Introduction to the proteomeR package

Arne Smits

10 March 2017

Package version: proteomeR 0.1.0

Contents

1 Overview of the analysis

This analysis workflow provides an integrated solution for robust and reproducible analysis of proteomics data. It requires tabular input (e.g. txt files) which are generated by quantitative analysis softwares of raw mass spectrometry data, such as MaxQuant or IsobarQuant.
Functions are provided for data preparation, data filtering, variance normalization and imputation of missing values, as well as statistical testing of differential enriched/expressed proteins. It also includes tools to check intermediate steps within the workflow, such as normalization and missing values imputation. Finally, visualization tools are provided to explore the results, including heatmap, volcano plot and barplot representations.
For scientists with limited experience in R, the package also entails wrapper functions that entail the complete analysis workflow and generate a report (see the chapter on Wrapper functions). Even easier to use are the interactive Shiny apps that are provided by the package (see the chapter on Shiny apps).

2 Installation

Install and load the package

```
devtools::install_git("git@git.embl.de:smits/proteomeR.git")
library("proteomeR")
```

3 Interactive analysis using the proteomeR Shiny apps

The package contains two *shiny* apps, which allow for interactive analysis entailing the entire workflow described below. These are especially relevant for users with limited or no experience in R. Currently, there are two different apps The first for label-free quantification (LFQ)-based analysis (output from MaxQuant) and the second for tandem-mass-tags (TMT)-based analysis (output from IsobarQuant). To run the apps, simply run this single command:

```
# For LFQ analysis
run_app("LFQ")

# For TMT analysis
run_app("TMT")
```

4 Example dataset: Ubiquitin interactors

We analyze a proteomics dataset in which Ubiquitin-protein interactors were characterized (Zhang et al. Mol Cell 2017). The raw mass spectrometry data were first analyzed using MaxQuant (Cox and Mann, Nat Biotech 2007).

4.1 Loading of the data

```
# Loading two packages required for data handling
library(tidyverse)
library(magrittr)

# The data is provided with the package
data <- UbIA_MS

# We filter for contaminant proteins and decoy database hits, which are indicated by "+"
# in the columns "Potential.contaminants" and "Reverse", respectively.
data %<>% filter(Reverse != "+", Potential.contaminant != "+")
```

The data.frame has the following dimensions:

```
dim(data)
## [1] 2941   35
```

The data.frame has the following column names:

```
colnames(data)
##  [1] "Protein.IDs"             "Majority.protein.IDs"
##  [3] "Protein.names"           "Gene.names"
##  [5] "Fasta.headers"           "Peptides"
##  [7] "Razor...unique.peptides" "Unique.peptides"
##  [9] "Intensity.4Ubi_1"        "Intensity.4Ubi_2"
## [11] "Intensity.4Ubi_3"        "Intensity.6Ubi_1"
## [13] "Intensity.6Ubi_2"        "Intensity.6Ubi_3"
## [15] "Intensity.Con_1"         "Intensity.Con_2"
## [17] "Intensity.Con_3"          "Intensity.Mono_1"
## [19] "Intensity.Mono_2"         "Intensity.Mono_3"
## [21] "LFQ.intensity.4Ubi_1"     "LFQ.intensity.4Ubi_2"
## [23] "LFQ.intensity.4Ubi_3"     "LFQ.intensity.6Ubi_1"
## [25] "LFQ.intensity.6Ubi_2"     "LFQ.intensity.6Ubi_3"
## [27] "LFQ.intensity.Con_1"      "LFQ.intensity.Con_2"
## [29] "LFQ.intensity.Con_3"      "LFQ.intensity.Mono_1"
## [31] "LFQ.intensity.Mono_2"     "LFQ.intensity.Mono_3"
## [33] "Only.identified.by.site" "Reverse"
## [35] "Potential.contaminant"
```

The "LFQ.intensity" columns will be used for subsequent analysis.

4.2 Data preparation

The dataset has unique Uniprot identifiers, however those are not immediately informative. The associated gene names are informative, however these are not always unique.

```
# Are there any duplicated gene names?
data$Gene.names %>% duplicated(.) %>% any(.)
## [1] TRUE

# Make a table of duplicated gene names
data %>% group_by(Gene.names) %>% summarize(frequency = n()) %>%
  arrange(desc(frequency)) %>% filter(frequency > 1)
## # A tibble: 51 × 2
##    Gene.names frequency
##         <chr>     <int>
## 1                     7
## 2       ATXN2         4
## 3      ATXN2L         4
## 4         SF1         4
## 5       HSPA8         3
## 6       RBM33         3
## 7        UGP2         3
## 8      ACTL6A         2
## 9      BCLAF1         2
## 10       BRAP         2
## # ... with 41 more rows
```

For further analysis these proteins must get unique names. Additionally, some proteins do not have an annotated gene name and for those we will use the Uniprot ID.

```
# Make unique names using the annotation in the "Gene.names" column as primary names
# and the annotation in "Protein.IDs" as name for those that do not have an gene name.
data_unique <- unique_names(data, "Gene.names", "Protein.IDs", delim = ";")
## Joining, by = c("Protein.IDs", "Gene.names")

# Are there any duplicated names?
data$name %>% duplicated(.) %>% any(.)
## [1] FALSE
```

4.3 Generate an ExpressionSet object

Many Bioconductor packages use ExpressionSet objects as input and/or output. This class of objects contains and coordinates the actual (expression) data, information on the samples as well as feature annotation. We can generate the ExpressionSet object from our data using two different approaches. We can extract sample information directly from the column names or we add sample information using an experimental design template. For our example dataset, an experimental design is included in the package.

label	sample	replicate
4Ubi_1	4Ubi	1
4Ubi_2	4Ubi	2
4Ubi_3	4Ubi	3
6Ubi_1	6Ubi	1
6Ubi_2	6Ubi	2
6Ubi_3	6Ubi	3
Con_1	Con	1
Con_2	Con	2
Con_3	Con	3
Mono_1	Mono	1
Mono_2	Mono	2
Mono_3	Mono	3

```
# Generate an ExpressionSet object using sample information from the column names
LFQ_columns <- grep("LFQ.", colnames(data_unique)) # get LFQ column numbers
data_exprset <- into_exprset(data_unique, LFQ_columns)

# Generate an ExpressionSet object with an experimental design
LFQ_columns <- grep("LFQ.", colnames(data_unique)) # get LFQ column numbers
experimental_design <- ExpDesign_UbIA_MS
data_exprset <- into_exprset_expdesign(data_unique, LFQ_columns, experimental_design)

# Let's have a look at the ExpressionSet object
data_exprset
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 2941 features, 12 samples
##   element names: exprs
## protocolData: none
## phenoData
##   sampleNames: X4Ubi_1 X4Ubi_2 ... Mono_3 (12 total)
##   varLabels: label ID sample replicate
##   varMetadata: labelDescription
## featureData
##   featureNames: RBM47 UBA6 ... X6RHB9 (2941 total)
##   fvarLabels: Protein.IDs Majority.protein.IDs ... ID (25 total)
##   fvarMetadata: labelDescription
## experimentData: use 'experimentData(object)'
## Annotation:
```

