Package: dittoViz (via r-universe)

August 29, 2024

Type Package

Title User Friendly Data Visualization

Version 1.0.2

Description A comprehensive visualization toolkit built with coders of all skill levels and color-vision impaired audiences in mind. It allows creation of finely-tuned, publication-quality figures from single function calls. Visualizations include scatter plots, compositional bar plots, violin, box, and ridge plots, and more. Customization ranges from size and title adjustments to discrete-group circling and labeling, hidden data overlay upon cursor hovering via ggplotly() conversion, and many more, all with simple, discrete inputs. Color blindness friendliness is powered by legend adjustments (enlarged keys), and by allowing the use of shapes or letter-overlay in addition to the carefully selected dittoColors().

License MIT + file LICENCE

Encoding UTF-8

Depends ggplot2

Imports cowplot, ggrepel, ggridges, stats

Suggests plotly, testthat (>= 3.0.0), ggplot.multistats, palmerpenguins, ggrastr (>= 0.2.0)

RoxygenNote 7.3.1

Config/testthat/edition 3

URL https://github.com/dtm2451/dittoViz

BugReports https://github.com/dtm2451/dittoViz/issues

Repository https://dtm2451.r-universe.dev

RemoteUrl https://github.com/dtm2451/dittoviz

RemoteRef HEAD

RemoteSha 9966a0cbe803d03564a3bc1aeb826c613d27df89

Contents

`barPlot` *Outputs a stacked bar plot to show the percent composition of samples, groups, clusters, or other groupings*

Description

Outputs a stacked bar plot to show the percent composition of samples, groups, clusters, or other groupings

Usage

```
barPlot(
  data_frame,
  var,
  group.by,
  scale = c("percent", "count"),
  split.by = NULL,
  rows.use = NULL,
  retain.factor.levels = TRUE,
  data.out = FALSE,
  data.only = FALSE,
  do.hover = FALSE,
  hover.round.digits = 5,
  color.panel = dittoColors(),
  colors = seq_along(color.panel),
  split.nrow = NULL,
  split.ncol = NULL,
  split.adjust = list(),
  y.breaks = NA,
  min = 0,
  max = NA,
  var.labels.rename = NULL,
  var.labels.reorder = NULL,
  x.labels = NULL,
  x.labels.rotate = TRUE,
```

```
  x.reorder = NULL,
  theme = theme_classic(),
  xlab = group.by,
  ylab = "make",
  main = "make",
  sub = NULL,
  legend.show = TRUE,
  legend.title = NULL
)
```

Arguments

data_frame	A data_frame where columns are features and rows are observations you might wish to visualize.
var	Single string representing the name of a column of data_frame to quantify within x-axis groups.
group.by	Single string representing the name of a column of data_frame to use for separating data across discrete x-axis groups.
scale	"count" or "percent". Sets whether data should be shown as counts versus percentage.
split.by	1 or 2 strings denoting the name(s) of column(s) of data_frame containing discrete data to use for faceting / separating data points into separate plots. When 2 columns are named, c(row,col), the first is used as rows and the second is used for columns of the resulting facet grid. When 1 column is named, shape control can be achieved with split.nrow and split.ncol
rows.use	String vector of rownames of data_frame OR an integer vector specifying the row-indices of data points which should be plotted. Alternatively, a Logical vector, the same length as the number of rows in data_frame, where TRUE values indicate which rows to plot.
retain.factor.levels	Logical which controls whether factor identities of var and group.by data should be respected. Set to TRUE to faithfully reflect ordering of groupings encoded in factor levels, but Note that this will also force retention of groupings that could otherwise be removed via rows.use.
data.out	Logical. When set to TRUE, changes the output, from the plot alone, to a list containing the plot ("p") and a data.frame ("data") containing the underlying data.
data.only	Logical. When set to TRUE, the underlying data will be returned, but not the plot itself.
do.hover	Logical which sets whether the ggplot output should be converted to a ggplotly object with data about individual bars displayed when you hover your cursor over them.
hover.round.digits	Integer number specifying the number of decimal digits to round displayed numeric values to, when do.hover is set to TRUE.

`color.panel` String vector which sets the colors to draw from for data representation fills. Default = `dittoColors()`.

A named vector can be used if names are matched to the distinct values of the `color.by` data.

`colors` Integer vector, the indexes / order, of colors from `color.panel` to actually use.

Useful for quickly swapping around colors of the default set (when not using names for color matching).

`split.nrow`, `split.ncol`

Integers which set the dimensions of faceting/splitting when faceting by a single feature.

`split.adjust` A named list which allows extra parameters to be pushed through to the faceting function call. List elements should be valid inputs to the faceting functions, e.g. 'list(scales = "free")'.

For options, when giving 1 column to `split.by`, see `facet_wrap`, OR when giving 2 columns to `split.by`, see `facet_grid`.

`y.breaks` Numeric vector which sets the plot's tick marks / major gridlines. c(break1,break2,break3,etc.)

`min`, `max` Scalars which control the zoom of the plot. These inputs set the minimum / maximum values of the y-axis. Default = set based on the limits of the data, 0 to 1 for `scale = "percent"`, or 0 to maximum count for 0 to 1 for `scale = "count"`.

`var.labels.rename`

String vector for renaming the distinct identities of `var`-values. This vector must be the same length as the number of levels or unique values in the `var`-data.

Hint: use `colLevels` or `unique(data_frame[,var])` to original values.

`var.labels.reorder`

Integer vector. A sequence of numbers, from 1 to the number of distinct `var`-value identities, for rearranging the order labels' groupings within the plot space.

Method: Make a first plot without this input. Then, treating the top-most grouping as index 1, and the bottom-most as index n. Values of `var.labels.reorder` should be these indices, but in the order that you would like them rearranged to be.

`x.labels` String vector which will replace the x-axis groupings' labels. Regardless of `x.reorder`, the first component of `x.labels` sets the name for the left-most x-axis grouping.

`x.labels.rotate`

Logical which sets whether the x-axis grouping labels should be rotated.

`x.reorder` Integer vector. A sequence of numbers, from 1 to the number of groupings, for rearranging the order of x-axis groupings.

Method: Make a first plot without this input. Then, treating the leftmost grouping as index 1, and the rightmost as index n. Values of `x.reorder` should be these indices, but in the order that you would like them rearranged to be.

Recommendation for advanced users: If you find yourself coming back to this input too many times, an alternative solution that can be easier long-term is to make the target data into a factor, and to put its levels in the desired order: `factor(data, levels = c("level1", "level2", ...))`.

theme	A ggplot theme which will be applied before dittoViz adjustments. Default = theme_classic(). See https://ggplot2.tidyverse.org/reference/ggtheme.html for other options and ideas.
xlab	String which sets the x-axis title. Default is group.by so it defaults to the name of the grouping information. Set to NULL to remove.
ylab	String which sets the y-axis title. Default = "make" and if left as make, a title will be automatically generated.
main	String, sets the plot title
sub	String, sets the plot subtitle
legend.show	Logical. Whether the legend should be displayed. Default = TRUE.
legend.title	String which adds a title to the legend.

Details

The function creates a dataframe containing counts and percent makeup of var identities for each x-axis grouping (determined by the group.by input). If a subset of data points to use is indicated with the rows.use input, only those rows of the data_frame are used for counts and percent makeup calculations. In other words, the row.use input adjusts the universe that compositions are calculated within. Then, a vertical bar plot is generated (ggplot2::geom_col()) showing either percent makeup if scale = "percent", which is the default, or raw counts if scale = "count".

Value

A ggplot plot where discrete data, grouped by sample, condition, cluster, etc. on the x-axis, is shown on the y-axis as either counts or percent-of-total-per-grouping in a stacked barplot.

Alternatively, if data.out = TRUE, a list containing the plot ("p") and a dataframe of the underlying data ("data").

Alternatively, if do.hover = TRUE, a plotly conversion of the ggplot output in which underlying data can be retrieved upon hovering the cursor over the plot.

Many characteristics of the plot can be adjusted using discrete inputs

- Colors can be adjusted with color.panel and/or colors.
- y-axis zoom and tick marks can be adjusted using min, max, and y.breaks.
- Titles can be adjusted with main, sub, xlab, ylab, and legend.title arguments.
- The legend can be removed by setting legend.show = FALSE.
- x-axis labels and groupings can be changed / reordered using x.labels and x.reorder, and rotation of these labels can be turned off with x.labels.rotate = FALSE.
- y-axis var-group labels and their order can be changed / reordered using var.labels and var.labels.reorder.

Author(s)

Daniel Bunis

Examples

```
example("dittoExampleData", echo = FALSE)

# There are two main inputs for this function, in addition to 'data_frame'.
#  var = typically this will be observation-type annotations or clustering
#    This is the set of observations for which we will calculate frequencies
#    (per each unique value of this data) within each group
#  group.by = how to group observations together
barPlot(
    data_frame = example_df,
    var = "clustering",
    group.by = "groups")

# 'scale' then allows choice of scaling by 'percent' (default) or 'count'
barPlot(example_df, "clustering", group.by = "groups",
    scale = "count")

# Particular observations can be ignored from calculations and plotting using
#   the 'rows.use' input.
#   Here, we'll remove an entire "cluster" from consideration, but notice the
#     fractions will still sum to 1.
barPlot(example_df, "clustering", group.by = "groups",
    rows.use = example_df$clustering!="1")

### Accessing underlying data:
# as data.frame, with plot returned too
barPlot(example_df, "clustering", group.by = "groups",
    data.out = TRUE)
# as data.frame, no plot
barPlot(example_df, "clustering", group.by = "groups",
    data.out = TRUE,
    data.only = TRUE)
# through hovering the cursor over the relevant parts of the plot
if (requireNamespace("plotly", quietly = TRUE)) {
    barPlot(example_df, "clustering", group.by = "groups",
        do.hover = TRUE)
    }
```

---

colLevels *Gives the distinct values of a column of data from the data_frame*

Description

Gives the distinct values of a column of data from the data_frame

Usage

```
colLevels(col, data_frame, rows.use = NULL, used.only = TRUE)
```

Arguments

col	quoted column name. the data column whose potential values should be retrieved.
data_frame	A data.frame.
rows.use	String vector of rows names OR an integer vector specifying the indices of rows which should be included. Alternatively, a Logical vector, the same length as the number of rows in the data_frame, which indicates which rows to include.
used.only	TRUE by default, for target data that are factors, whether levels nonexistent in the target data should be ignored.

Value

String vector, the distinct values of the col data column (among the rows.use targeted rows) of data_frame.

Author(s)

Daniel Bunis

Examples

```
example("dittoExampleData", echo = FALSE)

colLevels("conditions", example_df)

# Note: Set 'used.only' (default = TRUE) to FALSE to show unused levels
#  of data that are already factors.  By default, only the used options
#  of the data will be given.
colLevels("conditions", example_df,
    rows.use = example_df$conditions!="condition1"
    )
colLevels("conditions", example_df,
    rows.use = example_df$conditions!="condition1",
    used.only = FALSE)
```

---

dittoColors *Extracts the dittoViz default colors*

Description

Creates a string vector of 40 unique colors, in hexadecimal form, repeated 100 times. Or, if get.names is set to TRUE, outputs the names of the colors which can be helpful as reference when adjusting how colors get used.

