Package ‘tidybayes’

June 2, 2019

Title Tidy Data and ‘Geoms’ for Bayesian Models
Version 1.1.0
Date 2019-06-02
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. Functions 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, ggplot2 (>= 3.0.0), coda, purrr (>= 0.2.3), rlang (>= 0.3.0), ggstance, grid, forcats, HDInterval, arrayhelpers, tidyselect, tibble, magrittr
Suggests knitr, testthat, vdiffr (>= 0.3.0), svglite, rstan (>= 2.17.0), runjags, rjags, jagsUI, rstanarm (>= 2.18.2), emmeans, broom (>= 0.4.3), dotwhisker, MCMCglmm, bayesplot, modelr, brms (>= 2.9.0), cowplot, ggridges, 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 6.1.1
LazyData true
Encoding UTF-8

NeedsCompilation no

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

Repository CRAN

Date/Publication 2019-06-02 21:10:03 UTC

\textbf{R topics documented:}

\begin{verbatim}
add_draws .................................................. 3
add_fitted_draws ........................................ 4
combine_chains .......................................... 8
compare_levels .......................................... 10
compose_data ............................................ 12
data_list ................................................ 14
density_bins ............................................ 16
gather_draws ............................................. 18
gather_emmeans_draws .................................. 23
gather_pairs ............................................. 25
gather_variables ........................................ 27
geom_eye .................................................. 29
geom_halfeyeh .......................................... 31
geom_interval ......................................... 33
geom_lineribbon ...................................... 35
gem_pointinterval ....................................... 36
get_variables ........................................... 38
n_prefix ............................................... 39
point_interval ......................................... 40
predict_curve .......................................... 44
recover_types .......................................... 46
sample_draws ........................................... 49
stat_interval ........................................... 50
stat_lineribbon ........................................ 52
stat_pointinterval ..................................... 53
theme_tidybayes ........................................ 55
tidy-format-translators ................................ 56
tidybayes-deprecated .................................... 57
tidybayes-models ....................................... 59
tidy_draws ............................................... 60
ungather_draws ......................................... 62
x_at_y .................................................... 63
\end{verbatim}

Index 65
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
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 data frame of draws.

add_fitted_draws(df, m) is roughly equivalent to add_draws(df, posterior_linpred(m)), except that add_fitted_draws standardizes argument names and values across packages.

add_predicted_draws(df, m) is roughly equivalent to add_draws(df, posterior_predict(m)), 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.

Author(s)
Matthew Kay

See Also
add_fitted_draws, add_predicted_draws
add_fitted_draws

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

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

residual_draws(model, newdata, residual = ".residual", ..., n = NULL,
  seed = NULL, re_formula = NULL, category = ".category")
```
## add_fitted_draws

```
## Default S3 method:
residual_draws <- function(model, newdata, ...) {
  residual_draws <- function(model, newdata, residual = ".residual",
    ..., n = NULL, seed = NULL, re_formula = NULL,
    category = ".category")

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 predic-
tions, 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.

**seed**
A seed to use when subsampling draws (i.e. when n is not NULL).

**re_formula**
formula containing group-level effects to be considered in the prediction. If
NULL (default), include all group-level effects; if NA, include no group-level ef-
facts. Some model types (such as brm and stanreg-objects) allow marginal-
izing over grouping factors by specifying new levels of a factor in newdata. In
the case of brm, you must also pass allow_new_levels = TRUE here to include
new levels (see predict.brmsfit).

**category**
For some ordinal, multinomial, and multivariate models (notably, brm models
but not stan_polr models), multiple sets of rows will be returned per input
row for fitted_draws or 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 differ-
ent response variables. The category argument specifies the name of the col-
umn to put the category names (or variable names) into in the resulting data
frame. The default name of this column (".category") reflects the fact that
this functionality was originally used only for ordinal models and has been
re-used for multivariate models. The fact that multiple rows per response are
returned only for some model types reflects the fact that tidybayes takes the
approach of tidying whatever output is given to us, and the output from differ-
ent modeling functions differs on this point. See vignette("tidy-brms")
and vignette("tidy-rstanarm") for examples of dealing with output from
ordinal models using both approaches.
add_fitted_draws

- **dpar**: For fitted_draws and 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 brm). If 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. c(sigma.hat = "sigma") would output the "sigma" parameter from a model as a column named "sigma.hat"). If 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 predicted_draws; default ".prediction".

- **residual**: The name of the output column for residual_draws; default ".residual".