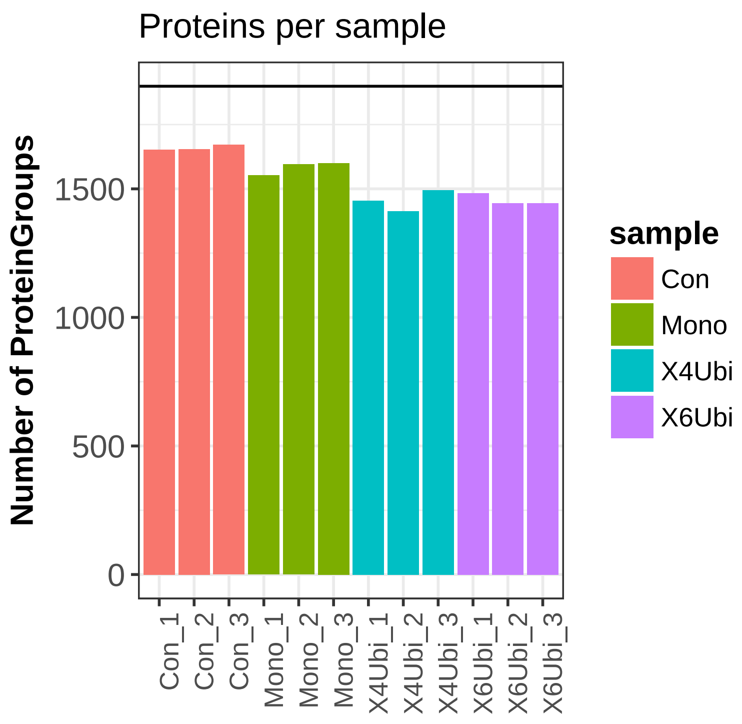
4.4 Filter for missing values

The dataset contains proteins which are not quantified in all replicates. Some proteins are even only deteced in a single replicate. This leaves our dataset with missing values, which need to be imputed. However, this should not be done for proteins that contain too many missing values. Therefore, we first filter out proteins that contain to many missing values. This is done by setting the threshold for the allowed number of missing values per condition in the *miss_val_filter* function.

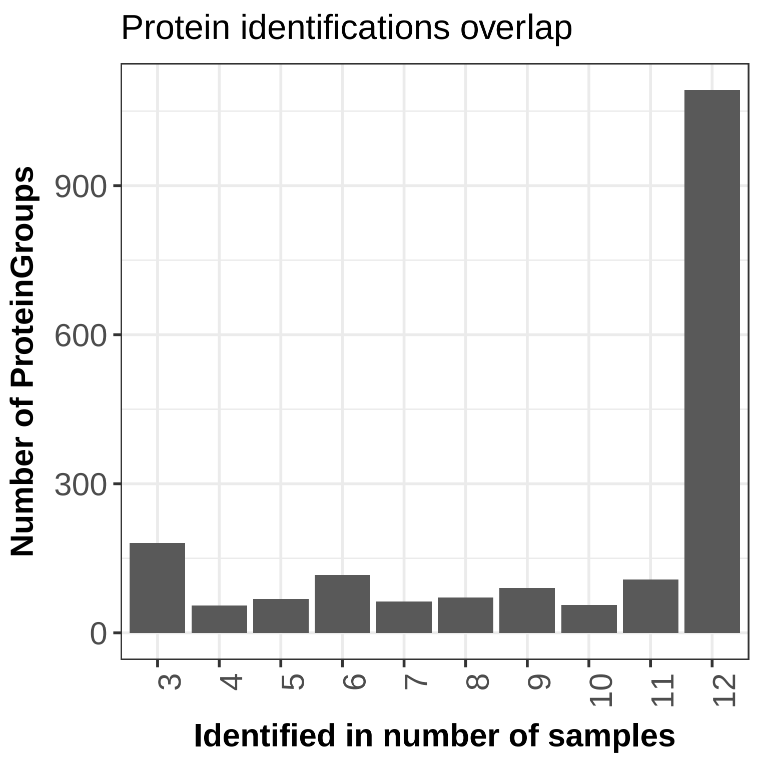
```r
# Filter for proteins that are identified in all replicates of at least one sample.
data_filt <- miss_val_filter(data_exprset, thr = 0)

# Less stringent filtering:
# Filter for proteins that are identified in 2 out of 3 replicates of at least one sample.
data_filt2 <- miss_val_filter(data_exprset, thr = 1)
```

After filtering the number of identified proteins per sample can be plotted as well as the overlap in identifications between samples.

```r
# Plot a barplot of the number of identified proteins per samples
plot_numbers(data_filt)
```



```r
# Plot a barplot of the protein identification overlap between samples
plot_frequency(data_filt)
```



