Package 'tidybayes'

September 15, 2024

Title Tidy Data and 'Geoms' for Bayesian Models

Version 3.0.7

Maintainer Matthew Kay <mjskay@northwestern.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 (>= 4.0.0)

Imports methods, ggdist (>= 3.0.0), dplyr (>= 1.1.0), tidyr (>= 1.0.0), ggplot2 (>= 3.3.5), coda, rlang (>= 0.3.0), arrayhelpers, tidyselect, tibble, magrittr, posterior (>= 1.0.1), withr, cli, vctrs

Suggests knitr, testthat, purrr (>= 0.2.3), forcats, vdiffr (>= 1.0.0), svglite, rstan (>= 2.17.0), rstantools (>= 2.1.0), runjags, rjags, jagsUI, rstanarm (>= 2.19.2), emmeans, broom (>= 0.4.3), MCMCglmm, bayesplot, modelr, brms (>= 2.16.0), cowplot, covr, rmarkdown, ggrepel, bindrcpp, RColorBrewer, gganimate, gifski, png, ragg, pkgdown, distributional, transformr

License GPL (>= 3)

Language en-US

BugReports https://github.com/mjskay/tidybayes/issues

URL https://mjskay.github.io/tidybayes/, https://github.com/mjskay/tidybayes/

VignetteBuilder knitr

RoxygenNote 7.3.2

Encoding UTF-8

Collate ``ggdist-curve_interval.R'' ``ggdist-cut_cdf_qi.R''
``ggdist-geom_slabinterval.R'' ``ggdist-geom_dotsinterval.R''
``ggdist-geom_interval.R'' ``ggdist-geom_lineribbon.R''
``ggdist-geom_pointinterval.R'' ``ggdist-lkjcorr_marginal.R''
``ggdist-parse_dist.R'' ``ggdist-scales.R''
``ggdist-stat_slabinterval.R'' ``ggdist-stat_dist_slabinterval.R''
``ggdist-stat_sample_slabinterval.R''
``ggdist-stat_dotsinterval.R'' ``ggdist-stat_pointinterval.R''
``ggdist-stat_interval.R'' ``ggdist-stat_lineribbon.R''
``ggdist-student_t.R'' ``ggdist-theme_ggdist.R''
``ggdist-tidy_format_translators.R'' ``tidybayes-package.R''
``add_draws.R'' ``combine_chains.R'' ``compare_levels.R''
``compose_data.R'' ``density_bins.R'' ``emmeans_comparison.R''
``epred_draws.R'' ``epred_rvars.R'' ``flip_aes.R'' ``gather_draws.R''
``gather_emmeans_draws.R'' ``gather_pairs.R'' ``gather_rvars.R''
``gather_variables.R'' ``get_variables.R'' ``global_variables.R''
``linpred_draws.R'' ``linpred_rvars.R'' ``nest_rvars.R'' ``onAttach.R''
``point_interval.R'' ``predict_curve.R'' ``predicted_draws.R''
``predicted_rvars.R'' ``recover_types.R'' ``residual_draws.R''
``sample_draws.R'' ``spread_draws.R'' ``spread_rvars.R''
``summarise_draws.R'' ``testthat.R'' ``tidy_draws.R''
``tidybayes-models.R'' ``ungather_draws.R'' ``unspread_draws.R''
``util.R'' ``x_at_y.R'' ``deprecated.R''

NeedsCompilation no

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

Repository CRAN

Date/Publication 2024-09-15 06:20:02 UTC

Contents

`tidybayes-package` *Tidy Data and 'Geoms' for Bayesian Models*

Description

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

Details

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

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

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

Author(s)

Maintainer: Matthew Kay <mjskay@northwestern.edu>

Other contributors:

- Timothy Mastny <tim.mastny@gmail.com> [contributor]

References

Wickham, Hadley. (2014). Tidy data. *Journal of Statistical Software*, 59(10), 1-23. doi:10.18637/jss.v059.i10.

See Also

Useful links:

- https://mjskay.github.io/tidybayes/
- https://github.com/mjskay/tidybayes/
- Report bugs at https://github.com/mjskay/tidybayes/issues

`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_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 dataframe of draws.

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

`add_predicted_draws(df, m)` is roughly equivalent to `add_draws(df, posterior_predict(m, newdata = df))`, 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_predicted_draws()`, `add_draws()`

Examples

```
## Not run:

library(ggplot2)
library(dplyr)
library(brms)
library(modelr)

theme_set(theme_light())

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

# plot posterior predictive intervals
mtcars %>%
  group_by(cyl) %>%
  data_grid(hp = seq_range(hp, n = 101)) %>%
  # the line below is roughly equivalent to add_epred_draws(m_mpg), except
  # that it does not standardize arguments across model types.
  add_draws(posterior_epred(m_mpg, newdata = .)) %>%
  ggplot(aes(x = hp, y = mpg, color = ordered(cyl))) +
  stat_lineribbon(aes(y = .value), alpha = 0.25) +
  geom_point(data = mtcars) +
  scale_fill_brewer(palette = "Greys")


## End(Not run)
```

`add_epred_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 linear/link-level predictor, the expectation of the posterior predictive, the posterior predictive, or the residuals of a model to the data frame in a long format.

Usage

```
add_epred_draws(
  newdata,
  object,
  ...,
  value = ".epred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = NULL
)

epred_draws(
  object,
  newdata,
  ...,
  value = ".epred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = NULL
)

## Default S3 method:
epred_draws(
  object,
  newdata,
  ...,
  value = ".epred",
  seed = NULL,
  category = NULL
)

## S3 method for class 'stanreg'
epred_draws(
  object,
  newdata,
  ...,
  value = ".epred",
  ndraws = NULL,
```

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

## S3 method for class 'brmsfit'
epred_draws(
  object,
  newdata,
  ...,
  value = ".epred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = NULL
)

add_linpred_draws(
  newdata,
  object,
  ...,
  value = ".linpred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = NULL,
  n
)

linpred_draws(
  object,
  newdata,
  ...,
  value = ".linpred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = NULL,
  n,
  scale
)

## Default S3 method:
linpred_draws(
```

```
  object,
  newdata,
  ...,
  value = ".linpred",
  seed = NULL,
  category = NULL
)

## S3 method for class 'stanreg'
linpred_draws(
  object,
  newdata,
  ...,
  value = ".linpred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = NULL
)

## S3 method for class 'brmsfit'
linpred_draws(
  object,
  newdata,
  ...,
  value = ".linpred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  dpar = NULL
)

add_predicted_draws(
  newdata,
  object,
  ...,
  value = ".prediction",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  n
)

predicted_draws(
  object,
```

```
  newdata,
  ...,
  value = ".prediction",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  n,
  prediction
)