These colors are a modification of the protanope and deuteranope friendly colors from Wong, B. Nature Methods, 2011.

Truly, only the first 1-7 are maximally (red-green) color-blindness friendly, but the lightened and darkened versions (plus grey) in slots 8-40 still work releatively well at extending their utility further. Note that past 40, the colors simply repeat in order to most easily allow dittoViz visualizations to handle situations requiring even more colors.

The colors are:

1-7 = Suggested color panel from Wong, B. Nature Methods, 2011, minus black

- 1- orange = "#E69F00"
- 2- skyBlue = "#56B4E9"
- 3- bluishGreen = "#009E73"
- 4- yellow = "#F0E442"
- 5- blue = "#0072B2"
- 6- vermillion = "#D55E00"
- 7- reddishPurple = "#CC79A7"

8 = gray40

9-16 = 25% darker versions of colors 1-8

17-24 = 25% lighter versions of colors 1-8

25-32 = 40% lighter versions of colors 1-8

33-40 = 40% darker versions of colors 1-8

Usage

```
dittoColors(reps = 100, get.names = FALSE)
```

Arguments

`reps`	Integer which sets how many times the original set of colors should be repeated
`get.names`	Logical, whether only the names of the default dittoViz color panel should be returned instead

Value

A string vector with length = 24.

Author(s)

Daniel Bunis

Examples

```
dittoColors()

#To retrieve names:
dittoColors(get.names = TRUE)
```

dittoExampleData *Example Data Generation*

Description

Example Data Generation

Details

This documentation point exists only to be a set source of example data for other dittoViz documentation. Running the examples section code creates a data.frame called 'example_df' containing data of various types. These data are randomly generated each time and simulate what a user might use as the 'data_frame' input of dittoViz visualization functions.

Value

Running `example("dittoExampleData")` creates a data.frame called example_df.

Author(s)

Daniel Bunis

Examples

```
# Generate some random data
nobs <- 120

# Fake "PCA" that we'll based some other attributes on
example_pca <- matrix(rnorm(nobs*2), nobs)

example_df <- data.frame(
        conditions = factor(rep(c("condition1", "condition2"), each=nobs/2)),
        timepoint = rep(c("d0", "d3", "d6", "d9"), each = nobs/4),
        SNP = rep(c(rep(TRUE,7),rep(FALSE,8)), nobs/15),
        groups = sample(c("A","B","C","D"), nobs, TRUE),
        score = seq_len(nobs)/2,
        gene1 = log2(rpois(nobs, 5) +1),
        gene2 = log2(rpois(nobs, 30) +1),
        gene3 = log2(rpois(nobs, 4) +1),
        gene4 = log2(rpois(nobs, 2) +1),
        gene5 = log2(rpois(nobs, 17) +1),
        PC1 = example_pca[,1],
        PC2 = example_pca[,2],
        clustering = as.character(1*(example_pca[,1]>0&example_pca[,2]>0) +
                       2*(example_pca[,1]<0&example_pca[,2]>0) +
                       3*(example_pca[,1]>0&example_pca[,2]<0) +
                       4*(example_pca[,1]<0&example_pca[,2]<0)),
        sample = rep(1:12, each = nobs/12),
        category = rep(c("A", "B"), each = nobs/2),
```

```
        subcategory = rep(as.character(rep(1:3,4)), each = nobs/12),
        row.names = paste0("obs", 1:nobs)
        )

# cleanup
rm(example_pca, nobs)

summary(example_df)
```

---

freqPlot *Plot discrete observation frequencies per sample and per grouping*

Description

Plot discrete observation frequencies per sample and per grouping

Usage

```
freqPlot(
  data_frame,
  var,
  sample.by = NULL,
  group.by,
  color.by = group.by,
  vars.use = NULL,
  scale = c("percent", "count"),
  max.normalize = FALSE,
  plots = c("boxplot", "jitter"),
  split.nrow = NULL,
  split.ncol = NULL,
  split.adjust = list(),
  rows.use = NULL,
  data.out = FALSE,
  data.only = FALSE,
  do.hover = FALSE,
  hover.round.digits = 5,
  color.panel = dittoColors(),
  colors = seq_along(color.panel),
  y.breaks = NULL,
  min = 0,
  max = NA,
  var.labels.rename = NULL,
  var.labels.reorder = NULL,
  x.labels = NULL,
  x.labels.rotate = TRUE,
  x.reorder = NULL,
  theme = theme_classic(),
  xlab = group.by,
```

```
  ylab = "make",
  main = "make",
  sub = NULL,
  jitter.size = 1,
  jitter.width = 0.2,
  jitter.color = "black",
  jitter.position.dodge = boxplot.position.dodge,
  do.raster = FALSE,
  raster.dpi = 300,
  boxplot.width = 0.4,
  boxplot.color = "black",
  boxplot.show.outliers = NA,
  boxplot.outlier.size = 1.5,
  boxplot.fill = TRUE,
  boxplot.position.dodge = vlnplot.width,
  boxplot.lineweight = 1,
  vlnplot.lineweight = 1,
  vlnplot.width = 1,
  vlnplot.scaling = "area",
  vlnplot.quantiles = NULL,
  ridgeplot.lineweight = 1,
  ridgeplot.scale = 1.25,
  ridgeplot.ymax.expansion = NA,
  ridgeplot.shape = c("smooth", "hist"),
  ridgeplot.bins = 30,
  ridgeplot.binwidth = NULL,
  add.line = NULL,
  line.linetype = "dashed",
  line.color = "black",
  legend.show = TRUE,
  legend.title = color.by
)
```

Arguments

`data_frame`	A data_frame where columns are features and rows are observations you might wish to visualize.
`var`	Single string representing the name of a column of `data_frame` that contains the discrete data you wish to quantify as frequencies.
`sample.by`	Single string representing the name of a column of `data_frame` that contains an indicator of which sample each observation belongs to. Note that when this is not provided, there will only be one data point per grouping. A warning can be expected then for all `plots` options except `"jitter"`.
`group.by`	Single string representing the name of a column of `data_frame` containing discrete data to use for separating the data points into groups.
`color.by`	Single string representing the name of a column of `data_frame` containing discrete data to use for setting data representation color fills. This data does not

need to be the same as group.by, which is great for highlighting supersets or subgroups when wanted, but it defaults to group.by so the input can often be skipped.

vars.use — String or string vector naming a subset of the values of var-data which should be shown. If left as NULL, all values are shown.

Hint: use colLevels or unique(data_frame[,var]) to assess options.

Note: When var.labels.rename is jointly utilized to update how the var-values are shown, the **updated** values must be used.

scale — "count" or "percent". Sets whether data should be shown as counts versus percentage.

max.normalize — Logical which sets whether the data for each var-data value (each facet) should be normalized to have the same maximum value.

When set to TRUE, lower frequency var-values will make use of just as much plot space as higher frequency vars.

Note: Similarly equal plot space utilization can be achieved by using split.adjust = list(scales = "free_y"), and that alternative route retains original values of the data.

plots — String vector which sets the types of plots to include: possibilities = "jitter", "boxplot", "vlnplot", "ridgeplot".

Order matters: c("vlnplot", "boxplot", "jitter") will put a violin plot in the back, boxplot in the middle, and then individual dots in the front.

See details section for more info.

split.nrow, split.ncol

Integers which set the dimensions of the facet grid.

split.adjust — A named list which allows extra parameters to be pushed through to the faceting function call. List elements should be valid inputs to the faceting function facet_wrap, e.g. 'list(scales = "free_y")'.

See facet_wrap for options.

rows.use — String vector of rownames of data_frame OR an integer vector specifying the row-indices of data points which should be plotted.

Alternatively, a Logical vector, the same length as the number of rows in data_frame, where TRUE values indicate which rows to plot.

data.out — Logical. When set to TRUE, changes the output, from the plot alone, to a list containing the plot (p), its underlying data (data).

data.only — Logical. When set to TRUE, the underlying data will be returned, but not the plot itself.

do.hover — Logical which sets whether the ggplot output should be converted to a ggplotly object with data about individual bars displayed when you hover your cursor over them.

hover.round.digits

Integer number specifying the number of decimal digits to round displayed numeric values to, when do.hover is set to TRUE.

color.panel — String vector which sets the colors to draw from for data representation fills. Default = dittoColors().

A named vector can be used if names are matched to the distinct values of the color.by data.

`colors` Integer vector, the indexes / order, of colors from `color.panel` to actually use.

Useful for quickly swapping around colors of the default set (when not using names for color matching).

`y.breaks` Numeric vector, a set of breaks that should be used as major grid lines. c(break1,break2,break3,etc.).

`min, max` Scalars which control the zoom on the continuous axis of the plot.

`var.labels.rename`

String vector for renaming the distinct identities of `var`-values. This vector must be the same length as the number of levels or unique values in the `var`-data.

Hint: use `colLevels` or `unique(data_frame[,var])` to original values.

`var.labels.reorder`

Integer vector. A sequence of numbers, from 1 to the number of distinct `var`-value identities, for rearranging the order of facets within the plot space.

Method: Make a first plot without this input. Then, treating the top-left-most grouping as index 1, and the bottom-right-most as index n. Values of `var.labels.reorder` should be these indices, but in the order that you would like them rearranged to be.

`x.labels` String vector, c("label1","label2","label3",...) which overrides the names of groupings.

`x.labels.rotate`

Logical which sets whether the labels should be rotated. Default: `TRUE` for violin and box plots, but `FALSE` for ridgeplots.

`x.reorder` Integer vector. A sequence of numbers, from 1 to the number of groupings, for rearranging the order of x-axis groupings.

Method: Make a first plot without this input. Then, treating the leftmost grouping as index 1, and the rightmost as index n. Values of x.reorder should be these indices, but in the order that you would like them rearranged to be.

Recommendation for advanced users: If you find yourself coming back to this input too many times, an alternative solution that can be easier long-term is to make the target data into a factor, and to put its levels in the desired order: `factor(data, levels = c("level1", "level2", ...))`.

`theme` A ggplot theme which will be applied before internal adjustments. Default = `theme_classic()`. See https://ggplot2.tidyverse.org/reference/ggtheme.html for other options and ideas.

`xlab` String which sets the grouping-axis label (=x-axis for box and violin plots, y-axis for ridgeplots). Set to `NULL` to remove.

`ylab` String, sets the continuous-axis label (=y-axis for box and violin plots, x-axis for ridgeplots). Default = "make" and if left as make, this title will be automatically generated.

`main` String, sets the plot title. Default = "make" and if left as make, a title will be automatically generated. To remove, set to `NULL`.

`sub` String, sets the plot subtitle.

`jitter.size` Scalar which sets the size of the jitter shapes.

`jitter.width` Scalar that sets the width/spread of the jitter in the x direction. Ignored in ridgeplots.

Note for when `color.by` is used to split x-axis groupings into additional bins: ggplot does not shrink jitter widths accordingly, so be sure to do so yourself! Ideally, needs to be 0.5/num_subgroups.