**Details**

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

add_predicted_draws adds draws from posterior predictions to the data. It corresponds to posterior_predict in rstanarm or predict.brmsfit in 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 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 = 10)) %>%
    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) +
    geom_point(data = mtcars)

  # plot posterior predictive intervals
  mtcars %>%
    group_by(cyl) %>%
    data_grid(hp = seq_range(hp, n = 10)) %>%
    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")
}
```

**combine_chains**

Combine the chain and iteration columns of tidy data frames of draws
**Description**

Combines the chain and iteration columns of a tidy data frame of draws from a Bayesian model fit into a new column that can uniquely identify each draw. Generally speaking **not needed for pure tidybayes code**, as tidybayes functions now automatically include a `.draw` column, but can be useful when interacting with packages that do not provide such a column.

**Usage**

```
combine_chains(data, chain = .chain, iteration = .iteration, into = ".draw")
```

**Arguments**

- `data`: Tidy data frame of draws with columns representing the chain and iteration of each draw.
- `chain`: Bare name of column in `data` indicating the chain of each row. The default `(.chain)` is the same as used by other functions in tidybayes.
- `iteration`: Bare name of column in `data` indicating the iteration of each row. The default `(.iteration)` is the same as used by other functions in tidybayes.
- `into`: Name (as a character vector) of the column to combine chains into. The default, `NULL`, replaces the chain column with `NA`s and writes the combined chain iteration numbers into `iteration`. If provided, `chain` and `iteration` will not be modified, and the combined iteration number will be written into a new column named `into`.

**Value**

A data frame of tidy draws with a combined iteration column

**Author(s)**

Matthew Kay

**See Also**

emmeans

**Examples**

```
library(magrittr)
library(coda)

data(line, package = "coda")

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

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

## 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 (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.

## 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 = `=`,
  draw_indices = c(".chain", ".iteration", ".draw"),
  ignore_groups = ".row")
```
**compare_levels**

**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` for 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.
- **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)) +
  geom_halfeye() +
  facet_grid(cols = vars(i))

# 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)) +
  geom_halfeye() +
  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**

```r
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.
- 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

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_list

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

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.

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`.

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.
```

density_bins  

Description
Generates a data frame of bins representing the kernel density (or histogram) of a vector, suitable for use in generating predictive distributions using `predict_curve`.

Usage

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

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 `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

Author(s)

Matthew Kay

See Also

See `add_predicted_draws` and `stat_lineribbon` for a better approach. This function 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())
```
gather_draws

```r
gather_draws(model = stan_glm(mpg ~ hp * cyl, data = mtcars),
regex = FALSE, sep = "[", "]")

spread_draws(model = stan_glm(mpg ~ hp * cyl, data = mtcars),
regex = FALSE, sep = "[", "]")
```

### Description

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

- `gather_draws(model, ..., regex = FALSE, sep = "[", "]")`
- `spread_draws(model, ..., regex = FALSE, sep = "[", "]")`

### Arguments

- **model**: A supported Bayesian model fit. Tidybayes supports a variety of model objects; for a full list of supported models, see [tidybayes-models](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.
Details

Imagine a JAGS or Stan fit named \texttt{fit}. The model may contain a variable named \texttt{b[i,v]} (in the JAGS or Stan language) with dimension \texttt{i} in 1:100 and dimension \texttt{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.

\texttt{spread_draws} and \texttt{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.

\texttt{spread_draws} and \texttt{gather_draws} return data frames already grouped by all dimensions used on the variables you specify.

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

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

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

\texttt{gather_draws(fit, a[i], b[i,v])} on the same fit would return a grouped data frame (grouped by \texttt{i} and \texttt{v}), with:

- column \texttt{.chain}: the chain number
- column \texttt{.iteration}: the iteration number
- column \texttt{.draw}: the draw number
- column \texttt{i}: value in 1:5
- column \texttt{v}: value in 1:10, or NA if \texttt{.variable} is \texttt{a}.
- column \texttt{.variable}: value in \texttt{c(a, b)}.
- column \texttt{.value}: value of \texttt{a[i]} (when \texttt{.variable} is \texttt{a}) or \texttt{b[i,v]} (when \texttt{.variable} is \texttt{b}) for draw \texttt{.draw}

\texttt{spread_draws} and \texttt{gather_draws} can use type information applied to the \texttt{fit} object by \texttt{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 \texttt{v} dimension in the original data frame data was a factor with levels \texttt{c(a, b, c)}, then we could use \texttt{recover_types} before \texttt{spread_draws}:
**gather_draws**

```r
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:

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

Which is roughly equivalent to this:

```r
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:

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

Is roughly equivalent to this:

```r
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:

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

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

# equivalent

Each of which is roughly equivalent to this:

```r
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:

`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.:

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

Description

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

Usage

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 ref_grid or emmeans.
value The name of the output column to use to contain the values of draws. Defaults to ".value".
... Additional arguments passed to the underlying method for the type of object given.
grid If object is an 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 provides a convenient syntax for generating draws from "estimated marginal means" from a model, and can be applied to various Bayesian models, like stanreg-objects and MCMCglmm. Given a ref_grid object as returned by functions like ref_grid or 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.