## Default S3 method:
predicted_draws(
  object,
  newdata,
  ...,
  value = ".prediction",
  seed = NULL,
  category = ".category"
)

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

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

add_residual_draws(
  newdata,
  object,
```

```
  ...,
  value = ".residual",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  n
)

residual_draws(
  object,
  newdata,
  ...,
  value = ".residual",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category",
  n,
  residual
)

## Default S3 method:
residual_draws(object, newdata, ...)

## S3 method for class 'brmsfit'
residual_draws(
  object,
  newdata,
  ...,
  value = ".residual",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  category = ".category"
)
```

Arguments

newdata	Data frame to generate predictions from.
object	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_epred_draws()`, `add_predicted_draws()`, etc. a model must provide an interface for generating predictions, thus more generic Bayesian modeling interfaces like `runjags` and `rstan` are not directly supported for these functions (only wrappers around those languages that provide predictions, like `rstanarm` and `brm`, are supported here).

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

`value` The name of the output column:

- for `[add_]epred_draws()`, defaults to `".epred"`.
- for `[add_]predicted_draws()`, defaults to `".prediction"`.
- for `[add_]linpred_draws()`, defaults to `".linpred"`.
- for `[add_]residual_draws()`, defaults to `".residual"`

`ndraws` The number of draws to return, or `NULL` to return all draws.

`seed` A seed to use when subsampling draws (i.e. when `ndraws` 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 effects. Some model types (such as brms::brmsfit and rstanarm::stanreg-objects) allow marginalizing over grouping factors by specifying new levels of a factor in `newdata`. In the case of `brms::brm()`, you must also pass `allow_new_levels = TRUE` here to include new levels (see `brms::posterior_predict()`).

`category` For *some* ordinal, multinomial, and multivariate models (notably, `brms::brm()` models but *not* `rstanarm::stan_polr()` models), multiple sets of rows will be returned per input row for `epred_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 different response variables. The `category` argument specifies the name of the column to put the category names (or variable names) into in the resulting data frame. The default name of this column (`".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 different 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.

`dpar` For `add_epred_draws()` and `add_linpred_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 `brms::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 `NULL` or `FALSE` (the default), distributional regression parameters are not included.

`n` (Deprecated). Use `ndraws`.

`scale` (Deprecated). Use the appropriate function (`epred_draws()` or `linpred_draws()`) depending on what type of distribution you want. For `linpred_draws()`, you may want the `transform` argument. See `rstanarm::posterior_linpred()` or `brms::posterior_linpred()`.

`prediction, residual` (Deprecated). Use `value`.

Details

Consider a model like:

$$\begin{aligned} y &\sim \text{SomeDist}(\theta_1, \theta_2) \\ f_1(\theta_1) &= \alpha_1 + \beta_1 x \\ f_2(\theta_2) &= \alpha_2 + \beta_2 x \end{aligned}$$

This model has:

- an outcome variable, y
- a response distribution, SomeDist, having parameters θ_1 (with link function f_1) and θ_2 (with link function f_2)
- a single predictor, x
- coefficients α_1, β_1, α_2, and β_2

We fit this model to some observed data, y_{obs}, and predictors, x_{obs}. Given new values of predictors, x_{new}, supplied in the data frame `newdata`, the functions for posterior draws are defined as follows:

- `add_predicted_draws()` adds draws from the **posterior predictive distribution**, $p(y_{\text{new}}|x_{\text{new}}, y_{\text{obs}})$, to the data. It corresponds to `rstanarm::posterior_predict()` or `brms::posterior_predict()`.
- `add_epred_draws()` adds draws from the **expectation of the posterior predictive distribution**, aka the conditional expectation, $E(y_{\text{new}}|x_{\text{new}}, y_{\text{obs}})$, to the data. It corresponds to `rstanarm::posterior_epred()` or `brms::posterior_epred()`. Not all models support this function.
- `add_linpred_draws()` adds draws from the **posterior linear predictors** to the data. It corresponds to `rstanarm::posterior_linpred()` or `brms::posterior_linpred()`. Depending on the model type and additional parameters passed, this may be:
  - The untransformed linear predictor, e.g. $p(f_1(\theta_1)|x_{\text{new}}, y_{\text{obs}}) = p(\alpha_1 + \beta_1 x_{\text{new}}|x_{\text{new}}, y_{\text{obs}})$. This is returned by `add_linpred_draws(transform = FALSE)` for **brms** and **rstanarm** models. It is analogous to `type = "link"` in `predict.glm()`.
  - The inverse-link transformed linear predictor, e.g. $p(\theta_1|x_{\text{new}}, y_{\text{obs}}) = p(f_1^{-1}(\alpha_1 + \beta_1 x_{\text{new}})|x_{\text{new}}, y_{\text{obs}})$. This is returned by `add_linpred_draws(transform = TRUE)` for **brms** and **rstanarm** models. It is analogous to `type = "response"` in `predict.glm()`.

  NOTE: `add_linpred_draws(transform = TRUE)` and `add_epred_draws()` may be equivalent but are not guaranteed to be. They are equivalent when the expectation of the response distribution is equal to its first parameter, i.e. when $E(y) = \theta_1$. Many distributions have this property (e.g. Normal distributions, Bernoulli distributions), but not all. If you want the expectation of the posterior predictive, it is best to use `add_epred_draws()` if available, and if not available, verify this property holds prior to using `add_linpred_draws()`.
- `add_residual_draws()` adds draws from residuals, $p(y_{\text{obs}} - y_{\text{new}}|x_{\text{new}}, y_{\text{obs}})$, to the data. It corresponds to `brms::residuals.brmsfit()`.

The corresponding functions without `add_` as a prefix are alternate spellings with the opposite order of the first two arguments: e.g. `add_predicted_draws(newdata, object)` versus `predicted_draws(object, newdata)`. This facilitates use in data processing pipelines that start either with a data frame or a model.

Given equal choice between the two, the spellings prefixed with `add_` are preferred.

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, `epred_draws` includes a column with its name specified by the `epred` argument (default `".epred"`); `linpred_draws` includes a column with its name specified by the `linpred` argument (default `".linpred"`), and `predicted_draws` contains a column with its name specified by the `.prediction` argument (default `".prediction"`). 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

```
## Not run:

library(ggplot2)
library(dplyr)
library(brms)
library(modelr)

theme_set(theme_light())