`jitter.color` String which sets the color of the jitter shapes

`jitter.position.dodge`
Scalar which adjusts the relative distance between jitter widths when multiple subgroups exist per `group.by` grouping (a.k.a. when `group.by` and `color.by` are not equal). Similar to `boxplot.position.dodge` input & defaults to the value of that input so that BOTH will actually be adjusted when only, say, `boxplot.position.dodge = 0.3` is given.

`do.raster` Logical. When set to `TRUE`, rasterizes the jitter plot layer, changing it from individually encoded points to a flattened set of pixels. This can be useful for editing in external programs (e.g. Illustrator) when there are many thousands of data points.

`raster.dpi` Number indicating dots/pixels per inch (dpi) to use for rasterization. Default = 300.

`boxplot.width` Scalar which sets the width/spread of the boxplot in the x direction

`boxplot.color` String which sets the color of the lines of the boxplot

`boxplot.show.outliers`
Logical, whether outliers should by including in the boxplot. Default is `FALSE` when there is a jitter plotted, `TRUE` if there is no jitter.

`boxplot.outlier.size`
Scalar which adjusts the size of points used to mark outliers.

`boxplot.fill` Logical, whether the boxplot should be filled in or not. Known bug: when boxplot fill is turned off, outliers do not render.

`boxplot.position.dodge`
Scalar which adjusts the relative distance between boxplots when multiple are drawn per grouping (a.k.a. when `group.by` and `color.by` are not equal). By default, this input actually controls the value of `jitter.position.dodge` unless the `jitter` version is provided separately.

`boxplot.lineweight`
Scalar which adjusts the thickness of boxplot lines.

`vlnplot.lineweight`
Scalar which sets the thickness of the line that outlines the violin plots.

`vlnplot.width` Scalar which sets the width/spread of violin plots in the x direction

`vlnplot.scaling`
String which sets how the widths of the of violin plots are set in relation to each other. Options are "area", "count", and "width". If the default is not right for your data, I recommend trying "width". For an explanation of each, see `geom_violin`.

`vlnplot.quantiles`
Single number or numeric vector of values in [0,1] naming quantiles at which to draw a horizontal line within each violin plot. Example: `c(0.1, 0.5, 0.9)`

`ridgeplot.lineweight`
: Scalar which sets the thickness of the ridgeplot outline.

`ridgeplot.scale`
: Scalar which sets the distance/overlap between ridgeplots. A value of 1 means the tallest density curve just touches the baseline of the next higher one. Higher numbers lead to greater overlap. Default = 1.25

`ridgeplot.ymax.expansion`
: Scalar which adjusts the minimal space between the topmost grouping and the top of the plot in order to ensure the curve is not cut off by the plotting grid. The larger the value, the greater the space requested. When left as NA, dittoViz will attempt to determine an ideal value itself based on the number of groups & linear interpolation between these goal posts: #groups of 3 or fewer: 0.6; #groups=12: 0.1; #groups or 34 or greater: 0.05.

`ridgeplot.shape`
: Either "smooth" or "hist", sets whether ridges will be smoothed (the typical, and default) versus rectangular like a histogram. (Note: as of the time shape "hist" was added, combination of jittered points is not supported by the `stat_binline` that dittoViz relies on.)

`ridgeplot.bins`
: Integer which sets how many chunks to break the x-axis into when `ridgeplot.shape = "hist"`. Overridden by `ridgeplot.binwidth` when that input is provided.

`ridgeplot.binwidth`
: Integer which sets the width of chunks to break the x-axis into when `ridgeplot.shape = "hist"`. Takes precedence over `ridgeplot.bins` when provided.

`add.line`
: numeric value(s) where one or multiple line(s) should be added

`line.linetype`
: String which sets the type of line for `add.line`. Defaults to "dashed", but any ggplot linetype will work.

`line.color`
: String that sets the color(s) of the `add.line` line(s)

`legend.show`
: Logical. Whether the legend should be displayed. Default = `TRUE`.

`legend.title`
: String or `NULL`, sets the title for the main legend which includes colors and data representations.

Details

The function creates a dataframe containing counts and percent makeup of `var` identities per sample if `sample.by` is given, or per group if only `group.by` is given. `color.by` can optionally be used to add subgroupings to calculations and ultimate plots, or to convey super-groups of `group.by` groupings.

Typically, `var` might target clustering or observation-type annotations, but in truth it can be given any discrete data.

If a set of rows to use was indicated with the `rows.use` input, only the targeted rows are used for counts and percent makeup calculations. In other words, the `row.use` input adjusts the universe that frequencies are calculated within.

If a set of `var`-values to show is indicated with the `vars.use` input, the data.frame is trimmed at the end to include only the corresponding rows. Thus, this input does not affect the universe for frequency calculation.

If max.normalized is set to TRUE, counts and percent data are transformed to a 0-1 scale, which is one method for making better use of white space for lower frequency var-values. Alternatively, split.adjust = list(scales = "free_y") can be used to achieve the same white-space utilization while retaining original data values.

Either percent of total (scale = "percent"), which is the default, or counts (if scale = "count") data is then (gg)plotted with the data representation types in plots by utilizing the same machinery as yPlot. Faceting by var-data values is utilized to achieve per var-value (e.g. cluster) granularity.

See below for additional customization options!

Value

A ggplot plot where frequencies of discrete var-data per sample, grouped by condition, timepoint, etc., is shown on the y-axis by a violin plot, boxplot, and/or jittered points, or on the x-axis by a ridgeplot with or without jittered points.

Alternatively, if data.out = TRUE, a list containing the plot ("p") and a dataframe of the underlying data ("data").

Alternatively, if do.hover = TRUE, a plotly conversion of the ggplot output in which underlying data can be retrieved upon hovering the cursor over the plot.

Calculation Details

The function is restricted in that each samples' observations, indicated by the unique values of sample.by-data, must exist within single group.by and color.by groupings. Thus, in order to ensure all valid var-data composition data points are generated, prior to calculations...

- var-data are ensured to be a factor, which ensures a calculation will be run for every var-value (a.k.a. cluster)
- group.by-data and color-by-data are treated as non-factor data, which ensures that calculations are run only for the groupings that each sample is associated with.

Plot Customization

The plots argument determines the types of **data representation** that will be generated, as well as their order from back to front. Options are "jitter", "boxplot", "vlnplot", and "ridgeplot".

Each plot type has specific associated options which are controlled by variables that start with their associated string. For example, all jitter adjustments start with "jitter.", such as jitter.size and jitter.width.

Inclusion of "ridgeplot" overrides "boxplot" and "vlnplot" presence and changes the plot to be horizontal.

Additionally:

- **Colors can be adjusted** with color.panel.
- **Subgroupings:** color.by can be utilized to split major group.by groupings into subgroups. When this is done in y-axis plotting, dittoViz automatically ensures the centers of all geoms will align, but users will need to manually adjust jitter.width to less than 0.5/num_subgroups to avoid overlaps. There are also three inputs through which one can use to control geom-center placement, but the easiest way to do all at once so is to just adjust vlnplot.width! The other two: boxplot.position.dodge, and jitter.position.dodge.

- **Line(s) can be added** at single or multiple value(s) by providing these values to `add.line`. Linetype and color are set with `line.linetype`, which is "dashed" by default, and `line.color`, which is "black" by default.
- **Titles and axes labels** can be adjusted with `main`, `sub`, `xlab`, `ylab`, and `legend.title` arguments.
- The **legend can be hidden** by setting `legend.show = FALSE`.
- **y-axis zoom and tick marks** can be adjusted using `min`, `max`, and `y.breaks`.
- **x-axis labels and groupings** can be changed / reordered using `x.labels` and `x.reorder`, and rotation of these labels can be turned on/off with `x.labels.rotate = TRUE/FALSE`.

Author(s)

Daniel Bunis

See Also

`barPlot` for a data representation that emphasizes total makeup of samples/groups rather than focusing on the `var`-data values individually.

Examples

```
example("dittoExampleData", echo = FALSE)

# There are three main inputs for this function, in addition to 'data_frame'.
#  var = typically this will be observation-type annotations or clustering
#    This is the set of observations for which we will calculate frequencies
#    (per each unique value of this data) within each sample
#  sample.by = the name of a column containing sample assignments
#    We'll treat all observations with the same value in this column as part
#    of the same sample.
#  group.by = how to group samples together
freqPlot(example_df,
    var = "clustering",
    sample.by = "sample",
    group.by = "category")

# 'color.by' can also be set differently from 'group.by' to have the effect
#  of highlighting supersets or subgroupings:
freqPlot(example_df, "clustering",
    group.by = "category",
    sample.by = "sample",
    color.by = "subcategory")

# The var-values shown can be subset with 'vars.use'
freqPlot(example_df, "clustering",
    group.by = "category", sample.by = "sample", color.by = "subcategory",
    vars.use = 1:2)

# Particular observations can be ignored from calculations and plotting using
#   the 'rows.use' input. Note that doing so adjusts the universe in which
```

```
#   frequencies are calculated; all frequencies will now be in terms of freq.
#   out of the rows.use cells.
#   This can be useful for quantifying subtypes within a given supertype,
#     rather than per all observations.
#   For our example, we'll calculate among clusters 1 and 2, treating clusters 3
#     and 4 observations as part of an unwanted other group of data. You'll
#     notice that frequencies are higher here than when we used 'vars.use' in
#     the previous example.
freqPlot(example_df, "clustering",
    group.by = "category", sample.by = "sample", color.by = "subcategory",
    rows.use = example_df$clustering %in% 1:2)

# Lower frequency targets can be expanded to use the entire y-axis by:
#  turning on 'max.normalize'-ation:
freqPlot(example_df, "clustering",
    group.by = "category", sample.by = "sample", color.by = "subcategory",
    max.normalize = TRUE)
#  or by setting y-scale limits to be set by the contents of facets:
freqPlot(example_df, "clustering",
    group.by = "category", sample.by = "sample", color.by = "subcategory",
    split.adjust = list(scales = "free_y"))

# Data representations can also be selected and reordered with the 'plots'
#  input, and further adjusted with inputs applying to each representation.
freqPlot(example_df,
    var = "clustering", sample.by = "sample", group.by = "category",
    plots = c("vlnplot", "boxplot", "jitter"),
    vlnplot.lineweight = 0.2,
    boxplot.fill = FALSE,
    boxplot.lineweight = 0.2)

# Finally, 'sample.by' is not technically required. When not given, a
#  single data point of overall composition stats will be shown for each
#  grouping.
#  Just note, all data representation other than "jitter" will complain
#  due to there only being the one datapoint per group unless you set
#  plots to "jitter".
freqPlot(example_df,
    var = "clustering", group.by = "category", color.by = "subcategory",
    plots = "jitter")
```

`scatterHex` *scatter plot where observations are grouped into hexagonal bins and then summarized*

Description

scatter plot where observations are grouped into hexagonal bins and then summarized

