Package ‘tidybayes’

April 4, 2020

Title Tidy Data and ’Geoms’ for Bayesian Models
Version 2.0.3
Date 2020-04-04
Maintainer Matthew Kay <mjskay@umich.edu>

Description
Compose data for and extract, manipulate, and visualize posterior draws from Bayesian models
(‘JAGS’, ‘Stan’, ’rstanarm’, ’brms’, ’MCMCglmm’, ’coda’, ...) in a tidy data format. Func-
tions are provided
to help extract tidy data frames of draws from Bayesian models and that generate point
summaries and intervals in a tidy format. In addition, ’ggplot2’ ‘geoms’ and ’stats’ are provided for
common visualization primitives like points with multiple uncertainty intervals, eye plots (intervals plus
densities), and fit curves with multiple, arbitrary uncertainty bands.

Depends R (>= 3.5.0)
Imports plyr, dplyr (>= 0.8.0), tidyr (>= 1.0.0), ggplot2 (>= 3.1.0),
coda, purrr (>= 0.2.3), rlang (>= 0.3.0), scales, grid,
forcats, HDInterval, arrayhelpers, tidyselect, tibble, magrittr
Suggests knitr, testthat, vdiffri (>= 0.3.0), svglite, ggstance (>=
0.3.2), rstan (>= 2.17.0), runjags, rjags, jagsUI, rstanarm (>=
2.19.2), emmeans, broom (>= 0.4.3), dotwhisker, MCMCglmm,
bayesplot, modelr, brms (>= 2.11.1), cowplot, covr, gdtools,
rmarkdown, ggrepel, bindrcpp, RColorBrewer, gganimate, gifski,
png, transformr

License GPL (>= 3)
Language en-US

BugReports https://github.com/mjskay/tidybayes/issues/new
URL http://mjskay.github.io/tidybayes,
       https://github.com/mjskay/tidybayes
VignetteBuilder knitr
RoxygenNote 7.1.0
LazyData true
R topics documented:

**Encoding** UTF-8

**Collate**
```
``combine_chains.R`` `compare_levels.R`` `compose_data.R``
``fitted_draws.R`` `flip_aes.R`` `gather_draws.R``
``gather_emmeans_draws.R`` `gather_pairs.R`` `gather_variables.R``
``geom_pointinterval.R`` `geom_pointintervalh.R``
``get_variables.R`` `global_variables.R`` `lkjcorr_marginal.R``
``onAttach.R`` `parse_dist.R`` `point_interval.R``
``predict_curve.R`` `predicted_draws.R`` `recover_types.R``
``stat.R`` `stat_slabinterval.R`` `stat_dist_slabinterval.R``
``stat_pointinterval.R`` `stat_pointintervalh.R``
``stat_interval.R`` `stat_intervalh.R`` `stat_lineribbon.R``
``theme_tidybayes.R`` `tidy_draws.R`` `tidy_format_translators.R``
``tidybayes-models.R`` `ungather_draws.R`` `unspread_draws.R``
``util.R`` `x_at_y.R``

**NeedsCompilation** no

**Author** Matthew Kay [aut, cre],
Timothy Mastny [ctb]

**Repository** CRAN

**Date/Publication** 2020-04-04 21:00:03 UTC

R topics documented:

```
tidybayes-package ........................................... 3
add_draws ..................................................... 4
add_fitted_draws ............................................. 5
combine_chains ............................................... 12
compare_levels ............................................... 13
compose_data ................................................ 16
data_list .................................................... 18
density_bins ................................................. 20
gather_draws ................................................ 21
gather_emmeans_draws ....................................... 26
gather_pairs ................................................ 28
gather_variables .......................................... 30
geom_dotsinterval ......................................... 32
geom_eye ..................................................... 39
geom_interval .............................................. 40
geom_lineribbon ........................................... 44
geom_pointinterval ....................................... 46
geom_slabinterval ........................................ 50```
tidybayes-package

Tidy Data and ‘Geoms’ for Bayesian Models

Description

tidybayes is an R package that aims to make it easy to integrate popular Bayesian modeling methods into a tidy data + ggplot workflow.

Details

Tidy data frames (one observation per row) are particularly convenient for use in a variety of R data manipulation and visualization packages (Wickham 2014). However, when using Bayesian modeling functions like JAGS or Stan in R, we often have to translate this data into a form the model understands, and then after running the model, translate the resulting sample (or predictions) into a more tidy format for use with other R functions. tidybayes aims to simplify these two common (often tedious) operations. It also provides a variety of ggplot geometries aimed at making the visualization of model output easier.

For a comprehensive overview of the package, see vignette("tidybayes"). For overviews aimed at the rstanarm and brms packages, see vignette("tidy-rstanarm") and vignette("tidy-brms"). For an overview of the majority of geoms in tidybayes, see vignette("slabinterval").

For a list of supported models, see tidybayes-models.

Index

| tidybayes-package | Tidy Data and ‘Geoms’ for Bayesian Models | 113 |
References


---

**add_draws**

*Add draws to a data frame in tidy format*

**Description**

Add draws from a matrix of draws (usually draws from a predictive distribution) to a data frame in tidy format. This is a generic version of `add_fitted_draws()`/`add_predicted_draws()` that can be used with model types that have their own prediction functions that are not yet supported by tidybayes.

**Usage**

```r
add_draws(data, draws, value = ".value")
```

**Arguments**

- `data` Data frame to add draws to, with M rows.
- `draws` N by M matrix of draws, with M columns corresponding to the M rows in `data`, and N draws in each column.
- `value` The name of the output column; default ".value".

**Details**

Given a data frame with M rows and an N by M matrix of N draws, adds a .row, .draw, and .value column (or another name if value is set) to data, and expands data into a long-format dataframe of draws.

`add_fitted_draws(df,m)` is roughly equivalent to `add_draws(df,posterior_linpred(m,newdata = df,summary = FALSE))`, except that `add_fitted_draws` standardizes argument names and values across packages and has additional features for some model types (like handling ordinal responses and distributional parameters in brms).

`add_predicted_draws(df,m)` is roughly equivalent to `add_draws(df,posterior_predict(m,newdata = df,summary = FALSE))`, except that `add_predicted_draws` standardizes argument names and values across packages.

**Value**

A data frame (actually, a tibble) with a .row column (a factor grouping rows from the input data), a .draw column (a unique index corresponding to each draw from the distribution), and a column with its name specified by the value argument (default is .value) containing the values of draws from draws. The data frame is grouped by all rows in data plus the .row column.
**add_fitted_draws**

Add draws from the posterior fit, predictions, or residuals of a model to a data frame

**Author(s)**

Matthew Kay

**See Also**

`add_fitted_draws()`, `add_predicted_draws()`, `add_draws()`

**Examples**

```r
library(ggplot2)
library(dplyr)

if (require("brms", quietly = TRUE) && require("modelr", quietly = TRUE)) {
  theme_set(theme_light())

  m_mpg = brm(mpg ~ hp * cyl, data = mtcars,
              # 1 chain / few iterations just so example runs quickly
              # do not use in practice
              chains = 1, iter = 500)

  # plot posterior predictive intervals
  mtcars %>%
    group_by(cyl) %>%
    data_grid(hp = seq_range(hp, n = 101)) %>%
    # the line below is equivalent to add_fitted_draws(m_mpg), except that it does not
    # standardize arguments across model types. `summary = FALSE` is not strictly necessary
    # with posterior_linpred(), but because it is necessary on some functions (otherwise
    # those functions return summaries instead of a matrix of draws) it is
    # included in this example.
    add_draws(posterior_linpred(m_mpg, newdata = ., summary = FALSE)) %>%
    ggplot(aes(x = hp, y = mpg, color = ordered(cyl))) +
    stat_lineribbon(aes(y = .value), alpha = 0.25) +
    geom_point(data = mtcars) +
    scale_fill_brewer(palette = "Greys")
}
```
add_fitted_draws

Description

Given a data frame and a model, adds draws from the (possibly transformed) posterior "fit" (aka the linear/link-level predictor), the posterior predictions of the model, or the residuals of a model to the data frame in a long format.

Usage

add_fitted_draws(
  newdata,
  model,
  value = ".value",
  ..., 
  n = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = FALSE,
  scale = c("response", "linear")
)

fitted_draws(
  model,
  newdata,
  value = ".value",
  ..., 
  n = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = FALSE,
  scale = c("response", "linear")
)

add_linpred_draws(
  newdata,
  model,
  value = ".value",
  ..., 
  n = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = FALSE,
  scale = c("response", "linear")
)

linpred_draws(
  model,
add_fitted_draws

```r
add_fitted_draws(newdata, 
  value = "value",
  ..., 
  n = NULL, 
  seed = NULL, 
  re_formula = NULL, 
  category = "category", 
  dpar = FALSE, 
  scale = c("response", "linear")
)

## Default S3 method:
  fitted_draws(model, newdata, ...)

## S3 method for class 'stanreg'
  fitted_draws(
    model, 
    newdata, 
    value = "value",
    ..., 
    n = NULL, 
    seed = NULL, 
    re_formula = NULL, 
    category = "category", 
    dpar = FALSE, 
    scale = c("response", "linear")
  )

## S3 method for class 'brmsfit'
  fitted_draws(
    model, 
    newdata, 
    value = "value",
    ..., 
    n = NULL, 
    seed = NULL, 
    re_formula = NULL, 
    category = "category", 
    dpar = FALSE, 
    scale = c("response", "linear")
  )

add_predicted_draws(
  newdata, 
  model, 
  prediction = "prediction",
  ..., 
  n = NULL,
)```
```r
add_fitted_draws

seed = NULL,
re_formula = NULL,
category = ".category"
)

predicted_draws(
  model,
  newdata,
  prediction = ".prediction",
  ..., 
  n = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category"
)

## Default S3 method:
predicted_draws(model, newdata, ...)

## S3 method for class 'stanreg'
predicted_draws(
  model,
  newdata,
  prediction = ".prediction",
  ..., 
  n = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category"
)

## S3 method for class 'brmsfit'
predicted_draws(
  model,
  newdata,
  prediction = ".prediction",
  ..., 
  n = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category"
)

add_residual_draws(
  newdata,
  model,
  residual = ".residual",
  ..., 
)
Arguments

newdata  Data frame to generate predictions from. If omitted, most model types will generate predictions from the data used to fit the model.

model  A supported Bayesian model fit that can provide fits and predictions. Supported models are listed in the second section of tidybayes-models: Models Supporting Prediction. While other functions in this package (like spread_draws()) support a wider range of models, to work with add_fitted_draws and add_predicted_draws a model must provide an interface for generating predictions, thus more generic Bayesian modeling interfaces like runjags and rstan are not directly supported for these functions (only wrappers around those languages that provide predictions, like rstanarm and brm, are supported here).

value  The name of the output column for fitted_draws; default ".value".

...  Additional arguments passed to the underlying prediction method for the type of model given.

n  The number of draws per prediction / fit to return, or NULL to return all draws.
add_fitted_draws adds draws from (possibly transformed) posterior linear predictors (or "link-level" predictors) to the data. It corresponds to \texttt{rstanarm::posterior_linpred()} in \texttt{rstanarm} or \texttt{brms::fitted.brmsfit()} in \texttt{brms}.

add_predicted_draws adds draws from posterior predictions to the data. It corresponds to \texttt{rstanarm::posterior_predict()} in \texttt{rstanarm} or \texttt{brms::predict.brmsfit()} in \texttt{brms}.

add_fitted_draws and fitted_draws are alternate spellings of the same function with opposite order of the first two arguments to facilitate use in data processing pipelines that start either with a data frame or a model. Similarly, add_predicted_draws and predicted_draws are alternate spellings.

**seed**

A seed to use when subsampling draws (i.e. when \(n\) is not \(\text{NULL}\)).

**re_formula**

A formula containing group-level effects to be considered in the prediction. If \(\text{NULL}\) (default), include all group-level effects; if \(\text{NA}\), include no group-level effects. Some model types (such as \texttt{brms::brmsfit} and \texttt{rstanarm::stanreg-objects}) allow marginalizing over grouping factors by specifying new levels of a factor in \texttt{newdata}. In the case of \texttt{brms::brm()}, you must also pass \texttt{allow_new_levels} = \texttt{TRUE} here to include new levels (see \texttt{brms::predict.brmsfit()}).

**category**

For some ordinal, multinomial, and multivariate models (notably, \texttt{brms::brm()} models but not \texttt{rstanarm::stan_polr()} models), multiple sets of rows will be returned per input row for \texttt{fitted_draws} or \texttt{predicted_draws}, depending on the model type. For ordinal/multinomial models, these rows correspond to different categories of the response variable. For multivariate models, these correspond to different response variables. The \texttt{category} argument specifies the name of the column to put the category names (or variable names) into in the resulting data frame. The default name of this column ("\texttt{.category}" \texttt{vignette("tidy-brms")}) and \texttt{vignette("tidy-rstanarm")} for examples of dealing with output from ordinal models using both approaches.

**dpar**

For \texttt{fitted_draws} and \texttt{add_fitted_draws}: Should distributional regression parameters be included in the output? Valid only for models that support distributional regression parameters, such as submodels for variance parameters (as in \texttt{brm}). If \texttt{TRUE}, distributional regression parameters are included in the output as additional columns named after each parameter (alternative names can be provided using a list or named vector, e.g. \texttt{c(sigma.hat = "sigma"}) would output the "sigma" parameter from a model as a column named "sigma.hat"). If \texttt{FALSE} (the default), distributional regression parameters are not included.

**scale**

Either "response" or "linear". If "response", results are returned on the scale of the response variable. If "linear", fitted values are returned on the scale of the linear predictor.

**prediction**

The name of the output column for \texttt{predicted_draws}; default "\texttt{.prediction}".

**residual**

The name of the output column for \texttt{residual_draws}; default "\texttt{.residual}".

**Details**

add_fitted_draws adds draws from (possibly transformed) posterior linear predictors (or "link-level" predictors) to the data. It corresponds to \texttt{rstanarm::posterior_linpred()} in \texttt{rstanarm} or \texttt{brms::fitted.brmsfit()} in \texttt{brms}.

add_predicted_draws adds draws from posterior predictions to the data. It corresponds to \texttt{rstanarm::posterior_predict()} in \texttt{rstanarm} or \texttt{brms::predict.brmsfit()} in \texttt{brms}.

add_fitted_draws and fitted_draws are alternate spellings of the same function with opposite order of the first two arguments to facilitate use in data processing pipelines that start either with a data frame or a model. Similarly, add_predicted_draws and predicted_draws are alternate spellings.
Given equal choice between the two, `add_fitted_draws` and `add_predicted_draws` are the preferred spellings.

`add_linpred_draws` and `linpred_draws` are alternative spellings of `fitted_draws` and `add_fitted_draws` for consistency with rstanarm terminology (specifically `rstanarm::posterior_linpred()`).

### Value

A data frame (actually, a tibble) with a `.row` column (a factor grouping rows from the input newdata), `.chain` column (the chain each draw came from, or NA if the model does not provide chain information), `.iteration` column (the iteration the draw came from, or NA if the model does not provide iteration information), and a `.draw` column (a unique index corresponding to each draw from the distribution). In addition, `fitted_draws` includes a column with its name specified by the `value` argument (default is `.value`) containing draws from the (transformed) linear predictor, and `predicted_draws` contains a `.prediction` column containing draws from the posterior predictive distribution. For convenience, the resulting data frame comes grouped by the original input rows.

### Author(s)

Matthew Kay

### See Also

`add_draws()` for the variant of these functions for use with packages that do not have explicit support for these functions yet. See `spread_draws()` for manipulating posteriors directly.

### Examples

```r
library(ggplot2)
library(dplyr)

if (
  require("rstanarm", quietly = TRUE) && 
  require("modelr", quietly = TRUE) 
) {
  theme_set(theme_light())

  m_mpg = stan_glm(mpg ~ hp * cyl, data = mtcars, 
  # 1 chain / few iterations just so example runs quickly
  # do not use in practice
  chains = 1, iter = 500)

  # draw 100 fit lines from the posterior and overplot them
  mtcars %>%
    group_by(cyl) %>%
    data_grid(hp = seq_range(hp, n = 101)) %>%
    add_fitted_draws(m_mpg, n = 100) %>%
    ggplot(aes(x = hp, y = mpg, color = ordered(cyl))) + 
      geom_line(aes(y = .value, group = paste(cyl, .draw)), alpha = 0.25) + 
```
```r
geom_point(data = mtcars)

# plot posterior predictive intervals
mtcars %>%
  group_by(cyl) %>%
data_grid(hp = seq_range(hp, n = 101)) %>%
  add_predicted_draws(m_mpg) %>%
ggplot(aes(x = hp, y = mpg, color = ordered(cyl))) +
  stat_lineribbon(aes(y = .prediction), .width = c(.99, .95, .8, .5), alpha = 0.25) +
  geom_point(data = mtcars) +
  scale_fill_brewer(palette = "Greys")
}
```
**compare_levels**

**See Also**

`emmeans::emmeans()`

**Examples**

```r
library(magrittr)
library(coda)

data(line, package = "coda")

# The 'line' posterior has two chains with 200 iterations each:
line %>%
tidy_draws() %>%
summary()

# combine_chains combines the chain and iteration column into the .draw column.
line %>%
tidy_draws() %>%
combine_chains() %>%
summary()
```

---

**compare_levels**  
*Compare the value of draws of some variable from a Bayesian model for different levels of a factor*

**Description**

Given posterior draws from a Bayesian model in long format (e.g. as returned by `spread_draws()`), compare the value of a variable in those draws across different paired combinations of levels of a factor.

**Usage**

```r
compare_levels(
  data,  
  variable,  
  by,  
  fun = "-",  
  comparison = "default",  
  draw_indices = c(".chain", ".iteration", ".draw"),  
  ignore_groups = ".row"
)
```
Arguments

data Long-format data.frame of draws such as returned by `spread_draws()` or `gather_draws()`. If `data` is a grouped data frame, comparisons will be made within groups (if one of the groups in the data frame is the by column, that specific group will be ignored, as it is not possible to make comparisons both within some variable and across it simultaneously).

variable Bare (unquoted) name of a column in data representing the variable to compare across levels.

by Bare (unquoted) name of a column in data that is a factor or ordered. The value of variable will be compared across pairs of levels of this factor.

fun Binary function to use for comparison. For each pair of levels of by we are comparing (as determined by `comparison`), compute the result of this function.

comparison One of (a) the comparison types `ordered`, `control`, `pairwise`, or `default` (may also be given as strings, e.g. "ordered"), see Details; (b) a user-specified function that takes a factor and returns a list of pairs of names of levels to compare (as strings) and/or unevaluated expressions containing representing the comparisons to make; or (c) a list of pairs of names of levels to compare (as strings) and/or unevaluated expressions representing the comparisons to make, e.g.: `list(c("a","b"),c("b","c"))` or `exprs(a ~ b, b ~ c)`, both of which would compare level "a" against "b" and level "b" against "c". Note that the unevaluated expression syntax ignores the `fun` argument, can include any other functions desired (e.g. variable transformations), and can even include more than two levels or other columns in data.

draw_indices Character vector of column names in data that should be treated as indices when making the comparison (i.e. values of variable within each level of by will be compared at each unique combination of levels of `draw_indices`). Columns in `draw_indices` not found in `data` are ignored. The default is `c(".chain",".iteration",".draw")`, which are the same names used for chain/iteration/draw indices returned by `spread_draws()` or `gather_draws()`: thus if you are using `compare_levels` with `spread_draws()` or `gather_draws()` you generally should not need to change this value.

ignore_groups character vector of names of groups to ignore by default in the input grouping. This is primarily provided to make it easier to pipe output of `add_fitted_draws()` into this function, as that function provides a ".row" output column that is grouped, but which is virtually never desired to group by when using `compare_levels`.

Details

This function simplifies conducting comparisons across levels of some variable in a tidy data frame of draws. It applies `fun` to all values of `variable` within each pair of levels of `by` as selected by `comparison`. By default, all pairwise comparisons are generated if `by` is an unordered factor and ordered comparisons are made if `by` is ordered.

The included comparison types are:

- `ordered`: compare each level i with level i - 1; e.g. `fun(i,i -1)`
- `pairwise`: compare each level of `by` with every other level.
compare_levels

- control: compare each level of by with the first level of by. If you wish to compare with a different level, you can first apply `relevel()` to by to set the control (reference) level.
- default: use ordered if `is.ordered(by)` and pairwise otherwise.

**Value**

A data.frame with the same columns as `data`, except that the `by` column contains a symbolic representation of the comparison of pairs of levels of `by` in `data`, and `variable` contains the result of that comparison.

**Author(s)**

Matthew Kay

**See Also**

`spread_draws()` and `gather_draws()`.