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

# draw 100 lines from the posterior means and overplot them
mtcars %>%
  group_by(cyl) %>%
  data_grid(hp = seq_range(hp, n = 101)) %>%
  # NOTE: only use ndraws here when making spaghetti plots; for
  # plotting intervals it is always best to use all draws (omit ndraws)
  add_epred_draws(m_mpg, ndraws = 100) %>%
  ggplot(aes(x = hp, y = mpg, color = ordered(cyl))) +
  geom_line(aes(y = .epred, 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 = 101)) %>%
```

```
  add_predicted_draws(m_mpg) %>%
  ggplot(aes(x = hp, y = mpg, color = ordered(cyl))) +
  stat_lineribbon(aes(y = .prediction), .width = c(.99, .95, .8, .5), alpha = 0.25) +
  geom_point(data = mtcars) +
  scale_fill_brewer(palette = "Greys")


## End(Not run)
```

`add_epred_rvars` *Add* `rvar`*s for the linear predictor, posterior expectation, posterior predictive, or residuals of a model to a data frame*

Description

Given a data frame and a model, adds `rvar`s of draws from the linear/link-level predictor, the expectation of the posterior predictive, or the posterior predictive to the data frame.

Usage

```
add_epred_rvars(
  newdata,
  object,
  ...,
  value = ".epred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  dpar = NULL,
  columns_to = NULL
)

epred_rvars(
  object,
  newdata,
  ...,
  value = ".epred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  dpar = NULL,
  columns_to = NULL
)

## Default S3 method:
epred_rvars(
  object,
  newdata,
```

```
  ...,
  value = ".epred",
  seed = NULL,
  dpar = NULL,
  columns_to = NULL
)

## S3 method for class 'stanreg'
epred_rvars(
  object,
  newdata,
  ...,
  value = ".epred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  dpar = NULL,
  columns_to = NULL
)

## S3 method for class 'brmsfit'
epred_rvars(
  object,
  newdata,
  ...,
  value = ".epred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  dpar = NULL,
  columns_to = NULL
)

add_linpred_rvars(
  newdata,
  object,
  ...,
  value = ".linpred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  dpar = NULL,
  columns_to = NULL
)

linpred_rvars(
  object,
  newdata,
```

```
  ...,
  value = ".linpred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  dpar = NULL,
  columns_to = NULL
)

## Default S3 method:
linpred_rvars(
  object,
  newdata,
  ...,
  value = ".linpred",
  seed = NULL,
  dpar = NULL,
  columns_to = NULL
)

## S3 method for class 'stanreg'
linpred_rvars(
  object,
  newdata,
  ...,
  value = ".linpred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  dpar = NULL,
  columns_to = NULL
)

## S3 method for class 'brmsfit'
linpred_rvars(
  object,
  newdata,
  ...,
  value = ".linpred",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  dpar = NULL,
  columns_to = NULL
)

add_predicted_rvars(
  newdata,
```

```
  object,
  ...,
  value = ".prediction",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  columns_to = NULL
)

predicted_rvars(
  object,
  newdata,
  ...,
  value = ".prediction",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  columns_to = NULL
)

## Default S3 method:
predicted_rvars(
  object,
  newdata,
  ...,
  value = ".prediction",
  seed = NULL,
  columns_to = NULL
)

## S3 method for class 'stanreg'
predicted_rvars(
  object,
  newdata,
  ...,
  value = ".prediction",
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  columns_to = NULL
)

## S3 method for class 'brmsfit'
predicted_rvars(
  object,
  newdata,
  ...,
  value = ".prediction",
```

```
  ndraws = NULL,
  seed = NULL,
  re_formula = NULL,
  columns_to = NULL
)
```

Arguments

newdata	Data frame to generate predictions from.
object	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_rvars()) support a wider range of models, to work with add_epred_rvars(), add_predicted_rvars(), etc. a model must provide an interface for generating predictions, thus more generic Bayesian modeling interfaces like runjags and rstan are not directly supported for these functions (only wrappers around those languages that provide predictions, like rstanarm and brm, are supported here).
...	Additional arguments passed to the underlying prediction method for the type of model given.
value	The name of the output column: • for [add_]epred_rvars(), defaults to ".epred". • for [add_]predicted_rvars(), defaults to ".prediction". • for [add_]linpred_rvars(), defaults to ".linpred".
ndraws	The number of draws to return, or NULL to return all draws.
seed	A seed to use when subsampling draws (i.e. when ndraws 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 effects. Some model types (such as brms::brmsfit and rstanarm::stanreg-objects) allow marginalizing over grouping factors by specifying new levels of a factor in newdata. In the case of brms::brm(), you must also pass allow_new_levels = TRUE here to include new levels (see brms::posterior_predict()).
dpar	For add_epred_rvars() and add_linpred_rvars(): 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 brms::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 NULL or FALSE (the default), distributional regression parameters are not included.
columns_to	For *some* models, such as ordinal, multinomial, and multivariate models (notably, brms::brm() models but *not* rstanarm::stan_polr() models), the column of predictions in the resulting data frame may include nested columns. For example, for ordinal/multinomial models, these columns correspond to different categories of the response variable. It may be more convenient to turn these nested columns into rows in the output; if this is desired, set columns_to to

a string representing the name of a column you would like the column names to be placed in. In this case, a `.row` column will also be added to the result indicating which rows of the output correspond to the same row in `newdata`. See `vignette("tidy-posterior")` for examples of dealing with output ordinal models.

Details

Consider a model like:

$$\begin{aligned} y &\sim \text{SomeDist}(\theta_1, \theta_2) \\ f_1(\theta_1) &= \alpha_1 + \beta_1 x \\ f_2(\theta_2) &= \alpha_2 + \beta_2 x \end{aligned}$$

This model has:

- an outcome variable, y
- a response distribution, SomeDist, having parameters θ_1 (with link function f_1) and θ_2 (with link function f_2)
- a single predictor, x
- coefficients α_1, β_1, α_2, and β_2

We fit this model to some observed data, y_{obs}, and predictors, x_{obs}. Given new values of predictors, x_{new}, supplied in the data frame `newdata`, the functions for posterior draws are defined as follows:

- `add_predicted_rvars()` adds `rvar`s containing draws from the **posterior predictive distribution**, $p(y_{\text{new}}|x_{\text{new}}, y_{\text{obs}})$, to the data. It corresponds to `rstanarm::posterior_predict()` or `brms::posterior_predict()`.
- `add_epred_rvars()` adds `rvar`s containing draws from the **expectation of the posterior predictive distribution**, aka the conditional expectation, $E(y_{\text{new}}|x_{\text{new}}, y_{\text{obs}})$, to the data. It corresponds to `rstanarm::posterior_epred()` or `brms::posterior_epred()`. Not all models support this function.
- `add_linpred_rvars()` adds `rvar`s containing draws from the **posterior linear predictors** to the data. It corresponds to `rstanarm::posterior_linpred()` or `brms::posterior_linpred()`. Depending on the model type and additional parameters passed, this may be:
  - The untransformed linear predictor, e.g. $p(f_1(\theta_1)|x_{\text{new}}, y_{\text{obs}}) = p(\alpha_1 + \beta_1 x_{\text{new}}|x_{\text{new}}, y_{\text{obs}})$. This is returned by `add_linpred_rvars(transform = FALSE)` for **brms** and **rstanarm** models. It is analogous to `type = "link"` in `predict.glm()`.
  - The inverse-link transformed linear predictor, e.g. $p(\theta_1|x_{\text{new}}, y_{\text{obs}}) = p(f_1^{-1}(\alpha_1 + \beta_1 x_{\text{new}})|x_{\text{new}}, y_{\text{obs}})$. This is returned by `add_linpred_rvars(transform = TRUE)` for **brms** and **rstanarm** models. It is analogous to `type = "response"` in `predict.glm()`.

  NOTE: `add_linpred_rvars(transform = TRUE)` and `add_epred_rvars()` may be equivalent but are not guaranteed to be. They are equivalent when the expectation of the response distribution is equal to its first parameter, i.e. when $E(y) = \theta_1$. Many distributions have this property (e.g. Normal distributions, Bernoulli distributions), but not all. If you want the expectation of the posterior predictive, it is best to use `add_epred_rvars()` if available, and if not available, verify this property holds prior to using `add_linpred_rvars()`.

The corresponding functions without add_ as a prefix are alternate spellings with the opposite order of the first two arguments: e.g. `add_predicted_rvars(newdata, object)` versus `predicted_rvars(object, newdata)`. This facilitates use in data processing pipelines that start either with a data frame or a model.

Given equal choice between the two, the spellings prefixed with add_ are preferred.

Value

A data frame (actually, a tibble) equal to the input `newdata` with additional columns added containing `rvar`s representing the requested predictions or fits.

Author(s)

Matthew Kay

See Also

`add_predicted_draws()` for the analogous functions that use a long-data-frame-of-draws format instead of a data-frame-of-`rvar`s format. See `spread_rvars()` for manipulating posteriors directly.

Examples