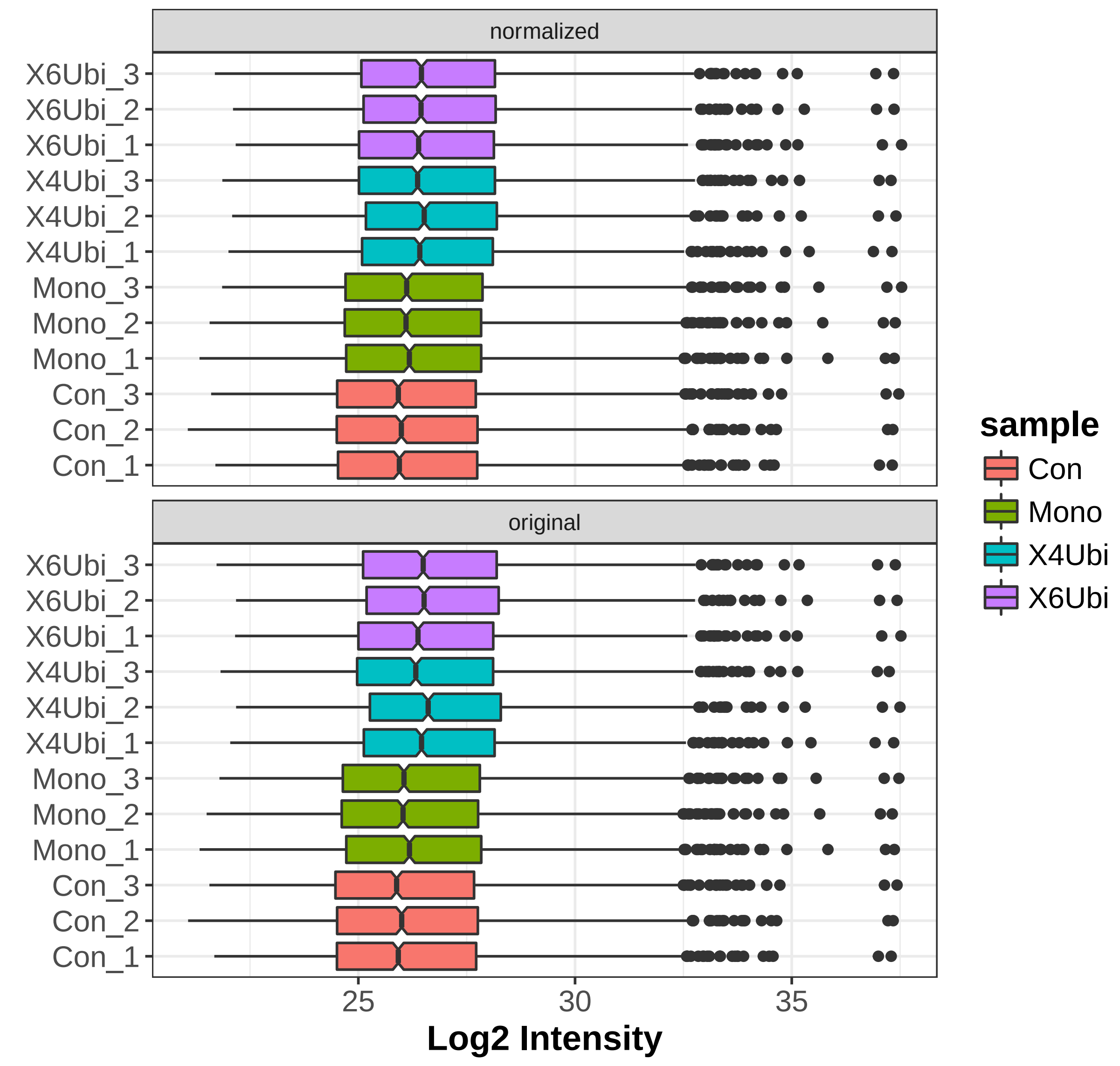
4.5 Normalization

The data is background corrected and normalized by variance stabilization transformation (*vsn*).

```
# Normalize the data
data_norm <- norm_vsn(data_filt)
## vsn2: 1899 x 12 matrix (1 stratum). Please use 'meanSdPlot' to verify the fit.
```

The normalization can be inspected by checking the distributions of the samples before and after normalization.

```
# Visualize normalization by boxplots for all samples before and after normalization
plot_norm(data_filt, data_norm)
```

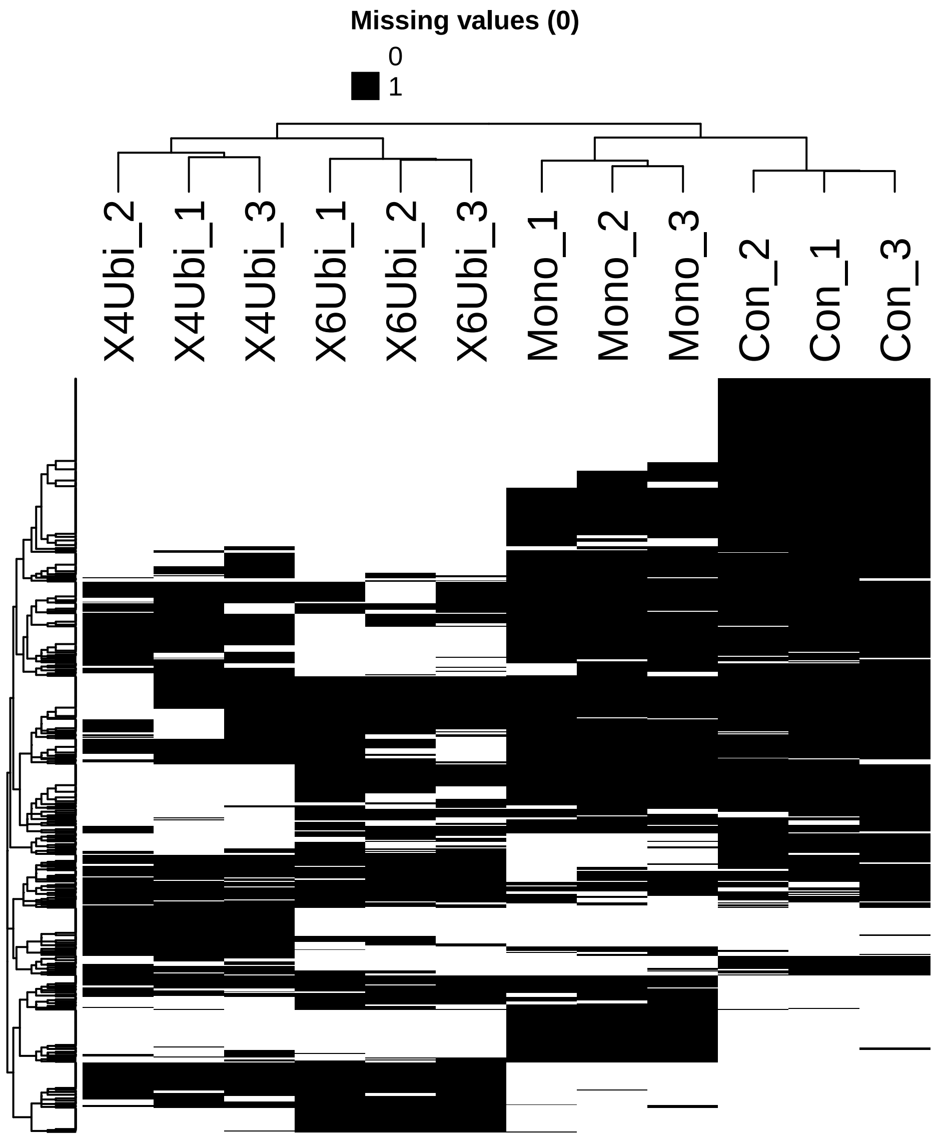


4.6 Imputate data for missing values

The remaining missing values in the dataset need to be imputed. The data can be missing at random (MAR), for example if proteins are quantified in some replicates but not in others, or missing not at random (MNAR), for example if proteins which are quantified in some replicates but not in others (e.g. in all replicates of the control). MNAR can indicate that the proteins are below the detection limit in specific samples, which could be very well the case for, for example, affinity enrichments. For these different conditions, different imputation methods have to be used, as described in the *MSnbase* vignette and more specifically in the *impute* function descriptions.

