Package 'MetaboAnnotation'

January 20, 2026

Title Utilities for Annotation of Metabolomics Data

Version 1.14.0

Description High level functions to assist in annotation of (metabolomics) data sets. These include functions to perform simple tentative annotations based on mass matching but also functions to consider m/z and retention times for annotation of LC-MS features given that respective reference values are available. In addition, the function provides high-level functions to simplify matching of LC-MS/MS spectra against spectral libraries and objects and functionality to represent and manage such matched data.

Depends R (>= 4.0.0)

Imports BiocGenerics, MsCoreUtils, MetaboCoreUtils, ProtGenerics, methods, S4Vectors, Spectra (>= 1.17.6), BiocParallel, SummarizedExperiment, QFeatures, AnnotationHub, graphics, CompoundDb

Suggests testthat, knitr, msdata, BiocStyle, rmarkdown, plotly, shiny, shinyjs, msentropy, DT, microbenchmark, mzR

Enhances RMariaDB, RSQLite

License Artistic-2.0

VignetteBuilder knitr

BugReports https://github.com/RforMassSpectrometry/MetaboAnnotation/issues

URL https://github.com/RforMassSpectrometry/MetaboAnnotation

biocViews Infrastructure, Metabolomics, MassSpectrometry

Roxygen list(markdown=TRUE)

RoxygenNote 7.3.3

Encoding UTF-8

Collate 'AllClassUnions.R' 'AllGenerics.R' 'CompAnnotationSource.R' 'CompDbSource.R' 'Matched.R' 'MatchedSpectra.R' 'group_standards.R' 'hidden-aliases.R' 'matchFormula.R' 'matchSpectra.R' 'matchValues.R' 'validateMatchedSpectra.R'

git_url https://git.bioconductor.org/packages/MetaboAnnotation

git_branch RELEASE_3_22

git_last_commit 790d88f

git_last_commit_date 2025-10-29

Repository Bioconductor 3.22

Date/Publication 2026-01-19

Author Michael Witting [aut] (ORCID: <https://orcid.org/0000-0002-1462-4426>),
Johannes Rainer [aut, cre] (ORCID:
<https://orcid.org/0000-0002-6977-7147>),
Andrea Vicini [aut] (ORCID: <https://orcid.org/0000-0001-9438-6909>),
Carolin Huber [aut] (ORCID: <https://orcid.org/0000-0002-9355-8948>),
Philippine Louail [aut] (ORCID:
<https://orcid.org/0009-0007-5429-6846>),
Nir Shachaf [ctb]

Maintainer Johannes Rainer <Johannes.Rainer@eurac.edu>

Contents

`addMatches` *Representation of generic objects matches*

Description

Matches between *query* and *target* generic objects can be represented by the `Matched` object. By default, all data accessors work as *left joins* between the *query* and the *target* object, i.e. values are returned for each *query* object with eventual duplicated entries (values) if the *query* object matches more than one *target* object. See also *Creation and subsetting* as well as *Extracting data* sections below for details and more information.

The `Matched` object allows to represent matches between one-dimensional `query` and `target` objects (being e.g. `numeric` or `list`), two-dimensional objects (`data.frame` or `matrix`) or more complex structures such as `SummarizedExperiments` or `QFeatures`. Combinations of all these different data types are also supported. Matches are represented between elements of one-dimensional objects, or rows for two-dimensional objects (including `SummarizedExperiment` or `QFeatures`). For `QFeatures::QFeatures()` objects matches to only one of the *assays* within the object is supported.

Usage

```
addMatches(object, ...)

endoapply(X, FUN, ...)

filterMatches(object, param, ...)

matchedData(object, ...)

queryVariables(object, ...)

targetVariables(object, ...)

Matched(
  query = list(),
  target = list(),
 matches = data.frame(query_idx = integer(), target_idx = integer(), score = numeric()),
  queryAssay = character(),
  targetAssay = character(),
  metadata = list()
)

## S4 method for signature 'Matched'
length(x)

## S4 method for signature 'Matched'
show(object)

## S4 method for signature 'Matched,ANY,ANY,ANY'
x[i, j, ..., drop = FALSE]

matches(object)

target(object)

## S4 method for signature 'Matched'
query(x, pattern, ...)

targetIndex(object)

queryIndex(object)

whichTarget(object)

whichQuery(object)

## S4 method for signature 'Matched'
x$name

## S4 method for signature 'Matched'
colnames(x)
```

```
scoreVariables(object)

## S4 method for signature 'Matched'
queryVariables(object)

## S4 method for signature 'Matched'
targetVariables(object)

## S4 method for signature 'Matched'
matchedData(object, columns = colnames(object), ...)

pruneTarget(object)

## S4 method for signature 'Matched,missing'
filterMatches(
  object,
  queryValue = integer(),
  targetValue = integer(),
  queryColname = character(),
  targetColname = character(),
  index = integer(),
  keep = TRUE,
  ...
)

SelectMatchesParam(
  queryValue = numeric(),
  targetValue = numeric(),
  queryColname = character(),
  targetColname = character(),
  index = integer(),
  keep = TRUE
)

TopRankedMatchesParam(n = 1L, decreasing = FALSE)

ScoreThresholdParam(threshold = 0, above = FALSE, column = "score")

## S4 method for signature 'Matched,SelectMatchesParam'
filterMatches(object, param, ...)

## S4 method for signature 'Matched,TopRankedMatchesParam'
filterMatches(object, param, ...)

## S4 method for signature 'Matched,ScoreThresholdParam'
filterMatches(object, param, ...)

SingleMatchParam(
  duplicates = c("remove", "closest", "top_ranked"),
  column = "score",
  decreasing = TRUE
)
```

```
## S4 method for signature 'Matched,SingleMatchParam'
filterMatches(object, param, ...)

## S4 method for signature 'Matched'
addMatches(
  object,
  queryValue = integer(),
  targetValue = integer(),
  queryColname = character(),
  targetColname = character(),
  score = rep(NA_real_, length(queryValue)),
  isIndex = FALSE
)

## S4 method for signature 'ANY'
endoapply(X, FUN, ...)

## S4 method for signature 'Matched'
endoapply(X, FUN, ...)

## S4 method for signature 'Matched'
lapply(X, FUN, ...)
```

Arguments

object	a Matched object.
...	additional parameters.
X	Matched object.
FUN	for lapply and endoapply: user defined function that takes a Matched object as a first parameter and possibly additional parameters (that need to be provided in the lapply or endoapply call. For lapply FUN can return any object while for endoapply it must return a Matched object.
param	for filterMatches: parameter object to select and customize the filtering procedure.
query	object with the query elements.
target	object with the elements against which query has been matched.
matches	data.frame with columns "query_idx" (integer), "target_idx" (integer) and "score" (numeric) representing the n:m mapping of elements between the query and the target objects.
queryAssay	character that needs to be specified when query is a QFeatures. In this case, queryAssay is expected to be the name of one of the assays in query (the one on which the matching was performed).
targetAssay	character that needs to be specified when target is a QFeatures. In this case, targetAssay is expected to be the name of one of the assays in target (the one on which the matching was performed).
metadata	list with optional additional metadata.
x	Matched object.
i	integer or logical defining the query elements to keep.

j	for [: ignored.
drop	for [: ignored.
pattern	for query: ignored.
name	for $: the name of the column (or variable) to extract.
columns	for matchedData: character vector with column names of variables that should be extracted.
queryValue	for SelectMatchesParam: vector of values to search for in query (if query is 1-dimensional) or in column queryColname of query (if query is 2-dimensional). For addMatches: either an index in query or value in column queryColname of query defining (together with targetValue) the pair of query and target elements for which a match should be manually added. Lengths of queryValue and targetValue have to match.
targetValue	for SelectMatchesParam: vector of values to search for in target (if target is 1-dimensional) or in column targetColname of target (if target is 2-dimensional). For addMatches: either an index in target or value in column targetColname of target defining (together with queryValue) the pair of query and target elements for which a match should be manually added. Lengths of queryValue and targetValue have to match.
queryColname	for SelectMatchesParam: if query is 2-dimensional it represents the column of query against which elements of queryValue are compared.
targetColname	for SelectMatchesParam: if query is 2-dimensional it represents the column of target against which elements of targetValue are compared.
index	for SelectMatchesParam: indices of the matches to keep (if keep = TRUE) or to drop if (keep = FALSE).
keep	for SelectMatchesParam: logical. If keep = TRUE the matches are kept, if keep = FALSE they are removed.
n	for TopRankedMatchesParam: integer(1) with number of best ranked matches to keep for each query element.
decreasing	for TopRankedMatchesParam: logical(1) whether scores should be ordered increasing or decreasing. Defaults to decreasing = FALSE.
threshold	for ScoreThresholdParam : numeric(1) specifying the threshold to consider for the filtering.
above	for ScoreThresholdParam : logical(1) specifying whether to keep matches above (above = TRUE) or below (above = FALSE) a certain threshold.
column	for ScoreThresholdParam: character(1) specifying the name of the score variable to consider for the filtering (the default is column = "score"). For SingleMatchParam: character(1) defining the name of the column to be used for de-duplication. See description of SingleMatchParam in the *Filtering and subsetting* section for details.
duplicates	for SingleMatchParam: character(1) defining the *de-duplication* strategy. See the description of SingleMatchParam in the *Filtering and subsetting* subsection for choices and details.
score	for addMatches: numeric (same length than queryValue) or data.frame (same number of rows than queryValue) specifying the scores for the matches to add. If not specified, a NA will be used as score.
isIndex	for addMatches: specifies if queryValue and targetValue are expected to be vectors of indices.

Value

See individual method description above for details.

Creation and general handling

`Matched` object is returned as result from the `matchValues()` function.

Alternatively, `Matched` objects can also be created with the `Matched` function providing the `query` and `target` objects as well as the `matches` `data.frame` with two columns of integer indices defining which elements from *query* match which element from *target*.