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 `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`

**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 =
    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()

```

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

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

Examples

library(ggplot2)
library(dplyr)

t_a = rnorm(100)
t_b = rnorm(100, t_a * 2)
t_c = rnorm(100)

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(x = t_x, y = t_y)) +
    geom_point() +
    facet_grid(vars(g_row), vars(g_col))

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

df |>
  gather_pairs(g, t, triangle = "both") |>
  ggplot(aes(x = .x, y = .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 = .x, y = .y)) +
    geom_point(alpha = .25) +
    facet_grid(vars(.row), vars(.col))

line |>
groom_variables

```r
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))
```

---

**groom_variables**

Groom 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`, groom variables and their values from that data frame into a `.variable` and `.value` column.

**Usage**

```r
groom_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"

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

- 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 ".variable" (when ".variable" is "a"; repeated for every value of ".v") or ".variable" (when ".variable" is "b") for draw number ".draw"

In this example, this call:

gather_variables(data)

Is roughly equivalent to:

```r
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

```r
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()
```
**Description**

Generates a combination of `geom_violin` and `stat_pointinterval` (for `geom_eye`) or `geom_violinh` and `stat_pointintervalh` (for `geom_eyeh`) representing the density, point summary, and uncertainty intervals for draws from a distribution. Useful for representing posteriors from Bayesian models; in that context this is variously called an eye plot, a raindrop plot, or a violin plot (though violin plot is also applied to plots of data, hence its use is not preferred here).

**Usage**

```r
geom_eye(mapping = NULL, data = NULL, position = "identity", trim = TRUE, scale = "area", relative_scale = 1, fill = NULL, violin.color = NA, ..., point_interval = median_qi, fun.data = NULL, fun.args = list(), .width = c(0.66, 0.95), .prob, color = NULL, size = NULL, size_domain = NULL, size_range = NULL, fatten_point = NULL)
```

```r
geom_eyeh(mapping = NULL, data = NULL, position = "identity", trim = TRUE, scale = "area", relative_scale = 1, fill = NULL, violin.color = NA, ..., point_interval = median_qi, fun.data = NULL, fun.args = list(), .width = c(0.66, 0.95), .prob, color = NULL, size = NULL, size_domain = NULL, size_range = NULL, fatten_point = NULL)
```

**Arguments**

- **mapping**
  The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.

- **data**
  A layer specific dataset - only needed if you want to override the plot defaults.

- **position**
  Passed to `geom_violin` / `geom_violinh`. The position adjustment to use for overlapping points on this layer.

- **trim**
  Passed to `geom_violin` / `geom_violinh`. If `TRUE` (default), trim the tails of the violins to the range of the data. If `FALSE`, don’t trim the tails.

- **scale**
  Passed to `geom_violin` / `geom_violinh`. If "area" (default), all violins have the same area (before trimming the tails). If "count", areas are scaled proportionally to the number of observations. If "width", all violins have the same maximum width.

- **relative_scale**
  A relative scaling factor to determine how much of the available space densities are scaled to fill: if 1, all available space is filled.

- **fill**
  Passed to `geom_violin` / `geom_violinh`. Fill color of the violin.
violin.color  Passed as the color argument of `geom_violin`/`geom_violinh`. The default, `NA`, suppresses the violin outline. Set to another value to set the violin outline color manually, or set to NULL if you want the outline color of the violin to be determined by the aesthetic mapping.

...  Currently unused.

point_interval  A function that when given a vector should return a data frame with variables `y`, `ymin`, `ymax`, and `.width`; or `x`, `xmin`, `xmax`, and `.width`. **Either is acceptable:** output will be converted into the y-based aesthetics for `geom_eye` and the x-based aesthetics for `geom_eyeh`. See the `point_interval` family of functions.

fun.data  Similar to `point_interval`, for compatibility with `stat_summary`. Note: if the summary function is passed using `fun.data`, the x and y-based aesthetics are not converted to the correct form automatically.

fun.args  Optional arguments passed to `fun.data`.

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

.prob  Deprecated. Use `.width` instead.

color  Passed to `stat_pointinterval`. Color of the point summary and uncertainty interval.

size  Passed to `stat_pointinterval`. Line weight of the point summary and uncertainty interval.

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

size_range  This geom scales the raw size aesthetic values, 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 `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 `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 line.

Details

An eye plot is a compact visual summary of the distribution of a sample, used (under various names and with subtle variations) to visualize posterior distributions in Bayesian inference. This instantiation is a combination of a violin plot, point summary, and one or more uncertainty intervals.

The vertical form, `geom_eye`, is equivalent to `geom_violin() + stat_pointinterval()` with some reasonable defaults, including color choices and the use of median with 95% and 66% quantile intervals.

The horizontal form, `geom_eyeh()`, is equivalent to `geom_violinh() + stat_pointintervalh()`.

Author(s)

Matthew Kay
See Also

See `geom_halfeyeh` for the non-mirrored density ("half eye") version. See `geom_violin` and `stat_pointinterval` for the geoms these functions are based on.

Examples

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

data(RankCorr, package = "tidybayes")

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(y = 1, x = u_tau)) +
  geom_eye()

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(x = 1, y = u_tau)) +
  geom_eye()
```

