Package: ggstats (via r-universe)

August 14, 2024

Type Package

Title Extension to 'ggplot2' for Plotting Stats

Version 0.6.0.9000

Description Provides new statistics, new geometries and new positions
for 'ggplot2' and a suite of functions to facilitate the
creation of statistical plots.

License GPL (>= 3)

URL `https://larmarange.github.io/ggstats/`,
`https://github.com/larmarange/ggstats`

BugReports `https://github.com/larmarange/ggstats/issues`

Imports broom.helpers (>= 1.14.0), cli, dplyr, forcats, ggplot2 (>=
3.4.0), lifecycle, magrittr, patchwork, purrr, rlang, scales,
stats, stringr, tidyr

Suggests betareg, broom, emmeans, glue, knitr, labelled (>= 2.11.0),
reshape, rmarkdown, nnet, parameters, pscl, testthat (>=
3.0.0), spelling, survey, survival, vdiffr

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

Roxygen list(markdown = TRUE)

Config/testthat/edition 3

Language en-US

VignetteBuilder knitr

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

RemoteUrl https://github.com/larmarange/ggstats

RemoteRef HEAD

RemoteSha a4eda388b1bb17022dbfba81e038483151ab8f3e

Contents

`augment_chisq_add_phi` *Augment a chi-squared test and compute phi coefficients*

Description

Augment a chi-squared test and compute phi coefficients

Usage

```
augment_chisq_add_phi(x)
```

Arguments

`x`	a chi-squared test as returned by `stats::chisq.test()`

Details

Phi coefficients are a measurement of the degree of association between two binary variables.

- A value between -1.0 to -0.7 indicates a strong negative association.
- A value between -0.7 to -0.3 indicates a weak negative association.
- A value between -0.3 to +0.3 indicates a little or no association.
- A value between +0.3 to +0.7 indicates a weak positive association.
- A value between +0.7 to +1.0 indicates a strong positive association.

Value

A `tibble`.

See Also

`stat_cross()`, `GDAtools::phi.table()` or `psych::phi()`

Examples

```
tab <- xtabs(Freq ~ Sex + Class, data = as.data.frame(Titanic))
augment_chisq_add_phi(chisq.test(tab))
```

`geom_stripped_rows` *Alternating Background Color*

Description

Add alternating background color along the y-axis. The geom takes default aesthetics odd and even that receive color codes.

Usage

```
geom_stripped_rows(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  show.legend = NA,
  inherit.aes = TRUE,
  xfrom = -Inf,
  xto = Inf,
  width = 1,
  nudge_y = 0
)

geom_stripped_cols(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  show.legend = NA,
  inherit.aes = TRUE,
  yfrom = -Inf,
  yto = Inf,
  width = 1,
  nudge_x = 0
)
```

Arguments

`mapping`
: Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.

`data`
: The data to be displayed in this layer. There are three options:

  If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`.

  A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created.

  A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).

`stat`
: The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used the override the default coupling between geoms and stats. The `stat` argument accepts the following:

  - A `Stat` ggproto subclass, for example `StatCount`.
  - A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`.
  - For more information and other ways to specify the stat, see the layer stat documentation.

`position`
: A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

  - The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
  - A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
  - For more information and other ways to specify the position, see the layer position documentation.

`...`
: Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

  - Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.

- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`show.legend`	logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes. It can also be a named logical vector to finely select the aesthetics to display.
`inherit.aes`	If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn't inherit behaviour from the default plot specification, e.g. `borders()`.
`xfrom, xto`	limitation of the strips along the x-axis
`width`	width of the strips
`yfrom, yto`	limitation of the strips along the y-axis
`nudge_x, nudge_y`	horizontal or vertical adjustment to nudge strips by

Value

A `ggplot2` plot with the added geometry.

Examples

```
data(tips, package = "reshape")

library(ggplot2)
p <- ggplot(tips) +
  aes(x = time, y = day) +
  geom_count() +
  theme_light()

p
p + geom_stripped_rows()
p + geom_stripped_cols()
p + geom_stripped_rows() + geom_stripped_cols()


p <- ggplot(tips) +
  aes(x = total_bill, y = day) +
  geom_count() +
  theme_light()
p
p + geom_stripped_rows()
p + geom_stripped_rows() + scale_y_discrete(expand = expansion(0, 0.5))
p + geom_stripped_rows(xfrom = 10, xto = 35)
p + geom_stripped_rows(odd = "blue", even = "yellow")
p + geom_stripped_rows(odd = "blue", even = "yellow", alpha = .1)
p + geom_stripped_rows(odd = "#00FF0022", even = "#FF000022")

p + geom_stripped_cols()
p + geom_stripped_cols(width = 10)
p + geom_stripped_cols(width = 10, nudge_x = 5)
```

ggcoef_model *Plot model coefficients*

Description

ggcoef_model(), ggcoef_table(), ggcoef_multinom(), ggcoef_multicomponents() and ggcoef_compare() use broom.helpers::tidy_plus_plus() to obtain a tibble of the model coefficients, apply additional data transformation and then pass the produced tibble to ggcoef_plot() to generate the plot.