- `addMatches`: add new matches to an existing object. Parameters `queryValue` and `targetValue` allow to define which element(s) in `query` and `target` should be considered matching. If `isIndex = TRUE`, both `queryValue` and `targetValue` are considered to be integer indices identifying the matching elements in `query` and `target`, respectively. Alternatively (with `isIndex = FALSE`) `queryValue` and `targetValue` can be elements in columns `queryColname` or `targetColname` which can be used to identify the matching elements. Note that in this case **only the first** matching pair is added. Parameter `score` allows to provide the score for the match. It can be a numeric with the score or a `data.frame` with additional information on the manually added matches. In both cases its length (or number of rows) has to match the length of `queryValue`. See examples below for more information.
- `endoapply`: applies a user defined function `FUN` to each subset of matches in a `Matched` object corresponding to a `query` element (i.e. for each `x[i]` with `i` being 1 to `length(x)`). The results are then combined in a single `Matched` object representing updated matches. Note that `FUN` has to return a `Matched` object.
- `lapply`: applies a user defined function `FUN` to each subset of matches in a `Matched` object for each `query` element (i.e. to each `x[i]` with `i` from 1 to `length(x)`). It returns a `list` of `length(object)` elements where each element is the output of `FUN` applied to each subset of matches.

Filtering and subsetting

- `[`: subset the object selecting `query` object elements to keep with parameter `i`. The resulting object will contain all the matches for the selected query elements. The `target` object will by default be returned as-is.
- `filterMatches`: filter matches in a `Matched` object using different approaches depending on the class of `param`:
  - `ScoreThresholdParam`: keeps only the matches whose score is strictly above or strictly below a certain threshold (respectively when parameter `above = TRUE` and `above = FALSE`). The name of the column containing the scores to be used for the filtering can be specified with parameter `column`. The default for `column` is `"score"`. Such variable is present in each `Matched` object. The name of other score variables (if present) can be provided (the names of all score variables can be obtained with `scoreVariables()` function). For example `column = "score_rt"` can be used to filter matches based on retention time scores for `Matched` objects returned by `matchValues()` when `param` objects involving a retention time comparison are used.
  - `SelectMatchesParam`: keeps or removes (respectively when parameter `keep = TRUE` and `keep = FALSE`) matches corresponding to certain indices or values of `query` and `target`. If `queryValue` and `targetValue` are provided, matches for these value pairs are kept or removed. Parameter index`allows to filter matches providing their index in the [matches()] matr` in the object. See examples below for more information.

  - SingleMatchParam: reduces matches to keep only (at most) a single match per query. The deduplication strategy can be defined with parameter duplicates:
    * duplicates = "remove": all matches for query elements matching more than one target element will be removed.
    * duplicates = "closest": keep only the *closest* match for each query element. The closest match is defined by the value(s) of *score* (and eventually *score_rt*, if present). The one match with the smallest value for this (these) column(s) is retained. This is equivalent to TopRankedMatchesParam(n = 1L, decreasing = FALSE).
    * duplicates = "top_ranked": select the *best ranking* match for each query element. Parameter column allows to specify the column by which matches are ranked (use targetVariables(object) or scoreVariables(object) to list possible columns). Parameter decreasing allows to define whether the match with the highest (decreasing = TRUE) or lowest (decreasing = FALSE) value in column for each *query* will be selected.
  - TopRankedMatchesParam: for each query element the matches are ranked according to their score and only the n best of them are kept (if n is larger than the number of matches for a given query element all the matches are returned). For the ranking (ordering) R's rank function is used on the absolute values of the scores (variable "score"), thus, smaller score values (representing e.g. smaller differences between expected and observed m/z values) are considered *better*. By setting parameter decreasing = TRUE matches can be ranked in decreasing order (i.e. higher scores are ranked higher and are thus selected). If besides variable "score" also variable "score_rt" is available in the Matched object (which is the case for the Matched object returned by matchValues() for param objects involving a retention time comparison), the ordering of the matches is based on the product of the ranks of the two variables (ranking of retention time differences is performed on the absolute value of "score_rt"). Thus, matches with small (or, depending on parameter decreasing, large) values for "score" **and** "score_rt" are returned.

- pruneTarget: *cleans* the object by removing non-matched **target** elements.

Extracting data

- $ extracts a single variable from the Matched x. The variables that can be extracted can be listed using colnames(x). These variables can belong to *query*, *target* or be related to the matches (e.g. the score of each match). If the *query* (*target*) object is two dimensional, its columns can be extracted (prefix "target_" is used for columns in the *target* object) otherwise if *query* (*target*) has only a single dimension (e.g. is a list or a character) the whole object can be extracted with x$query (x$target). More precisely, when *query* (*target*) is a SummarizedExperiment the columns from rowData(query) (rowData(target)) are extracted; when *query* (*target*) is a QFeatures::QFeatures() the columns from rowData of the assay specified in the queryAssay (targetAssay) slot are extracted. The matching scores are available as *variable* "score". Similar to a left join between the query and target elements, this function returns a value for each query element, with eventual duplicated values for query elements matching more than one target element. If variables from the target data.frame are extracted, an NA is reported for the entries corresponding to *query* elements that don't match any target element. See examples below for more details.
- length returns the number of **query** elements.
- matchedData allows to extract multiple variables contained in the Matched object as a DataFrame. Parameter columns allows to define which columns (or variables) should be returned (defaults to columns = colnames(object)). Each single column in the returned DataFrame is constructed in the same way as in $. That is, like $, this function performs a *left join* of variables

from the *query* and *target* objects returning all values for all *query* elements (eventually returning duplicated elements for query elements matching multiple target elements) and the values for the target elements matched to the respective query elements (or NA if the target element is not matched to any query element).

- matches returns a data.frame with the actual matching information with columns "query_idx" (index of the element in query), "target_idx" (index of the element in target) "score" (the score of the match) and eventual additional columns.
- target returns the *target* object.
- targetIndex returns the indices of the matched targets in the order they are assigned to the query elements. The length of the returned integer vector is equal to the total number of matches in the object. targetIndex and queryIndex are aligned, i.e. each element in them represent a matched query-target pair.
- query returns the *query* object.
- queryIndex returns the indices of the query elements with matches to target elements. The length of the returned integer vector is equal to the total number of matches in the object. targetIndex and queryIndex are aligned, i.e. each element in them represent a matched query-target pair.
- queryVariables returns the names of the variables (columns) in *query*.
- scoreVariables returns the names of the score variables stored in the Matched object (precisely the names of the variables in matches(object) containing the string "score" in their name ignoring the case).
- targetVariables returns the names of the variables (columns) in *target* (prefixed with "target_").
- whichTarget returns an integer with the indices of the elements in *target* that match at least one element in *query*.
- whichQuery returns an integer with the indices of the elements in *query* that match at least one element in *target*.

Author(s)

Andrea Vicini, Johannes Rainer

See Also

MatchedSpectra() for matched Spectra::Spectra() objects.

