Package: MRManalyzeR (via r-universe)

November 6, 2026

Type Package

Title Process and Analyse Targeted LC-MS Lipidomics and Metabolomics Data

Version 0.99.8

Description Reproducible post-acquisition processing of targeted LC-MS/MS lipidomics and metabolomics results exported from Waters TargetLynx or from Skyline, or supplied as a plain sample-by-analyte matrix. Reads the quantitative result tables, integrates them with sample and feature metadata, and applies configurable signal filtering (SNR, LOD or LOQ), blank filtering, batch correction, normalisation, internal-standard and volume adjustment of vendor-reported concentrations, and missing-value imputation. Each step is an exported function acting on a struct DatasetExperiment; a complete workflow can additionally be driven from a single YAML configuration via runMRManalyzeR(), which also renders self-contained HTML data-quality and statistical reports. Chromatographic peak detection, peak integration and calibration-curve fitting from raw mass-spectrometry data are outside its scope.

License MIT + file LICENSE

Encoding UTF-8

biocViews Metabolomics, Lipidomics, MassSpectrometry, QualityControl, Normalization, BatchEffect, Visualization, WorkflowStep

URL https://github.com/MJS-708/MRManalyzeR

BugReports https://github.com/MJS-708/MRManalyzeR/issues

Depends R (>= 4.5.0), struct

Imports dplyr, tidyr, tibble, yaml, openxlsx, janitor, rmarkdown, methods, ggplot2, ggnewscale, rlang, grDevices, pheatmap, withr, DT, stats, utils

Suggests pmp, structToolbox, pls, SummarizedExperiment, ggrepel, plotly, htmltools, knitr, cowplot, gridExtra, BiocStyle, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

Config/pak/sysreqs cmake make libicu-dev libuv1-dev libssl-dev zlib1g-dev

Repository https://bioc.r-universe.dev

Date/Publication 2026-11-05 13:24:58 UTC

RemoteUrl https://github.com/bioc/MRManalyzeR

RemoteRef HEAD

RemoteSha 15a43ca65d84ead4e947bcb3de22991740469859

Contents

addCVMetrics . . . 3
adjustConcentration . . . 4
assembleDataset . . . 5
combineDatasets . . . 6
correctBatch . . . 8
filterBlanks . . . 10
filterFeatures . . . 12
filterSamples . . . 13
imputeMissing . . . 15
loadConfig . . . 16
loadDataset . . . 17
mrm_validation . . . 18
mrmPlotHeatmap . . . 19
mrmPlotPCA . . . 21
mrmRunExample . . . 22
mrmRunPCA . . . 23
mrmValidateConfig . . . 25
mrmValidateDesign . . . 26
mrmValidateInput . . . 27
newConfig . . . 28
normaliseMatrix . . . 29
plotBoxplot . . . 30
plotCorrelation . . . 32
plotDrift . . . 33
plotGroupHeatmap . . . 35
plotLoadings . . . 37
plotQQ . . . 38
plotVolcano . . . 40
print.mrm_validation . . . 41
processDataset . . . 42
readPeakMatrix . . . 46
readSkyline . . . 47
readTargetLynx . . . 48

runMRManalyzeR 49
runMRManalyzeRCombine 51
runStats 53
subsetDataset 54
summariseFeatures 55
summariseGroups 56

Index **58**

`addCVMetrics` *Add per-compound CV metrics to a dataset*

Description

Computes the per-feature coefficient of variation (100 * sd / mean) over the QC injections (‘CV_QC‘) and over the biological samples (‘CV_sample‘), plus their ratio (‘CV_sample_vs_QC‘), and adds all three as columns of ‘variable_meta‘, matched to each feature by row. QC and sample rows are identified from ‘sample_type_head‘ in ‘sample_meta‘.

Usage

```
addCVMetrics(
  de,
  sample_type_head = "Sample_type",
  qc_label = "QC",
  sample_labels = "Sample"
)
```

Arguments

`de`	A ‘struct::DatasetExperiment‘.
`sample_type_head`	‘sample_meta‘ column classifying sample type.
`qc_label`	Value(s) in ‘sample_type_head‘ marking QC injections.
`sample_labels`	Value(s) marking biological samples.

Details

‘CV_QC‘ is technical variability: every QC injection is the same material, so its spread is the measurement noise of that compound in that run, and it is the usual quantification filter. ‘CV_sample‘ carries the biological spread *plus* that same noise. Their ratio says whether there is biology left to find: comfortably above 1 and the compound varies more between samples than between replicate injections of identical material; near 1 and it does not.

The column is named ‘CV_sample_vs_QC‘ rather than ‘CV_sample/CV_QC‘ because assigning into a ‘DatasetExperiment‘ passes the frame through [make.names()], which would rewrite the ‘/‘ to a ‘.‘ anyway.

Calling it twice is harmless - the columns are overwritten, not duplicated. [runMRManalyzeR()] applies it to every run, so a dataset read back from the output RDS already carries these columns.

Value

'de' with 'CV_QC', 'CV_sample' and 'CV_sample_vs_QC' added to 'variable_meta'. 'CV_QC' is 'NA' when the run holds fewer than two QC injections.

See Also

Other QC check: `mrmPlotPCA()`, `mrmRunPCA()`, `plotDrift()`, `plotLoadings()`, `plotQQ()`, `summariseFeatures()`

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
de <- addCVMetrics(de, qc_label = "QC")
head(as.data.frame(de$variable_meta)[, c("Compound", "CV_QC",
                                         "CV_sample", "CV_sample_vs_QC")])
```

`adjustConcentration` *Adjust a dataset to true sample concentrations*

Description

A calibration curve returns the concentration in the vial, and it does so based on the calibrant vial: it was built at the calibrant's internal-standard concentration and assumes the sample carries the same. This corrects that assumption, given the same volume of calibration and study sample being injected into the instrument:

Usage

```
adjustConcentration(
  de,
  sample_vol_col = "sample_volume_uL",
  cal_vol_col = "cal_vol_uL",
  sample_IS_col = "sample_IS_vol_uL",
  cal_IS_col = "cal_IS_vol_uL",
  starting_vol_col = FALSE
)
```

Arguments

de	A 'struct::DatasetExperiment'.
sample_vol_col, cal_vol_col, sample_IS_col, cal_IS_col	'sample_meta' columns holding, respectively, the sample's reconstitution volume, the calibration-standard volume, the IS volume added to the sample, and the IS volume added to the calibration standard.
starting_vol_col	'sample_meta' column holding the pre-dry-down starting volume, or 'FALSE' to skip the starting-volume correction.

Details

$$conc \times (cal_vol/sample_vol) \times (sample_IS/cal_IS)$$

Ultimately leading to

$$(sample_IS/sample_vol)/(cal_IS/cal_vol)$$

which is the ratio of internal-standard *concentrations*, sample vial over calibrant vial. More vial volume per unit of IS means a more dilute IS, a smaller IS peak, an inflated analyte/IS response ratio and therefore a curve that reads high - so the factor falls below 1 and scales it back down.

When ‘starting_vol_col‘ is supplied, a second step scales back to the concentration in the original, undried sample ($conc \times sample_vol/starting_vol$).

Value

‘de‘ with ‘data‘ scaled to concentration per sample.

See Also

Other peak-matrix processing: `correctBatch()`, `filterBlanks()`, `filterFeatures()`, `filterSamples()`, `imputeMissing()`, `normaliseMatrix()`, `processDataset()`, `subsetDataset()`

Examples

```
de <- struct::DatasetExperiment(
  data = data.frame(PGE2 = c(10, 20), PGD2 = c(30, 40),
                    row.names = c("S1", "S2")),
  sample_meta = data.frame(sample_volume_uL = c(100, 50),
                           cal_vol_uL       = c(100, 100),
                           sample_IS_vol_uL = c(10, 10),
                           cal_IS_vol_uL    = c(10, 10),
                           row.names = c("S1", "S2")),
  variable_meta = data.frame(Compound = c("PGE2", "PGD2"),
                             row.names = c("PGE2", "PGD2")))
adjustConcentration(de)$data
```

`assembleDataset` *Assemble a struct::DatasetExperiment from a matrix + metadata*

Description

Aligns ‘feature_meta‘ and ‘sample_meta‘ to the matrix (features to columns, samples to rows, both by name) and builds the ‘DatasetExperiment‘ so ‘variable_meta‘/‘sample_meta‘ rownames match the data assay exactly.

Usage

```
assembleDataset(x, feature_meta, sample_meta, name_col = "Name")
```

Arguments

x	Wide numeric matrix / data frame, rows = samples, columns = compounds (canonical ‘Compound‘ names).
feature_meta	Feature metadata; rows with ‘Report == "YES"‘ whose ‘Compound‘ is in ‘x‘ are kept, in matrix-column order.
sample_meta	Sample metadata; rows whose ‘name_col‘ value is a row of ‘x‘ are kept.
name_col	‘sample_meta‘ column matching the matrix row names.

Value

A ‘struct::DatasetExperiment‘.

See Also

Other data parse: `combineDatasets()`, `loadConfig()`, `loadDataset()`, `mrmValidateConfig()`, `mrmValidateDesign()`, `mrmValidateInput()`, `readPeakMatrix()`, `readSkyline()`, `readTargetLynx()`

Examples

```
m  <- data.frame(A = c(10, 20, 30), B = c(1, 2, 3),
                 row.names = c("S1", "S2", "S3"))
fm <- data.frame(Compound = c("A", "B"), Report = "YES",
                 row.names = c("A", "B"))
sm <- data.frame(Name = c("S1", "S2", "S3"), Group = c("x", "y", "x"),
                 row.names = c("S1", "S2", "S3"))
assembleDataset(m, fm, sm)
```

---

combineDatasets *Combine multiple processed datasets into one*

Description

Used to merge separately-acquired LC-MS panels (e.g. GOM, cysLT, SPM) that share a common set of biological samples. Inputs can be either ‘.RDS‘ files (containing a ‘struct::DatasetExperiment‘) or ‘.xlsx‘ workbooks produced by [‘runMRManalyzeR()‘] - see [‘loadDataset()‘].

Usage

```
combineDatasets(
  paths,
  feature_meta_cols = NULL,
  sample_meta_cols = NULL,
  feature_meta_rename = NULL,
  qc_remap = NULL,
  qc_regex = "^QC",
  sample_id_col = "Sample_ID",
  drop_samples = NULL,
```

```
  prefix_features = FALSE,
  combined_name = "combined",
  duplicate_samples = c("error", "first", "mean", "median")
)
```

Arguments

paths	Character vector of paths to '.RDS' or '.xlsx' files. May be named (names are used as dataset tags); otherwise the filename stem is used as the tag.
feature_meta_cols	Character vector - 'variable_meta' columns kept in the merged dataset. 'NULL' (default) = intersect across inputs. Missing columns in any single input are filled with 'NA'.
sample_meta_cols	Character vector - 'sample_meta' columns kept in the merged dataset. 'NULL' (default) = intersect across inputs.
feature_meta_rename	Optional named list. Names are dataset tags (or full paths); values are named character vectors of 'c(new_name = "old_name")' used to rename 'variable_meta' columns before alignment.
qc_remap	Optional named list. Names are dataset tags (or paths); values are named character vectors of 'c("OldQCID" = "NewQCID")' applied to 'Sample_ID'. QC samples whose original ID is NOT a key in the map are dropped.
qc_regex	Regex matching QC sample IDs (default '"^QC"').
sample_id_col	Column in 'sample_meta' used to match samples across datasets. Default '"Sample_ID"'.
drop_samples	Character vector of 'Sample_ID's to exclude from the merged result.
prefix_features	Controls feature-name prefixing. 'FALSE' (default): never prefix. 'TRUE': always prefix every feature with its dataset tag (e.g. 'GOM__PGE2'). '"auto"': prefix only features whose names collide across two or more inputs - note that the resulting names depend on which datasets are combined, so '"auto"' is not stable across different combine runs.
combined_name	Name attribute of the resulting 'DatasetExperiment'.
duplicate_samples	What to do when a dataset holds more than one row with the same 'sample_id_col'. '"error"' (default) stops and names them; '"first"' keeps the first occurrence; '"mean"' or '"median"' aggregate the measurements, taking metadata from the first row. A duplicate is either a deliberate re-injection or a data-entry error and those want opposite treatment, so the caller decides rather than the package.

Details

Defaults:

- Samples matched by 'Sample_ID' (intersect across inputs).

- ‘feature_meta_cols‘ / ‘sample_meta_cols‘ default to the intersection of column names across all inputs (so you only have to spell out an explicit list when you want to *narrow* the set).
- Data matrices are ‘cbind‘’d on the common samples.

Value

A single ‘struct::DatasetExperiment‘ containing the merged data.

See Also

Other data parse: `assembleDataset()`, `loadConfig()`, `loadDataset()`, `mrmValidateConfig()`, `mrmValidateDesign()`, `mrmValidateInput()`, `readPeakMatrix()`, `readSkyline()`, `readTargetLynx()`

Examples

```
mk <- function(feats) struct::DatasetExperiment(
  data = as.data.frame(matrix(1, 3, length(feats),
           dimnames = list(c("S1", "S2", "S3"), feats))),
  sample_meta = data.frame(Sample_ID = c("S1", "S2", "S3"),
           row.names = c("S1", "S2", "S3")),
  variable_meta = data.frame(Compound = feats, row.names = feats))
d <- tempfile("comb_"); dir.create(d)
p1 <- file.path(d, "a.RDS"); saveRDS(mk(c("A", "B")), p1)
p2 <- file.path(d, "c.RDS"); saveRDS(mk(c("C", "D")), p2)
combineDatasets(c(A = p1, B = p2))
```

---

correctBatch *Batch-correct a DatasetExperiment (median-ratio)*

Description

Scales each batch so its per-feature median matches the grand median across the reference (‘qc_label‘) samples, and returns the corrected ‘DatasetExperiment‘.

Usage