Usage

```
ggcoef_model(
  model,
  tidy_fun = broom.helpers::tidy_with_broom_or_parameters,
  tidy_args = NULL,
  conf.int = TRUE,
  conf.level = 0.95,
  exponentiate = FALSE,
  variable_labels = NULL,
  term_labels = NULL,
  interaction_sep = " * ",
  categorical_terms_pattern = "{level}",
  add_reference_rows = TRUE,
  no_reference_row = NULL,
  intercept = FALSE,
  include = dplyr::everything(),
  add_pairwise_contrasts = FALSE,
  pairwise_variables = broom.helpers::all_categorical(),
  keep_model_terms = FALSE,
  pairwise_reverse = TRUE,
  emmeans_args = list(),
  significance = 1 - conf.level,
  significance_labels = NULL,
  show_p_values = TRUE,
  signif_stars = TRUE,
  return_data = FALSE,
  ...
)

ggcoef_table(
  model,
  tidy_fun = broom.helpers::tidy_with_broom_or_parameters,
  tidy_args = NULL,
  conf.int = TRUE,
  conf.level = 0.95,
  exponentiate = FALSE,
  variable_labels = NULL,
  term_labels = NULL,
  interaction_sep = " * ",
  categorical_terms_pattern = "{level}",
  add_reference_rows = TRUE,
  no_reference_row = NULL,
  intercept = FALSE,
  include = dplyr::everything(),
  add_pairwise_contrasts = FALSE,
  pairwise_variables = broom.helpers::all_categorical(),
  keep_model_terms = FALSE,
  pairwise_reverse = TRUE,
  emmeans_args = list(),
  significance = 1 - conf.level,
  significance_labels = NULL,
  show_p_values = FALSE,
  signif_stars = FALSE,
  table_stat = c("estimate", "ci", "p.value"),
  table_header = NULL,
  table_text_size = 3,
  table_stat_label = NULL,
  ci_pattern = "{conf.low}, {conf.high}",
  table_witdhs = c(3, 2),
  plot_title = NULL,
  ...
)

ggcoef_compare(
  models,
  type = c("dodged", "faceted"),
  tidy_fun = broom.helpers::tidy_with_broom_or_parameters,
  tidy_args = NULL,
  conf.int = TRUE,
  conf.level = 0.95,
  exponentiate = FALSE,
  variable_labels = NULL,
  term_labels = NULL,
  interaction_sep = " * ",
  categorical_terms_pattern = "{level}",
  add_reference_rows = TRUE,
  no_reference_row = NULL,
  intercept = FALSE,
  include = dplyr::everything(),
  add_pairwise_contrasts = FALSE,
  pairwise_variables = broom.helpers::all_categorical(),
  keep_model_terms = FALSE,
  pairwise_reverse = TRUE,
  emmeans_args = list(),
  significance = 1 - conf.level,
  significance_labels = NULL,
  return_data = FALSE,
  ...
)

ggcoef_multinom(
  model,
  type = c("dodged", "faceted", "table"),
  y.level_label = NULL,
  tidy_fun = broom.helpers::tidy_with_broom_or_parameters,
  tidy_args = NULL,
  conf.int = TRUE,
  conf.level = 0.95,
  exponentiate = FALSE,
  variable_labels = NULL,
  term_labels = NULL,
  interaction_sep = " * ",
  categorical_terms_pattern = "{level}",
  add_reference_rows = TRUE,
  no_reference_row = NULL,
  intercept = FALSE,
  include = dplyr::everything(),
  significance = 1 - conf.level,
  significance_labels = NULL,
  return_data = FALSE,
  table_stat = c("estimate", "ci", "p.value"),
  table_header = NULL,
  table_text_size = 3,
  table_stat_label = NULL,
  ci_pattern = "{conf.low}, {conf.high}",
  table_witdhs = c(3, 2),
  ...
)
```

```
ggcoef_multicomponents(
  model,
  type = c("dodged", "faceted", "table"),
  component_col = "component",
  component_label = NULL,
  tidy_fun = broom.helpers::tidy_with_broom_or_parameters,
  tidy_args = NULL,
  conf.int = TRUE,
  conf.level = 0.95,
  exponentiate = FALSE,
  variable_labels = NULL,
  term_labels = NULL,
  interaction_sep = " * ",
  categorical_terms_pattern = "{level}",
  add_reference_rows = TRUE,
  no_reference_row = NULL,
  intercept = FALSE,
  include = dplyr::everything(),
  significance = 1 - conf.level,
  significance_labels = NULL,
  return_data = FALSE,
  table_stat = c("estimate", "ci", "p.value"),
  table_header = NULL,
  table_text_size = 3,
  table_stat_label = NULL,
  ci_pattern = "{conf.low}, {conf.high}",
  table_witdhs = c(3, 2),
  ...
)

ggcoef_plot(
  data,
  x = "estimate",
  y = "label",
  exponentiate = FALSE,
  point_size = 2,
  point_stroke = 2,
  point_fill = "white",
  colour = NULL,
  colour_guide = TRUE,
  colour_lab = "",
  colour_labels = ggplot2::waiver(),
  shape = "significance",
  shape_values = c(16, 21),
  shape_guide = TRUE,
  shape_lab = "",
  errorbar = TRUE,
  errorbar_height = 0.1,
  errorbar_coloured = FALSE,
  stripped_rows = TRUE,
  strips_odd = "#11111111",
  strips_even = "#00000000",
  vline = TRUE,
  vline_colour = "grey50",
  dodged = FALSE,
  dodged_width = 0.8,
  facet_row = "var_label",
  facet_col = NULL,
  facet_labeller = "label_value"
)
```

Arguments

`model`	a regression model object
`tidy_fun`	option to specify a custom tidier function
`tidy_args`	Additional arguments passed to `broom.helpers::tidy_plus_plus()` and to `tidy_fun`
`conf.int`	should confidence intervals be computed? (see `broom::tidy()`)
`conf.level`	the confidence level to use for the confidence interval if `conf.int` = `TRUE`; must be strictly greater than 0 and less than 1; defaults to 0.95, which corresponds to a 95 percent confidence interval
`exponentiate`	if `TRUE` a logarithmic scale will be used for x-axis
`variable_labels`	a named list or a named vector of custom variable labels
`term_labels`	a named list or a named vector of custom term labels
`interaction_sep`	separator for interaction terms
`categorical_terms_pattern`	a glue pattern for labels of categorical terms with treatment or sum contrasts (see `model_list_terms_levels()`)
`add_reference_rows`	should reference rows be added?
`no_reference_row`	variables (accepts tidyselect notation) for those no reference row should be added, when `add_reference_rows` = `TRUE`
`intercept`	should the intercept(s) be included?
`include`	variables to include. Accepts tidyselect syntax. Use `-` to remove a variable. Default is `everything()`. See also `all_continuous()`, `all_categorical()`, `all_dichotomous()` and `all_interaction()`
`add_pairwise_contrasts`	apply `tidy_add_pairwise_contrasts()`? **[Experimental]**
`pairwise_variables`	variables to add pairwise contrasts (accepts tidyselect notation)