First, the missingness pattern of the data is explored by plotting a heatmap indicating whether values are missing (0) or not (1).

```
# Plot a heatmap of proteins with missing values
plot_missval(data_filt)
```



The example dataset is an affinity enrichment dataset of ubiquitin interactors, which is likely to have proteins which are below the detection limit in specific samples. These can be proteins that are specifically enriched in the ubiquitin purifications, but are not enriched in the controls samples, or vice versa. This expectation is also reflected in the missingness heatmap; missing values are highly biased to specific samples.

The imputation method to be used depends on the missingness pattern of the data (see the *MSnbase* vignette and more specifically in the *impute* function descriptions for more information). Values which are missing not at random (MNAR) should be imputed by left-censored imputation method, such as the quantile regression-based left-consored function ("QRILC") or the manually defined left-shifted distribution function. In contrast, data missing at random (MAR) should be imputed with methods such as k-nearest neighbour ("knn") or maximum likelihood ("MLE") functions.

As the example data has values missing not at random, QRILC imputation is the method of choice.

```
# Impute missing data using a quantile regression-based left-censored imputation method (QRILC)
# (for data not missing at random)
data_imp_QRILC <- imputation_MSn(data_norm, fun = "QRILC")

# Impute missing data using a manually defined left-shifted distribution
# (for data not missing at random)
```

```
data_imp_man <- imputation_perseus(data_norm, shift = 1.8, scale = 0.3)

# Impute missing data using the k-nearest neighbour approach
# (for data missing at random)
data_imp_knn <- imputation_MSn(data_norm, fun = "knn")

# Impute missing data using the k-nearest neighbour approach
#(for data missing at random)
data_imp_MLE <- imputation_MSn(data_norm, fun = "MLE")
```

4.7 Differential enrichment analysis

Linear models combined with empherical Bayes statistics is used for the analysis of differential enrichment (or differential expression for analysis on expression levels). The *linear_model* wrapper function introduced here uses *limma* and automatically generates the contrasts to be tested. For the contrasts generation, the control sample has to be specified. Additionally, the types of contrasts to be produced need to be indicated, allowing the generation of all possible comparisons ("all") or the generation of contrasts of every sample versus control ("control").

```
# Differential enrichment analysis  based on linear models and empherical Bayes statistics
# Test every sample versus control
data_lm <- linear_model(data_imp_QRILC, control = "Con", type = "control")
## Tested contrasts:
## [1] "Mono - Con"  "X4Ubi - Con" "X6Ubi - Con"

# Test all possible comparisons of samples
data_lm_all_contrasts <- linear_model(data_imp_QRILC, control = "Con", type = "all")
## Tested contrasts:
## [1] "X4Ubi - X6Ubi" "X4Ubi - Con"   "X4Ubi - Mono"  "X6Ubi - Con"
## [5] "X6Ubi - Mono"  "Mono - Con"
```

An ANOVA and post-hoc Tukey Honest Significant Differences analysis is also implemented, although the use of *linear_models* based on *limma* is recommended.

```
# Differential enrichment analysis  based on ANOVA and Tukey
data_anova <- anova_tukey(data_imp_QRILC, control = "Con", type = "control")
```

Finally, significant proteins are defined by user-defined cutoffs.

```
# Denote significant proteins based on user defined cutoffs
data_sign <- cutoffs(data_lm, alpha = 0.05, lfc = 1)

# Number of significant proteins
results(data_sign) %>% filter(sign == "+") %>% nrow()
## [1] 186
```