Usage

```
scatterHex(
  data_frame,
  x.by,
  y.by,
  color.by = NULL,
  bins = 30,
  color.method = NULL,
  split.by = NULL,
  rows.use = NULL,
  color.panel = dittoColors(),
  colors = seq_along(color.panel),
  x.adjustment = NULL,
  y.adjustment = NULL,
  color.adjustment = NULL,
  x.adj.fxn = NULL,
  y.adj.fxn = NULL,
  color.adj.fxn = NULL,
  multivar.split.dir = c("col", "row"),
  split.nrow = NULL,
  split.ncol = NULL,
  split.adjust = list(),
  min.density = NA,
  max.density = NA,
  min.color = "#F0E442",
  max.color = "#0072B2",
  min.opacity = 0.2,
  max.opacity = 1,
  min = NA,
  max = NA,
  rename.color.groups = NULL,
  xlab = x.by,
  ylab = y.by,
  main = "make",
  sub = NULL,
  theme = theme_bw(),
  do.contour = FALSE,
  contour.color = "black",
  contour.linetype = 1,
  do.ellipse = FALSE,
  do.label = FALSE,
  labels.size = 5,
  labels.highlight = TRUE,
  labels.use.numbers = FALSE,
  labels.numbers.spacer = ": ",
  labels.repel = TRUE,
  labels.split.by = split.by,
  labels.repel.adjust = list(),
```

```
  add.trajectory.by.groups = NULL,
  add.trajectory.curves = NULL,
  trajectory.group.by,
  trajectory.arrow.size = 0.15,
  add.xline = NULL,
  xline.linetype = "dashed",
  xline.color = "black",
  add.yline = NULL,
  yline.linetype = "dashed",
  yline.color = "black",
  legend.show = TRUE,
  legend.color.title = "make",
  legend.color.breaks = waiver(),
  legend.color.breaks.labels = waiver(),
  legend.density.title = "Observations",
  legend.density.breaks = waiver(),
  legend.density.breaks.labels = waiver(),
  show.grid.lines = TRUE,
  data.out = FALSE
)
```

Arguments

`data_frame`	A data_frame where columns are features and rows are observations you might wish to visualize.
`x.by`, `y.by`	Single strings denoting the name of a column of `data_frame` containing numeric data to use for the x- and y-axis of the scatterplot.
`color.by`	Single string denoting the name of a column of `data_frame` to use, instead of point density, for setting the color of plotted hexagons. Alternatively, a string vector naming multiple such columns of data to plot at once.
`bins`	Numeric or numeric vector giving the number of hexagonal bins in the x and y directions. Set to 30 by default.
`color.method`	Single string that specifies how `color.by` data should be summarized per each hexagonal bin. Options, and the default, depend on whether the `color.by`-data is continuous versus discrete: **Continuous**: String naming a function for how target data should be summarized for each bin. Can be any function that inputs (summarizes) a numeric vector and outputs a single numeric value. Default is `median`. Other useful options are `sum`, `mean`, `sd`, or `max`. You can also use a custom function as long as you give it a name; e.g. first run `logsum <- function(x) { log(sum(x)) }` externally, then give `color.method = "logsum"`. **Discrete**: A string signifying whether the color should (default) be simply based on the "max" grouping of the bin, based on "prop.<value>" the proportion of a specific value (e.g. "prop.A" or "prop.TRUE"), or based on the "max.prop"ortion of observations belonging to any grouping.
`split.by`	1 or 2 strings denoting the name(s) of column(s) of `data_frame` containing discrete data to use for faceting / separating data points into separate plots.

When 2 columns are named, c(row,col), the first is used as rows and the second is used for columns of the resulting facet grid.

When 1 column is named, shape control can be achieved with `split.nrow` and `split.ncol`

`rows.use` String vector of rownames of `data_frame` OR an integer vector specifying the row-indices of data points which should be plotted.

Alternatively, a Logical vector, the same length as the number of rows in `data_frame`, where `TRUE` values indicate which rows to plot.

`color.panel` String vector which sets the colors to draw from when `color.by` indicates discrete data. `dittoColors()` by default, see `dittoColors` for contents.

A named vector can be used if names are matched to the distinct values of the `color.by` data.

`colors` Integer vector, the indexes / order, of colors from `color.panel` to actually use.

Useful for quickly swapping around colors of the default set (when not using names for color matching).

`x.adjustment`, `y.adjustment`, `color.adjustment`

A recognized string indicating whether numeric `x.by`, `y.by`, and `color.by` data should be used directly (default) or should be adjusted to be

- "z-score": scaled with the scale() function to produce a relative-to-mean z-score representation
- "relative.to.max": divided by the maximum value to give percent of max values between [0,1]

Ignored if the target data is not numeric as these known adjustments target numeric data only.

In order to leave the unedited data available for use in other features, the adjusted data are put in a new column and that new column is used for plotting.

`x.adj.fxn`, `y.adj.fxn`, `color.adj.fxn`

If you wish to apply a function to edit the `x.by`, `y.by`, or `color.by` data before use, in a way not possible with the `color.adjustment` input, this input can be given a function which takes in a vector of values as input and returns a vector of values of the same length as output.

For example, `function(x) {log2(x)}` or `as.factor`.

In order to leave the unedited data available for use in other features, the adjusted data are put in a new column and that new column is used for plotting.

`multivar.split.dir`

"row" or "col", sets the direction of faceting used for 'var' values when:

- `var` is given multiple column names
- AND `split.by` is used to provide an additional feature to facet by

`split.nrow`, `split.ncol`

Integers which set the dimensions of faceting/splitting when faceting by a single feature.

`split.adjust` A named list which allows extra parameters to be pushed through to the faceting function call. List elements should be valid inputs to the faceting functions, e.g. 'list(scales = "free")'.

For options, when giving 1 column to `split.by`, see `facet_wrap`, OR when giving 2 columns to `split.by`, see `facet_grid`.

`min.density`, `max.density`
: Number which sets the min/max values used for the density scale. Used no matter whether density is represented through opacity or color.

`min.color`, `max.color`
: color for the min/max values of the color scale.

`min.opacity`, `max.opacity`
: Scalar between [0,1] which sets the minimum or maximum opacity used for the density legend (when color is used for `color.by` data and density is shown via opacity).

`min`, `max`
: Number which sets the values associated with the minimum or maximum color for `color.by` data.

`rename.color.groups`
: String vector which sets new names for the identities of `color.by` groups.

`xlab`, `ylab`
: Strings which set the labels for the axes. To remove, set to `NULL`.

`main`
: String, sets the plot title. The default title is either "Density", `color.by`, or NULL, depending on the identity of `color.by`. To remove, set to `NULL`.

`sub`
: String, sets the plot subtitle.

`theme`
: A ggplot theme which will be applied before internal adjustments. Default = `theme_bw()`. See https://ggplot2.tidyverse.org/reference/ggtheme.html for other options and ideas.

`do.contour`
: Logical. Whether density-based contours should be displayed.

`contour.color`
: String that sets the color of the `do.contour` contours.

`contour.linetype`
: String or numeric which sets the type of line used for `do.contour` contours. Defaults to "solid", but see `linetype` for other options.

`do.ellipse`
: Logical. Whether `color.by` groups should be surrounded by median-centered ellipses.

`do.label`
: Logical. Whether to add text labels near the center (median) of `color.by` groups.

`labels.size`
: Number which sets the size of labels text when `do.label = TRUE`.

`labels.highlight`
: Logical. Whether labels should have a box behind them when `do.label = TRUE`.

`labels.use.numbers`
: Logical which controls whether numbers will be used in place of original data-values. When turned on, so number to value mapping can be known, these numbers are added to the legend.

`labels.numbers.spacer`
: String. When `do.label = TRUE` and `labels.use.numbers`, this string is used in the legend between the numbers and their associated data values.

`labels.repel`
: Logical, that sets whether the labels' placements will be adjusted with ggrepel to avoid intersections between labels and plot bounds when `do.label = TRUE`. TRUE by default.

`labels.split.by`
: String of one or two column names which controls the facet-split calculations for label placements. Defaults to `split.by`, so generally there is no need to adjust this except when if you plan to apply faceting externally.

`labels.repel.adjust`
: A named list which allows extra parameters to be pushed through to ggrepel function calls. List elements should be valid inputs to the `geom_label_repel` by default, or `geom_text_repel` when `labels.highlight = FALSE`.

`add.trajectory.by.groups`
: List of vectors representing trajectory paths, each from start-group to end-group, where vector contents are the group-names indicated by the `trajectory.group.by` column of `data_frame`.

`add.trajectory.curves`
: List of matrices, each representing coordinates for a trajectory path, from start to end, where matrix columns represent x and y coordinates of the paths.

`trajectory.group.by`
: String denoting the name of a column of `data_frame` to use for generating trajectories from data point groups.

`trajectory.arrow.size`
: Number representing the size of trajectory arrows, in inches. Default = 0.15.

`add.xline`
: numeric value(s) where one or multiple vertical line(s) should be added.

`xline.linetype`
: String which sets the type of line for `add.xline`. Defaults to "dashed", but any ggplot linetype will work.

`xline.color`
: String that sets the color(s) of the `add.xline` line(s).

`add.yline`
: numeric value(s) where one or multiple vertical line(s) should be added.

`yline.linetype`
: String which sets the type of line for `add.yline`. Defaults to "dashed", but any ggplot linetype will work.

`yline.color`
: String that sets the color(s) of the `add.yline` line(s).

`legend.show`
: Logical. Whether any legend should be displayed. Default = `TRUE`.

`legend.density.title`, `legend.color.title`
: Strings which set the title for the legends.

`legend.density.breaks`, `legend.color.breaks`
: Numeric vector which sets the discrete values to label in the density and color.by legends.

`legend.density.breaks.labels`, `legend.color.breaks.labels`
: String vector, with same length as `legend.*.breaks`, which sets the labels for the tick marks or hex icons of the associated legend.

`show.grid.lines`
: Logical which sets whether grid lines should be shown within the plot space.

`data.out`
: Logical. When set to `TRUE`, changes the output from the plot alone to a list containing the plot ("plot"), and data.frame of the underlying data for target observations ("data"), and the ultimately used mapping of columns to given aesthetic sets, because modification of newly made columns is required for many features ("cols_used").

Details

This function first makes any requested adjustments to data in the given `data_frame`, internally only, such as scaling the `color.by`-column if `color.adjustment` was given `"z-score"`.

Next, data_frame is then subset to only target rows based on the `rows.use` input.

Finally, a hex plot is created using this dataframe:

If `color.by` is not rovided, coloring is based on the density of observations within each hex bin. When `color.by` is provided, density is represented through opacity while coloring is based on a summarization, chosen with the `color.method` input, of the target `color.by` data.

If `split.by` was used, the plot will be split into a matrix of panels based on the associated groupings.

Value

A ggplot object where colored hexagonal bins are used to summarize observations in a scatter plot.

Alternatively, if `data.out=TRUE`, a list containing three slots is output: the plot (named 'plot'), a data.table containing the updated underlying data for target rows (named 'data'), and a list providing mappings of final column names in 'data' to given plot aesthetics (named 'cols_used'), because modification of newly made columns is required for many features.

Many characteristics of the plot can be adjusted using discrete inputs

- Colors: `min.color` and `max.color` adjust the colors for continuous data.
- For discrete `color.by` plotting with `color.method = "max"`, colors are instead adjusted with `color.panel` and/or `colors` & the labels of the groupings can be changed using `rename.color.groups`.
- Titles and axes labels can be adjusted with `main`, `sub`, `xlab`, `ylab`, and `legend.color.title` and `legend.density.title` arguments.
- Legends can also be adjusted in other ways, using variables that all start with `"legend."` for easy tab completion lookup.