**Examples**

```r
library(dplyr)
library(ggplot2)

data(RankCorr, package = "tidybayes")

# Let's do all pairwise comparisons of b[i,1]:
RankCorr %>%
  spread_draws(b[i,j]) %>%
  filter(j == 1) %>%
  compare_levels(b, by = i) %>%
  median_qi()

# Or let's plot all comparisons against the first level (control):
RankCorr %>%
  spread_draws(b[i,j]) %>%
  filter(j == 1) %>%
  compare_levels(b, by = i, comparison = control) %>%
  ggplot(aes(x = b, y = i)) +
  stat_halfeyeh()

# Or let's plot comparisons of all levels of j within # all levels of i
RankCorr %>%
  spread_draws(b[i,j]) %>%
  group_by(i) %>%
  compare_levels(b, by = j) %>%
  ggplot(aes(x = b, y = j)) +
  stat_halfeyeh() +
  facet_grid(cols = vars(i))
```
compose_data

Compose data for input into a Bayesian model

Description

Compose data into a list suitable to be passed into a Bayesian model (JAGS, BUGS, Stan, etc).

Usage

compose_data(..., .n_name = n_prefix("n"))

Arguments

... Data to be composed into a list suitable for being passed into Stan, JAGS, etc. Named arguments will have their name used as the name argument to as_data_list when translated; unnamed arguments that are not lists or data frames will have their bare value (passed through make.names) used as the name argument to as_data_list. Each argument is evaluated using eval_tidy in an environment that includes all list items composed so far.

.n_name A function that is used to form dimension index variables (a variable whose value is number of levels in a factor or the length of a data frame in ...). For example, if a data frame with 20 rows and a factor "foo" (having 3 levels) is passed to compose_data, the list returned by compose_data will include an element named .n_name("foo"), which by default would be "n_foo", containing the value 3, and a column named "n" containing the value 20. See n_prefix().

Details

This function recursively translates each argument into list elements using as_data_list(), merging all resulting lists together. By default this means that:

- numerics are included as-is.
- logicals are translated into numeric using as.numeric().
- factors are translated into numeric using as.numeric() and an additional element named .n_name(argument_name) is added with the number of levels in the factor. The default .n_name function prefixes "n_" before the factor name; e.g. a factor named foo will have an element named n_foo added containing the number of levels in foo.
- character vectors are converted into factors then translated into numeric in the same manner as factors are.
- lists are translated by translating all elements of the list (recursively) and adding them to the result.
- data.frames are translated by translating every column of the data.frame and adding them to the result. A variable named "n" (or .n_name(argument_name) if the data.frame is passed as a named argument argument_name) is also added containing the number of rows in the data frame.
compose_data

- NULL values are dropped. Setting a named argument to NULL can be used to drop that item from the resulting list (if an unwanted element was added to the list by a previous argument, such as a column from a data frame that is not needed in the model).
- all other types are dropped (and a warning given)

As in functions like `mutate()`, each expression is evaluated in an environment containing the data list built up so far.

For example, this means that if the first argument to `compose_data` is a data frame, subsequent arguments can include direct references to columns from that data frame. This allows you, for example, to easily use `x_at_y()` to generate indices for nested models.

If you wish to add support for additional types not described above, provide an implementation of `as_data_list()` for the type. See the implementations of `as_data_list.numeric`, `as_data_list.logical`, etc for examples.

**Value**

A list where each element is a translated variable as described above.

**Author(s)**

Matthew Kay

**See Also**

`x_at_y()`, `spread_draws()`, `gather_draws()`.

**Examples**

```r
library(magrittr)

df = data.frame(
  plot = factor(paste0("p", rep(1:8, times = 2))),
  site = factor(paste0("s", rep(1:4, each = 2, times = 2)))
)

# without changing `.n_name`, compose_data() will prefix indices
# with "n" by default
df %>%
  compose_data()

# you can use n_prefix() to define a different prefix (e.g. "N"):
df %>%
  compose_data(.n_name = n_prefix("N"))

# If you have nesting, you may want a nested index, which can be generated using x_at_y()
# Here, site[p] will give the site for plot p
df %>%
  compose_data(site = x_at_y(site, plot))
```
Data lists for input into Bayesian models

Description

Functions used by `compose_data()` to create lists of data suitable for input into a Bayesian modeling function. These functions typically should not be called directly (instead use `compose_data()`), but are exposed for the rare cases in which you may need to provide your own conversion routines for a data type not already supported (see Details).

Usage

```r
data_list(...)

as_data_list(object, name = "", ...)  
## Default S3 method:
as_data_list(object, name = "", ...)

## S3 method for class 'numeric'
as_data_list(object, name = "", scalar_as_array = FALSE, ...)

## S3 method for class 'logical'
as_data_list(object, name = "", ...)  

## S3 method for class 'factor'
as_data_list(object, name = "", .n_name = n_prefix("n"), ...)

## S3 method for class 'character'
as_data_list(object, name = "", ...)  

## S3 method for class 'list'
as_data_list(object, name = "", ...)

## S3 method for class 'data.frame'
as_data_list(object, name = "", .n_name = n_prefix("n"), ...)

## S3 method for class 'data_list'
as_data_list(object, name = "", ...)  
```

Arguments

- `...`: Additional arguments passed to other implementations of `as_data_list`, or for `data_list`, passed to `list()`.
- `object`: The object to convert (see Details).
- `name`: The name of the element in the returned list corresponding to this object.
**data_list**

**scalar_as_array**  
If TRUE, returns single scalars as an 1-dimensional array with one element. This is used by `as_data_list.data.frame` to ensure that columns from a data frame with only one row are still returned as arrays instead of scalars.

**.n_name**  
A function that is used to form variables storing the number of rows in data frames or the number of levels in factors in ...). For example, if a factor with `name = "foo"` (having three levels) is passed in, the list returned will include an element named `.n_name("foo")`, which by default would be "n_foo", containing the value 3.

**Details**

data_list creates a list with class `c("data_list","list")` instead of `c("list")`, but largely otherwise acts like the `list()` function.

as_data_list recursively translates its first argument into list elements, concatenating all resulting lists together. By default this means that:

- numerics are included as-is.
- logicals are translated into numeric using `as.numeric()`.
- factors are translated into numeric using `as.numeric()`, and an additional element named `.n_name(name)` is added with the number of levels in the factor.
- character vectors are converted into factors then translated into numeric in the same manner as factors are.
- lists are translated by translating all elements of the list (recursively) and adding them to the result.
- data.frames are translated by translating every column of the data.frame and adding them to the result. A variable named "n" (or `.n_name(name)` if name is not "") is also added containing the number of rows in the data frame.
- all other types are dropped (and a warning given)

If you wish to add support for additional types not described above, provide an implementation of `as_data_list()` for the type. See the implementations of `as_data_list.numeric`, `as_data_list.logical`, etc for examples.

**Value**

An object of class `c("data_list","list")`, where each element is a translated variable as described above.

**Author(s)**

Matthew Kay

**See Also**

`compose_data()`.
**density_bins**

*Density bins and histogram bins as data frames*

**Description**

Generates a data frame of bins representing the kernel density (or histogram) of a vector, suitable for use in generating predictive distributions for visualization. These functions were originally designed for use with the now-deprecated `predict_curve()`, and may be deprecated in the future.

**Usage**

```r
density_bins(x, n = 101, ...)
```

```r
histogram_bins(x, n = 30, breaks = n, ...)
```

**Arguments**

- `x`  
  A numeric vector
- `n`  
  Number of bins
- `...`  
  Additional arguments passed to `density()` or `hist()`.
- `breaks`  
  Used to set bins for `histogram_bins`. Can be number of bins (by default it is set to the value of `n`) or a method for setting bins. See the `breaks` argument of `hist()`.

**Details**

These functions are simple wrappers to `density()` and `hist()` that compute density estimates and return their results in a consistent format: a data frame of bins suitable for use with the now-deprecated `predict_curve()`.

- `density_bins` computes a kernel density estimate using `density()`.
- `histogram_bins` computes a density histogram using `hist()`.

**Value**

A data frame representing bins and their densities with the following columns:

- `mid`  Bin midpoint
- `lower`  Lower endpoint of each bin
- `upper`  Upper endpoint of each bin
- `density`  Density estimate of the bin

**Examples**

```r
# Typically these functions should not be used directly.
# See the compose_data function for examples of how to translate
# data in lists for input to Bayesian modeling functions.
```
Author(s)

Matthew Kay

See Also

See `add_predicted_draws()` and `stat_lineribbon()` for a better approach. These functions may be deprecated in the future.

Examples

```r
library(ggplot2)
library(dplyr)
library(purrr)
library(tidyr)

if (require("rstanarm", quietly = TRUE) && require("modelr", quietly = TRUE)) {

  theme_set(theme_light())

  m_mpg = stan_glm(mpg ~ hp * cyl, data = mtcars)

  step = 1

  mtcars %>%
    group_by(cyl) %>%
    data_grid(hp = seq_range(hp, by = step)) %>%
    add_predicted_draws(m_mpg) %>%
    summarise(densities = list(density_bins(.prediction))) %>%
    unnest(densities) %>%
    ggplot() +
    geom_rect(aes(
      xmin = hp - step/2, ymin = lower, ymax = upper, xmax = hp + step/2,
      fill = ordered(cyl), alpha = density
    )) +
    geom_point(aes(x = hp, y = mpg, fill = ordered(cyl)), shape = 21, data = mtcars) +
    scale_alpha_continuous(range = c(0, 1)) +
    scale_fill_brewer(palette = "Set2")
}
```

---

gather_draws

Extract draws of variables in a Bayesian model fit into a tidy data format
gather_draws

Description

Extract draws from a Bayesian model for one or more variables (possibly with named dimensions) into one of two types of long-format data frames.

Usage

```r
gather_draws(model, ..., regex = FALSE, sep = "[, ]", n = NULL, seed = NULL)
```

```r
spread_draws(model, ..., regex = FALSE, sep = "[, ]", n = NULL, seed = NULL)
```

Arguments

- `model`: A supported Bayesian model fit. Tidybayes supports a variety of model objects; for a full list of supported models, see `tidybayes-models`.
- `...`: Expressions in the form of `variable_name[dimension_1,dimension_2,...]` | `wide_dimension`. See Details.
- `regex`: If TRUE, variable names are treated as regular expressions and all column matching the regular expression and number of dimensions are included in the output. Default FALSE.
- `sep`: Separator used to separate dimensions in variable names, as a regular expression.
- `n`: The number of draws to return, or NULL to return all draws.
- `seed`: A seed to use when subsampling draws (i.e. when `n` is not NULL).

Details

Imagine a JAGS or Stan fit named `fit`. The model may contain a variable named `b[i,v]` (in the JAGS or Stan language) with dimension `i` in `1:100` and dimension `v` in `1:3`. However, the default format for draws returned from JAGS or Stan in R will not reflect this indexing structure, instead they will have multiple columns with names like "b[1,1]", "b[2,1]", etc.

`spread_draws` and `gather_draws` provide a straightforward syntax to translate these columns back into properly-indexed variables in two different tidy data frame formats, optionally recovering dimension types (e.g. factor levels) as it does so.

`spread_draws` and `gather_draws` return data frames already grouped by all dimensions used on the variables you specify.

The difference between `spread_draws` is that names of variables in the model will be spread across the data frame as column names, whereas `gather_draws` will gather variables into a single column named ".variable" and place values of variables into a column named ".value". To use naming schemes from other packages (such as `broom`), consider passing results through functions like `to_broom_names()` or `to_ggmc_names()`.

For example, `spread_draws(fit,a[i],b[i,v])` might return a grouped data frame (grouped by `i` and `v`), with:

- column ".chain": the chain number. NA if not applicable to the model type; this is typically only applicable to MCMC algorithms.
- column ".iteration": the iteration number. Guaranteed to be unique within-chain only. NA if not applicable to the model type; this is typically only applicable to MCMC algorithms.
• column ".draw": a unique number for each draw from the posterior. Order is not guaranteed to be meaningful.
• column "i": value in 1:5
• column "v": value in 1:10
• column "a": value of "a[i]" for draw ".draw"
• column "b": value of "b[i,v]" for draw ".draw"

gather_draws(fit,a[i],b[i,v]) on the same fit would return a grouped data frame (grouped by i and v), with:
• column ".chain": the chain number
• column ".iteration": the iteration number
• column ".draw": the draw number
• column "i": value in 1:5
• column "v": value in 1:10, or NA if ".variable" is "a".
• column ".variable": value in c("a", "b").
• column ".value": value of "a[i]" (when ".variable" is "a") or "b[i,v]" (when ".variable" is "b") for draw ".draw"

spread_draws and gather_draws can use type information applied to the fit object by recover_types() to convert columns back into their original types. This is particularly helpful if some of the dimensions in your model were originally factors. For example, if the v dimension in the original data frame data was a factor with levels c("a", "b", "c"), then we could use recover_types before spread_draws:

fit %>%
  recover_types(data)
  spread_draws(fit, b[i,v])

Which would return the same data frame as above, except the "v" column would be a value in c("a", "b", "c") instead of 1:3.

For variables that do not share the same subscripts (or share some but not all subscripts), we can supply their specifications separately. For example, if we have a variable d[i] with the same i subscript as b[i,v], and a variable x with no subscripts, we could do this:

spread_draws(fit, x, d[i], b[i,v])

Which is roughly equivalent to this:

spread_draws(fit, x) %>%
  inner_join(spread_draws(fit, d[i])) %>%
  inner_join(spread_draws(fit, b[i,v])) %>%
  group_by(i, v)

Similarly, this:
gather_draws(fit, x, d[i], b[i,v])

Is roughly equivalent to this:

bind_rows(
    gather_draws(fit, x),
    gather_draws(fit, d[i]),
    gather_draws(fit, b[i,v])
)

The `c` and `cbind` functions can be used to combine multiple variable names that have the same dimensions. For example, if we have several variables with the same subscripts `i` and `v`, we could do either of these:

spread_draws(fit, c(w, x, y, z)[i,v])

spread_draws(fit, cbind(w, x, y, z)[i,v])  # equivalent

Each of which is roughly equivalent to this:

spread_draws(fit, w[i,v], x[i,v], y[i,v], z[i,v])

Besides being more compact, the `c()`-style syntax is currently also faster (though that may change).

Dimensions can be omitted from the resulting data frame by leaving their names blank; e.g. `spread_draws(fit, b[,v])` will omit the first dimension of `b` from the output. This is useful if a dimension is known to contain all the same value in a given model.

The shorthand `..` can be used to specify one column that should be put into a wide format and whose names will be the base variable name, plus a dot (`"."`), plus the value of the dimension at `..`.

For example:

spread_draws(fit, b[i,..]) would return a grouped data frame (grouped by `i`), with:

- column `".chain"`: the chain number
- column `".iteration"`: the iteration number
- column `".draw"`: the draw number
- column `"i"`: value in `1:20`
- column `"b.1"`: value of "b[i,1]" for draw ".draw"
- column `"b.2"`: value of "b[i,2]" for draw ".draw"
- column `"b.3"`: value of "b[i,3]" for draw ".draw"

An optional clause in the form `| wide_dimension` can also be used to put the data frame into a wide format based on `wide_dimension`. For example, this:

spread_draws(fit, b[i,v] | v)

is roughly equivalent to this:
`spread_draws(fit, b[i,v]) %>% spread(v, b)`

The main difference between using the `|` syntax instead of the `.` syntax is that the `|` syntax respects prototypes applied to dimensions with `recover_types()`, and thus can be used to get columns with nicer names. For example:

```r
fit %>% recover_types(data) %>% spread_draws(b[i,v] | v)
```

would return a grouped data frame (grouped by `i`), with:

- column `.chain`: the chain number
- column `.iteration`: the iteration number
- column `.draw`: the draw number
- column `i`: value in `1:20`
- column `a`: value of `b[i,1]` for draw `.draw`
- column `b`: value of `b[i,2]` for draw `.draw`
- column `c`: value of `b[i,3]` for draw `.draw`

The shorthand `.` can be used to specify columns that should be nested into vectors, matrices, or n-dimensional arrays (depending on how many dimensions are specified with `.`). For example, `spread_draws(fit,a[.],b[.,.])` might return a data frame, with:

- column `.chain`: the chain number.
- column `.iteration`: the iteration number.
- column `.draw`: a unique number for each draw from the posterior.
- column `a`: a list column of vectors.
- column `b`: a list column of matrices.

Ragged arrays are turned into non-ragged arrays with missing entries given the value `NA`. Finally, variable names can be regular expressions by setting `regex = TRUE`; e.g.:

```r
spread_draws(fit, 'b_.*'[i], regex = TRUE)
```

Would return a tidy data frame with variables starting with `b_` and having one dimension.

**Value**

A data frame.

**Author(s)**

Matthew Kay

**See Also**

`recover_types()`, `compose_data()`.
Examples

```r
library(dplyr)
library(ggplot2)

data(RankCorr, package = "tidybayes")

RankCorr %>%
  spread_draws(b[i, j])

RankCorr %>%
  spread_draws(b[i, j], tau[i], u_tau[i])

RankCorr %>%
  gather_draws(b[i, j], tau[i], u_tau[i])

RankCorr %>%
  gather_draws(tau[i], typical_r) %>%
  median_qi()
```

gather_emmeans_draws

Extract a tidy data frame of draws of posterior distributions of "estimated marginal means" (emmeans/lsmeans) from a Bayesian model fit.

Description

Extract draws from the result of a call to `emmeans::emmeans()` (formerly `lsmeans`) or `emmeans::ref_grid()` applied to a Bayesian model.

Usage

```r
gather_emmeans_draws(object, value = ".value", ...)
```

## Default S3 method:
gather_emmeans_draws(object, value = ".value", ...)

## S3 method for class 'emm_list'
gather_emmeans_draws(object, value = ".value", grid = ".grid", ...)

Arguments

- `object`: An `emmGrid` object such as returned by `emmeans::ref_grid()` or `emmeans::emmeans()`.
- `value`: The name of the output column to use to contain the values of draws. Defaults to ".value".
gather_emmeans_draws

Additional arguments passed to the underlying method for the type of object given.

grid

If object is an `emmeans::emm_list()`, the name of the output column to use to contain the name of the reference grid that a given row corresponds to. Defaults to ".grid".

Details

`emmeans::emmeans()` provides a convenient syntax for generating draws from "estimated marginal means" from a model, and can be applied to various Bayesian models, like `rstanarm::stanreg-objects` and `MCMCglmm::MCMCglmm()`. Given a `emmeans::ref_grid()` object as returned by functions like `emmeans::ref_grid()` or `emmeans::emmeans()` applied to a Bayesian model, `gather_emmeans_draws` returns a tidy format data frame of draws from the marginal posterior distributions generated by `emmeans::emmeans()`.

Value

A tidy data frame of draws. The columns of the reference grid are returned as-is, with an additional column called `.value` (by default) containing marginal draws. The resulting data frame is grouped by the columns from the reference grid to make use of summary functions like `point_interval()` straightforward.

If object is an `emmeans::emm_list()`, which contains estimates from different reference grids, an additional column with the default name of ".grid" is added to indicate the reference grid for each row in the output. The name of this column is controlled by the `grid` argument.

Author(s)

Matthew Kay

See Also

`emmeans::emmeans()`

Examples

```r
library(dplyr)
library(magrittr)

if (require("rstanarm", quietly = TRUE) &
    require("emmeans", quietly = TRUE)) {

  # Here's an example dataset with a categorical predictor ('condition') with several levels:
  set.seed(5)
  n = 10
  n_condition = 5
  ABC =
```

```r
tibble(
  condition = rep(c("A","B","C","D","E"), n),
  response = rnorm(n * 5, c(0,1,2,1,-1), 0.5)
)

m = stan_glm(response ~ condition, data = ABC,
  # 1 chain / few iterations just so example runs quickly
  # do not use in practice
  chains = 1, iter = 500)

# Once we've fit the model, we can use emmeans() (and functions
# from that package) to get whatever marginal distributions we want.
# For example, we can get marginal means by condition:
  m %>%
    emmeans(~ condition) %>%
    gather_emmeans_draws() %>%
    median_qi()

# or we could get pairwise differences:
  m %>%
    emmeans(~ condition) %>%
    contrast(method = "pairwise") %>%
    gather_emmeans_draws() %>%
    median_qi()

  # see the documentation of emmeans() for more examples of types of
  # contrasts supported by that package.
}
```

gather_pairs  Gather pairwise combinations of values from key/value columns in a long-format data frame

### Description

Fast method for producing combinations of values in a value column for different levels of a key column, assuming long-format (tidy) data with an equal number of values per key. Among other things, this is useful for producing scatter-plot matrices.

### Usage

```r
gather_pairs(
  data,
  key,
  value,
  row = ".row",
  col = ".col",
  x = ".x",
```
y = ".y",
triangle = c("lower only", "upper only", "lower", "upper", "both only", "both")
)

Arguments

data Tidy data frame.
key Bare name of column in data containing the key.
value Bare name of column in data containing the value.
row Character vector giving the name of the output column identifying rows in the matrix of pairs (takes values of key).
col Character vector giving the name of the output column identifying columns in the matrix of pairs (takes values of key).
x Character vector giving the name of the output column with x values in the matrix of pairs (takes values of value).
y Character vector giving the name of the output column with y values in the matrix of pairs (takes values of value).
triangle Should the upper or lower triangle of the matrix of all possible combinations be returned? The default, "lower only", returns the lower triangle without the diagonal; "lower" returns the lower triangle with the diagonal ("upper" and "upper only" operate analogously), "both" returns the full set of possible combinations, and "both only" returns all combinations except the diagonal.