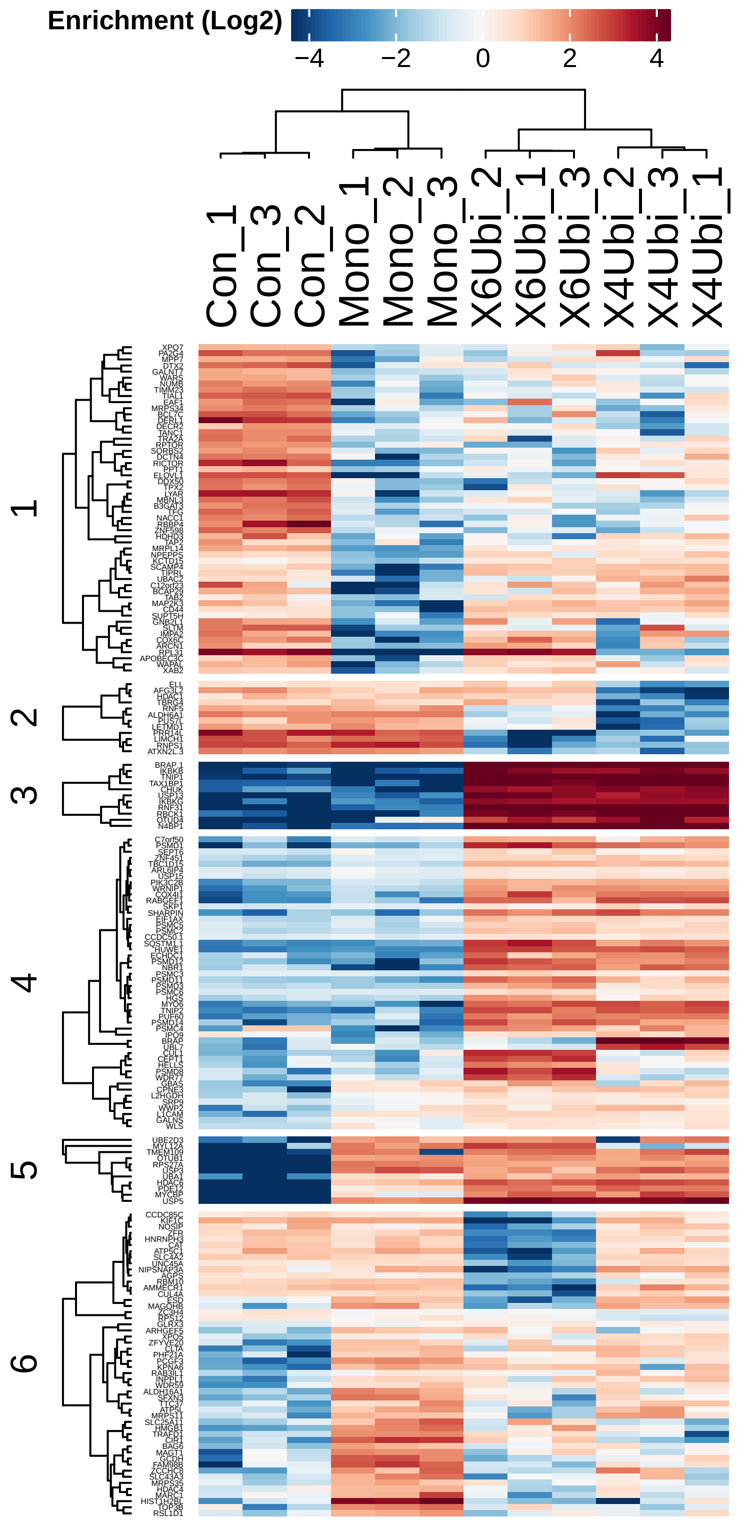
4.8 Visualization of the results

The data resulting from the previous analysis can easily be visualized by a number of functions. These visualizations assist in the determination of the optimal cutoffs to be used, highlight the most interesting samples and contrasts, and pinpoint differentially enriched/expressed proteins.

4.8.1 Heatmap of all significant proteins

The heatmap representation gives an overview of all significant proteins (rows) in all samples (columns). This allows to see general trends, e.g. one sample really different or one replicate which is off in all samples. Additionally, the clustering of samples (columns) can indicate closer related samples and clustering of proteins (rows) indicates similarly behaving proteins. The proteins are additionally clustered by k-means clustering and the number of clusters can be defined by k.

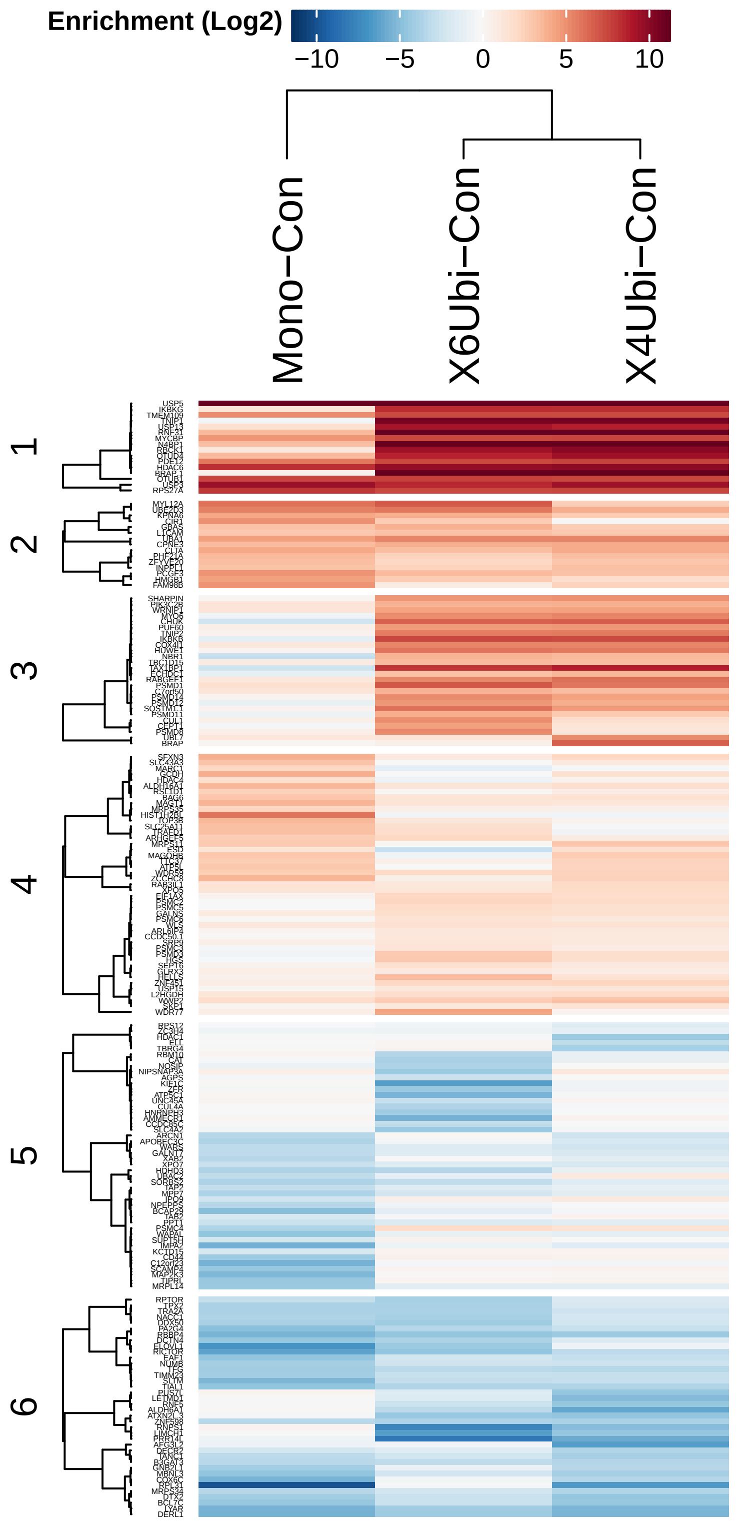
```r
# Plot a heatmap of all significant proteins with the data centered per protein
plot_heatmap(data_sign, type = "centered", k = 6, col_limit = 4, labelsize = 3)
```



The heatmap shows a really nice clustering of the replicates and indicates that 4Ubi and 6Ubi enrich a similar repetoire of proteins. The k-means clustering of proteins (general clusters of rows) nicely separates protein classes with different binding behaviours.

Alternatively, a heatmap can be plotted with the direct sample comparisons (columns) for all significant proteins (rows). In this example this emphasises the enrichment of ubiquitin interactors over the control sample.