```
correctBatch(
  de,
  qc_label = "QC",
  factor_name = "Sample_type",
  batch_head = "Chrom_Batch",
  check_factor = NULL,
  min_ref = 2,
  feature_col = NULL
)
```

Arguments

de	A ‘struct::DatasetExperiment‘.
qc_label	Label identifying reference samples in ‘factor_name‘.
factor_name	‘sample_meta‘ column holding ‘qc_label‘.
batch_head	‘sample_meta‘ column identifying the batch. Every injection needs a value; an empty one stops with an error naming it.
check_factor	Optional ‘sample_meta‘ column holding the study factor. When given, the reference samples are cross-tabulated against it and a warning is issued if any batch carries only one level - the signature of a design where correcting on biological samples would remove real differences. ‘NULL‘ skips the check.
min_ref	Minimum reference samples required in each batch. Below this the batch median is not estimable and the correction would be noise.
feature_col	‘NULL‘ (default) to correct every compound, or the name of a ‘variable_meta‘ column of ‘YES‘ / ‘NO‘ selecting the compounds to correct.

Details

The reference has to be material whose true value does not differ between batches, because any difference that remains is treated as technical and removed. Pooled QC injections satisfy that by construction - the same extract, injected repeatedly - which is why ‘"QC"‘ is the default.

Using the biological samples as the reference is a legitimate alternative, and often a more stable one because there are more of them, but it rests on the study being balanced across batches: it assumes the biology is the same on average in each batch. Where that holds it is fine. Where batch is confounded with the study factor - all treated animals run on day 2, say - the between-batch difference is partly the effect you are looking for, and correcting removes it silently. Pass ‘check_factor‘ to have that assumption tested rather than assumed.

Medians count a missing value as zero, so the median of ‘c(NA, NA, 1, 2, 3)‘ is 1, not 2. A compound missing from most of a batch’s reference injections is at or below detection there, and a median of only the injections where it was found would describe that batch by its highest values. When more than half are missing the median is zero and the ratio is undefined. A compound whose median is zero or negative in any batch, or across all the reference injections, is corrected in no batch at all: scaling only some of its batches would leave them on different footings, which is the problem correction exists to remove. Its values are left as measured - never turned into ‘NA‘ - the failing batches are recorded in ‘bc_failed_batches‘, and each such compound is listed on screen. Negative medians only arise from signed data such as blank-subtracted values. Missing values themselves are not filled in: impute afterwards, if at all, with [imputeMissing()].

Every injection needs a ‘batch_head‘ value. One with an empty value - a blank run outside the sequence, say - cannot be scaled against anything, so the function stops and names it: give it a batch, or exclude it from processing.

‘feature_col‘ restricts the correction to part of the panel. It names a ‘variable_meta‘ column holding ‘YES‘ or ‘NO‘ per compound; case and surrounding spaces are ignored, and ‘TRUE‘ / ‘FALSE‘ work too. A ‘NO‘ compound is left as measured. A blank cell is treated as ‘NO‘, with a warning, because leaving a value alone is the choice that can be undone.

Value

The batch-corrected ‘struct::DatasetExperiment‘, with two columns added to ‘variable_meta‘: ‘batch_corrected‘, ‘TRUE‘ where the compound was corrected, and ‘bc_failed_batches‘, which for a selected compound that could not be corrected names the batches whose reference median was zero or negative - ‘"grand median"‘ when the median across all reference injections was - and is empty otherwise.

See Also

Other peak-matrix processing: `adjustConcentration()`, `filterBlanks()`, `filterFeatures()`, `filterSamples()`, `imputeMissing()`, `normaliseMatrix()`, `processDataset()`, `subsetDataset()`

Examples

```
m  <- data.frame(A = c(10, 12, 20, 24), B = c(5, 6, 10, 12),
                 row.names = c("S1", "S2", "S3", "S4"))
fm <- data.frame(Compound = c("A", "B"), Correct = c("YES", "NO"),
                 row.names = c("A", "B"))
sm <- data.frame(Sample_type = "Sample",
                 Chrom_Batch = c("b1", "b1", "b2", "b2"),
                 row.names = c("S1", "S2", "S3", "S4"))
de <- struct::DatasetExperiment(data = m, sample_meta = sm, variable_meta = fm)
correctBatch(de, qc_label = "Sample", factor_name = "Sample_type",
             batch_head = "Chrom_Batch")

# Only the compounds marked YES
out <- correctBatch(de, qc_label = "Sample", factor_name = "Sample_type",
                    batch_head = "Chrom_Batch", feature_col = "Correct")
out$data
```

`filterBlanks` *Blank-filter a dataset (mask values at or below the blank level)*

Description

For each feature, computes a summary of the blank injections times ‘blank_filter‘ and sets any value below that threshold to ‘NA‘. Blank injections are identified from ‘blank_head‘ in the dataset’s ‘sample_meta‘, so nothing has to be passed separately.

Usage

```
filterBlanks(
  de,
  blank_filter,
  blank_head = "Sample_type",
  blank_name = "Blank",
  summary = c("mean", "median"),
  min_blanks = 2,
  missing_blanks = c("warn", "error", "skip")
)
```

Arguments

de	A ‘struct::DatasetExperiment‘.
blank_filter	Numeric fold-change multiplier applied to the blank summary (e.g. ‘3‘ keeps only peaks at least three times the blank level).
blank_head	‘sample_meta‘ column identifying blank injections.
blank_name	Value in ‘blank_head‘ marking a blank injection.
summary	‘"mean"‘ (default) or ‘"median"‘ of the blank injections.
min_blanks	Minimum blank injections required. Below this the threshold rests on too little to be meaningful, and ‘missing_blanks‘ decides what happens.
missing_blanks	What to do when fewer than ‘min_blanks‘ are found: ‘"warn"‘ (default) filters anyway and says so, ‘"error"‘ stops, ‘"skip"‘ returns the dataset unfiltered.

Details

‘summary = "median"‘ is worth considering over the default mean: a single contaminated blank drags a mean threshold up and can mask most of a compound’s real measurements, and with the two or three blanks a typical run carries there is no way to see that from the result alone.

The per-feature threshold and the number of values it masked are recorded in ‘variable_meta‘ as ‘blank_threshold‘ and ‘n_masked_blank‘, so the filter’s effect is inspectable afterwards rather than only visible as missing data.

Value

‘de‘ with sub-threshold values in ‘data‘ set to ‘NA‘, and ‘blank_threshold‘ / ‘n_masked_blank‘ added to ‘variable_meta‘.

See Also

Other peak-matrix processing: `adjustConcentration()`, `correctBatch()`, `filterFeatures()`, `filterSamples()`, `imputeMissing()`, `normaliseMatrix()`, `processDataset()`, `subsetDataset()`

Examples

```
de <- struct::DatasetExperiment(
  data = data.frame(PGE2 = c(100, 90, 5, 4), PGD2 = c(50, 60, 1, 2),
                    row.names = c("S1", "S2", "B1", "B2")),
  sample_meta = data.frame(
    Sample_type = c("Sample", "Sample", "Blank", "Blank"),
    row.names = c("S1", "S2", "B1", "B2")),
  variable_meta = data.frame(Compound = c("PGE2", "PGD2"),
                             row.names = c("PGE2", "PGD2")))
out <- filterBlanks(de, blank_filter = 3)
out$data
as.data.frame(out$variable_meta)[, c("blank_threshold", "n_masked_blank")]
```

`filterFeatures` *Filter features by detection frequency*

Description

Drops features that are missing from too much of the data. A feature's detection fraction is the proportion of samples in which it has a non-‘NA‘ value; ‘method‘ decides which samples that proportion is computed over.

Usage

```
filterFeatures(
  de,
  min_frac = 0.5,
  method = c("within", "across", "QC"),
  group_col = NULL,
  sample_col = "Sample_type",
  sample_labels = "Sample",
  qc_label = "QC"
)
```

Arguments

`de`	A ‘struct::DatasetExperiment‘.
`min_frac`	Numeric in ‘[0, 1]‘. A feature is kept when its detection fraction is greater than or equal to this.
`method`	‘"within"‘ (default), ‘"across"‘ or ‘"QC"‘; see Details.
`group_col`	‘sample_meta‘ column whose levels are tested separately when ‘method = "within"‘. Required for that method.
`sample_col`	‘sample_meta‘ column identifying the sample type.
`sample_labels`	Values of ‘sample_col‘ marking study samples. Only these count towards ‘"within"‘ and ‘"across"‘.
`qc_label`	Value of ‘sample_col‘ marking QC injections, used by ‘method = "QC"‘.

Details

‘"across"‘ pools every study sample into one fraction. It is the strictest option and the easiest to misuse: a feature present throughout one treatment group and absent from the others scores roughly ‘1 / n_groups‘, so a threshold of 0.5 deletes it *because* it is treatment-specific. Use it only when the question is "is this measured everywhere".

‘"within"‘ (the default) computes the fraction separately within each level of ‘group_col‘ and keeps a feature that clears ‘min_frac‘ in **any one** level, so the treatment-specific feature above survives on its own group. Small groups quantise the threshold: with ‘n = 3‘ a fraction can only be 0, 1/3, 2/3 or 1, so every ‘min_frac‘ in (1/3, 2/3] behaves identically.

‘"QC"‘ computes the fraction over QC injections only, which answers "is this measured reproducibly" rather than "is this present in the biology". QCs are pooled, so a feature confined to one treatment is diluted in the pool and can fall below detection there - this mode also removes treatment-specific features. With no QCs present the function warns and falls back.

QC and blank injections never count towards ‘"within"‘ or ‘"across"‘. They remain in the returned dataset, since the blank filter and [correctBatch()] still need them, but they are excluded from the denominator so a compound seen only in the QCs cannot keep itself alive.

The fraction actually used is recorded per feature in ‘variable_meta‘ as ‘detected_frac‘, so the decision is inspectable rather than only visible as absent columns.

Value

‘de‘ with failing features removed from ‘data‘ and ‘variable_meta‘, and a ‘detected_frac‘ column added to ‘variable_meta‘.

See Also

[filterSamples()], which drops injections rather than features and is meant to run after this one.

Other peak-matrix processing: `adjustConcentration()`, `correctBatch()`, `filterBlanks()`, `filterSamples()`, `imputeMissing()`, `normaliseMatrix()`, `processDataset()`, `subsetDataset()`

Examples

```
de <- struct::DatasetExperiment(
  data = data.frame(
    good   = c(10, 12, 11, 9, 10, 11),
    npOnly = c(NA, NA, NA, 20, 22, 21),
    sparse = c(5, NA, NA, NA, NA, 6),
    row.names = paste0("S", 1:6)),
  sample_meta = data.frame(
    Sample_type = rep("Sample", 6),
    Treatment   = rep(c("Control", "NP"), each = 3),
    row.names   = paste0("S", 1:6)),
  variable_meta = data.frame(
    Compound  = c("good", "npOnly", "sparse"),
    row.names = c("good", "npOnly", "sparse")))

# 'within' keeps npOnly on the strength of its own group; 'across' would
# delete it for being absent from two thirds of the samples.
filterFeatures(de, min_frac = 0.5, method = "within", group_col = "Treatment")
```

`filterSamples` *Filter injections by missing-value fraction*

Description

Drops study samples in which too large a proportion of features is ‘NA‘. Where [filterFeatures()] asks a question about the biology, this asks one about the run: an injection that missed most of the panel is usually a failure rather than a finding.

Usage

```
filterSamples(
  de,
  max_na = 0.8,
  sample_col = "Sample_type",
  sample_labels = "Sample"
)
```

Arguments

de	A ‘struct::DatasetExperiment‘.
max_na	Numeric in ‘[0, 1]‘. A study sample is dropped when the proportion of ‘NA‘ features exceeds this.
sample_col	‘sample_meta‘ column identifying the sample type.
sample_labels	Values of ‘sample_col‘ marking study samples - the only rows this function will remove.

Details

Run this after [filterFeatures()], not before. Judging an injection against a feature list that still contains everything the instrument barely saw gives a meaningless denominator - every sample looks poor because the feature set is padded with noise. Once the undetected features are gone, the fraction answers the question worth asking: of the features that are real, how many did this injection get?

Only study samples are eligible for removal. A QC or blank that fails the threshold is reported but kept, because dropping a QC can leave a batch without one and [correctBatch()] would then have nothing to anchor to. If a QC really did fail, exclude it deliberately in the sample metadata rather than letting a threshold do it silently.

The fraction computed for every row is recorded in ‘sample_meta‘ as ‘na_frac‘, including for the QCs and blanks that were exempt, so a failing QC is visible afterwards.

Value

‘de‘ with failing study samples removed from ‘data‘ and ‘sample_meta‘, and an ‘na_frac‘ column added to ‘sample_meta‘.

See Also

[filterFeatures()], which should run first.

Other peak-matrix processing: `adjustConcentration()`, `correctBatch()`, `filterBlanks()`, `filterFeatures()`, `imputeMissing()`, `normaliseMatrix()`, `processDataset()`, `subsetDataset()`

Examples

```
de <- struct::DatasetExperiment(
  data = data.frame(
    f1 = c(10, 11, NA, 12),
```