```
## Not run:

library(ggplot2)
library(dplyr)
library(posterior)
library(brms)
library(modelr)

theme_set(theme_light())

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

# Look at mean predictions for some cars (epred) and compare to
# the exponeniated mu parameter of the lognormal distribution (linpred).
# Notice how they are NOT the same. This is because exp(mu) for a
# lognormal distribution is equal to its median, not its mean.
mtcars %>%
  select(hp, cyl, mpg) %>%
  add_epred_rvars(m_mpg) %>%
  add_linpred_rvars(m_mpg, value = "mu") %>%
  mutate(expmu = exp(mu), .epred - expmu)

# plot intervals around conditional means (epred_rvars)
mtcars %>%
  group_by(cyl) %>%
  data_grid(hp = seq_range(hp, n = 101)) %>%
  add_epred_rvars(m_mpg) %>%
```

```
  ggplot(aes(x = hp, color = ordered(cyl), fill = ordered(cyl))) +
  stat_lineribbon(aes(dist = .epred), .width = c(.95, .8, .5), alpha = 1/3) +
  geom_point(aes(y = mpg), data = mtcars) +
  scale_color_brewer(palette = "Dark2") +
  scale_fill_brewer(palette = "Set2")

# plot posterior predictive intervals (predicted_rvars)
mtcars %>%
  group_by(cyl) %>%
  data_grid(hp = seq_range(hp, n = 101)) %>%
  add_predicted_rvars(m_mpg) %>%
  ggplot(aes(x = hp, color = ordered(cyl), fill = ordered(cyl))) +
  stat_lineribbon(aes(dist = .prediction), .width = c(.95, .8, .5), alpha = 1/3) +
  geom_point(aes(y = mpg), data = mtcars) +
  scale_color_brewer(palette = "Dark2") +
  scale_fill_brewer(palette = "Set2")


## End(Not run)
```

`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::emmeans()`

Examples

```
library(magrittr)
library(coda)

data(line, package = "coda")

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

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

---

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

Description

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

Usage

```
compare_levels(
  data,
  variable,
  by,
  fun = `-`,
  comparison = "default",
```

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

Arguments

data	Long-format data.frame of draws such as returned by spread_draws() or gather_draws(). If data is a grouped data frame, comparisons will be made within groups (if one of the groups in the data frame is the by column, that specific group will be ignored, as it is not possible to make comparisons both within some variable and across it simultaneously).
variable	Bare (unquoted) name of a column in data representing the variable to compare across levels. Can be a numeric variable (as in long-data-frame-of-draws format) or a posterior::rvar.
by	Bare (unquoted) name of a column in data that is a factor or ordered. The value of variable will be compared across pairs of levels of this factor.
fun	Binary function to use for comparison. For each pair of levels of by we are comparing (as determined by comparison), compute the result of this function.
comparison	One of (a) the comparison types ordered, control, pairwise, or default (may also be given as strings, e.g. "ordered"), see *Details*; (b) a user-specified function that takes a factor and returns a list of pairs of names of levels to compare (as strings) and/or unevaluated expressions containing representing the comparisons to make; or (c) a list of pairs of names of levels to compare (as strings) and/or unevaluated expressions representing the comparisons to make, e.g.: list(c("a", "b"), c("b", "c")) or exprs(a - b, b - c), both of which would compare level "a" against "b" and level "b" against "c". Note that the unevaluated expression syntax ignores the fun argument, can include any other functions desired (e.g. variable transformations), and can even include more than two levels or other columns in data. Types (b) and (c) may use named lists, in which case the provided names are used in the output variable column instead converting the unevaluated expression to a string. You can also use emmeans_comparison() to generate a comparison function based on contrast methods from the emmeans package.
draw_indices	Character vector of column names that should be treated as indices of draws. Operations are done within combinations of these values. The default is c(".chain", ".iteration", ".draw"), which is the same names used for chain, iteration, and draw indices returned by tidy_draws(). Names in draw_indices that are not found in the data are ignored.
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_epred_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

emmeans_comparison() to use emmeans-style contrast methods with compare_levels().

