Package 'modelbased'

February 5, 2025

Type Package

Title Estimation of Model-Based Predictions, Contrasts and Means

Version 0.9.0

Maintainer Dominique Makowski <dom.makowski@gmail.com>

Description Implements a general interface for model-based estimations for a wide variety of models, used in the computation of marginal means, contrast analysis and predictions. For a list of supported models, see 'insight::supported_models()'.

License GPL-3

URL https://easystats.github.io/modelbased/

BugReports https://github.com/easystats/modelbased/issues

Depends R (>= 3.6)

Imports bayestestR (>= 0.15.1), datawizard (>= 1.0.0), insight (>= 1.0.1), parameters (>= 0.24.1), graphics, stats, tools, utils

Suggests BH, brms, coda, collapse, correlation, curl, easystats, effectsize (>= 1.0.0), emmeans (>= 1.10.2), Formula, gamm4, gganimate, ggplot2, glmmTMB, httr2, knitr, lme4, lmerTest, logspline, MASS, marginaleffects (>= 0.25.0), mgcv, nanoparquet, performance (>= 0.13.0), patchwork, pbkrtest, poorman, RcppEigen, report, rmarkdown, rstanarm, rtdists, see (>= 0.9.0), testthat (>= 3.2.1), vdiffr, withr

VignetteBuilder knitr

Encoding UTF-8

Language en-US

RoxygenNote 7.3.2

Config/testthat/edition 3

Config/testthat/parallel true

Config/Needs/check stan-dev/cmdstanr

Config/Needs/website easystats/easystatstemplate

LazyData true

NeedsCompilation no

Author Dominique Makowski [aut, cre] (<https://orcid.org/0000-0001-5375-9967>),
Daniel Lüdecke [aut] (<https://orcid.org/0000-0002-8895-3206>),
Mattan S. Ben-Shachar [aut] (<https://orcid.org/0000-0002-4287-4801>),
Indrajeet Patil [aut] (<https://orcid.org/0000-0003-1995-6531>)

Repository CRAN

Date/Publication 2025-02-05 11:50:23 UTC

Contents

`coffee_data` *Sample dataset from a course about analysis of factorial designs*

Description

A sample data set from a course about the analysis of factorial designs, by Mattan S. Ben-Shachar. See following link for more information: https://github.com/mattansb/Analysis-of-Factorial-Designs-foR-Psychologists

The data consists of five variables from 120 observations:

- `ID`: A unique identifier for each participant
- `sex`: The participant's sex
- `time`: The time of day the participant was tested (morning, noon, or afternoon)
- `coffee`: Group indicator, whether participant drank coffee or not (`"coffee"` or `"control"`).
- `alertness`: The participant's alertness score.

`describe_nonlinear` *Describe the smooth term (for GAMs) or non-linear predictors*

Description

This function summarises the smooth term trend in terms of linear segments. Using the approximate derivative, it separates a non-linear vector into quasi-linear segments (in which the trend is either positive or negative). Each of this segment its characterized by its beginning, end, size (in proportion, relative to the total size) trend (the linear regression coefficient) and linearity (the R2 of the linear regression).

Usage

```
describe_nonlinear(data, ...)

## S3 method for class 'data.frame'
describe_nonlinear(data, x = NULL, y = NULL, ...)

estimate_smooth(data, ...)
```

Arguments

`data`	The data containing the link, as for instance obtained by `estimate_relation()`.
`...`	Other arguments to be passed to or from.
`x, y`	The name of the responses variable (`y`) predicting variable (`x`).

Value

A data frame of linear description of non-linear terms.

Examples

```
# Create data
data <- data.frame(x = rnorm(200))
data$y <- data$x^2 + rnorm(200, 0, 0.5)

model <<- lm(y ~ poly(x, 2), data = data)
link_data <- estimate_relation(model, length = 100)

describe_nonlinear(link_data, x = "x")
```

efc *Sample dataset from the EFC Survey*

Description

Selected variables from the EUROFAMCARE survey. Useful when testing on "real-life" data sets, including random missing values. This data set also has value and variable label attributes.

estimate_contrasts *Estimate Marginal Contrasts*

Description

Run a contrast analysis by estimating the differences between each level of a factor. See also other related functions such as `estimate_means()` and `estimate_slopes()`.