```
    f2 = c(20, 21, NA, 22),
    f3 = c(30, 31, NA, 33),
    row.names = c("S1", "S2", "S3_failed", "QC1")),
  sample_meta = data.frame(
    Sample_type = c("Sample", "Sample", "Sample", "QC"),
    row.names   = c("S1", "S2", "S3_failed", "QC1")),
  variable_meta = data.frame(
    Compound  = c("f1", "f2", "f3"),
    row.names = c("f1", "f2", "f3")))

# S3_failed is dropped; a QC at the same missingness would only be reported
filterSamples(de, max_na = 0.5)
```

`imputeMissing` *Impute missing values below each compound's lowest measurement*

Description

Zeros are treated as missing. For each feature, the 'NA's among the non-blank injections are filled with values below the lowest one actually measured, because a non-detect sits under the detection floor rather than at zero. The ceiling is 'scalar' times that minimum.

Usage

```
imputeMissing(
  de,
  scalar,
  blank_head = "Sample_type",
  blank_name = "Blank",
  warn_frac = 0.5,
  method = c("gaussian", "constant"),
  seed = 42,
  by = NULL
)
```

Arguments

`de`	A 'struct::DatasetExperiment'.
`scalar`	Fraction of the per-feature minimum used as the ceiling, e.g. '0.2'.
`blank_head`	'sample_meta' column identifying blank injections.
`blank_name`	Value in 'blank_head' marking a blank injection.
`warn_frac`	Warn about features imputed in more than this fraction of the non-blank injections. 'NULL' disables the warning.
`method`	'"gaussian"' (default) for random draws below the ceiling, or '"constant"' to fill every cell with the ceiling.
`seed`	Random seed for 'method = "gaussian"'; 'NULL' uses the current random stream.
`by`	Optional 'sample_meta' column of processing batches, each given its own ceiling. 'NULL' (default) treats all injections together.

Details

With ‘method = "gaussian"‘ (the default) every missing cell gets its own random draw from a normal distribution centred at half the ceiling, with a standard deviation of a sixth of it, truncated to lie between 0 and the ceiling. A single constant would give every non-detect the same value, which shrinks the variance of a compound and ties its ranks - both of which flatter a statistical test. ‘method = "constant"‘ fills every cell with the ceiling itself, as earlier versions did. The draws use ‘seed‘, so a rerun gives the same values; ‘seed = NULL‘ uses the session’s random stream.

‘by‘ names a ‘sample_meta‘ column of processing batches. Each batch then gets its own ceiling from its own minimum, so a batch measured at lower response is not filled with another batch’s values. Injections with an empty ‘by‘ value are treated as one batch of their own.

Imputation is the one step that invents numbers, so it records what it did in ‘variable_meta‘: ‘impute_max‘, the ceiling (one column per batch, ‘impute_max_<batch>‘, when ‘by‘ gives more than one); ‘n_imputed‘, how many cells were filled; and ‘frac_imputed‘, the share of the compound’s non-blank injections that were filled. A compound whose ‘frac_imputed‘ is high is mostly invented, and a difference found in it is an artefact of the fill. Check that column before interpreting anything, and consider excluding compounds above ‘warn_frac‘.

Blank injections (identified from ‘blank_head‘ in ‘sample_meta‘) are left untouched and do not count towards the minimum.

Value

‘de‘ with ‘data‘ imputed in the original row order, and ‘impute_max‘ (or ‘impute_max_<batch>‘), ‘n_imputed‘ and ‘frac_imputed‘ added to ‘variable_meta‘.

See Also

Other peak-matrix processing: `adjustConcentration()`, `correctBatch()`, `filterBlanks()`, `filterFeatures()`, `filterSamples()`, `normaliseMatrix()`, `processDataset()`, `subsetDataset()`

Examples

```
de <- struct::DatasetExperiment(
  data = data.frame(PGE2 = c(10, 5, NA, 8), PGD2 = c(2, NA, 4, NA),
                    row.names = c("S1", "S2", "S3", "S4")),
  sample_meta = data.frame(Sample_type = rep("Sample", 4),
                           row.names = c("S1", "S2", "S3", "S4")),
  variable_meta = data.frame(Compound = c("PGE2", "PGD2"),
                             row.names = c("PGE2", "PGD2")))
imputeMissing(de, scalar = 0.5)$data
imputeMissing(de, scalar = 0.5, method = "constant")$data
```

`loadConfig` *Load a project YAML, resolving !concat_path and !concat_str anchors*

Description

Load a project YAML, resolving !concat_path and !concat_str anchors

Usage

```
loadConfig(path_yaml)
```

Arguments

path_yaml	Full path to YAML file

Value

Named list of parameters parsed from the YAML

See Also

Other data parse: assembleDataset(), combineDatasets(), loadDataset(), mrmValidateConfig(), mrmValidateDesign(), mrmValidateInput(), readPeakMatrix(), readSkyline(), readTargetLynx()

Examples

```
cfg <- loadConfig(system.file("extdata", "example_config.yml",
                              package = "MRManalyzeR"))
names(cfg$project)
```

---

loadDataset *Load a 'DatasetExperiment' from '.RDS' or '.xlsx'*

Description

Used by ['combineDatasets()'] so the combine workflow can mix RDS and xlsx inputs (xlsx loading reconstructs a 'DatasetExperiment' from the standard sheets 'feature_metadata', 'sample_metadata', 'matrix' written by ['runMRManalyzeR()']).

Usage

```
loadDataset(path, sample_id_col = "Sample_ID")
```

Arguments

path	Path to either an '.RDS' containing a 'struct::DatasetExperiment', or an '.xlsx' produced by ['runMRManalyzeR()'] (with 'feature_metadata', 'sample_metadata', 'matrix' tabs).
sample_id_col	Column in 'sample_metadata' used as the row-name key when loading from xlsx. Default '"Sample_ID"'.

Value

A 'struct::DatasetExperiment'.

See Also

Other data parse: assembleDataset(), combineDatasets(), loadConfig(), mrmValidateConfig(), mrmValidateDesign(), mrmValidateInput(), readPeakMatrix(), readSkyline(), readTargetLynx()

Examples

```
de <- struct::DatasetExperiment(
  data          = data.frame(A = c(1, 2), B = c(3, 4),
                             row.names = c("S1", "S2")),
  sample_meta   = data.frame(Sample_ID = c("S1", "S2"),
                             row.names = c("S1", "S2")),
  variable_meta = data.frame(Compound = c("A", "B"),
                             row.names = c("A", "B")))
f <- tempfile(fileext = ".RDS"); saveRDS(de, f)
loadDataset(f)
```

---

mrm_validation *The result of a validation check*

Description

What [mrmValidateInput()], [mrmValidateConfig()] and [mrmValidateDesign()] return. A plain list of three elements, given the class ‘mrm_validation‘ so that printing it reports the findings rather than dumping the list. Use it as a list: ‘res$errors‘, ‘res$summary‘, and so on.

Value

Nothing: this page documents the object the validation functions return, which is the list described above with class ‘mrm_validation‘. It is not built directly.

Structure

‘errors‘ Character vector. Problems that stop a run: a missing sheet, a column the configuration names that the workbook does not have, a comparison level that no sample carries. Empty when there are none.

‘warnings‘ Character vector. Things worth knowing that do not stop a run: a group below the minimum size, a batch without enough QC injections, a factor confounded with the batch variable.

‘summary‘ Data frame, one row per check, with columns ‘check‘ (its short name), ‘status‘ (‘"ok"‘, ‘"warning"‘ or ‘"error"‘) and ‘message‘ (‘NA‘ where the check passed). This is every check that ran, passing ones included, so it also says what was looked at.

Methods

‘print()‘ reports the counts, then the errors and warnings in full, or ‘All checks passed.‘ when both are empty. It returns the object invisibly, so a check can be printed and kept in one line.

See Also

[mrmValidateInput()], [mrmValidateConfig()], [mrmValidateDesign()]

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
res <- mrmValidateDesign(de, list(project = list(stats_report = list())))
res$summary
length(res$errors)
```

`mrmPlotHeatmap` *Sample-by-feature heatmap with class annotations*

Description

A z-scored heatmap of samples against compounds, with samples blocked by a study factor and compounds blocked by their class. Where the comparisons test one compound at a time, this shows the whole panel at once - which is how you see that a treatment moved an entire pathway rather than a handful of compounds that happened to clear a threshold.

Usage

```
mrmPlotHeatmap(
  de,
  features = NULL,
  title = NULL,
  color_samples_by,
  group_features_by = NULL,
  transform = c("none", "log2", "sqrt"),
  scale = c("z", "none"),
  cluster_rows = FALSE,
  cluster_within_groups = TRUE,
  orientation = c("samples_y", "samples_x", "auto"),
  feature_labels = c("show", "small", "hide"),
  sample_id_head = NULL,
  cap = 2.5,
  cell_size = 10
)
```

Arguments

`de`	A ‘struct::DatasetExperiment‘.
`features`	Compounds to include; ‘NULL‘ uses all of them.
`title`	Plot title.
`color_samples_by`	‘sample_meta‘ column used to block and annotate samples.

`group_features_by`
: ‘variable_meta‘ column used to block and annotate compounds; ‘NULL‘ leaves them unblocked.

`transform`
: ‘"none"‘, ‘"log2"‘ or ‘"sqrt"‘, applied before scaling. Negative values are floored at 0 first.

`scale`
: ‘"z"‘ to scale each compound, or ‘"none"‘.

`cluster_rows`
: Cluster the sample axis.

`cluster_within_groups`
: Cluster compounds inside each class block.

`orientation`
: ‘"samples_y"‘, ‘"samples_x"‘ or ‘"auto"‘.

`feature_labels`
: ‘"show"‘, ‘"small"‘ or ‘"hide"‘ - the compound axis only; sample labels always stay readable.

`sample_id_head`
: ‘sample_meta‘ column supplying sample labels; ‘NULL‘ uses row names.

`cap`
: Colour-scale limit in standard deviations.

`cell_size`
: Cell size in points: one value for square cells, or ‘c(width, height)‘. Because these are absolute units they survive being stretched to fill a canvas, which a proportional cell does not - so leave this set unless you want cells to take whatever aspect the figure dimensions imply. ‘NULL‘ restores that fill-the-canvas behaviour.

Details

Values are transformed, then scaled per compound (‘scale = "z"‘) so that compounds spanning three orders of magnitude are comparable in one picture - without it the abundant compounds dominate and everything else is a flat wash. Scores are capped at ‘cap‘ standard deviations so one outlier cannot consume the whole colour range. Zero-variance compounds are dropped. Missing values are drawn grey; only for ordering and clustering do they count as the compound’s mean (0 after scaling).

Compounds are ordered into contiguous blocks by ‘group_features_by‘, with gaps drawn between blocks, and clustered *within* each block when ‘cluster_within_groups‘ is set - so a class stays together and the clustering tells you about structure inside it, not about which classes happen to resemble each other. The class palette matches [plotLoadings()] and [plotCorrelation()].

‘orientation‘ decides which axis carries samples: ‘"samples_y"‘ puts them on rows, ‘"samples_x"‘ transposes, and ‘"auto"‘ puts whichever dimension is larger on the y-axis so a wide panel does not become tall and unreadable.

Value

A ‘ggplot‘ wrapping the heatmap grob, or ‘NULL‘ when there is too little data to draw (fewer than 3 samples, 2 features, or no variable compound).

See Also

Other stats: `plotBoxplot()`, `plotCorrelation()`, `plotGroupHeatmap()`, `plotVolcano()`, `runStats()`, `summariseGroups()`

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
de <- subsetDataset(de, conditions = list(Sample_type = "Sample"))
mrmPlotHeatmap(de, color_samples_by = "Treatment",
             group_features_by = "Enzymatic_pathway",
             title = "All features")
```

`mrmPlotPCA` *Plot PCA scores, coloured by a sample-metadata column*

Description

Draws the scores from [mrmRunPCA()] with the percentage of variance explained on each axis. The colouring is the whole point: the same scores answer a different question depending on what you colour them by. Colour by sample type and the question is whether the QC injections cluster tightly in the middle of the samples. Colour by chromatographic batch and it is whether the variance driving component 1 is analytical rather than biological - if the batches separate, correct before interpreting anything. Colour by the study factor and it is whether there is any separation to find at all.

Usage

```
mrmPlotPCA(
  pca,
  sample_meta,
  colour_by,
  components = c(1, 2),
  label = c("none", "all", "outliers"),
  label_factor = NULL,
  ellipse = c("none", "group"),
  ellipse_level = 0.95,
  numeric = NA
)
```

Arguments

`pca`	The list returned by [mrmRunPCA()].
`sample_meta`	Sample metadata, or a 'DatasetExperiment' to take it from. Rows are matched to the PCA scores by name, so samples dropped by 'sample_na_max' do not have to be removed by the caller.
`colour_by`	'sample_meta' column to colour points by.
`components`	Length-2 integer vector: which principal components to plot.
`label`	'"none"', '"all"', or '"outliers"' (beyond 2.5 SD on either component).
`label_factor`	'sample_meta' column supplying point labels and the tooltip id; 'NULL' uses row names.

`ellipse`	'"none"' or '"group"' - a normal-probability ellipse per level of 'colour_by', drawn only for categorical colourings with more than one level.
`ellipse_level`	Confidence level for the ellipse.
`numeric`	'NA' (default) to decide from the column's type, or 'TRUE'/'FALSE' to force a continuous or discrete colour scale.

Details

Which is why the reports draw one plot per column listed under 'qc_pca.color_by' rather than choosing one.

Numeric colourings (injection order, a numeric batch id) get a continuous viridis scale; categorical ones get a discrete palette. Pass 'numeric' explicitly to override the automatic choice - useful when a column is stored as character but is really a sequence.

Value

A 'ggplot', or 'NULL' if the PCA could not be fit ('pca$pr' is 'NULL'), which lets a report skip the section without erroring.

See Also

Other QC check: `addCVMetrics()`, `mrmRunPCA()`, `plotDrift()`, `plotLoadings()`, `plotQQ()`, `summariseFeatures()`

Examples

