Package ‘ttservice’

July 10, 2025

Type Package

Title A Service for Tidy Transcriptomics Software Suite

Version 0.5.3

Description It provides generic methods that are used by more than one package, avoiding conflicts. This package will be imported by 'tidySingleCellExperiment' and 'tidyseurat'.

License GPL-3

Depends R (>= 4.0.0)

Imports dplyr, Matrix, plotly

Suggests methods

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation no

Author Stefano Mangiola [aut, cre]

Maintainer Stefano Mangiola <`mangiolastefano@gmail.com`>

Repository CRAN

Date/Publication 2025-07-10 12:30:02 UTC

Contents

`aggregate_cells` *Aggregate cells*

Description

Combine cells into groups based on shared variables and aggregate feature counts.

Usage

```
aggregate_cells(
  .data,
  .sample = NULL,
  slot = "data",
  assays = NULL,
  aggregation_function = Matrix::rowSums,
  ...
)
```

Arguments

`.data`	A tidySingleCellExperiment object
`.sample`	A vector of variables by which cells are aggregated
`slot`	The slot to which the function is applied
`assays`	The assay to which the function is applied
`aggregation_function`	The method of cell-feature value aggregation
`...`	Used for future extendibility

Value

A tibble object

Examples

```
print("pbmc_small |> aggregate_cells(c(groups, ident), assays = \"counts\")")
```

`append_samples` *Append samples*

Description

Append multiple samples or datasets together, combining their data while preserving sample-specific information.

Usage

```
append_samples(x, ...)
```

Arguments

`x`	A genomic data container to combine with others
`...`	Additional genomic data containers to combine Each argument should be a genomic data object such as a SummarizedExperiment, SingleCellExperiment, SpatialExperiment, or Seurat object (provided that the appropriate method extensions are available). You may also provide a list of such objects. When row-binding, features (e.g., genes) are matched by name, and any missing features will be filled with NA or zero as appropriate for the container. When column-binding, samples (e.g., cells) are matched by position, so all objects must have the same number of features. To match by value, not position, see mutate-joins.

Value

A combined genomic object

Examples

```
print("combined_data <- append_samples(sample1, sample2, .id = \"sample\")")
```

`bind_rows` *Efficiently bind multiple data frames by row and column*

Description

This is an efficient implementation of the common pattern of ‘do.call(rbind, dfs)‘ or ‘do.call(cbind, dfs)‘ for binding many data frames into one.

This is an efficient implementation of the common pattern of ‘do.call(rbind, dfs)‘ or ‘do.call(cbind, dfs)‘ for binding many data frames into one.

Usage

```
bind_rows(..., .id = NULL, add.cell.ids = NULL)

bind_cols(..., .id = NULL)
```

Arguments

`...`	Data frames to combine. Each argument can either be a data frame, a list that could be a data frame, or a list of data frames. When row-binding, columns are matched by name, and any missing columns will be filled with NA. When column-binding, rows are matched by position, so all data frames must have the same number of rows. To match by value, not position, see mutate-joins.
`.id`	Data frame identifier. When ‘.id‘ is supplied, a new column of identifiers is created to link each row to its original data frame. The labels are taken from the named arguments to ‘bind_rows()‘. When a list of data frames is supplied, the labels are taken from the names of the list. If no names are found a numeric sequence is used instead.
`add.cell.ids`	from Seurat 3.0 A character vector of length(x = c(x, y)). Appends the corresponding values to the start of each objects’ cell names.

Details

The output of ‘bind_rows()‘ will contain a column if that column appears in any of the inputs.

The output of ‘bind_rows()‘ will contain a column if that column appears in any of the inputs.

Value

‘bind_rows()‘ and ‘bind_cols()‘ return the same type as the first input, either a data frame, ‘tbl_df‘, or ‘grouped_df‘.

‘bind_rows()‘ and ‘bind_cols()‘ return the same type as the first input, either a data frame, ‘tbl_df‘, or ‘grouped_df‘.