---

**geom_halfeyeh**  
*Half-eye plots of densities with point and interval summaries (ggplot geom)*

**Description**

Generates a combination of a density and `stat_pointintervalh` representing the density, point summary, and uncertainty intervals for draws from a distribution. Useful for representing posteriors from Bayesian models; in that context the mirrored version is variously called an eye plot, a raindrop plot, violin plot; hence "half-eye" for this plot.

**Usage**

```r
geom_halfeyeh(mapping = NULL, data = NULL, position = "identity",
  trim = TRUE, scale = "area", relative_scale = 1, fill = NULL,
  density.color = NA, ..., point_interval = median_qi,
  fun.data = NULL, fun.args = list(), .width = c(0.66, 0.95), .prob,
  color = NULL, size = NULL, size_domain = NULL, size_range = NULL,
  fatten_point = NULL)
```

**Arguments**

- `mapping`  
The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.
- `data`  
A layer specific dataset - only needed if you want to override the plot defaults.
position
trim
scale
relative_scale
fill
density.color
point_interval
fun.data
fun.args
.width
.probab
.color
.size
.size_domain
.size_range
fatten_point

details

A half-eye plot is a compact visual summary of the distribution of a sample, used (under various names and with subtle variations) to visualize posterior distributions in Bayesian inference. This
instantiation is a combination of a density plot, point summary, and one or more uncertainty intervals.

gem_halfeyeh() is roughly equivalent to geom_density_ridges() + stat_pointintervalh() with some reasonable defaults, including color choices and the use of median with 95% and 66% quantile intervals.

Author(s)
Matthew Kay

See Also
See geom_eye and geom_eyeh for the mirrored-density (full "eye") versions. See geom_density_ridges and stat_summaryh for the geoms this function is based on.

Examples

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

data(RankCorr, package = "tidybayes")

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

---

**geom_interval**

*Multiple probability interval plots (ggplot geom)*

**Description**

Modified versions of geom_linerange and geom_linerangeh with default aesthetics designed for use with output from point_interval.

**Usage**

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

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

Arguments

mapping
The aesthetic mapping, usually constructed with aes or aes_string. Only needs to be set at the layer level if you are overriding the plot defaults.

data
A layer specific dataset - only needed if you want to override the plot defaults.

stat
The statistical transformation to use on the data for this layer.

position
The position adjustment to use for overlapping points on this layer.

... 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
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 behavior from the default plot specification, e.g. borders.

Details

geom_interval is a modified version of geom_linerange, and geom_intervalh is a modified version of geom_linerangeh. 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 or geom_linerangeh 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))".

Author(s)

Matthew Kay

See Also

See geom_lineribbon for a similar geom designed for curves plus probability bands. See geom_linerange and geom_linerangeh for the geoms these are based on.

Examples

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 = 1, x = u_tau)) +
  geom_intervalh() +
geom_lineribbon

```
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()
```

### Description

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

### Usage

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

### Arguments

- **mapping**
  The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.
- **data**
  A layer specific dataset - only needed if you want to override the plot defaults.
- **stat**
  The statistical transformation to use on the data for this layer.
- **position**
  The position adjustment to use for overlapping points on this layer.
- **...**
  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**
  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 behavior from the default plot specification, e.g. borders.

### Details

`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, size = -.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**

```r
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()
```

---

**geom_pointinterval**  
*Point + multiple probability interval plots (ggplot geom)*

**Description**

Modified versions of `geom_pointrange` and `geom_pointrangeh` with default aesthetics designed for use with output from `point_interval`.

**Usage**

```r
geom_pointinterval(mapping = NULL, data = NULL, stat = "identity",
  position = "identity", ..., size_domain = c(1, 6),
  size_range = c(0.6, 1.4), fatten_point = 1.8, na.rm = FALSE,
  show.legend = c(size = FALSE), inherit.aes = TRUE)

geom_pointintervalh(mapping = NULL, data = NULL, stat = "identity",
  position = "identity", ..., size_domain = c(1, 6),
  size_range = c(0.6, 1.4), fatten_point = 1.8, na.rm = FALSE,
  show.legend = c(size = FALSE), inherit.aes = TRUE)
```
Arguments

- **mapping**: The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.
- **data**: A layer specific dataset - only needed if you want to override the plot defaults.
- **stat**: The statistical transformation to use on the data for this layer.
- **position**: The position adjustment to use for overlapping points on this layer.
- **size_domain**: The minimum and maximum of the values of the size aesthetic that will be translated into actual sizes drawn according to `size_range` (see the documentation for that argument, below.)
- **size_range**: This geom scales the raw size aesthetic values, 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 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 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 line.
- **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 behavior from the default plot specification, e.g. borders.