Additional Features

Other tweaks and features can be added as well. Each is accessible through 'tab' autocompletion starting with `"do."---` or `"add."---`, and if additional inputs are involved in implementing or tweaking these, the associated inputs will start with the `"---."`:

- If `do.contour` is provided, density gradient contour lines will be overlaid with color and linetype adjustable via `contour.color` and `contour.linetype`.
- If `add.trajectory.by.groups` is provided a list of vectors (each vector being group names from start-group-name to end-group-name), and a column name pointing to the relevant grouping information is provided to `trajectory.group.by`, then median centers of the groups will be calculated and arrows will be overlayed to show trajectory inference paths.
- If `add.trajectory.curves` is provided a list of matrices (each matrix containing x, y coordinates from start to end), paths and arrows will be overlayed to show trajectory inference curves. Arrow size is controlled with the `trajectory.arrow.size` input.

Author(s)

Daniel Bunis with some code adapted from Giuseppe D'Agostino

See Also

scatterPlot for making non-hex-binned scatter plots showing each individual data point. It is often best to investigate your data with both the individual and hex-bin methods, then pick whichever is the best representation for your particular goal.

Examples

```
example("dittoExampleData", echo = FALSE)

# The minimal inputs for scatterHex are the 'data_frame', and 2 column names,
#   given to 'x.by' and 'y.by', indicating which data to use for the x and y
#   axes, respectively.
scatterHex(
    example_df, x.by = "PC1", y.by = "PC2")

# 'color.by' can also be given a column name in order to represent that
#   column's data in the color of the hexes.
# Note: This capability requires the suggested package 'ggplot.multistats'.
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(
        example_df, x.by = "PC1", y.by = "PC2",
        color.by = "groups")
}
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(
        example_df, x.by = "PC1", y.by = "PC2",
        color.by = "gene1")
}

# 'color.method' is then used to adjust how the target data is summarized
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(example_df, x.by = "PC1", y.by = "PC2",
        color.by = "groups",
        color.method = "max.prop")
}
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(example_df, x.by = "PC1", y.by = "PC2",
        color.by = "gene1",
        color.method = "mean")
}
# One particularly useful 'color.method' for discrete 'color.by'-data is
#   to use 'prop.<value>' to color by the proportion of a particular value
#   within each bin:
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(example_df, x.by = "PC1", y.by = "PC2",
        color.by = "groups",
        color.method = "prop.A")
}

# Data can be "split" or faceted by a discrete variable as well.
scatterHex(example_df, x.by = "PC1", y.by = "PC2",
    split.by = "timepoint") # single split.by element
```

```
scatterHex(example_df, x.by = "PC1", y.by = "PC2",
    split.by = c("groups","SNP")) # row and col split.by elements

# Modify the look with intuitive inputs
scatterHex(example_df, x.by = "PC1", y.by = "PC2",
    show.grid.lines = FALSE,
    ylab = NULL, xlab = "PC2 by PC1",
    main = "Plot Title",
    sub = "subtitle",
    legend.density.title = "Items")
# 'max.density' is one of these intuitively named inputs that can be
#   extremely useful for saying "I only can for opacity to be decreased
#   in regions with exceptionally low observation numbers."
# (A good value for this in "real" data might be 10 or 50 or higher, but for
#   our sparse example data, we need to do a lot to show this off at all!)
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(
        example_df, x.by = "PC1", y.by = "PC2",
        color.by = "gene1", bins = 10,
        sub = "Default density scale")
}
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(
        example_df, x.by = "PC1", y.by = "PC2",
        color.by = "gene1", bins = 10,
        sub = "Density capped low for ignoring sparse regions",
        max.density = 2)
}

# You can restrict to only certain data points using the 'rows.use' input.
#   The input can be given rownames, indexes, or a logical vector
scatterHex(example_df, x.by = "PC1", y.by = "PC2",
    sub = "show only first 40 observations, by index",
    rows.use = 1:40)
scatterHex(example_df, x.by = "PC1", y.by = "PC2",
    sub = "show only 3 obs, by name (plotting gets a bit wonky for few points)",
    rows.use = c("obs1", "obs2", "obs25"))
scatterHex(example_df, x.by = "PC1", y.by = "PC2",
    sub = "show groups A,B,D only, by logical",
    rows.use = example_df$groups!="C")

# Many extra features are easy to add as well:
#   Each is started via an input starting with 'do.FEATURE*' or 'add.FEATURE*'
#   And when tweaks for that feature are possible, those inputs will start be
#   named starting with 'FEATURE*'. For example, color.by groups can be labeled
#   with 'do.label = TRUE' and the tweaks for this feature are given with inputs
#   'labels.size', 'labels.highlight', and 'labels.repel':
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(example_df, x.by = "PC1", y.by = "PC2", color.by = "groups",
        sub = "default labeling",
        do.label = TRUE)          # Turns on the labeling feature
}
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
```

```
    scatterHex(example_df, x.by = "PC1", y.by = "PC2", color.by = "groups",
        sub = "tweaked labeling",
        do.label = TRUE,          # Turns on the labeling feature
        labels.size = 8,          # Adjust the text size of labels
        labels.highlight = FALSE, # Removes white background behind labels
        # labels.use.numbers = TRUE,# Swap to number placeholders
        labels.repel = FALSE)     # Turns off anti-overlap location adjustments
}

# Faceting can also be used to show multiple continuous variables side-by-side
#   by giving a vector of column names to 'color.by'.
#   This can also be combined with 1 'split.by' variable, with direction then
#   controlled via 'multivar.split.dir':
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(example_df, x.by = "PC1", y.by = "PC2", bins = 10,
        color.by = c("gene1", "gene2"))
}
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(example_df, x.by = "PC1", y.by = "PC2", bins = 10,
        color.by = c("gene1", "gene2"),
        split.by = "groups")
}
if (requireNamespace("ggplot.multistats", quietly = TRUE)) {
    scatterHex(example_df, x.by = "PC1", y.by = "PC2", bins = 10,
        color.by = c("gene1", "gene2"),
        split.by = "groups",
        multivar.split.dir = "row")
}

# Sometimes, it can be useful for external editing or troubleshooting purposes
#   to see the underlying data that was directly used for plotting.
# 'data.out = TRUE' can be provided in order to obtain not just plot ("plot"),
#   but also the "data" and "cols_used" returned as a list.
out <- scatterHex(example_df, x.by = "PC1", y.by = "PC2",
    rows.use = 1:40,
    data.out = TRUE)
out$plot
summary(out$data)
out$cols_use
```

---

scatterPlot *Show RNAseq data overlayed on a scatter plot*

Description

Show RNAseq data overlayed on a scatter plot

Usage

```
scatterPlot(
  data_frame,
  x.by,
  y.by,
  color.by = NULL,
  shape.by = NULL,
  split.by = NULL,
  size = 1,
  rows.use = NULL,
  show.others = TRUE,
  x.adjustment = NULL,
  y.adjustment = NULL,
  color.adjustment = NULL,
  x.adj.fxn = NULL,
  y.adj.fxn = NULL,
  color.adj.fxn = NULL,
  split.show.all.others = TRUE,
  opacity = 1,
  color.panel = dittoColors(),
  colors = seq_along(color.panel),
  split.nrow = NULL,
  split.ncol = NULL,
  split.adjust = list(),
  multivar.split.dir = c("col", "row"),
  shape.panel = c(16, 15, 17, 23, 25, 8),
  rename.color.groups = NULL,
  rename.shape.groups = NULL,
  min.color = "#F0E442",
  max.color = "#0072B2",
  min.value = NA,
  max.value = NA,
  plot.order = c("unordered", "increasing", "decreasing", "randomize"),
  xlab = x.by,
  ylab = y.by,
  main = "make",
  sub = NULL,
  theme = theme_bw(),
  do.hover = FALSE,
  hover.data = unique(c(color.by, paste0(color.by, ".color.adj"), "color.multi",
   "color.which", x.by, paste0(x.by, ".x.adj"), y.by, paste0(y.by, ".y.adj"), shape.by,
     split.by)),
  hover.round.digits = 5,
  do.contour = FALSE,
  contour.color = "black",
  contour.linetype = 1,
  add.trajectory.by.groups = NULL,
  add.trajectory.curves = NULL,
```

```
  trajectory.group.by,
  trajectory.arrow.size = 0.15,
  add.xline = NULL,
  xline.linetype = "dashed",
  xline.color = "black",
  add.yline = NULL,
  yline.linetype = "dashed",
  yline.color = "black",
  do.letter = FALSE,
  do.ellipse = FALSE,
  do.label = FALSE,
  labels.size = 5,
  labels.highlight = TRUE,
  labels.use.numbers = FALSE,
  labels.numbers.spacer = ": ",
  labels.repel = TRUE,
  labels.repel.adjust = list(),
  labels.split.by = split.by,
  legend.show = TRUE,
  legend.color.title = "make",
  legend.color.size = 5,
  legend.color.breaks = waiver(),
  legend.color.breaks.labels = waiver(),
  legend.shape.title = shape.by,
  legend.shape.size = 5,
  show.grid.lines = TRUE,
  do.raster = FALSE,
  raster.dpi = 300,
  data.out = FALSE
)
```

Arguments

`data_frame`	A data_frame where columns are features and rows are observations you might wish to visualize.
`x.by, y.by`	Single strings denoting the name of a column of `data_frame` containing numeric data to use for the x- and y-axis of the scatterplot.
`color.by`	Single string denoting the name of a column of `data_frame` to use for setting the color of plotted points. Alternatively, a string vector naming multiple such columns of data to plot at once.
`shape.by`	Single string denoting the name of a column of `data_frame` containing discrete data to use for setting the shape of plotted points.
`split.by`	1 or 2 strings denoting the name(s) of column(s) of `data_frame` containing discrete data to use for faceting / separating data points into separate plots. When 2 columns are named, c(row,col), the first is used as rows and the second is used for columns of the resulting facet grid. When 1 column is named, shape control can be achieved with `split.nrow` and `split.ncol`

`size` Number which sets the size of data points. Default = 1.

`rows.use` String vector of rownames of `data_frame` OR an integer vector specifying the row-indices of data points which should be plotted.

Alternatively, a Logical vector, the same length as the number of rows in `data_frame`, where `TRUE` values indicate which rows to plot.

`show.others` Logical. TRUE by default, whether rows not targeted by `rows.use` should be shown in the background in light gray.

`x.adjustment`, `y.adjustment`, `color.adjustment`

A recognized string indicating whether numeric `x.by`, `y.by`, and `color.by` data should be used directly (default) or should be adjusted to be

- "z-score": scaled with the scale() function to produce a relative-to-mean z-score representation
- "relative.to.max": divided by the maximum value to give percent of max values between [0,1]

Ignored if the target data is not numeric as these known adjustments target numeric data only.

In order to leave the unedited data available for use in other features, the adjusted data are put in a new column and that new column is used for plotting.

`x.adj.fxn`, `y.adj.fxn`, `color.adj.fxn`

If you wish to apply a function to edit the `x.by`, `y.by`, or `color.by` data before use, in a way not possible with the `color.adjustment` input, this input can be given a function which takes in a vector of values as input and returns a vector of values of the same length as output.