`keep_model_terms`	keep original model terms for variables where pairwise contrasts are added? (default is `FALSE`)
`pairwise_reverse`	determines whether to use `"pairwise"` (if `TRUE`) or `"revpairwise"` (if `FALSE`), see `emmeans::contrast()`
`emmeans_args`	list of additional parameter to pass to `emmeans::emmeans()` when computing pairwise contrasts
`significance`	level (between 0 and 1) below which a coefficient is consider to be significantly different from 0 (or 1 if `exponentiate = TRUE`), `NULL` for not highlighting such coefficients
`significance_labels`	optional vector with custom labels for significance variable
`show_p_values`	if `TRUE`, add p-value to labels
`signif_stars`	if `TRUE`, add significant stars to labels
`return_data`	if `TRUE`, will return the data.frame used for plotting instead of the plot
`...`	parameters passed to `ggcoef_plot()`
`table_stat`	statistics to display in the table, use any column name returned by the tidier or `"ci"` for confidence intervals formatted according to `ci_pattern`
`table_header`	optional custom headers for the table
`table_text_size`	text size for the table
`table_stat_label`	optional named list of labeller functions for the displayed statistic (see examples)
`ci_pattern`	glue pattern for confidence intervals in the table
`table_witdhs`	relative widths of the forest plot and the coefficients table
`plot_title`	an optional plot title
`models`	named list of models
`type`	a dodged plot, a faceted plot or multiple table plots?
`y.level_label`	an optional named vector for labeling `y.level` (see examples)
`component_col`	name of the component column
`component_label`	an optional named vector for labeling components
`data`	a data frame containing data to be plotted, typically the output of `ggcoef_model()`, `ggcoef_compare()` or `ggcoef_multinom()` with the option `return_data = TRUE`
`x, y`	variables mapped to x and y axis
`point_size`	size of the points
`point_stroke`	thickness of the points
`point_fill`	fill colour for the points
`colour`	optional variable name to be mapped to colour aesthetic
`colour_guide`	should colour guide be displayed in the legend?

`colour_lab`	label of the colour aesthetic in the legend
`colour_labels`	labels argument passed to `ggplot2::scale_colour_discrete()` and `ggplot2::discrete_scale()`
`shape`	optional variable name to be mapped to the shape aesthetic
`shape_values`	values of the different shapes to use in `ggplot2::scale_shape_manual()`
`shape_guide`	should shape guide be displayed in the legend?
`shape_lab`	label of the shape aesthetic in the legend
`errorbar`	should error bars be plotted?
`errorbar_height`	height of error bars
`errorbar_coloured`	should error bars be colored as the points?
`stripped_rows`	should stripped rows be displayed in the background?
`strips_odd`	color of the odd rows
`strips_even`	color of the even rows
`vline`	should a vertical line be drawn at 0 (or 1 if `exponentiate = TRUE`)?
`vline_colour`	colour of vertical line
`dodged`	should points be dodged (according to the colour aesthetic)?
`dodged_width`	width value for `ggplot2::position_dodge()`
`facet_row`	variable name to be used for row facets
`facet_col`	optional variable name to be used for column facets
`facet_labeller`	labeller function to be used for labeling facets; if labels are too long, you can use `ggplot2::label_wrap_gen()` (see examples), more information in the documentation of `ggplot2::facet_grid()`

Details

For more control, you can use the argument `return_data = TRUE` to get the produced `tibble`, apply any transformation of your own and then pass your customized `tibble` to `ggcoef_plot()`.

Value

A `ggplot2` plot or a `tibble` if `return_data = TRUE`.

Functions

- `ggcoef_table()`: a variation of `ggcoef_model()` adding a table with estimates, confidence intervals and p-values
- `ggcoef_compare()`: designed for displaying several models on the same plot.
- `ggcoef_multinom()`: a variation of `ggcoef_model()` adapted to multinomial logistic regressions performed with `nnet::multinom()`.
- `ggcoef_multicomponents()`: a variation of `ggcoef_model()` adapted to multi-component models such as zero-inflated models or beta regressions. `ggcoef_multicomponents()` has been tested with `pscl::zeroinfl()`, `pscl::hurdle()` and `betareg::betareg()`
- `ggcoef_plot()`: plot a tidy `tibble` of coefficients

See Also

```
vignette("ggcoef_model")
```

Examples

```
mod <- lm(Sepal.Length ~ Sepal.Width + Species, data = iris)
ggcoef_model(mod)

ggcoef_table(mod)



ggcoef_table(mod, table_stat = c("estimate", "ci"))

ggcoef_table(
  mod,
  table_stat_label = list(
    estimate = scales::label_number(.001)
  )
)

ggcoef_table(mod, table_text_size = 5, table_witdhs = c(1, 1))

# a logistic regression example
d_titanic <- as.data.frame(Titanic)
d_titanic$Survived <- factor(d_titanic$Survived, c("No", "Yes"))
mod_titanic <- glm(
  Survived ~ Sex * Age + Class,
  weights = Freq,
  data = d_titanic,
  family = binomial
)

# use 'exponentiate = TRUE' to get the Odds Ratio
ggcoef_model(mod_titanic, exponentiate = TRUE)

ggcoef_table(mod_titanic, exponentiate = TRUE)

# display intercepts
ggcoef_model(mod_titanic, exponentiate = TRUE, intercept = TRUE)

# customize terms labels
ggcoef_model(
  mod_titanic,
  exponentiate = TRUE,
  show_p_values = FALSE,
  signif_stars = FALSE,
  add_reference_rows = FALSE,
  categorical_terms_pattern = "{level} (ref: {reference_level})",
  interaction_sep = " x "
) +
  ggplot2::scale_y_discrete(labels = scales::label_wrap(15))
```

```
# display only a subset of terms
ggcoef_model(mod_titanic, exponentiate = TRUE, include = c("Age", "Class"))

# do not change points' shape based on significance
ggcoef_model(mod_titanic, exponentiate = TRUE, significance = NULL)

# a black and white version
ggcoef_model(
  mod_titanic,
  exponentiate = TRUE,
  colour = NULL, stripped_rows = FALSE
)

# show dichotomous terms on one row
ggcoef_model(
  mod_titanic,
  exponentiate = TRUE,
  no_reference_row = broom.helpers::all_dichotomous(),
  categorical_terms_pattern =
    "{ifelse(dichotomous, paste0(level, ' / ', reference_level), level)}",
  show_p_values = FALSE
)




data(tips, package = "reshape")
mod_simple <- lm(tip ~ day + time + total_bill, data = tips)
ggcoef_model(mod_simple)

# custom variable labels
# you can use the labelled package to define variable labels
# before computing model
if (requireNamespace("labelled")) {
  tips_labelled <- tips %>%
    labelled::set_variable_labels(
      day = "Day of the week",
      time = "Lunch or Dinner",
      total_bill = "Bill's total"
    )
  mod_labelled <- lm(tip ~ day + time + total_bill, data = tips_labelled)
  ggcoef_model(mod_labelled)
}

# you can provide custom variable labels with 'variable_labels'
ggcoef_model(
  mod_simple,
  variable_labels = c(
    day = "Week day",
    time = "Time (lunch or dinner ?)",
    total_bill = "Total of the bill"
  )
)
# if labels are too long, you can use 'facet_labeller' to wrap them
ggcoef_model(
  mod_simple,
  variable_labels = c(
    day = "Week day",
    time = "Time (lunch or dinner ?)",
    total_bill = "Total of the bill"
  ),
  facet_labeller = ggplot2::label_wrap_gen(10)
)

# do not display variable facets but add colour guide
ggcoef_model(mod_simple, facet_row = NULL, colour_guide = TRUE)

# works also with with polynomial terms
mod_poly <- lm(
  tip ~ poly(total_bill, 3) + day,
  data = tips,
)
ggcoef_model(mod_poly)

# or with different type of contrasts
# for sum contrasts, the value of the reference term is computed
if (requireNamespace("emmeans")) {
  mod2 <- lm(
    tip ~ day + time + sex,
    data = tips,
    contrasts = list(time = contr.sum, day = contr.treatment(4, base = 3))
  )
  ggcoef_model(mod2)
}



# Use ggcoef_compare() for comparing several models on the same plot
mod1 <- lm(Fertility ~ ., data = swiss)
mod2 <- step(mod1, trace = 0)
mod3 <- lm(Fertility ~ Agriculture + Education * Catholic, data = swiss)
models <- list(
  "Full model" = mod1,
  "Simplified model" = mod2,
  "With interaction" = mod3
)

ggcoef_compare(models)
ggcoef_compare(models, type = "faceted")

# you can reverse the vertical position of the point by using a negative
# value for dodged_width (but it will produce some warnings)
ggcoef_compare(models, dodged_width = -.9)
```

```
# specific function for nnet::multinom models
mod <- nnet::multinom(Species ~ ., data = iris)
ggcoef_multinom(mod, exponentiate = TRUE)
ggcoef_multinom(mod, type = "faceted")
ggcoef_multinom(
  mod,
  type = "faceted",
  y.level_label = c("versicolor" = "versicolor\n(ref: setosa)")
)




library(pscl)
data("bioChemists", package = "pscl")
mod <- zeroinfl(art ~ fem * mar | fem + mar, data = bioChemists)
ggcoef_multicomponents(mod)

ggcoef_multicomponents(mod, type = "f")

ggcoef_multicomponents(mod, type = "t")

ggcoef_multicomponents(
  mod,
  type = "t",
  component_label = c(conditional = "Count", zero_inflated = "Zero-inflated")
)

mod2 <- zeroinfl(art ~ fem + mar | 1, data = bioChemists)
ggcoef_multicomponents(mod2, type = "t")
```