Usage

```
estimate_contrasts(model, ...)

## Default S3 method:
estimate_contrasts(
  model,
  contrast = NULL,
  by = NULL,
  predict = NULL,
  ci = 0.95,
  comparison = "pairwise",
  estimate = "average",
  p_adjust = "none",
  transform = NULL,
  backend = getOption("modelbased_backend", "marginaleffects"),
  verbose = TRUE,
  ...
)
```

Arguments

`model`	A statistical model.
`...`	Other arguments passed, for instance, to `insight::get_datagrid()`, to functions from the **emmeans** or **marginaleffects** package, or to process Bayesian models via `bayestestR::describe_posterior()`. Examples: • `insight::get_datagrid()`: Argument such as `length` or `range` can be used to control the (number of) representative values.

- **marginaleffects**: Internally used functions are `avg_predictions()` for means and contrasts, and `avg_slope()` for slopes. Therefore, arguments for instance like `vcov`, `transform`, `equivalence`, `slope` or even `newdata` can be passed to those functions.
- **emmeans**: Internally used functions are `emmeans()` and `emtrends()`. Additional arguments can be passed to these functions.
- Bayesian models: For Bayesian models, parameters are cleaned using `describe_posterior()`, thus, arguments like, for example, `centrality`, `rope_range`, or `test` are passed to that function.

`contrast` A character vector indicating the name of the variable(s) for which to compute the contrasts.

`by` The (focal) predictor variable(s) at which to evaluate the desired effect / mean / contrasts. Other predictors of the model that are not included here will be collapsed and "averaged" over (the effect will be estimated across them). The `by` argument is used to create a "reference grid" or "data grid" with representative values for the focal predictors. `by` can be a character (vector) naming the focal predictors (and optionally, representative values or levels), or a list of named elements. See details in `insight::get_datagrid()` to learn more about how to create data grids for predictors of interest.

`predict` Is passed to the `type` argument in `emmeans::emmeans()` (when `backend = "emmeans"`) or in `marginaleffects::avg_predictions()` (when `backend = "marginaleffects"`). For emmeans, see also this vignette. Valid options for 'predict" are:

- `backend = "emmeans"`: `predict` can be `"response"`, `"link"`, `"mu"`, `"unlink"`, or `"log"`. If `predict = NULL` (default), the most appropriate transformation is selected (which usually is `"response"`).
- `backend = "marginaleffects"`: `predict` can be `"response"`, `"link"` or any valid `type` option supported by model's class `predict()` method (e.g., for zero-inflation models from package **glmmTMB**, you can choose `predict = "zprob"` or `predict = "conditional"` etc., see glmmTMB::predict.glmmTMB). By default, when `predict = NULL`, the most appropriate transformation is selected, which usually returns predictions or contrasts on the response-scale.

`"link"` will leave the values on scale of the linear predictors. `"response"` (or `NULL`) will transform them on scale of the response variable. Thus for a logistic model, `"link"` will give estimations expressed in log-odds (probabilities on logit scale) and `"response"` in terms of probabilities. To predict distributional parameters (called "dpar" in other packages), for instance when using complex formulae in `brms` models, the `predict` argument can take the value of the parameter you want to estimate, for instance `"sigma"`, `"kappa"`, etc.

`ci` Confidence Interval (CI) level. Default to `0.95` (95%).

`comparison` Specify the type of contrasts or tests that should be carried out.

- When `backend = "emmeans"`, can be one of `"pairwise"`, `"poly"`, `"consec"`, `"eff"`, `"del.eff"`, `"mean_chg"`, `"trt.vs.ctrl"`, `"dunnett"`, `"wtcon"` and some more. See also `method` argument in emmeans::contrast and the `?emmeans::emmc-functions`.

- For `backend = "marginaleffects"`, can be a numeric value, vector, or matrix, a string equation specifying the hypothesis to test, a string naming the comparison method, a formula, or a function. Strings, string equations and formula are probably the most common options and described below. For other options and detailed descriptions of those options, see also marginaleffects::comparisons and this website.
  - String: One of `"pairwise"`, `"reference"`, `"sequential"`, `"meandev"` `"meanotherdev"`, `"poly"`, `"helmert"`, or `"trt_vs_ctrl"`.
  - String equation: To identify parameters from the output, either specify the term name, or `"b1"`, `"b2"` etc. to indicate rows, e.g.:`"hp = drat"`, `"b1 = b2"`, or `"b1 + b2 + b3 = 0"`.
  - Formula: A formula like `comparison ~ pairs | group`, where the left-hand side indicates the type of comparison (`difference` or `ratio`), the right-hand side determines the pairs of estimates to compare (`reference`, `sequential`, `meandev`, etc., see string-options). Optionally, comparisons can be carried out within subsets by indicating the grouping variable after a vertical bar (`|`).

