = 'dodge',
           aes(fill = dataset)) +
  theme_minimal() +
  labs(title = 'ComDim Saliences')
```

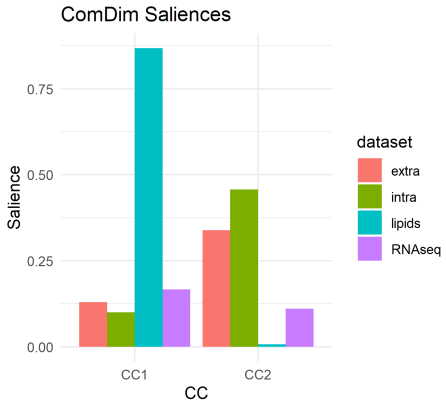


Figure 5.1: Saliences

5.2.2 Scores

The scores give the contribution of each sample. There are two types of scores, **Global** (Q.scores) and **Local** (T.scores). **Global scores** show the contribution for the total model (all blocks) while the **Local scores** give the contribution for each of the blocks.

```
scoresTable <- MakeComDimScoresTable(model = resultsPCA)

head(scoresTable) # The 6 first rows of the scores table

scoresTable_wider <- scoresTable %>%
  mutate(sample.type = case_when(grepl('DOX', sample.id) ~ 'DOX',
                                 grepl('NI', sample.id) ~ 'NI',
                                 grepl('OFF', sample.id) ~ 'OFF')) %>%
  dplyr::select(sample.id, sample.type, block.name, scores.type.dim, value) %>%
  dplyr::group_by(sample.id, sample.type, scores.type.dim, block.name) %>%
  pivot_wider(names_from = scores.type.dim, values_from = value)

ggplot(data = scoresTable_wider) +
```

```
  geom_point(aes(x = T.scores1, y = T.scores2, color = sample.type,
                 shape = block.name)) +
  geom_point(aes(x = Q.scores1, y = Q.scores2,
                 fill = sample.type, shape = block.name),
             size = 3, shape = 24, color = 'black') +
  theme_minimal() +
  labs(title = 'ComDim scores', x = 'CC1', y = 'CC2')
  # OFF samples are very different than DOX and NI by their lipid content.
  # Extracellular metabolites are not very different
```

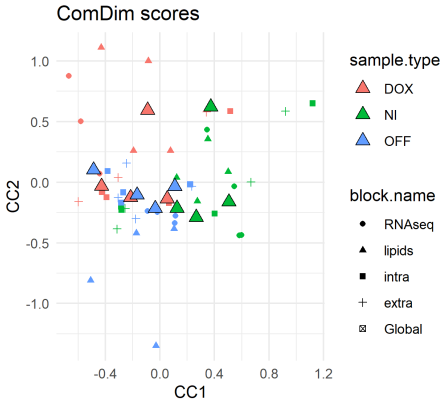


Figure 5.2: Scores

5.2.3 Loadings

The **loadings** give the contribution of each variable in the model. And example of how to use the loadings is presented in **CHAPTER6**.

5.3 Data prediction

The built model can be used to investigate new samples if available.

```
pred.resultsPCA <- Predict_MB(MB = allMB, model = resultsPCA)
data.frame(original = resultsPCA@T.scores[[2]][,1],
           predicted = pred.resultsPCA@T.scores[[2]][,1])
data.frame(original = resultsPCA@Q.scores[,2],
           predicted = resultsPCA@Q.scores[,2])
```

In this example, the original and predicted scores are identical because the new samples are the same as the original ones.

Chapter 6

Data from a single omics data are multi-blocks

Metabolic pathways are composed by a group of related metabolites. Extrapolating this concept into the MultiBlock domain, it is possible to convert a **metabolomics dataset** into a `MultiBlock`, where each of the blocks will be characteristic of one **metabolic pathway**.

This strategy can be applied to other types of omics datasets. For example, **transcriptomics datasets** can be transformed to `MultiBlocks` by the **Gene Ontology** information, and **phylogenetic data** can be split according to any of the **taxonomic levels** (class, family, gender, species,...).

In the `R.ComDim` package, this data transformation can be mediated with the `ExpandMultiBlock()` function and a reference metadata file with the list of categories each variable can be listed in. In the resulting `MultiBlock`, a variable will be included in as many blocks as groups (i.e. molecular function) it belongs to.