```
de  <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                               package = "MRManalyzeR"))
pca <- mrmRunPCA(de, transform = "log2")
mrmPlotPCA(pca, de, colour_by = "Chrom_Batch")
```

---

mrmRunExample *Render the bundled example MRManalyzeR reports*

Description

Convenience wrapper that drives the full workflow against the bundled oxylipin example dataset (a subset of Kolmert et al. 2018, doi:10.1016/j.prostaglandins.2018.05.005). It loads 'inst/extdata/example_data.xlsx' and 'inst/extdata/example_config.yml', rewrites the YAML's 'paths:' block to point at a writable results directory + the bundled xlsx, and calls ['runMRManalyzeR()']. After the run the generated HTML reports are opened in interactive sessions.

Usage

```
mrmRunExample(
  results_dir = tempfile("MRManalyzeR_example_"),
  open = interactive(),
  render = TRUE
)
```

Arguments

results_dir	Directory to write outputs (xlsx, RDS, HTML reports) to. Defaults to a fresh 'tempdir()' so repeated calls do not collide.
open	Logical. Open the HTML reports in the browser / RStudio viewer after rendering? Default 'TRUE' in interactive sessions.
render	Logical. Render the HTML reports? Default 'TRUE'. Set 'FALSE' to build the peak matrix + xlsx/RDS outputs only (no pandoc) - a quick smoke test of the processing workflow.

Value

Invisibly, the list returned by ['runMRManalyzeR()'], whose 'output_directory' element is the resolved results directory.

See Also

Other entry points: newConfig(), runMRManalyzeR(), runMRManalyzeRCombine()

Examples

```
# Light run: build the peak matrix + outputs, skip HTML report rendering.
res <- mrmRunExample(render = FALSE, open = FALSE)
res$output_directory       # where the xlsx + rds landed
```

---

mrmRunPCA *Pre-process a sample x feature matrix and run PCA*

Description

Standard metabolomics-style preprocessing pipeline:

1. Drop all-NA features (always). Optionally apply a stricter feature missing-value filter ('feat_na_max').
2. Optionally drop samples with too many missing values ('sample_na_max').
3. Impute remaining NAs below each feature's lowest measured value (random draws by default - see 'impute').
4. Optional transform (log2(x+1) or sqrt(x)).
5. Drop zero-variance features.
6. 'prcomp(center, scale.)' - mean-centre and (optionally) autoscale.

Usage

```
mrmRunPCA(
  X,
  impute = c("gaussian", "frac_min", "min", "half_min", "none"),
  impute_frac = 0.5,
  transform = c("log2", "sqrt", "none"),
  center = TRUE,
  scale = TRUE,
  feat_na_max = 1,
  sample_na_max = 1,
  seed = 42
)
```

Arguments

X	A 'struct::DatasetExperiment' (its 'data' is used), or a numeric matrix / data frame of samples x features. May contain NAs.
impute	'"gaussian"' (default), '"frac_min"', '"min"', '"half_min"' or '"none"'. '"gaussian"' fills each NA with its own random value between 0 and 'impute_frac' x the feature's minimum (a normal centred at half that ceiling, SD a sixth of it - see [imputeMissing()]); '"frac_min"', '"min"' and '"half_min"' fill every NA with 'impute_frac', 1 or 0.5 x the minimum. Zeros count as missing for every option except '"none"'.
impute_frac	Fraction of the per-feature minimum used by '"gaussian"' and '"frac_min"' (default 0.5).
transform	one of '"none"', '"log2"', '"sqrt"' (default '"log2"').
center	logical, passed to [stats::prcomp()].
scale	logical, passed to [stats::prcomp()] as 'scale.'.
feat_na_max	Maximum allowed fraction of NA values per feature (0-1). Features exceeding this threshold are dropped before imputation. Default '1' retains the original behaviour (only all-NA features are dropped). Set e.g. '0.5' to also drop features missing in more than 50 percent of samples.
sample_na_max	Maximum allowed fraction of NA values per sample (0-1). Samples exceeding this threshold are dropped before imputation. Default '1' applies no sample filtering (original behaviour). Set e.g. '0.8' to drop samples missing more than 80 percent of features.
seed	Random seed for 'impute = "gaussian"', so a rerun gives the same PCA; 'NULL' uses the current random stream.

Details

Returns the prcomp object plus a human-readable trace of the steps that were actually applied - reports embed this so the user can see at a glance what transformation went into the PCA.

Value

A list with elements

'pr' the 'prcomp' object, or 'NULL' if PCA could not be fit.

'X_clean' the matrix actually fed to 'prcomp'.

'n_samples', 'n_features' dimensions after cleaning.

'dropped_all_na', 'dropped_feat_filter', 'dropped_sample_filter', 'dropped_zero_var' counts of features/samples dropped at each step.

'steps' character vector of human-readable steps applied.

See Also

Other QC check: `addCVMetrics()`, `mrmPlotPCA()`, `plotDrift()`, `plotLoadings()`, `plotQQ()`, `summariseFeatures()`

Examples

```
X <- matrix(abs(rnorm(60, 100, 20)), nrow = 12,
            dimnames = list(paste0("S", 1:12), paste0("F", 1:5)))
res <- mrmRunPCA(X, transform = "log2")
res$steps
```

`mrmValidateConfig` *Validate a YAML configuration*

Description

Structural checks on the config alone - blocks present, exactly one data source enabled, enumerated values spelled correctly. This catches the mistakes that would otherwise surface as an obscure error deep in report rendering.

Usage

```
mrmValidateConfig(config)
```

Arguments

`config`	A config list, or a path to a YAML file.

Value

An 'mrm_validation' object.

See Also

Other data parse: `assembleDataset()`, `combineDatasets()`, `loadConfig()`, `loadDataset()`, `mrmValidateDesign()`, `mrmValidateInput()`, `readPeakMatrix()`, `readSkyline()`, `readTargetLynx()`

Examples

```
mrmValidateConfig(system.file("extdata", "example_config.yml",
                              package = "MRManalyzeR"))
```

`mrmValidateDesign` *Validate a study design against the data*

Description

The checks that need the data and the configuration together: are the reference samples actually present, does every batch have them, is the design balanced enough for the correction and the tests requested. These are the failures that produce a plausible-looking result rather than an error, which is what makes them worth checking up front.

Usage

```
mrmValidateDesign(
  de,
  config = NULL,
  sample_type_head = "Sample_type",
  qc_label = "QC",
  blank_name = "Blank",
  batch_head = "Chrom_Batch",
  min_group = 3
)
```

Arguments

`de`	A 'struct::DatasetExperiment'.
`config`	Optional config list or path; when supplied, the requested comparisons and normalisation are checked against the data.
`sample_type_head, qc_label, blank_name`	Sample-type column and the values marking QC and blank injections.
`batch_head`	Batch column.
`min_group`	Minimum observations per group before a comparison is worth running.

Value

An 'mrm_validation' object.

See Also

Other data parse: `assembleDataset()`, `combineDatasets()`, `loadConfig()`, `loadDataset()`, `mrmValidateConfig()`, `mrmValidateInput()`, `readPeakMatrix()`, `readSkyline()`, `readTargetLynx()`

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
mrmValidateDesign(de)
```

`mrmValidateInput` *Validate a workbook before processing it*

Description

Checks the shape of the inputs - the things that make a run fail halfway through, or worse, succeed on the wrong data. Every check runs, so one call reports every problem rather than stopping at the first.

Usage

```
mrmValidateInput(
  feature_meta,
  sample_meta,
  data = NULL,
  name_col = "Name",
  compound_col = "Compound",
  processing_name_col = "Processing_name",
  report_col = "Report",
  report_value = "YES",
  include_col = "Include",
  include_value = "YES",
  injection_order_head = NULL
)
```

Arguments

`feature_meta`	Feature metadata sheet.
`sample_meta`	Sample metadata sheet.
`data`	Optional wide matrix (samples x compounds), e.g. from [readTargetLynx()]. Cross-checks between data and metadata are skipped when absent.
`name_col`, `compound_col`, `processing_name_col`, `report_col`, `include_col`	Column names, matching the arguments of [processDataset()].
`report_value`	Value of 'report_col' marking features to report. Any other value, blank cells included, counts as not reported.
`include_value`	Value of 'include_col' marking injections to process.
`injection_order_head`	'sample_meta' column holding acquisition order, or 'NULL' (default) to skip the check. When given, every included injection must have a numeric value there: drift plots and the QC-PCA colour scale read it as a number, and a

text value (a formula returning ‘"_48"‘, say) would otherwise leave them empty without an error. An absent column is only a warning, since the plots then fall back to row order.

Value

An ‘mrm_validation‘ object with ‘errors‘, ‘warnings‘ and a ‘summary‘ data frame of every check.

See Also

Other data parse: `assembleDataset()`, `combineDatasets()`, `loadConfig()`, `loadDataset()`, `mrmValidateConfig()`, `mrmValidateDesign()`, `readPeakMatrix()`, `readSkyline()`, `readTargetLynx()`

Examples

```
xlsx <- system.file("extdata", "example_data.xlsx",
                    package = "MRManalyzeR")
fdata <- openxlsx::read.xlsx(xlsx, sheet = "feature_metadata")
meta  <- openxlsx::read.xlsx(xlsx, sheet = "sample_metadata")
mrmValidateInput(fdata, meta)
```

---

newConfig *Write a configuration file to start from*

Description

A run is driven by one YAML file, so the first thing anyone needs is a file to edit. This writes one: ‘minimal = TRUE‘ gives the keys a run cannot do without, each on one line with its options beside it; ‘minimal = FALSE‘ copies the fully commented reference configuration shipped with the package, which documents every key the package reads.

Usage

```
newConfig(
  path,
  template = c("study", "combine"),
  source = c("targetlynx", "skyline", "matrix"),
  minimal = TRUE,
  overwrite = FALSE
)
```

Arguments

`path`	File to write. A directory is also accepted, and the file is then called ‘project_config.yml‘ inside it.
`template`	‘"study"‘ for a single acquisition, the usual case, or ‘"combine"‘ for merging several processed panels by sample.

source	Data source for a ‘"study"‘ template: ‘"targetlynx"‘, ‘"skyline"‘ or ‘"matrix"‘ (any sample x analyte table). Ignored for ‘"combine"‘.
minimal	‘TRUE‘ (default) writes the short form. ‘FALSE‘ copies the bundled reference configuration, comments and all.
overwrite	Overwrite ‘path‘ if it exists. ‘FALSE‘ (default) stops rather than replace a configuration you have already edited.

Details

Nothing is guessed from your data. The paths, the sheet names and the metadata columns in the file are placeholders, and the run stops with a message naming whichever one does not match your workbook - so editing is a matter of answering those messages, in order.

Value

The path written, invisibly.

See Also

[runMRManalyzeR()], which runs a ‘"study"‘ file, and [runMRManalyzeRCombine()], which runs a ‘"combine"‘ one.

Other entry points: `mrmRunExample()`, `runMRManalyzeR()`, `runMRManalyzeRCombine()`

Examples

```
cfg <- newConfig(tempfile(fileext = ".yml"))
cat(readLines(cfg)[1:12], sep = "\n")
```

---

normaliseMatrix *Normalise a dataset by a per-sample metadata column*

Description

Divides each sample (row) of the dataset by that sample's value in ‘sample_meta[[column]]‘ – for example protein content, tissue weight, or volume of biofluid – putting all samples on a per-unit basis.

Usage

```
normaliseMatrix(de, column)
```

Arguments

de	A ‘struct::DatasetExperiment‘.
column	‘sample_meta‘ column to divide by.

Value

'de' with each row of 'data' divided by its per-sample divisor.

See Also

Other peak-matrix processing: `adjustConcentration()`, `correctBatch()`, `filterBlanks()`, `filterFeatures()`, `filterSamples()`, `imputeMissing()`, `processDataset()`, `subsetDataset()`

Examples

```
de <- struct::DatasetExperiment(
  data = data.frame(PGE2 = c(10, 20), PGD2 = c(30, 40),
                    row.names = c("S1", "S2")),
  sample_meta = data.frame(protein_ug = c(2, 4), row.names = c("S1", "S2")),
  variable_meta = data.frame(Compound = c("PGE2", "PGD2"),
                             row.names = c("PGE2", "PGD2")))
normaliseMatrix(de, column = "protein_ug")$data
```

---

plotBoxplot *Per-compound box or bar plot by group*

Description

One compound's values split by a grouping factor - the plot that actually gets looked at when a statistical test comes back significant, because it shows whether the difference rests on the whole group or on one animal.