This method is particularly useful for constructing scatterplot matrices. See examples below.

Value

A tidy data frame of combinations of values in key and value, with columns row and col (default names ".row" and ".col") containing values from key, and columns y and x (default names ".y" and ".x") containing values from value.

Author(s)

Matthew Kay

See Also

emmeans::emmeans()

Examples

library(ggplot2)
library(dplyr)

t_a = rnorm(100)
t_b = rnorm(100, t_a * 2)
t_c = rnorm(100)
```r
df = rbind(
  data.frame(g = "a", t = t_a),
  data.frame(g = "b", t = t_b),
  data.frame(g = "c", t = t_c)
)

df %>%
gather_pairs(g, t, row = "g_row", col = "g_col", x = "t_x", y = "t_y") %>%
  ggplot(aes(t_x, t_y)) +
  geom_point() +
  facet_grid(vars(g_row), vars(g_col))

df %>%
gather_pairs(g, t, triangle = "upper") %>%
  ggplot(aes(.x, .y)) +
  geom_point() +
  facet_grid(vars(.row), vars(.col))

df %>%
gather_pairs(g, t, triangle = "both") %>%
  ggplot(aes(.x, .y)) +
  geom_point() +
  facet_grid(vars(.row), vars(.col))

data(line, package = "coda")

line %>%
tidy_draws() %>%
gather_variables() %>%
gather_pairs(.variable, .value) %>%
  ggplot(aes(.x, .y)) +
  geom_point(alpha = .25) +
  facet_grid(vars(.row), vars(.col))

line %>%
tidy_draws() %>%
gather_variables() %>%
gather_pairs(.variable, .value) %>%
  ggplot(aes(.x, .y, color = factor(.chain))) +
  geom_density_2d(alpha = .5) +
  facet_grid(vars(.row), vars(.col))
```

**gather_variables**

Gather variables from a tidy data frame of draws from variables into a single column
**Description**

Given a data frame such as might be returned by tidy_draws() or spread_draws(), gather variables and their values from that data frame into a ".variable" and ".value" column.

**Usage**

```r
gather_variables(data, exclude = c(".chain", ".iteration", ".draw", ".row"))
```

**Arguments**

- `data`: A data frame with variable names spread across columns, such as one returned by tidy_draws() or spread_draws().
- `exclude`: A character vector of names of columns to be excluded from the gather. Default ignores several meta-data column names used in tidybayes.

**Details**

This function gathers every column except grouping columns and those matching the expression `exclude` into key/value columns ".variable" and ".value".

Imagine a data frame `data` as returned by spread_draws(fit,a[i],b[i,v]), like this:

- column ".chain": the chain number
- column ".iteration": the iteration number
- column ".draw": the draw number
- column ".i": value in 1:5
- column ".v": value in 1:10
- column "a": value of "a[i]" for draw number ".draw"
- column "b": value of "b[i,v]" for draw number ".draw"

`gather_variables(data)` on that data frame would return a grouped data frame (grouped by i and v), with:

- column ".chain": the chain number
- column ".iteration": the iteration number
- column ".draw": the draw number
- column ".i": value in 1:5
- column ".v": value in 1:10
- column ".variable": value in c("a", "b").
- column ".value": value of "a[i]" (when ".variable" is "a"; repeated for every value of ".v") or "b[i,v]" (when ".variable" is "b") for draw number ".draw"

In this example, this call:

```r
gather_variables(data)
```

Is roughly equivalent to:
data %>%
  gather(.variable, .value, -c(.chain, .iteration, .draw, i, v)) %>%
group_by(.variable, .add = TRUE)

Value
A data frame.

Author(s)
Matthew Kay

See Also
spread_draws(), tidy_draws().

Examples

library(dplyr)

data(RankCorr, package = "tidybayes")

RankCorr %>%
  spread_draws(b[i,v], tau[i]) %>%
gather_variables() %>%
median_qi()

# the first three lines below are roughly equivalent to ggmcmc::ggs(RankCorr)
RankCorr %>%
  tidy_draws() %>%
gather_variables() %>%
median_qi()

geom_dotsinterval
Automatic dotplots, dots + intervals, and quantile dotplots (ggplot geom)

Description
Geoms and stats for creating dotplots that automatically determines a bin width that ensures the plot fits within the available space. Also ensures dots do not overlap, and allows generation of quantile dotplots using the quantiles argument to stat_dotsinterval/stat_dots and stat_dist_dotsinterval/stat_dist_dots. Generally follows the naming scheme and arguments of the geom_slabinterval() and stat_slabinterval() family of geoms and stats.
Usage

geom_dotsinterval(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ..., 
  dotsize = 1,
  stackratio = 1,
  binwidth = NA,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_dotsintervalh(..., orientation = "horizontal")

geom_dots(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ..., 
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_dotsh(..., orientation = "horizontal")

stat_dotsinterval(
  mapping = NULL,
  data = NULL,
  geom = "dotsinterval",
  position = "identity",
  ..., 
  quantiles = NA,
  point_interval = median_qi,
  na.rm = FALSE,
  show.legend = c(size = FALSE),
  inherit.aes = TRUE
)

stat_dotsintervalh(..., orientation = "horizontal")

stat_dots(
  mapping = NULL,
  data = NULL,
Arguments

mapping  Set of aesthetic mappings created by `aes()` or `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. \( \sim \text{head}(x, 10) \)).

**stat**

The statistical transformation to use on the data for this layer, as a string.

**position**

Position adjustment, either as a string, or the result of a call to a position adjustment function.

... Arguments passed on to \texttt{geom_slabinterval}

**side**

Which side to draw the slab on. "topright", "top", and "right" are synonyms which cause the slab to be drawn on the top or the right depending on if orientation is "horizontal" or "vertical". "bottomleft", "bottom", and "left" are synonyms which cause the slab to be drawn on the bottom of the left depending on if orientation is "horizontal" or "vertical". "both" draws the slab mirrored on both sides (as in a violin plot).

**scale**

What proportion of the region allocated to this geom to use to draw the slab. If scale = 1, slabs that use the maximum range will just touch each other. Default is 0.9 to leave some space.

**justification**

Justification of the interval relative to the slab, where 0 indicates bottom/left justification and 1 indicates top/right justification (depending on orientation). If justification is NULL (the default), then it is set automatically based on the value of side: when side is "top"/"right" justification is set to 0, when side is "bottom"/"left" justification is set to 1, and when side is "both" justification is set to 0.5.

**normalize**

How to normalize heights of functions input to the thickness aesthetic. If "all" (the default), normalize so that the maximum height across all data is 1; if "panels", normalize within panels so that the maximum height in each panel is 1; if "xy", normalize within the x/y axis opposite the orientation of this geom so that the maximum height at each value of the opposite axis is 1; if "groups", normalize within values of the opposite axis and within groups so that the maximum height in each group is 1; if "none", values are taken as is with no normalization (this should probably only be used with functions whose values are in \([0,1]\), such as CDFs).

**interval_size_domain**

The minimum and maximum of the values of the size aesthetic that will be translated into actual sizes for intervals drawn according to interval_size_range (see the documentation for that argument.)

**interval_size_range**

This geom scales the raw size aesthetic values when drawing interval and point sizes, as they tend to be too thick when using the default settings of \texttt{scale_size_continuous()}, which give sizes with a range of \(c(1,6)\). The interval_size_domain value indicates the input domain of raw size values (typically this should be equal to the value of the range argument of the \texttt{scale_size_continuous()} function), and interval_size_range indicates the desired output range of the size values (the min and max of the actual sizes used to draw intervals).

**fatten_point**

A multiplicative factor used to adjust the size of the point relative to the size of the thickest interval line. If you wish to specify point sizes directly, you can also use the point_size aesthetic and \texttt{scale_point_size_continuous()} or \texttt{scale_point_size_discrete()}; sizes specified with that aesthetic will not be adjusted using fatten_point.
show_slab  Should the slab portion of the geom be drawn? Default TRUE.
show_point   Should the point portion of the geom be drawn? Default TRUE.
show_interval Should the interval portion of the geom be drawn? Default TRUE.

dotsize     The size of the dots relative to the bin width. The default, 1, makes dots be just about as wide as the bin width.
stackratio  The distance between the center of the dots in the same stack relative to the bin height. The default, 1, makes dots in the same stack just touch each other.
binwidth    The bin width to use for drawing the dotplots. The default value, NA, will dynamically select a bin width based on the size of the plot when drawn.
na.rm       If FALSE, 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().
orientation Whether this geom is drawn horizontally ("horizontal") or vertically ("vertical"). When horizontal (resp. vertical), the geom uses the y (resp. x) aesthetic to identify different groups, then for each group uses the x (resp. y) aesthetic and the thickness aesthetic to draw a function as an slab, and draws points and intervals horizontally (resp. vertically) using the xmin, x, and xmax (resp. ymin, y, and ymax) aesthetics.
geom        Use to override the default connection between stat_slabinterval and geom_slabinterval()
quantiles   For the stat_ and stat_dist_ stats, setting this to a value other than NA will produce a quantile dotplot: that is, a dotplot of quantiles from the sample (for stat_) or a dotplot of quantiles from the distribution (for stat_dist_). The value of quantiles determines the number of quantiles to plot. See Kay et al. (2016) and Fernandes et al. (2018) for more information on quantile dotplots.
point_interval A function from the point_interval() family (e.g., median_qi, mean_qi, etc). This function should take in a vector of value, and should obey the .width and .simple_names parameters of point_interval() functions, such that when given a vector with .simple_names = TRUE should return a data frame with variables .value, .lower, .upper, and .width. Output will be converted to the appropriate x- or y-based aesthetics depending on the value of orientation. See the point_interval() family of functions for more information.

details
The dots geoms are similar to geom_dotplot() but with a number of differences:

- Dots geoms act like slabs in geom_slabinterval() and can be given x positions (or y positions when in a horizontal orientation).
- Given the available space to lay out dots, the dots geoms will automatically determine how many bins to use to fit the available space.
• Dots geoms use a dynamic layout algorithm that lays out dots from the center out if the input data are symmetrical, guaranteeing that symmetrical data results in a symmetrical plot. The layout algorithm also prevents dots from overlapping each other.

• The shape of the dots in these geoms can be changed using the slab_shape aesthetic (when using the dotsinterval family) or the shape or slab_shape aesthetic (when using the dots family).

The stat_... and stat_dist_... versions of the stats when used with the quantiles argument are particularly useful for constructing quantile dotplots, which can be an effective way to communicate uncertainty using a frequency framing that may be easier for laypeople to understand (Kay et al. 2016, Fernandes et al. 2018).

Aesthetics

These stats support the following aesthetics:

• x
• y
• datatype
• thickness
• size
• group

In addition, in their default configuration (paired with geom_dotsinterval()) the following aesthetics are supported by the underlying geom:

• slab_shape
• datatype
• alpha
• colour
• linetype
• fill
• shape
• stroke
• point_colour
• point_fill
• point_alpha
• point_size
• size
• interval_colour
• interval_alpha
• interval_size
• interval_linetype
• slab_size
• slab_colour
• slab_fill
• slab_alpha
• slab_linetype
• y
• ymin
• ymax
• x
• xmin
• xmax
• width
• height
• thickness
• group

See examples of some of these aesthetics in action in vignette("slabinterval"). Learn more about the sub-geom aesthetics (like interval_color) in the scales documentation. Learn more about basic ggplot aesthetics in vignette("ggplot2-specs").

Author(s)
Matthew Kay

References


See Also
See stat_sample_slabinterval() and stat_dist_slabinterval() for families of other stats built on top of geom_slabinterval(). See vignette("slabinterval") for a variety of examples of use.

Examples

# TODO
**geom_eye**

Eye plots of densities with point and interval summaries (Deprecated spellings)

**Description**

These are deprecated spellings of `stat_eye()`, `stat_eyeh()`, and `stat_halfeyeh()`. Use those functions instead.

**Usage**

```r
geom_eye(..., scale = 0.9, .width = c(0.66, 0.95), relative_scale, .prob)
geom_eyeh(..., scale = 0.9, .width = c(0.66, 0.95), relative_scale, .prob)
geom_halfeyeh(..., scale = 0.9, .width = c(0.66, 0.95), relative_scale, .prob)
```

**Arguments**

- `...` Arguments passed to `stat_sample_slabinterval()`
- `scale` What proportion of the region allocated to this geom to use to draw the slab. If `scale = 1`, slabs that use the maximum range will just touch each other. Default is `0.9` to leave some space.
- `.width` The `.width` argument passed to `interval_function` or `point_interval`.
- `relative_scale` Deprecated. Use `scale` instead.
- `.prob` Deprecated. Use `.width` instead.

**Author(s)**

Matthew Kay

**See Also**

See `stat_sample_slabinterval()` and the shortcut stats documented there for the preferred way to generate these geoms.

**Examples**

```r
library(magrittr)
library(ggplot2)

data(RankCorr, package = "tidybayes")

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(y = i, x = u_tau)) +
```
```r
geom_eyeh()

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(x = i, y = u_tau)) +
  geom_eye()
```

---

**Description**

Multiple interval geoms with default aesthetics designed for use with output from `point_interval()`. Wrapper around `geom_slabinterval()`.

**Usage**

```r
geom_interval(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  side = "both",
  orientation = "vertical",
  interval_size_range = c(1, 6),
  show_slab = FALSE,
  show_point = FALSE
)
```

```r
geom_intervalh(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  side = "both",
  orientation = "horizontal",
  interval_size_range = c(1, 6),
  show_slab = FALSE,
  show_point = FALSE
)
```

**Arguments**

- **mapping**
  
  Set of aesthetic mappings created by `aes()` or `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.
geom_interval

- **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, as a string.

- **position**: The position adjustment to use for overlapping points on this layer. Setting this equal to "dodge" or "dodgev" (if the ggstance package is loaded) can be useful if you have overlapping intervals.

... Arguments passed on to `geom_slabinterval`

- **scale**: What proportion of the region allocated to this geom to use to draw the slab. If scale = 1, slabs that use the maximum range will just touch each other. Default is 0.9 to leave some space.

- **justification**: Justification of the interval relative to the slab, where 0 indicates bottom/left justification and 1 indicates top/right justification (depending on orientation). If justification is NULL (the default), then it is set automatically based on the value of side: when side is "top"/"right" justification is set to 0, when side is "bottom"/"left" justification is set to 1, and when side is "both" justification is set to 0.5.

- **normalize**: How to normalize heights of functions input to the thickness aesthetic. If "all" (the default), normalize so that the maximum height across all data is 1; if "panels", normalize within panels so that the maximum height in each panel is 1; if "xy", normalize within the x/y axis opposite the orientation of this geom so that the maximum height at each value of the opposite axis is 1; if "groups", normalize within values of the opposite axis and within groups so that the maximum height in each group is 1; if "none", values are taken as is with no normalization (this should probably only be used with functions whose values are in [0,1], such as CDFs).

- **interval_size_domain**: The minimum and maximum of the values of the size aesthetic that will be translated into actual sizes for intervals drawn according to `interval_size_range` (see the documentation for that argument.)

- **fatten_point**: A multiplicative factor used to adjust the size of the point relative to the size of the thickest interval line. If you wish to specify point sizes directly, you can also use the point_size aesthetic and `scale_point_size_continuous()` or `scale_point_size_discrete()`; sizes specified with that aesthetic will not be adjusted using fatten_point.

- **show_interval**: Should the interval portion of the geom be drawn? Default TRUE.

- **na.rm**: If FALSE, 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()`.

**side** Which side to draw the slab on. "topright", "top", and "right" are synonyms which cause the slab to be drawn on the top or the right depending on if orientation is "horizontal" or "vertical". "bottomleft", "bottom", and "left" are synonyms which cause the slab to be drawn on the bottom of the left depending on if orientation is "horizontal" or "vertical". "both" draws the slab mirrored on both sides (as in a violin plot).

**orientation** Whether this geom is drawn horizontally ("horizontal") or vertically ("vertical"). When horizontal (resp. vertical), the geom uses the y (resp. x) aesthetic to identify different groups, then for each group uses the x (resp. y) aesthetic and the thickness aesthetic to draw a function as a slab, and draws points and intervals horizontally (resp. vertically) using the xmin, x, and xmax (resp. ymin, y, and ymax) aesthetics.

**interval_size_range** This geom scales the raw size aesthetic values when drawing interval and point sizes, as they tend to be too thick when using the default settings of `scale_size_continuous()`, which give sizes with a range of c(1,6). The interval_size_domain value indicates the input domain of raw size values (typically this should be equal to the value of the range argument of the `scale_size_continuous()` function), and interval_size_range indicates the desired output range of the size values (the min and max of the actual sizes used to draw intervals).

**show_slab** Should the slab portion of the geom be drawn? Default TRUE.

**show_point** Should the point portion of the geom be drawn? Default TRUE.

**Details**

These geoms are wrappers around `geom_slabinterval()` with defaults designed to produce multiple interval plots. These geoms set some default aesthetics equal to the .lower, .upper, and .width columns generated by the point_interval family of functions, making them often more convenient than vanilla `geom_linerange()` when used with functions like `median_qi()`, `mean_qi()`, `mode_hdi()`, etc.

Specifically, `geom_interval` acts as if its default aesthetics are `aes(ymin = .lower, ymax = .upper, color = fct_rev(ordered(.width)))`. `geom_intervalh` acts as if its default aesthetics are `aes(xmin = .lower, xmax = .upper, color = fct_rev(ordered(.width)))`.

**Aesthetics**

These geoms support the following aesthetics:

- datatype
- alpha
- colour
• linetype
• fill
• shape
• stroke
• point_colour
• point_fill
• point_alpha
• point_size
• size
• interval_colour
• interval_alpha
• interval_size
• interval_linetype
• slab_size
• slab_colour
• slab_fill
• slab_alpha
• slab_linetype
• y
• ymin
• ymax
• x
• xmin
• xmax
• width
• height
• thickness
• group

See examples of some of these aesthetics in action in vignette("slabinterval"). Learn more about the sub-geom aesthetics (like interval_color) in the scales documentation. Learn more about basic ggplot aesthetics in vignette("ggplot2-specs").

Author(s)

Matthew Kay

See Also

See `stat_interval()` / `stat_intervalh()` for the stat versions, intended for use on samples from a distribution. See `geom_interval()` / `geom_intervalh()` for a similar geom intended for intervals without point summaries. See `stat_sample_slabinterval()` for a variety of other stats that combine intervals with densities and CDFs. See `geom_slabinterval()` for the geom that these geoms wrap. All parameters of that geom are available to these geoms.
Examples

```r
library(magrittr)
library(ggplot2)
data(RankCorr, package = "tidybayes")

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  median_qi(.width = c(.5, .8, .95, .99)) %>%
  ggplot(aes(y = i, x = u_tau)) +
  geom_intervalh() +
  scale_color_brewer()

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  median_qi(.width = c(.5, .8, .95, .99)) %>%
  ggplot(aes(x = i, y = u_tau)) +
  geom_interval() +
  scale_color_brewer()
```

---

**geom_lineribbon**  
*Line + multiple probability ribbon plots (ggplot geom)*

**Description**

A combination of `geom_line()` and `geom_ribbon()` with default aesthetics designed for use with output from `point_interval()`.