```
data(mouse_ds)
lipidsMB <- ExpandMultiBlock(data = lipids, metadata = metadata_lipids,
                             minblock = 0, loquace = FALSE)
extraMB <- ExpandMultiBlock(data = extra, metadata = KEGG_table_metabolites,
                            minblock = 10, loquace = FALSE)
intraMB <- ExpandMultiBlock(data = intra, metadata = KEGG_table_metabolites,
                            minblock = 10, loquace = FALSE)
RNAseqMB <- ExpandMultiBlock(data = RNAseq3[,1:12],
                             metadata = metadata_RNAseq3,
                             minblock = 500, loquace = FALSE)
# We can count the number of blocks in each MultiBlock
length(getBlockNames(lipidsMB)) # 4 blocks
```

```
length(getBlockNames(extraMB))  # 12 blocks
length(getBlockNames(intraMB))  # 12 blocks
length(getBlockNames(RNAseqMB)) # 16 blocks
```

Since the blocks from this `MultiBlock` are related to a specific biological role, the **ComDim analysis** can be used to determine the biological roles more important in the the studied dataset.

In the `MultiBlock`s above, we only kept those blocks containing equal or more than `minblock` variables (i.e. only the RNAseq-related blocks containing 500 or more variables were kept). In order to find the most relevant pathways, ComDim will consider all blocks equally important, causing that the variables from the smallest blocks will contribute more to the final model than the variables from the largest blocks. Then, the `minblock` filter is applied to avoid that the smallest blocks (which usually relate to poorly-characterized biological roles, and thus hardly interpretable) influence the ComDim model construction.

Let's continue with the example from before, but before the ComDim analysis we can apply some data transformations.

```
# Blocks are relabelled for clarity
lipidsMB <- setBlockNames(lipidsMB,
                          paste("lipids", getBlockNames(lipidsMB), sep = '.'))
intraMB <- setBlockNames(intraMB,
                         paste("intra", getBlockNames(intraMB), sep = '.'))
extraMB <- setBlockNames(extraMB,
                         paste("extra", getBlockNames(extraMB), sep = '.'))
RNAseqMB <- setBlockNames(RNAseqMB,
                          paste("RNAseq", getBlockNames(RNAseqMB), sep = '.'))

allMB2 <- BuildMultiBlock(RNAseqMB, lipidsMB, intraMB, extraMB)
```

Now, all 4 `MultiBlock`s were merged into a single `MultiBlock`, and each block contains a suffix denoting the omics data type.

We apply some data pre-processings:

```
# We apply some pre-processings
  library(DESeq2)
  # Remove blocks relative to map01100
  # (not very informative, it's the map with all metabolic pathways)
  allMB2 <- ProcessMultiBlock(allMB2,
    blocks = which(grepl('map01100', getBlockNames(allMB2))),
    # All blocks with map01100 are deleted, since ncol(x) is always > 0.
    FUN.SelectBlocks = function(x){ncol(x) < 0})
  # Calculate the absolute maximum from the RNAseq data.
```

```
maxMB <- max(MultiBlock2matrix(allMB2,
                               blocks = grep('RNAseq',getBlockNames(allMB2))
                               ),
             na.rm = TRUE)
# Exclude normalized variables with max intensity reported below 0.1%
# of the max from all RNAseq blocks.
allMB2 <- ProcessMultiBlock(allMB2,
   blocks = grep('RNAseq',getBlockNames(allMB2)),
   FUN.SelectVars = function(x) {apply(x,2,max) > maxMB * 0.001})
# Add 1 to each value in RNAseq data to remove 0s.
allMB2 <- NARemoveMultiBlock(allMB2,
   blocks = grep('RNAseq',getBlockNames(allMB2)),
   method = 'fixed.value.all',
   constant = 1)
# Do rlog transform of the RNAseq data.
allMB2 <- ProcessMultiBlock(allMB2,
   blocks = grep('RNAseq',getBlockNames(allMB2)),
   # Normalize rlog transcript counts
   FUN = function(x){t(DESeq2::rlog(t(x)))})
# Replace NAs by random noise
allMB2 <- NARemoveMultiBlock(allMB2, method = 'random.noise')
# Normalize (mean-center and divided by each block-norm)
allMB2 <- NormalizeMultiBlock(allMB2, method = 'norm')
```

We continue with the ComDim analysis:

```
resultsPCA2 <- ComDim_PCA_MB(allMB2, ndim = 2)
```

A first look at the scores can tell us how the blocks relate to the sample type.

```
scoresTable <- MakeComDimScoresTable(model = resultsPCA2)
scoresTable_wider <- scoresTable %>%
  mutate(sample.type = case_when(grepl('DOX', sample.id) ~ 'DOX',
                                 grepl('NI', sample.id) ~ 'NI',
                                 grepl('OFF', sample.id) ~ 'OFF')) %>%
  dplyr::select(sample.id, sample.type, block.name, scores.type.dim, value) %>%
  dplyr::group_by(sample.id, sample.type, scores.type.dim, block.name) %>%
  pivot_wider(names_from = scores.type.dim, values_from = value) %>%
  mutate(omics = case_when(grepl('RNAseq', block.name) ~ 'RNAseq',
                           grepl('lipids', block.name) ~ 'lipids',
                           grepl('extra', block.name) ~ 'extra',
                           grepl('intra', block.name) ~ 'intra',
                           grepl('Global', block.name) ~ 'Global'))
```

```
  ggplot(data = scoresTable_wider) +
    geom_point(aes(x = T.scores1, y = T.scores2, color = sample.type, shape = omics)) +
    geom_point(aes(x = Q.scores1, y = Q.scores2, fill = sample.type),
               size = 3, shape = 24, color = 'black') +
    theme_minimal() +
    labs(title = 'ComDim scores', x = 'CC1', y = 'CC2')
```

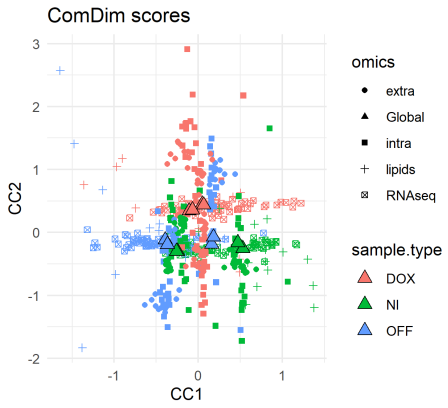


Figure 6.1: Scores

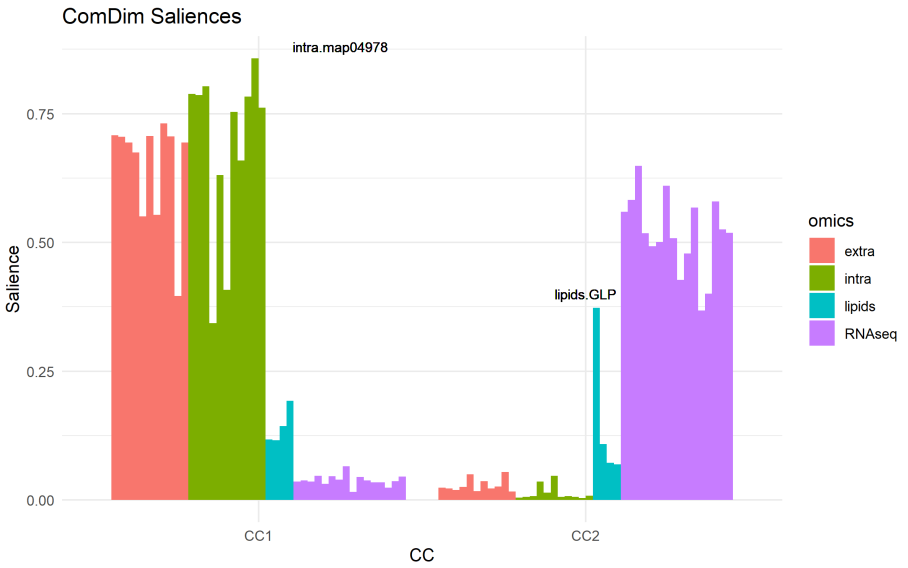
The metabolomic and the RNAseq profiles appear to be very orthogonal (uncorrelated), since intra and extra does not change much (within each group) in CC2 whereas the RNAseq does not change much in CC1. For RNAseq molecular functions, CC1 separates OFF from NI and DOX. CC2 separates DOX from NI and OFF. Despite this, NI seems to be different from DOX and OFF at the lipidomic level.

6.1 Molecular functions

Let's start by looking at the **saliences**:

```
# Plot saliences
saliences2 <- resultsPCA2@Saliences %>%
  as.data.frame() %>%
  mutate(dataset = rownames(.)) %>%
  pivot_longer(cols = c('CC1','CC2'),
               names_to = 'CC',
               values_to = 'Salience') %>%
  mutate(omics = case_when(grepl('RNAseq', dataset) ~ 'RNAseq',
                           grepl('lipids', dataset) ~ 'lipids',
```

```
                          grepl('extra', dataset) ~ 'extra',
                          grepl('intra', dataset) ~ 'intra'))
ggplot(data = saliences2,
     aes(x = CC, y = Salience, group = dataset )) +
geom_bar(stat = 'identity', position = 'dodge',
         aes(fill = omics)) +
theme_minimal() +
geom_text(label = 'lipids.GLP', x = 2, y = 0.4, size = 3) +
geom_text(label = 'intra.map04978', x = 1.25, y = 0.88, size = 3) +
labs(title = 'ComDim Saliences')
```



CC1 is descriptive of metabolomics data (both intra and extra) CC2 is descriptive of RNAseq and glycerophospholipids (lipids.GLP) The most altered molecular function is map04978 (mineral absorption), in the "intra" data.

6.1.1 The molecular patwhays

By inspecting the loadings, we can have an idea of the most altered omics features in the block. We will plot now the loadings for the two most relevant blocks as seen in the previous section.

```
# MINERAL ABSORPTION
LoadingsTable <- MakeComDimLoadingsTable(model = resultsPCA2,
                                         block = 'intra.map04978',
                                         dim = 1)
ggplot(LoadingsTable, aes(x = variable.id, y = value)) +
```

```
  geom_bar(stat = 'identity') +
  labs(title = 'MINERAL ABSORPTION') +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1))