Usage

```
plotBoxplot(
  de,
  compound,
  group_by,
  facet_by = NULL,
  type = c("box", "bar"),
  error_bar = c("SD", "SE", "none"),
  value_label = "value",
  sample_id_head = NULL,
  tooltip_factor = NULL,
  yaxis_quant = 0.95,
  yaxis_scalar = 1.3,
  fill_brewer = TRUE
)
```

Arguments

de	A ‘struct::DatasetExperiment‘.
compound	Feature (column) name to plot.
group_by	‘sample_meta‘ column defining the x-axis groups.
facet_by	Optional ‘sample_meta‘ column to facet across, e.g. one panel per comparison.
type	‘"box"‘ or ‘"bar"‘.
error_bar	‘"SD"‘, ‘"SE"‘ or ‘"none"‘ - bars only.
value_label	Y-axis label, e.g. the datatype units.
sample_id_head	‘sample_meta‘ column identifying points in the tooltip; ‘NULL‘ uses row names.
tooltip_factor	Additional ‘sample_meta‘ column shown in the tooltip, typically the primary study factor.
yaxis_quant, yaxis_scalar	Quantile used to cap the y-axis, and the headroom multiplier above it.
fill_brewer	Use the package’s qualitative palette for group fills; ‘FALSE‘ keeps ggplot2’s default hue scale.

Details

‘type = "box"‘ draws a box with the individual injections jittered over it, which is the honest view for the small groups a targeted study usually has: the reader can count the points. ‘type = "bar"‘ draws group means with SD or SE error bars, matching [summariseGroups()] exactly. Reports commonly show both, the box for distribution and the bar for the summary the statistics were computed on.

The y-axis is capped at ‘quantile(value, yaxis_quant) * yaxis_scalar‘ so one extreme injection cannot flatten the groups.

Value

A ‘ggplot‘.

See Also

[summariseGroups()] for the numbers the bars are drawn from.

Other stats: `mrmPlotHeatmap()`, `plotCorrelation()`, `plotGroupHeatmap()`, `plotVolcano()`, `runStats()`, `summariseGroups()`

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
de <- subsetDataset(de, conditions = list(Sample_type = "Sample"))
plotBoxplot(de, "Analyte_02", "Treatment", value_label = "ng/mL")
plotBoxplot(de, "Analyte_02", "Treatment", type = "bar",
            error_bar = "SE")
```

`plotCorrelation` *Feature-by-feature correlation heatmap*

Description

Draws one triangle of the correlation matrix for a set of compounds. Where the comparisons ask whether a compound differs between groups, this asks which compounds move *together* - which in a targeted panel usually means they share a branch of the pathway, so blocks of correlation along the diagonal are the pathway structure showing itself.

Usage

```
plotCorrelation(
  correlations,
  variable_meta = NULL,
  name = NULL,
  subset = NULL,
  method = NULL,
  group_by = NULL,
  cluster = TRUE,
  show_values = FALSE
)
```

Arguments

`correlations`	The ‘correlations‘ data frame from [runStats()], or the whole list. Long format: ‘feature_a‘, ‘feature_b‘, ‘estimate‘, plus ‘correlation‘ / ‘subset‘ / ‘method‘ identifying each run.
`variable_meta`	Feature metadata, or a ‘DatasetExperiment‘ to take it from. Only needed for ‘group_by‘.
`name`, `subset`, `method`	Select one run when the table holds several. ‘NULL‘ requires that only one is present.
`group_by`	‘variable_meta‘ column used to block and colour features.
`cluster`	Cluster features within each block (globally when there is no ‘group_by‘), using average linkage on ‘1 - |r|‘.
`show_values`	Print the coefficient in each cell. Ignored above 30 features, where the text is unreadable anyway.

Details

Only one triangle is drawn, because a correlation matrix is symmetric and showing both halves doubles the ink for no information. The diagonal is left grey unless self-correlations are present in the input.

Features are ordered by their ‘group_by‘ class into contiguous blocks and clustered within each block, so a class stays together and reads as a unit; with no ‘group_by‘, clustering is global. A

coloured strip along each axis carries the class, using the same palette as [plotLoadings()] and the feature annotation in [mrmPlotHeatmap()] - so a class is one colour everywhere in the report.

Value

A ‘ggplot‘, or ‘NULL‘ if the selection is empty.

See Also

Other stats: `mrmPlotHeatmap()`, `plotBoxplot()`, `plotGroupHeatmap()`, `plotVolcano()`, `runStats()`, `summariseGroups()`

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
de <- subsetDataset(de, conditions = list(Sample_type = "Sample"))
params <- list(correlations = list(enabled = TRUE, entries = list(
  list(name = "pairs", methods = "spearman", subsets = list(list())))))
st <- runStats(de, params)
plotCorrelation(st, de, group_by = "Enzymatic_pathway")
```

---

plotDrift *Plot one compound’s intensity against injection order*

Description

The core analytical-drift check: a compound’s response plotted against the order its injections were acquired in, coloured by sample type so the pooled QCs can be tracked through the sequence. QCs are identical material, so a trend, a step or a widening spread in them is instrumental - loss of sensitivity, a column change, a batch boundary - and not biology.

Usage

```
plotDrift(
  de,
  compound,
  sample_type_head = "Sample_type",
  levels = NULL,
  injection_order_head = "Injection_order",
  sample_id_head = NULL,
  value_label = "value",
  yaxis_quant = 0.95,
  yaxis_scalar = 1.3
)
```

Arguments

de	A ‘struct::DatasetExperiment‘.
compound	Feature (column) name to plot.
sample_type_head	‘sample_meta‘ column classifying sample type.
levels	Value(s) of ‘sample_type_head‘ to include; ‘NULL‘ keeps all.
injection_order_head	‘sample_meta‘ column holding acquisition order. Must be numeric; falls back to row position when the column is absent.
sample_id_head	‘sample_meta‘ column used to label points in the tooltip; ‘NULL‘ uses the row names.
value_label	Axis / tooltip label for the measured value, e.g. the datatype units ("ng/mL", "Area").
yaxis_quant, yaxis_scalar	Quantile used to cap the y-axis, and the headroom multiplier applied above it.

Details

The y-axis is capped at ‘quantile(value, yaxis_quant) * yaxis_scalar‘ so a single large value cannot flatten the rest of the series; points above the cap are drawn outside the panel rather than dropped.

The returned plot carries a ‘text‘ aesthetic holding the per-point tooltip, which [plotly::ggplotly()] picks up with ‘tooltip = "text"‘; it is ignored by a plain ‘print()‘.

Value

A ‘ggplot‘.

See Also

Other QC check: addCVMetrics(), mrmPlotPCA(), mrmRunPCA(), plotLoadings(), plotQQ(), summariseFeatures()

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
plotDrift(de, "Analyte_02", value_label = "ng/mL")
```

plotGroupHeatmap *Group-mean heatmap that shows which samples produced the mean*

Description

The same picture [mrmPlotHeatmap()] draws - compounds on rows, samples on columns, a coloured annotation bar for the sample factor along the top and for the compound class down the left, gaps between blocks - with one thing changed: what a cell's colour means. The **hue** is the group's mean scaled value, and the **opacity** of each sample's cell is how much that sample contributed to it.

Usage

```
plotGroupHeatmap(
  de,
  color_samples_by,
  features = NULL,
  group_features_by = NULL,
  transform = c("none", "log2", "sqrt"),
  cap = NULL,
  group_scale = c("shared", "feature"),
  alpha_floor = 0.6,
  feature_labels = c("show", "small", "hide"),
  sample_labels = TRUE,
  sample_id_head = NULL,
  title = NULL
)
```

Arguments

de	A ‘struct::DatasetExperiment‘.
color_samples_by	‘sample_meta‘ column defining the groups to average within, and the annotation bar along the top. Named to match [mrmPlotHeatmap()], where it does the annotation half of the same job.
features	Compounds to include; ‘NULL‘ uses all of them.
group_features_by	‘variable_meta‘ column used to block compounds and colour the annotation bar down the left; ‘NULL‘ leaves them unblocked.
transform	‘"none"‘, ‘"log2"‘ or ‘"sqrt"‘, applied before scaling. Negative values are floored at 0 first.
cap	Colour limit, in standard deviations of the group means. ‘NULL‘ takes the 97th percentile of ‘abs()‘ of the group means, so a few cells saturate and the rest get the full range. Ignored when ‘group_scale‘ is ‘"feature"‘.

group_scale	‘"shared"‘ (default) draws every compound on one colour limit, so compounds can be compared with each other. ‘"feature"‘ gives each compound its own limit, so each row fills the colour range and only the shape along the groups can be read.
alpha_floor	Opacity given to a sample that did not contribute to its group’s mean. Below about 0.4 those cells read as missing data.
feature_labels	‘"show"‘, ‘"small"‘ or ‘"hide"‘. With ‘group_features_by‘ set, the names are drawn between the class colour bar and the cells - block, then names, then data, which is how a pathway figure is read. Without it they sit on the axis as usual.
sample_labels	Draw the sample axis. ‘FALSE‘ for wide cohorts, where the samples stay individually visible but not individually named.
sample_id_head	‘sample_meta‘ column supplying sample labels; ‘NULL‘ uses row names.
title	Plot title.

Details

A block that is uniformly solid is an effect the whole group shares. A block carrying one opaque cell among faded ones is a mean resting on a single sample - which is what a plain group-mean heatmap hides, and what a reader needs in order to judge whether a colour contrast is worth believing.

[mrmPlotHeatmap()] never averages, so it does not have this problem, and it is the better choice while the sample axis stays legible - in practice, while there are two groups. Past three the cells get narrow and the eye stops resolving individual samples anyway, which is where this earns its place.

Value

A ‘ggplot‘, or ‘NULL‘ when there is too little data to draw (fewer than 2 samples, or fewer than 2 compounds with any variance).

How a cell is built

Values are transformed, then scaled per compound across **all** samples, so a cell reads "high or low for this compound" rather than "abundant or not". Rows are therefore not comparable to each other in absolute terms.

The hue is the mean of those scores within a group. The opacity is the sample’s own score multiplied by the sign of its group mean, so a sample that moved with its group is opaque and one that sat near zero, or pulled the other way, fades. ‘alpha_floor‘ stops a non-contributing sample becoming invisible and reading as missing data; genuinely missing values are drawn in ‘na.value‘ grey at full opacity, so the two cannot be confused.

Note that "contributed" is measured against the mean of **all** samples, not against the group’s own members, so it is not the same as "has the largest value". A group of four holding one very high sample and three low ones has a *negative* mean if the three sit further from the overall centre than the one sits above it - and then the three are what produced that mean, and the high sample is the one opposing it and fading out. The opacity answers "did this sample push its group where the group ended up", not "is this sample large".

Choosing the colour limit

‘cap‘ defaults to a quantile of the group means actually being drawn, and that default matters more than it looks. Averaging *n* replicates shrinks a scaled value by roughly ‘sqrt(n)‘, so a limit chosen for per-sample scores - the ‘+/-2.5‘ that suits [mrmPlotHeatmap()] - leaves a group-mean panel using a small part of the colour range and looking washed out, whatever the opacity is set to. Take the limit from the quantity being drawn.

One cap or many

‘group_scale = "shared"‘ puts every compound on one limit, so a compound whose groups separate twice as far is drawn twice as saturated and the colour can be read across rows: pale means it moved less. What is compared is separation in units of that compound’s own spread, not fold change - a compound that doubles but scatters wildly sits below one that moves a fifth as much in every sample.

‘group_scale = "feature"‘ gives each compound its own limit, so each row fills the ramp whatever it did. Nothing is pale, so nothing can be ranked; use it to read one compound along the groups. ‘cap‘ is ignored, since the limit is each compound’s own largest group mean.

See Also

[mrmPlotHeatmap()], the same layout without the averaging.

Other stats: `mrmPlotHeatmap()`, `plotBoxplot()`, `plotCorrelation()`, `plotVolcano()`, `runStats()`, `summariseGroups()`

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
de <- subsetDataset(de, conditions = list(Sample_type = "Sample"))
plotGroupHeatmap(de, color_samples_by = "Treatment",
                  group_features_by = "Enzymatic_pathway")
```

---

plotLoadings *Plot PCA loadings as ranked bars, one panel per component*

Description

The scores plot says *whether* samples separate; the loadings say *which compounds* drove it. Each feature’s contribution to a component is drawn as a bar, sorted within its colour group so the compounds pulling hardest in each direction sit at the ends.

Usage

```
plotLoadings(pca, variable_meta, components = c(1, 2), colour_by = NULL)
```

Arguments

pca	The list returned by [mrmRunPCA()].
variable_meta	Feature metadata, or a 'DatasetExperiment' to take it from. Matched to the loadings by 'Compound'.
components	Length-2 integer vector: which components to show. Falls back to the first two if the fit has fewer.
colour_by	'variable_meta' column used to colour and group the bars; 'NULL' draws every bar in one colour.

Details

Colouring by a 'variable_meta' column - typically the enzymatic pathway - is what turns this from a ranked list into something interpretable: if the bars at one end of PC1 are predominantly one pathway, the separation in the scores plot has a biochemical reading rather than just a statistical one. The palette is shared with the heatmap feature annotation, so a class is the same colour in both figures.

Value

A 'ggplot', or 'NULL' if the PCA could not be fit.

See Also

[mrmPlotPCA()] for the matching scores plot.

Other QC check: addCVMetrics(), mrmPlotPCA(), mrmRunPCA(), plotDrift(), plotQQ(), summariseFeatures()

Examples

```
de  <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                               package = "MRManalyzeR"))
pca <- mrmRunPCA(de, transform = "log2")
plotLoadings(pca, de, colour_by = "Enzymatic_pathway")
```

---

plotQQ *Normal Q-Q plot for one compound*

Description

Plots the sample quantiles of one compound against theoretical normal quantiles, with the reference line, and reports the Shapiro-Wilk statistic in the title. Points falling along the line mean the parametric tests in [runStats()] are on safe ground for that compound; a curve means they are not, and the Wilcoxon or Kruskal-Wallis result reported alongside is the one to trust.