Examples

```
## Creating a `Matched` object.
q1 <- data.frame(col1 = 1:5, col2 = 6:10)
t1 <- data.frame(col1 = 11:16, col2 = 17:22)
## Define matches between query row 1 with target row 2 and, query row 2
## with target rows 2,3,4 and query row 5 with target row 5.
mo <- Matched(
    q1, t1, matches = data.frame(query_idx = c(1L, 2L, 2L, 2L, 5L),
                                 target_idx = c(2L, 2L, 3L, 4L, 5L),
                                 score = seq(0.5, 0.9, by = 0.1)))
mo

## Which of the query elements (rows) match at least one target
## element (row)?
whichQuery(mo)
```

```
## Which target elements (rows) match at least one query element (row)?
whichTarget(mo)

## Extracting variable "col1" from query object .
mo$col1

## We have duplicated values for the entries of `col1` related to query
## elements (rows) matched to multiple rows of the target object). The
## value of `col1` is returned for each element (row) in the query.

## Extracting variable "col1" from target object. To access columns from
## target we have to prefix the name of the column by `"target_"`.
## Note that only values of `col1` for rows matching at least one query
## row are returned and an NA is reported for query rows without matching
## target rows.
mo$target_col1

## The 3rd and 4th query rows do not match any target row, thus `NA` is
## returned.

## `matchedData` can be used to extract all (or selected) columns
## from the object. Same as with `$`, a left join between the columns
## from the query and the target is performed. Below we extract selected
## columns from the object as a DataFrame.
res <- matchedData(mo, columns = c("col1", "col2", "target_col1",
                                   "target_col2"))
res
res$col1
res$target_col1

## With the `queryIndex` and `targetIndex` it is possible to extract the
## indices of the matched query-target pairs:
queryIndex(mo)
targetIndex(mo)

## Hence, the first match is between the query with index 1 to the target
## with index 2, then, query with index 2 is matched to target with index 2
## and so on.

## The example matched object contains all query and all target
## elements (rows). Below we subset the object keeping only query rows that
## are matched to at least one target row.
mo_sub <- mo[whichQuery(mo)]

## mo_sub contains now only 3 query rows:
nrow(query(mo_sub))

## while the original object contains all 5 query rows:
nrow(query(mo))

## Both objects contain however still the full target object:
nrow(target(mo))
nrow(target(mo_sub))

## With the `pruneTarget` we can however reduce also the target rows to
## only those that match at least one query row
```

```
mo_sub <- pruneTarget(mo_sub)
nrow(target(mo_sub))

########
## Creating a `Matched` object with a `data.frame` for `query` and a `vector`
## for `target`. The matches are specified in the same way as the example
## before.

q1 <- data.frame(col1 = 1:5, col2 = 6:10)
t2 <- 11:16
mo <- Matched(q1, t2, matches = data.frame(query_idx = c(1L, 2L, 2L, 2L, 5L),
    target_idx = c(2L, 2L, 3L, 4L, 5L), score = seq(0.5, 0.9, by = 0.1)))

## *target* is a simple vector and has thus no columns. The matched values
## from target, if it does not have dimensions and hence column names, can
## be retrieved with `$target`
mo$target

## Note that in this case "target" is returned by the function `colnames`
colnames(mo)

## As before, we can extract all data as a `DataFrame`
res <- matchedData(mo)
res

## Note that the columns of the obtained `DataFrame` are the same as the
## corresponding vectors obtained with `$`
res$col1
res$target

## Also subsetting and pruning works in the same way as the example above.

mo_sub <- mo[whichQuery(mo)]

## mo_sub contains now only 3 query rows:
nrow(query(mo_sub))

## while the original object contains all 5 query rows:
nrow(query(mo))

## Both object contain however still the full target object:
length(target(mo))
length(target(mo_sub))

## Reducing the target elements to only those that match at least one query
## row
mo_sub <- pruneTarget(mo_sub)
length(target(mo_sub))

########
## Filtering `Matched` with `filterMatches`

## Inspecting the matches in `mo`:
mo$col1
mo$target

## We have thus target *12* matched to both query elements with values 1 and
```

```
## 2, and query element 2 is matching 3 target elements. Let's assume we want
## to resolve this multiple mappings to keep from them only the match between
## query 1 (column `"col1"` containing value `1`) with target 1 (value `12`)
## and query 2 (column `"col1"` containing value `2`) with target 2 (value
## `13`). In addition we also want to keep query element 5 (value `5` in
## column `"col1"`) with the target with value `15`:
mo_sub <- filterMatches(mo,
    SelectMatchesParam(queryValue = c(1, 2, 5), queryColname = "col1",
                       targetValue = c(12, 13, 15)))
matchedData(mo_sub)

## Alternatively to specifying the matches to filter with `queryValue` and
## `targetValue` it is also possible to specify directly the index of the
## match(es) in the `matches` `data.frame`:
matches(mo)

## To keep only matches like in the example above we could use:
mo_sub <- filterMatches(mo, SelectMatchesParam(index = c(1, 3, 5)))
matchedData(mo_sub)

## Note also that, instead of keeping the specified matches, it would be
## possible to remove them by setting `keep = FALSE`. Below we remove
## selected matches from the object:
mo_sub <- filterMatches(mo,
    SelectMatchesParam(queryValue = c(2, 2), queryColname = "col1",
                       targetValue = c(12, 14), keep = FALSE))
mo_sub$col1
mo_sub$target

## As alternative to *manually* selecting matches it is also possible to
## filter matches keeping only the *best matches* using the
## `TopRankedMatchesParam`. This will rank matches for each query based on
## their *score* value and select the best *n* matches with lowest score
## values (i.e. smallest difference in m/z values).
mo_sub <- filterMatches(mo, TopRankedMatchesParam(n = 1L))
matchedData(mo_sub)

## Additionally it is possible to select matches based on a threshold
## for their *score*. Below we keep matches with score below 0.75 (one
## could select matches with *score* greater than the threshold by setting
## `ScoreThresholdParam` parameter `above = TRUE`.
mo_sub <- filterMatches(mo, ScoreThresholdParam(threshold = 0.75))
matchedData(mo_sub)

########
## Selecting the best match for each `query` element with `endoapply`

## It is also possible to select for each `query` element the match with the
## lowest score using `endoapply`. We manually define a function to select
## the best match for each query and give it as input to `endoapply`
## together with the `Matched` object itself. We obtain the same results as
## in the `filterMatches` example above.

FUN <- function(x) {
    if(nrow(x@matches) > 1)
        x@matches <- x@matches[order(x@matches$score)[1], , drop = FALSE]
    x
```

```
}

mo_sub <- endoapply(mo, FUN)
matchedData(mo_sub)

########
## Adding matches using `addMatches`

## `addMatches` allows to manually add matches. Below we add a new match
## between the `query` element with a value of `1` in column `"col1"` and
## the target element with a value of `15`. Parameter `score` allows to
## assign a score value to the match.
mo_add <- addMatches(mo, queryValue = 1, queryColname = "col1",
    targetValue = 15, score = 1.40)
matchedData(mo_add)
## Matches are always sorted by `query`, thus, the new match is listed as
## second match.

## Alternatively, we can also provide a `data.frame` with parameter `score`
## which enables us to add additional information to the added match. Below
## we define the score and an additional column specifying that this match
## was added manually. This information will then also be available in the
## `matchedData`.
mo_add <- addMatches(mo, queryValue = 1, queryColname = "col1",
    targetValue = 15, score = data.frame(score = 5, manual = TRUE))
matchedData(mo_add)

## The match will get a score of NA if we're not providing any score.
mo_add <- addMatches(mo, queryValue = 1, queryColname = "col1",
    targetValue = 15)
matchedData(mo_add)

## Creating a `Matched` object with a `SummarizedExperiment` for `query` and
## a `vector` for `target`. The matches are specified in the same way as
## the example before.
library(SummarizedExperiment)
q1 <- SummarizedExperiment(
  assays = data.frame(matrix(NA, 5, 2)),
  rowData = data.frame(col1 = 1:5, col2 = 6:10),
  colData = data.frame(cD1 = c(NA, NA), cD2 = c(NA, NA)))
t1 <- data.frame(col1 = 11:16, col2 = 17:22)
## Define matches between row 1 in rowData(q1) with target row 2 and,
## rowData(q1) row 2 with target rows 2,3,4 and rowData(q1) row 5 with target
## row 5.
mo <- Matched(
    q1, t1, matches = data.frame(query_idx = c(1L, 2L, 2L, 2L, 5L),
                                 target_idx = c(2L, 2L, 3L, 4L, 5L),
                                  score = seq(0.5, 0.9, by = 0.1)))
mo

## Which of the query elements (rows) match at least one target
## element (row)?
whichQuery(mo)

## Which target elements (rows) match at least one query element (row)?
whichTarget(mo)
```

```
## Extracting variable "col1" from rowData(q1).
mo$col1

## We have duplicated values for the entries of `col1` related to rows of
## rowData(q1) matched to multiple rows of the target data.frame t1. The
## value of `col1` is returned for each row in the rowData of query.

## Extracting variable "col1" from target object. To access columns from
## target we have to prefix the name of the column by `"target_"`.
## Note that only values of `col1` for rows matching at least one row in
## rowData of query are returned and an NA is reported for those without
## matching target rows.
mo$target_col1

## The 3rd and 4th query rows do not match any target row, thus `NA` is
## returned.

## `matchedData` can be used to extract all (or selected) columns
## from the object. Same as with `$`, a left join between the columns
## from the query and the target is performed. Below we extract selected
## columns from the object as a DataFrame.
res <- matchedData(mo, columns = c("col1", "col2", "target_col1",
                                   "target_col2"))
res
res$col1
res$target_col1

## The example `Matched` object contains all rows in the
## `rowData` of the `SummarizedExperiment` and all target rows. Below we
## subset the object keeping only rows that are matched to at least one
## target row.
mo_sub <- mo[whichQuery(mo)]

## mo_sub contains now a `SummarizedExperiment` with only 3 rows:
nrow(query(mo_sub))

## while the original object contains a `SummarizedExperiment` with all 5
## rows:
nrow(query(mo))

## Both objects contain however still the full target object:
nrow(target(mo))
nrow(target(mo_sub))

## With the `pruneTarget` we can however reduce also the target rows to
## only those that match at least one in the `rowData` of query
mo_sub <- pruneTarget(mo_sub)
nrow(target(mo_sub))
```

CompAnnotationSource *Compound Annotation Sources*

Description

CompAnnotationSources (i.e. classes extending the base virtual CompAnnotationSource class) define and provide access to a (potentially remote) compound annotation resource. This aims to

simplify the integration of external annotation resources by automating the actual connection (or data resource download) process from the user. In addition, since the reference resource is not directly exposed to the user it allows integration of annotation resources that do not allow access to the full data.

Objects extending `CompAnnotationSource` available in this package are:

- `CompDbSource()`: annotation source referencing an annotation source in the `[CompoundDb::CompDb()]` format (from the `CompoundDb` Bioconductor package).

Classes extending `CompAnnotationSource` need to implement the `matchSpectra` method with parameters `query`, `target` and `param` where `query` is the `Spectra` object with the (experimental) query spectra, `target` the object extending the `CompAnnotationSource` and `param` the parameter object defining the similarity calculation (e.g. `CompareSpectraParam()`. The method is expected to return a MatchedSpectra object.

`CompAnnotationSource` objects are not expected to contain any annotation data. Access to the annotation data (in form of a `Spectra` object) is suggested to be only established within the object's `matchSpectra` method. This would also enable parallel processing of annotations as no e.g. database connection would have to be shared across processes.

Usage

```
## S4 method for signature 'Spectra,CompAnnotationSource,Param'
matchSpectra(query, target, param, ...)

## S4 method for signature 'CompAnnotationSource'
show(object)

## S4 method for signature 'CompAnnotationSource'
metadata(x, ...)
```

Arguments

`query`	for `matchSpectra`: Spectra::Spectra object with the query spectra.
`target`	for `matchSpectra`: object extending CompAnnotationSource (such as CompDbSource) with the target (reference) spectra to compare `query` against.
`param`	for `matchSpectra`: parameter object (such as CompareSpectraParam) defining the settings for the matching.
`...`	additional parameters passed to `matchSpectra`.
`object`	A `CompAnnotationSource` object.
`x`	A `CompAnnotationSource` object.

Methods that need to be implemented

For an example implementation see `CompDbSource()`.

- `matchSpectra`: function to match experimental MS2 spectra against the annotation source. See `matchSpectra()` for parameters.
- `metadata`: function to provide metadata on the annotation resource (host, source, version etc).
- `show` (optional): method to provide general information on the data source.

Author(s)

Johannes Rainer, Nir Shachaf

CompDbSource *Compound Annotation Sources for* CompDb *databases*

Description

CompDbSource objects represent references to CompoundDb::CompDb database-backed annotation resources. Instances are expected to be created with the dedicated construction functions such as MassBankSource or the generic CompDbSource. The annotation data is not stored within the object but will be accessed/loaded within the object's matchSpectra method.

New CompDbSource objects can be created using the functions:

- CompDbSource: create a new CompDbSource object from an existing CompDb database. The (SQLite) database file (including the full path) needs to be provided with parameter dbfile.
- MassBankSource: retrieves a CompDb database for the specified MassBank release from Bioconductor's online AnnotationHub (if it exists) and uses that. Note that AnnotationHub resources are cached locally and thus only downloaded the first time. The function has parameters release which allows to define the desired MassBank release (e.g. release = "2021.03" or release = "2022.06") and ... which allows to pass optional parameters to the AnnotationHub constructor function, such as localHub = TRUE to use only the cached data and avoid updating/retrieving updates from the internet.

Other functions:

- metadata: get metadata (information) on the annotation resource.

Usage

```
CompDbSource(dbfile = character())

## S4 method for signature 'CompDbSource'
metadata(x, ...)

## S4 method for signature 'CompDbSource'
show(object)

MassBankSource(release = "2021.03", ...)
```

Arguments

dbfile	character(1) with the database file (including the full path).
x	A CompDbSource object.
...	For CompDbSource: ignored. For MassBankSource: optional parameters passed to the AnnotationHub constructor function.
object	A CompDbSource object.
release	A character(1) defining the version/release of MassBank that should be used.