`gglikert` *Plotting Likert-type items*

Description

Combines several factor variables using the same list of ordered levels (e.g. Likert-type scales) into a unique data frame and generates a centered bar plot.

Usage

```
gglikert(
  data,
  include = dplyr::everything(),
  weights = NULL,
  y = ".question",
  variable_labels = NULL,
  sort = c("none", "ascending", "descending"),
  sort_method = c("prop", "prop_lower", "mean", "median"),
  sort_prop_include_center = totals_include_center,
  factor_to_sort = ".question",
  exclude_fill_values = NULL,
  cutoff = NULL,
  data_fun = NULL,
  add_labels = TRUE,
  labels_size = 3.5,
  labels_color = "auto",
  labels_accuracy = 1,
  labels_hide_below = 0.05,
  add_totals = TRUE,
  totals_size = labels_size,
  totals_color = "black",
  totals_accuracy = labels_accuracy,
  totals_fontface = "bold",
  totals_include_center = FALSE,
  totals_hjust = 0.1,
  y_reverse = TRUE,
  y_label_wrap = 50,
  reverse_likert = FALSE,
  width = 0.9,
  facet_rows = NULL,
  facet_cols = NULL,
  facet_label_wrap = 50,
  symmetric = FALSE
)

gglikert_data(
  data,
  include = dplyr::everything(),
  weights = NULL,
  variable_labels = NULL,
  sort = c("none", "ascending", "descending"),
  sort_method = c("prop", "prop_lower", "mean", "median"),
  sort_prop_include_center = TRUE,
  factor_to_sort = ".question",
  exclude_fill_values = NULL,
  cutoff = NULL,
  data_fun = NULL
)

gglikert_stacked(
  data,
  include = dplyr::everything(),
  weights = NULL,
  y = ".question",
  variable_labels = NULL,
  sort = c("none", "ascending", "descending"),
  sort_method = c("prop", "prop_lower", "mean", "median"),
  sort_prop_include_center = FALSE,
  factor_to_sort = ".question",
  data_fun = NULL,
  add_labels = TRUE,
  labels_size = 3.5,
  labels_color = "auto",
  labels_accuracy = 1,
  labels_hide_below = 0.05,
  add_median_line = FALSE,
  y_reverse = TRUE,
  y_label_wrap = 50,
  reverse_fill = TRUE,
  width = 0.9
)
```

Arguments

`data`	a data frame
`include`	variables to include, accepts tidy-select syntax
`weights`	optional variable name of a weighting variable, accepts tidy-select syntax
`y`	name of the variable to be plotted on `y` axis (relevant when `.question` is mapped to "facets, see examples), accepts tidy-select syntax
`variable_labels`	a named list or a named vector of custom variable labels
`sort`	should the factor defined by `factor_to_sort` be sorted according to the answers (see `sort_method`)? One of "none" (default), "ascending" or "descending"
`sort_method`	method used to sort the variables: `"prop"` sort according to the proportion of answers higher than the centered level, `"prop_lower"` according to the proportion lower than the centered level, `"mean"` considers answer as a score and sort according to the mean score, `"median"` used the median and the majority judgment rule for tie-breaking.
`sort_prop_include_center`	when sorting with `"prop"` and if the number of levels is uneven, should half of the central level be taken into account to compute the proportion?
`factor_to_sort`	name of the factor column to sort if `sort` is not equal to `"none"`; by default the list of questions passed to `include`; should be one factor column of the tibble returned by `gglikert_data()`; accepts tidy-select syntax
`exclude_fill_values`	Vector of values that should not be displayed (but still taken into account for computing proportions), see `position_likert()`