Usage

```
plotQQ(
  de,
  compound,
  transform = c("none", "log2", "sqrt"),
  sample_type_head = "Sample_type",
  levels = NULL,
  value_label = "value"
)
```

Arguments

de	A ‘struct::DatasetExperiment‘.
compound	Feature (column) name to plot.
transform	One of ‘"none"‘, ‘"log2"‘ (log2(x + 1)) or ‘"sqrt"‘.
sample_type_head	‘sample_meta‘ column classifying sample type.
levels	Value(s) of ‘sample_type_head‘ to include; ‘NULL‘ keeps all.
value_label	Axis label for the measured value, e.g. the datatype units.

Details

Targeted panels are frequently right-skewed, so the same compound is worth looking at under more than one ‘transform‘ - which is why the data-quality report renders a tab per transform rather than picking one.

Only the biological samples should normally be included: blanks and QCs are different populations and would distort the distribution. Restrict with ‘levels‘.

Value

A ‘ggplot‘. When fewer than three non-missing values are available the plot is an empty panel whose title says so, so a report loop does not have to special-case sparse compounds.

See Also

Other QC check: addCVMetrics(), mrmPlotPCA(), mrmRunPCA(), plotDrift(), plotLoadings(), summariseFeatures()

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
plotQQ(de, "Analyte_02", transform = "log2", levels = "Sample")
```

`plotVolcano` *Volcano plot for one comparison*

Description

Effect size against significance for every compound in a comparison: log2 fold-change on x, '-log10(p)' on y. Together they say what neither says alone - a large fold-change on three noisy animals is not a finding, and a tiny but exquisitely reproducible shift usually is not interesting either. The compounds worth attention sit in the upper corners.

Usage

```
plotVolcano(
  stats,
  comparison = NULL,
  pair = NULL,
  use_adjusted = FALSE,
  sig_threshold = 0.05,
  top_n_label = 10
)
```

Arguments

`stats`	The 'stats' data frame from [runStats()], or the whole list.
`comparison`	Name of the comparison to plot. 'NULL' uses the only one present, and errors if the table holds more than one.
`pair`	For post-hoc rows, the group pair as '"A vs B"'. 'NULL' uses the two-group test rows.
`use_adjusted`	Plot BH-adjusted p on the y-axis instead of raw p.
`sig_threshold`	Significance cut-off; drawn as a dashed line and used to colour and label points.
`top_n_label`	Maximum number of significant features to label.

Details

Rows are taken from the 'stats' table returned by [runStats()], so the plot and the 'stats' sheet of the output xlsx are the same numbers. Only rows carrying a fold change are eligible, which excludes the global tests ('anova', 'kruskal') - they ask whether *any* group differs and have no direction. A two-level comparison contributes its single test row; a three-or-more-level comparison contributes one volcano per post-hoc pair, selected with 'pair'.

Every significant feature is labelled, up to 'top_n_label'; beyond that the most significant are kept, since a volcano labelled with eighty compounds communicates nothing.

Value

A 'ggplot', or 'NULL' when the comparison has no plottable rows.

See Also

Other stats: mrmPlotHeatmap(), plotBoxplot(), plotCorrelation(), plotGroupHeatmap(), runStats(), summariseGroups()

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
de <- subsetDataset(de, conditions = list(Sample_type = "Sample"))
params <- list(comparisons = list(enabled = TRUE, entries = list(
  list(name = "PBS_vs_HDM", method = "welch",
       compare = list(factor = "Treatment",
                      levels = c("PBS", "HDM"))))))
st <- runStats(de, params)
plotVolcano(st, "PBS_vs_HDM")
```

---

print.mrm_validation *Print a validation result*

Description

Print a validation result

Usage

```
## S3 method for class 'mrm_validation'
print(x, ...)
```

Arguments

x	An ‘mrm_validation‘ object.
...	Ignored.

Value

‘x‘, invisibly.

processDataset *Build a sample x compound matrix from TargetLynx or Skyline input*

Description

Applies (optionally) signal filtering (SNR for TargetLynx; LOD/LOQ for Skyline), blank and detection filtering, batch correction, normalisation, concentration adjustment and missing-value imputation to produce a 'struct::DatasetExperiment'.

Usage

```
processDataset(
  fdata,
  metadata,
  xlsx_path = NULL,
  data_source = "targetlynx",
  data_tab_names = NULL,
  datatype = "Area",
  tl_headers = c("ID", "Name", "Area", "ng/mL", "Response", "S/N"),
  signal_filter = NULL,
  snr = 5,
  blank_filter = 5,
  replace_MVs = FALSE,
  impute_method = "gaussian",
  impute_seed = 42,
  batch_correction = FALSE,
  bc_qc_label = "QC",
  bc_factor_name = "Sample_type",
  bc_header = "Chrom_Batch",
  bc_check_factor = NULL,
  processing_batch = NULL,
  matrix_id_col = NULL,
  matrix_orientation = "samples_rows",
  blank_head = "Sample_type",
  filter_features = FALSE,
  filter_samples = FALSE,
  blank_name = "Blank",
  normalize = FALSE,
  adjust_conc = FALSE,
  starting_vol_col = FALSE,
  sample_vol_col = "sample_volume_uL",
  cal_vol_col = "cal_vol_uL",
  sample_IS_col = "sample_IS_vol_uL",
  cal_IS_col = "cal_IS_vol_uL",
  name_col = "Name",
  include_col = "Include",
  include_value = "YES",
```

```
  compound_col = "Compound",
  processing_name_col = "Processing_name",
  report_col = "Report",
  report_value = "YES",
  comment_col = "Comment"
)
```

Arguments

fdata	Feature metadata. Must contain the columns named by 'compound_col', 'processing_name_col', 'report_col' (defaults 'Compound' / 'Processing_name' / 'Report'). For 'signal_filter = "LOD"' or '"LOQ"', must also contain the matching 'LOD'/'LOQ' column.
metadata	Sample metadata. Must contain the columns named by 'name_col', 'include_col' and the headers referenced by 'bc_header', 'bc_factor_name', 'blank_head'.
xlsx_path	Path to the source workbook (read by [readTargetLynx()] or [readSkyline()]).
data_source	'"targetlynx"' (default), '"skyline"' or '"matrix"' - selects the reader. '"matrix"' is the vendor-neutral route via [readPeakMatrix()]: any software that can export a rectangular sample x analyte table can be used without a dedicated parser.
data_tab_names	Sheet name(s) to read from 'xlsx_path'. 'NULL' (default) uses each reader's usual sheets: every sheet whose name contains '"lcms_data"' for '"targetlynx"', '"skyline_data"' for '"skyline"' and '"matrix_data"' for '"matrix"'. Multiple Skyline sheets are treated as separate acquisition batches and row-bound together - see '.read_skyline_matrix()'. Sheet membership does not itself imply batch-correction grouping; that remains driven by 'sample_metadata'.
datatype	TargetLynx column to report (e.g. "Area", "Response", "ng/mL"). Only used when 'data_source = "targetlynx"'.
tl_headers	TargetLynx column headers to extract (passed to [readTargetLynx()]).
signal_filter	'"SNR"', '"LOD"', '"LOQ"', or 'FALSE'. '"SNR"' is only valid for 'data_source = "targetlynx"'; '"LOD"'/'"LOQ"' are only valid for 'data_source = "skyline"' or '"matrix"'. Default 'NULL' infers '"SNR"' unless 'snr = FALSE', for backwards compatibility with configs that only set 'snr:'.
snr	S/N threshold. Only consulted when 'signal_filter = "SNR"'.
blank_filter	Blank-filter factor; 'FALSE' or 'NULL' to skip.
replace_MVs	Scalar passed to ['imputeMissing()']; 'FALSE' or 'NULL' to skip. Applied last, after batch correction, normalisation and concentration adjustment, per 'processing_batch'.
impute_method, impute_seed	Passed to [imputeMissing()] as 'method' and 'seed': '"gaussian"' (default) draws each fill at random below the per-feature ceiling, '"constant"' uses the ceiling itself.
batch_correction	'FALSE' (default) to skip, 'TRUE' to correct every feature, or the name of an 'fdata' column holding 'YES' / 'NO' per feature, so that only the 'YES' features are corrected. Runs after the blank and detection filters, on the instrument signal, and before normalisation, concentration adjustment and imputation - see [correctBatch()].

`bc_qc_label`, `bc_factor_name`, `bc_header`
: Batch-correction parameters: the reference-sample label, the column holding it, and the batch column. Defaults to pooled QCs, which are the only reference guaranteed not to differ between batches for biological reasons - see [correctBatch()].

`bc_check_factor`
: Optional study factor; when given, ‘correctBatch()‘ warns if batch is confounded with it.

`processing_batch`
: ‘sample_meta‘ column used to partition the *per-batch processing* - blank filtering and imputation are each applied within a partition, so blanks and per-feature minima are taken from the same acquisition as the samples they apply to. ‘NULL‘ (default) reuses ‘bc_header‘, which is what earlier versions did implicitly; ‘FALSE‘ processes every sample together. This is a separate question from which column defines the batch-correction reference, even when the same column answers both.

`matrix_id_col`, `matrix_orientation`
: Passed to [readPeakMatrix()] when ‘data_source = "matrix"‘: the sample-identifier column (‘NULL‘ = first) and whether samples are rows or columns.

`blank_head`, `blank_name`
: ‘sample_meta‘ column and value identifying blank injections. The defaults are the canonical names this package documents (‘Sample_type‘ / ‘Blank‘), not a particular laboratory’s sheet - set them to whatever your workbook uses.

`filter_features`
: ‘FALSE‘ (default) to skip, or a named list of arguments for [filterFeatures()] - ‘min_frac‘, ‘method‘, ‘group_col‘, ‘sample_col‘, ‘sample_labels‘, ‘qc_label‘. Applied once to the whole dataset after blank masking and before imputation, never per batch.

`filter_samples`
: ‘FALSE‘ (default) to skip, or a named list of arguments for [filterSamples()] - ‘max_na‘, ‘sample_col‘, ‘sample_labels‘. Runs after ‘filter_features‘, on the reduced feature set, so the missing-value fraction means something.

`normalize`
: Metadata column used as divisor, or ‘FALSE‘/‘NULL‘ for none.

`adjust_conc`
: Logical master toggle for concentration adjustment via [adjustConcentration()], using ‘sample_vol_col‘, ‘cal_vol_col‘, ‘sample_IS_col‘, ‘cal_IS_col‘ and ‘starting_vol_col‘.

`starting_vol_col`
: ‘FALSE‘ to skip the starting-volume correction, or the ‘metadata‘ column holding each sample’s pre-dry-down starting volume.

`sample_vol_col`, `cal_vol_col`, `sample_IS_col`, `cal_IS_col`
: ‘metadata‘ columns holding, respectively: the vial reconstitution volume, the calibration-standard vial volume, the IS volume added to the sample, and the IS volume added to the calibration standard. Only consulted when ‘adjust_conc = TRUE‘; different studies name these columns differently, hence configurable rather than hardcoded.

`name_col`
: ‘sample_metadata‘ column whose values match the data-matrix row identifiers (LC-MS injection / sample names). Default ‘"Name"‘.

`include_col`, `include_value`
: ‘sample_metadata‘ inclusion filter: rows where ‘metadata[[include_col]] == include_value‘ are processed. Defaults ‘"Include"‘ / ‘"YES"‘.

`compound_col`, `processing_name_col`
: ‘feature_metadata‘ columns holding the canonical feature name (becomes the data-matrix column names) and the raw name that joins to the source data. Defaults ‘"Compound"‘ / ‘"Processing_name"‘.

`report_col`, `report_value`
: ‘feature_metadata‘ inclusion filter: features where ‘fdata[[report_col]] == report_value‘ are reported. Defaults ‘"Report"‘ / ‘"YES"‘. Optional - when ‘report_col‘ is not a column of ‘fdata‘ every feature is reported, which is the normal case for a feature table curated outside a TargetLynx workflow.

`comment_col`
: ‘feature_metadata‘ column used to record why a feature was excluded. Default ‘"Comment"‘.

Details

The peak-matrix workflow behind [runMRManalyzeR()], composed of the public step functions, in this order: it reads the workbook ([readTargetLynx()] / [readSkyline()]) into a wide sample x compound matrix and assembles it into a ‘struct::DatasetExperiment‘ ([assembleDataset()]); applies [filterBlanks()] per acquisition batch, then [filterFeatures()] and [filterSamples()] to the whole dataset; optionally batch-corrects ([correctBatch()]); applies [normaliseMatrix()] and [adjustConcentration()]; and imputes last ([imputeMissing()], per batch). Batch correction works on the instrument signal, before any per-sample rescaling, and imputation comes last so every earlier step sees only measured values.

A value of 0 is read as a non-detect and set to ‘NA‘ straight after reading, so no step counts it as a measurement: the detection filters count it as missing, the blank filter leaves it out of the blank mean, and a matrix that is not imputed shows a gap.

Value

A named list with ‘dataset‘ (the ‘DatasetExperiment‘, imputed when ‘replace_MVs‘ is set), ‘excluded_features‘ (a data frame of features dropped along the way, with ‘Failed_step‘ naming the step that took each one - ‘User exclude‘, ‘Detection filter‘, ‘Signal filter (...)‘, ‘Never measured‘, ‘Not in source data‘ or ‘No metadata entry‘ - and the reason recorded) and ‘measured‘ (the same dataset before imputation, gaps included; identical to ‘dataset‘ when nothing was imputed). The first two stay in that order, so existing code indexing ‘[[1]]‘ and ‘[[2]]‘ keeps working.