Author(s)

Johannes Rainer

Examples

```
## Locate a CompDb SQLite database file. For this example we use the test
## database from the `CompoundDb` package.
fl <- system.file("sql", "CompDb.MassBank.sql", package = "CompoundDb")
ann_src <- CompDbSource(fl)

## The object contains only the reference/link to the annotation resource.
ann_src

## Retrieve a CompDb with MassBank data for a certain MassBank release
mb_src <- MassBankSource("2021.03")
mb_src
```

---

createStandardMixes *Create Standard Mixes from a Matrix of Standard Compounds*

Description

The `createStandardMixes` function defines groups (mixes) of compounds (standards) with dissimilar m/z values. The expected size of the groups can be defined with parameters `max_nstd` and `min_nstd` and the minimum required difference between m/z values within each group with parameter `min_diff`. The group assignment will be reported in an additional column in the result data frame.

Usage

```
createStandardMixes(
  x,
  max_nstd = 10,
  min_nstd = 5,
  min_diff = 2,
  iterativeRandomization = FALSE
)
```

Arguments

`x`	`numeric` matrix with row names representing the compounds and columns representing different adducts. Such a matrix with m/z values for different adducts for compounds could e.g. be created with the `MetaboCoreUtils::mass2mz()` function.
`max_nstd`	`numeric` number of maximum standards per group.
`min_nstd`	`numeric` number of minimum standards per group. Only needed when using `iterativeRandomization = TRUE`.
`min_diff`	`numeric` Minimum difference for considering two values as distinct.
`iterativeRandomization`	`logical` default `FALSE`. If set to `TRUE`, `createStandardMixes` will randomly rearrange the rows of `x` until the user inputs are satisfied.

Details

Users should be aware that because the function iterates through x, the compounds at the bottom of the matrix are more complicated to group, and there is a possibility that some compounds will not be grouped with others. We advise specifyiong `iterativeRandomization = TRUE` even if it takes more time.

Value

`data.frame` created by adding a column `group` to the input `x` matrix, comprising the group number for each compound.

Author(s)

Philippine Louail

Examples

```
## Iterative grouping only
x <- matrix(c(135.0288, 157.0107, 184.0604, 206.0424, 265.1118, 287.0937,
              169.0356, 191.0176, 468.9809, 490.9628, 178.0532, 200.0352),
            ncol = 2, byrow = TRUE,
            dimnames = list(c("Malic Acid", "Pyridoxic Acid", "Thiamine",
                              "Uric acid", "dUTP", "N-Formyl-L-methionine"),
                            c("adduct_1", "adduct_2")))
result <- createStandardMixes(x, max_nstd = 3, min_diff = 2)

## Randomize grouping
set.seed(123)
x <- matrix(c(349.0544, 371.0363, 325.0431, 347.0251, 581.0416, 603.0235,
              167.0564, 189.0383, 150.0583, 172.0403, 171.0053, 192.9872,
              130.0863, 152.0682, 768.1225, 790.1044),
            ncol = 2, byrow = TRUE,
            dimnames = list(c("IMP", "UMP", "UDP-glucuronate",
                              "1-Methylxanthine", "Methionine",
                              "Dihydroxyacetone phosphate",
                              "Pipecolic acid", "CoA"),
                            c("[M+H]+", "[M+Na]+")))
result <- createStandardMixes(x, max_nstd = 4, min_nstd = 3, min_diff = 2,
                              iterativeRandomization = TRUE)
```

---

hidden_aliases *Internal page for hidden aliases*

Description

For S4 methods that require a documentation entry but only clutter the index.

Value

Not applicable

`MatchedSpectra` *Representation of Spectra matches*

Description

Matches between query and target spectra can be represented by the `MatchedSpectra` object. Functions like the `matchSpectra()` function will return this type of object. By default, all data accessors work as *left joins* between the *query* and the *target* spectra, i.e. values are returned for each *query* spectrum with eventual duplicated entries (values) if the query spectrum matches more than one target spectrum.

Usage

```
MatchedSpectra(
  query = Spectra(),
  target = Spectra(),
 matches = data.frame(query_idx = integer(), target_idx = integer(), score = numeric())
)

## S4 method for signature 'MatchedSpectra'
spectraVariables(object)

## S4 method for signature 'MatchedSpectra'
queryVariables(object)

## S4 method for signature 'MatchedSpectra'
targetVariables(object)

## S4 method for signature 'MatchedSpectra'
colnames(x)

## S4 method for signature 'MatchedSpectra'
x$name

## S4 method for signature 'MatchedSpectra'
spectraData(object, columns = spectraVariables(object))

## S4 method for signature 'MatchedSpectra'
matchedData(object, columns = spectraVariables(object), ...)

## S4 method for signature 'MatchedSpectra'
addProcessing(object, FUN, ..., spectraVariables = character())

## S4 method for signature 'MatchedSpectra'
plotSpectraMirror(
  x,
  xlab = "m/z",
  ylab = "intensity",
  main = "",
  scalePeaks = FALSE,
  ...
```

```
)

## S4 method for signature 'MatchedSpectra,MsBackend'
setBackend(object, backend, ...)
```

Arguments

query	Spectra with the query spectra.
target	Spectra with the spectra against which query has been matched.
matches	data.frame with columns "query_idx" (integer), "target_idx" (integer) and "score" (numeric) representing the *n:m* mapping of elements between the query and the target Spectra.
object	MatchedSpectra object.
x	MatchedSpectra object.
name	for $: the name of the spectra variable to extract.
columns	for spectraData: character vector with spectra variable names that should be extracted.
...	for addProcessing: additional parameters for the function FUN. For plotSpectraMirror: additional parameters passed to the plotting functions.
FUN	for addProcessing: function to be applied to the peak matrix of each spectrum in object. See Spectra::Spectra() for more details.
spectraVariables	for addProcessing: character with additional spectra variables that should be passed along to the function defined with FUN. See Spectra::Spectra() for details.
xlab	for plotSpectraMirror: the label for the x-axis.
ylab	for plotSpectraMirror: the label for the y-axis.
main	for plotSpectraMirror: an optional title for each plot.
scalePeaks	for plotSpectraMirror: logical(1) if peak intensities (per spectrum) should be scaled to a total sum of one (per spectrum) prior to plotting.
backend	for setBackend: instance of an object extending Spectra::MsBackend. See help for Spectra::setBackend() for more details.

Value

See individual method desciption above for details.

Creation, subset and filtering

MatchedSpectra objects are the result object from the matchSpectra(). While generally not needed, MatchedSpectra objects can also be created with the MatchedSpectra function providing the query and target Spectra objects as well as a data.frame with the *matches* between query and target elements. This data frame is expected to have columns "query_idx", "target_idx" with the integer indices of query and target objects that are *matched* and a column "score" with a numeric score for the match.

MatchedSpectra objects can be subset using:

- [subset the MatchedSpectra selecting query spectra to keep with parameter i. The target spectra will by default be returned as-is.

- pruneTarget *cleans* the MatchedSpectra object by removing non-matched target spectra.

In addition, MatchedSpectra can be filtered with any of the filtering approaches defined for Matched() objects: SelectMatchesParam(), TopRankedMatchesParam() or ScoreThresholdParam().

Extracting data

- $ extracts a single spectra variable from the MatchedSpectra x. Use spectraVariables to get all available spectra variables. Prefix "target_" is used for spectra variables from the *target* Spectra. The matching scores are available as *spectra variable* "score". Similar to a left join between the query and target spectra, this function returns a value for each query spectrum with eventual duplicated values for query spectra matching more than one target spectrum. If spectra variables from the target spectra are extracted, an NA is reported for *query* spectra that don't match any target spectra. See examples below for more details.
- length returns the number of **query** spectra.
- matchedData same as spectraData below.
- query returns the *query* Spectra.
- queryVariables returns the spectraVariables of *query*.
- spectraData returns spectra variables from the query and/or target Spectra as a DataFrame. Parameter columns allows to define which variables should be returned (defaults to columns = spectraVariables(object)), spectra variable names of the target spectra need to be prefixed with target_ (e.g. target_msLevel to get the MS level from target spectra). The score from the matching function is returned as spectra variable "score". Similar to $, this function performs a *left join* of spectra variables from the *query* and *target* spectra returning all values for all query spectra (eventually returning duplicated elements for query spectra matching multiple target spectra) and the values for the target spectra matched to the respective query spectra. See help on $ above or examples below for details.
- spectraVariables returns all available spectra variables in the *query* and *target* spectra. The prefix "target_" is used to label spectra variables of target spectra (e.g. the name of the spectra variable for the MS level of target spectra is called "target_msLevel").
- target returns the *target* Spectra.
- targetVariables returns the spectraVariables of *target* (prefixed with "target_").
- whichTarget returns an integer with the indices of the spectra in *target* that match at least on spectrum in *query*.
- whichQuery returns an integer with the indices of the spectra in *query* that match at least on spectrum in *target*.

Data manipulation and plotting

- addProcessing: add a processing step to both the *query* and *target* Spectra in object. Additional parameters for FUN can be passed *via* .... See addProcessing documentation in Spectra::Spectra() for more information.
- plotSpectraMirror: creates a mirror plot between the query and each matching target spectrum. Can only be applied to a MatchedSpectra with a single query spectrum. Setting parameter scalePeaks = TRUE will scale the peak intensities per spectrum to a total sum of one for a better graphical visualization. Additional plotting parameters can be passed through .... The parameters ppm and tolerance can be used to define the m/z tolerance for matching peaks between the query and target spectra. If not provided by the user, the values from the param object used to create the MatchedSpectra object are used; if these are missing, the default values (ppm =20 and tolerance = 0) are used.

- setBackend: allows to change the *backend* of both the query and target Spectra::Spectra() object. The function will return a MatchedSpectra object with the query and target Spectra changed to the specified backend, which can be any backend extending Spectra::MsBackend.

Author(s)

Johannes Rainer

See Also

Matched() for additional functions available for MatchedSpectra.