`estimate` Character string, indicating the type of target population predictions refer to. This dictates how the predictions are "averaged" over the non-focal predictors, i.e. those variables that are not specified in `by` or `contrast`.

- `"average"` (default): Takes the mean value for non-focal numeric predictors and marginalizes over the factor levels of non-focal terms, which computes a kind of "weighted average" for the values at which these terms are hold constant. These predictions are a good representation of the sample, because all possible values and levels of the non-focal predictors are considered. It answers the question, "What is the predicted value for an 'average' observation in *my data*?". Cum grano salis, it refers to randomly picking a subject of your sample and the result you get on average. This approach is the one taken by default in the `emmeans` package.
- `"population"`: Non-focal predictors are marginalized over the observations in the sample, where the sample is replicated multiple times to produce "counterfactuals" and then takes the average of these predicted values (aggregated/grouped by the focal terms). It can be considered as extrapolation to a hypothetical target population. Counterfactual predictions are useful, insofar as the results can also be transferred to other contexts (Dickerman and Hernan, 2020). It answers the question, "What is the predicted response value for the 'average' observation in *the broader target population*?". It does not only refer to the actual data in your observed sample, but also "what would be if" we had more data, or if we had data from a different sample.

In other words, the distinction between estimate types resides in whether the prediction are made for:

- A specific "individual" from the sample (i.e., a specific combination of predictor values): this is what is obtained when using `estimate_relation()` and the other prediction functions.
- An average individual from the sample: obtained with `estimate_means(..., estimate = "average")`

- The broader, hypothetical target population: obtained with `estimate_means(..., estimate = "population")`

`p_adjust` The p-values adjustment method for frequentist multiple comparisons. Can be one of `"none"` (default), `"hochberg"`, `"hommel"`, `"bonferroni"`, `"BH"`, `"BY"`, `"fdr"`, `"tukey"` or `"holm"`. See the p-value adjustment section in the `emmeans::test` documentation or `?stats::p.adjust`.

`transform` A function applied to predictions and confidence intervals to (back-) transform results, which can be useful in case the regression model has a transformed response variable (e.g., `lm(log(y) ~ x)`). For Bayesian models, this function is applied to individual draws from the posterior distribution, before computing summaries. Can also be `TRUE`, in which case `insight::get_transformation()` is called to determine the appropriate transformation-function.

`backend` Whether to use `"emmeans"` or `"marginaleffects"` as a backend. Results are usually very similar. The major difference will be found for mixed models, where `backend = "marginaleffects"` will also average across random effects levels, producing "marginal predictions" (instead of "conditional predictions", see Heiss 2022).

You can set a default backend via `options()`, e.g. use `options(modelbased_backend = "emmeans")` to use the **emmeans** package or `options(modelbased_backend = "marginaleffects")` to set **marginaleffects** as default backend.

`verbose` Use `FALSE` to silence messages and warnings.

Details

The `estimate_slopes()`, `estimate_means()` and `estimate_contrasts()` functions are forming a group, as they are all based on *marginal* estimations (estimations based on a model). All three are built on the **emmeans** or **marginaleffects** package (depending on the `backend` argument), so reading its documentation (for instance `emmeans::emmeans()`, `emmeans::emtrends()` or this website) is recommended to understand the idea behind these types of procedures.