See Also

[runMRManalyzeR()] to drive the whole workflow from a YAML config.

Other peak-matrix processing: `adjustConcentration()`, `correctBatch()`, `filterBlanks()`, `filterFeatures()`, `filterSamples()`, `imputeMissing()`, `normaliseMatrix()`, `subsetDataset()`

Examples

```
xlsx  <- system.file("extdata", "example_data.xlsx", package = "MRManalyzeR")
fdata <- openxlsx::read.xlsx(xlsx, sheet = "feature_metadata")
```

```
meta  <- openxlsx::read.xlsx(xlsx, sheet = "sample_metadata")
out <- processDataset(fdata, meta, xlsx_path = xlsx,
                      data_source = "targetlynx", datatype = "Area",
                      data_tab_names = "lcms_data", snr = 3,
                      blank_filter = FALSE, bc_header = "Chrom_Batch",
                      blank_head = "Sample_type")
out[[1]]
```

`readPeakMatrix` *Read a plain sample x compound matrix*

Description

The vendor-neutral route in. ‘readTargetLynx()‘ and ‘readSkyline()‘ exist because those two exports are awkward shapes; anything that can produce a rectangular table of samples against analytes needs no parser, only a documented schema. That covers Sciex MultiQuant, Agilent MassHunter, Thermo TraceFinder, and the hand-curated spreadsheets that most laboratories end up with at least once.

Usage

```
readPeakMatrix(
  x,
  data_tab_names = "matrix_data",
  id_col = NULL,
  orientation = c("samples_rows", "samples_cols")
)
```

Arguments

`x`	Path to an xlsx workbook, or a data frame / matrix already in memory.
`data_tab_names`	Sheet name(s) to read when ‘x‘ is a path. Multiple sheets are row-bound, so an acquisition split across sheets can be given as a vector; sample identifiers must be unique across all of them.
`id_col`	The label column - the one holding identifiers rather than measurements. It is dropped from the numeric matrix and used as row names. ‘NULL‘ uses the first column. What it identifies follows ‘orientation‘: sample names under ‘"samples_rows"‘, compound names under ‘"samples_cols"‘, with the other axis coming from the header row.
`orientation`	‘"samples_rows"‘ (default: one row per sample) or ‘"samples_cols"‘ (one row per analyte, transposed on read).

Details

The schema is deliberately minimal: one identifier column, then one column per analyte (or the transpose, with ‘orientation = "samples_cols"‘). The identifiers must match ‘sample_metadata‘’s name column, and the analyte names must match ‘feature_metadata‘; everything else - filtering, normalisation, statistics - is the same from here on.

No signal filtering is applied, because a generic matrix carries no S/N or LOD information. Mask values before import, or supply an ‘LOD‘/‘LOQ‘ column in ‘feature_meta‘ and use [processDataset()] with ‘signal_filter‘.

Value

A data frame, rows = samples, columns = compounds, numeric - the same contract as [readTargetLynx()] and [readSkyline()].

See Also

Other data parse: `assembleDataset()`, `combineDatasets()`, `loadConfig()`, `loadDataset()`, `mrmValidateConfig()`, `mrmValidateDesign()`, `mrmValidateInput()`, `readSkyline()`, `readTargetLynx()`

Examples

```
m <- data.frame(Name = c("S1", "S2", "S3"),
                PGE2 = c(120, 95, 140),
                PGD2 = c(45, 51, 38))
readPeakMatrix(m)
```

---

readSkyline *Read Skyline sheet(s) into a wide sample x compound matrix*

Description

Reads the Skyline "molecule x sample" sheet(s), renaming molecule columns to the canonical ‘Processing_name‘ where matched in ‘feature_meta‘, and optionally masks values below the per-compound ‘LOD‘/‘LOQ‘ threshold.

Usage

```
readSkyline(
  xlsx_path,
  data_tab_names = "skyline_data",
  feature_meta,
  signal_filter = FALSE
)
```

Arguments

`xlsx_path`	Path to the Skyline xlsx workbook.
`data_tab_names`	Sheet name(s) holding the molecule x sample matrix.
`feature_meta`	Feature metadata (canonical ‘Processing_name‘ / ‘Report‘ columns; plus the ‘LOD‘/‘LOQ‘ column when ‘signal_filter‘ is set).
`signal_filter`	‘"LOD"‘, ‘"LOQ"‘, or ‘FALSE‘ - the per-compound floor below which values are set to ‘NA‘.

Value

A data frame, rows = samples, columns = compounds, numeric.

See Also

Other data parse: `assembleDataset()`, `combineDatasets()`, `loadConfig()`, `loadDataset()`, `mrmValidateConfig()`, `mrmValidateDesign()`, `mrmValidateInput()`, `readPeakMatrix()`, `readTargetLynx()`

Examples

```
sky <- data.frame(Molecule = c("PGE2", "PGD2"),
                  S1 = c(100, 50), S2 = c(120, 55))
f <- tempfile(fileext = ".xlsx")
openxlsx::write.xlsx(list(skyline_data = sky), f)
fm <- data.frame(Processing_name = c("PGE2", "PGD2"), Report = "YES",
                 row.names = c("PGE2", "PGD2"))
readSkyline(f, "skyline_data", fm)
```

---

readTargetLynx *Read a TargetLynx workbook into a wide sample x compound matrix*

Description

Reads the TargetLynx sheet(s) with [extractTable()], pivots the long table to a wide samples x compound matrix of the requested ‘datatype‘, and optionally masks values whose per-injection S/N is below ‘snr‘. Columns are the raw TargetLynx compound (‘Processing_name‘) identifiers; the rename-to-‘Compound‘ and feature/sample filtering happen later in [processDataset()].

Usage

```
readTargetLynx(
  xlsx_path,
  datatype = "Area",
  tl_headers = c("ID", "Name", "Area", "ng/mL", "Response", "S/N"),
  snr = FALSE,
  data_tab_names = NULL
)
```

Arguments

`xlsx_path`	Path to the TargetLynx xlsx workbook.
`datatype`	TargetLynx column to report (e.g. "Area", "Response", "ng/mL").
`tl_headers`	TargetLynx column headers to extract.
`snr`	S/N threshold; values below it are set to ‘NA‘. ‘FALSE‘ to skip.
`data_tab_names`	Sheet name(s) to read; ‘NULL‘ auto-matches sheets whose name contains "lcms_data".

Value

A data frame, rows = samples, columns = compounds (‘Processing_name‘), numeric.

See Also

Other data parse: `assembleDataset()`, `combineDatasets()`, `loadConfig()`, `loadDataset()`, `mrmValidateConfig()`, `mrmValidateDesign()`, `mrmValidateInput()`, `readPeakMatrix()`, `readSkyline()`

Examples

```
xlsx <- system.file("extdata", "example_data.xlsx", package = "MRManalyzeR")
m <- readTargetLynx(xlsx, datatype = "Area", snr = 3)
dim(m)
```

---

runMRManalyzeR *Run the post-acquisition MRManalyzeR workflow from a YAML config*

Description

Single entry point that:

- loads a project YAML,
- (optionally) reads the TargetLynx xlsx workbook and builds the processed peak matrix - gated by ‘PeakMatrixProcessing.execute‘,
- writes the processed matrix to xlsx + RDS and persists the YAML parameters alongside - imputed when ‘replace_MVs‘ is set, with the measured values added as a ‘matrix_measured‘ sheet and ‘<name>_measured.RDS‘,
- runs the statistical analyses defined under ‘stats_report:‘ (comparisons / correlations / linear_models) and writes them as extra tabs in the results xlsx,
- (optionally) renders the bundled ‘data_quality_report‘ and ‘stats_report‘ HTML vignettes.

Usage

```
runMRManalyzeR(path_yaml)
```

Arguments

`path_yaml`	Full path to a project YAML.

Details

Per-compound quality metrics are written into the returned dataset's ‘variable_meta‘, matched to the feature they describe: ‘CV_QC‘ (coefficient of variation across the pooled QC injections - technical variability, and the usual quantification filter), ‘CV_sample‘ (the same across the biological samples, so biological spread plus that technical noise), and ‘CV_sample_vs_QC‘. A ratio near 1 flags a compound varying no more between samples than between replicate injections of identical material. ‘CV_QC‘ is ‘NA‘ when a run contains no QC injections. They are computed on the measured values, never on imputed ones.

Both reports and the statistics use the measured values unless their block sets ‘use_imputed: True‘. The PCAs use the values ‘replace_MVs‘ imputed when it is set, and otherwise impute for themselves (‘preprocess: impute:‘).

Output filenames are derived from ‘paths.fn‘ + ‘datatype‘ + ‘paths.suffix‘, so when ‘PeakMatrixProcessing.execute: False‘ the code can still locate the existing RDS without re-reading the source xlsx. Those three must therefore match the stored file; if they do not, the error lists the ‘.RDS‘ files that are in ‘paths.result_dir‘.

‘datatype‘ is read from the enabled data-source block: ‘datatype:‘ under ‘tl_data:‘ / ‘matrix_data:‘, and ‘signal_filter:‘ under ‘skyline_data:‘, where the LOD / LOQ floor is what distinguishes one run from another. Any of them may be a list, in which case the whole workflow runs once per entry.

Backwards compatibility:

- ‘datatype‘ falls back to ‘PeakMatrixProcessing:‘ and then ‘paths:‘, where earlier configs put it. Setting it in two places warns.
- Legacy ‘UVA_report:‘ / ‘MVA_report:‘ blocks are recognised when ‘data_quality_report:‘ / ‘stats_report:‘ are missing.

Value

Invisibly, a list with the ‘DatasetExperiment‘ (imputed when ‘replace_MVs‘ is set), the ‘measured‘ one, ‘removed_features‘, ‘stats_tables‘, and the resolved output paths. When ‘datatype‘ is a vector (e.g. ‘["Area", "Response", "Conc"]‘), the function loops over each datatype and returns a list of per-datatype results.

See Also

Other entry points: `mrmRunExample()`, `newConfig()`, `runMRManalyzeRCombine()`

Examples

```
# mrmRunExample() writes a config for the bundled dataset and drives
# runMRManalyzeR(); render = FALSE keeps it to the peak-matrix build.
mrmRunExample(render = FALSE, open = FALSE)
```

runMRManalyzeRCombine *Run the combine-mode workflow from a dedicated combine YAML*

Description

Merges several previously-saved datasets (‘.RDS‘ or ‘.xlsx‘ outputs of [‘runMRManalyzeR()‘]) into one ‘DatasetExperiment‘, then runs the ‘stats_report‘ analyses on the merged data. Skips PeakMatrixProcessing and the data_quality_report.

Usage

```
runMRManalyzeRCombine(path_yaml)
```

Arguments

path_yaml	Full path to the combine YAML.

Details

Defaults: samples are matched by ‘Sample_ID‘ (intersect across inputs); ‘feature_meta_cols‘ and ‘sample_meta_cols‘ default to the intersection of column names across inputs (specify them only to *narrow* the set).

Outputs:

- ‘<output_stub>.RDS‘ - merged ‘DatasetExperiment‘
- ‘<output_stub>.xlsx‘ - feature_metadata / sample_metadata / matrix tabs
- ‘<output_stub>_measured.RDS‘ - the same merge of the panels’ measured values, when any panel stored a ‘_measured.RDS‘; the xlsx then gains a ‘matrix_measured‘ tab
- ‘<output_stub>_stats.xlsx‘ - key / stats / correlations / linear_models
- ‘<output_stub>_stats_report.html‘

A panel run with ‘replace_MVs:‘ exports an imputed matrix and stores its measured values beside it as ‘<name>_measured.RDS‘. The merged ‘.RDS‘ and ‘matrix‘ tab hold what each panel exported; the statistics and the report use the merge of the measured values unless ‘stats_report‘ sets ‘use_imputed: True‘.

YAML schema (see ‘inst/extdata/example_combine_config.yml‘):

```
combine:
  datasets:
    - path: ".../GOM_..._.RDS"      # .RDS or .xlsx, mix is fine
      tag:  "GOM"
      qc_remap: { "QC1": "QC_pooled" }
    - path: ".../cysLT_..._.xlsx"
      tag:  "cysLT"
  feature_meta_cols: ~              # NULL = intersect across inputs
  sample_meta_cols:  ~              # NULL = intersect across inputs
```

```
  prefix_features:   True
  sample_id_col:     Sample_ID
  output_stub:       ".../Combined/combined_HDM"
  units:             "combined"
stats_report:
  execute: True
  use_imputed: False            # True = statistics on the imputed merge
  comparisons:   [...]
  correlations:  [...]
  linear_models: [...]
```

Value

Invisibly, a list with the merged 'DatasetExperiment', its 'measured' counterpart (the same object when no panel stored one), the 'stats_tables', and the output paths.