`cutoff`	number of categories to be displayed negatively (i.e. on the left of the x axis or the bottom of the y axis), could be a decimal value: `2` to display negatively the two first categories, `2.5` to display negatively the two first categories and half of the third, `2.2` to display negatively the two first categories and a fifth of the third (see examples). By default (`NULL`), it will be equal to the number of categories divided by 2, i.e. it will be centered.
`data_fun`	for advanced usage, custom function to be applied to the generated dataset at the end of `gglikert_data()`
`add_labels`	should percentage labels be added to the plot?
`labels_size`	size of the percentage labels
`labels_color`	color of the percentage labels (`"auto"` to use `hex_bw()` to determine a font color based on background color)
`labels_accuracy`	accuracy of the percentages, see `scales::label_percent()`
`labels_hide_below`	if provided, values below will be masked, see `label_percent_abs()`
`add_totals`	should the total proportions of negative and positive answers be added to plot? **This option is not compatible with facets!**
`totals_size`	size of the total proportions
`totals_color`	color of the total proportions
`totals_accuracy`	accuracy of the total proportions, see `scales::label_percent()`
`totals_fontface`	font face of the total proportions
`totals_include_center`	if the number of levels is uneven, should half of the center level be added to the total proportions?
`totals_hjust`	horizontal adjustment of totals labels on the x axis
`y_reverse`	should the y axis be reversed?
`y_label_wrap`	number of characters per line for y axis labels, see `scales::label_wrap()`
`reverse_likert`	if `TRUE`, will reverse the default stacking order, see `position_likert()`
`width`	bar width, see `ggplot2::geom_bar()`
`facet_rows, facet_cols`	A set of variables or expressions quoted by `ggplot2::vars()` and defining faceting groups on the rows or columns dimension (see examples)
`facet_label_wrap`	number of characters per line for facet labels, see `ggplot2::label_wrap_gen()`
`symmetric`	should the x-axis be symmetric?
`add_median_line`	add a vertical line at 50%?
`reverse_fill`	if `TRUE`, will reverse the default stacking order, see `ggplot2::position_fill()`

Details

You could use `gglikert_data()` to just produce the dataset to be plotted.

If variable labels have been defined (see `labelled::var_label()`), they will be considered. You can also pass custom variables labels with the `variable_labels` argument.

Value

A `ggplot2` plot or a `tibble`.

See Also

`vignette("gglikert")`, `position_likert()`, `stat_prop()`

Examples

```
library(ggplot2)
library(dplyr)

likert_levels <- c(
  "Strongly disagree",
  "Disagree",
  "Neither agree nor disagree",
  "Agree",
  "Strongly agree"
)
set.seed(42)
df <-
  tibble(
    q1 = sample(likert_levels, 150, replace = TRUE),
    q2 = sample(likert_levels, 150, replace = TRUE, prob = 5:1),
    q3 = sample(likert_levels, 150, replace = TRUE, prob = 1:5),
    q4 = sample(likert_levels, 150, replace = TRUE, prob = 1:5),
    q5 = sample(c(likert_levels, NA), 150, replace = TRUE),
    q6 = sample(likert_levels, 150, replace = TRUE, prob = c(1, 0, 1, 1, 0))
  ) %>%
  mutate(across(everything(), ~ factor(.x, levels = likert_levels)))

gglikert(df)

gglikert(df, include = q1:3) +
  scale_fill_likert(pal = scales::brewer_pal(palette = "PRGn"))

gglikert(df, sort = "ascending")


gglikert(df, sort = "ascending", sort_prop_include_center = TRUE)

gglikert(df, sort = "ascending", sort_method = "mean")

gglikert(df, reverse_likert = TRUE)
gglikert(df, add_totals = FALSE, add_labels = FALSE)

gglikert(
  df,
  totals_include_center = TRUE,
  totals_hjust = .25,
  totals_size = 4.5,
  totals_fontface = "italic",
  totals_accuracy = .01,
  labels_accuracy = 1,
  labels_size = 2.5,
  labels_hide_below = .25
)

gglikert(df, exclude_fill_values = "Neither agree nor disagree")

if (require("labelled")) {
  df %>%
    set_variable_labels(
      q1 = "First question",
      q2 = "Second question"
    ) %>%
    gglikert(
      variable_labels = c(
        q4 = "a custom label",
        q6 = "a very very very very very very very very very very long label"
      ),
      y_label_wrap = 25
    )
}

# Facets
df_group <- df
df_group$group <- sample(c("A", "B"), 150, replace = TRUE)

gglikert(df_group, q1:q6, facet_rows = vars(group))

gglikert(df_group, q1:q6, facet_cols = vars(group))

gglikert(df_group, q1:q6, y = "group", facet_rows = vars(.question))

# Custom function to be applied on data
f <- function(d) {
  d$.question <- forcats::fct_relevel(d$.question, "q5", "q2")
  d
}
gglikert(df, include = q1:q6, data_fun = f)

# Custom center
gglikert(df, cutoff = 2)

gglikert(df, cutoff = 1)
gglikert(df, cutoff = 1, symmetric = TRUE)


gglikert_stacked(df, q1:q6)

gglikert_stacked(df, q1:q6, add_median_line = TRUE, sort = "asc")


gglikert_stacked(df_group, q1:q6, y = "group", add_median_line = TRUE) +
  facet_grid(rows = vars(.question))
```

---

ggsurvey *Easy ggplot2 with survey objects*

Description

A function to facilitate ggplot2 graphs using a survey object. It will initiate a ggplot and map survey weights to the corresponding aesthetic.

Usage

```
ggsurvey(design = NULL, mapping = NULL, ...)
```

Arguments

`design`	A survey design object, usually created with `survey::svydesign()`
`mapping`	Default list of aesthetic mappings to use for plot, to be created with `ggplot2::aes()`.
`...`	Other arguments passed on to methods. Not currently used.

Details

Graphs will be correct as long as only weights are required to compute the graph. However, statistic or geometry requiring correct variance computation (like `ggplot2::geom_smooth()`) will be statistically incorrect.

Value

A `ggplot2` plot.

Examples

```
data(api, package = "survey")
dstrat <- survey::svydesign(
  id = ~1, strata = ~stype,
  weights = ~pw, data = apistrat,
  fpc = ~fpc
)
ggsurvey(dstrat) +
  ggplot2::aes(x = cnum, y = dnum) +
  ggplot2::geom_count()

d <- as.data.frame(Titanic)
dw <- survey::svydesign(ids = ~1, weights = ~Freq, data = d)
ggsurvey(dw) +
  ggplot2::aes(x = Class, fill = Survived) +
  ggplot2::geom_bar(position = "fill")
```

hex_bw — *Identify a suitable font color (black or white) given a background HEX color*

Description

Identify a suitable font color (black or white) given a background HEX color

Usage

```
hex_bw(hex_code)
```

Arguments

`hex_code`	Background color in hex-format.