Details

gem_pointinterval is a modified version of `geom_pointrange`, and `geom_pointintervalh` is a modified version of `geom_pointrangeh`. 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` or `geom_pointrangeh` 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)`.

Both geoms provides a scaling factor for line width as well as point size through the `fatten_interval` and `fatten_point` arguments; this scaling factor is designed to give multiple probability intervals reasonable scaling at the default settings for `scale_size_continuous`. Finally, these geoms default to not displaying the legend, though this can be overridden through setting `show.legend = NA` (the setting for most geoms) or `show.legend = TRUE`. 
Author(s)

Matthew Kay

See Also

See `geom_lineribbon` for a similar geom designed for curves plus probability bands. See `geom_pointrange` and `geom_pointrangeh` for the geoms these are based on.

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_pointinterval()

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

---

`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:
get_variables(model)

## S3 method for class 'mcmc'
get_variables(model)

## S3 method for class 'mcmc.list'
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.

Value

A character vector of variable names in the fitted model.

Author(s)

Matthew Kay

See Also

spread_draws, gather_draws.

Examples

data(line, package = "coda")
get_variables(line)

data(RankCorr, package = "tidybayes")
get_variables(RankCorr)

---

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 prefixing 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 create index columns. Typically something like "n" (that is the default used in the `.n_name` argument of `compose_data`).

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

```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"))
```

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

```r
point_interval(.data, ..., .width = 0.95, .point = median, 
.interval = qi, .simple_names = TRUE, na.rm = FALSE, 
.exclude = c("chain", "iteration", "draw", "row"), .prob)
```

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

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

```r
point_intervalh(...)
```

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

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

```r
Mode(x, na.rm = FALSE)
```

```r
hdcix, .width = 0.95, na.rm = FALSE)
```

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

```r
mean_qih(...)
```

```r
median_qi(.data, ..., .width = 0.95)
```

```r
median_qih(...)
```

```r
mode_qi(.data, ..., .width = 0.95)
```

```r
mode_qih(...)
```

```r
mean_hdi(.data, ..., .width = 0.95)
```

```r
mean_hdih(...)
```

```r
median_hdi(.data, ..., .width = 0.95)
```

```r
median_hdih(...)
```

```r
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`.

.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 summarised 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.
point_interval

.probab Variance. Use .width instead.

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

Details

If .data is a data frame, then . data 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 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. group_by), the points and intervals are calculated within the groups.

If .data is a vector, .data 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 using methods like geom_eye, geom_eyeh, 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, geom_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.

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 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 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)

rnorm(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))
)

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

multimodal_draws %>%
  ggplot(aes(x = x, y = 0)) +
  geom_halfeye(fun.data = 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).
predict_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 link{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 data_grid combined with point_interval or 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**

```r
# 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**

```r
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 \( \text{NNN} \) specifies a list or data.frame. The \text{model} is decorated with a list of constructors that can convert a numeric column into the data types in the lists in \( \text{NNN} \).

Then, when \text{spread_draws} or \text{gather_draws} is called on the decorated \text{model}, each list entry with the same name as the variable or a dimension in \text{variable_spec} is 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 \text{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 \text{prototypes}\( v \) is a factor, the \( v \) column in the returned draws is translated into a factor using 
  \[
  \text{factor}(v, \text{labels}=\text{levels(prototypes}\( v \)), \text{ordered}=\text{is.ordered(prototypes}\( v \))).
  \]
- if \text{prototypes}\( v \) is a logical, the \( v \) column is translated into a logical using \text{as.logical}(v).

Additional data types can be supported by providing a custom implementation of the generic function \text{as_constructor}.

Value

A decorated version of \text{model}.

Author(s)

Matthew Kay

See Also

\text{spread_draws}, \text{gather_draws}, \text{compose_data}.

Examples

```r
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);

  for (i in 1:n) {
    response[i] ~ normal(condition_mean[condition[i]], response_sd);
  }
}
```

```r
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:

```r
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)

```r
m %>%
  recover_types(ABC)
```

# now the `condition` column with be a factor with levels "A", "B", "C", ...