See Also

Other entry points: mrmRunExample(), newConfig(), runMRManalyzeR()

Examples

```
# Two tiny processed panels (normally .RDS/.xlsx outputs of
# runMRManalyzeR()) that share sample IDs, merged via a combine YAML.
# stats_report execute = FALSE keeps the example to the merge itself.
dir <- tempfile("combine_"); dir.create(dir)
mk <- function(feats) struct::DatasetExperiment(
  data          = as.data.frame(matrix(1, 3, length(feats),
                    dimnames = list(c("S1", "S2", "S3"), feats))),
  sample_meta   = data.frame(Sample_ID = c("S1", "S2", "S3"),
                    Sample_type = "Sample",
                    row.names = c("S1", "S2", "S3")),
  variable_meta = data.frame(Compound = feats, Class = "lipid",
                    row.names = feats))
p1 <- file.path(dir, "panelA.RDS"); saveRDS(mk(c("A", "B")), p1)
p2 <- file.path(dir, "panelB.RDS"); saveRDS(mk(c("C", "D")), p2)
cfg <- list(
  combine = list(
    datasets        = list(list(path = p1, tag = "A"),
                           list(path = p2, tag = "B")),
    prefix_features = TRUE,
    sample_id_col   = "Sample_ID",
    output_stub     = file.path(dir, "combined")),
  stats_report = list(execute = FALSE))
yml <- file.path(dir, "combine.yml"); yaml::write_yaml(cfg, yml)
runMRManalyzeRCombine(yml)
```

`runStats` *Run all statistics defined by the YAML 'stats_report:' block*

Description

Iterates over 'comparisons:', 'correlations:', 'linear_models:' and 'ion_ratios:' and returns tidy data frames suitable for writing to xlsx tabs.

Usage

```
runStats(de, st_params)
```

Arguments

`de`	A 'struct::DatasetExperiment' (from 'runMRManalyzeR()' or 'readRDS()').
`st_params`	The 'stats_report:' block from the YAML (a list).

Details

Each comparison names its own test, because the choice has to be made before the p-values are seen:

- 2 levels -> 'welch' (default), 'student', or 'wilcoxon'
- 3+ levels -> 'anova' (default, with Tukey HSD post-hoc) or 'kruskal' (with Holm-adjusted pairwise Wilcoxon post-hoc)

A comparison with no 'method:' falls back to the default and says so, so existing configurations keep running.

Add 'paired: true' together with 'subject:' naming the 'sample_meta' column that links the two observations of each subject - pre/post, matched tissues, the same animal at two timepoints. Only subjects present in both groups contribute, and the effect size becomes Cohen's dz (or the matched-pairs rank-biserial correlation for 'wilcoxon'), which is *not* comparable to the between-group values and is labelled distinctly so the two are never pooled. The method label gains a '_paired' suffix for the same reason.

Multiple-testing correction is applied *within families*, not across the whole table: one family per comparison and method, per correlation analysis and subset and method, and per linear-model term. The family used for each row is recorded in its 'adjust_family' column. Adjusting across unrelated analyses would make the size of the correction depend on how many other things happened to be configured in the same YAML.

Value

A list with elements 'stats', 'correlations', 'linear_models', each a data frame (possibly with 0 rows if the corresponding YAML block was empty).

See Also

Other stats: mrmPlotHeatmap(), plotBoxplot(), plotCorrelation(), plotGroupHeatmap(), plotVolcano(), summariseGroups()

Examples

```
m  <- data.frame(A = c(10, 12, 11, 20, 24, 22), B = c(5, 6, 5, 11, 13, 12),
                 row.names = paste0("S", 1:6))
fm <- data.frame(Compound = c("A", "B"), row.names = c("A", "B"))
sm <- data.frame(Sample_ID = paste0("S", 1:6),
                 Group = rep(c("ctrl", "trt"), each = 3),
                 row.names = paste0("S", 1:6))
de <- struct::DatasetExperiment(data = m, sample_meta = sm, variable_meta = fm)
params <- list(comparisons = list(enabled = TRUE, entries = list(
  list(name = "ctrl_vs_trt", method = "welch",
       compare = list(factor = "Group", levels = c("ctrl", "trt"))))))
runStats(de, params)$stats
```

`subsetDataset` *Subset a 'DatasetExperiment' by sample-metadata conditions*

Description

Filters 'sample_meta' to rows matching all 'conditions' (a named list of 'column = value' pairs, joined by AND) and slices 'data' to the matching samples. Optionally restricts to a feature subset.

Usage

```
subsetDataset(
  de,
  conditions = NULL,
  features = NULL,
  drop_empty_features = TRUE
)
```

Arguments

de	A 'struct::DatasetExperiment'.
conditions	A named list, e.g. 'list(Treatment = "HDM", Sex = "Male")'. Values may be a scalar or vector (interpreted as ' empty list returns the input unchanged on the sample dimension.
features	Optional character vector of feature (column) names to retain. 'NULL' keeps all features.
drop_empty_features	If 'TRUE' (default), features that are entirely 'NA' after subsetting are dropped from both 'data' and 'variable_meta'.

Value

A new ‘DatasetExperiment‘ with ‘data‘, ‘sample_meta‘, and ‘variable_meta‘ consistently sliced.

See Also

Other peak-matrix processing: `adjustConcentration()`, `correctBatch()`, `filterBlanks()`, `filterFeatures()`, `filterSamples()`, `imputeMissing()`, `normaliseMatrix()`, `processDataset()`

Examples

```
m  <- data.frame(A = c(10, 20, 30), B = c(1, 2, 3),
                 row.names = c("S1", "S2", "S3"))
fm <- data.frame(Compound = c("A", "B"), row.names = c("A", "B"))
sm <- data.frame(Sample_ID = c("S1", "S2", "S3"), Group = c("x", "y", "x"),
                 row.names = c("S1", "S2", "S3"))
de <- struct::DatasetExperiment(data = m, sample_meta = sm, variable_meta = fm)
subsetDataset(de, conditions = list(Group = "x"))
```

---

summariseFeatures *Summarise which compounds were reported and which were dropped*

Description

Returns one row per compound the run started with, marking whether it survived into the reported matrix and why not if it did not. Exclusions accumulate from several places - signal-to-noise or LOD/LOQ masking that left a feature entirely missing, features whose ‘Processing_name‘ had no rows in the source data, and source columns with no ‘feature_metadata‘ entry - so the reason recorded in ‘Comment‘ is often the only trace of what happened to a compound.

Usage

```
summariseFeatures(de, removed_features = NULL)
```

Arguments

`de`	A ‘struct::DatasetExperiment‘ - the reported dataset.
`removed_features`	Optional data frame of dropped features, as returned in ‘[[2]]‘ by [processDataset()]. ‘NULL‘ (or zero rows) yields an included-only table, which is what happens when a run reuses a stored RDS rather than reprocessing.

Details

This is the table the data-quality report shows as "compounds included" and "compounds excluded"; keeping it as one frame with a ‘Status‘ column makes it sortable and searchable in one place, and writable to a single sheet.

Value

A data frame with columns ‘Compound‘, ‘Status‘ (‘"included"‘ / ‘"excluded"‘) and ‘Comment‘.

See Also

Other QC check: addCVMetrics(), mrmPlotPCA(), mrmRunPCA(), plotDrift(), plotLoadings(), plotQQ()

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
head(summariseFeatures(de))
```

---

summariseGroups *Per-group summary statistics for every compound*

Description

The numbers behind a boxplot: n, mean, standard deviation and standard error for each compound in each group. [plotBoxplot()] draws bars and error bars from exactly these values, and [runMRManalyzeR()] writes them to the ‘summary‘ sheet of the stats workbook - so what a reader measures off the figure and what they read in the spreadsheet are the same numbers.

Usage

```
summariseGroups(de, group_by, compounds = NULL)
```

Arguments

de	A ‘struct::DatasetExperiment‘.
group_by	‘sample_meta‘ column defining the groups.
compounds	Features to summarise; ‘NULL‘ uses all of them.

Details

Standard deviation describes the spread of the animals; standard error describes how well the mean is pinned down, and shrinks with n. They answer different questions and the choice belongs to the reader, so both are returned and ‘plotBoxplot(error_bar =)‘ picks one.

Value

A data frame with one row per compound per group: ‘Compound‘, ‘group‘, ‘n‘ (non-missing values), ‘mean‘, ‘sd‘, ‘se‘.

See Also

Other stats: `mrmPlotHeatmap()`, `plotBoxplot()`, `plotCorrelation()`, `plotGroupHeatmap()`, `plotVolcano()`, `runStats()`

Examples

```
de <- loadDataset(system.file("extdata", "example_synthetic.RDS",
                              package = "MRManalyzeR"))
de <- subsetDataset(de, conditions = list(Sample_type = "Sample"))
head(summariseGroups(de, "Treatment"))
```

Index

∗ **QC check**
addCVMetrics, 3
mrmPlotPCA, 21
mrmRunPCA, 23
plotDrift, 33
plotLoadings, 37
plotQQ, 38
summariseFeatures, 55
∗ **data parse**
assembleDataset, 5
combineDatasets, 6
loadConfig, 16
loadDataset, 17
mrmValidateConfig, 25
mrmValidateDesign, 26
mrmValidateInput, 27
readPeakMatrix, 46
readSkyline, 47
readTargetLynx, 48
∗ **entry points**
mrmRunExample, 22
newConfig, 28
runMRManalyzeR, 49
runMRManalyzeRCombine, 51
∗ **peak-matrix processing**
adjustConcentration, 4
correctBatch, 8
filterBlanks, 10
filterFeatures, 12
filterSamples, 13
imputeMissing, 15
normaliseMatrix, 29
processDataset, 42
subsetDataset, 54
∗ **stats**
mrmPlotHeatmap, 19
plotBoxplot, 30
plotCorrelation, 32
plotGroupHeatmap, 35
plotVolcano, 40
runStats, 53
summariseGroups, 56

addCVMetrics, 3
addCVMetrics(), *22*, *25*, *34*, *38, 39*, *56*
adjustConcentration, 4
adjustConcentration(), *10, 11*, *13, 14*, *16*, *30*, *45*, *55*
assembleDataset, 5
assembleDataset(), *8*, *17, 18*, *25, 26*, *28*, *47–49*

combineDatasets, 6
combineDatasets(), *6*, *17, 18*, *25, 26*, *28*, *47–49*
correctBatch, 8
correctBatch(), *5*, *11*, *13, 14*, *16*, *30*, *45*, *55*

filterBlanks, 10
filterBlanks(), *5*, *10*, *13, 14*, *16*, *30*, *45*, *55*
filterFeatures, 12
filterFeatures(), *5*, *10, 11*, *14*, *16*, *30*, *45*, *55*
filterSamples, 13
filterSamples(), *5*, *10, 11*, *13*, *16*, *30*, *45*, *55*

imputeMissing, 15
imputeMissing(), *5*, *10, 11*, *13, 14*, *30*, *45*, *55*

loadConfig, 16
loadConfig(), *6*, *8*, *18*, *25, 26*, *28*, *47–49*
loadDataset, 17
loadDataset(), *6*, *8*, *17*, *25, 26*, *28*, *47–49*

mrm_validation, 18
mrmPlotHeatmap, 19
mrmPlotHeatmap(), *31*, *33*, *37*, *41*, *54*, *57*
mrmPlotPCA, 21
mrmPlotPCA(), *4*, *25*, *34*, *38, 39*, *56*
mrmRunExample, 22

mrmRunExample(), *29*, *50*, *52*
mrmRunPCA, 23
mrmRunPCA(), *4*, *22*, *34*, *38, 39*, *56*
mrmValidateConfig, 25
mrmValidateConfig(), *6*, *8*, *17, 18*, *26*, *28*, *47–49*
mrmValidateDesign, 26
mrmValidateDesign(), *6*, *8*, *17, 18*, *25*, *28*, *47–49*
mrmValidateInput, 27
mrmValidateInput(), *6*, *8*, *17, 18*, *25, 26*, *47–49*

newConfig, 28
newConfig(), *23*, *50*, *52*
normaliseMatrix, 29
normaliseMatrix(), *5*, *10, 11*, *13, 14*, *16*, *45*, *55*

plotBoxplot, 30
plotBoxplot(), *20*, *33*, *37*, *41*, *54*, *57*
plotCorrelation, 32
plotCorrelation(), *20*, *31*, *37*, *41*, *54*, *57*
plotDrift, 33
plotDrift(), *4*, *22*, *25*, *38, 39*, *56*
plotGroupHeatmap, 35
plotGroupHeatmap(), *20*, *31*, *33*, *41*, *54*, *57*
plotLoadings, 37
plotLoadings(), *4*, *22*, *25*, *34*, *39*, *56*
plotQQ, 38
plotQQ(), *4*, *22*, *25*, *34*, *38*, *56*
plotVolcano, 40
plotVolcano(), *20*, *31*, *33*, *37*, *54*, *57*
print.mrm_validation, 41
processDataset, 42
processDataset(), *5*, *10, 11*, *13, 14*, *16*, *30*, *55*

readPeakMatrix, 46
readPeakMatrix(), *6*, *8*, *17, 18*, *25, 26*, *28*, *48, 49*
readSkyline, 47
readSkyline(), *6*, *8*, *17, 18*, *25, 26*, *28*, *47*, *49*
readTargetLynx, 48
readTargetLynx(), *6*, *8*, *17, 18*, *25, 26*, *28*, *47, 48*
runMRManalyzeR, 49
runMRManalyzeR(), *23*, *29*, *52*
runMRManalyzeRCombine, 51
runMRManalyzeRCombine(), *23*, *29*, *50*
runStats, 53
runStats(), *20*, *31*, *33*, *37*, *41*, *57*

subsetDataset, 54
subsetDataset(), *5*, *10, 11*, *13, 14*, *16*, *30*, *45*
summariseFeatures, 55
summariseFeatures(), *4*, *22*, *25*, *34*, *38, 39*
summariseGroups, 56
summariseGroups(), *20*, *31*, *33*, *37*, *41*, *54*