Examples

```
library(dplyr)
library(ggplot2)

data(RankCorr, package = "ggdist")

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

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

# Or let's plot comparisons of all levels of j within
# all levels of i
RankCorr %>%
  spread_draws(b[i,j]) %>%
  group_by(i) %>%
```

```
  compare_levels(b, by = j) %>%
  ggplot(aes(x = b, y = j)) +
  stat_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

```
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 dplyr::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

```
# 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` *Density bins and histogram bins as data frames*

Description

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

Usage

```
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 the now-deprecated `predict_curve()`.

`density_bins` computes a kernel density estimate using `density()`.

`histogram_bins` computes a density histogram using `hist()`.

Value

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

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

Author(s)

Matthew Kay

See Also

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

Examples

```
## Not run:

library(ggplot2)
library(dplyr)
library(brms)
library(modelr)

theme_set(theme_light())

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

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


## End(Not run)
```

---

emmeans_comparison *Use emmeans contrast methods with compare_levels*

Description

Convert emmeans contrast methods into comparison functions suitable for use with compare_levels().

Usage

```
emmeans_comparison(method, ...)
```

Arguments

method	An emmeans-style contrast method. One of: (1) a string specifying the name of an emmeans contrast method, like "pairwise", "trt.vs.ctrl", "eff"; or (2) an emmeans-style contrast function itself, like emmeans::pairwise.emmc, emmeans::trt.vs.ctrl.emmc, etc, or a custom function that takes a vector of factor levels and returns a contrast matrix.
...	Arguments passed on to the contrast method.

Details

Given an emmeans contrast method name as a string (e.g., "pairwise", "trt.vs.ctrl", etc) or an emmeans-style contrast function (e.g., emmeans::pairwise.emmc, emmeans::trt.vs.ctrl.emmc, etc), emmeans_comparison() returns a new function that can be used in the comparison argument to compare_levels() to compute those contrasts.

Value

A function that takes a single argument, var, containing a variable to generate contrasts for (e.g., a factor or a character vector) and returns a function that generates a list of named unevaluated expressions representing different contrasts of that variable. This function is suitable to be used as the comparison argument in compare_levels().

Author(s)

Matthew Kay

See Also

compare_levels(), emmeans::contrast-methods. See gather_emmeans_draws() for a different approach to using emmeans with tidybayes.

Examples

```
library(dplyr)
library(ggplot2)

data(RankCorr, package = "ggdist")

# emmeans contrast methods return matrices. E.g. the "eff" comparison
# compares each level to the average of all levels:
emmeans:::eff.emmc(c("a","b","c","d"))

# tidybayes::compare_levels() can't use a contrast matrix like this
# directly; it takes arbitrary expressions of factor levels. But
# we can use `emmeans_comparison` to generate the equivalent expressions:
emmeans_comparison("eff")(c("a","b","c","d"))

# We can use the "eff" comparison type with `compare_levels()` as follows:
RankCorr %>%
```

```
  spread_draws(b[i,j]) %>%
  filter(j == 1) %>%
  compare_levels(b, by = i, comparison = emmeans_comparison("eff")) %>%
  median_qi()
```

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

Description

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

Usage

```
gather_draws(
  model,
  ...,
  regex = FALSE,
  sep = "[, ]",
  ndraws = NULL,
  seed = NULL,
  draw_indices = c(".chain", ".iteration", ".draw"),
  n
)

spread_draws(
  model,
  ...,
  regex = FALSE,
  sep = "[, ]",
  ndraws = NULL,
  seed = NULL,
  draw_indices = c(".chain", ".iteration", ".draw"),
  n
)
```

Arguments

`model`	A supported Bayesian model fit. Tidybayes supports a variety of model objects; for a full list of supported models, see tidybayes-models.
`...`	Expressions in the form of `variable_name[dimension_1, dimension_2, ...] | wide_dimension`. See *Details*.
`regex`	If `TRUE`, variable names are treated as regular expressions and all column matching the regular expression and number of dimensions are included in the output. Default `FALSE`.

sep	Separator used to separate dimensions in variable names, as a regular expression.
ndraws	The number of draws to return, or NULL to return all draws.
seed	A seed to use when subsampling draws (i.e. when ndraws is not NULL).
draw_indices	Character vector of column names that should be treated as indices of draws. Operations are done within combinations of these values. The default is c(".chain", ".iteration", ".draw"), which is the same names used for chain, iteration, and draw indices returned by tidy_draws(). Names in draw_indices that are not found in the data are ignored.
n	(Deprecated). Use ndraws.

Details

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

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

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

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

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

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

gather_draws(model, a[i], b[i,v]) on the same model would return a grouped data frame (grouped by i and v), with:

- column ".chain": the chain number
- column ".iteration": the iteration number

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

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

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

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

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

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

Which is roughly equivalent to this:

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

Similarly, this:

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

Is roughly equivalent to this:

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

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

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

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

Each of which is roughly equivalent to this:

```
spread_draws(model, 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(model, 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(model, 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(model, b[i,v] | v)
```

is roughly equivalent to this:

```
spread_draws(model, 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:

```
model %>% 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(model, 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(model, `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

spread_rvars(), recover_types(), compose_data().

Examples

```
library(dplyr)
library(ggplot2)

data(RankCorr, package = "ggdist")

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

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


RankCorr %>%
```

```
  gather_draws(b[i, j], tau[i], u_tau[i])

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

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

Description

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

Usage

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

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

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

Arguments

`object`	An `emmGrid` object such as returned by `emmeans::ref_grid()` or `emmeans::emmeans()`.
`value`	The name of the output column to use to contain the values of draws. Defaults to `".value"`.
`...`	Additional arguments passed to the underlying method for the type of object given.
`grid`	If `object` is an `emmeans::emm_list()`, the name of the output column to use to contain the name of the reference grid that a given row corresponds to. Defaults to `".grid"`.

Details

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

Value

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

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

Author(s)

Matthew Kay

See Also

`emmeans::emmeans()`

Examples

```
## Not run:

library(dplyr)
library(magrittr)
library(brms)
library(emmeans)

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

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

# or we could get pairwise differences:
m %>%
  emmeans(~ condition) %>%
```

```
  contrast(method = "pairwise") %>%
  gather_emmeans_draws() %>%
  median_qi()

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


## End(Not run)
```

`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::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(t_x, t_y)) +
  geom_point() +
  facet_grid(vars(g_row), vars(g_col))

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

df %>%
  gather_pairs(g, t, triangle = "both") %>%
```

```
  ggplot(aes(.x, .y)) +
  geom_point() +
  facet_grid(vars(.row), vars(.col))

data(line, package = "coda")

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

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

`gather_rvars` *Extract draws from a Bayesian model into tidy data frames of random variables*

Description

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

Usage