Examples

```
print("small_pbmc |> bind_rows(small_pbmc)")



print("small_pbmc |> bind_cols(annotation_column)")
```

`join_features` *join_features*

Description

join_features() extracts and joins information for specific features

Usage

```
join_features(
  .data,
  features = NULL,
  all = FALSE,
  exclude_zeros = FALSE,
  shape = "long",
  ...
)
```

Arguments

`.data`	A tidy SingleCellExperiment object
`features`	A vector of feature identifiers to join
`all`	If TRUE return all
`exclude_zeros`	If TRUE exclude zero values
`shape`	Format of the returned table "long" or "wide"
`...`	Parameters to pass to join wide, i.e. assay name to extract feature abundance from and gene prefix, for shape="wide"

Details

This function extracts information for specified features and returns the information in either long or wide format.

Value

A ‘tbl‘ containing the information.for the specified features

Examples

```
print("this is a method generics Example is not applicable")
# <object> |> join_features(features=c("HLA-DRA", "LYZ"))
```

`plot_ly` *Initiate a plotly visualization*

Description

This function maps R objects to plotly.js, an (MIT licensed) web-based interactive charting library. It provides abstractions for doing common things (e.g. mapping data values to fill colors (via `color`) or creating animations (via `frame`)) and sets some different defaults to make the interface feel more 'R-like' (i.e., closer to `plot()` and `ggplot2::qplot()`).

Usage

```
plot_ly(
  data = data.frame(),
  ...,
  type = NULL,
  name = NULL,
  color = NULL,
  colors = NULL,
  alpha = NULL,
  stroke = NULL,
  strokes = NULL,
  alpha_stroke = 1,
  size = NULL,
  sizes = c(10, 100),
  span = NULL,
  spans = c(1, 20),
  symbol = NULL,
  symbols = NULL,
  linetype = NULL,
  linetypes = NULL,
  split = NULL,
  frame = NULL,
  width = NULL,
  height = NULL,
  source = "A"
)
```

Arguments

`data`	A data frame (optional) or crosstalk::SharedData object.
`...`	Arguments (i.e., attributes) passed along to the trace type. See `schema()` for a list of acceptable attributes for a given trace type (by going to `traces` -> `type` -> `attributes`). Note that attributes provided at this level may override other arguments (e.g. `plot_ly(x = 1:10, y = 1:10, color = I("red"), marker = list(color = "blue"))`).

`type`	A character string specifying the trace type (e.g. `"scatter"`, `"bar"`, `"box"`, etc). If specified, it *always* creates a trace, otherwise
`name`	Values mapped to the trace's name attribute. Since a trace can only have one name, this argument acts very much like `split` in that it creates one trace for every unique value.
`color`	Values mapped to relevant 'fill-color' attribute(s) (e.g. fillcolor, marker.color, textfont.color, etc.). The mapping from data values to color codes may be controlled using `colors` and `alpha`, or avoided altogether via `I()` (e.g., `color = I("red")`). Any color understood by `grDevices::col2rgb()` may be used in this way.
`colors`	Either a colorbrewer2.org palette name (e.g. "YlOrRd" or "Blues"), or a vector of colors to interpolate in hexadecimal "#RRGGBB" format, or a color interpolation function like `colorRamp()`.
`alpha`	A number between 0 and 1 specifying the alpha channel applied to `color`. Defaults to 0.5 when mapping to fillcolor and 1 otherwise.
`stroke`	Similar to `color`, but values are mapped to relevant 'stroke-color' attribute(s) (e.g., marker.line.color and line.color for filled polygons). If not specified, `stroke` inherits from `color`.
`strokes`	Similar to `colors`, but controls the `stroke` mapping.
`alpha_stroke`	Similar to `alpha`, but applied to `stroke`.
`size`	(Numeric) values mapped to relevant 'fill-size' attribute(s) (e.g., marker.size, textfont.size, and error_x.width). The mapping from data values to symbols may be controlled using `sizes`, or avoided altogether via `I()` (e.g., `size = I(30)`).
`sizes`	A numeric vector of length 2 used to scale `size` to pixels.
`span`	(Numeric) values mapped to relevant 'stroke-size' attribute(s) (e.g., marker.line.width, line.width for filled polygons, and error_x.thickness) The mapping from data values to symbols may be controlled using `spans`, or avoided altogether via `I()` (e.g., `span = I(30)`).
`spans`	A numeric vector of length 2 used to scale `span` to pixels.
`symbol`	(Discrete) values mapped to marker.symbol. The mapping from data values to symbols may be controlled using `symbols`, or avoided altogether via `I()` (e.g., `symbol = I("pentagon")`). Any pch value or symbol name may be used in this way.
`symbols`	A character vector of pch values or symbol names.
`linetype`	(Discrete) values mapped to line.dash. The mapping from data values to symbols may be controlled using `linetypes`, or avoided altogether via `I()` (e.g., `linetype = I("dash")`). Any `lty` (see par) value or dash name may be used in this way.
`linetypes`	A character vector of `lty` values or dash names
`split`	(Discrete) values used to create multiple traces (one trace per value).
`frame`	(Discrete) values used to create animation frames.
`width`	Width in pixels (optional, defaults to automatic sizing).
`height`	Height in pixels (optional, defaults to automatic sizing).