- Model-based **predictions** is the basis for all that follows. Indeed, the first thing to understand is how models can be used to make predictions (see `estimate_link()`). This corresponds to the predicted response (or "outcome variable") given specific predictor values of the predictors (i.e., given a specific data configuration). This is why the concept of `reference grid()` is so important for direct predictions.
- **Marginal "means"**, obtained via `estimate_means()`, are an extension of such predictions, allowing to "average" (collapse) some of the predictors, to obtain the average response value at a specific predictors configuration. This is typically used when some of the predictors of interest are factors. Indeed, the parameters of the model will usually give you the intercept value and then the "effect" of each factor level (how different it is from the intercept). Marginal means can be used to directly give you the mean value of the response variable at all the levels of a factor. Moreover, it can also be used to control, or average over predictors, which is useful in the case of multiple predictors with or without interactions.
- **Marginal contrasts**, obtained via `estimate_contrasts()`, are themselves at extension of marginal means, in that they allow to investigate the difference (i.e., the contrast) between the marginal means. This is, again, often used to get all pairwise differences between all levels of a factor. It works also for continuous predictors, for instance one could also be interested in whether the difference at two extremes of a continuous predictor is significant.

- Finally, **marginal effects**, obtained via `estimate_slopes()`, are different in that their focus is not values on the response variable, but the model's parameters. The idea is to assess the effect of a predictor at a specific configuration of the other predictors. This is relevant in the case of interactions or non-linear relationships, when the effect of a predictor variable changes depending on the other predictors. Moreover, these effects can also be "averaged" over other predictors, to get for instance the "general trend" of a predictor over different factor levels.

Example: Let's imagine the following model `lm(y ~ condition * x)` where `condition` is a factor with 3 levels A, B and C and `x` a continuous variable (like age for example). One idea is to see how this model performs, and compare the actual response y to the one predicted by the model (using `estimate_expectation()`). Another idea is evaluate the average mean at each of the condition's levels (using `estimate_means()`), which can be useful to visualize them. Another possibility is to evaluate the difference between these levels (using `estimate_contrasts()`). Finally, one could also estimate the effect of x averaged over all conditions, or instead within each condition (using `[estimate_slopes]`).

Value

A data frame of estimated contrasts.

Examples

```
## Not run:
# Basic usage
model <- lm(Sepal.Width ~ Species, data = iris)
estimate_contrasts(model)

# Dealing with interactions
model <- lm(Sepal.Width ~ Species * Petal.Width, data = iris)

# By default: selects first factor
estimate_contrasts(model)

# Can also run contrasts between points of numeric, stratified by "Species"
estimate_contrasts(model, contrast = "Petal.Width", by = "Species")

# Or both
estimate_contrasts(model, contrast = c("Species", "Petal.Width"), length = 2)

# Or with custom specifications
estimate_contrasts(model, contrast = c("Species", "Petal.Width=c(1, 2)"))

# Or modulate it
estimate_contrasts(model, by = "Petal.Width", length = 4)

# Standardized differences
estimated <- estimate_contrasts(lm(Sepal.Width ~ Species, data = iris))
standardize(estimated)

# Other models (mixed, Bayesian, ...)
data <- iris
data$Petal.Length_factor <- ifelse(data$Petal.Length < 4.2, "A", "B")
```

```
model <- lme4::lmer(Sepal.Width ~ Species + (1 | Petal.Length_factor), data = data)
estimate_contrasts(model)

data <- mtcars
data$cyl <- as.factor(data$cyl)
data$am <- as.factor(data$am)

model <- rstanarm::stan_glm(mpg ~ cyl * wt, data = data, refresh = 0)
estimate_contrasts(model)
estimate_contrasts(model, by = "wt", length = 4)

model <- rstanarm::stan_glm(
  Sepal.Width ~ Species + Petal.Width + Petal.Length,
  data = iris,
  refresh = 0
)
estimate_contrasts(model, by = "Petal.Length = [sd]", test = "bf")

## End(Not run)
```

`estimate_expectation` *Model-based predictions*

Description

After fitting a model, it is useful generate model-based estimates of the response variables for different combinations of predictor values. Such estimates can be used to make inferences about **relationships** between variables, to make predictions about individual cases, or to compare the **predicted** values against the observed data.

The `modelbased` package includes 4 "related" functions, that mostly differ in their default arguments (in particular, `data` and `predict`):

- `estimate_prediction(data = NULL, predict = "prediction", ...)`
- `estimate_expectation(data = NULL, predict = "expectation", ...)`
- `estimate_relation(data = "grid", predict = "expectation", ...)`
- `estimate_link(data = "grid", predict = "link", ...)`

While they are all based on model-based predictions (using `insight::get_predicted()`), they differ in terms of the **type** of predictions they make by default. For instance, `estimate_prediction()` and `estimate_expectation()` return predictions for the original data used to fit the model, while `estimate_relation()` and `estimate_link()` return predictions on a `insight::get_datagrid()`. Similarly, `estimate_link` returns predictions on the link scale, while the others return predictions on the response scale. Note that the relevance of these differences depends on the model family (for instance, for linear models, `estimate_relation` is equivalent to `estimate_link()`, since there is no difference between the link-scale and the response scale).