```
gather_rvars(model, ..., ndraws = NULL, seed = NULL)

spread_rvars(model, ..., ndraws = NULL, seed = NULL)
```

Arguments

`model`	A supported Bayesian model fit. Tidybayes supports a variety of model objects; for a full list of supported models, see tidybayes-models.
`...`	Expressions in the form of `variable_name[dimension_1, dimension_2, ...]`. See *Details*.
`ndraws`	The number of draws to return, or `NULL` to return all draws.
`seed`	A seed to use when subsampling draws (i.e. when `ndraws` is not `NULL`).

Details

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

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

`spread_rvars` will spread names of variables in the model across the data frame as column names, whereas `gather_rvars` will gather variable names into a single column named `".variable"` and place values of variables into a column named `".value"`. To use naming schemes from other packages (such as `broom`), consider passing results through functions like `to_broom_names()` or `to_ggmcmc_names()`.

For example, `spread_rvars(model, a[i], b[i,v])` might return a data frame with:

- column `"i"`: value in `1:5`
- column `"v"`: value in `1:10`
- column `"a"`: `rvar` containing draws from `"a[i]"`
- column `"b"`: `rvar` containing draws from `"b[i,v]"`

`gather_rvars(model, a[i], b[i,v])` on the same model would return a data frame with:

- column `"i"`: value in `1:5`
- column `"v"`: value in `1:10`, or `NA` on rows where `".variable"` is `"a"`.
- column `".variable"`: value in `c("a", "b")`.
- column `".value"`: `rvar` containing draws from `"a[i]"` (when `".variable"` is `"a"`) or `"b[i,v]"` (when `".variable"` is `"b"`)

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

```
model %>%
 recover_types(data)
 spread_rvars(model, b[i,v])
```

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

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

```
spread_rvars(model, x, d[i], b[i,v])
```

Which is roughly equivalent to this:

```
spread_rvars(model, x) %>%
 inner_join(spread_rvars(model, d[i])) %>%
 inner_join(spread_rvars(model, b[i,v]))
```

Similarly, this:

```
gather_rvars(model, x, d[i], b[i,v])
```

Is roughly equivalent to this:

```
bind_rows(
 gather_rvars(model, x),
 gather_rvars(model, d[i]),
 gather_rvars(model, b[i,v])
)
```

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

```
spread_rvars(model, c(w, x, y, z)[i,v])

spread_rvars(model, cbind(w, x, y, z)[i,v])  # equivalent
```

Each of which is roughly equivalent to this:

```
spread_rvars(model, w[i,v], x[i,v], y[i,v], z[i,v])
```

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

Dimensions can be left nested in the resulting `rvar` objects by leaving their names blank; e.g. `spread_rvars(model, b[i,])` will place the first index (`i`) into rows of the data frame but leave the second index nested in the `b` column (see *Examples* below).

Value

A data frame.

Author(s)

Matthew Kay

See Also

`spread_draws()`, `recover_types()`, `compose_data()`. See also `posterior::rvar()` and `posterior::as_draws_rvars()` the functions that power `spread_rvars` and `gather_rvars`.

Examples

```
library(dplyr)

data(RankCorr, package = "ggdist")

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

# leaving an index out nests the index in the column containing the rvar
RankCorr %>%
  spread_rvars(b[i, ])

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

# gather_rvars places variables and values in a longer format data frame
RankCorr %>%
  gather_rvars(b[i, j], tau[i], typical_r)
```

---

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

Description

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

Usage

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

Arguments

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

Details

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

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

- column `".chain"`: the chain number
- column `".iteration"`: the iteration number

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

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

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

In this example, this call:

```
gather_variables(data)
```

Is roughly equivalent to:

```
data %>%
  gather(.variable, .value, -c(.chain, .iteration, .draw, i, v)) %>%
  group_by(.variable, .add = TRUE)
```

Value

A data frame.

Author(s)

Matthew Kay

See Also

spread_draws(), tidy_draws().

Examples

```
library(dplyr)

data(RankCorr, package = "ggdist")

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()
```

---

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

```
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 = "ggdist")
get_variables(RankCorr)
```

---

nest_rvars *Nest and unnest* rvar *columns in data frames*

Description

Converts between data-frame-of-rvars format and long-data-frame-of-draws formats by nesting or unnesting all columns containing `posterior::rvar` objects.

Usage

```
nest_rvars(data)

unnest_rvars(data)
```

Arguments

data	A data frame to nest or unnest. • For `nest_rvars()`, the data frame should be in long-data-frame-of-draws format; i.e. it should contain a `.draw` column (and optionally `.chain` and `.iteration` columns) indexing draws. It should be a grouped by any columns that are not intended to be nested. • For `unnest_rvars()`, the data frame should have at least one column that is an `rvar`; all `rvar` columns will be unnested.

Value

For `nest_rvars()`, returns a data frame without `.chain`, `.iteration`, and `.draw` columns, where all non-grouped columns have been converted to rvars.

For `unnest_rvars()`, returns a data frame with `.chain`, `.iteration`, and `.draw` columns added, where every rvar column in the input has been converted to (one or more) columns containing draws from those rvars in long format. The result is grouped by all non-rvar columns in the input; this ensures that `nest_rvars(unnest_rvars(x))` returns `x`.

Examples

```
library(dplyr)

data(RankCorr, package = "ggdist")

# here's a data frame with some rvars
rvar_df = RankCorr %>%
  spread_rvars(b[i,], tau[i])
rvar_df

# we can unnest it into long format.
# note how the result is grouped by all non-rvar input columns,
# and nested indices in `b` are converted into columns.
draws_df = rvar_df %>%
  unnest_rvars()
draws_df

# calling nest_rvars() again on the result of unnest_rvars()
# recovers the original data frame
nest_rvars(draws_df)
```

---

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

```
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"))
```

---

predict_curve *Deprecated: Prediction curves for arbitrary functions of posteriors*

Description

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

Usage

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

predict_curve_density(
  data,
  formula,
```

```
  summary = function(...) density_bins(..., n = n),
  n = 50,
  ...
)
```

Arguments

data	A data.frame, tibble, or grouped_df representing posteriors from a Bayesian model as might be obtained through spread_draws(). Grouped data frames as returned by group_by() are supported.
formula	A formula specifying the prediction curve. The left-hand side of the formula should be a name representing the name of the column that will hold the predicted response in the returned data frame. The right-hand side is an expression that may include numeric columns from data and variables passed into this function in ....
summary	The function to apply to summarize each predicted response. Useful functions (if you just want a curve) might be median(), mean(), or Mode(). If you want predictive distribution at each point on the curve, try density_bins() or histogram_bins().
...	Variables defining the curve. The right-hand side of formula is evaluated for every combination of values of variables in ....
n	For predict_curve_density, the number of bins to use to represent the distribution at each point on the curve.