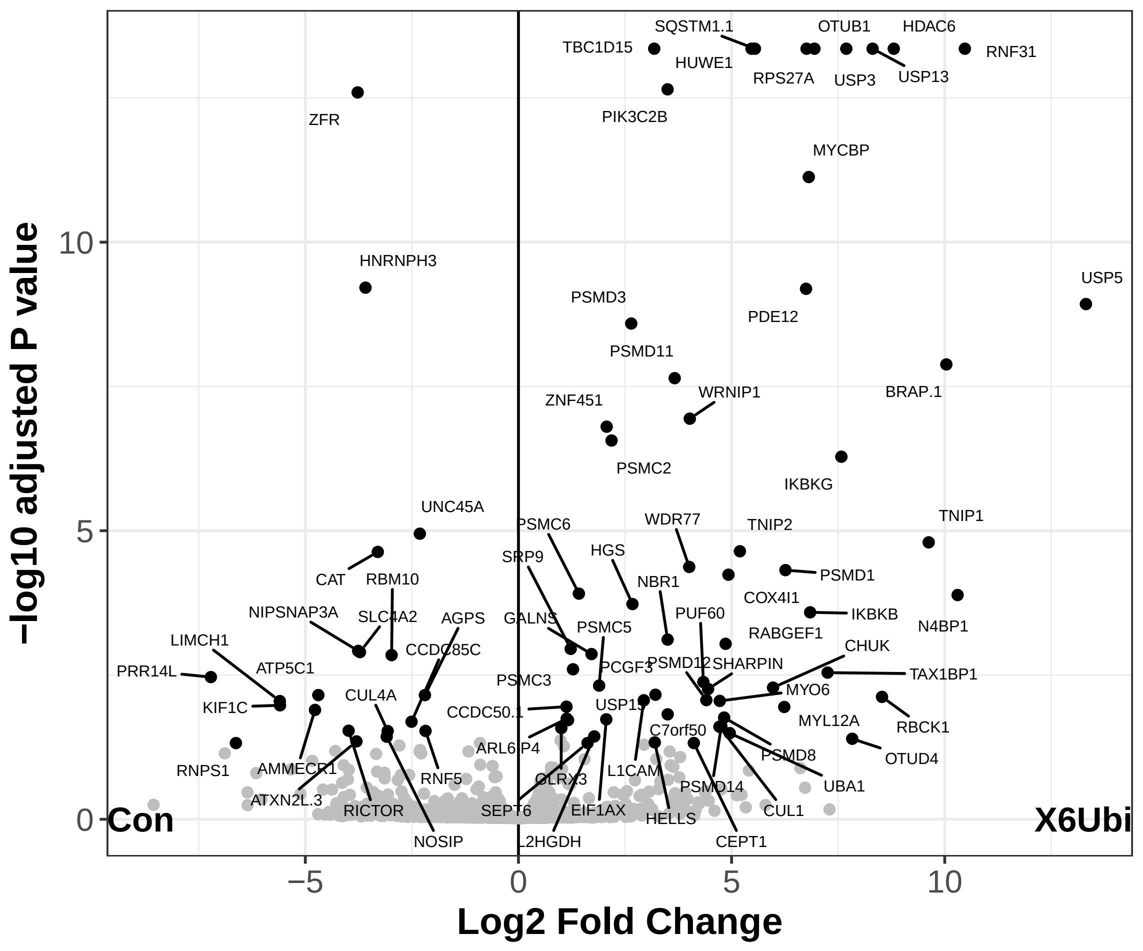
```
# Plot a heatmap of all significant proteins (rows) and the tested contrasts (columns)
plot_heatmap(data_sign, type = "contrast", k = 6, col_limit = 10, labelsize = 3)
```



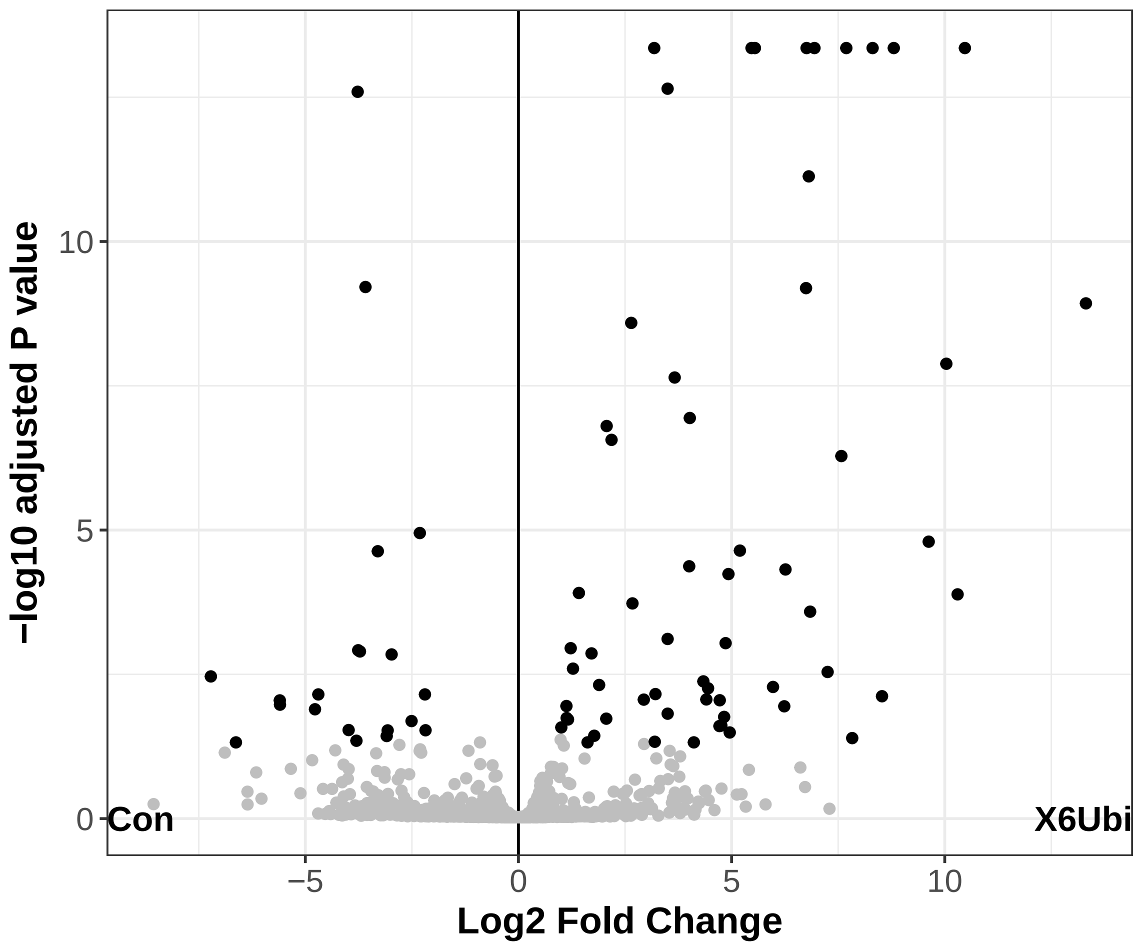
4.8.2 Volcano plots of specific contrasts