**Usage**

```r
geom_lineribbon(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

**Arguments**

- `mapping`  
  Set of aesthetic mappings created by `aes()` or `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.
geom_lineribbon

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, as a string.

position
Position adjustment, either as a string, or the result of a call to a position adjustment function.

... Other arguments passed to layer().

na.rm If FALSE, 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().

Details
g geom_lineribbon is a combination version of a geom_line(), and geom_ribbon designed for use with output from point_interval(). This geom sets some default aesthetics equal to the .lower, .upper, and .width columns generated by the point_interval family of functions, making them often more convenient than a vanilla geom_ribbon() + geom_line() .

Specifically, geom_lineribbon acts as if its default aesthetics are aes(ymin = .lower, ymax = .upper, forcats::fct_rev(.width)).

Author(s)
Matthew Kay

See Also
See stat_lineribbon() for a version that does summarizing of samples into points and intervals within ggplot. See geom_pointinterval() / geom_pointintervalh() for a similar geom intended for point summaries and intervals. See geom_ribbon() and geom_line() for the geoms this is based on.

Examples

library(dplyr)
library(ggplot2)
tibble(x = 1:10) %>%
group_by_all() %>%
do(tibble(y = rnorm(100, x))) %>%
median_qi(.width = c(.5, .8, .95)) %>%
ggplot(aes(x = x, y = y)) +
  # automatically uses aes(ymin = .lower, ymax = .upper, fill = fct_rev(ordered(.width)))
  geom_lineribbon() +
  scale_fill_brewer()
**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, as a string.

**position**

The position adjustment to use for overlapping points on this layer. Setting this equal to "dodge" or "dodgev" (if the ggstance package is loaded) can be useful if you have overlapping intervals.

... Arguments passed on to `geom_slabinterval`

**scale**

What proportion of the region allocated to this geom to use to draw the slab. If `scale = 1`, slabs that use the maximum range will just touch each other. Default is `0.9` to leave some space.

**justification**

Justification of the interval relative to the slab, where `0` indicates bottom/left justification and `1` indicates top/right justification (depending on orientation). If justification is `NULL` (the default), then it is set automatically based on the value of `side`:
- when `side` is "top"/"right", justification is set to `0`,
- when `side` is "bottom"/"left" justification is set to `1`, and
- when `side` is "both" justification is set to `0.5`.

**normalize**

How to normalize heights of functions input to the thickness aesthetic. If "all" (the default), normalize so that the maximum height across all data is `1`; if "panels", normalize within panels so that the maximum height in each panel is `1`; if "xy", normalize within the x/y axis opposite the orientation of this geom so that the maximum height at each value of the opposite axis is `1`; if "groups", normalize within values of the opposite axis and within groups so that the maximum height in each group is `1`; if "none", values are taken as is with no normalization (this should probably only be used with functions whose values are in `[0, 1]`, such as CDFs).

**interval_size_domain**

The minimum and maximum of the values of the size aesthetic that will be translated into actual sizes for intervals drawn according to `interval_size_range` (see the documentation for that argument).

**interval_size_range**

This geom scales the raw size aesthetic values when drawing interval and point sizes, as they tend to be too thick when using the default settings of `scale_size_continuous()`, which give sizes with a range of `c(1, 6)`. The `interval_size_domain` value indicates the input domain of raw size values (typically this should be equal to the value of the range argument of the `scale_size_continuous()` function), and `interval_size_range` indicates the desired output range of the size values (the min and max of the actual sizes used to draw intervals).

**fatten_point**

A multiplicative factor used to adjust the size of the point relative to the size of the thickest interval line. If you wish to specify point sizes directly, you can also use the `point_size` aesthetic and `scale_point_size_continuous()`
or `scale_point_size_discrete()`: sizes specified with that aesthetic will not be adjusted using `fatten_point`.

- **show_point**: Should the point portion of the geom be drawn? Default TRUE.
- **show_interval**: Should the interval portion of the geom be drawn? Default TRUE.
- **na.rm**: If FALSE, the default, missing values are removed with a warning. If TRUE, missing values are silently removed.
- **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()`.

- **side**: Which side to draw the slab on. "topright", "top", and "right" are synonyms which cause the slab to be drawn on the top or the right depending on if orientation is "horizontal" or "vertical". "bottomleft", "bottom", and "left" are synonyms which cause the slab to be drawn on the bottom of the left depending on if orientation is "horizontal" or "vertical". "both" draws the slab mirrored on both sides (as in a violin plot).

- **orientation**: Whether this geom is drawn horizontally ("horizontal") or vertically ("vertical"). When horizontal (resp. vertical), the geom uses the y (resp. x) aesthetic to identify different groups, then for each group uses the x (resp. y) aesthetic and the thickness aesthetic to draw a function as an slab, and draws points and intervals horizontally (resp. vertically) using the xmin, x, and xmax (resp. ymin, y, and ymax) aesthetics.

- **show_slab**: Should the slab portion of the geom be drawn? Default TRUE.

- **show.legend**: Should this layer be included in the legends? Default is c(size = FALSE), unlike most geoms, to match its common use cases. FALSE hides all legends, TRUE shows all legends, and NA shows only those that are mapped (the default for most geoms).

### Details

These geoms are wrappers around `geom_slabinterval()` with defaults designed to produce points+interval plots. These geoms set some default aesthetics equal to the `.lower`, `.upper`, and `.width` columns generated by the `point_interval` family of functions, making them often more convenient than vanilla `geom_pointrange()` when used with functions like `median_qi()`, `mean_qi()`, `mode_hdi()`, etc.

Specifically, `geom_pointinterval` acts as if its default aesthetics are `aes(ymin = .lower, ymax = .upper, size = -.width)`. `geom_pointintervalh` acts as if its default aesthetics are `aes(xmin = .lower, xmax = .upper, size = -.width)`. 

### Aesthetics

These geoms support the following aesthetics:

- datatype
- alpha
- colour
geom_pointinterval

- linetype
- fill
- shape
- stroke
- point_colour
- point_fill
- point_alpha
- point_size
- size
- interval_colour
- interval_alpha
- interval_size
- interval_linetype
- slab_size
- slab_colour
- slab_fill
- slab_alpha
- slab_linetype
- y
- ymin
- ymax
- x
- xmin
- xmax
- width
- height
- thickness
- group

See examples of some of these aesthetics in action in vignette("slabinterval"). Learn more about the sub-geom aesthetics (like interval_color) in the scales documentation. Learn more about basic ggplot aesthetics in vignette("ggplot2-specs").

Author(s)

Matthew Kay
geom_slabinterval

See Also

See `geom_slabinterval()` for the geom that these geoms wrap. All parameters of that geom are available to these geoms.

See `stat_pointinterval()` / `stat_pointintervalh()` for the stat versions, intended for use on samples from a distribution. See `geom_interval()` / `geom_intervalh()` for a similar stat intended for intervals without point summaries. See `stat_sample_slabinterval()` for a variety of other stats that combine intervals with densities and CDFs. See `geom_slabinterval()` for the geom that these geoms wrap. All parameters of that geom are available to these geoms.

Examples

```r
library(magrittr)
library(ggplot2)

data(RankCorr, package = "tidybayes")

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  median_qi(.width = c(.8, .95)) %>%
  ggplot(aes(y = i, x = u_tau)) +
  geom_pointintervalh()

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  median_qi(.width = c(.8, .95)) %>%
  ggplot(aes(x = i, y = u_tau)) +
  geom_pointinterval()
```

**Description**

This meta-geom supports drawing combinations of functions (as slabs, aka ridge plots or joy plots), points, and intervals. It acts as a meta-geom for many other tidybayes geoms that are wrappers around this geom, including eye plots, half-eye plots, CCDF barplots, and point+multiple interval plots, and supports both horizontal and vertical orientations, dodging (via the `position` argument), and relative justification of slabs with their corresponding intervals.

**Usage**

```r
geom_slabinterval(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  position_dodge = NULL,
  position_dodge_width = 0.5,
  position_dodge_units = NULL,
  orientation = NULL,
  show.legend = NA,
  inherit.aes = TRUE,
  ...)
```
geom_slabinterval

side = c("topright", "top", "right", "bottomleft", "bottom", "left", "both"),
scale = 0.9,
orientation = c("vertical", "horizontal"),
justification = NULL,
normalize = c("all", "panels", "xy", "groups", "none"),
interval_size_domain = c(1, 6),
interval_size_range = c(0.6, 1.4),
fatten_point = 1.8,
show_slab = TRUE,
show_point = TRUE,
show_interval = TRUE,
na.rm = FALSE,
show.legend = NA,
inherit.aes = TRUE
)

geom_slab(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...
)

Arguments

mapping Set of aesthetic mappings created by aes() or 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, as a string.

position Position adjustment, either as a string, or the result of a call to a position adjustment function.
... Other arguments passed to `layer()`.

`side` Which side to draw the slab on. "topright", "top", and "right" are synonyms which cause the slab to be drawn on the top or the right depending on if `orientation` is "horizontal" or "vertical". "bottomleft", "bottom", and "left" are synonyms which cause the slab to be drawn on the bottom of the left depending on if `orientation` is "horizontal" or "vertical". "both" draws the slab mirrored on both sides (as in a violin plot).

`scale` What proportion of the region allocated to this geom to use to draw the slab. If `scale` = 1, slabs that use the maximum range will just touch each other. Default is 0.9 to leave some space.

`orientation` Whether this geom is drawn horizontally ("horizontal") or vertically ("vertical"). When horizontal (resp. vertical), the geom uses the y (resp. x) aesthetic to identify different groups, then for each group uses the x (resp. y) aesthetic and the thickness aesthetic to draw a function as an slab, and draws points and intervals horizontally (resp. vertically) using the `xmin`, x, and `xmax` (resp. `ymin`, y, and `ymax`) aesthetics.

`justification` Justification of the interval relative to the slab, where 0 indicates bottom/left justification and 1 indicates top/right justification (depending on `orientation`). If `justification` is NULL (the default), then it is set automatically based on the value of `side`: when `side` is "top"/"right" justification is set to 0, when `side` is "bottom"/"left" justification is set to 1, and when `side` is "both" justification is set to 0.5.

`normalize` How to normalize heights of functions input to the `thickness` aesthetic. If "all" (the default), normalize so that the maximum height across all data is 1; if "panels", normalize within panels so that the maximum height in each panel is 1; if "xy", normalize within the x/y axis opposite the `orientation` of this geom so that the maximum height at each value of the opposite axis is 1; if "groups", normalize within values of the opposite axis and within groups so that the maximum height in each group is 1; if "none", values are taken as is with no normalization (this should probably only be used with functions whose values are in [0,1], such as CDFs).

`interval_size_domain` The minimum and maximum of the values of the size aesthetic that will be translated into actual sizes for intervals drawn according to `interval_size_range` (see the documentation for that argument.)

`interval_size_range` This geom scales the raw size aesthetic values when drawing interval and point sizes, as they tend to be too thick when using the default settings of `scale_size_continuous()`, which give sizes with a range of c(1, 6). The `interval_size_domain` value indicates the input domain of raw size values (typically this should be equal to the value of the range argument of the `scale_size_continuous()` function), and `interval_size_range` indicates the desired output range of the size values (the min and max of the actual sizes used to draw intervals).

`fatten_point` A multiplicative factor used to adjust the size of the point relative to the size of the thickest interval line. If you wish to specify point sizes directly, you can also use the `point_size` aesthetic and `scale_point_size_continuous()` or
**geom_slabinterval**

- `scale_point_size_discrete()`: sizes specified with that aesthetic will not be adjusted using `fatten_point`.
- `show_slab`: Should the slab portion of the geom be drawn? Default TRUE.
- `show_point`: Should the point portion of the geom be drawn? Default TRUE.
- `show_interval`: Should the interval portion of the geom be drawn? Default TRUE.
- `na.rm`: If FALSE, 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()`.

**Details**

`geom_slabinterval` is a flexible meta-geom that you can use directly or through a variety of "shortcut" geoms that represent useful combinations of the various parameters of this geom. In many cases you will want to use the shortcut geoms instead as they create more useful mnemonic primitives, such as eye plots, half-eye plots, point+interval plots, or CCDF barplots.

The slab portion of the geom is much like a ridge or "joy" plot: it represents the value of a function scaled to fit between values on the x or y access (depending on the value of orientation). Values of the functions are specified using the thickness aesthetic and are scaled to fit into scale times the distance between points on the relevant axis. E.g., if orientation is "horizontal", scale is 0.9, and y is a discrete variable, then the thickness aesthetic specifies the value of some function of x that is drawn for every y value and scaled to fit into 0.9 times the distance between points on the y axis.

For the interval portion of the geom, x and y aesthetics specify the location of the point and ymin/ymax or xmin/xmax (depending on the value of orientation specifying the endpoints of the interval. A scaling factor for interval line width and point size is applied through the interval_size_domain, interval_size_range, and fatten_point parameters. These scaling factors are designed to give multiple probability intervals reasonable scaling at the default settings for `scale_size_continuous()`.

As a combination geom, this geom expects a datatype aesthetic specifying which part of the geom a given row in the input data corresponds to: "slab" or "interval". However, specifying this aesthetic manually is typically only necessary if you use this geom directly; the numerous wrapper geoms will usually set this aesthetic for you as needed, and their use is recommended unless you have a very custom use case.

Wrapper geoms and stats include:

- `stat_sample_slabinterval()` and associated stats
- `stat_dist_slabinterval()` and associated stats
- `geom_pointinterval()` / `geom_pointintervalh()`
- `stat_pointinterval()` / `stat_pointintervalh()`
- `geom_interval()` / `geom_intervalh()`
• `stat_interval() / stat_intervalh()`

Typically, the geom_* versions are meant for use with already-summarized data (such as intervals) and the stat_* versions are summarize the data themselves (usually draws from a distribution) to produce the geom.

Aesthetics

These geoms support the following aesthetics:

• `datatype`
• `alpha`
• `colour`
• `linetype`
• `fill`
• `shape`
• `stroke`
• `point_colour`
• `point_fill`
• `point_alpha`
• `point_size`
• `size`
• `interval_colour`
• `interval_alpha`
• `interval_size`
• `interval_linetype`
• `slab_size`
• `slab_colour`
• `slab_fill`
• `slab_alpha`
• `slab_linetype`
• `y`
• `ymin`
• `ymax`
• `x`
• `xmin`
• `xmax`
• `width`
• `height`
• `thickness`
• `group`

See examples of some of these aesthetics in action in vignette("slabinterval"). Learn more about the sub-geom aesthetics (like `interval_color`) in the scales documentation. Learn more about basic ggplot aesthetics in vignette("ggplot2-specs").
get_variables

Author(s)
Matthew Kay

See Also
See `geom_lineribbon()` for a combination geom designed for fit curves plus probability bands. See `stat_sample_slabinterval()` and `stat_dist_slabinterval()` for families of stats built on top of this geom for common use cases (like `stat_halfeyeh`). See `vignette("slabinterval")` for a variety of examples of use.

Examples

```r
# geom_slabinterval() is typically not that useful on its own.
# See vignette("slabinterval") for a variety of examples of the use of its
# shortcut geoms and stats, which are more useful than using
# geom_slabinterval() directly.
```

---

get_variables Get the names of the variables in a fitted Bayesian model

Description

Get a character vector of the names of the variables in a variety of fitted Bayesian model types. All models supported by `tidy_draws()` are supported.

Usage

```r
get_variables(model)
```

## Default S3 method:
```r
get_variables(model)
```

## S3 method for class `mcmc`
```r
get_variables(model)
```

## S3 method for class `mcmc.list`
```r
get_variables(model)
```

Arguments

- `model` A supported Bayesian model fit. Tidybayes supports a variety of model objects; for a full list of supported models, see `tidybayes-models`.

Details

This function is often useful for inspecting a model interactively in order to construct calls to `spread_draws()` or `gather_draws()` in order to extract draws from models in a tidy format.
lkjcorr_marginal

Marginal distribution of a single correlation from an LKJ distribution

Description

Marginal distribution for the correlation in a single cell from a correlation matrix distributed according to an LKJ distribution.

Usage

dlkjcorr_marginal(x, K, eta, log = FALSE)
plkjcorr_marginal(q, K, eta, lower.tail = TRUE, log.p = FALSE)
qlkjcorr_marginal(p, K, eta, lower.tail = TRUE, log.p = FALSE)
rlkjcorr_marginal(n, K, eta)

Arguments

x vector of quantiles.
K Dimension of the correlation matrix. Must be greater than or equal to 2.
eta Parameter controlling the shape of the distribution
log logical; if TRUE, probabilities p are given as log(p).
q vector of quantiles.
lower.tail logical; if TRUE (default), probabilities are \( P[X \leq x] \) otherwise, \( P[X > x] \).

log.p logical; if TRUE, probabilities \( p \) are given as \( \log(p) \).

p vector of probabilities.

n number of observations. If \( \text{length}(n) > 1 \), the length is taken to be the number required.

Details

The LKJ distribution is a distribution over correlation matrices with a single parameter, \( \eta \). For a given \( \eta \) and a \( K \times K \) correlation matrix \( R \):

\[
R \sim \text{LKJ}(\eta)
\]

Each off-diagonal entry of \( R \), \( r_{ij} : i \neq j \), has the following marginal distribution (Lewandowski, Kurowicka, and Joe 2009):

\[
\frac{r_{ij} + 1}{2} \sim \text{Beta} \left( \eta - 1 + \frac{K}{2}, \eta - 1 + \frac{K}{2} \right)
\]

In other words, \( r_{ij} \) is marginally distributed according to the above Beta distribution scaled into \((-1, 1)\).

References


See Also

`parse_dist()` and `marginalize_lkjcorr()` for parsing specs that use the LKJ correlation distribution and the `stat_dist_slabinterval()` family of stats for visualizing them.

Examples

```r
library(dplyr)
library(ggplot2)

data.frame(prior = "lkjcorr_marginal(2, 3)") %>%
  parse_dist(prior) %>%
  ggplot(aes(y = prior, dist = .dist, args = .args)) +
  stat_dist_halfeyeh() +
  xlim(-1, 1) +
  xlab("Marginal correlation for LKJ(3) prior on 2x2 correlation matrix")
```
marginalize_lkjcorr  

Turn spec for LKJ distribution into spec for marginal LKJ distribution

Description

Turns specs for an LKJ correlation matrix distribution as returned by `parse_dist()` into specs for the marginal distribution of a single cell in an LKJ-distributed correlation matrix (i.e., `lkjcorr_marginal()`). Useful for visualizing prior correlations from LKJ distributions.

Usage

```r
marginalize_lkjcorr(data, K, predicate = NULL, dist = ".dist", args = ".args")
```

Arguments

- **data**: A data frame containing a column with distribution names (".dist" by default) and a list column of distribution arguments (".args" by default), such as output by `parse_dist()`.
- **K**: Dimension of the correlation matrix. Must be greater than or equal to 2.
- **predicate**: a bare expression for selecting the rows of `data` to modify. This is useful if `data` contains more than one row with an LKJ prior in it and you only want to modify some of the distributions; if this is the case, give row a predicate expression (such as you might supply to `dplyr::filter()`) that evaluates to `TRUE` on the rows you want to modify. If `NULL` (the default), all `lkjcorr` distributions in `data` are modified.
- **dist**: The name of the column containing distribution names. See `parse_dist()`.
- **args**: The name of the column containing distribution arguments. See `parse_dist()`.

Details

The LKJ(eta) prior on a correlation matrix induces a marginal prior on each correlation in the matrix that depends on both the value of eta and K, the dimension of the K x K correlation matrix. Thus to visualize the marginal prior on the correlations, it is necessary to specify the value of K, which depends on what your model specification looks like.

Given a data frame representing parsed distribution specifications (such as returned by `parse_dist()`), this function updates any rows with `.dist == "lkjcorr"` so that the first argument to the distribution is equal to the specified dimension of the correlation matrix (K) and changes the distribution name to "lkjcorr_marginal", allowing the distribution to be easily visualized using the `stat_dist_slabinterval()` family of ggplot2 stats.

See Also

- `parse_dist()`
- `lkjcorr_marginal()`
Examples

library(dplyr)
library(ggplot2)

# Say we have an LKJ(3) prior on a 2x2 correlation matrix. We can visualize
# its marginal distribution as follows...
data.frame(prior = "lkjcorr(3)") %>%
  parse_dist(prior) %>%
  marginalize_lkjcorr(K = 2) %>%
  ggplot(aes(y = prior, dist = .dist, args = .args)) +
  stat_dist_halfeyeh() +
  xlim(-1, 1) +
  xlab("Marginal correlation for LKJ(3) prior on 2x2 correlation matrix")

# Say our prior list has multiple LKJ priors on correlation matrices
# of different sizes, we can supply a predicate expression to select
# only those rows we want to modify
data.frame(coef = c("a", "b"), prior = "lkjcorr(3)") %>%
  parse_dist(prior) %>%
  marginalize_lkjcorr(K = 2, coef == "a") %>%
  marginalize_lkjcorr(K = 4, coef == "b")

---

n_prefix

Prefix function generator for composing dimension index columns

Description

Generates a function for generating names of index columns for factors in `compose_data()` by
prefixed a character vector to the original column name.

Usage

n_prefix(prefix)

Arguments

prefix

Character vector to be prepended to column names by `compose_data()` to cre-
te index columns. Typically something like "n" (that is the default used in the

Returns a function. The function returned takes a character vector, name and
returns `paste0(prefix, ",", name)`, unless name is empty, in which case it will
return `prefix`.
n_prefix("n") is the default method that `compose_data()` uses to generate
column names for variables storing the number of levels in a factor. Under
this method, given a data frame `df` with a factor column "foo" containing 5
levels, the results of `compose_data(df)` will include an element named "n"
(the result of n_prefix("n")("")) equal to the number of rows in df and an element named "n_foo" (the result of n_prefix("n")("foo")) equal to the number of levels in df$foo.

See Also

The .n_name argument of compose_data().

Examples

library(magrittr)

df = data.frame(
  plot = factor(paste0("p", rep(1:8, times = 2))),
  site = factor(paste0("s", rep(1:4, each = 2, times = 2)))
)

# without changing `.n_name`, compose_data() will prefix indices
# with "n" by default
df %>%
  compose_data()

# you can use n_prefix() to define a different prefix (e.g. "N"):
df %>%
  compose_data(.n_name = n_prefix("N"))

parse_dist

Parse distribution specifications into columns of a data frame

Description

Parses simple string distribution specifications, like "normal(0,1)", into two columns of a data frame, suitable for use with stat_dist_slabinterval() and its shortcut stats (like stat_dist_halfeye). This format is output by brms::get_prior(), making it particularly useful for visualizing priors from brms models.

Usage

parse_dist(object, ..., dist = ".dist", args = ".args", to_r_names = TRUE)

## Default S3 method:
parse_dist(object, ...)

## S3 method for class 'data.frame'
parse_dist(
  object,
  dist_col,
... 