```r
m %>%
  spread_draws(condition_mean[condition])) %>%
```
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)
}

---

### stat_interval

**Multiple probability interval plots (ggplot stat)**

#### Description

A combination of `stat_summary` / `stat_summaryh` and `geom_interval` / `geom_intervalh` 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", ..., point_interval = median_qi,
              fun.data = NULL, .width = c(0.5, 0.8, 0.95), .prob,
              fun.args = list(), na.rm = FALSE, show.legend = NA,
              inherit.aes = TRUE)

stat_intervalh(mapping = NULL, data = NULL, geom = "intervalh",
              position = "identity", ..., point_interval = median_qi,
              fun.data = NULL, .width = c(0.5, 0.8, 0.95), .prob,
              fun.args = list(), na.rm = FALSE, show.legend = NA,
              inherit.aes = TRUE)
```

#### Arguments

- **mapping**
  The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.

- **data**
  A layer specific dataset - only needed if you want to override the plot defaults.
stat_interval

geom
Use to override the default connection between geom_interval/geom_interval and stat_interval/stat_intervalh.

position
The position adjustment to use for overlapping points on this layer.

...
Other arguments passed to layer. They may also be arguments to the paired geom.

point_interval
A function that when given a vector should return a data frame with variables y, ymin, ymax, and .width; or x, xmin, xmax, and .width. Either is acceptable: output will be converted into the y-based aesthetics for stat_interval and the x-based aesthetics for stat_intervalh. See the point_interval family of functions.

fun.data
Similar to point_interval, for compatibility with stat_summary. Note: if the summary function is passed using fun.data, the x and y-based aesthetics are not converted to the correct form automatically.

.width
The .width argument passed to point_interval.

.prob
Deprecated. Use .width instead.

fun.args
Other optional arguments passed to fun.data.

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 behavior from the default plot specification, e.g. borders.

See Also

See geom_interval / geom_intervalh for the geom versions, intended for use on 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(magrittr)
library(ggplot2)

data(RankCorr, package = "tidybayes")

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(y = 1, x = u_tau)) +
  stat_intervalh() +
  scale_color_brewer()

RankCorr %>%
  spread_draws(u_tau[i]) %>%
  ggplot(aes(x = i, y = u_tau)) +
  ...
**Description**

A combination of `stat_summary` 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.

**Usage**

```r
stat_lineribbon(mapping = NULL, data = NULL, geom = "lineribbon",
position = "identity", ..., point_interval = median_qi,
fun.data = NULL, .width = c(0.5, 0.8, 0.95), .prob,
fun.args = list(), na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE)
```

**Arguments**

- `mapping` The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.
- `data` A layer specific dataset - only needed if you want to override the plot defaults.
- `geom` Use to override the default connection between `geom_lineribbon` and `stat_lineribbon`.
- `position` The position adjustment to use for overlapping points on this layer.
- `...` Other arguments passed to `layer`. They may also be arguments to the paired `geom`.
- `point_interval` A function that when given a vector should return a data frame with variables `y`, `ymin`, `ymax`, and `.width`; or `x`, `xmin`, `xmax`, and `.width`. Either is acceptable: output will be converted into the y-based aesthetics. See the `point_interval` family of functions.
- `fun.data` Similar to `point_interval`, for compatibility with `stat_summary`. Note: if the summary function is passed using `fun.data`, x-based aesthetics are not converted to the correct form automatically.
- `.width` The `.width` argument passed to `point_interval`.
- `.prob` Deprecated. Use `.width` instead.
- `fun.args` Other optional arguments passed to `fun.data`.
- `na.rm` If FALSE, the default, missing values are removed with a warning. If TRUE, missing values are silently removed.
**stat_pointinterval**

- **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 behavior from the default plot specification, e.g. borders.

**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**

```r
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()
```

---

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

**Description**

A combination of *stat_summary*/*stat_summaryh* and *geom_pointinterval*/*geom_pointintervalh* 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", ...,
point_interval = median_qi, fun.data = NULL, .width = c(0.66, 0.95), .prob = NULL, fun.args = list(), na.rm = FALSE,
show.legend = c(size = FALSE), inherit.aes = TRUE)
```

```r
stat_pointintervalh(mapping = NULL, data = NULL,
geom = "pointintervalh", position = "identity", ...,
point_interval = median_qi, fun.data = NULL, .width = c(0.66, 0.95), .prob = NULL, fun.args = list(), na.rm = FALSE,
show.legend = c(size = FALSE), inherit.aes = TRUE)
```
Arguments

- **mapping**: The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.
- **data**: A layer specific dataset - only needed if you want to override the plot defaults.
- **geom**: Use to override the default connection between `geom_pointinterval`/`geom_pointintervalh` and `stat_pointinterval`/`stat_pointintervalh`.
- **position**: The position adjustment to use for overlapping points on this layer.
- **...**: Other arguments passed to `layer`. They may also be arguments to the paired geom.
- **point_interval**: A function that when given a vector should return a data frame with variables `y`, `ymin`, `ymax`, and `.width`; or `x`, `xmin`, `xmax`, and `.width`. **Either is acceptable**: output will be converted into the y-based aesthetics for `stat_pointinterval` and the x-based aesthetics for `stat_pointintervalh`. See the `point_interval` family of functions.
- **fun.data**: Similar to `point_interval`, for compatibility with `stat_summary`. Note: if the summary function is passed using `fun.data`, the x and y-based aesthetics are not converted to the correct form automatically.
- **.width**: The `.width` argument passed to `point_interval`.
- **.prob**: Deprecated. Use `.width` instead.
- **fun.args**: Other optional arguments passed to `fun.data`.
- **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 behavior from the default plot specification, e.g. borders.

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.

Examples