Volcano plots can be used to visualize a specific contrast (comparison between two samples). This allows to inspect the enrichment of proteins between the two samples (x axis) and their corresponding adjusted p value (y axis).

```
# Plot a volcano plot for the contrast "X6Ubi vs Con""
volcano(data_sign, contrast = "X6Ubi.Con", labelsize = 2)
```



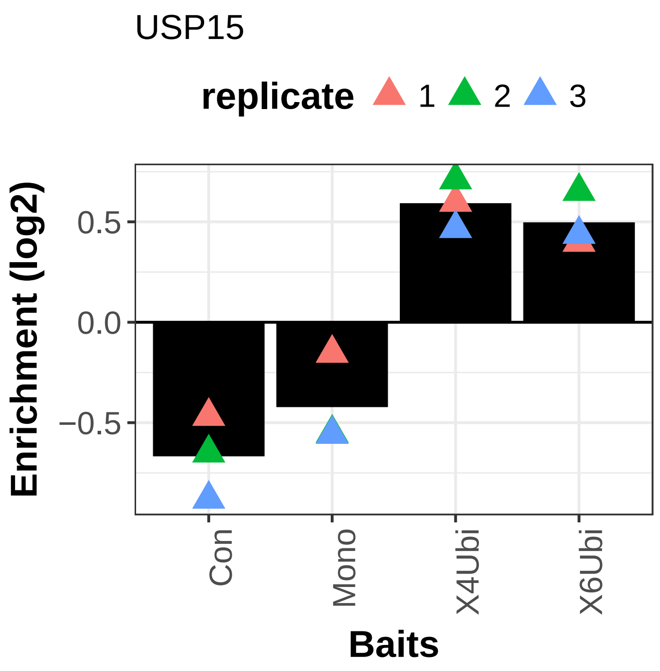
```
# Plot a volcano plot for the contrast "X6Ubi vs Con"" without labels
volcano(data_sign, contrast = "X6Ubi.Con", add_names = FALSE)
```



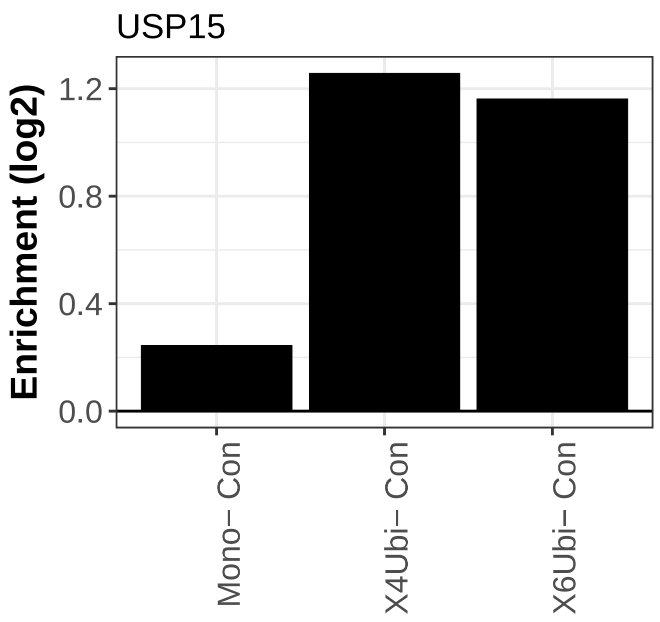
4.8.3 Barplots of a protein of interest

It can also be useful to plot the data of a single protein, for example if this protein is of special interest.

```
# Plot a barplot for the protein USP15 with the data centered
single_prot_plot(data_sign, protein = "USP15", type = "centered")
```



```
# Plot a barplot for the protein USP15
single_prot_plot(data_sign, protein = "USP15", type = "contrast")
```



4.9 Results table

To extract a table containing the essential results, the *results* function can be used.

```
# Generate a results table
data_results <- results(data_sign)
```

The resulting table contains the following columns:

```
# Column names of the results table
colnames(data_results)
##  [1] "name"             "ID"                "Mono-Con_p.adj"
##  [4] "X4Ubi-Con_p.adj" "X6Ubi-Con_p.adj" "Mono.Con_sign"
##  [7] "X4Ubi.Con_sign"   "X6Ubi.Con_sign"   "sign"
## [10] "Mono-Con_ratio"   "X4Ubi-Con_ratio" "X6Ubi-Con_ratio"
## [13] "Con_centered"     "Mono_centered"    "X4Ubi_centered"
## [16] "X6Ubi_centered"
```

Of these columns, the **p.adj** columns contain the adjusted p values for the contrast as depicted in the column name. The **ratio** columns contain the average log2 fold changes. The **sign** columns indicate whether the protein is differentially enriched/expressed, as defined by the chosen cutoffs. The **centered** columns contain the average log2 enrichment values scaled by protein-wise centering.

5 Wrapper functions for the entire workflow

The package also entails wrapper functions that entail the complete analysis workflow and generate a report.

5.1 LFQ analysis

```
# The data is provided with the package
data <- UbIA_MS

# The wrapper function performs the full analysis
data_results <- LFQ(data, fun = "QRILC", control = "Con_", type = "control", alpha = 0.05, lfc = 1)
## Joining, by = c("Protein.IDs", "Gene.names")
## vsn2: 1899 x 12 matrix (1 stratum). Please use 'meanSdPlot' to verify the fit.
## Tested contrasts:
## [1] "Mono_ - Con_"  "X4Ubi_ - Con_" "X6Ubi_ - Con_"
```

This wrapper produces two reports (pdf and html), which are saved in a generated “Report” folder. Additionally, the resulting object contains the *results table* (data.frame object) and the *full data set* (ExpressionSet object).

```
# data_results contains two objects
names(data_results)
## [1] "results" "data"

# Colnames of the results object
colnames(data_results$results)
##  [1] "name"               "ID"                 "Mono_-Con__p.adj"
##  [4] "X4Ubi_-Con__p.adj" "X6Ubi_-Con__p.adj" "Mono_.Con__sign"
##  [7] "X4Ubi_.Con__sign"   "X6Ubi_.Con__sign"   "sign"
## [10] "Mono_-Con__ratio"   "X4Ubi_-Con__ratio" "X6Ubi_-Con__ratio"
## [13] "Con__centered"      "Mono__centered"     "X4Ubi__centered"
## [16] "X6Ubi__centered"

# data object
data_results$data
## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 1899 features, 12 samples
##   element names: exprs
## protocolData: none
## phenoData
##   sampleNames: X4Ubi_1 X4Ubi_2 ... Mono_3 (12 total)
##   varLabels: ID replicate sample
##   varMetadata: labelDescription
## featureData
##   featureNames: AAAS ABCB7 ... ZYX (1899 total)
##   fvarLabels: name Protein.IDs ... sign (36 total)
##   fvarMetadata: labelDescription
## experimentData: use 'experimentData(object)'
## Annotation:
```