For example, `function(x) {log2(x)}` or `as.factor`.

In order to leave the unedited data available for use in other features, the adjusted data are put in a new column and that new column is used for plotting.

`split.show.all.others`

Logical which sets whether gray "others" points of facets should include all points of other facets (`TRUE`) versus just points left out by `rows.use` which would exist in the current facet (`FALSE`).

`opacity` Number between 0 and 1. 1 = opaque. 0 = invisible. Default = 1. (In terms of typical ggplot variables, = alpha)

`color.panel` String vector which sets the colors to draw from when `color.by` indicates discrete data. `dittoColors()` by default, see `dittoColors` for contents.

A named vector can be used if names are matched to the distinct values of the `color.by` data.

`colors` Integer vector, the indexes / order, of colors from `color.panel` to actually use.

Useful for quickly swapping around colors of the default set (when not using names for color matching).

`split.nrow`, `split.ncol`

Integers which set the dimensions of faceting/splitting when faceting by a single feature.

`split.adjust` A named list which allows extra parameters to be pushed through to the faceting function call. List elements should be valid inputs to the faceting functions, e.g. ‘list(scales = "free")‘.

For options, when giving 1 column to split.by, see facet_wrap, OR when giving 2 columns to split.by, see facet_grid.

multivar.split.dir
: "row" or "col", sets the direction of faceting used for 'var' values when:
  - var is given multiple column names
  - AND split.by is used to provide an additional feature to facet by

shape.panel
: Vector of integers, corresponding to ggplot shapes, which sets what shapes to use in conjunction with shape.by. When nothing is supplied to shape.by, only the first value is used. Default is a set of 6, c(16,15,17,23,25,8), the first being a simple, solid, circle.

rename.color.groups
: String vector which sets new names for the identities of color.by groups.

rename.shape.groups
: String vector which sets new names for the identities of shape.by groups.

min.color
: color for min value of numeric color.by-data. Default = yellow

max.color
: color for max value of numeric color.by-data. Default = blue

min.value, max.value
: Number which sets the color.by-data value associated with the minimum or maximum colors.

plot.order
: String. If the data should be plotted based on the order of the color data, sets whether to plot in "increasing", "decreasing", or "randomize"d order.

xlab, ylab
: Strings which set the labels for the axes. To remove, set to NULL.

main
: String, sets the plot title. A default title is automatically generated based on color.by and shape.by when either are provided. To remove, set to NULL.

sub
: String, sets the plot subtitle.

theme
: A ggplot theme which will be applied before internal adjustments. Default = theme_bw(). See https://ggplot2.tidyverse.org/reference/ggtheme.html for other options and ideas.

do.hover
: Logical which controls whether the ggplot output will be converted to a plotly object so that data about individual points can be displayed when you hover your cursor over them. The hover.data argument is used to determine what data to show upon hover.

hover.data
: String vector which denotes what data to show for each data point, upon hover, when do.hover is set to TRUE. Defaults to all data expected to be useful. Only values present in the plotting data are actually used. These can be column names of data_frame and any column names which will be created to accommodate multivar and data adjustment functionality. You can run the function with data.out = TRUE and inspect the $Target_data output's columns to view your available options.

hover.round.digits
: Integer number specifying the number of decimal digits to round displayed numeric values to, when do.hover is set to TRUE.

do.contour
: Logical. Whether density-based contours should be displayed.

contour.color
: String that sets the color of the do.contour contours.

`contour.linetype`
: String or numeric which sets the type of line used for `do.contour` contours. Defaults to "solid", but see `linetype` for other options.

`add.trajectory.by.groups`
: List of vectors representing trajectory paths, each from start-group to end-group, where vector contents are the group-names indicated by the `trajectory.group.by` column of `data_frame`.

`add.trajectory.curves`
: List of matrices, each representing coordinates for a trajectory path, from start to end, where matrix columns represent x and y coordinates of the paths.

`trajectory.group.by`
: String denoting the name of a column of `data_frame` to use for generating trajectories from data point groups.

`trajectory.arrow.size`
: Number representing the size of trajectory arrows, in inches. Default = 0.15.

`add.xline`
: numeric value(s) where one or multiple vertical line(s) should be added.

`xline.linetype`
: String which sets the type of line for `add.xline`. Defaults to "dashed", but any ggplot linetype will work.

`xline.color`
: String that sets the color(s) of the `add.xline` line(s).

`add.yline`
: numeric value(s) where one or multiple vertical line(s) should be added.

`yline.linetype`
: String which sets the type of line for `add.yline`. Defaults to "dashed", but any ggplot linetype will work.

`yline.color`
: String that sets the color(s) of the `add.yline` line(s).

`do.letter`
: Logical which sets whether letters should be added on top of the colored dots. For extended colorblindness compatibility. NOTE: `do.letter` is ignored if `do.hover` = `TRUE` or `shape.by` is used because lettering is incompatible with plotly and with changing the dots' to be different shapes.

`do.ellipse`
: Logical. Whether `color.by` groups should be surrounded by median-centered ellipses.

`do.label`
: Logical. Whether to add text labels near the center (median) of `color.by` groups.

`labels.size`
: Number which sets the size of labels text when `do.label` = `TRUE`.

`labels.highlight`
: Logical. Whether labels should have a box behind them when `do.label` = `TRUE`.

`labels.use.numbers`
: Logical which controls whether numbers will be used in place of original data-values. When turned on, so number to value mapping can be known, these numbers are added to the legend.

`labels.numbers.spacer`
: String. When `do.label` = `TRUE` and `labels.use.numbers`, this string is used in the legend between the numbers and their associated data values.

`labels.repel`
: Logical, that sets whether the labels' placements will be adjusted with ggrepel to avoid intersections between labels and plot bounds when `do.label` = `TRUE`. TRUE by default.

`labels.repel.adjust`
: A named list which allows extra parameters to be pushed through to ggrepel function calls. List elements should be valid inputs to the `geom_label_repel` by default, or `geom_text_repel` when `labels.highlight = FALSE`.

`labels.split.by`
: String of one or two column names which controls the facet-split calculations for label placements. Defaults to `split.by`, so generally there is no need to adjust this except when if you plan to apply faceting externally.

`legend.show`
: Logical. Whether any legend should be displayed. Default = `TRUE`.

`legend.color.title`, `legend.shape.title`
: Strings which set the title for the color or shape legends.

`legend.color.size`, `legend.shape.size`
: Numbers representing the size of shapes in the color and shape legends (for discrete variable plotting). Default = 5. *Enlarging the icons in the colors legend is incredibly helpful for making colors more distinguishable by color blind individuals.

`legend.color.breaks`
: Numeric vector which sets the discrete values to label in the color-scale legend for `color.by`-data.

`legend.color.breaks.labels`
: String vector, with same length as `legend.color.breaks`, which sets the labels for the tick marks of the color-scale.

`show.grid.lines`
: Logical which sets whether grid lines should be shown within the plot space.

`do.raster`
: Logical. When set to `TRUE`, rasterizes the internal plot layer, changing it from individually encoded points to a flattened set of pixels. This can be useful for editing in external programs (e.g. Illustrator) when there are many thousands of data points.

`raster.dpi`
: Number indicating dots/pixels per inch (dpi) to use for rasterization. Default = 300.

`data.out`
: Logical. When set to `TRUE`, changes the output, from the plot alone, to a list containing the plot ("p"), a data.frame containing the underlying data for target rows ("Target_data"), a data.frame containing the underlying data for non-target rows ("Others_data"), and the ultimately used mapping of columns to given aesthetic sets ("cols_used"), because modification of newly made columns is required for many features.

Details

This function first makes any requested adjustments to data in the given `data_frame`, internally only, such as scaling the `color.by`-column if `color.adjustment` was given `"z-score"`.

Next, if a set of rows to target was indicated with the `rows.use` input, then the data_frame is split into `Target_data` and `Others_data`.

Then, rows are reordered to match with the requested `plot.order` behavior.

Finally, a scatter plot is created from the resultant data.frames. Non-target data points are colored in gray if `show.others=TRUE`, and target data points are displayed on top, colored and shaped based

on the color.by- and shape.by-associated data. If split.by was used, the plot will be split into a matrix of panels based on the associated groupings.

Value

a ggplot scatterplot where colored dots and/or shapes represent individual rows of the given data_frame.

Alternatively, if data.out=TRUE, a list containing four slots is output: the plot (named 'p'), a data.frame containing the underlying data for target rows (named 'Target_data'), a data.frame containing the underlying data for non-target rows (named 'Others_data'), and a list providing mappings of final column names in 'Target_data' to given plot aesthetics (named 'cols_used') because modification of newly made columns is required for many features.

Alternatively, if do.hover is set to TRUE, the plot is coverted from ggplot to plotly & additional information about each data point, determined by the hover.data input, is displayed upon hovering the cursor over the plot.

Many characteristics of the plot can be adjusted using discrete inputs

- size and opacity can be used to adjust the size and transparency of the data points. size can be given a number, or a column name of data_frame.
- Colors used can be adjusted with color.panel and/or colors for discrete data, or min, max, min.color, and max.color for continuous data.
- Shapes used can be adjusted with shape.panel.
- Color and shape labels can be changed using rename.color.groups and rename.shape.groups.
- Titles and axes labels can be adjusted with main, sub, xlab, ylab, and legend.title arguments.
- Legends can also be adjusted in other ways, using variables that all start with "legend." for easy tab completion lookup.

Author(s)

Daniel Bunis

See Also

scatterHex for a hex-binned version that can be useful when points are very dense.