```r
library(magrittr)
library(ggplot2)
```
### theme_tidybayes

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 `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

`theme`, `theme_set`
Examples

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

**tidy-format-translators**

*Translate between different tidy data frame formats for draws from distributions*

**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` <> `term`
- `.value` <> `estimate`
- `.prediction` <> `.fitted`
- `.lower` <> `conf.low`
- `.upper` <> `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

library(magrittr)

data(line, package = "coda")

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

deprecated

Description

Deprecated functions, arguments, and column names and their alternatives are listed below. Many of the deprecations are due to a naming scheme overhaul in tidybayes version 1.0.

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`

• `.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 abbreviated 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
• brm models
• rstanarm models
• runjags models
• jags.model models, if sampled using coda.samples
• jags models
• MCMCglmm models
• mcmc and mcmc.list objects, which are output by several model types.
• Any object with an implementation of 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), 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`:

- **brm** models
- **rstanarm** models

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 a `.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**

tidy_draws(model)

## Default S3 method:
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

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 as.mcmc.list (tidy_draws should work on any model with an implementation of as.mcmc.list)

**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**

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

line %>%
```
ugather_draws

tidy_draws()

---

**ugather_draws**

Turn tidy data frames of variables from a Bayesian model back into untidy data

---

**Description**

Inverse operations of `spread_draws` and `gather_draws`, giving results that look like `tidy_draws`.