# GLYCEROPHOSPHOLIPIDS
LoadingsTable <- MakeComDimLoadingsTable(model = resultsPCA2,
                                         block = 'lipids.GPL',
                                         dim = 2)
ggplot(LoadingsTable, aes(x = variable.id, y = value)) +
  geom_bar(stat = 'identity') +
      labs(title = 'GLYCEROPHOSPHOLIPIDS') +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1,
                                   size = 3))
```

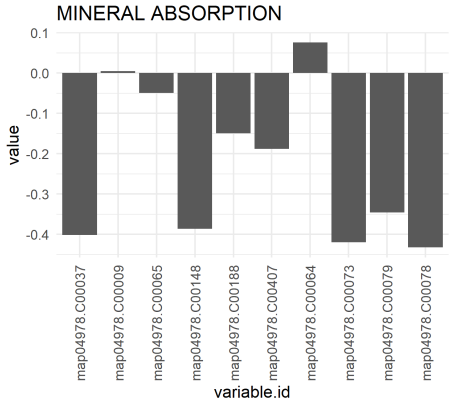
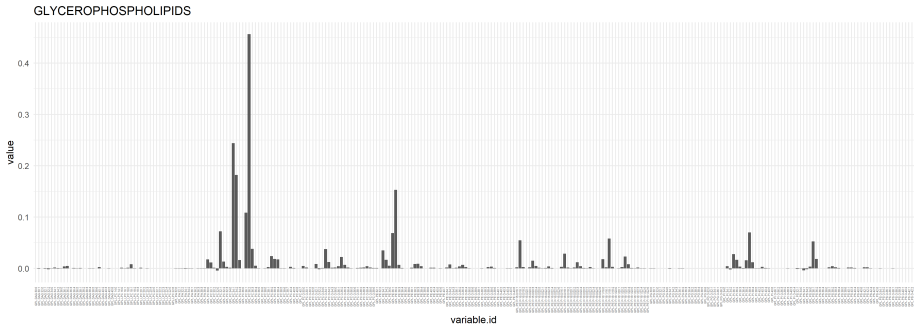


Figure 6.2: Mineral absorption



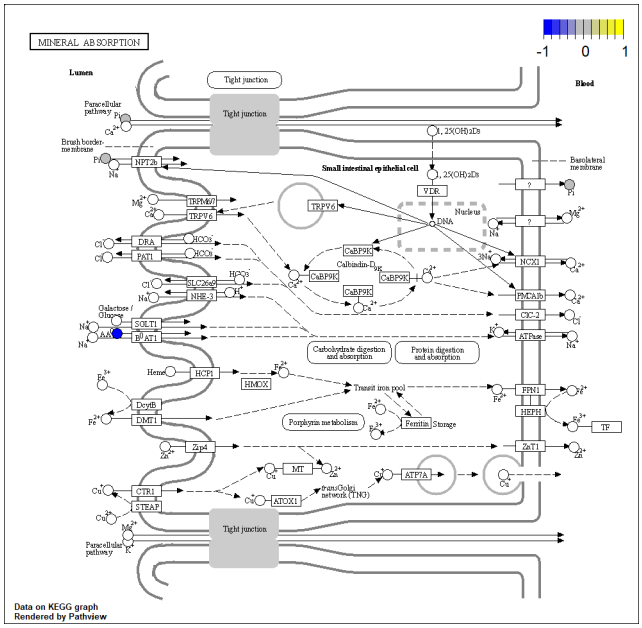
Most important changes in the GPL group occur for PC36:2, PC34:1, PC:34:2, PE36:2, PC36:1,...

We can also plot the loadings into **KEGG pathway maps** (if the omics variables can be matched with the KEGG identifiers).

Here is an example for the metabolomics data:

```
# MINERAL ABSORPTION
# Prepare input data
loadingVector <- LoadingsTable$value * 10 # 10 is a factor used to maximize
                                          # the color contrast in the resulting
                                          # KEGG map.
names(loadingVector) <- gsub("map04978.","",LoadingsTable$variable.id)

library(pathview) # From Bioconductor
pv.out <- pathview(cpd.data = loadingVector,
                   pathway.id = "04978",
                   species = "mmu",
                   out.suffix = "ComDim")
```



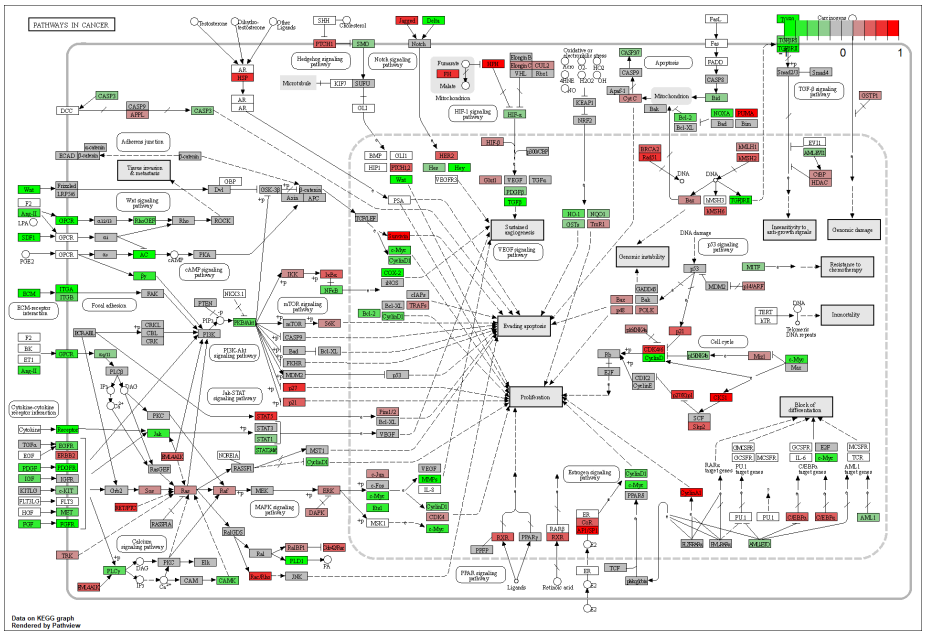
The map is saved in the working directory:

And here, there is an example for the RNAseq-data (block mmu05200, "Pathways in Cancer"):