`source`	a character string of length 1. Match the value of this string with the source argument in `event_data()` to retrieve the event data corresponding to a specific plot (shiny apps can have multiple plots).

Details

Unless `type` is specified, this function just initiates a plotly object with 'global' attributes that are passed onto downstream uses of `add_trace()` (or similar). A formula must always be used when referencing column name(s) in `data` (e.g. `plot_ly(mtcars, x = ~wt)`). Formulas are optional when supplying values directly, but they do help inform default axis/scale titles (e.g., `plot_ly(x = mtcars$wt)` vs `plot_ly(x = ~mtcars$wt)`)

Author(s)

Carson Sievert

References

https://plotly-r.com/overview.html

See Also

- For initializing a plotly-geo object: `plot_geo()`
- For initializing a plotly-mapbox object: `plot_mapbox()`
- For translating a ggplot2 object to a plotly object: `ggplotly()`
- For modifying any plotly object: `layout()`, `add_trace()`, `style()`
- For linked brushing: `highlight()`
- For arranging multiple plots: `subplot()`, `crosstalk::bscols()`
- For inspecting plotly objects: `plotly_json()`
- For quick, accurate, and searchable plotly.js reference: `schema()`

Examples

```
## Not run:

# plot_ly() tries to create a sensible plot based on the information you
# give it. If you don't provide a trace type, plot_ly() will infer one.
plot_ly(economics, x = ~pop)
plot_ly(economics, x = ~date, y = ~pop)
# plot_ly() doesn't require data frame(s), which allows one to take
# advantage of trace type(s) designed specifically for numeric matrices
plot_ly(z = ~volcano)
plot_ly(z = ~volcano, type = "surface")

# plotly has a functional interface: every plotly function takes a plotly
# object as it's first input argument and returns a modified plotly object
add_lines(plot_ly(economics, x = ~date, y = ~unemploy/pop))

# To make code more readable, plotly imports the pipe operator from magrittr
```

```
economics %>% plot_ly(x = ~date, y = ~unemploy/pop) %>% add_lines()

# Attributes defined via plot_ly() set 'global' attributes that
# are carried onto subsequent traces, but those may be over-written
plot_ly(economics, x = ~date, color = I("black")) %>%
 add_lines(y = ~uempmed) %>%
 add_lines(y = ~psavert, color = I("red"))

# Attributes are documented in the figure reference -> https://plotly.com/r/reference
# You might notice plot_ly() has named arguments that aren't in this figure
# reference. These arguments make it easier to map abstract data values to
# visual attributes.
p <- plot_ly(palmerpenguins::penguins, x = ~bill_length_mm, y = ~body_mass_g)
add_markers(p, color = ~bill_depth_mm, size = ~bill_depth_mm)
add_markers(p, color = ~species)
add_markers(p, color = ~species, colors = "Set1")
add_markers(p, symbol = ~species)
add_paths(p, linetype = ~species)


## End(Not run)
```

Index