Examples

```
## Creating a dummy MatchedSpectra object.
library(Spectra)
df1 <- DataFrame(
    msLevel = 2L, rtime = 1:10,
    spectrum_id = c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j"))
df2 <- DataFrame(
    msLevel = 2L, rtime = rep(1:10, 20),
    spectrum_id = rep(c("A", "B", "C", "D", "E"), 20))
sp1 <- Spectra(df1)
sp2 <- Spectra(df2)
## Define matches between query spectrum 1 with target spectra 2 and 5,
## query spectrum 2 with target spectrum 2 and query spectrum 4 with target
## spectra 8, 12 and 15.
ms <- MatchedSpectra(
    sp1, sp2, matches = data.frame(query_idx = c(1L, 1L, 2L, 4L, 4L, 4L),
                                   target_idx = c(2L, 5L, 2L, 8L, 12L, 15L),
                                   score = 1:6))

## Which of the query spectra match at least one target spectrum?
whichQuery(ms)

## Extracting spectra variables: accessor methods for spectra variables act
## as "left joins", i.e. they return a value for each query spectrum, with
## eventually duplicated elements if one query spectrum matches more than
## one target spectrum.

## Which target spectrum matches at least one query spectrum?
whichTarget(ms)

## Extracting the retention times of the query spectra.
ms$rtime

## We have duplicated retention times for query spectrum 1 (matches 2 target
## spectra) and 4 (matches 3 target spectra). The retention time is returned
## for each query spectrum.

## Extracting retention times of the target spectra. Note that only retention
## times for target spectra matching at least one query spectrum are returned
## and an NA is reported for query spectra without matching target spectrum.
ms$target_rtime

## The first query spectrum matches target spectra 2 and 5, thus their
## retention times are returned as well as the retention time of the second
```

```
## target spectrum that matches also query spectrum 2. The 3rd query spectrum
## does match any target spectrum, thus `NA` is returned. Query spectrum 4
## matches target spectra 8, 12, and 15, thus the next reported retention
## times are those from these 3 target spectra. None of the remaining 6 query
## spectra matches any target spectra and thus `NA` is reported for each of
## them.

## With `queryIndex` and `targetIndex` it is possible to extract the indices
## of the matched query-index pairs
queryIndex(ms)
targetIndex(ms)

## The first match is between query index 1 and target index 2, the second
## match between query index 1 and target index 5 and so on.
## We could use these indices to extract a `Spectra` object containing only
## matched target spectra and assign a spectra variable with the indices of
## the query spectra
matched_target <- target(ms)[targetIndex(ms)]
matched_target$query_index <- queryIndex(ms)

## This `Spectra` object thus contains information from the matching, but
## is a *conventional* `Spectra` object that could be used for further
## analyses.

## `spectraData` can be used to extract all (or selected) spectra variables
## from the object. Same as with `$`, a left join between the specta
## variables from the query spectra and the target spectra is performed. The
## prefix `"target_"` is used to label the spectra variables from the target
## spectra. Below we extract selected spectra variables from the object.
res <- spectraData(ms, columns = c("rtime", "spectrum_id",
    "target_rtime", "target_spectrum_id"))
res
res$spectrum_id
res$target_spectrum_id

## Again, all values for query spectra are returned and for query spectra not
## matching any target spectrum NA is reported as value for the respecive
## variable.

## The example matched spectra object contains all query and all target
## spectra. Below we subset the object keeping only query spectra that are
## matched to at least one target spectrum.
ms_sub <- ms[whichQuery(ms)]

## ms_sub contains now only 3 query spectra:
length(query(ms_sub))

## while the original object contains all 10 query spectra:
length(query(ms))

## Both object contain however still the full target `Spectra`:
length(target(ms))
length(target(ms_sub))

## With the `pruneTarget` we can however reduce also the target spectra to
## only those that match at least one query spectrum
ms_sub <- pruneTarget(ms_sub)
```

```
length(target(ms_sub))
```

`matchFormula` *Chemical Formula Matching*

Description

The `matchFormula` method matches chemical formulas from different inputs (parameter `query` and `target`). Before comparison all formulas are normalized using `MetaboCoreUtils::standardizeFormula()`. Inputs can be either a `character` or `data.frame` containing a column with formulas. In case of `data.frames` parameter `formulaColname` needs to be used to specify the name of the column containing the chemical formulas.

Usage

```
matchFormula(query, target, ...)

## S4 method for signature 'character,character'
matchFormula(query, target, BPPARAM = SerialParam())

## S4 method for signature 'data.frameOrSimilar,data.frameOrSimilar'
matchFormula(
  query,
  target,
  formulaColname = c("formula", "formula"),
  BPPARAM = SerialParam()
)

## S4 method for signature 'character,data.frameOrSimilar'
matchFormula(
  query,
  target,
  formulaColname = "formula",
  BPPARAM = SerialParam()
)

## S4 method for signature 'data.frameOrSimilar,character'
matchFormula(
  query,
  target,
  formulaColname = "formula",
  BPPARAM = SerialParam()
)
```

Arguments

`query`	`character` or `data.frame` with chemical formulas to search.
`target`	`character` or `data.frame` with chemical formulas to compare against.
`...`	currently ignored
`BPPARAM`	parallel processing setup. See `BiocParallel::bpparam()` for details.

formulaColname `character` with the name of the column containing chemical formulas. Can be of length 1 if both `query` and `target` are `data.frames` and the name of the column with chemical formulas is the same for both. If different columns are used, `formulaColname[1]` can be used to define the column name in `query` and `formulaColname[2]` the one of `target`.

Value

Matched object representing the result.

Author(s)

Michael Witting

Examples

```
## input formula
query <- c("H12C6O6", "C11H12O2", "HN3")
target <- c("HCl", "C2H4O", "C6H12O6")

query_df <- data.frame(
    formula = c("H12C6O6", "C11H12O2", "HN3"),
    name = c("A", "B", "C")
)
target_df <- data.frame(
    formula = c("HCl", "C2H4O", "C6H12O6"),
    name = c("D", "E", "F")
)

## character vs character
matches <- matchFormula(query, target)
matchedData(matches)

## data.frame vs data.frame
matches <- matchFormula(query_df, target_df)
matchedData(matches)
## data.frame vs character
matches <- matchFormula(query_df, target)
matchedData(matches)
## character vs data.frame
matches <- matchFormula(query, target_df)
matchedData(matches)
```

---

matchSpectra *Spectral matching*

Description

The `matchSpectra` method matches (compares) spectra from `query` with those from `target` based on settings specified with `param` and returns the result from this as a MatchedSpectra object.

Usage

```
matchSpectra(query, target, param, ...)
```

Arguments

query	Spectra::Spectra object with the (experimental) spectra.
target	MS data to compare against. Can be another Spectra::Spectra.
param	parameter object containing the settings for the matching (e.g. eventual pre-filtering settings, cut-off value for similarity above which spectra are considered matching etc).
...	optional parameters.

Value

a MatchedSpectra object with the spectra matching results.

Author(s)

Johannes Rainer

See Also

CompareSpectraParam() for the comparison between Spectra::Spectra objects.

---

matchSpectra,Spectra,CompDbSource,Param-method

Matching MS Spectra against a reference

Description

matchSpectra compares experimental (*query*) MS2 spectra against reference (*target*) MS2 spectra and reports matches with a similarity that passing a specified threshold. The function performs the similarity calculation between each query spectrum against each target spectrum. Parameters query and target can be used to define the query and target spectra, respectively, while parameter param allows to define and configure the similarity calculation and matching condition. Parameter query takes a Spectra::Spectra object while target can be either a Spectra::Spectra object, a CompoundDb::CompDb (reference library) object defined in the CompoundDb package or a CompAnnotationSource (e.g. a CompDbSource()) with the reference or connection information to a supported annotation resource).

Some notes on performance and information on parallel processing are provided in the vignette.

Currently supported parameter objects defining the matching are:

- CompareSpectraParam: the *generic* parameter object allowing to set all settings for the Spectra::compareSpectr call that is used to perform the similarity calculation. This includes MAPFUN and FUN defining the peak-mapping and similarity calculation functions and ppm and tolerance to define an acceptable difference between m/z values of the compared peaks. Parameter matchedPeaksCount is also passed to compareSpectra() and, if set to TRUE (default is FALSE) will report the number of peaks defined to be *matching* by the MAPFUN. Additional parameters to the compareSpectra call can be passed along with .... See the help of Spectra::Spectra() for more information on these parameters. Importantly, if *msentropy* or a GNPS-like similarity calculation is used, MAPFUN should be selected accordingly (see section *Using alternative spectra similarity functions* in the package vignette for more information). By default, parameters ppm and tolerance are passed to the similarity calculation function, but if this function uses different parameters (e.g., msentropy_similarity() uses ms2_tolerance_in_ppm instead of

ppm), these should be submitted to the CompareSpectraParam() function throught the ... parameter. Parameters requirePrecursor (default TRUE) and requirePrecursorPeak (default FALSE) allow to pre-filter the target spectra prior to the actual similarity calculation for each individual query spectrum. Parameters ppm and tolerance are also used to define the maximal acceptable difference in precursor m/z if requirePrecursor or requirePrecursorPeak are set to TRUE. Target spectra can also be pre-filtered based on retention time if parameter toleranceRt is set to a value different than the default toleranceRt = Inf. Only target spectra with a retention time within the query's retention time +/- (toleranceRt + percentRt% of the query's retention time) are considered. Note that while for ppm and tolerance only a single value is accepted, toleranceRt and percentRt can be also of length equal to the number of query spectra hence allowing to define different rt boundaries for each query spectrum. While these pre-filters can considerably improve performance, it should be noted that no matches will be found between query and target spectra with missing values in the considered variable (precursor m/z or retention time). For target spectra without retention times (such as for Spectra from a public reference database such as MassBank) the default toleranceRt = Inf should thus be used. Finally, parameter THRESHFUN allows to define a function to be applied to the similarity scores to define which matches to report. See below for more details.