Examples

```
example("dittoExampleData", echo = FALSE)

# The minimal inputs for scatterPlot are the 'data_frame', and 2 column names,
#   given to 'x.by' and 'y.by', indicating which data to use for the x and y
#   axes, respectively.
scatterPlot(
    example_df, x.by = "PC1", y.by = "PC2")

# 'color.by' and/or 'shape.by' can also be given column names in order to
#   show represent that columns data in the color or shape of the data points.
```

```
#   'shape.by' must be pointed to discrete data, but 'color.by' can be given
#   discrete or numeric data.
scatterPlot(
    example_df, x.by = "PC1", y.by = "PC2",
    color.by = "groups",
    shape.by = "SNP",
    size = 3)
scatterPlot(
    example_df, x.by = "PC1", y.by = "PC2",
    color.by = "gene1",
    size = 3)

# Data can be "split" or faceted by a discrete variable as well.
scatterPlot(example_df, x.by = "PC1", y.by = "PC2", color.by = "gene1",
    split.by = "timepoint") # single split.by element
scatterPlot(example_df, x.by = "PC1", y.by = "PC2", color.by = "gene1",
    split.by = c("groups","SNP")) # row and col split.by elements

# Modify the look with intuitive inputs
scatterPlot(example_df, x.by = "PC1", y.by = "PC2", color.by = "groups",
    size = 5,
    opacity = 0.3,
    show.grid.lines = FALSE,
    ylab = NULL, xlab = "PC2 by PC1",
    main = "Plot Title",
    sub = "subtitle",
    legend.color.title = "Legend\nRetitle")

# You can restrict to only certain data points using the 'rows.use' input.
#   The input can be given rownames, indexes, or a logical vector
#   All "other" points will now only be shown as a gray background, or will not
#   be shown add all if you also add 'show.others = FALSE'
scatterPlot(example_df, x.by = "PC1", y.by = "PC2", color.by = "groups",
    sub = "show only first 40 observations, by index",
    rows.use = 1:40)
scatterPlot(example_df, x.by = "PC1", y.by = "PC2", color.by = "groups",
    sub = "show only 3 observations, by name",
    rows.use = c("obs1", "obs2", "obs25"))
scatterPlot(example_df, x.by = "PC1", y.by = "PC2", color.by = "groups",
    sub = "show groups A,B,D only, by logical, without others as background",
    rows.use = example_df$groups!="C",
    show.others = FALSE)

# Many extra features are easy to add as well:
#   Each is started via an input starting with 'do.FEATURE*' or 'add.FEATURE*'
#   And when tweaks for that feature are possible, those inputs will start be
#   named starting with 'FEATURE*'. For example, color.by groups can be labeled
#   with 'do.label = TRUE' and the tweaks for this feature are given with inputs
#   'labels.size', 'labels.highlight', and 'labels.repel':
scatterPlot(example_df, x.by = "PC1", y.by = "PC2", color.by = "groups",
    sub = "default labeling",
    do.label = TRUE)          # Turns on the labeling feature
scatterPlot(example_df, x.by = "PC1", y.by = "PC2", color.by = "groups",
```

```
    sub = "tweaked labeling",
    do.label = TRUE,            # Turns on the labeling feature
    labels.size = 8,            # Adjust the text size of labels
    labels.highlight = FALSE,   # Removes white background behind labels
    # labels.use.numbers = TRUE,# Swap to number placeholders
    labels.repel = FALSE)       # Turns off anti-overlap location adjustments

# Faceting can also be used to show multiple continuous variables side-by-side
#   by giving a vector of column names to 'color.by'.
#   This can also be combined with 1 'split.by' variable, with direction then
#   controlled via 'multivar.split.dir':
scatterPlot(example_df, x.by = "PC1", y.by = "PC2",
    color.by = c("gene1", "gene2"))
scatterPlot(example_df, x.by = "PC1", y.by = "PC2",
    color.by = c("gene1", "gene2"),
    split.by = "groups")
scatterPlot(example_df, x.by = "PC1", y.by = "PC2",
    color.by = c("gene1", "gene2"),
    split.by = "groups",
    multivar.split.dir = "row")

# Sometimes, it can be useful for external editing or troubleshooting purposes
#   to see the underlying data that was directly used for plotting.
# 'data.out = TRUE' can be provided in order to obtain not just plot ("plot"),
#   but also the "Target_data" and "Others_data" data.frames and "cols_used"
#   returned as a list.
out <- scatterPlot(example_df, x.by = "PC1", y.by = "PC2", color.by = "groups",
    rows.use = 1:40,
    data.out = TRUE)
out$plot
summary(out$Target_data)
summary(out$Others_data)
out$cols_used
```

`yPlot` *Plots continuous data per group on a y- (or x-) axis using customizable data representations*

Description

Plots continuous data per group on a y- (or x-) axis using customizable data representations

Usage

```
yPlot(
  data_frame,
  var,
  group.by,
  color.by = group.by,
```

```
  shape.by = NULL,
  split.by = NULL,
  rows.use = NULL,
  plots = c("vlnplot", "boxplot", "jitter"),
  multivar.aes = c("split", "group", "color"),
  multivar.split.dir = c("col", "row"),
  var.adjustment = NULL,
  var.adj.fxn = NULL,
  do.hover = FALSE,
 hover.data = unique(c(var, paste0(var, ".adj"), "var.multi", "var.which", group.by,
    color.by, shape.by, split.by)),
  hover.round.digits = 5,
  color.panel = dittoColors(),
  colors = seq_along(color.panel),
  shape.panel = c(16, 15, 17, 23, 25, 8),
  theme = theme_classic(),
  main = "make",
  sub = NULL,
  ylab = "make",
  y.breaks = NULL,
  min = NA,
  max = NA,
  xlab = "make",
  x.labels = NULL,
  x.labels.rotate = NA,
  x.reorder = NULL,
  split.nrow = NULL,
  split.ncol = NULL,
  split.adjust = list(),
  do.raster = FALSE,
  raster.dpi = 300,
  jitter.size = 1,
  jitter.width = 0.2,
  jitter.color = "black",
  jitter.shape.legend.size = 5,
  jitter.shape.legend.show = TRUE,
  jitter.position.dodge = boxplot.position.dodge,
  boxplot.width = 0.2,
  boxplot.color = "black",
  boxplot.show.outliers = NA,
  boxplot.outlier.size = 1.5,
  boxplot.fill = TRUE,
  boxplot.position.dodge = vlnplot.width,
  boxplot.lineweight = 1,
  vlnplot.lineweight = 1,
  vlnplot.width = 1,
  vlnplot.scaling = "area",
  vlnplot.quantiles = NULL,
```

```
  ridgeplot.lineweight = 1,
  ridgeplot.scale = 1.25,
  ridgeplot.ymax.expansion = NA,
  ridgeplot.shape = c("smooth", "hist"),
  ridgeplot.bins = 30,
  ridgeplot.binwidth = NULL,
  add.line = NULL,
  line.linetype = "dashed",
  line.color = "black",
  legend.show = TRUE,
  legend.title = "make",
  data.out = FALSE
)

ridgePlot(..., plots = c("ridgeplot"))

ridgeJitter(..., plots = c("ridgeplot", "jitter"))

boxPlot(..., plots = c("boxplot", "jitter"))
```

Arguments

data_frame	A data_frame where columns are features and rows are observations you might wish to visualize.
var	Single string representing the name of a column of data_frame to be used as the primary, y-axis, data. Alternatively, a string vector naming multiple such columns of data to plot at once. See the input multivar.aes to understand or tweak how multiple var-data will be shown.
group.by	Single string representing the name of a column of data_frame containing discrete data to use for separating the data points into groups.
color.by	Single string representing the name of a column of data_frame containing discrete data to use for setting data representation color fills. This data does not need to be the same as group.by, which is great for highlighting supersets or subgroups when wanted, but it defaults to group.by so the input can often be skipped.
shape.by	Single string representing the name of a column of data_frame containing discrete data to use for setting shapes of the jitter points. When not provided, all jitter points will be dots.
split.by	1 or 2 strings denoting the name(s) of column(s) of data_frame containing discrete data to use for faceting / separating data points into separate plots. When 2 columns are named, c(row,col), the first is used as rows and the second is used for columns of the resulting facet grid. When 1 column is named, shape control can be achieved with split.nrow and split.ncol
rows.use	String vector of rownames of data_frame OR an integer vector specifying the row-indices of data points which should be plotted.

Alternatively, a Logical vector, the same length as the number of rows in `data_frame`, where `TRUE` values indicate which rows to plot.

`plots` String vector which sets the types of plots to include: possibilities = "jitter", "boxplot", "vlnplot", "ridgeplot".

Order matters: c("vlnplot", "boxplot", "jitter") will put a violin plot in the back, boxplot in the middle, and then individual dots in the front.

See details section for more info.

`multivar.aes` "split", "group", or "color", the plot feature to utilize for displaying 'var' value when `var` is given multiple column names. When set to "split" (the default), note that displaying the `var`-identity of the data will be prioritized so the `split.by` input becomes limited to receiving a single usable element.

`multivar.split.dir`

"row" or "col", sets the direction of faceting used for 'var' values when:

- `var` is given multiple column names
- `multivar.aes = "split"` (default)
- AND `split.by` is used to provide an additional feature to facet by

`var.adjustment` A recognized string indicating whether numeric `var` data should be used directly (default) or should be adjusted to be

- "z-score": scaled with the scale() function to produce a relative-to-mean z-score representation
- "relative.to.max": divided by the maximum expression value to give percent of max values between [0,1]

Ignored if the `var` data is not numeric as these known adjustments target numeric data only.

In order to leave the unedited data available for use in other features, the adjusted data are put in a new column and that new column is used for plotting.

`var.adj.fxn` If you wish to apply a function to edit the `var` data before use, in a way not possible with the `var.adjustment` input, this input can be given a function which takes in a vector of values as input and returns a vector of values of the same length as output.

For example, `function(x) {log2(x)}` or `as.factor`.

In order to leave the unedited data available for use in other features, the adjusted data are put in a new column and that new column is used for plotting.

`do.hover` Logical which controls whether the ggplot output will be converted to a plotly object so that data about individual points can be displayed when you hover your cursor over them. The `hover.data` argument is used to determine what data to show upon hover.

`hover.data` String vector which denotes what data to show for each jitter data point, upon hover, when `do.hover` is set to `TRUE`. Defaults to all data expected to be useful. Only values present in the plotting data are actually used. These can be column names of `data_frame` and any column names which will be created to accommodate multivar and data adjustment functionality. You can run the function with `data.out = TRUE` and inspect the `$data` output's columns to view your available options.

hover.round.digits
: Integer number specifying the number of decimal digits to round displayed numeric values to, when do.hover is set to TRUE.

color.panel
: String vector which sets the colors to draw from for data representation fills. Default = dittoColors().

  A named vector can be used if names are matched to the distinct values of the color.by data.

colors
: Integer vector, the indexes / order, of colors from color.panel to actually use.

  Useful for quickly swapping around colors of the default set (when not using names for color matching).

shape.panel
: Vector of integers corresponding to ggplot shapes which sets what shapes to use. When discrete groupings are supplied by shape.by, this sets the panel of shapes which will be used. When nothing is supplied to shape.by, only the first value is used. Default is a set of 6, c(16,15,17,23,25,8), the first being a simple, solid, circle.

theme
: A ggplot theme which will be applied before internal adjustments. Default = theme_classic(). See https://ggplot2.tidyverse.org/reference/ggtheme.html for other options and ideas.

main
: String, sets the plot title. Default = "make" and if left as make, a title will be automatically generated. To remove, set to NULL.

sub
: String, sets the plot subtitle.

ylab
: String, sets the continuous-axis label (=y-axis for box and violin plots, x-axis for ridgeplots). Defaults to "var".

y.breaks
: Numeric vector, a set of breaks that should be used as major grid lines. c(break1,break2,break3,etc.).

min, max
: Scalars which control the zoom on the continuous axis of the plot.

xlab
: String which sets the grouping-axis label (=x-axis for box and violin plots, y-axis for ridgeplots). Set to NULL to remove.

x.labels
: String vector, c("label1","label2","label3",...) which overrides the names of groupings.

x.labels.rotate
: Logical which sets whether the labels should be rotated. Default: TRUE for violin and box plots, but FALSE for ridgeplots.

x.reorder
: Integer vector. A sequence of numbers, from 1 to the number of groupings, for rearranging the order of x-axis groupings.

  Method: Make a first plot without this input. Then, treating the leftmost grouping as index 1, and the rightmost as index n. Values of x.reorder should be these indices, but in the order that you would like them rearranged to be.