Details

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

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

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

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

Value

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

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

Author(s)

Matthew Kay

See Also

See density_bins().

Examples

```
# Deprecated; see examples for density_bins
```

---

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

Description

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

Usage

```
recover_types(model, ...)
```

Arguments

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

Details

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

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

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

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

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

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

Value

A decorated version of model.

Author(s)

Matthew Kay

See Also

spread_draws(), gather_draws(), compose_data().

Examples

```
## Not run:

library(dplyr)
library(magrittr)
library(rstan)

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

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

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

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

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

```
## End(Not run)
```

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

Description

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

Usage

```
sample_draws(data, ndraws, draw = ".draw", seed = NULL)
```

Arguments

`data`	Data frame to sample from
`ndraws`	The number of draws to return, or `NULL` to return all draws.
`draw`	The name of the column indexing the draws; default `".draw"`.
`seed`	A seed to use when subsampling draws (i.e. when `ndraws` is not `NULL`).

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

```
## Not run:

library(ggplot2)
library(dplyr)
library(brms)
library(modelr)

theme_set(theme_light())

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

```
  chains = 1, iter = 500)

# draw 100 fit lines from the posterior and overplot them
mtcars %>%
  group_by(cyl) %>%
  data_grid(hp = seq_range(hp, n = 101)) %>%
  add_epred_draws(m_mpg) %>%
  # NOTE: only use sample_draws here when making spaghetti plots; for
  # plotting intervals it is always best to use all draws
  sample_draws(100) %>%
  ggplot(aes(x = hp, y = mpg, color = ordered(cyl))) +
  geom_line(aes(y = .epred, group = paste(cyl, .draw)), alpha = 0.25) +
  geom_point(data = mtcars)


## End(Not run)
```

`summarise_draws.grouped_df`
Summaries of draws in `grouped_df` *objects*

Description

An implementation of `posterior::summarise_draws()` for grouped data frames (`dplyr::grouped_df` objects) such as returned by `dplyr::group_by()` and the various grouped-data-aware functions in tidybayes, such as `spread_draws()`, `gather_draws()`, `add_epred_draws()`, and `add_predicted_draws()`. This function provides a quick way to get a variety of summary statistics and diagnostics on draws.

Usage

```
## S3 method for class 'grouped_df'
summarise_draws(.x, ...)
```

Arguments

`.x`	A grouped data frame (`dplyr::grouped_df` object) such as returned by `dplyr::group_by()` where the data frame in each group (ignoring grouping columns) has the structure of a `posterior::draws_df()` object: `".chain"`, `".iteration"`, and `".draw"` columns, with the remaining (non-grouping) columns being draws from variables.
`...`	Name-value pairs of summary or diagnostic functions. The provided names will be used as the names of the columns in the result *unless* the function returns a named vector, in which case the latter names are used. The functions can be specified in any format supported by as_function(). See **Examples**.

Details

While `posterior::summarise_draws()` can operate on tidy data frames of draws in the `posterior::draws_df()` format, that format does not support grouping columns. This provides an implementation of `summarise_draws()` that does support grouped data tables, essentially applying `posterior::summarise_draws()` to every sub-table of `.x` implied by the groups defined on the data frame.

See `posterior::summarise_draws()` for more details on the summary statistics and diagnostics you can use with this function. If you just want point summaries and intervals (not diagnostics), particularly for plotting, see `point_interval()`, which returns long-format data tables more suitable for that purpose (especially if you want to plot multiple uncertainty levels).

Value

A data frame (actually, a tibble) with all grouping columns from `.x`, a `"variable"` column containing variable names from `.x`, and the remaining columns containing summary statistics and diagnostics.

Author(s)

Matthew Kay

See Also

`posterior::summarise_draws()`, `point_interval()`

Examples

```
library(posterior)
library(dplyr)

d = posterior::example_draws()

# The default posterior::summarise_draws() summarises all variables without
# splitting out indices:
summarise_draws(d)

# The grouped_df implementation of summarise_draws() in tidybayes can handle
# output from spread_draws(), which is a grouped data table with the indices
# (here, `i`) left as columns:
d %>%
  spread_draws(theta[i]) %>%
  summarise_draws()

# Summary functions can also be provided, as in posterior::summarise_draws():
d %>%
  spread_draws(theta[i]) %>%
  summarise_draws(median, mad, rhat, ess_tail)
```

`tidybayes-deprecated` *Deprecated functions, arguments, and column names in tidybayes*

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 (see *Deprecated Functions* and *Deprecated Arguments and Column Names* below) or due to the deprecation of horizontal shortcut geoms and stats in tidybayes 2.1 (see *Deprecated Horizontal Shortcut Geoms and Stats*).

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.

Functions deprecated in tidybayes 3.0:

- `fitted_draws` and `add_fitted_draws` are deprecated because their names were confusing: it was unclear to many users if these functions returned draws from the posterior predictive, the mean of the posterior predictive, or the linear predictor (and depending on model type it might have been either of the latter). Use `epred_draws()`/`add_epred_draws()` if you want the expectation of the posterior predictive and use `linpred_draws()`/`add_linpred_draws()` if you want the linear predictor.

Functions deprecated in tidybayes 1.0:

- `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. (though see the note above about the deprecation of `fitted_draws` in favor of `epred_draws()` and `linpred_draws()`).

- 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. stat_eye() offers a horizontal eye plot geom that can be used instead of ggeye.
- See the sections below for additional deprecated functions, including horizontal geoms, stats, and point_intervals

Deprecated Eye Geom Spellings

geom_eye, geom_eyeh, and geom_halfeyeh are deprecated spellings of stat_eye() and stat_halfeye() from before name standardization of stats and geoms. Use those functions instead.

Deprecated Horizontal Shortcut Geoms and Stats

Due to the introduction of automatic orientation detection in tidybayes 2.1, shortcut geoms and stats (which end in h) are no longer necessary, and are deprecated. In most cases, these can simply be replaced with the same geom without the h suffix and they will remain horizontal; e.g. stat_halfeyeh(...) can simply be replaced with stat_halfeye(...). If automatic orientation detection fails, override it with the orientation parameter; e.g. stat_halfeye(orientation = "horizontal").