- MatchForwardReverseParam: performs spectra matching as with CompareSpectraParam but reports, similar to MS-DIAL, also the *reverse* similarity score and the *presence ratio*. Please refer to the documentation of CompareSpectraParam for explanation of the parameters. With MatchForwardReverseParam, the matching of query spectra to target spectra is performed by considering all peaks from the query and all peaks from the target (reference) spectrum (i.e. *forward* matching using an *outer join*-based peak matching strategy). For matching spectra also the *reverse* similarity is calculated considering only peaks present in the target (reference) spectrum (i.e. using a *right join*-based peak matching). This is reported as spectra variable "reverse_score". In addition, the ratio between the number of matched peaks and the total number of peaks in the target (reference) spectra is reported as the *presence ratio* (spectra variable "presence_ratio") and the total number of matched peaks as "matched_peaks_count". See examples below for details. Parameter THRESHFUN_REVERSE allows to define an additional *threshold function* to filter matches. If THRESHFUN_REVERSE is defined only matches with a spectra similarity fulfilling both THRESHFUN **and** THRESHFUN_REVERSE are returned. With the default THRESHFUN_REVERSE = NULL all matches passing THRESHFUN are reported.

Usage

```
## S4 method for signature 'Spectra,CompDbSource,Param'
matchSpectra(
  query,
  target,
  param,
  BPPARAM = BiocParallel::SerialParam(),
  addOriginalQueryIndex = TRUE
)

CompareSpectraParam(
  MAPFUN = joinPeaks,
  tolerance = 0,
  ppm = 5,
  FUN = MsCoreUtils::ndotproduct,
  requirePrecursor = TRUE,
  requirePrecursorPeak = FALSE,
```

```
  THRESHFUN = function(x) which(x >= 0.7),
  toleranceRt = Inf,
  percentRt = 0,
  matchedPeaksCount = FALSE,
  ...
)

MatchForwardReverseParam(
  MAPFUN = joinPeaks,
  tolerance = 0,
  ppm = 5,
  FUN = MsCoreUtils::ndotproduct,
  requirePrecursor = TRUE,
  requirePrecursorPeak = FALSE,
  THRESHFUN = function(x) which(x >= 0.7),
  THRESHFUN_REVERSE = NULL,
  toleranceRt = Inf,
  percentRt = 0,
  ...
)

## S4 method for signature 'Spectra,Spectra,CompareSpectraParam'
matchSpectra(
  query,
  target,
  param,
  rtColname = c("rtime", "rtime"),
  BPPARAM = BiocParallel::SerialParam(),
  addOriginalQueryIndex = TRUE
)

## S4 method for signature 'Spectra,CompDb,Param'
matchSpectra(
  query,
  target,
  param,
  rtColname = c("rtime", "rtime"),
  BPPARAM = BiocParallel::SerialParam(),
  addOriginalQueryIndex = TRUE
)

## S4 method for signature 'Spectra,Spectra,MatchForwardReverseParam'
matchSpectra(
  query,
  target,
  param,
  rtColname = c("rtime", "rtime"),
  BPPARAM = BiocParallel::SerialParam(),
  addOriginalQueryIndex = TRUE
)
```

Arguments

query	for matchSpectra: Spectra::Spectra object with the query spectra.
target	for matchSpectra: Spectra::Spectra, CompoundDb::CompDb or object extending CompAnnotationSource (such as CompDbSource) with the target (reference) spectra to compare query against.
param	for matchSpectra: parameter object (such as CompareSpectraParam) defining the settings for the matching.
BPPARAM	for matchSpectra: parallel processing setup (see the BiocParallel package for more information). Parallel processing is disabled by default (with the default setting BPPARAM = SerialParam()).
addOriginalQueryIndex	for matchSpectra(): logical(1) whether an additional spectra variable ".original_query_index" should be added to the query Spectra object providing the index of the spectrum in this originally provided object. This spectra variable can be useful to link back to the original Spectra object if the MatchedSpectra object gets subsetted/processed.
MAPFUN	function used to map peaks between the compared spectra. Defaults for CompareSpectraParam to Spectra::joinPeaks(). See Spectra::compareSpectra() for details.
tolerance	numeric(1) for an absolute maximal accepted difference between m/z values. This will be used in compareSpectra as well as for eventual precursor m/z matching.
ppm	numeric(1) for a relative, m/z-dependent, maximal accepted difference between m/z values. This will be used in compareSpectra as well as for eventual precursor m/z matching.
FUN	function used to calculate similarity between spectra. Defaults for CompareSpectraParam to MsCoreUtils::ndotproduct(). See MsCoreUtils::ndotproduct() for details.
requirePrecursor	logical(1) whether only target spectra are considered in the similarity calculation with a precursor m/z that matches the precursor m/z of the query spectrum (considering also ppm and tolerance). With requirePrecursor = TRUE (the default) the function will complete much faster, but will not find any hits for target (or query spectra) with missing precursor m/z. It is suggested to check first the availability of the precursor m/z in target and query.
requirePrecursorPeak	logical(1) whether only target spectra will be considered in the spectra similarity calculation that have a peak with an m/z matching the precursor m/z of the query spectrum. Defaults to requirePrecursorPeak = FALSE. It is suggested to check first the availability of the precursor m/z in query, as no match will be reported for query spectra with missing precursor m/z.
THRESHFUN	function applied to the similarity score to define which target spectra are considered *matching*. Defaults to THRESHFUN = function(x) which(x >= 0.7) hence selects all target spectra matching a query spectrum with a similarity higher or equal than 0.7. Any function that takes a numeric vector with similarity scores from the comparison of a query spectrum with all target spectra (as returned by Spectra::compareSpectra()) as input and returns a logical vector (same dimensions as the similarity scores) or an integer with the matches is supported.
toleranceRt	numeric of length 1 or equal to the number of query spectra defining the maximal accepted (absolute) difference in retention time between query and target

spectra. By default (with `toleranceRt = Inf`) the retention time-based filter is not considered. See help of `CompareSpectraParam` above for more information.

`percentRt` `numeric` of length 1 or equal to the number of query spectra defining the maximal accepted relative difference in retention time between query and target spectra expressed in percentage of the query rt. For `percentRt = 10`, similarities are defined between the query spectrum and all target spectra with a retention time within query rt +/- 10% of the query. By default (with `toleranceRt = Inf`) the retention time-based filter is not considered. Thus, to consider the `percentRt` parameter, `toleranceRt` should be set to a value different than that. See help of `CompareSpectraParam` above for more information.

`matchedPeaksCount` `logical(1)` for `CompareSpectraParam()`: whether also the number of matching peaks should be reported (in column `"matched_peaks_count"`). This number represents the number of peaks reported *matching* by the `MAPFUN`.

`...` for `CompareSpectraParam`: additional parameters passed along to the `Spectra::compareSpectra()` call, including eventual additional parameters of the selected mapping or similarity calculation functions.

`THRESHFUN_REVERSE` for `MatchForwardReverseParam`: optional additional *thresholding function* to filter the results on the reverse score. If specified the same format than `THRESHFUN` is expected.

`rtColname` `character(2)` with the name of the spectra variable containing the retention time information for compounds to be used in retention time matching (only used if `toleranceRt` is not `Inf`). It can also be `character(1)` if the two names are the same. Defaults to `rtColname = c("rtime", "rtime")`.

Value

`matchSpectra` returns a `MatchedSpectra()` object with the matching results. If `target` is a `CompAnnotationSource` only matching target spectra will be reported.

Constructor functions return an instance of the class.

Author(s)

Johannes Rainer, Michael Witting

Examples

```
library(Spectra)
library(msdata)
fl <- system.file("TripleTOF-SWATH", "PestMix1_DDA.mzML", package = "msdata")
pest_ms2 <- filterMsLevel(Spectra(fl), 2L)

## subset to selected spectra.
pest_ms2 <- pest_ms2[c(808, 809, 945:955)]

## Load a small example MassBank data set
load(system.file("extdata", "minimb.RData", package = "MetaboAnnotation"))

## Match spectra with the default similarity score (normalized dot product)
csp <- CompareSpectraParam(requirePrecursor = TRUE, ppm = 10)
mtches <- matchSpectra(pest_ms2, minimb, csp)
```

```
mtches

## Are there any matching spectra for the first query spectrum?
mtches[1]
## No

## And for the second query spectrum?
mtches[2]
## The second query spectrum matches 4 target spectra. The scores for these
## matches are:
mtches[2]$score

## To access the score for the full data set
mtches$score

## Below we use a THRESHFUN that returns for each query spectrum the (first)
## best matching target spectrum.
csp <- CompareSpectraParam(requirePrecursor = FALSE, ppm = 10,
    THRESHFUN = function(x) which.max(x))
mtches <- matchSpectra(pest_ms2, minimb, csp)
mtches

## Each of the query spectra is matched to one target spectrum
length(mtches)
matches(mtches)

## Match spectra considering also measured retention times. This requires
## that both query and target spectra have non-missing retention times.
rtime(pest_ms2)
rtime(minimb)

## Target spectra don't have retention times. Below we artificially set
## retention times to show how an additional retention time filter would
## work.
rtime(minimb) <- rep(361, length(minimb))

## Matching spectra requiring a matching precursor m/z and the difference
## of retention times between query and target spectra to be <= 2 seconds.
csp <- CompareSpectraParam(requirePrecursor = TRUE, ppm = 10,
    toleranceRt = 2)
mtches <- matchSpectra(pest_ms2, minimb, csp)
mtches
matches(mtches)

## Note that parameter `rtColname` can be used to define different spectra
## variables with retention time information (such as retention indices etc).

## A `CompDb` compound annotation database could also be used with
## parameter `target`. Below we load the test `CompDb` database from the
## `CompoundDb` Bioconductor package.
library(CompoundDb)
fl <- system.file("sql", "CompDb.MassBank.sql", package = "CompoundDb")
cdb <- CompDb(fl)
res <- matchSpectra(pest_ms2, cdb, CompareSpectraParam())

## We do however not find any matches since the used compound annotation
## database contains only a very small subset of the MassBank.
```

```
res

## As `target` we have now however the MS2 spectra data from the compound
## annotation database
target(res)

## See the package vignette for details, descriptions and more examples,
## also on how to retrieve e.g. MassBank reference databases from
## Bioconductor's AnnotationHub.
```

`validateMatchedSpectra` *Validating MatchedSpectra*

Description

The `validateMatchedSpectra()` function opens a simple shiny application that allows to browse results stored in a `MatchedSpectra` object and to *validate* the presented matches. For each query spectrum a table with matched target spectra are shown (if available) and an interactive mirror plot is generated. Valid matches can be selected using a check box which is displayed below the mirror plot. Upon pushing the "Save & Close" button the app is closed and a filtered `MatchedSpectra` is returned, containing only *validated* matches.