```r

## S3 method for class 'character'
parse_dist(object, ..., dist = ".dist", args = ".args", to_r_names = TRUE)

## S3 method for class 'factor'
parse_dist(object, ..., dist = ".dist", args = ".args", to_r_names = TRUE)

r_dist_name(dist_name)
```

**Arguments**

- `object`: A character vector containing distribution specifications or a data frame with a column containing distribution specifications.
- `...`: Arguments passed to other implementations of `parse_dist`.
- `dist`: The name of the output column to contain the distribution name.
- `args`: The name of the output column to contain the arguments to the distribution.
- `to_r_names`: If `TRUE` (the default), certain common aliases for distribution names are automatically translated into names that R can recognize (i.e., names which have functions starting with `r`, `p`, `q`, and `d` representing random number generators, distribution functions, etc. for that distribution), using the `r_dist_name` function. For example, "normal" is translated into "norm" and "lognormal" is translated into "lnorm".
- `dist_col`: A bare (unquoted) column or column expression that resolves to a character vector of distribution specifications.
- `dist_name`: For `r_dist_name`, a character vector of distribution names to be translated into distribution names R recognizes. Unrecognized names are left as-is.

**Details**

`parse_dist()` can be applied to character vectors or to a data frame + bare column name of the column to parse, and returns a data frame with ".dist" and ".args" columns added. `parse_dist()` uses `r_dist_name()` to translate distribution names into names recognized by R.

`r_dist_name()` takes a character vector of names and translates common names into R distribution names. Names are first made into valid R names using `make.names()`, then translated (ignoring character case, ".", and ":"). Thus, "lognormal", "LogNormal", "log_normal", "log-Normal", and any number of other variants all get translated into "lnorm".

**See Also**

See `stat_dist_slabinterval()` and its shortcut stats, which can easily make use of the output of this function using the `dist` and `args` aesthetics.
Examples

library(dplyr)

# parse dist can operate on strings directly...
pars dist(c("normal(0,1)", "student_t(3,0,1)"))

# ... or on columns of a data frame, where it adds the
# parsed specs back on as columns
data.frame(prior = c("normal(0,1)", "student_t(3,0,1)")) %>%
  parse_dist(prior)

if (
  require("brms", quietly = TRUE)
) {
  # parse_dist is particularly useful with brms prior specifications,
  # which follow the same format...

  # get priors for a brms model
  priors = get_prior(mpg ~ log(hp), data = mtcars, family = lognormal)

  # The 'prior' column output by 'get_prior()' is a character vector
  # of distribution specifications. We can parse this directly...
pars dist(priors$prior)

  # ... or we can parse it and have it added back onto the original data frame
  # (this form is likely more useful for plotting, since the other columns
  # are retained)
pars dist(priors, prior)

}

point_interval  
Point and interval summaries for tidy data frames of draws from distributions

Description

Translates draws from distributions in a (possibly grouped) data frame into point and interval summaries (or set of point and interval summaries, if there are multiple groups in a grouped data frame).

Usage

point_interval(
  .data,
  ...,  
  .width = 0.95,
point_interval

```r
.point = median,
.interval = qi,
.simple_names = TRUE,
na.rm = FALSE,
.exclude = c(".chain", ".iteration", ".draw", ".row"),
.proba

## Default S3 method:
point_interval(
  .data,
  ..., 
  .width = 0.95,
  .point = median,
  .interval = qi,
  .simple_names = TRUE,
  na.rm = FALSE,
  .exclude = c(".chain", ".iteration", ".draw", ".row"),
  .prob
)

## S3 method for class 'numeric'
point_interval(
  .data,
  ..., 
  .width = 0.95,
  .point = median,
  .interval = qi,
  .simple_names = FALSE,
  na.rm = FALSE,
  .exclude = c(".chain", ".iteration", ".draw", ".row"),
  .prob
)

point_intervalh(...)

qi(x, .width = 0.95, .prob, na.rm = FALSE)

hdi(x, .width = 0.95, .prob, na.rm = FALSE)

Mode(x, na.rm = FALSE)

hdci(x, .width = 0.95, na.rm = FALSE)

mean_qi(.data, ..., .width = 0.95)

mean_qih(...)
```
median_qi(.data, ..., .width = 0.95)
median_qih(...)
mode_qi(.data, ..., .width = 0.95)
mode_qih(...)
mean_hdi(.data, ..., .width = 0.95)
mean_hdih(...)
median_hdi(.data, ..., .width = 0.95)
median_hdih(...)
mode_hdi(.data, ..., .width = 0.95)
mode_hdih(...)
mean_hdci(.data, ..., .width = 0.95)
mean_hdcih(...)
median_hdci(.data, ..., .width = 0.95)
median_hdcih(...)
mode_hdci(.data, ..., .width = 0.95)
mode_hdcih(...)

**Arguments**

- `.data` Data frame (or grouped data frame as returned by `group_by()`) that contains draws to summarize.
- `...` Bare column names or expressions that, when evaluated in the context of `.data`, represent draws to summarize. If this is empty, then by default all columns that are not group columns and which are not in `.exclude` (by default `.chain`, `.iteration`, `.draw`, and `.row`) will be summarized. This can be list columns.
- `.width` vector of probabilities to use that determine the widths of the resulting intervals. If multiple probabilities are provided, multiple rows per group are generated, each with a different probability interval (and value of the corresponding `.width` column).
- `.point` Point summary function, which takes a vector and returns a single value, e.g. `mean()`, `median()`, or `Mode()`.
point_interval

.interval Interval function, which takes a vector and a probability (.width) and returns a two-element vector representing the lower and upper bound of an interval; e.g. qi(), hdi()

.simple_names When TRUE and only a single column / vector is to be summarized, use the name .lower for the lower end of the interval and .upper for the upper end. If .data is a vector and this is TRUE, this will also set the column name of the point summary to .value. When FALSE and .data is a data frame, names the lower and upper intervals for each column x.lower and x.upper. When FALSE and .data is a vector, uses the naming scheme y, ymin and ymax (for use with ggplot).

.na.rm logical value indicating whether NA values should be stripped before the computation proceeds. If FALSE (the default), any vectors to be summarized that contain NA will result in point and interval summaries equal to NA.

.exclude A character vector of names of columns to be excluded from summarization if no column names are specified to be summarized. Default ignores several meta-data column names used in tidybayes.

.prob Deprecated. Use .width instead.

.x vector to summarize (for interval functions: qi and hdi)

Details

If .data is a data frame, then ... is a list of bare names of columns (or expressions derived from columns) of .data, on which the point and interval summaries are derived. Column expressions are processed using the tidy evaluation framework (see rlang::eval_tidy()).

For a column named x, the resulting data frame will have a column named x containing its point summary. If there is a single column to be summarized and .simple_names is TRUE, the output will also contain columns .lower (the lower end of the interval), .upper (the upper end of the interval). Otherwise, for every summarized column x, the output will contain x.lower (the lower end of the interval) and x.upper (the upper end of the interval). Finally, the output will have a .width column containing the probability for the interval on each output row.

If .data includes groups (see e.g. dplyr::group_by()), the points and intervals are calculated within the groups.

If .data is a vector, ... is ignored and the result is a data frame with one row per value of .width and three columns: y (the point summary), ymin (the lower end of the interval), ymax (the upper end of the interval), and .width, the probability corresponding to the interval. This behavior allows point_interval and its derived functions (like median_qi, mean_qi, mode_hdi, etc) to be easily used to plot intervals in ggplot stats using methods like stat_eye(), stat_halfeyeh(), or stat_summary().

The functions ending in h (e.g., point_intervalh, median_qih) behave identically to the function without the h, except that when passed a vector, they return a data frame with x/xmin/xmax instead of y/ymin/ymax. This allows them to be used as values of the fun.data = argument of stat_summaryh. **Note:** these functions are not necessary if you use the point_interval argument of stats and geoms in the tidybayes package (e.g. stat_pointintervalh(), stat_halfeyeh(), etc), as these automatically adjust the function output to match their required aesthetics.

median_qi, mode_hdi, etc are short forms for point_interval(...,.point = median,.interval = qi), etc.
point_interval

qi yields the quantile interval (also known as the percentile interval or equi-tailed interval) as a 1x2 matrix.

hdi yields the highest-density interval(s) (also known as the highest posterior density interval).

**Note:** If the distribution is multimodal, hdi may return multiple intervals for each probability level (these will be spread over rows). You may wish to use hdci (below) instead if you want a single highest-density interval, with the caveat that when the distribution is multimodal hdci is not a highest-density interval. Internally hdi uses HDInterval::hdi() with allowSplit = TRUE (when multimodal) and with allowSplit = FALSE (when not multimodal).

hdci yields the highest-density *continuous* interval. **Note:** If the distribution is multimodal, this may not actually be the highest-density interval (there may be a higher-density discontinuous interval). Internally hdci uses HDInterval::hdi() with allowSplit = FALSE; see that function for more information on multimodality and continuous versus discontinuous intervals.

**Author(s)**

Matthew Kay

**Examples**

```r
library(dplyr)
library(ggplot2)

set.seed(123)
	norm(1000) %>%
	median_qi()

data.frame(x = rnorm(1000)) %>%
	median_qi(x, .width = c(.50, .80, .95))

data.frame(
  x = rnorm(1000),
  y = rnorm(1000, mean = 2, sd = 2)
) %>%
median_qi(x, y)

data.frame(
  x = rnorm(1000),
  group = "a"
) %>%
rbind(data.frame(
  x = rnorm(1000, mean = 2, sd = 2),
  group = "b"
)) %>%
group_by(group) %>%
median_qi(.width = c(.50, .80, .95))

multimodal_draws = data.frame(
  x = c(rnorm(5000, 0, 1), rnorm(2500, 4, 1))
)
```

predict_curve

multimodal_draws %>%
  mode_hdi(.width = c(.66, .95))

multimodal_draws %>%
ggplot(aes(x = x, y = 0)) +
  stat_halfeyeh(point_interval = mode_hdi, .width = c(.66, .95))

predict_curve

Deprecated: Prediction curves for arbitrary functions of posteriors

Description

Deprecated function for generating prediction curves (or a density for a prediction curve).

Usage

predict_curve(data, formula, summary = median, ...)

predict_curve_density(
  data,
  formula,
  summary = function(...) density_bins(..., n = n),
  n = 50,
  ...
)

Arguments

data A data.frame, tbl_df or grouped_df representing posteriors from a Bayesian model as might be obtained through spread_draws(). Grouped data frames as returned by group_by() are supported.

formula A formula specifying the prediction curve. The left-hand side of the formula should be a name representing the name of the column that will hold the predicted response in the returned data frame. The right-hand side is an expression that may include numeric columns from data and variables passed into this function in ....

summary The function to apply to summarize each predicted response. Useful functions (if you just want a curve) might be median(), mean(), or Mode(). If you want predictive distribution at each point on the curve, try density_bins() or histogram_bins().

... Variables defining the curve. The right-hand side of formula is evaluated for every combination of values of variables in ....

n For predict_curve_density, the number of bins to use to represent the distribution at each point on the curve.
Details

This function is deprecated. Use modelr::data_grid() combined with point_interval() or dplyr::do() and density_bins() instead.

The function generates a predictive curve given posterior draws (data), an expression (formula), and a set of variables defining the curve (...). For every group in data (if it is a grouped data frame—see group_by(); otherwise the entire data frame is taken at once), and for each combination of values in ..., the right-hand side of formula is evaluated and its results passed to the summary function. This allows a predictive curve to be generated, given (e.g.) some samples of coefficients in data and a set of predictors defining the space of the curve in ... .

Given a summary function like median() or mean(), this function will produce the median (resp. mean) prediction at each point on the curve.

Given a summary function like density_bins(), this function will produce a predictive distribution for each point on the curve. predict_curve_density is a shorthand for such a call, with a convenient argument for adjusting the number of bins per point on the curve.

Value

If formula is in the form lhs ~ rhs and summary is a function that returns a single value, such as median or mode, then predict_curve returns a data frame with a column for each group in data (if it was grouped), a column for each variable in ..., and a column named lhs with the value of summary(rhs) evaluated for every group in data and combination of variables in ....

If summary is a function that returns a data frame, such as density_bins(), predict_curve has the same set of columns as above, except that in place of the lhs column is a set of columns named lhs.x for every column named x returned by summary. For example, density_bins() returns a data frame with the columns mid, lower, upper, and density, so the data frame returned by predict_curve with summary = density_bins will have columns lhs.mid, lhs.lower, lhs.upper, and lhs.density in place of lhs.

Author(s)

Matthew Kay

See Also

See density_bins().

Examples

# Deprecated; see examples for density_bins
recover_types

Decorate a model fit or sample with data types recovered from the input data

Description

Decorate a Bayesian model fit or a sample from it with types for variable and dimension data types. Meant to be used before calling `spread_draws()` or `gather_draws()` so that the values returned by those functions are translated back into useful data types.

Usage

`recover_types(model, ...)`

Arguments

- `model`: A supported Bayesian model fit. Tidybayes supports a variety of model objects; for a full list of supported models, see `tidybayes-models`.
- `...`: Lists (or data frames) providing data prototypes used to convert columns returned by `spread_draws()` and `gather_draws()` back into useful data types. See Details.

Details

Each argument in `...` specifies a list or data.frame. The `model` is decorated with a list of constructors that can convert a numeric column into the data types in the lists in `...`.

Then, when `spread_draws()` or `gather_draws()` is called on the decorated `model`, each list entry with the same name as the variable or a dimension in `variable_spec` is a used as a prototype for that variable or dimension — i.e., its type is taken to be the expected type of that variable or dimension. Those types are used to translate numeric values of variables back into useful values (for example, levels of a factor).

The most common use of `recover_types` is to automatically translate dimensions of a variable that correspond to levels of a factor in the original data back into levels of that factor. The simplest way to do this is to pass in the data frame from which the original data came.

Supported types of prototypes are factor, ordered, and logical. For example:

- if `prototypes$v` is a factor, the `v` column in the returned draws is translated into a factor using `factor(v, labels=levels(prototypes$v), ordered=is.ordered(prototypes$v))`.
- if `prototypes$v` is a logical, the `v` column is translated into a logical using `as.logical(v)`.

Additional data types can be supported by providing a custom implementation of the generic function `as_constructor`.

Value

A decorated version of `model`.
Author(s)
Matthew Kay

See Also
spread_draws(), gather_draws(), compose_data().

Examples

library(dplyr)
library(magrittr)

if(require("rstan", quietly = TRUE)) {

  # Here's an example dataset with a categorical predictor ("condition") with several levels:
  set.seed(5)
  n = 10
  n_condition = 5
  ABC =
    tibble(
      condition = rep(c("A","B","C","D","E"), n),
      response = rnorm(n * 5, c(0,1,2,1,-1), 0.5)
    )

  # We'll fit the following model to it:
  stan_code = "
  data {
    int<lower=1> n;
    int<lower=1> n_condition;
    int<lower=1, upper=n_condition> condition[n];
    real response[n];
  }
  parameters {
    real overall_mean;
    vector[n_condition] condition_zoffset;
    real<lower=0> response_sd;
    real<lower=0> condition_mean_sd;
  }
  transformed parameters {
    vector[n_condition] condition_mean;
    condition_mean = overall_mean + condition_zoffset * condition_mean_sd;
  }
  model {
    response_sd ~ cauchy(0, 1);  // => half-cauchy(0, 1)
    condition_mean_sd ~ cauchy(0, 1); // => half-cauchy(0, 1)
    overall_mean ~ normal(0, 5);
    // => condition_mean ~ normal(overall_mean, condition_mean_sd)
    condition_zoffset ~ normal(0, 1);
  }
"
```r
for (i in 1:n) {
  response[i] ~ normal(condition_mean[condition[i]], response_sd);
}

m = stan(model_code = stan_code, data = compose_data(ABC), control = list(adapt_delta=0.99),
  # 1 chain / few iterations just so example runs quickly
  # do not use in practice
  chains = 1, iter = 500)

# without using recover_types(), the 'condition' column returned by spread_draws()
# will be an integer:
m %>%
  spread_draws(condition_mean[condition]) %>%
  median_qi()

# If we apply recover_types() first, subsequent calls to other tidybayes functions will
# automatically back-convert factors so that they are labeled with their original levels
# (assuming the same name is used)
m %<>% recover_types(ABC)

# now the 'condition' column with be a factor with levels "A", "B", "C", ...
m %>%
  spread_draws(condition_mean[condition]) %>%
  median_qi()
```

---

**sample_draws**

**Sample draws from a tidy-format data frame of draws**

**Description**

Given a tidy-format data frame of draws with a column indexing each draw, subsample the data frame to a given size based on a column indexing draws, ensuring that rows in sub-groups of a grouped data frame are sampled from the same draws.

**Usage**

```r
sample_draws(data, n, draw = ".draw")
```

**Arguments**

- **data** Data frame to sample from
- **n** The number of draws to select
- **draw** The name of the column indexing the draws
Details

`sample_draws()` makes it easier to sub-sample a grouped, tidy-format data frame of draws. On a grouped data frame, the naive approach of using `filter` with the `.draw` column will give incorrect results as it will select a different sample within each group. `sample_draws()` ensures the same sample is selected within each group.

Author(s)

Matthew Kay

Examples

```r
library(ggplot2)
library(dplyr)

if (
  require("rstanarm", quietly = TRUE) &&
  require("modelr", quietly = TRUE)
) {

  theme_set(theme_light())

  m_mpg = stan_glm(mpg ~ hp * cyl, data = mtcars,
    # 1 chain / few iterations just so example runs quickly
    # do not use in practice
    chains = 1, iter = 500)

  # draw 100 fit lines from the posterior and overplot them
  mtcars %>%
    group_by(cyl) %>%
    data_grid(hp = seq_range(hp, n = 101)) %>%
    add_fitted_draws(m_mpg) %>%
    sample_draws(100) %>%
    ggplot(aes(x = hp, y = mpg, color = ordered(cyl))) +
    geom_line(aes(y = .value, group = paste(cyl, .draw)), alpha = 0.25) +
    geom_point(data = mtcars)
}
```

---

scales  

Custom ggplot scales for `geom_slabinterval` (and derivatives)

Description

These scales allow more specific aesthetic mappings to be made when using `geom_slabinterval()` and stats/geoms based on it (like eye plots).
Usage

scale_point_colour_discrete(..., aesthetics = "point_colour")

scale_point_color_discrete(..., aesthetics = "point_colour")

scale_point_colour_continuous(
    ..., 
    aesthetics = "point_colour",
    guide = "colourbar2"
)

scale_point_color_continuous(
    ..., 
    aesthetics = "pointColour",
    guide = "colourbar2"
)

scale_point_fill_discrete(..., aesthetics = "point_fill")

scale_point_fill_continuous(
    ..., 
    aesthetics = "point_fill",
    guide = "colourbar2"
)

scale_point_alpha_continuous(..., range = c(0.1, 1))

scale_point_alpha_discrete(..., range = c(0.1, 1))

scale_point_size_continuous(..., range = c(1, 6))

scale_point_size_discrete(..., range = c(1, 6), na.translate = FALSE)

scale_interval_colour_discrete(..., aesthetics = "interval_colour")

scale_interval_color_discrete(..., aesthetics = "interval_colour")

scale_interval_colour_continuous(
    ..., 
    aesthetics = "interval_colour",
    guide = "colourbar2"
)

scale_interval_color_continuous(
    ..., 
    aesthetics = "intervalColour",
    guide = "colourbar2"
)
scale_interval_alpha_continuous(..., range = c(0.1, 1))
scale_interval_alpha_discrete(..., range = c(0.1, 1))
scale_interval_size_continuous(..., range = c(1, 6))
scale_interval_size_discrete(..., range = c(1, 6), na.translate = FALSE)
scale_interval_linetype_discrete(..., na.value = "blank")
scale_interval_linetype_continuous(...) 
scale_slab_colour_discrete(..., aesthetics = "slab_colour")
scale_slab_color_discrete(..., aesthetics = "slab_colour")
scale_slab_colour_continuous(..., aesthetics = "slab_colour", guide = "colourbar2"
)
scale_slab_color_continuous(..., aesthetics = "slab_colour", guide = "colourbar2"
)
scale_slab_fill_discrete(..., aesthetics = "slab_fill")
scale_slab_fill_continuous(..., aesthetics = "slab_fill", guide = "colourbar2")
scale_slab_alpha_continuous(..., limits = function(l) c(min(0, l[[1]]), l[[2]]), range = c(0, 1))
scale_slab_alpha_discrete(..., range = c(0.1, 1))
scale_slab_size_continuous(..., range = c(1, 6))
scale_slab_size_discrete(..., range = c(1, 6), na.translate = FALSE)
scale_slab_linetype_discrete(..., na.value = "blank")
scale_slab_linetype_continuous(...)
scale_slab_shape_discrete(..., solid = TRUE)

scale_slab_shape_continuous(...)

guide_colourbar2(...)

guide_colorbar2(...)

Arguments

... Arguments passed to underlying scale or guide functions. E.g. scale_point_color_discrete passes arguments to scale_color_discrete(). See those functions for more details.

aesthetics Names of aesthetics to set scales for.

guide Guide to use for legends for an aesthetic.

range a numeric vector of length 2 that specifies the minimum and maximum size of the plotting symbol after transformation.

na.translate In discrete scales, should we show missing values?

na.value When na.translate is true, what value should be shown?

limits One of:
  • NULL to use the default scale range
  • A numeric vector of length two providing limits of the scale. Use NA to refer to the existing minimum or maximum
  • A function that accepts the existing (automatic) limits and returns new limits Note that setting limits on positional scales will remove data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see coord_cartesian()).

solid Should the shapes be solid, TRUE, or hollow, FALSE?

Details

The following additional scales / aesthetics are defined for use with geom_slabinterval() and related geoms:

1. scale_point_color_* Point color
2. scale_point_fill_* Point fill color
3. scale_point_alpha_* Point alpha level / opacity
4. scale_point_size_* Point size
5. scale_interval_color_* Interval line color
6. scale_interval_alpha_* Interval alpha level / opacity
7. scale_interval_size_* Interval line width
8. scale_interval_linetype_* Interval line type
9. scale_slab_color_* Slab outline color
10. scale_slab_fill_* Slab fill color
11. `scale_slab_alpha_*` Slab alpha level/opacity. The default settings of `scale_slab_alpha_continuous` differ from `scale_alpha_continuous()` and are designed for gradient plots (e.g. `stat_gradientinterval()`) by ensuring that densities of 0 get mapped to 0 in the output.

12. `scale_slab_size_*` Slab outline line width

13. `scale_slab_linetype_*` Slab outline line type

14. `scale_slab_shape_*` Slab dot shape (for `geom_dotsinterval()`) See the corresponding scale documentation in ggplot for more information; e.g. `scale_color_discrete()`, `scale_color_continuous()`, etc.

Other scale functions can be used with the aesthetics/scales defined here by using the aesthetics argument to that scale function. For example, to use color brewer scales with the `point_color` aesthetic:

```r
scale_color_brewer(..., aesthetics = "point_color")
```

With continuous color scales, you may also need to provide a guide as the default guide does not work properly; this is what `guide_colorbar2` is for:

```r
scale_color_distiller(..., guide = "colorbar2", aesthetics = "point_color")
```

**Author(s)**

Matthew Kay

**See Also**

`scale_color_discrete()`, `scale_color_continuous()`, etc.

**Examples**

```r
library(dplyr)
library(ggplot2)

# This plot shows how to set multiple specific aesthetics
# NB it is very ugly and is only for demo purposes.
data.frame(distribution = "Normal(1,2)") %>%
  parse_dist(distribution) %>%
ggplot(aes(y = distribution, dist = .dist, args = .args)) +
  stat_dist_halfeyeh(
    shape = 21,  # this point shape has a fill and outline
    point_color = "red",
    point_fill = "black",
    point_alpha = .1,
    point_size = 6,
    stroke = 2,
    interval_color = "blue",
    # interval sizes are scaled from [1, 6] onto [0.6, 1.4] by default
    # see the interval_size_range option in help("geom_slabinterval")
    interval_size = 8,
    interval_linetype = "dashed",
    interval_alpha = .25,
    # fill sets the fill color of the slab (here the density)
```
stat_dist_slabinterval

Distribution + interval plots (eye plots, half-eye plots, CCDF barplots, etc) for analytical distributions (ggplot stat)

Description
Stats for computing distribution functions (densities or CDFs) + intervals for use with geom_slabinterval(). Uses dist aesthetic to specify a distribution name and arg1, ... arg9 aesthetics (or args as a list column) to specify distribution arguments.

Usage

stat_dist_slabinterval(
  mapping = NULL,
  data = NULL,
  geom = "slabinterval",
  position = "identity",
  ...,
  slab_type = c("pdf", "cdf", "ccdf"),
  p_limits = c(0.001, 0.999),
  orientation = c("vertical", "horizontal"),
  limits = NULL,
  n = 501,
  .width = c(0.66, 0.95),
  show_slab = TRUE,
  show_interval = TRUE,
  na.rm = FALSE,
  show.legend = c(size = FALSE),
  inherit.aes = TRUE
)

stat_dist_halfeye(...)

stat_dist_halfeyeh(..., orientation = "horizontal")

stat_dist_eye(..., side = "both")

stat_dist_eyeh(..., side = "both", orientation = "horizontal"
\begin{verbatim}
stat_dist_ccdfinterval(
    ..., 
    slab_type = "ccdf",
    justification = 0.5,
    side = "left",
    normalize = "none"
)

stat_dist_ccdfintervalh(
    ..., 
    slab_type = "ccdf",
    justification = 0.5,
    side = "top",
    orientation = "horizontal",
    normalize = "none"
)

stat_dist_cdfinterval(
    ..., 
    slab_type = "cdf",
    justification = 0.5,
    side = "left",
    normalize = "none"
)

stat_dist_cdfintervalh(
    ..., 
    slab_type = "cdf",
    justification = 0.5,
    side = "top",
    orientation = "horizontal",
    normalize = "none"
)

stat_dist_gradientinterval(
    mapping = NULL,
    data = NULL,
    geom = "slabinterval",
    position = "identity",
    ..., 
    justification = 0.5,
    thickness = 1,
    show.legend = c(size = FALSE, slab_alpha = FALSE),
    inherit.aes = TRUE
)

stat_dist_gradientintervalh(..., orientation = "horizontal")
\end{verbatim}
stat_dist_pointinterval(..., show_slab = FALSE)

stat_dist_pointintervalh(..., show_slab = FALSE, orientation = "horizontal")

stat_dist_interval(
  mapping = NULL,
  data = NULL,
  geom = "interval",
  position = "identity",
  ...
  show_slab = FALSE,
  show_point = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_dist_intervalh(..., orientation = "horizontal")

stat_dist_slab(
  mapping = NULL,
  data = NULL,
  geom = "slab",
  position = "identity",
  ...
  show.legend = NA,
  inherit.aes = TRUE
)

stat_dist_slabh(..., orientation = "horizontal")

Arguments

mapping Set of aesthetic mappings created by aes() or 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 Use to override the default connection between stat_slabinterval and geom_slabinterval()

position Position adjustment, either as a string, or the result of a call to a position adjustment function.
... Other arguments passed to `layer()`. They may also be arguments to the paired `geom` (e.g., `geom_pointinterval()`)

`slab_type` The type of slab function to calculate: probability density (or mass) function ("pdf"), cumulative distribution function ("cdf"), or complementary CDF ("ccdf").

`p_limits` Probability limits (as a vector of size 2) used to determine the lower and upper limits of the slab. E.g., if this is `c(.001,.999)` (the default), then a slab is drawn for the distribution from the quantile at $p = .001$ to the quantile at $p = .999$.

`orientation` Whether this geom is drawn horizontally ("horizontal") or vertically ("vertical"). When horizontal (resp. vertical), the geom uses the y (resp. x) aesthetic to identify different groups, then for each group uses the x (resp. y) aesthetic and the thickness aesthetic to draw a function as an slab, and draws points and intervals horizontally (resp. vertically) using the `xmin`, x, and `xmax` (resp. `ymin`, y, and `ymax`) aesthetics.

`limits` Manually-specified limits for the slab, as a vector of length two. These limits are combined with those computed based on `p_limits` as well as the limits defined by the scales of the plot to determine the limits used to draw the slab functions: these limits specify the maximal limits; i.e., if specified, the limits will not be wider than these (but may be narrower). Use `NA` to leave a limit alone; e.g. limits = `c(0,NA)` will ensure that the lower limit does not go below 0, but let the upper limit be determined by either `p_limits` or the scale settings.

`n` Number of points at which to evaluate `slab_function`

`.width` The `.width` argument passed to `interval_function` or `point_interval`.

`show_slab` Should the slab portion of the geom be drawn? Default `TRUE`.

`show_interval` Should the interval portion of the geom be drawn? Default `TRUE`.

`na.rm` If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`show.legend` Should this layer be included in the legends? Default is `c(size = FALSE)`, unlike most geoms, to match its common use cases. `FALSE` hides all legends, `TRUE` shows all legends, and `NA` shows only those that are mapped (the default for most geoms).

`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()`.

`side` Which side to draw the slab on. "topright", "top", and "right" are synonyms which cause the slab to be drawn on the top or the right depending on if `orientation` is "horizontal" or "vertical". "bottomleft", "bottom", and "left" are synonyms which cause the slab to be drawn on the bottom of the left depending on if `orientation` is "horizontal" or "vertical". "both" draws the slab mirrored on both sides (as in a violin plot).

`justification` Justification of the interval relative to the slab, where 0 indicates bottom/left justification and 1 indicates top/right justification (depending on `orientation`). If `justification` is `NULL` (the default), then it is set automatically based on the value of `side`: when `side` is "top"/"right" justification is set to 0, when `side` is "bottom"/"left" justification is set to 1, and when `side` is "both" justification is set to 0.5.
How to normalize heights of functions input to the thickness aesthetic. If "all" (the default), normalize so that the maximum height across all data is 1; if "panels", normalize within panels so that the maximum height in each panel is 1; if "xy", normalize within the x/y axis opposite the orientation of this geom so that the maximum height at each value of the opposite axis is 1; if "groups", normalize within values of the opposite axis and within groups so that the maximum height in each group is 1; if "none", values are taken as is with no normalization (this should probably only be used with functions whose values are in [0,1], such as CDFs).

Override for the thickness aesthetic in geom_slabinterval(): the thickness of the slab at each x/y value of the slab (depending on orientation).

Should the point portion of the geom be drawn? Default TRUE.

A highly configurable stat for generating a variety of plots that combine a "slab" that describes a distribution plus an interval. Several "shortcut" stats are provided which combine multiple options to create useful geoms, particularly eye plots (a combination of a violin plot and interval), half-eye plots (a density plus interval), and CCDF bar plots (a complementary CDF plus interval).

The shortcut stat names follow the pattern stat_dist_[name][h], where the trailing h (if present) indicates the horizontal version of the stat.

Stats include:

- **stat_dist_eye / stat_dist_eyeh**: Eye plots (violin + interval)
- **stat_dist_halfeye / stat_dist_halfeyeh**: Half-eye plots (density + interval)
- **stat_dist_ccdfinterval / stat_dist_ccdfintervalh**: CCDF bar plots (CCDF + interval)
- **stat_dist_cdfinterval / stat_dist_cdfintervalh**: CDF bar plots (CDF + interval)
- **stat_dist_gradientinterval / stat_dist_gradientintervalh**: Density gradient + interval plots
- **stat_dist_pointinterval / stat_dist_pointintervalh**: Point + interval plots
- **stat_dist_interval / stat_dist_intervalh**: Interval plots

These stats expect a dist aesthetic to specify a distribution name and arg1, ... arg9 aesthetics (or args as a list column) to specify distribution arguments. Distribution names should correspond to R functions that have "p", "q", and "d" functions; e.g. "norm" is a valid distribution name because R defines the pnorm(), qnorm(), and dnorm() functions for Normal distributions.

See the parse_dist() function for a useful way to generate dist and args values from human-readable distribution specs (like "normal(0,1)"). Such specs are also produced by other packages (like the brms::get_prior() function in brms); thus, parse_dist() combined with the stats described here can help you visualize the output of those functions.

These stats support the following aesthetics:
In addition, in their default configuration (paired with `geom_slabinterval()`) the following aesthetics are supported by the underlying geom:

- `datatype`
- `alpha`
- `colour`
- `linetype`
- `fill`
- `shape`
- `stroke`
- `pointColour`
- `point_fill`
- `point_alpha`
- `point_size`
- `size`
- `interval_colour`
- `interval_alpha`
- `interval_size`
- `interval_linetype`
- `slab_size`
- `slab_colour`
• slab_fill
• slab_alpha
• slab_linetype
• y
• ymin
• ymax
• x
• xmin
• xmax
• width
• height
• thickness
• group

See examples of some of these aesthetics in action in vignette("slabinterval"). Learn more about the sub-geom aesthetics (like interval_color) in the scales documentation. Learn more about basic ggplot aesthetics in vignette("ggplot2-specs").

Computed Variables

• x or y: For slabs, the input values to the slab function. For intervals, the point summary from the interval function. Whether it is x or y depends on orientation
• xmin or ymin: For intervals, the lower end of the interval from the interval function.
• xmax or ymax: For intervals, the upper end of the interval from the interval function.
• f: For slabs, the output values from the slab function.

See Also

See geom_slabinterval() for more information on the geom these stats use by default and some of the options they have. See stat_sample_slabinterval() for the versions of these stats that can be used on samples. See vignette("slabinterval") for a variety of examples of use.

Examples

library(dplyr)
library(ggplot2)

tribble(~group, ~subgroup, ~mean, ~sd,
"a", "h", 5, 1,
"b", "h", 7, 1.5,
"c", "h", 8, 1,
"c", "i", 9, 1,
"c", "j", 7, 1)

%>%
stat_interval

Multiple probability interval plots (ggplot stat)

Description

A combination of `stat_sample_slabinterval()` and `geom_slabinterval()` with sensible defaults. While the corresponding geoms are intended for use on data frames that have already been summarized using a `point_interval()` function, these stats are intended for use directly on data frames of draws, and will perform the summarization using a `point_interval()` function.

Usage

```r
stat_interval(
  mapping = NULL,
  data = NULL,
  geom = "interval",
  position = "identity",
  ...,  
  orientation = "vertical",
  interval_function = NULL,
  interval_args = list(),
  point_interval = median_qi,
  .width = c(0.5, 0.8, 0.95),
  show_point = FALSE,
  show_slab = FALSE,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE,
  .prob,
  fun.data,
  fun.args
)
```
stat_intervalh(
    mapping = NULL,
    data = NULL,
    geom = "interval",
    position = "identity",
    ...,
    orientation = "horizontal",
    interval_function = NULL,
    interval_args = list(),
    point_interval = median_qi,
    .width = c(0.5, 0.8, 0.95),
    show_point = FALSE,
    show_slab = FALSE,
    na.rm = FALSE,
    show.legend = NA,
    inherit.aes = NA,
    .prob,
    fun.data,
    fun.args
)

Arguments

mapping  Set of aesthetic mappings created by `aes()` or `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  Use to override the default connection between `stat_slabinterval` and `geom_slabinterval()`

position  Position adjustment, either as a string, or the result of a call to a position adjustment function.

...  Other arguments passed to `layer()`. They may also be arguments to the paired geom (e.g., `geom_pointinterval()`)  

orientation  Whether this geom is drawn horizontally ("horizontal") or vertically ("vertical"). When horizontal (resp. vertical), the geom uses the y (resp. x) aesthetic to identify different groups, then for each group uses the x (resp. y) aesthetic and the thickness aesthetic to draw a function as an slab, and draws points and intervals horizontally (resp. vertically) using the xmin, x, and xmax (resp. ymin, y, and ymax) aesthetics.
stat_interval

interval_function
Custom function for generating intervals (for most common use cases the point_interval argument will be easier to use). This function takes a data frame of aesthetics and a .width parameter (a vector of interval widths), and returns a data frame with columns .width (from the .width vector), .value (point summary) and .lower and .upper (endpoints of the intervals, given the .width). Output will be converted to the appropriate x- or y-based aesthetics depending on the value of orientation. If interval_function is NULL, point_interval is used instead.

interval_args
Additional arguments passed to interval_function or point_interval.

point_interval
A function from the point_interval() family (e.g., median_qi, mean_qi, etc). This function should take in a vector of value, and should obey the .width and .simple_names parameters of point_interval() functions, such that when given a vector with .simple_names = TRUE should return a data frame with variables .value, .lower, .upper, and .width. Output will be converted to the appropriate x- or y-based aesthetics depending on the value of orientation. See the point_interval() family of functions for more information.

.width
The .width argument passed to interval_function or point_interval.

show_point
Should the point portion of the geom be drawn? Default TRUE.

show_slab
Should the slab portion of the geom be drawn? Default TRUE.

na.rm
If FALSE, the default, missing values are removed with a warning. If TRUE, missing values are silently removed.

show.legend
Should this layer be included in the legends? Default is c(size = FALSE), unlike most geoms, to match its common use cases. FALSE hides all legends, TRUE shows all legends, and NA shows only those that are mapped (the default for most geoms).

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().

.prob
Deprecated. Use .width instead.

fun.data
Deprecated. Use point_interval or interval_function instead.

fun.args
Deprecated. Use interval_args instead.

Aesthetics
These stats support the following aesthetics:

- colour
- datatype
- x
- y
- thickness
- group
In addition, in their default configuration (paired with `geom_interval()`) the following aesthetics are supported by the underlying geom:

- datatype
- alpha
- colour
- linetype
- fill
- shape
- stroke
- point_colour
- point_fill
- point_alpha
- point_size
- size
- interval_colour
- interval_alpha
- interval_size
- interval_linetype
- slab_size
- slab_colour
- slab_fill
- slab_alpha
- slab_linetype
- y
- ymin
- ymax
- x
- xmin
- xmax
- width
- height
- thickness
- group

See examples of some of these aesthetics in action in vignette("slabinterval"). Learn more about the sub-geom aesthetics (like interval_color) in the `scales` documentation. Learn more about basic ggplot aesthetics in vignette("ggplot2-specs").
See Also

See `geom_interval()` / `geom_intervalh()` for the geom versions, intended for use on points and intervals that have already been summarized using a `point_interval()` function. See `stat_pointinterval()` / `stat_pointintervalh()` for a similar stat intended for point summaries and intervals. See `stat_sample_slabinterval()` for a variety of other stats that combine intervals with densities and CDFs. See `geom_slabinterval()` for the geom that these geoms wrap. All parameters of that geom are available to these geoms.

Examples

```r
library(magrittr)
library(ggplot2)

data(RankCorr, package = "tidybayes")

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(y = i, x = u_tau)) +
  stat_intervalh() +
  scale_color_brewer()

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(x = i, y = u_tau)) +
  stat_interval() +
  scale_color_brewer()
```

Description

A combination of `stat_slabinterval()` and `geom_lineribbon()` with sensible defaults. While `geom_lineribbon` is intended for use on data frames that have already been summarized using a `point_interval()` function, `stat_lineribbon` is intended for use directly on data frames of draws, and will perform the summarization using a `point_interval()` function; `stat_dist_lineribbon` is intended for use on analytical distributions through the `dist`, `arg1`, ... `arg9`, and `args` aesthetics.

Usage

```r
stat_lineribbon(
  mapping = NULL,
  data = NULL,
  geom = "lineribbon",
  position = "identity",
  ...,
```
\texttt{stat_lineribbon}

\begin{verbatim}
  interval_function = NULL,
  interval_args = list(),
  point_interval = median_qi,
  .width = c(0.5, 0.8, 0.95),
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE,
  .prob,
  fun.data,
  fun.args
)

  stat_dist_lineribbon(
    mapping = NULL,
    data = NULL,
    geom = "lineribbon",
    position = "identity",
    ...,
    n = 501,
    .width = c(0.5, 0.8, 0.95),
    na.rm = FALSE,
    show.legend = NA,
    inherit.aes = TRUE
)
\end{verbatim}

\textbf{Arguments}

\begin{description}
\item[mapping] Set of aesthetic mappings created by \texttt{aes()} or \texttt{aes_().} If specified and \texttt{inherit.aes = TRUE} (the default), it is combined with the default mapping at the top level of the plot. You must supply \texttt{mapping} if there is no plot mapping.
\item[data] The data to be displayed in this layer. There are three options: If \texttt{NULL}, the default, the data is inherited from the plot data as specified in the call to \texttt{ggplot().}
A \texttt{data.frame}, or other object, will override the plot data. All objects will be fortified to produce a data frame. See \texttt{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 \texttt{data.frame}, and will be used as the layer data. A function can be created from a \texttt{formula} (e.g. \texttt{~ head(.x,10)}).
\item[geom] Use to override the default connection between \texttt{geom_lineribbon} and \texttt{stat_lineribbon}.
\item[position] Position adjustment, either as a string, or the result of a call to a position adjustment function.
\item[...] Other arguments passed to \texttt{layer().} They may also be arguments to the paired \texttt{geom} (e.g. \texttt{geom_pointinterval()})
\item[interval_function] Custom function for generating intervals (for most common use cases the \texttt{point_interval} argument will be easier to use). This function takes a data frame of aesthetics
and a .width parameter (a vector of interval widths), and returns a data frame with columns .width (from the .width vector), .value (point summary) and .lower and .upper (endpoints of the intervals, given the .width). Output will be converted to the appropriate x- or y-based aesthetics depending on the value of orientation. If interval_function is NULL, point_interval is used instead.

interval_args Additional arguments passed to interval_function or point_interval.

point_interval A function from the point_interval() family (e.g., median_qi, mean_qi, etc). This function should take in a vector of value, and should obey the .width and .simple_names parameters of point_interval() functions, such that when given a vector with .simple_names = TRUE should return a data frame with variables .value, .lower, .upper, and .width. Output will be converted to the appropriate x- or y-based aesthetics depending on the value of orientation. See the point_interval() family of functions for more information.

.width The .width argument passed to interval_function or point_interval.

na.rm If FALSE, the default, missing values are removed with a warning. If TRUE, missing values are silently removed.

show.legend Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

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().

.prob Deprecated. Use .width instead.

fun.data Deprecated. Use point_interval or interval_function instead.

fun.args Deprecated. Use interval_args instead.

n Number of points at which to evaluate slab_function

See Also

See geom_lineribbon() for the geom version, intended for use on points and intervals that have already been summarized using a point_interval() function. See stat_pointinterval() / stat_pointintervalh() for a similar stat intended for point summaries and intervals.

Examples

library(dplyr)
library(ggplot2)

tibble(x = 1:10) %>%
  group_by_all() %>%
do(tibble(y = rnorm(100, .$x))) %>%
ggplot(aes(x = x, y = y)) +
  stat_lineribbon() +
  scale_fill_brewer()

tibble(
stat_pointinterval

```r
x = 1:10,
sd = seq(1, 3, length.out = 10)
```%>%
```r
ggplot(aes(x = x, dist = "norm", arg1 = x, arg2 = sd)) +
  stat_dist_lineribbon() +
  scale_fill_brewer()
```

---

**stat_pointinterval**  
*Point summary + multiple probability interval plots (ggplot stat)*

### Description

A combination of `stat_sample_slabinterval()` and `geom_slabinterval()` with sensible defaults. While the corresponding geoms are intended for use on data frames that have already been summarized using a `point_interval()` function, these stats are intended for use directly on data frames of draws, and will perform the summarization using a `point_interval()` function.

### Usage

```r
stat_pointinterval(
  mapping = NULL,
  data = NULL,
  geom = "pointinterval",
  position = "identity",
  ...,
  orientation = "vertical",
  interval_function = NULL,
  interval_args = list(),
  point_interval = median_qi,
  .width = c(0.66, 0.95),
  show_slab = FALSE,
  na.rm = FALSE,
  show.legend = c(size = FALSE),
  inherit.aes = TRUE,
  .prob,
  fun.data,
  fun.args
)
```

```r
stat_pointintervalh(
  mapping = NULL,
  data = NULL,
  geom = "pointintervalh",
  position = "identity",
  ...,
  orientation = "horizontal",
)```

---
interval_function = NULL,
interval_args = list(),
point_interval = median_qi,
.width = c(0.66, 0.95),
show_slab = FALSE,
na.rm = FALSE,
show.legend = c(size = FALSE),
inherit.aes = TRUE,
.probs,
fun.data,
fun.args
)

Arguments

mapping  
Set of aesthetic mappings created by aes() or 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  
Use to override the default connection between stat_slabinterval and geom_slabinterval()

position  
Position adjustment, either as a string, or the result of a call to a position adjustment function.