Value

Either black or white, in hex-format

Source

Adapted from `saros`

Examples

```
hex_bw("#0dadfd")
```

`label_number_abs` *Label absolute values*

Description

Label absolute values

Usage

```
label_number_abs(..., hide_below = NULL)

label_percent_abs(..., hide_below = NULL)
```

Arguments

`...`	arguments passed to `scales::label_number()` or `scales::label_percent()`
`hide_below`	if provided, values below `hide_below` will be masked (i.e. an empty string `""` will be returned)

Value

A "labelling" function, , i.e. a function that takes a vector and returns a character vector of same length giving a label for each input value.

See Also

`scales::label_number()`, `scales::label_percent()`

Examples

```
x <- c(-0.2, -.05, 0, .07, .25, .66)

scales::label_number()(x)
label_number_abs()(x)

scales::label_percent()(x)
label_percent_abs()(x)
label_percent_abs(hide_below = .1)(x)
```

`pal_extender` *Extend a discrete colour palette*

Description

If the palette returns less colours than requested, the list of colours will be expanded using `scales::pal_gradient_n()`. To be used with a sequential or diverging palette. Not relevant for qualitative palettes.

Usage

```
pal_extender(pal = scales::brewer_pal(palette = "BrBG"))

scale_fill_extended(
  name = waiver(),
  ...,
  pal = scales::brewer_pal(palette = "BrBG"),
  aesthetics = "fill"
)

scale_colour_extended(
  name = waiver(),
  ...,
  pal = scales::brewer_pal(palette = "BrBG"),
  aesthetics = "colour"
)
```

Arguments

`pal`	A palette function, such as returned by scales::brewer_pal, taking a number of colours as entry and returning a list of colours.
`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`...`	Other arguments passed on to `discrete_scale()` to control name, limits, breaks, labels and so forth.
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the colour and fill aesthetics at the same time, via `aesthetics = c("colour", "fill")`.

Value

A palette function.

Examples

```
pal <- scales::pal_brewer(palette = "PiYG")
scales::show_col(pal(16))
scales::show_col(pal_extender(pal)(16))
```

position_likert *Stack objects on top of each another and center them around 0*

Description

`position_likert()` stacks proportion bars on top of each other and center them around zero (the same number of categories are displayed on each side). This type of presentation is commonly used to display Likert-type scales. `position_likert_count()` uses counts instead of proportions.

Usage

```
position_likert(
  vjust = 1,
  reverse = FALSE,
  exclude_fill_values = NULL,
  cutoff = NULL
)

position_likert_count(
  vjust = 1,
  reverse = FALSE,
  exclude_fill_values = NULL,
  cutoff = NULL
)
```

Arguments

`vjust`	Vertical adjustment for geoms that have a position (like points or lines), not a dimension (like bars or areas). Set to `0` to align with the bottom, `0.5` for the middle, and `1` (the default) for the top.
`reverse`	If `TRUE`, will reverse the default stacking order. This is useful if you're rotating both the plot and legend.
`exclude_fill_values`	Vector of values from the variable associated with the `fill` aesthetic that should not be displayed (but still taken into account for computing proportions)
`cutoff`	number of categories to be displayed negatively (i.e. on the left of the x axis or the bottom of the y axis), could be a decimal value: `2` to display negatively the two first categories, `2.5` to display negatively the two first categories and half of the third, `2.2` to display negatively the two first categories and a fifth of the third (see examples). By default (`NULL`), it will be equal to the number of categories divided by 2, i.e. it will be centered.

Details

It is recommended to use `position_likert()` with `stat_prop()` and its `complete` argument (see examples).

See Also

See `ggplot2::position_stack()` and `ggplot2::position_fill()`

Examples

```
library(ggplot2)

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = "fill") +
  scale_x_continuous(label = scales::label_percent()) +
  xlab("proportion")

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = "likert") +
  scale_x_continuous(label = label_percent_abs()) +
  scale_fill_likert() +
  xlab("proportion")

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = "stack") +
  scale_fill_likert(pal = scales::brewer_pal(palette = "PiYG"))

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = "likert_count") +
  scale_x_continuous(label = label_number_abs()) +
  scale_fill_likert()


# Reverse order -------------------------------------------------------------

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = position_likert(reverse = TRUE)) +
  scale_x_continuous(label = label_percent_abs()) +
  scale_fill_likert() +
  xlab("proportion")

# Custom center -------------------------------------------------------------

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = position_likert(cutoff = 1)) +
  scale_x_continuous(label = label_percent_abs()) +
  scale_fill_likert(cutoff = 1) +
  xlab("proportion")

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = position_likert(cutoff = 3.75)) +
  scale_x_continuous(label = label_percent_abs()) +
  scale_fill_likert(cutoff = 3.75) +
  xlab("proportion")

# Missing items -------------------------------------------------------------
# example with a level not being observed for a specific value of y
d <- diamonds
d <- d[!(d$cut == "Premium" & d$clarity == "I1"), ]
d <- d[!(d$cut %in% c("Fair", "Good") & d$clarity == "SI2"), ]

# by default, the two lowest bar are not properly centered
ggplot(d) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = "likert") +
  scale_fill_likert()

# use stat_prop() with `complete = "fill"` to fix it
ggplot(d) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = "likert", stat = "prop", complete = "fill") +
  scale_fill_likert()

# Add labels ----------------------------------------------------------------

custom_label <- function(x) {
  p <- scales::percent(x, accuracy = 1)
  p[x < .075] <- ""
  p
}

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = "likert") +
  geom_text(
    aes(by = clarity, label = custom_label(after_stat(prop))),
    stat = "prop",
    position = position_likert(vjust = .5)
  ) +
  scale_x_continuous(label = label_percent_abs()) +
  scale_fill_likert() +
  xlab("proportion")

# Do not display specific fill values ---------------------------------------
# (but taken into account to compute proportions)

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = position_likert(exclude_fill_values = "Very Good")) +
  scale_x_continuous(label = label_percent_abs()) +
  scale_fill_likert() +
  xlab("proportion")
```

`round_any` *Round to multiple of any number.*

Description

Round to multiple of any number.

Usage

```
round_any(x, accuracy, f = round)
```

Arguments

`x`	numeric or date-time (POSIXct) vector to round
`accuracy`	number to round to; for POSIXct objects, a number of seconds
`f`	rounding function: `floor`, `ceiling` or `round`

Source

adapted from `plyr`

Examples

```
round_any(1.865, accuracy = .25)
```

`scale_fill_likert` *Colour scale for Likert-type plots*

Description

This scale is similar to other diverging discrete colour scales, but allows to change the "center" of the scale using `cutoff` argument, as used by `position_likert()`.