To save the results table in the Reports folder:

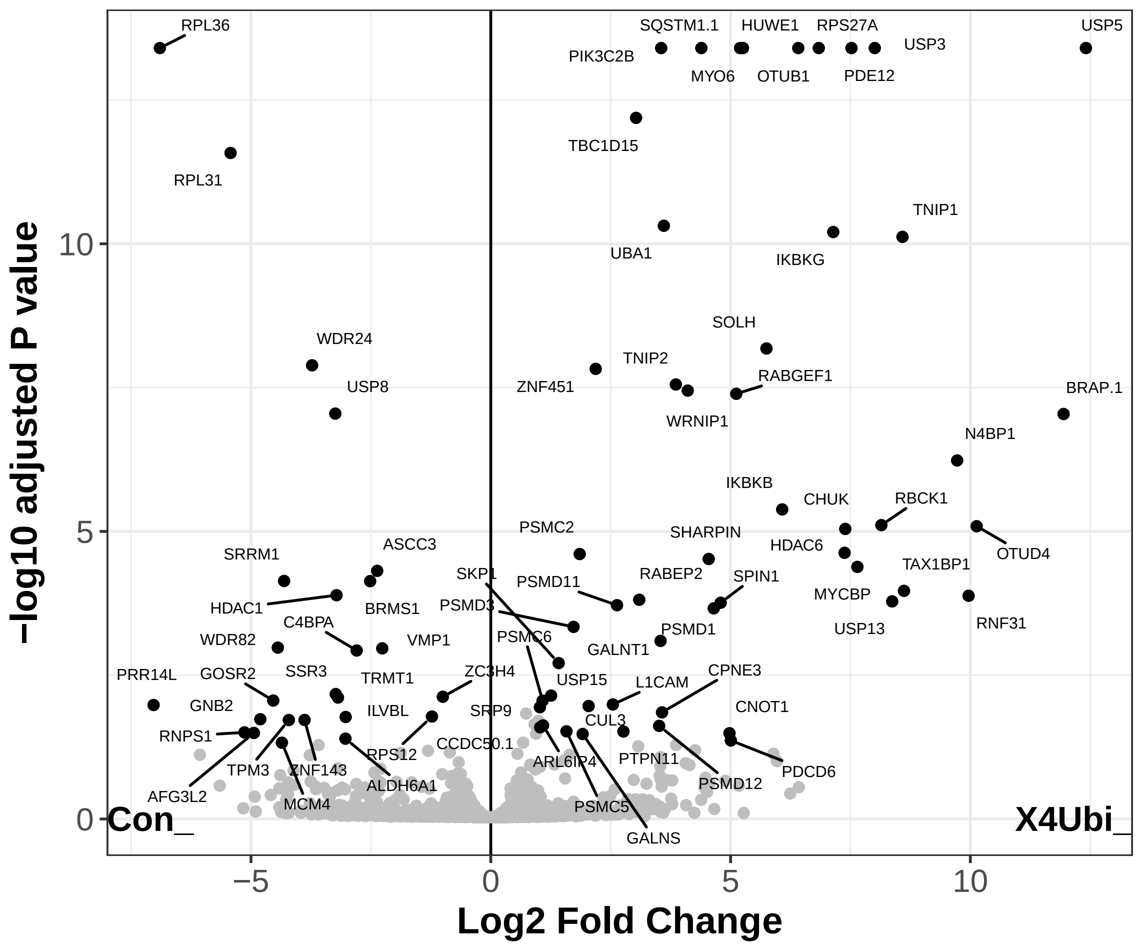
```
write.table(data_results$results, paste(getwd(), "Report", "results.txt", sep = "/"),
    row.names = FALSE, sep = "\t")
```

The results table can be explored by selecting the *$results* object

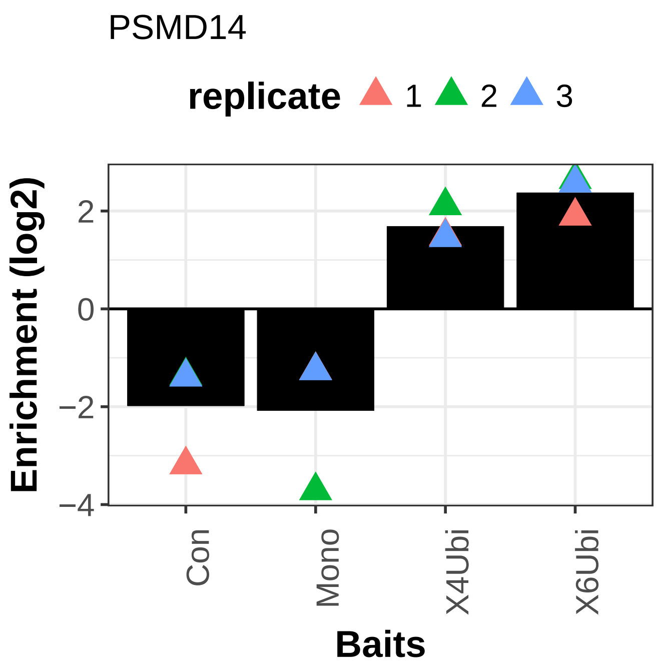
```
# Extract the results table
results_table <- data_results$results

# Number of significant proteins
results_table %>% filter(sign == "+") %>% nrow()
## [1] 194
```

The full data can be used for the plotting functions as desribed in the chapter “Visualization of the results”.

```
# Extract the full data object
full_data <- data_results$data

# Use the full data to generate a volcano plot
volcano(data_results$data, contrast = "X4Ubi_.Con_", labelsize = 2)
```



```r
# Use the full data to generate a barplot
single_prot_plot(data_results$data, protein = "PSMD14", type = "centered")
```



5.2 TMT analysis

```
# Need example data
TMTdata <- example_data
Exp_Design <- example_Exp_Design

# The wrapper function performs the full analysis
TMTdata_results <- TMT(data, expdesign = Exp_Design, fun = "QRILC",
    control = "Con_", type = "control", alpha = 0.05, lfc = 1)
```