```
LoadingsTable <- MakeComDimLoadingsTable(model = resultsPCA2,
                                         block = 'RNAseq.mmu05200',
                                         dim = 2)
# Prepare input data
loadingVector <- LoadingsTable$value * 10 # 10 is a factor to maximize
```

```
                                        # the color contrast in the
                                        # resulting KEGG map.
names(loadingVector) <- gsub("mmu05200.","",LoadingsTable$variable.id)

library(pathview) # From Bioconductor
pv.out <- pathview(gene.data = loadingVector,
                   pathway.id = "05200",
                   species = "mmu",
                   out.suffix = "ComDim")
```



Chapter 7

Other ComDim methods

In the previous sections, we have only used **ComDim-PCA**, but many other ComDim versions exist. ComDim can be regarded as a chemometric method able to compact the relevant information of the data into a reduced subspace, defined by the **scores**, the **loadings** and the **saliences**. Although most ComDim analyses employ **Principal Component Analysis (PCA)** as the core method to find this reduced space, other approaches can be used.

The `R.Comdim` package includes a few of those other methods, as well as some additional functions **to allow the users to create their own ComDim methods**.

7.1 ComDim-Partial Least Squares (ComDim-PLS)

ComDim-PLS can be executed with the function `ComDim_PLS_MB()`, by providing a `y` regression vector (i.e. numerical) and indicating that the ComDim `method='PLS-R`. The output of the **ComDim-PLS** method is an object of the `ComDim` class (like when `ComDim_PCA_MB()` is used) that contains the typical PLS figures of merit (**R2X**, **R2Y**, **Q2**), the PLS regression coefficients **B** and **B0**, and the **predicted y** according to the ComDim-PLS model.

```
resultsPLSR <- ComDim_PLS_MB(allMB,
                             y= c(rep(1,4),rep(5,4),rep(10,4)),
                             # y is an arbitrary numerical vector,
                             # just for testing the function.
                             ndim = 2, method = 'PLS-R')
resultsPLSR@R2X
resultsPLSR@R2Y
```

```
resultsPLSR@Q2

resultsPLSR@PLS.model$B
resultsPLSR@PLS.model$B0

resultsPLSR@Prediction$Y.pred
```

7.2 ComDim-Partial Least Squares Discriminant Analysis (ComDim-PLSDA)

ComDim-PLSDA is executed with the same function as **ComDim-PLS**, `ComDim_PLS_MB()`, but it requires a `y` vector with the sample classes and that `method='PLS-DA`. In addition to the *ComDim-PLS* outputs, the object resulting from the **ComDim-PLSDA** analysis has additional elements for assessing the discriminant power of the model. These outputs are the **discriminant Q2 DOI**, the model **specificity**, the model **sensitivity**, and a **confusion matrix** with the list of correct and wrong class assignments.