**Usage**

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

```r
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").
  These 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

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
# of b[i,j] for j in 1:3
RankCorr %>%
  spread_draws(b[i,j]) %>%
  filter(i == 1, j %in% 1:3) %>%
  compare_levels(b, by = j) %>%
  unspread_draws(b[j], drop_indices = TRUE) %>%
  bayesplot::mcmc_areas()

---

**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 \( \text{max}(y) \); if y is a factor, \( k \) will contain entries for all values from 1 to \( \text{nlevels}(y) \). Missing values are replaced with `missing` (default \( \text{NA} \)).

Author(s)

Matthew Kay

See Also

`compose_data`.

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)))
)

# 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, 3
  add_fitted_draws, 4
  combine_chains, 8
  compare_levels, 10
  compose_data, 12
  data_list, 14
  density_bins, 16
  gather_draws, 18
  gather_emmeans_draws, 23
  gather_pairs, 25
  gather_variables, 27
  geom_eye, 29
  geom_halfeyeh, 31
  geom_interval, 33
  geom_lineribbon, 35
  geom_pointinterval, 36
  get_variables, 38
  predict_curve, 44
  recover_types, 46
  sample_draws, 49
  theme_tidybayes, 55
  tidy-format-translators, 56
  tidy_draws, 60
  ungather_draws, 62

add_draws, 3
add_fitted_draws, 4, 11, 58, 60
add_fitted_samples
  (tidybayes-deprecated), 57
add_linpred_draws (add_fitted_draws), 4
add_predicted_draws, 3, 17, 58, 60
add_predicted_draws (add_fitted_draws), 4
add_predicted_samples
  (tidybayes-deprecated), 57
add_residual_draws (add_fitted_draws), 4
aes, 29, 31, 34, 35, 37, 50, 52, 54
aes_string, 29, 31, 34, 35, 37, 50, 52, 54
apply_prototypes (recover_types), 46
as.mcmc_list, 59, 61
as.numeric, 13, 15
as_data_list, 13, 16
as_data_list (data_list), 14
as_sample_data_frame
  (tidybayes-deprecated), 57
as_sample_tibble
  (tidybayes-deprecated), 57
axis_titles_bottom_left
  (theme_tidybayes), 55
brm, 6, 59, 60
coda.samples, 59
combine_chains, 8
compare_levels, 10
compose_data, 12, 14, 16, 22, 39, 40, 47, 63, 64
data.frame, 45
data_grid, 45
data_list, 14
density, 17
density_bins, 16, 45, 46
do, 45
emm_list, 23, 24
emmeans, 9, 23, 24, 26
eval_tidy, 43
extract_samples (tidybayes-deprecated), 57
facet_title_left_horizontal
  (theme_tidybayes), 55
facet_title_right_horizontal
  (theme_tidybayes), 55
fitted.brmsfit, 7
fitted_draws, 58, 60
fitted_draws (add_fitted_draws), 4
fitted_samples (tidybayes-deprecated), 57
INDEX

from_broom_names, 58
from_broom_names (tidy-format-translators), 56
from_ggmcmmc_names (tidy-format-translators), 56
gather_draws, 10, 11, 13, 18, 39, 46, 47, 57, 59–63
gather_emmeans_draws, 23, 58
gather_emmeans_samples (tidybayes-deprecated), 57
gather_lsmeans_samples (tidybayes-deprecated), 57
gather_pairs, 25
gather_samples (tidybayes-deprecated), 57
gather_terms (tidybayes-deprecated), 57
gather_variables, 27, 60
gather_density_ridges, 33
gom_e, 29, 33, 43
gom_e (geom_e), 29
gom_h (geom_h), 31, 31, 43
gom_interval, 33, 30, 31
gom_intervalh, 50, 51
gom_intervalh (geom_interval), 33
geom_line, 35, 36
geom_linrange, 33, 34
geom_linrangeh, 33, 34
geom_lineribbon, 34, 35, 38, 32, 33
geom_pointinterval, 36, 36, 36, 34
geom_pointintervalh, 36, 36, 36, 34
geom_pointintervalh (geom_pointinterval), 36
geom_pointrange, 36–38
geom_pointrangeh, 36–38
geom_ribbon, 35, 36
geom_violin, 29–31
geom_violinh, 29, 30
get_variables, 38
gey (tidybayes-deprecated), 57
group_by, 42, 43, 45

hdi (point_interval), 40
hdci (point_interval), 40
hist, 17
histogram_bins, 45
histogram_bins (density_bins), 16
jags, 59
jags.model, 59

layer, 34, 35, 37, 51, 52, 54
linpred_draws (add_fitted_draws), 4
list, 15
mcmc, 59
mcmc.list, 59
MCMCglmm, 23, 59
mean, 42, 45
mean_hdi (point_interval), 40
mean_hdi (point_interval), 40
mean_hdi (point_interval), 40
mean_qi, 34, 37
mean_qi (point_interval), 40
mean_qi (point_interval), 40
mean_qi (point_interval), 40
Median, 42, 45
median_hdi (point_interval), 40
median_hdi (point_interval), 40
median_hdi (point_interval), 40
median_qi, 34, 37, 60
median_qi (point_interval), 40
median_qi (point_interval), 40
Mode, 42, 45
Mode (point_interval), 40
Mode_hdi (point_interval), 40
Mode_hdi (point_interval), 40
Mode_hdi (point_interval), 40
Mode_qi (point_interval), 40
Mode_qi (point_interval), 40
mutate, 13

n_prefix, 13, 39
parameters (tidybayes-deprecated), 57
point_interval, 23, 33, 35, 36, 40, 43, 50–54
point_intervalh (point_interval), 40
posterior_linpred, 7
posterior_predict, 7
predict.brmsfit, 6, 7
predict_curve, 17, 44
predict_curve_density (predict_curve), 44
predicted_draws, 58, 60
predicted_draws (add_fitted_draws), 4
predicted_samples
  (tidybayes-deprecated), 57
qi, 42
qi (point_interval), 40
recover_types, 19, 21, 22, 46
ref_grid, 23
relevel, 11
residual_draws (add_fitted_draws), 4
rstan, 59
rstanarm, 59, 60
runjags, 59
sample_draws, 49
scale_size_continuous, 30, 32, 37
spread_draws, 6, 8, 10, 11, 13, 27, 28, 39, 45–47, 57, 59–63
spread_draws (gather_draws), 18
spread_samples (tidybayes-deprecated), 57
stan_polr, 6
stat_interval, 50, 54
stat_intervalh, 54
stat_intervalh (stat_interval), 50
stat_lineribbon, 17, 36, 52
stat_pointinterval, 29–31, 51, 53, 53
stat_pointintervalh, 29, 31, 32, 43, 51, 53
stat_pointintervalh
  (stat_pointinterval), 53
stat_summary, 43, 50, 52, 53
stat_summaryh, 33, 50, 53
tbl_df, 45
theme, 55
theme_light, 55
theme_set, 55
theme_tidypayes, 55
tibble, 3, 7, 61
tidy-format-translators, 56
tidy_draws, 27, 28, 38, 58–60, 60, 62, 63
tidy_samples (tidybayes-deprecated), 57
tidybayes-deprecated, 57
tidybayes-models, 6, 18, 39, 46, 59
to_broom_names, 19, 58
to_broom_names
  (tidy-format-translators), 56
to_ggmcnc_names, 19