...  
Other arguments passed to layer(). They may also be arguments to the paired geom (e.g., geom_pointinterval())

orientation  
Whether this geom is drawn horizontally ("horizontal") or vertically ("vertical"). When horizontal (resp. vertical), the geom uses the y (resp. x) aesthetic to identify different groups, then for each group uses the x (resp. y) aesthetic and the thickness aesthetic to draw a function as an slab, and draws points and intervals horizontally (resp. vertically) using the xmin, x, and xmax (resp. ymin, y, and ymax) aesthetics.

interval_function  
Custom function for generating intervals (for most common use cases the point_interval argument will be easier to use). This function takes a data frame of aesthetics and a .width parameter (a vector of interval widths), and returns a data frame with columns .width (from the .width vector), .value (point summary) and .lower and .upper (endpoints of the intervals, given the .width). Output will be converted to the appropriate x- or y-based aesthetics depending on the value of orientation. If interval_function is NULL, point_interval is used instead.
**stat_pointinterval**

interval_args  Additional arguments passed to interval_function or point_interval.

point_interval  A function from the point_interval() family (e.g., median_qi, mean_qi, etc). This function should take in a vector of value, and should obey the .width and .simple_names parameters of point_interval() functions, such that when given a vector with .simple_names = TRUE should return a data frame with variables .value, .lower, .upper, and .width. Output will be converted to the appropriate x- or y-based aesthetics depending on the value of orientation. See the point_interval() family of functions for more information.

.width  The .width argument passed to interval_function or point_interval.

show_slab  Should the slab portion of the geom be drawn? Default TRUE.

na.rm  If FALSE, the default, missing values are removed with a warning. If TRUE, missing values are silently removed.

show.legend  Should this layer be included in the legends? Default is c(size = FALSE), unlike most geoms, to match its common use cases. FALSE hides all legends, TRUE shows all legends, and NA shows only those that are mapped (the default for most geoms).

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().

.probab  Deprecated. Use .width instead.

fun.data  Deprecated. Use point_interval or interval_function instead.

fun.args  Deprecated. Use interval_args instead.

### Aesthetics

These stats support the following aesthetics:

- datatype
- x
- y
- thickness
- size
- group

In addition, in their default configuration (paired with geom_pointinterval()) the following aesthetics are supported by the underlying geom:

- datatype
- alpha
- colour
- linetype
- fill
- shape
- stroke
• point_colour
• point_fill
• point_alpha
• point_size
• size
• interval_colour
• interval_alpha
• interval_size
• interval_linetype
• slab_size
• slab_colour
• slab_fill
• slab_alpha
• slab_linetype
• y
• ymin
• ymax
• x
• xmin
• xmax
• width
• height
• thickness
• group

See examples of some of these aesthetics in action in vignette("slabinterval"). Learn more about the sub-geom aesthetics (like interval_color) in the scales documentation. Learn more about basic ggplot aesthetics in vignette("ggplot2-specs").

See Also

See geom_pointinterval() / geom_pointintervalh() for the geom versions, intended for use on points and intervals that have already been summarized using a point_interval() function.
See stat_interval() / stat_intervalh() for a similar stat intended for intervals without point summaries. See stat_sample_slabinterval() for a variety of other stats that combine intervals with densities and CDFs.
See geom_pointinterval() / geom_pointintervalh() for the geom versions, intended for use on points and intervals that have already been summarized using a point_interval() function.
See stat_interval() / stat_intervalh() for a similar stat intended for intervals without point summaries. See stat_sample_slabinterval() for a variety of other stats that combine intervals with densities and CDFs. See geom_slabinterval() for the geom that these geoms wrap. All parameters of that geom are available to these geoms.
Examples

```r
library(magrittr)
library(ggplot2)

data(RankCorr, package = "tidybayes")

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(y = i, x = u_tau)) +
  stat_pointintervalh(.width = c(0.66, 0.95))

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(x = i, y = u_tau)) +
  stat_pointinterval(.width = c(0.66, 0.95))
```

Description

Stats for computing densities and CDFs + intervals from samples for use with `geom_slabinterval()`. Useful for creating eye plots, half-eye plots, CCDF bar plots etc.

Usage

```r
stat_sample_slabinterval(
  mapping = NULL,
  data = NULL,
  geom = "slabinterval",
  position = "identity",
  ...
  slab_type = c("pdf", "cdf", "ccdf", "histogram"),
  adjust = 1,
  trim = TRUE,
  breaks = "Sturges",
  outline_bars = FALSE,
  orientation = c("vertical", "horizontal"),
  limits = NULL,
  n = 501,
  interval_function = NULL,
  interval_args = list(),
  point_interval = median_qi,
  .width = c(0.66, 0.95),
)```
na.rm = FALSE,
show.legend = c(size = FALSE),
inherit.aes = TRUE
)

stat_halfeye(...)

stat_halfeyeh(..., orientation = "horizontal")

stat_eye(..., side = "both")

stat_eyeh(..., side = "both", orientation = "horizontal")

stat_ccdfinterval(
  ..., 
  slab_type = "ccdf", 
  justification = 0.5, 
  side = "left", 
  normalize = "none"
)

stat_ccdfintervalh(
  ..., 
  slab_type = "ccdf", 
  justification = 0.5, 
  side = "top", 
  orientation = "horizontal", 
  normalize = "none"
)

stat_cdfinterval(
  ..., 
  slab_type = "cdf", 
  justification = 0.5, 
  side = "left", 
  normalize = "none"
)

stat_cdfintervalh(
  ..., 
  slab_type = "cdf", 
  justification = 0.5, 
  side = "top", 
  orientation = "horizontal", 
  normalize = "none"
)

stat_gradientinterval(}
stat_sample_slabinterval

mapping = NULL,
data = NULL,
geom = "slabinterval",
position = "identity",
..., justification = 0.5,
thickness = 1,
show.legend = c(size = FALSE, slab_alpha = FALSE),
inherit.aes = TRUE
)

stat_gradientintervalh(..., orientation = "horizontal")

stat_histinterval(..., slab_type = "histogram")

stat_histintervalh(..., slab_type = "histogram", orientation = "horizontal")

stat_slab(
mapping = NULL,
data = NULL,
geom = "slab",
position = "identity",
..., show.legend = NA,
inherit.aes = TRUE
)

stat_slabh(..., orientation = "horizontal")

Arguments

mapping Set of aesthetic mappings created by aes() or 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 Use to override the default connection between stat_slabinterval and geom_slabinterval() position Position adjustment, either as a string, or the result of a call to a position adjustment function.
... Other arguments passed to layer(). They may also be arguments to the paired geom (e.g., geom_pointinterval())
<table>
<thead>
<tr>
<th>Slab Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slab_type</td>
<td>The type of slab function to calculate: probability density (or mass) function (&quot;pdf&quot;), cumulative distribution function (&quot;cdf&quot;), complementary CDF (&quot;ccdf&quot;), or histogram (&quot;histogram&quot;).</td>
</tr>
<tr>
<td>adjust</td>
<td>If slab_type is &quot;pdf&quot;, bandwidth for the density estimator is adjusted by multiplying it by this value. See <code>density()</code> for more information.</td>
</tr>
<tr>
<td>trim</td>
<td>If slab_type is &quot;pdf&quot;, should the density estimate be trimmed to the range of the input data? Default TRUE.</td>
</tr>
<tr>
<td>breaks</td>
<td>If slab_type is &quot;histogram&quot;, the breaks parameter that is passed to <code>hist()</code> to determine where to put breaks in the histogram.</td>
</tr>
<tr>
<td>outline_bars</td>
<td>If slab_type is &quot;histogram&quot;, outline_bars determines if outlines in between the bars are drawn when the slab_color aesthetic is used. If FALSE (the default), the outline is drawn only along the tops of the bars; if TRUE, outlines in between bars are also drawn.</td>
</tr>
<tr>
<td>orientation</td>
<td>Whether this geom is drawn horizontally (&quot;horizontal&quot;) or vertically (&quot;vertical&quot;). When horizontal (resp. vertical), the geom uses the y (resp. x) aesthetic to identify different groups, then for each group uses the x (resp. y) aesthetic and the thickness aesthetic to draw a function as an slab, and draws points and intervals horizontally (resp. vertically) using the xmin, x, and xmax (resp. ymin, y, and ymax) aesthetics.</td>
</tr>
<tr>
<td>limits</td>
<td>Limits for slab_function, as a vector of length two. These limits are combined with those computed by the limits_function as well as the limits defined by the scales of the plot to determine the limits used to draw the slab functions: these limits specify the maximal limits; i.e., if specified, the limits will not be wider than these (but may be narrower). Use NA to leave a limit alone; e.g. limits = c(0,NA) will ensure that the lower limit does not go below 0.</td>
</tr>
<tr>
<td>n</td>
<td>Number of points at which to evaluate slab_function</td>
</tr>
<tr>
<td>interval_function</td>
<td>Custom function for generating intervals (for most common use cases the point_interval argument will be easier to use). This function takes a data frame of aesthetics and a .width parameter (a vector of interval widths), and returns a data frame with columns .width (from the .width vector), .value (point summary) and .lower and .upper (endpoints of the intervals, given the .width). Output will be converted to the appropriate x- or y-based aesthetics depending on the value of orientation. If interval_function is NULL, point_interval is used instead.</td>
</tr>
<tr>
<td>interval_args</td>
<td>Additional arguments passed to interval_function or point_interval.</td>
</tr>
<tr>
<td>point_interval</td>
<td>A function from the point_interval() family (e.g., median_qi, mean_qi, etc). This function should take in a vector of value, and should obey the .width and .simple_names parameters of point_interval() functions, such that when given a vector with .simple_names = TRUE should return a data frame with variables .value, .lower, .upper, and .width. Output will be converted to the appropriate x- or y-based aesthetics depending on the value of orientation. See the point_interval() family of functions for more information.</td>
</tr>
<tr>
<td>.width</td>
<td>The .width argument passed to interval_function or point_interval.</td>
</tr>
<tr>
<td>na.rm</td>
<td>If FALSE, the default, missing values are removed with a warning. If TRUE, missing values are silently removed.</td>
</tr>
</tbody>
</table>
**stat_sample_slabinterval**

- **show.legend** Should this layer be included in the legends? Default is `c(size = FALSE)`, unlike most geoms, to match its common use cases. `FALSE` hides all legends, `TRUE` shows all legends, and `NA` shows only those that are mapped (the default for most geoms).

- **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()`.

- **side** Which side to draw the slab on. "topright", "top", and "right" are synonyms which cause the slab to be drawn on the top or the right depending on if orientation is "horizontal" or "vertical". "bottomleft", "bottom", and "left" are synonyms which cause the slab to be drawn on the bottom of the left depending on if orientation is "horizontal" or "vertical". "both" draws the slab mirrored on both sides (as in a violin plot).

- **justification** Justification of the interval relative to the slab, where 0 indicates bottom/left justification and 1 indicates top/right justification (depending on orientation). If justification is `NULL` (the default), then it is set automatically based on the value of `side`: when `side` is "top"/"right" justification is set to 0, when `side` is "bottom"/"left" justification is set to 1, and when `side` is "both" justification is set to 0.5.

- **normalize** How to normalize heights of functions input to the thickness aesthetic. If "all" (the default), normalize so that the maximum height across all data is 1; if "panels", normalize within panels so that the maximum height in each panel is 1; if "xy", normalize within the x/y axis opposite the orientation of this geom so that the maximum height at each value of the opposite axis is 1; if "groups", normalize within values of the opposite axis and within groups so that the maximum height in each group is 1; if "none", values are taken as is with no normalization (this should probably only be used with functions whose values are in [0,1], such as CDFs).

- **thickness** Override for the thickness aesthetic in `geom_slabinterval()`: the thickness of the slab at each x/y value of the slab (depending on orientation).

**Details**

A highly configurable stat for generating a variety of plots that combine a "slab" that summarizes a sample plus an interval. Several "shortcut" stats are provided which combine multiple options to create useful geoms, particularly eye plots (a combination of a violin plot and interval), half-eye plots (a density plus interval), and CCDF bar plots (a complementary CDF plus interval). These can be handy for visualizing posterior distributions in Bayesian inference, amongst other things.

The shortcut stat names follow the pattern `stat_[name]h`, where the trailing `h` (if present) indicates the horizontal version of the stat.

Stats include:

- **stat_eye / stat_eyeh**: Eye plots (violin + interval)
- **stat_halfeye / stat_halfeyeh**: Half-eye plots (density + interval)
- **stat_ccdfinterval / stat_ccdfintervalh**: CCDF bar plots (CCDF + interval)
- **stat_cdfinterval / stat_cdfintervalh**: CDF bar plots (CDF + interval)
• **stat_gradientinterval / stat_gradientintervalh**: Density gradient + interval plots
• **stat_histinterval / stat_histintervalh**: Histogram + interval plots
• **stat_pointinterval / stat_pointintervalh**: Point + interval plots
• **stat_interval / stat_intervalh**: Interval plots

**Aesthetics**

These stats support the following aesthetics:

• x
• y
• datatype
• thickness
• size
• group

In addition, in their default configuration (paired with `geom_slabinterval()`) the following aesthetics are supported by the underlying geom:

• datatype
• alpha
• colour
• linetype
• fill
• shape
• stroke
• point_colour
• point_fill
• point_alpha
• point_size
• size
• interval_colour
• interval_alpha
• interval_size
• interval_linetype
• slab_size
• slab_colour
• slab_fill
• slab_alpha
• slab_linetype
• y
• ymin
• ymax
• x
• xmin
• xmax
• width
• height
• thickness
• group

See examples of some of these aesthetics in action in vignette("slabinterval"). Learn more about the sub-geom aesthetics (like interval_color) in the scales documentation. Learn more about basic ggplot aesthetics in vignette("ggplot2-specs").

Computed Variables

• x or y: For slabs, the input values to the slab function. For intervals, the point summary from the interval function. Whether it is x or y depends on orientation
• xmin or ymin: For intervals, the lower end of the interval from the interval function.
• xmax or ymax: For intervals, the upper end of the interval from the interval function.
• f: For slabs, the output values from the slab function.
• n: For slabs, the number of data points summarized into that slab.

See Also

See geom_slabinterval() for more information on the geom these stats use by default and some of the options they have. See stat_dist_slabinterval() for the versions of these stats that can be used on analytical distributions. See vignette("slabinterval") for a variety of examples of use.

Examples

library(dplyr)
library(tidyr)
library(ggplot2)

# consider the following example data:
set.seed(1234)
df = tribble(~group, ~subgroup, ~value,
           "a", "h", rnorm(500, mean = 5),
           "b", "h", rnorm(500, mean = 7, sd = 1.5),
           "c", "h", rnorm(500, mean = 8),
           "c", "i", rnorm(500, mean = 9),
           "c", "j", rnorm(500, mean = 7)
) %>%
unnest(value)

# here are vertical eyes:
df %>%
  ggplot(aes(x = group, y = value)) +
  stat_eye()

# note the sample size is not automatically incorporated into the
# area of the densities in case one wishes to plot densities against
# a reference (e.g. a prior generated by a stat_dist_... function).
# But you may wish to account for sample size if using these geoms
# for something other than visualizing posteriors; in which case
# you can use stat(f*n):
df %>%
  ggplot(aes(x = group, y = value)) +
  stat_eye(aes(thickness = stat(f*n)))

# see vignette("slabinterval") for many more examples.

---

**theme_tidybayes**

*Simple, light ggplot2 theme for tidybayes*

---

**Description**

A simple, relatively minimalist ggplot2 theme, and some helper functions to go with it.

**Usage**

```r
theme_tidybayes()

facet_title_left_horizontal()

facet_title_right_horizontal()

axis_titles_bottom_left()
```

**Value**

A ggplot2 theme

This is a relatively minimalist ggplot2 theme, intended to be used for making publication-ready plots. It is currently based on `ggplot2::theme_light()`.

A word of warning: this theme may (and very likely will) change in the future as I tweak it to my taste.

**Author(s)**

Matthew Kay
See Also

`ggplot2::theme()`, `ggplot2::theme_set()`

Examples

```r
library(ggplot2)
theme_set(theme_tidybayes())
```

Description

These functions translate tidybayes-style tidy data frames of draws to/from different tidy data frame formats (each format using a different naming scheme).

Usage

```r
to_broom_names(data)
from_broom_names(data)
to_ggmcmc_names(data)
from_ggmcmc_names(data)
```

Arguments

- `data` A data frame to translate.

Details

Function prefixed with `to_` translate from the tidybayes format to another format, functions prefixed with `from_` translate from that format back to the tidybayes format. Formats include:

- `to_broom_names()` / `from_broom_names()`:
  - `.variable` $\rightarrow$ `term`
  - `.value` $\rightarrow$ `estimate`
  - `.prediction` $\rightarrow$ `.fitted`
  - `.lower` $\rightarrow$ `conf.low`
  - `.upper` $\rightarrow$ `conf.high`
to_ggmcmc_names() / from_ggmcmc_names():

• `.chain` <-> `Chain`
• `.iteration` <-> `Iteration`
• `.variable` <-> `Parameter`
• `.value` <-> `value`

Value

A data frame with (possibly) new names in some columns, according to the translation scheme above.

Author(s)

Matthew Kay

Examples