Note that column `"query_index_"` and `"target_index_"` are temporarily added to the query and target `Spectra` object to display them in the interactive graphics for easier identification of the compared spectra.

Usage

```
validateMatchedSpectra(object)
```

Arguments

`object`	A non-empty instance of class `MatchedSpectra`.

Value

A `MatchedSpectra` with validated results.

Author(s)

Carolin Huber, Michael Witting, Johannes Rainer

Examples

```
library(Spectra)
## Load test data
fl <- system.file("TripleTOF-SWATH", "PestMix1_DDA.mzML", package = "msdata")
pest_ms2 <- filterMsLevel(Spectra(fl), 2L)
pest_ms2 <- pest_ms2[c(808, 809, 945:955)]
load(system.file("extdata", "minimb.RData", package = "MetaboAnnotation"))

## Normalize intensities and match spectra
csp <- CompareSpectraParam(requirePrecursor = TRUE,
```

```
                            THRESHFUN = function(x) x >= 0.7)
norm_int <- function(x) {
    x[, "intensity"] <- x[, "intensity"] / max(x[, "intensity"]) * 100
    x
}
ms <- matchSpectra(addProcessing(pest_ms2, norm_int),
                   addProcessing(minimb, norm_int), csp)

## validate matches using the shiny app. Note: the call is only executed
## in interactive mode.
if (interactive()) {
    res <- validateMatchedSpectra(ms)
}
```

`ValueParam` *Matching of numeric values*

Description

The `matchValues` method matches elements from `query` with those in `target` using different matching approaches depending on parameter `param`. Generally, `query` is expected to contain MS experimental values (m/z and possibly retention time) while `target` reference values. `query` and `target` can be `numeric`, a two dimensional array (such as a `data.frame`, `matrix` or `DataFrame`), a `SummarizedExperiment` or a `QFeatures`, `target` can in addition be a `Spectra::Spectra()` object. For `SummarizedExperiment`, the information for the matching is expected to be in the object's `rowData`. For `QFeatures` matching is performed for values present in the `rowData` of one of the object's assays (which needs to be specified with the `assayQuery` parameter - if a `QFeatures` is used as `target` the name of the assay needs to be specified with parameter `assayTarget`). If `target` is a `Spectra` matching is performed against spectra variables of this object and the respective variable names need to be specified e.g. with `mzColname` and/or `rtColname`. `matchMz` is an alias for `matchValues` to allow backward compatibility.

Available `param` objects and corresponding matching approaches are:

- `ValueParam`: generic matching between values in `query` and `target` given acceptable differences expressed in `ppm` and `tolerance`. If `query` or `target` are not numeric, parameter `valueColname` has to be used to specify the name of the column that contains the values to be matched. The function returns a `Matched()` object.
- `MzParam`: match query m/z values against reference compounds for which also m/z are known. Matching is performed similarly to the `ValueParam` above. If `query` or `target` are not numeric, the column name containing the values to be compared must be defined with `matchValues`' parameter `mzColname`, which defaults to `"mz"`. `MzParam` parameters `tolerance` and `ppm` allow to define the maximal acceptable (constant or m/z relative) difference between query and target m/z values.
- `MzRtParam`: match m/z **and** retention time values between `query` and `target`. Parameters `mzColname` and `rtColname` of the `matchValues` function allow to define the columns in `query` and `target` containing these values (defaulting to `c("mz", "mz")` and `c("rt", "rt")`, respectively). `MzRtParam` parameters `tolerance` and `ppm` have the same meaning as in `MzParam`; `MzRtParam` parameter `toleranceRt` allows to specify the maximal acceptable difference between query and target retention time values.

- `Mass2MzParam`: match m/z values against reference compounds for which only the (exact) mass is known. Before matching, m/z values are calculated from the compounds masses in the *target* table using the adducts specified via `Mass2MzParam` `adducts` parameter (defaults to `adducts = "[M+H]+"`). After conversion of adduct masses to m/z values, matching is performed similarly to `MzParam` (i.e. the same parameters `ppm` and `tolerance` can be used). If `query` is not `numeric`, parameter `mzColname` of `matchValues` can be used to specify the column containing the query's m/z values (defaults to `"mz"`). If `target` is a is not `numeric`, parameter `massColname` can be used to define the column containing the reference compound's masses (defaults to `"exactmass"`).
- `Mass2MzRtParam`: match m/z **and** retention time values against reference compounds for which the (exact) mass **and** retention time are known. Before matching, exact masses in `target` are converted to m/z values as for `Mass2MzParam`. Matching is then performed similarly to `MzRtParam`, i.e. m/z and retention times of entities are compared. With `matchValues`' parameters `mzColname`, `rtColname` and `massColname` the columns containing m/z values (in `query`), retention time values (in `query` and `target`) and exact masses (in `target`) can be specified.
- `Mz2MassParam`: input values for `query` and `target` are expected to be m/z values but matching is performed on exact masses calculated from these (based on the provided adduct definitions). In detail, m/z values in `query` are first converted to masses with the `MetaboCoreUtils::mz2mass()` function based on the adducts defined with `queryAdducts` (defaults to `"[M+H]+"`). The same is done for m/z values in `target` (adducts can be defined with `targetAdducts` which defaults to `"[M-H-]"`). Matching is then performed on these converted values similarly to ValueParam. If queryortargetare not numeric, the column containing the m/z values can be specified wi Colname(defaults to"mz"‘).
- `Mz2MassRtParam`: same as `Mz2MassParam` but with additional comparison of retention times between `query` and `target`. Parameters `rtColname` and `mzColname` of `matchValues` allow to specify which columns contain the retention times and m/z values, respectively.

Usage

```
ValueParam(tolerance = 0, ppm = 5)

MzParam(tolerance = 0, ppm = 5)

Mass2MzParam(adducts = c("[M+H]+"), tolerance = 0, ppm = 5)

Mass2MzRtParam(adducts = c("[M+H]+"), tolerance = 0, ppm = 5, toleranceRt = 0)

MzRtParam(tolerance = 0, ppm = 0, toleranceRt = 0)

Mz2MassParam(
  queryAdducts = c("[M+H]+"),
  targetAdducts = c("[M-H]-"),
  tolerance = 0,
  ppm = 5
)

Mz2MassRtParam(
  queryAdducts = c("[M+H]+"),
  targetAdducts = c("[M+H]+"),
  tolerance = 0,
  ppm = 5,
```

```
  toleranceRt = 0
)

matchValues(query, target, param, ...)

## S4 method for signature 'numeric,numeric,ValueParam'
matchValues(query, target, param)

## S4 method for signature 'numeric,data.frameOrSimilar,ValueParam'
matchValues(
  query,
  target,
  param,
  valueColname = character(),
  targetAssay = character()
)

## S4 method for signature 'data.frameOrSimilar,numeric,ValueParam'
matchValues(
  query,
  target,
  param,
  valueColname = character(),
  queryAssay = character()
)

## S4 method for signature 'data.frameOrSimilar,data.frameOrSimilar,ValueParam'
matchValues(
  query,
  target,
  param,
  valueColname = character(),
  queryAssay = character(),
  targetAssay = character()
)

## S4 method for signature 'numeric,numeric,Mass2MzParam'
matchValues(query, target, param)

## S4 method for signature 'numeric,data.frameOrSimilar,Mass2MzParam'
matchValues(
  query,
  target,
  param,
  massColname = "exactmass",
  targetAssay = character()
)

## S4 method for signature 'data.frameOrSimilar,numeric,Mass2MzParam'
matchValues(query, target, param, mzColname = "mz", queryAssay = character())

## S4 method for signature
```

```
## 'data.frameOrSimilar,data.frameOrSimilar,Mass2MzParam'
matchValues(
  query,
  target,
  param,
  mzColname = "mz",
  massColname = "exactmass",
  queryAssay = character(0),
  targetAssay = character(0)
)

## S4 method for signature 'numeric,data.frameOrSimilar,MzParam'
matchValues(query, target, param, mzColname = "mz", targetAssay = character())

## S4 method for signature 'numeric,Spectra,MzParam'
matchValues(query, target, param, mzColname = "mz", targetAssay = character())

## S4 method for signature 'data.frameOrSimilar,numeric,MzParam'
matchValues(query, target, param, mzColname = "mz", queryAssay = character())

## S4 method for signature 'data.frameOrSimilar,data.frameOrSimilar,MzParam'
matchValues(
  query,
  target,
  param,
  mzColname = c("mz", "mz"),
  queryAssay = character(),
  targetAssay = character()
)

## S4 method for signature 'data.frameOrSimilar,Spectra,MzParam'
matchValues(
  query,
  target,
  param,
  mzColname = c("mz", "mz"),
  queryAssay = character(),
  targetAssay = character()
)

## S4 method for signature
## 'data.frameOrSimilar,data.frameOrSimilar,Mass2MzRtParam'
matchValues(
  query,
  target,
  param,
  massColname = "exactmass",
  mzColname = "mz",
  rtColname = c("rt", "rt"),
  queryAssay = character(),
  targetAssay = character()
)
```

```
## S4 method for signature 'data.frameOrSimilar,data.frameOrSimilar,MzRtParam'
matchValues(
  query,
  target,
  param,
  mzColname = c("mz", "mz"),
  rtColname = c("rt", "rt"),
  queryAssay = character(),
  targetAssay = character()
)

## S4 method for signature 'data.frameOrSimilar,Spectra,MzRtParam'
matchValues(
  query,
  target,
  param,
  mzColname = c("mz", "mz"),
  rtColname = c("rt", "rt"),
  queryAssay = character(),
  targetAssay = character()
)

## S4 method for signature 'numeric,numeric,Mz2MassParam'
matchValues(query, target, param)

## S4 method for signature 'numeric,data.frameOrSimilar,Mz2MassParam'
matchValues(query, target, param, mzColname = "mz", targetAssay = character())

## S4 method for signature 'data.frameOrSimilar,numeric,Mz2MassParam'
matchValues(query, target, param, mzColname = "mz", queryAssay = character())

## S4 method for signature
## 'data.frameOrSimilar,data.frameOrSimilar,Mz2MassParam'
matchValues(
  query,
  target,
  param,
  mzColname = c("mz", "mz"),
  queryAssay = character(),
  targetAssay = character()
)

## S4 method for signature
## 'data.frameOrSimilar,data.frameOrSimilar,Mz2MassRtParam'
matchValues(
  query,
  target,
  param,
  mzColname = c("mz", "mz"),
  rtColname = c("rt", "rt"),
  queryAssay = character(),
```

```
  targetAssay = character()
)
```