Note that you can run `plot()` on the output of these functions to get some visual insights (see the plotting examples).

See the **details** section below for details about the different possibilities.

Usage

```
estimate_expectation(
  model,
  data = NULL,
  by = NULL,
  predict = "expectation",
  ci = 0.95,
  transform = NULL,
  keep_iterations = FALSE,
  ...
)

estimate_link(
  model,
  data = "grid",
  by = NULL,
  predict = "link",
  ci = 0.95,
  transform = NULL,
  keep_iterations = FALSE,
  ...
)

estimate_prediction(
  model,
  data = NULL,
  by = NULL,
  predict = "prediction",
  ci = 0.95,
  transform = NULL,
  keep_iterations = FALSE,
  ...
)

estimate_relation(
  model,
  data = "grid",
  by = NULL,
  predict = "expectation",
  ci = 0.95,
  transform = NULL,
  keep_iterations = FALSE,
  ...
```

)

Arguments

model	A statistical model.
data	A data frame with model's predictors to estimate the response. If NULL, the model's data is used. If "grid", the model matrix is obtained (through insight::get_datagrid()).
by	The predictor variable(s) at which to estimate the response. Other predictors of the model that are not included here will be set to their mean value (for numeric predictors), reference level (for factors) or mode (other types). The by argument will be used to create a data grid via insight::get_datagrid(), which will then be used as data argument. Thus, you cannot specify both data and by but only of these two arguments.
predict	This parameter controls what is predicted (and gets internally passed to insight::get_predicted()). In most cases, you don't need to care about it: it is changed automatically according to the different predicting functions (i.e., estimate_expectation(), estimate_prediction(), estimate_link() or estimate_relation()). The only time you might be interested in manually changing it is to estimate other distributional parameters (called "dpar" in other packages) - for instance when using complex formulae in brms models. The predict argument can then be set to the parameter you want to estimate, for instance "sigma", "kappa", etc. Note that the distinction between "expectation", "link" and "prediction" does not then apply (as you are directly predicting the value of some distributional parameter), and the corresponding functions will then only differ in the default value of their data argument.
ci	Confidence Interval (CI) level. Default to 0.95 (95%).
transform	A function applied to predictions and confidence intervals to (back-) transform results, which can be useful in case the regression model has a transformed response variable (e.g., lm(log(y) ~ x)). Can also be TRUE, in which case insight::get_transformation() is called to determine the appropriate transformation-function. **Note:** Standard errors are not (back-) transformed!
keep_iterations	If TRUE, will keep all iterations (draws) of bootstrapped or Bayesian models. They will be added as additional columns named iter_1, iter_2, ... You can reshape them to a long format by running reshape_iterations().
...	You can add all the additional control arguments from insight::get_datagrid() (used when data = "grid") and insight::get_predicted().

Value

A data frame of predicted values and uncertainty intervals, with class "estimate_predicted". Methods for visualisation_recipe() and plot() are available.

Expected (average) values

The most important way that various types of response estimates differ is in terms of what quantity is being estimated and the meaning of the uncertainty intervals. The major choices are **expected**

values for uncertainty in the regression line and **predicted values** for uncertainty in the individual case predictions.

Expected values refer to the fitted regression line - the estimated *average* response value (i.e., the "expectation") for individuals with specific predictor values. For example, in a linear model $y = 2 + 3x + 4z + e$, the estimated average y for individuals with $x = 1$ and $z = 2$ is 11.

For expected values, uncertainty intervals refer to uncertainty in the estimated **conditional average** (where might the true regression line actually fall)? Uncertainty intervals for expected values are also called "confidence intervals".

Expected values and their uncertainty intervals are useful for describing the relationship between variables and for describing how precisely a model has been estimated.

For generalized linear models, expected values are reported on one of two scales:

- The **link scale** refers to scale of the fitted regression line, after transformation by the link function. For example, for a logistic regression (logit binomial) model, the link scale gives expected log-odds. For a log-link Poisson model, the link scale gives the expected log-count.
- The **response scale** refers to the original scale of the response variable (i.e., without any link function transformation). Expected values on the link scale are back-transformed to the original response variable metric (e.g., expected probabilities for binomial models, expected counts for Poisson models).