```r
library(magrittr)

data(line, package = "coda")

line %>%
  gather_draws(alpha, beta, sigma) %>%
  median_qi() %>%
  to_broom_names()
```

Description

Deprecated functions, arguments, and column names in tidybayes

Deprecated Functions

Several deprecated versions of functions use slightly different output formats (e.g., they use names like `term` and `estimate` where new functions use `.variable` and `.value`; or they set `.iteration` even when iteration information is not available — new functions always set `.draw` but may not set `.iteration`), so be careful when upgrading to new function names. See *Deprecated Arguments and Column Names*, below, for more information.

Deprecated functions are:
• `spread_samples`, `extract_samples`, and `tidy_samples` are deprecated names for `spread_draws()`. The spread/gather terminology better distinguishes the resulting data frame format, and `draws` is more correct terminology than `samples` for describing multiple realizations from a posterior distribution.

• `gather_samples` is a deprecated name for `gather_draws()`, reflecting a package-wide move to using `draws` instead of `samples` for describing multiple realizations from a distribution.

• `unspread_samples` is a deprecated name for `unspread_draws()`, reflecting a package-wide move to using `draws` instead of `samples` for describing multiple realizations from a distribution.

• `ungather_samples` is a deprecated name for `ungather_draws()`, reflecting a package-wide move to using `draws` instead of `samples` for describing multiple realizations from a distribution.

• `fitted_samples` / `add_fitted_samples` are deprecated names for `fitted_draws()` / `add_fitted_draws()`, reflecting a package-wide move to using `draws` instead of `samples` for describing multiple realizations from a distribution.

• `predicted_samples` / `add_predicted_samples` are deprecated names for `predicted_draws()` / `add_predicted_draws()`, reflecting a package-wide move to using `draws` instead of `samples` for describing multiple realizations from a distribution.

• `gather_lsmeans_samples` and `gather_emmeans_samples` are deprecated aliases for `gather_emmeans_draws()`. The new name (estimated marginal means) is more appropriate for Bayesian models than the old name (least-squares means), and reflects the naming of the newer `emmeans` package. It also reflects a package-wide move to using `draws` instead of `samples` for describing multiple realizations from a distribution.

• `as_sample_tibble` and `as_sample_data_frame` are deprecated aliases for `tidy_draws()`. The original intent of `as_sample_tibble` was to be used primarily internally (hence its less user-friendly name); however, increasingly I have come across use cases of `tidy_draws` that warrant a more user-friendly name. It also reflects a package-wide move to using `draws` instead of `samples` for describing multiple realizations from a distribution.

• `ggeye` is deprecated: for a package whose goal is flexible and customizable visualization, monolithic functions are inflexible and do not sufficiently capitalize on users’ existing knowledge of `ggplot`; instead, I think it is more flexible to design geoms and stats that can used within a complete `ggplot` workflow. `geom_eyeh()` offers a horizontal eye plot geom that can be used instead of `ggeye`.

### Deprecated Arguments and Column Names

Versions of tidybayes before version 1.0 used a different naming scheme for several arguments and output columns.

Deprecated arguments and column names are:

• `term` is now `.variable`
• `estimate` is now `.value`
• `pred` is now `.prediction`
• `conf.low` is now `.lower`
• `conf.high` is now `.upper`
tidybayes-models

- .prob is now .width
- The .draw column was added, and should be used instead of .chain and .iteration to uniquely identify draws when you do not care about chains. (.chain and .iteration are still provided for identifying draws within chains, if desired).

To translate to/from the old naming scheme in output, use to_broom_names() and from_broom_names().

Many of these names were updated in version 1.0 in order to make terminology more consistent and in order to satisfy these criteria:

- Ignore compatibility with broom names on the assumption an adapter function can be created.
- Use names that could be compatible with frequentist approaches (hence .width instead of .prob).
- Always precede with "." to avoid collisions with variable names in models.
- No abbreviations (remembering if something is abbrevia-ted or not can be a pain).
- No two-word names (multi-word names can always be standardized on and used in documentation, but I think data frame output should be succinct).
- Names should be nouns (I made an exception for lower/upper because they are common).

Author(s)

Matthew Kay

**tidybayes-models**  
*Models supported by tidybayes*

**Description**

Tidybayes supports two classes of models and sample formats: Models/formats that provide prediction functions, and those that do not.

**All Supported Models/Sample Formats**

All supported models/formats support the base tidybayes sample extraction functions, such as tidy_draws(), spread_draws(), and gather_draws(). These models/formats include:

- rstan models
- brms::brm() models
- rstanarm models
- runjags::runjags() models
- rjags::jags.model() models, if sampled using rjags::coda.samples()
- jagsUI::jags() models
- MCMCglmm::MCMCglmm() models
- coda::mcmc() and coda::mcmc.list() objects, which are output by several model types.
tidy_draws

- Any object with an implementation of `coda::as.mcmc.list()`. For a list of those available in your environment, run `methods(as.mcmc.list)`

If you install the tidybayes.rethinking package (available at [https://github.com/mjskay/tidybayes.rethinking](https://github.com/mjskay/tidybayes.rethinking)), map and map2stan models from the `rethinking` package are also supported.

Models Supporting Prediction

In addition, the **following models support fit and prediction** extraction functions, such as `add_fitted_draws()` and `add_predicted_draws()`:

- `brms::brm()` models
- `rstanarm` models

If your model type is not in the above list, you may still be able to use the `add_draws()` function to turn matrices of predictive draws (or fit draws) into tidy data frames.

If you install the tidybayes.rethinking package, models from the `rethinking` package are also supported. Note that in tidybayes.rethinking, `tidy_link` takes the place of `add_fitted_draws()` and `tidy_sim` takes the place of `add_predicted_draws()`.

Extending tidybayes

To include basic support for new models, one need only implement the `tidy_draws()` generic function for that model.

To include support for estimation and prediction, one must implement the `fitted_draws()` and `predicted_draws()` generic functions.

---

**tidy_draws**

*Get a sample of posterior draws from a model as a tibble*

**Description**

Extract draws from a Bayesian fit into a wide-format data frame with `.chain`, `.iteration`, and `.draw` column, as well as all variables as columns. While this function can be useful for quick glances at models (especially combined with `gather_variables()` and `median_qi()`), it is generally speaking not as useful as `spread_draws()` or `gather_draws()` for most applications, and is mainly used internally (see Details).

**Usage**

```r
 tidy_draws(model)
```

## Default S3 method:
tidy_draws(model)

## S3 method for class 'data.frame'

tidy_draws(model)

## S3 method for class 'mcmc.list'
tidy_draws(model)

## S3 method for class 'stanfit'
 tidy_draws(model)

## S3 method for class 'stanreg'
 tidy_draws(model)

## S3 method for class 'runjags'
 tidy_draws(model)

## S3 method for class 'jagsUI'
 tidy_draws(model)

## S3 method for class 'brmsfit'
 tidy_draws(model)

## S3 method for class 'matrix'
 tidy_draws(model)

## S3 method for class 'MCMCglmm'
 tidy_draws(model)

### Arguments

- **model** A supported Bayesian model fit object. See `tidybayes-models()` for a list of supported models.

### Details

In practice, apart from quick looks at a model you will probably not call this directly; `spread_draws()` or `gather_draws()`, which are build on top of this function, provide support for extracting variable dimensions are so are often more useful.

To provide support for new models in tidybayes, you must provide an implementation of this function or an implementation of `coda::as.mcmc.list()` (tidy_draws should work on any model with an implementation of `coda::as.mcmc.list()`)

`tidy_draws` can be applied to a data frame that is already a tidy-format data frame of draws, provided it has one row per draw. In other words, it can be applied to data frames that have the same format it returns, and it will return the same data frame back, while checking to ensure the `.chain`, `.iteration`, and `.draw` columns are all integers (converting if possible) and that the `.draw` column is unique. This allows you to pass already-tidy-format data frames into other tidybayes functions, like `spread_draws()` or `gather_draws()`. 
ungather_draws

Value
A data frame (actually, a tibble) with a .chain column, .iteration column, .draw column, and one column for every variable in model.

Author(s)
Matthew Kay

See Also
spread_draws() or gather_draws(), which use this function internally and provides a friendly interface for extracting tidy data frames from model fits.

Examples

library(magrittr)
data(line, package = "coda")

line %>%
tidy_draws()


gather_draws

Description
Inverse operations of spread_draws() and gather_draws(), giving results that look like tidy_draws().

Usage

ungather_draws(
data,
...,
variable = ".variable",
value = ".value",
draw_indices = c(".chain", ".iteration", ".draw"),
drop_indices = FALSE
)

unspread_draws(
data,
...,
draw_indices = c(".chain", ".iteration", ".draw"),
drop_indices = FALSE
)
Arguments

data A tidy data frame of draws, such as one output by `spread_draws` or `gather_draws`.

... Expressions in the form of `variable_name[dimension_1, dimension_2, ...]`. See `spread_draws()`.

variable The name of the column in `data` that contains the names of variables from the model.

value The name of the column in `data` that contains draws from the variables.

draw_indices Character vector of column names in `data` that should be treated as indices of draws. The default is `c(".chain", ".iteration", ".draw")`, which are the same names used for chain, iteration, and draw indices returned by `spread_draws()` or `gather_draws()`.

drop_indices Drop the columns specified by `draw_indices` from the resulting data frame. Default `FALSE`.

Details

These functions take symbolic specifications of variable names and dimensions in the same format as `spread_draws()` and `gather_draws()` and invert the tidy data frame back into a data frame whose column names are variables with dimensions in them.

Value

A data frame.

Author(s)

Matthew Kay

See Also

`spread_draws()`, `gather_draws()`, `tidy_draws()`.

Examples

```r
library(dplyr)

data(RankCorr, package = "tidybayes")

# We can use unspread_draws to allow us to manipulate draws with tidybayes
# and then transform the draws into a form we can use with packages like bayesplot.
# Here we subset `b[i,j]` to just values of `i` in 1:2 and `j == 1`, then plot with bayesplot
RankCorr %>%
  spread_draws(b[i,j]) %>%
  filter(i %in% 1:2, j == 1) %>%
  unspread_draws(b[i,j], drop_indices = TRUE) %>%
  bayesplot::mcmc_areas()

# As another example, we could use compare_levels to plot all pairwise comparisons
```
x_at_y

Generate lookup vectors for composing nested indices

Description

Generates a lookup vector such that x_at_y(x, y)[y] == x. Particularly useful for generating lookup tables for nested indices in conjunction with compose_data().

Usage

x_at_y(x, y, missing = NA)

Arguments

x Values in the resulting lookup vector. There should be only one unique value of x for every corresponding value of y.

y Keys in the resulting lookup vector. Should be factors or integers.

missing Missing levels from y will be filled in with this value in the resulting lookup vector. Default NA.

Details

x_at_y(x, y) returns a vector k such that k[y] == x. It also fills in missing values in y: if y is an integer, k will contain entries for all values from 1 to max(y); if y is a factor, k will contain entries for all values from 1 to nlevels(y). Missing values are replaced with missing (default NA).

Author(s)

Matthew Kay

See Also

compose_data().
Examples

library(magrittr)

df = data.frame(
  plot = factor(paste0("p", rep(1:8, times = 2))),
  site = factor(paste0("s", rep(1:4, each = 2, times = 2)))
)

# turns site into a nested index: site[p] gives the site for plot p
df %>%
  compose_data(site = x_at_y(site, plot))
Index

*Topic manip

add_draws, 4
add_fitted_draws, 5
combine_chains, 12
compare_levels, 13
compose_data, 16
data_list, 18
density_bins, 20
gather_draws, 21
gather_emmeans_draws, 26
gather_pairs, 28
gather_variables, 30
get_variables, 55
predict_curve, 67
recover_types, 69
sample_draws, 71
tidy-format-translators, 103
tidy_draws, 107
ungather_draws, 109

add_draws, 4
add_draws(), 5, 11, 107
add_fitted_draws, 5
add_fitted_draws(), 4, 5, 14, 105, 107
add_fitted_samples
  (tidybayes-deprecated), 104
add_lmlpred_draws (add_fitted_draws), 5
add_predicted_draws (add_fitted_draws),
  5
add_predicted_draws(), 4, 5, 21, 105, 107
add_predicted_samples
  (tidybayes-deprecated), 104
add_residual_draws (add_fitted_draws), 5
aes(), 34, 40, 44, 46, 51, 79, 85, 89, 92, 97
aes_(), 34, 40, 44, 46, 51, 79, 85, 89, 92, 97
apply_prototypes (recover_types), 69
as.numeric(), 16, 19
as_data_list (data_list), 18
as_data_list(), 16, 17, 19

as_sample_data_frame
  (tidybayes-deprecated), 104
as_sample_tibble
  (tidybayes-deprecated), 104
axis_titles_bottom_left
  (theme_tidybayes), 102
borders(), 36, 42, 45, 48, 53, 80, 86, 90, 93, 99
brms::brm(), 10, 106, 107
brms::brmsfit, 10
brms::fitted.brmsfit(), 10
brms::get_prior(), 60, 81
brms::predict.brmsfit(), 10
coda::as.mcmc.list(), 107, 108
coda::mcmc(), 106
coda::mcmc.list(), 106
combine_chains, 12
compare_levels, 13
compose_data, 16
compose_data(), 18, 19, 25, 59, 60, 70, 111
coord_cartesian(), 75
data.frame, 67
data_list, 18
density(), 20, 98
density_bins, 20
density_bins(), 67, 68
dlkjcorr_marginal (lkjcorr_marginal), 56
dnorm(), 81
dplyr::do(), 68
dplyr::filter(), 58
dplyr::group_by(), 65
dmeans::emm_list(), 27
dmeans::emmeans(), 13, 26, 27, 29
dmeans::ref_grid(), 26, 27
extract_samples (tidybayes-deprecated),
  104
gather_emmeans_samples
gather_emmeans_draws()
gather_lsmeans_samples
gather_emmeans_draws
gather_draws
from_broom_names
   (tidy-format-translators), 103
gather_emmeans_draws, 26
gather_emmeans_draws(), 105
gather_emmeans_samples
   (tidybayes-deprecated), 104
gather_lsmeans_samples
   (tidybayes-deprecated), 104
gather_pairs, 28
gather_samples (tidybayes-deprecated), 104
gather_terms (tidybayes-deprecated), 104
gather_variables, 30
gather_variables(), 107
gem_dotplot(), 36
gem_dots (geom_dotsinterval), 32
gem_dotsh (geom_dotsinterval), 32
gem_dotsinterval, 32
gem_dotsinterval(), 37, 76
gem_dotsintervalh (geom_dotsinterval), 32
gem_eye, 39
gem_eyeh (geom_eye), 39
gem_eyeh(), 105
gem_halfeye (geom_eye), 39
gem_interval, 40
gem_interval(), 43, 50, 53, 87, 88
gem_intervalh (geom_interval), 40
gem_intervalh(), 43, 50, 53, 88
gem_line(), 44, 45
gem_linerange(), 42
gem_lineribbon, 44
gem_lineribbon(), 55, 88, 90
gem_pointinterval, 46
gem_pointinterval(), 45, 53, 80, 85, 89, 92–94, 97
gem_pointintervalh
   (geom_pointinterval), 46
gem_pointintervalh(), 45, 53, 94
gem_pointrange(), 48
gem_slab(), 44, 45
gem_slabh (geom_slabinterval), 50
gem_slabh (geom_slabinterval), 50
gem_slabinterval, 35, 41, 47, 50
gem_slabinterval(), 32, 36, 38, 40, 42, 43, 46, 48, 50, 72, 75, 77, 79, 81–85, 88, 91, 92, 94, 95, 97, 99–101
grouped_df, 67
guide_colorbar2 (scales), 72
guide_colourbar2 (scales), 72
hdci (point_interval), 62
hdci (point_interval), 62
hdci(), 65
HDInterval::hdci(), 66
hist(), 20, 98
histogram_bins (density_bins), 20
histogram_bins(), 67
jagsUI::jags(), 106
layer(), 45, 52, 80, 85, 89, 92, 97
linpred_draws (add_fitted_draws), 5
list(), 18, 19
lkjcorr_marginal, 56
lkjcorr_marginal(), 58
make.names(), 61
marginalize_lkjcorr, 58
marginalize_lkjcorr(), 57
MCMCglmm::MCMCglmm(), 27, 106
mean(), 64, 67, 68
mean_hdi (point_interval), 62
mean_hdicl (point_interval), 62
scale_point_colour_continuous (scales), 72
scale_point_colour_discrete (scales), 72
scale_point_fill_continuous (scales), 72
scale_point_fill_discrete (scales), 72
scale_point_size_continuous (scales), 72
scale_point_size_continuous (), 35, 41, 47, 52
scale_point_size_discrete (scales), 72
scale_point_size_discrete (), 35, 41, 48, 53
scale_size_continuous (), 35, 42, 47, 52, 53
scale_slab_alpha_continuous (scales), 72
scale_slab_alpha_discrete (scales), 72
scale_slab_color_continuous (scales), 72
scale_slab_color_discrete (scales), 72
scale_slab_colour_continuous (scales), 72
scale_slab_colour_discrete (scales), 72
scale_slab_fill_continuous (scales), 72
scale_slab_fill_discrete (scales), 72
scale_slab_linetype_continuous (scales), 72
scale_slab_linetype_discrete (scales), 72
scale_slab_shape_continuous (scales), 72
scale_slab_shape_discrete (scales), 72
scale_slab_size_continuous (scales), 72
scale_slab_size_discrete (scales), 72
scale_slab_size_continuous (), 35, 43, 49, 54, 72, 83, 87, 94, 101
spread_draws (gather_draws), 21
spread_draws (), 9, 11, 13–15, 17, 31, 32, 55, 56, 67, 69, 70, 105–110
spread_samples (tidybayes-deprecated), 104
stat_ccdfinterval
  (stat_sample_slabinterval), 95
stat_ccdfintervalh
  (stat_sample_slabinterval), 95
stat_cdfinterval
  (stat_sample_slabinterval), 95
stat_cdfintervalh
  (stat_sample_slabinterval), 95
stat_dist_ccdfinterval
  (stat_dist_slabinterval), 77
stat_dist_ccdfintervalh
  (stat_dist_slabinterval), 77
stat_dist_cdfinterval
  (stat_dist_slabinterval), 77
stat_dist_cdfintervalh
  (stat_dist_slabinterval), 77
stat_dist_colorinterval
  (stat_dist_slabinterval), 77
stat_dist_colorintervalh
  (stat_dist_slabinterval), 77
stat_dist_colorinterval
  (stat_dist_slabinterval), 77
stat_dist_colorintervalh
  (stat_dist_slabinterval), 77
stat_dist_dots (geom_dotsinterval), 32
stat_dist_dotsh (geom_dotsinterval), 32
stat_dist_dotsinterval
  (geom_dotsinterval), 32
stat_dist_dotsintervalh
  (geom_dotsinterval), 32
stat_dist_dotsinterval
  (stat_dist_slabinterval), 77
stat_dist_eye (stat_dist_slabinterval), 77
stat_dist_eye
  (stat_dist_slabinterval), 77
stat_dist_gradientinterval
  (stat_dist_slabinterval), 77
stat_dist_gradientintervalh
  (stat_dist_slabinterval), 77
stat_dist_halfeye
  (stat_dist_slabinterval), 77
stat_dist_halfeye
  (stat_dist_slabinterval), 77
stat_dist_interval
  (stat_dist_slabinterval), 77
stat_dist_intervalh
  (stat_dist_slabinterval), 77
stat_dist_lineribbon (stat_lineribbon), 88
stat_dist_lineribbon
  (stat_dist_slabinterval), 77
stat_dist_sample_slabinterval
  (stat_dist_slabinterval), 77
stat_dist_sample_slabinterval
  (stat_dist_slabinterval), 77
stat_dist_slab
  (stat_dist_slabinterval), 77
stat_dist_slabh
  (stat_dist_slabinterval), 77
stat_dist_slabinterval
  (stat_dist_slabinterval), 77
stat_dist_slabinterval
  (stat_dist_slabinterval), 77
stat_dist_slabinterval (), 35, 53, 55, 57, 58, 60, 61, 101
stat_dots (geom_dotsinterval), 32
stat_dotsh (geom_dotsinterval), 32
stat_dotsinterval (geom_dotsinterval), 32
stat_dotsintervalh (geom_dotsinterval), 32
stat_eye (stat_sample_slabinterval), 95
stat_eye()
  (stat_sample_slabinterval), 95
stat_eye
  (stat_sample_slabinterval), 95
stat_eye
  (stat_sample_slabinterval), 95
INDEX

stat_eyeh(), 39

stat_gradientinterval
  (stat_sample_slabinterval), 95

stat_gradientinterval(), .76

stat_gradientintervalh
  (stat_sample_slabinterval), 95

stat_halfeye
  (stat_sample_slabinterval), 95

stat_halfeyeh
  (stat_sample_slabinterval), 95

stat_halfeyeh(), 39, 65

stat_histinterval
  (stat_sample_slabinterval), 95

stat_histintervalh
  (stat_sample_slabinterval), 95

stat_interval, 84

stat_interval(), 43, 54, 94

stat_intervalh(stat_interval), 84

stat_intervalh(), 43, 54, 94

stat_lineribbon, 88

stat_lineribbon(), 21, 45

stat_pointinterval, 91

stat_pointinterval(), 50, 53, 88, 90

stat_pointintervalh
  (stat_pointinterval), 91

stat_pointintervalh(), 50, 53, 65, 88, 90

stat_sample_slabinterval, 95

stat_sample_slabinterval(), 38, 39, 43, 50, 53, 55, 83, 84, 88, 91, 94

stat_slab(stat_sample_slabinterval), 95

stat_slabh(stat_sample_slabinterval), 95

stat_slabinterval(), 32, 88

stat_summary(), 65

tbl_df, 67

theme_tidybayes, 102

tibble, 4, 11, 109

tidy-format-translators, 103

 tidy_draws, 107

 tidy_draws(), 31, 32, 55, 105–107, 109, 110

 tidy_samples(tidybayes-deprecated), 104

 tidybayes (tidybayes-package), 3

 tidybayes-deprecated, 104

 tidybayes-models, 3, 9, 22, 55, 69, 106

 tidybayes-package, 3

to_broom_names
  (tidy-format-translators), 103

to_broom_names(), 22, 106

to_ggmcmc_names
  (tidy-format-translators), 103

to_ggmcmc_names(), 22

ungather_draws, 109

ungather_draws(), 105

ungather_samples
  (tidybayes-deprecated), 104

unspread_draws (ungather_draws), 109

unspread_draws(), 105

unspread_samples
  (tidybayes-deprecated), 104

x_at_y, 111

x_at_y(), 17