These deprecated stats and geoms include:

- stat_eyeh / stat_dist_eyeh
- stat_halfeyeh / stat_dist_halfeyeh
- geom_slabh / stat_slabh / stat_dist_slabh
- geom_intervalh / stat_intervalh / stat_dist_intervalh
- geom_pointintervalh / stat_pointintervalh / stat_dist_pointintervalh
- stat_gradientintervalh / stat_dist_gradientintervalh
- stat_cdfintervalh / stat_dist_cdfintervalh
- stat_ccdfintervalh / stat_dist_ccdfintervalh
- geom_dotsh / stat_dotsh / stat_dist_dotsh
- geom_dotsintervalh / stat_intervalh / stat_dist_intervalh
- stat_histintervalh

Deprecated Horizontal Point/Interval Functions

These functions ending in h (e.g., point_intervalh, median_qih) used to be needed for use with ggstance::stat_summaryh, but are no longer necessary because ggplot2::stat_summary() supports automatic orientation detection, so they have been deprecated. They behave identically to the corresponding function without the h, except that when passed a vector, they return a data frame with x/xmin/xmax instead of y/ymin/ymax.

- point_intervalh
- mean_qih / median_qih / mode_qih
- mean_hdih / median_hdih / mode_hdih
- mean_hdcih / median_hdcih / mode_hdcih

Deprecated Arguments and Column Names

Arguments deprecated in tidybayes 3.0 are:

- The n argument is now called ndraws in predicted_draws(), linpred_draws(), etc. This prevents some bugs due to partial matching of argument names where n might be mistaken for newdata.
- The value argument in linpred_draws() is now spelled linpred and defaults to ".linpred" in the same way that the predicted_draws() and epred_draws() functions work.
- The scale argument in linpred_draws() is no longer allowed (use transform instead) as this naming scheme only made sense when linpred_draws() was an alias for fitted_draws(), which it no longer is (see note above about the deprecation of fitted_draws()).

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

Arguments and column names deprecated in tidybayes 1.0 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()`, `gather_draws()`, `spread_rvars()`, and `gather_rvars()`. These models/formats include:

- rstan models
- cmdstanr models
- `brms::brm()` models
- rstanarm models
- `runjags::runjags()` models
- `rjags::jags.model()` models, if sampled using `rjags::coda.samples()`
- `jagsUI::jags()` models
- `MCMCglmm::MCMCglmm()` models
- `coda::mcmc()` and `coda::mcmc.list()` objects, which are output by several model types.
- posterior::draws objects
- Any object with an implementation of `posterior::as_draws_df()` or `posterior::as_draws()`. For a list of those available in your environment, run `methods(as_draws_df)` or `methods(as_draws)`
- Any object with an implementation of `coda::as.mcmc.list()`. For a list of those available in your environment, run `methods(as.mcmc.list)`

If you install the tidybayes.rethinking package, 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_epred_draws()`, `add_predicted_draws()`, `add_linpred_draws()`, `add_epred_rvars()`, `add_predicted_rvars()`, and `add_linpred_rvars()`:

- `brms::brm()` models
- rstanarm models
- any package with implementations of `rstantools::posterior_epred()`, `rstantools::posterior_predict()`, or `rstantools::posterior_linpred()` that include an argument called `newdata` which takes a data frame of predictors.

If your model type is not in the above list, you may still be able to use the `add_draws()` function to turn matrices of predictive draws (or fit draws) into tidy data frames. Or, you can wrap output from a prediction function in `posterior::rvar()` and add it to a data frame so long as that output is a matrix with draws as rows.

If you install the tidybayes.rethinking package, models from the rethinking package are also supported.

Extending tidybayes

To include basic support for new models, one need only implement the `tidy_draws()` generic function for that model. Alternatively, objects that support `posterior::as_draws()` or `coda::as.mcmc.list()` will automatically be supported by `tidy_draws()`.

To include support for estimation and prediction, one must either implement the `epred_draws()`, `predicted_draws()`, and `linpred_draws()` functions or their correspond functions from **rstantools**: `rstantools::posterior_epred()`, `rstantools::posterior_predict()`, and `rstantools::posterior_linpred`. If you take the latter approach, you should include `newdata` and `ndraws` arguments that work as documented in `predicted_draws()`.

`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. This function does not parse indices from variable names (e.g. for variable names like `"x[1]"`); see `spread_draws()` or `gather_draws()` for functions that parse variable indices.

Usage

```
tidy_draws(model, ...)

## Default S3 method:
tidy_draws(model, ...)
```

```
## S3 method for class 'draws'
tidy_draws(model, ...)

## S3 method for class 'data.frame'
tidy_draws(model, ...)

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

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

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

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

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

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

## S3 method for class 'CmdStanFit'
tidy_draws(model, ...)

## S3 method for class 'CmdStanMCMC'
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. Tidybayes supports a variety of model objects; for a full list of supported models, see tidybayes-models.
`...`	Further arguments passed to other methods (mostly unused).

Details

This function can be useful for quick glances at models (especially combined with `gather_variables()` and `median_qi()`), and for models with parameters without indices in their names (like `"x[1]"`). `spread_draws()` and `gather_draws()`, which *do* parse variable name indices, call this function internally if their input is not already a tidy data frame.

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

`tidy_draws()` can be applied to a data frame that is already a tidy-format data frame of draws, provided it has one row per draw. In other words, it can be applied to data frames that have the same format it returns, and it will return the same data frame back, while checking to ensure the `.chain`, `.iteration`, and `.draw` columns are all integers (converting if possible) and that the `.draw` column is unique. This allows you to pass already-tidy-format data frames into other tidybayes functions, like `spread_draws()` or `gather_draws()`. This functionality can be useful if the tidying step is expensive: you can tidy once, possibly subsetting to some particular variables of interest, then call `spread_draws()` or `gather_draws()` repeatedly to extract variables and indices from the already-tidied data frame.

Value

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

Author(s)

Matthew Kay

See Also

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

Examples

```
library(magrittr)

data(line, package = "coda")

line %>%
  tidy_draws()
```

---

ungather_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

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

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

Arguments

`data`	A tidy data frame of draws, such as one output by `spread_draws` or `gather_draws`.
`...`	Expressions in the form of `variable_name[dimension_1, dimension_2, ...]`. See `spread_draws()`.
`variable`	The name of the column in `data` that contains the names of variables from the model.
`value`	The name of the column in `data` that contains draws from the variables.
`draw_indices`	Character vector of column names that should be treated as indices of draws. Operations are done within combinations of these values. The default is `c(".chain", ".iteration", ".draw")`, which is the same names used for chain, iteration, and draw indices returned by `tidy_draws()`. Names in `draw_indices` that are not found in the data are ignored.
`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 = "ggdist")

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

Author(s)

Matthew Kay

See Also

`compose_data()`.

Examples

```
library(magrittr)

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

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

Index