Individual case predictions

In contrast to expected values, **predicted values** refer to predictions for **individual cases**. Predicted values are also called "posterior predictions" or "posterior predictive draws".

For predicted values, uncertainty intervals refer to uncertainty in the **individual response values for each case** (where might any single case actually fall)? Uncertainty intervals for predicted values are also called "prediction intervals" or "posterior predictive intervals".

Predicted values and their uncertainty intervals are useful for forecasting the range of values that might be observed in new data, for making decisions about individual cases, and for checking if model predictions are reasonable ("posterior predictive checks").

Predicted values and intervals are always on the scale of the original response variable (not the link scale).

Functions for estimating predicted values and uncertainty

modelbased provides 4 functions for generating model-based response estimates and their uncertainty:

- `estimate_expectation()`:
  - Generates **expected values** (conditional average) on the **response scale**.
  - The uncertainty interval is a *confidence interval*.
  - By default, values are computed using the data used to fit the model.
- `estimate_link()`:
  - Generates **expected values** (conditional average) on the **link scale**.
  - The uncertainty interval is a *confidence interval*.

  - By default, values are computed using a reference grid spanning the observed range of predictor values (see `insight::get_datagrid()`).

- `estimate_prediction()`:
  - Generates **predicted values** (for individual cases) on the **response scale**.
  - The uncertainty interval is a *prediction interval*.
  - By default, values are computed using the data used to fit the model.
- `estimate_relation()`:
  - Like `estimate_expectation()`.
  - Useful for visualizing a model.
  - Generates **expected values** (conditional average) on the **response scale**.
  - The uncertainty interval is a *confidence interval*.
  - By default, values are computed using a reference grid spanning the observed range of predictor values (see `insight::get_datagrid()`).

Data for predictions

If the `data = NULL`, values are estimated using the data used to fit the model. If `data = "grid"`, values are computed using a reference grid spanning the observed range of predictor values with `insight::get_datagrid()`. This can be useful for model visualization. The number of predictor values used for each variable can be controlled with the `length` argument. `data` can also be a data frame containing columns with names matching the model frame (see `insight::get_data()`). This can be used to generate model predictions for specific combinations of predictor values.

Note

These functions are built on top of `insight::get_predicted()` and correspond to different specifications of its parameters. It may be useful to read its documentation, in particular the description of the `predict` argument for additional details on the difference between expected vs. predicted values and link vs. response scales.

Additional control parameters can be used to control results from `insight::get_datagrid()` (when `data = "grid"`) and from `insight::get_predicted()` (the function used internally to compute predictions).

For plotting, check the examples in `visualisation_recipe()`. Also check out the Vignettes and README examples for various examples, tutorials and usecases.

Examples

```
library(modelbased)

# Linear Models
model <- lm(mpg ~ wt, data = mtcars)

# Get predicted and prediction interval (see insight::get_predicted)
estimate_expectation(model)

# Get expected values with confidence interval
pred <- estimate_relation(model)
```

```
pred

# Visualisation (see visualisation_recipe())
plot(pred)

# Standardize predictions
pred <- estimate_relation(lm(mpg ~ wt + am, data = mtcars))
z <- standardize(pred, include_response = FALSE)
z
unstandardize(z, include_response = FALSE)

# Logistic Models
model <- glm(vs ~ wt, data = mtcars, family = "binomial")
estimate_expectation(model)
estimate_relation(model)

# Mixed models
model <- lme4::lmer(mpg ~ wt + (1 | gear), data = mtcars)
estimate_expectation(model)
estimate_relation(model)

# Bayesian models

model <- suppressWarnings(rstanarm::stan_glm(
  mpg ~ wt,
  data = mtcars, refresh = 0, iter = 200
))
estimate_expectation(model)
estimate_relation(model)
```

`estimate_grouplevel` *Group-specific parameters of mixed models random effects*

Description

Extract random parameters of each individual group in the context of mixed models. Can be reshaped to be of the same dimensions as the original data, which can be useful to add the random effects to the original data.