Arguments

tolerance	for any param object: numeric(1) defining the maximal acceptable absolute difference in m/z (or in mass for Mz2MassParam) to consider them *matching*.
ppm	for any param object: numeric(1) defining the maximal acceptable m/z-dependent (or mass-dependent for Mz2MassParam) difference (in parts-per-million) in m/z values to consider them to be *matching*.
adducts	for Mass2MzParam or Mass2MzRtParam: either character with adduct names from MetaboCoreUtils::adducts() or data.frame with a custom adduct definition. This parameter is used to calculate m/z from target compounds' masses. Custom adduct definitions can be passed to the adduct parameter in form of a data.frame. This data.frame is expected to have columns "mass_add" and "mass_multi" defining the *additive* and *multiplicative* part of the calculation. See MetaboCoreUtils::adducts() for the expected format or use MetaboCoreUtils::adductName and MetaboCoreUtils::adductNames("negative") for valid adduct names.
toleranceRt	for Mass2MzRtParam or MzRtParam: numeric(1) defining the maximal acceptable absolute difference in retention time values to consider them them *matching*.
queryAdducts	for Mz2MassParam. Adducts used to derive mass values from query m/z values. The expected format is the same as that for parameter adducts.
targetAdducts	for Mz2MassParam. Adducts used to derive mass values from target m/z values. The expected format is the same as that for parameter adducts.
query	feature table containing information on MS1 features. Can be a numeric, data.frame, DataFrame, matrix, SummarizedExperiment or QFeatures. It is expected to contain m/z values and can contain also other variables. Matchings based on both m/z and retention time can be performed when a column with retention times is present in both query and target.
target	compound table with metabolites to compare against. The expected types are the same as those for query.
param	parameter object defining the matching approach and containing the settings for that approach. See description above for details.
...	currently ignored.
valueColname	character specifying the name of the column in query or/and the one in targetwith the desired values for the matching. This parameter should only be used when param is valueParam and in this case it must be provided (unless both query and target are numeric). It can be character(1) or character(2) in a similar way to mzColname.
targetAssay	character(1) specifying the name of the assay of the provided QFeatures that should be used for the matching (values from this assay's rowData will be used for matching). Only used if target is an instance of a QFeatures object.
queryAssay	character(1) specifying the name of the assay of the provided QFeatures that should be used for the matching (values from this assay's rowData will be used for matching). Only used if query is an instance of a QFeatures object.
massColname	character(1) with the name of the column in target containing the mass of compounds. To be used when param is Mass2MzParam or Mass2MzRtParam (and target is not already numeric with the masses). Defaults to massColname = "exactmass".

mzColname — character specifying the name(s) of the column(s) in query or/and targetwith the m/z values. If one among query and target is numeric (and therefore there is no need to specify the column name) or query is not numeric and param is Mass2MzParam or Mass2MzRtParam (and therefore the name of the column with m/z needs only to be specified for query) then mzColname is expected to be character(1). If both query and target are not numeric mzColname is expected to be character(2) (or character(1) and in this last case the two column names are assumed to be the same). If not specified the assumed default name for columns with m/z values is "mz". If target is a Spectra::Spectra() object, the name of the spectra variable that should be used for the matching needs to be specified with mzColname.

rtColname — character(2) with the name of the column containing the compounds retention times in query and the name for the one in target. It can also be character(1) if the two names are the same. To be used when param is MzRtParam or Mass2MzRtParam. Defaults to rtColname = c("rt", "rt"). If target is a Spectra::Spectra() object, the name of the spectra variable that should be used for the matching needs to be specified with mzColname.

Value

Matched object representing the result.

Depending on the param object different *scores* representing the quality of the match are provided. This comprises absolute as well as relative differences (column/variables "score" and "ppm_error" respectively). If param is a Mz2MassParam, "score" and "ppm_error" represent differences of the compared masses (calculated from the provided m/z values). If param an MzParam, MzRtParam, Mass2MzParam or Mass2MzRtParam, "score" and "ppm_error" represent absolute and relative differences of m/z values. Additionally, if param is either an MzRtParam or Mass2MzRtParam differences between query and target retention times for each matched element is available in the column/variable "score_rt" in the returned Matched object. Negative values of "score" (or "score_rt") indicate that the m/z or mass (or retention time) of the query element is smaller than that of the target element.

Author(s)

Andrea Vicini, Michael Witting

See Also

matchSpectra or CompareSpectraParam() for spectra data matching

Examples

```
library(MetaboCoreUtils)
## Create a simple "target/reference" compound table
target_df <- data.frame(
   name = c("Tryptophan", "Leucine", "Isoleucine"),
   formula = c("C11H12N2O2", "C6H13NO2", "C6H13NO2"),
   exactmass = c(204.089878, 131.094629, 131.094629)
)

## Create a "feature" table with m/z of features. We calculate m/z for
## certain adducts of some of the compounds in the reference table.
fts <- data.frame(
    feature_id = c("FT001", "FT002", "FT003"),
```

```
    mz = c(mass2mz(204.089878, "[M+H]+"),
           mass2mz(131.094629, "[M+H]+"),
           mass2mz(204.089878, "[M+Na]+") + 1e-6))

## Define the parameters for the matching
parm <- Mass2MzParam(
    adducts = c("[M+H]+", "[M+Na]+"),
    tolerance = 0,
    ppm = 20)
res <- matchValues(fts, target_df, parm)
res

## List the available variables/columns
colnames(res)

## feature_id and mz are from the query data frame, while target_name,
## target_formula and target_exactmass are from the query object (columns
## from the target object have a prefix *target_* added to the original
## column name. Columns adduct, score and ppm_error represent the results
## of the matching: adduct the adduct/ion of the original compound for which
## the m/z matches, score the absolute difference of the query and target
## m/z and ppm_error the relative difference in m/z values.

## Get the full matching result:
matchedData(res)

## We have thus matches of FT002 to two different compounds (but with the
## same mass).

## Individual columns can also be accessed with the $ operator:
res$feature_id
res$target_name
res$ppm_error


## We repeat the matching requiring an exact match
parm <- Mass2MzParam(
    adducts = c("[M+H]+", "[M+Na]+"),
    tolerance = 0,
    ppm = 0)
res <- matchValues(fts, target_df, parm)
res

matchedData(res)

## The last feature could thus not be matched to any compound.

## At last we use also different adduct definitions.
parm <- Mass2MzParam(
    adducts = c("[M+K]+", "[M+Li]+"),
    tolerance = 0,
    ppm = 20)
res <- matchValues(fts, target_df, parm)
res

matchedData(res)
```

```
## No matches were found.

## We can also match a "feature" table with a target data.frame taking into
## account both m/z and retention time values.
target_df <- data.frame(
  name = c("Tryptophan", "Leucine", "Isoleucine"),
  formula = c("C11H12N2O2", "C6H13NO2", "C6H13NO2"),
  exactmass = c(204.089878, 131.094629, 131.094629),
  rt = c(150, 140, 140)
)

fts <- data.frame(
  feature_id = c("FT001", "FT002", "FT003"),
  mz = c(mass2mz(204.089878, "[M+H]+"),
         mass2mz(131.094629, "[M+H]+"),
         mass2mz(204.089878, "[M+Na]+") + 1e-6),
  rt = c(150, 140, 150.1)
)

## Define the parameters for the matching
parm <- Mass2MzRtParam(
  adducts = c("[M+H]+", "[M+Na]+"),
  tolerance = 0,
  ppm = 20,
  toleranceRt = 0)

res <- matchValues(fts, target_df, parm)
res

## Get the full matching result:
matchedData(res)

## FT003 could not be matched to any compound, FT002 was matched to two
## different compounds (but with the same mass).

## We repeat the matching allowing a positive tolerance for the matches
## between rt values

## Define the parameters for the matching
parm <- Mass2MzRtParam(
  adducts = c("[M+H]+", "[M+Na]+"),
  tolerance = 0,
  ppm = 20,
  toleranceRt = 0.1)

res <- matchValues(fts, target_df, parm)
res

## Get the full matching result:
matchedData(res)

## Also FT003 was matched in this case

## It is also possible to match directly m/z values
mz1 <- c(12, 343, 23, 231)
mz2 <- mz1 + rnorm(4, sd = 0.001)
```

```
res <- matchValues(mz1, mz2, MzParam(tolerance = 0.001))

matchedData(res)

## Matching with a SummarizedExperiment or a QFeatures work analogously,
## only that the matching is performed on the object's `rowData`.

## Below we create a simple SummarizedExperiment with some random assay data.
## Note that results from a data preprocessing with the `xcms` package could
## be extracted as a `SummarizedExperiment` with the `quantify` method from
## the `xcms` package.
library(SummarizedExperiment)
se <- SummarizedExperiment(
    assays = matrix(rnorm(12), nrow = 3, ncol = 4,
    dimnames = list(NULL, c("A", "B", "C", "D"))),
    rowData = fts)

## We can now perform the matching of this SummarizedExperiment against the
## target_df as before.
res <- matchValues(se, target_df,
    param = Mass2MzParam(adducts = c("[M+H]+", "[M+Na]+"),
        tolerance = 0, ppm = 20))
res

## Getting the available columns
colnames(res)

## The query columns represent the columns of the object's `rowData`
rowData(se)

## matchedData also returns the query object's rowData along with the
## matching entries in the target object.
matchedData(res)

## While `query` will return the full SummarizedExperiment.
query(res)

## To illustrate use with a QFeatures object we first create a simple
## QFeatures object with two assays, `"ions"` representing the full feature
## data.frame and `"compounds"` a subset of it.
library(QFeatures)
qf <- QFeatures(list(ions = se, compounds = se[2,]))

## We can perform the same matching as before, but need to specify which of
## the assays in the QFeatures should be used for the matching. Below we
## perform the matching using the "ions" assay.
res <- matchValues(qf, target_df, queryAssay = "ions",
    param = Mass2MzParam(adducts = c("[M+H]+", "[M+Na]+"),
        tolerance = 0, ppm = 20))
res

## colnames returns now the colnames of the `rowData` of the `"ions"` assay.
colnames(res)

matchedData(res)
```

Index