```
resultsPLSDA <- ComDim_PLS_MB(allMB,
                              y= c(rep('NI',4),rep('DOX',4),rep('OFF',4)),
                              ndim = 2, method = 'PLS-DA')

# Some of the outputs
resultsPLSDA@R2X
resultsPLSDA@R2Y
resultsPLSDA@Q2
resultsPLSDA@DQ2
resultsPLSDA@Prediction$confusionMatrix

# Plot saliences
saliencesPLSDA <- resultsPLSDA@Saliences %>%
  as.data.frame() %>%
  mutate(dataset = rownames(.)) %>%
  pivot_longer(cols = c('CC1','CC2'),
               names_to = 'CC',
               values_to = 'Salience')

ggplot(data = saliencesPLSDA,
       aes(x = CC, y = Salience, group = dataset )) +
  geom_bar(stat = 'identity', position = 'dodge',
           aes(fill = dataset)) +
  theme_minimal() +
  labs(title = 'ComDim Saliences')
```

```
# Plot scores
scoresTable <- MakeComDimScoresTable(model = resultsPLSDA)
scoresTable_wider <- scoresTable %>%
  mutate(sample.type = case_when(grepl('DOX', sample.id) ~ 'DOX',
                                 grepl('NI', sample.id) ~ 'NI',
                                 grepl('OFF', sample.id) ~ 'OFF')) %>%
  dplyr::select(sample.id, sample.type, block.name, scores.type.dim, value) %>%
  dplyr::group_by(sample.id, sample.type, scores.type.dim, block.name) %>%
  pivot_wider(names_from = scores.type.dim, values_from = value)

ggplot(data = scoresTable_wider) +
  geom_point(aes(x = T.scores1, y = T.scores2,
                 color = sample.type, shape = block.name)) +
  geom_point(aes(x = Q.scores1, y = Q.scores2,
                 fill = sample.type, shape = block.name),
             size = 3, shape = 24, color = 'black') +
  theme_minimal() +
  labs(title = 'ComDim scores - PLSDA', x = 'CC1', y = 'CC2')
```

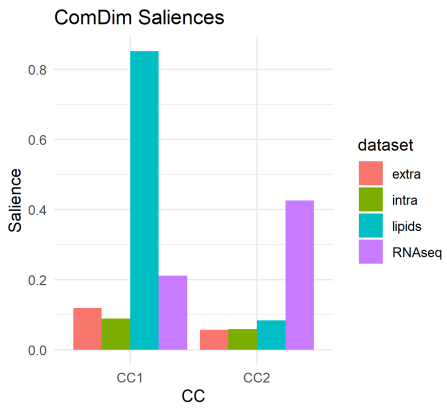


Figure 7.1: Saliences

7.3 ComDim-kernel-OPLS

```
source('./R/ComDim_KOPLS_MB.R')
resultsKOPLSDA <- ComDim_KOPLS_MB(allMB, y= c(rep('NI',4),rep('DOX',4),rep('OFF',4)),
                                  max.ort = 2, method = 'k-OPLS-DA', nrcv = 5)
```

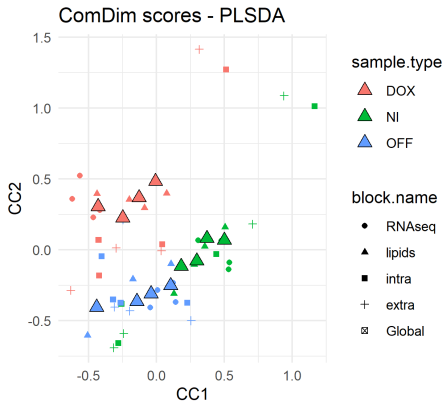


Figure 7.2: Scores

7.4 Other extensions

The functions `ComDim_Exploratory_MB()` and `ComDim_y_MB()` can be employed to use customized versions of Com-Dim for exploratory and regression/discriminant purposes. In both functions, the parameter `FUN` allows the user to run ComDim with their chemometric method of preference.

7.4.1 Example of ComDim-ICA

The function used to execute **Independent Component Analysis (ICA)** was `ica()` from the `ica` package. In order to make the `ComDim_Exploratory_MB()` function to understand the output of `ica()`, we embedded it into another function that returns the source estimates (representative of the samples information, analogous to the PCA scores) named `fun.ICA`.