Usage

```
estimate_grouplevel(model, type = "random", ...)

reshape_grouplevel(x, indices = "all", group = "all", ...)
```

Arguments

model	A mixed model with random effects.
type	If "random" (default), the coefficients are the ones estimated natively by the model (as they are returned by, for instance, lme4::ranef()). They correspond to the deviation of each individual group from their fixed effect. As such, a coefficient close to 0 means that the participants' effect is the same as the population-level effect (in other words, it is "in the norm"). If "total", it will return the sum of the random effect and its corresponding fixed effects. These are known as BLUPs (Best Linear Unbiased Predictions). This argument can be used to reproduce the results given by lme4::ranef() and coef() (see ?coef.merMod). Note that BLUPs currently don't have uncertainty indices (such as SE and CI), as these are not computable.
...	Other arguments passed to or from other methods.
x	The output of estimate_grouplevel().
indices	A list containing the indices to extract (e.g., "Coefficient").
group	A list containing the random factors to select.

Examples

```
# lme4 model
data(mtcars)
model <- lme4::lmer(mpg ~ hp + (1 | carb), data = mtcars)
random <- estimate_grouplevel(model)
random

# Visualize random effects
plot(random)

# Show group-specific effects
estimate_grouplevel(model, deviation = FALSE)

# Reshape to wide data so that it matches the original dataframe...
reshaped <- reshape_grouplevel(random, indices = c("Coefficient", "SE"))

# ... and can be easily combined
alldata <- cbind(mtcars, reshaped)

# Use summary() to remove duplicated rows
summary(reshaped)

# Compute BLUPs
estimate_grouplevel(model, type = "total")
```

`estimate_means` *Estimate Marginal Means (Model-based average at each factor level)*

Description

Estimate average value of response variable at each factor level or representative value, respectively at values defined in a "data grid" or "reference grid". For plotting, check the examples in `visualisation_recipe()`. See also other related functions such as `estimate_contrasts()` and `estimate_slopes()`.

Usage

```
estimate_means(
  model,
  by = "auto",
  predict = NULL,
  ci = 0.95,
  estimate = "average",
  transform = NULL,
  backend = getOption("modelbased_backend", "marginaleffects"),
  verbose = TRUE,
  ...
)
```

Arguments

`model`	A statistical model.
`by`	The (focal) predictor variable(s) at which to evaluate the desired effect / mean / contrasts. Other predictors of the model that are not included here will be collapsed and "averaged" over (the effect will be estimated across them). The by argument is used to create a "reference grid" or "data grid" with representative values for the focal predictors. by can be a character (vector) naming the focal predictors (and optionally, representative values or levels), or a list of named elements. See details in `insight::get_datagrid()` to learn more about how to create data grids for predictors of interest.
`predict`	Is passed to the type argument in `emmeans::emmeans()` (when `backend = "emmeans"`) or in `marginaleffects::avg_predictions()` (when `backend = "marginaleffects"`). For emmeans, see also this vignette. Valid options for 'predict" are: • `backend = "emmeans"`: `predict` can be `"response"`, `"link"`, `"mu"`, `"unlink"`, or `"log"`. If `predict = NULL` (default), the most appropriate transformation is selected (which usually is `"response"`). • `backend = "marginaleffects"`: `predict` can be `"response"`, `"link"` or any valid `type` option supported by model's class `predict()` method (e.g., for zero-inflation models from package **glmmTMB**, you can choose `predict = "zprob"` or `predict = "conditional"` etc., see glmmTMB::predict.glmmTMB). By default, when `predict = NULL`, the most appropriate transformation is

selected, which usually returns predictions or contrasts on the response-scale.

"link" will leave the values on scale of the linear predictors. "response" (or NULL) will transform them on scale of the response variable. Thus for a logistic model, "link" will give estimations expressed in log-odds (probabilities on logit scale) and "response" in terms of probabilities. To predict distributional parameters (called "dpar" in other packages), for instance when using complex formulae in brms models, the predict argument can take the value of the parameter you want to estimate, for instance "sigma", "kappa", etc.

ci — Confidence Interval (CI) level. Default to 0.95 (95%).

estimate — Character string, indicating the type of target population predictions refer to. This dictates how the predictions are "averaged" over the non-focal predictors, i.e. those variables that are not specified in by or contrast.