Usage

```
scale_fill_likert(
  name = waiver(),
  ...,
  pal = scales::brewer_pal(palette = "BrBG"),
  cutoff = NULL,
  aesthetics = "fill"
)

likert_pal(pal = scales::brewer_pal(palette = "BrBG"), cutoff = NULL)
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`...`	Other arguments passed on to `discrete_scale()` to control name, limits, breaks, labels and so forth.
`pal`	A palette function taking a number of colours as entry and returning a list of colours (see examples), ideally a diverging palette
`cutoff`	Number of categories displayed negatively (see `position_likert()`) and therefore changing the center of the colour scale (see examples).
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the colour and fill aesthetics at the same time, via `aesthetics = c("colour", "fill")`.

Examples

```
library(ggplot2)
ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = "likert") +
  scale_x_continuous(label = label_percent_abs()) +
  xlab("proportion")

ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = "likert") +
  scale_x_continuous(label = label_percent_abs()) +
  xlab("proportion") +
  scale_fill_likert()

 ggplot(diamonds) +
  aes(y = clarity, fill = cut) +
  geom_bar(position = position_likert(cutoff = 1)) +
  scale_x_continuous(label = label_percent_abs()) +
  xlab("proportion") +
  scale_fill_likert(cutoff = 1)
```

`signif_stars` *Significance Stars*

Description

Calculate significance stars

Usage

```
signif_stars(x, three = 0.001, two = 0.01, one = 0.05, point = 0.1)
```

Arguments

`x`	numeric values that will be compared to the `point`, `one`, `two`, and `three` values
`three`	threshold below which to display three stars
`two`	threshold below which to display two stars
`one`	threshold below which to display one star
`point`	threshold below which to display one point (`NULL` to deactivate)

Value

Character vector containing the appropriate number of stars for each `x` value.

Author(s)

Joseph Larmarange

Examples

```
x <- c(0.5, 0.1, 0.05, 0.01, 0.001)
signif_stars(x)
signif_stars(x, one = .15, point = NULL)
```

`stat_cross` *Compute cross-tabulation statistics*

Description

Computes statistics of a 2-dimensional matrix using broom::augment.htest.

Usage

```
stat_cross(
  mapping = NULL,
  data = NULL,
  geom = "point",
  position = "identity",
  ...,
  na.rm = TRUE,
  show.legend = NA,
  inherit.aes = TRUE,
  keep.zero.cells = FALSE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	Override the default connection with `ggplot2::geom_point()`.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the

params. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.

- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`na.rm`	If `TRUE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`show.legend`	logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes. It can also be a named logical vector to finely select the aesthetics to display.
`inherit.aes`	If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn't inherit behaviour from the default plot specification, e.g. `borders()`.
`keep.zero.cells`	If `TRUE`, cells with no observations are kept.

Value

A `ggplot2` plot with the added statistic.

Aesthetics

`stat_cross()` requires the **x** and the **y** aesthetics.

Computed variables

observed number of observations in x,y

prop proportion of total

row.prop row proportion

col.prop column proportion

expected expected count under the null hypothesis

resid Pearson's residual

std.resid standardized residual

row.observed total number of observations within row

col.observed total number of observations within column

total.observed total number of observations within the table

phi phi coefficients, see `augment_chisq_add_phi()`

See Also

```
vignette("stat_cross")
```

Examples

```
library(ggplot2)
d <- as.data.frame(Titanic)

# plot number of observations
ggplot(d) +
  aes(x = Class, y = Survived, weight = Freq, size = after_stat(observed)) +
  stat_cross() +
  scale_size_area(max_size = 20)

# custom shape and fill colour based on chi-squared residuals
ggplot(d) +
  aes(
    x = Class, y = Survived, weight = Freq,
    size = after_stat(observed), fill = after_stat(std.resid)
  ) +
  stat_cross(shape = 22) +
  scale_fill_steps2(breaks = c(-3, -2, 2, 3), show.limits = TRUE) +
  scale_size_area(max_size = 20)


# custom shape and fill colour based on phi coeffients
ggplot(d) +
  aes(
    x = Class, y = Survived, weight = Freq,
    size = after_stat(observed), fill = after_stat(phi)
  ) +
  stat_cross(shape = 22) +
  scale_fill_steps2(show.limits = TRUE) +
  scale_size_area(max_size = 20)


# plotting the number of observations as a table
ggplot(d) +
  aes(
    x = Class, y = Survived, weight = Freq, label = after_stat(observed)
  ) +
  geom_text(stat = "cross")

# Row proportions with standardized residuals
ggplot(d) +
  aes(
    x = Class, y = Survived, weight = Freq,
    label = scales::percent(after_stat(row.prop)),
    size = NULL, fill = after_stat(std.resid)
  ) +
  stat_cross(shape = 22, size = 30) +
  geom_text(stat = "cross") +
  scale_fill_steps2(breaks = c(-3, -2, 2, 3), show.limits = TRUE) +
  facet_grid(Sex ~ .) +
  labs(fill = "Standardized residuals") +
  theme_minimal()
```

`stat_prop` *Compute proportions according to custom denominator*

Description

`stat_prop()` is a variation of `ggplot2::stat_count()` allowing to compute custom proportions according to the **by** aesthetic defining the denominator (i.e. all proportions for a same value of **by** will sum to 1). The **by** aesthetic should be a factor. If **by** is not specified, proportions of the total will be computed.

Usage

```
stat_prop(
  mapping = NULL,
  data = NULL,
  geom = "bar",
  position = "fill",
  ...,
  width = NULL,
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE,
  complete = NULL
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	Override the default connection with `ggplot2::geom_bar()`.

`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`width`	Bar width. By default, set to 90% of the `resolution()` of the data.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`orientation`	The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.
`show.legend`	logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes. It can also be a named logical vector to finely select the aesthetics to display.
`inherit.aes`	If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn't inherit behaviour from the default plot specification, e.g. `borders()`.

`complete`	Name (character) of an aesthetic for those statistics should be completed for unobserved values (see example)

Value

A `ggplot2` plot with the added statistic.

Aesthetics

`stat_prop()` understands the following aesthetics (required aesthetics are in bold):

- **x *or* y**
- by (this aesthetic should be a **factor**)
- group
- weight

Computed variables

count number of points in bin

prop computed proportion

See Also

`vignette("stat_prop")`, `ggplot2::stat_count()`. For an alternative approach, see `https://github.com/tidyverse/ggplot2/issues/5505#issuecomment-1791324008`.

Examples