```
library(ica)

fun.ICA <- function(W, ndim,...){
  # W is the concatenated MB.
  # ndim is the number of components.
  # X and nc are the arguments of ica::ica.
  result <- ica::ica(W, ndim)
  sources <- result$S
  # The function must return the source estimates
  # The analogue to the PCA scores for ICA.
  return(sources)
}
resultsICA <- ComDim_Exploratory_MB(allMB, ndim = 2,
```

```
                                    FUN = fun.ICA,
                                    method = 'ICA')
```

7.4.2 Example of ComDim-PLS (version ropls)

The function used to execute **PLS** here was `opls()`, obtained from the `ropls` package. In this case, `fun.PLS` is used to capture the output from the PLS analysis (scores,P,W,U,Q,y). `type=regression` is set to calculate R2Y, Q2,... as with `ComDim_Pls_MB()`.

```
library(ropls) # From bioconductor
source('./R/ComDim_y_MB.R')
fun.PLS <- function(W, y, ndim,...){
  # W is the concatenated MB.
  # ndim is the number of components.
  # X and nc are the arguments of ica::ica.
  output <- list()
  result <- ropls::opls(x = W, y = y, predI = ndim,
                        fig.pdfC = 'none',
                        info.txtC = 'none')
  # The returning object must be a list containing the following 6 elements:
  output$scores <- result@scoreMN[,1]
  output$P <- result@loadingMN[,1]
  output$W <- result@weightMN[,1]
  output$U <- result@uMN[,1]
  output$Q <- result@cMN[,1]
  output$y <- result@suppLs$yModelMN # To evaluate if y is transformed within opls. (it is!)
  return(output)
}
resultsPLS <- ComDim_y_MB(allMB,
                          y =c(rep(1,4),rep(5,4),rep(10,4)),
                          ndim = 2,
                          type = 'regression',
                          orthogonalization = FALSE,
                          FUN = fun.PLS,
                          method = 'PLS(ropls)')
```

7.4.3 Example of ComDim-PLSDA (version ropls)

For ComDim-PLS-DA, we can use the same function `fun.PLS` as before, but we need to specify that the `type` of the method is `'discriminant'` and we need to provide the sample classes in `y`:

```
resultsPLSDA <- ComDim_y_MB(allMB,
                            y =c(rep('NI',4),rep('DOX',4),rep('OFF',4)),
                            ndim = 2,
                            type = 'discriminant',
                            orthogonalization = FALSE,
                            FUN = fun.PLS,
                            method = 'PLSDA(ropls)')
```

7.4.4 Example of ComDim-OPLSDA (version ropls)

For ComDim-OPLSDA, the function in `FUN` needs to capture additional outputs related to the orthogonal components. See the example below:

```
# We will apply ComDim-OPLSDA on a subset with 2 classes only.
allMB_small <- FilterSamples_MB(allMB,
                                getSampleNames(allMB)[c(1:4,9:12)])
fun.OPLSDA <- function(W, y, ndim,...){
  # W is the concatenated MB.
  # ndim is the number of components.
  output <- list()
  Y <- c(-1,1)[apply(y, 1, function(x) match(1,x))]
  # Y must be a vector for ropls
  result <- ropls::opls(x = W, y = Y,
                        predI = 1,
                        orthoI = NA, # The number of orthogonal components
                                     # is optimized for every block.
                        fig.pdfC = 'none',
                        info.txtC = 'none')
  # The returning object must be a list containing the following 9 elements:
  output$scores <- result@scoreMN[,1]
  output$P <- result@loadingMN[,1]
  output$W <- result@weightMN[,1]
  output$U <- result@uMN[,1]
  output$Q <- result@cMN[,1]
  output$Q <- c(-output$Q,output$Q)
  # Y has two columns (one per class): -1 and 1.
  output$y <- result@suppLs$yModelMN
  # To evaluate if y is transformed within opls. (it is!)
  output$orthoscores <- result@orthoScoreMN
  output$orthoP <- result@orthoLoadingMN
  output$ort <- result@summaryDF$ort # Number of orthogonal components
  return(output)
}
resultsOPLSDA <- ComDim_y_MB(allMB_small,
```

```
                                y =c(rep('NI',4),rep('OFF',4)),
                                ndim = 1,
                                type = 'discriminant',
                                orthogonalization = TRUE,
                                FUN = fun.OPLSDA,
                                method = 'OPLSDA(ropls)')
```