  Recommendation for advanced users: If you find yourself coming back to this input too many times, an alternative solution that can be easier long-term is to make the target data into a factor, and to put its levels in the desired order: factor(data, levels = c("level1", "level2", ...)).

split.nrow, split.ncol
: Integers which set the dimensions of faceting/splitting when faceting by a single feature.

split.adjust
: A named list which allows extra parameters to be pushed through to the faceting function call. List elements should be valid inputs to the faceting functions, e.g. 'list(scales = "free")'.

  For options, when giving 1 column to split.by, see facet_wrap, OR when giving 2 columns to split.by, see facet_grid.

do.raster
: Logical. When set to TRUE, rasterizes the jitter plot layer, changing it from individually encoded points to a flattened set of pixels. This can be useful for editing in external programs (e.g. Illustrator) when there are many thousands of data points.

raster.dpi
: Number indicating dots/pixels per inch (dpi) to use for rasterization. Default = 300.

jitter.size
: Scalar which sets the size of the jitter shapes.

jitter.width
: Scalar that sets the width/spread of the jitter in the x direction. Ignored in ridgeplots.

  Note for when color.by is used to split x-axis groupings into additional bins: ggplot does not shrink jitter widths accordingly, so be sure to do so yourself! Ideally, needs to be 0.5/num_subgroups.

jitter.color
: String which sets the color of the jitter shapes

jitter.shape.legend.size
: Scalar which changes the size of the shape key in the legend. If set to NA, jitter.size is used.

jitter.shape.legend.show
: Logical which sets whether the shapes legend will be shown when its shape is determined by shape.by.

jitter.position.dodge
: Scalar which adjusts the relative distance between jitter widths when multiple subgroups exist per group.by grouping (a.k.a. when group.by and color.by are not equal). Similar to boxplot.position.dodge input & defaults to the value of that input so that BOTH will actually be adjusted when only, say, boxplot.position.dodge = 0.3 is given.

boxplot.width
: Scalar which sets the width/spread of the boxplot in the x direction

boxplot.color
: String which sets the color of the lines of the boxplot

boxplot.show.outliers
: Logical, whether outliers should by including in the boxplot. Default is FALSE when there is a jitter plotted, TRUE if there is no jitter.

boxplot.outlier.size
: Scalar which adjusts the size of points used to mark outliers.

boxplot.fill
: Logical, whether the boxplot should be filled in or not. Known bug: when boxplot fill is turned off, outliers do not render.

boxplot.position.dodge
: Scalar which adjusts the relative distance between boxplots when multiple are drawn per grouping (a.k.a. when group.by and color.by are not equal). By default, this input actually controls the value of jitter.position.dodge unless the jitter version is provided separately.

`boxplot.lineweight`
: Scalar which adjusts the thickness of boxplot lines.

`vlnplot.lineweight`
: Scalar which sets the thickness of the line that outlines the violin plots.

`vlnplot.width`
: Scalar which sets the width/spread of violin plots in the x direction

`vlnplot.scaling`
: String which sets how the widths of the of violin plots are set in relation to each other. Options are "area", "count", and "width". If the default is not right for your data, I recommend trying "width". For an explanation of each, see `geom_violin`.

`vlnplot.quantiles`
: Single number or numeric vector of values in [0,1] naming quantiles at which to draw a horizontal line within each violin plot. Example: `c(0.1, 0.5, 0.9)`

`ridgeplot.lineweight`
: Scalar which sets the thickness of the ridgeplot outline.

`ridgeplot.scale`
: Scalar which sets the distance/overlap between ridgeplots. A value of 1 means the tallest density curve just touches the baseline of the next higher one. Higher numbers lead to greater overlap. Default = 1.25

`ridgeplot.ymax.expansion`
: Scalar which adjusts the minimal space between the topmost grouping and the top of the plot in order to ensure the curve is not cut off by the plotting grid. The larger the value, the greater the space requested. When left as NA, dittoViz will attempt to determine an ideal value itself based on the number of groups & linear interpolation between these goal posts: #groups of 3 or fewer: 0.6; #groups=12: 0.1; #groups or 34 or greater: 0.05.

`ridgeplot.shape`
: Either "smooth" or "hist", sets whether ridges will be smoothed (the typical, and default) versus rectangular like a histogram. (Note: as of the time shape "hist" was added, combination of jittered points is not supported by the `stat_binline` that dittoViz relies on.)

`ridgeplot.bins`
: Integer which sets how many chunks to break the x-axis into when `ridgeplot.shape = "hist"`. Overridden by `ridgeplot.binwidth` when that input is provided.

`ridgeplot.binwidth`
: Integer which sets the width of chunks to break the x-axis into when `ridgeplot.shape = "hist"`. Takes precedence over `ridgeplot.bins` when provided.

`add.line`
: numeric value(s) where one or multiple line(s) should be added

`line.linetype`
: String which sets the type of line for `add.line`. Defaults to "dashed", but any ggplot linetype will work.

`line.color`
: String that sets the color(s) of the `add.line` line(s)

`legend.show`
: Logical. Whether the legend should be displayed. Default = `TRUE`.

`legend.title`
: String or `NULL`, sets the title for the main legend which includes colors and data representations.

data.out	Logical. When set to TRUE, changes the output, from the plot alone, to a list containing the plot (p), its underlying data (data), and the ultimately used mapping of columns to given aesthetic sets, because modification of newly made columns is required for many features ("cols_used").
...	arguments passed to yPlot by ridgePlot, ridgeJitter, and boxPlot wrappers. Options are all the ones above.

Details

The function plots the targeted var data of data_frame, grouped by the columns of data given to group.by and color.by, using data representations given by plots. Data representations will also be colored (filled) based on color.by. If a subset of data points to use is indicated with the rows.use input, the data_frame is internally subset to include only those indicated rows before plotting.

The plots argument determines the types of data representation that will be generated, as well as their order from back to front. Options are "jitter", "boxplot", "vlnplot", and "ridgeplot". Inclusion of "ridgeplot" overrides "boxplot" and "vlnplot" presence and changes the plot to be horizontal.

When split.by is provided a column name of data_frame, separate plots will be produced representing each of the distinct groupings of the split.by data using ggplots facetting functionality.

ridgePlot, ridgeJitter, and boxPlot are included as wrappers of the basic yPlot function that simply change the default for the plots input to be "ridgeplot", c("ridgeplot","jitter"), or c("boxplot","jitter"), to make such plots even easier to produce.

Value

a ggplot where continuous data, grouped by sample, age, cluster, etc., shown on either the y-axis by a violin plot, boxplot, and/or jittered points, or on the x-axis by a ridgeplot with or without jittered points.

Alternatively when data.out=TRUE, a list containing the plot ("p") the underlying data as a dataframe ("data"), and the ultimately used mapping of columns to given aesthetic sets ("cols_used"), because modification of newly made columns is required for many features.

Alternatively when do.hover = TRUE, a plotly converted version of the ggplot where additional data will be displayed when the cursor is hovered over jitter points.

Functions

- ridgePlot(): simple yPlot wrapper with distinct plots input defaults
- ridgeJitter(): simple yPlot wrapper with distinct plots input defaults
- boxPlot(): simple yPlot wrapper with distinct plots input defaults

Many characteristics of the plot can be adjusted using discrete inputs

The plots argument determines the types of **data representation** that will be generated, as well as their order from back to front. Options are "jitter", "boxplot", "vlnplot", and "ridgeplot".

Each plot type has specific associated options which are controlled by variables that start with their associated string. For example, all jitter adjustments start with "jitter.", such as jitter.size and jitter.width.

Inclusion of "ridgeplot" overrides "boxplot" and "vlnplot" presence and changes the plot to be horizontal.

Additionally:

- **Colors can be adjusted** with color.panel.
- **Subgroupings:** color.by can be utilized to split major group.by groupings into subgroups. When this is done in y-axis plotting, dittoViz automatically ensures the centers of all geoms will align, but users will need to manually adjust jitter.width to less than 0.5/num_subgroups to avoid overlaps. There are also three inputs through which one can use to control geom-center placement, but the easiest way to do all at once so is to just adjust vlnplot.width! The other two: boxplot.position.dodge, and jitter.position.dodge.
- **Line(s) can be added** at single or multiple value(s) by providing these values to add.line. Linetype and color are set with line.linetype, which is "dashed" by default, and line.color, which is "black" by default.
- **Titles and axes labels** can be adjusted with main, sub, xlab, ylab, and legend.title arguments.
- The **legend can be hidden** by setting legend.show = FALSE.
- **y-axis zoom and tick marks** can be adjusted using min, max, and y.breaks.
- **x-axis labels and groupings** can be changed / reordered using x.labels and x.reorder, and rotation of these labels can be turned on/off with x.labels.rotate = TRUE/FALSE.
- **Shapes used** in conjunction with shape.by can be adjusted with shape.panel. This can be very useful for making manual additional alterations *after* dittoViz plot generation.

Author(s)

Daniel Bunis

See Also

ridgePlot, ridgeJitter, and boxPlot for shortcuts to a few 'plots' input shortcuts

Examples

```
example("dittoExampleData", echo = FALSE)

# Basic yPlot, with jitter behind a vlnplot (looks better with more points)
yPlot(data_frame = example_df, var = "gene1", group.by = "timepoint")
yPlot(data_frame = example_df, var = c("gene1", "gene2"), group.by = "timepoint")

# Color distinctly from the grouping variable using 'color.by'
yPlot(data_frame = example_df, var = "gene1", group.by = "timepoint",
    color.by = "conditions")

# Update the 'plots' input to change / reorder the data representations
```

```
yPlot(example_df, "gene1", "timepoint",
    plots = c("vlnplot", "boxplot", "jitter"))
yPlot(example_df, "gene1", "timepoint",
    plots = c("ridgeplot", "jitter"))

# Provided wrappers enable certain easy adjustments of the 'plots' parameter.
# Quickly make a Boxplot
boxPlot(example_df, "gene1", "timepoint")
# Quickly make a Ridgeplot, with or without jitter
ridgePlot(example_df, "gene1", "timepoint")
ridgeJitter(example_df, "gene1", "timepoint")

# Modify the look with intuitive inputs
yPlot(example_df, "gene1", "timepoint",
    plots = c("vlnplot", "boxplot", "jitter"),
    boxplot.color = "white",
    main = "CD3E",
    legend.show = FALSE)

# Data can also be split in other ways with 'shape.by' or 'split.by'
yPlot(data_frame = example_df, var = "gene1", group.by = "timepoint",
    plots = c("vlnplot", "boxplot", "jitter"),
    shape.by = "clustering",
    split.by = "SNP") # single split.by element
yPlot(data_frame = example_df, var = "gene1", group.by = "timepoint",
    plots = c("vlnplot", "boxplot", "jitter"),
    split.by = c("groups","SNP")) # row and col split.by elements

# Multiple features can also be plotted at once by giving them as a vector to
#   the 'var' input. One aesthetic of the plot will then be used to display the
#   'var'-info, and you can control which (faceting / "split", x-axis grouping
#   / "group", or color / "color") with 'multivar.aes':
yPlot(data_frame = example_df, group.by = "timepoint",
    var = c("gene1", "gene2"))
yPlot(data_frame = example_df, group.by = "timepoint",
    var = c("gene1", "gene2"),
    multivar.aes = "group")
yPlot(data_frame = example_df, group.by = "timepoint",
    var = c("gene1", "gene2"),
    multivar.aes = "color")
```

Index