```
library(ggplot2)
d <- as.data.frame(Titanic)

p <- ggplot(d) +
  aes(x = Class, fill = Survived, weight = Freq, by = Class) +
  geom_bar(position = "fill") +
  geom_text(stat = "prop", position = position_fill(.5))
p
p + facet_grid(~Sex)

ggplot(d) +
  aes(x = Class, fill = Survived, weight = Freq) +
  geom_bar(position = "dodge") +
  geom_text(
    aes(by = Survived),
    stat = "prop",
    position = position_dodge(0.9), vjust = "bottom"
  )

if (requireNamespace("scales")) {
  ggplot(d) +
    aes(x = Class, fill = Survived, weight = Freq, by = 1) +
    geom_bar() +
    geom_text(
      aes(label = scales::percent(after_stat(prop), accuracy = 1)),
      stat = "prop",
      position = position_stack(.5)
    )
}

# displaying unobserved levels with complete
d <- diamonds %>%
  dplyr::filter(!(cut == "Ideal" & clarity == "I1")) %>%
  dplyr::filter(!(cut == "Very Good" & clarity == "VS2")) %>%
  dplyr::filter(!(cut == "Premium" & clarity == "IF"))
p <- ggplot(d) +
  aes(x = clarity, fill = cut, by = clarity) +
  geom_bar(position = "fill")
p + geom_text(stat = "prop", position = position_fill(.5))
p + geom_text(stat = "prop", position = position_fill(.5), complete = "fill")
```

stat_weighted_mean *Compute weighted y mean*

Description

This statistic will compute the mean of **y** aesthetic for each unique value of **x**, taking into account **weight** aesthetic if provided.

Usage

```
stat_weighted_mean(
  mapping = NULL,
  data = NULL,
  geom = "point",
  position = "identity",
  ...,
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

mapping	Set of aesthetic mappings created by aes(). If specified and inherit.aes = TRUE (the default), it is combined with the default mapping at the top level of the plot. You must supply mapping if there is no plot mapping.

`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	Override the default connection with `ggplot2::geom_point()`.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`orientation`	The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be

	given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.
`show.legend`	logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes. It can also be a named logical vector to finely select the aesthetics to display.
`inherit.aes`	If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn't inherit behaviour from the default plot specification, e.g. `borders()`.

Value

A `ggplot2` plot with the added statistic.

Computed variables

y weighted y (numerator / denominator)

numerator numerator

denominator denominator

See Also

`vignette("stat_weighted_mean")`

Examples

```
library(ggplot2)

data(tips, package = "reshape")

ggplot(tips) +
  aes(x = day, y = total_bill) +
  geom_point()

ggplot(tips) +
  aes(x = day, y = total_bill) +
  stat_weighted_mean()


ggplot(tips) +
  aes(x = day, y = total_bill, group = 1) +
  stat_weighted_mean(geom = "line")

ggplot(tips) +
  aes(x = day, y = total_bill, colour = sex, group = sex) +
  stat_weighted_mean(geom = "line")

ggplot(tips) +
  aes(x = day, y = total_bill, fill = sex) +
  stat_weighted_mean(geom = "bar", position = "dodge")
```

```
# computing a proportion on the fly
if (requireNamespace("scales")) {
  ggplot(tips) +
    aes(x = day, y = as.integer(smoker == "Yes"), fill = sex) +
    stat_weighted_mean(geom = "bar", position = "dodge") +
    scale_y_continuous(labels = scales::percent)
}

library(ggplot2)

# taking into account some weights
if (requireNamespace("scales")) {
  d <- as.data.frame(Titanic)
  ggplot(d) +
    aes(
      x = Class, y = as.integer(Survived == "Yes"),
      weight = Freq, fill = Sex
    ) +
    geom_bar(stat = "weighted_mean", position = "dodge") +
    scale_y_continuous(labels = scales::percent) +
    labs(y = "Survived")
}
```

`symmetric_limits` *Symmetric limits*

Description

Expand scale limits to make them symmetric around zero. Can be passed as argument to parameter `limits` of continuous scales from packages `{ggplot2}` or `{scales}`. Can be also used to obtain an enclosing symmetric range for numeric vectors.

Usage

```
symmetric_limits(x)
```

Arguments

`x`	a vector of numeric values, possibly a range, from which to compute enclosing range

Value

A numeric vector of length two with the new limits, which are always such that the absolute value of upper and lower limits is the same.

Source

Adapted from the homonym function in `{ggpmisc}`

Examples

```
library(ggplot2)

ggplot(iris) +
  aes(x = Sepal.Length - 5, y = Sepal.Width - 3, colour = Species) +
  geom_vline(xintercept = 0) +
  geom_hline(yintercept = 0) +
  geom_point()

last_plot() +
  scale_x_continuous(limits = symmetric_limits) +
  scale_y_continuous(limits = symmetric_limits)
```

`weighted.median` *Weighted Median and Quantiles*

Description

Compute the median or quantiles a set of numbers which have weights associated with them.

Usage

```
weighted.median(x, w, na.rm = TRUE, type = 2)

weighted.quantile(x, w, probs = seq(0, 1, 0.25), na.rm = TRUE, type = 4)
```

Arguments

`x`	a numeric vector of values
`w`	a numeric vector of weights
`na.rm`	a logical indicating whether to ignore `NA` values
`type`	Integer specifying the rule for calculating the median or quantile, corresponding to the rules available for `stats:quantile()`. The only valid choices are type=1, 2 or 4. See Details.
`probs`	probabilities for which the quantiles should be computed, a numeric vector of values between 0 and 1

Details

The `i`th observation `x[i]` is treated as having a weight proportional to `w[i]`.

The weighted median is a value `m` such that the total weight of data less than or equal to `m` is equal to half the total weight. More generally, the weighted quantile with probability `p` is a value `q` such that the total weight of data less than or equal to `q` is equal to `p` times the total weight.

If there is no such value, then

- if `type = 1`, the next largest value is returned (this is the right-continuous inverse of the left-continuous cumulative distribution function);

- if `type = 2`, the average of the two surrounding values is returned (the average of the right-continuous and left-continuous inverses);
- if `type = 4`, linear interpolation is performed.

Note that the default rule for `weighted.median()` is `type = 2`, consistent with the traditional definition of the median, while the default for `weighted.quantile()` is `type = 4`.

Value

A numeric vector.

Source

These functions are adapted from their homonyms developed by Adrian Baddeley in the `spatstat` package.

Examples

```
x <- 1:20
w <- runif(20)
weighted.median(x, w)
weighted.quantile(x, w)
```

Index