- "average" (default): Takes the mean value for non-focal numeric predictors and marginalizes over the factor levels of non-focal terms, which computes a kind of "weighted average" for the values at which these terms are hold constant. These predictions are a good representation of the sample, because all possible values and levels of the non-focal predictors are considered. It answers the question, "What is the predicted value for an 'average' observation in *my data*?". Cum grano salis, it refers to randomly picking a subject of your sample and the result you get on average. This approach is the one taken by default in the emmeans package.
- "population": Non-focal predictors are marginalized over the observations in the sample, where the sample is replicated multiple times to produce "counterfactuals" and then takes the average of these predicted values (aggregated/grouped by the focal terms). It can be considered as extrapolation to a hypothetical target population. Counterfactual predictions are useful, insofar as the results can also be transferred to other contexts (Dickerman and Hernan, 2020). It answers the question, "What is the predicted response value for the 'average' observation in *the broader target population*?". It does not only refer to the actual data in your observed sample, but also "what would be if" we had more data, or if we had data from a different sample.

In other words, the distinction between estimate types resides in whether the prediction are made for:

- A specific "individual" from the sample (i.e., a specific combination of predictor values): this is what is obtained when using estimate_relation() and the other prediction functions.
- An average individual from the sample: obtained with estimate_means(..., estimate = "average")
- The broader, hypothetical target population: obtained with estimate_means(..., estimate = "population")

transform — A function applied to predictions and confidence intervals to (back-) transform results, which can be useful in case the regression model has a transformed response variable (e.g., lm(log(y) ~ x)). For Bayesian models, this function is applied to individual draws from the posterior distribution, before computing

summaries. Can also be TRUE, in which case insight::get_transformation() is called to determine the appropriate transformation-function.

backend — Whether to use "emmeans" or "marginaleffects" as a backend. Results are usually very similar. The major difference will be found for mixed models, where backend = "marginaleffects" will also average across random effects levels, producing "marginal predictions" (instead of "conditional predictions", see Heiss 2022).

You can set a default backend via options(), e.g. use options(modelbased_backend = "emmeans") to use the **emmeans** package or options(modelbased_backend = "marginaleffects") to set **marginaleffects** as default backend.

verbose — Use FALSE to silence messages and warnings.

... — Other arguments passed, for instance, to insight::get_datagrid(), to functions from the **emmeans** or **marginaleffects** package, or to process Bayesian models via bayestestR::describe_posterior(). Examples:

- insight::get_datagrid(): Argument such as length or range can be used to control the (number of) representative values.
- **marginaleffects**: Internally used functions are avg_predictions() for means and contrasts, and avg_slope() for slopes. Therefore, arguments for instance like vcov, transform, equivalence, slope or even newdata can be passed to those functions.
- **emmeans**: Internally used functions are emmeans() and emtrends(). Additional arguments can be passed to these functions.
- Bayesian models: For Bayesian models, parameters are cleaned using describe_posterior(), thus, arguments like, for example, centrality, rope_range, or test are passed to that function.

Details

The estimate_slopes(), estimate_means() and estimate_contrasts() functions are forming a group, as they are all based on *marginal* estimations (estimations based on a model). All three are built on the **emmeans** or **marginaleffects** package (depending on the backend argument), so reading its documentation (for instance emmeans::emmeans(), emmeans::emtrends() or this website) is recommended to understand the idea behind these types of procedures.

- Model-based **predictions** is the basis for all that follows. Indeed, the first thing to understand is how models can be used to make predictions (see estimate_link()). This corresponds to the predicted response (or "outcome variable") given specific predictor values of the predictors (i.e., given a specific data configuration). This is why the concept of reference grid() is so important for direct predictions.
- **Marginal "means"**, obtained via estimate_means(), are an extension of such predictions, allowing to "average" (collapse) some of the predictors, to obtain the average response value at a specific predictors configuration. This is typically used when some of the predictors of interest are factors. Indeed, the parameters of the model will usually give you the intercept value and then the "effect" of each factor level (how different it is from the intercept). Marginal means can be used to directly give you the mean value of the response variable at all the levels of a factor. Moreover, it can also be used to control, or average over predictors, which is useful in the case of multiple predictors with or without interactions.

- **Marginal contrasts**, obtained via `estimate_contrasts()`, are themselves at extension of marginal means, in that they allow to investigate the difference (i.e., the contrast) between the marginal means. This is, again, often used to get all pairwise differences between all levels of a factor. It works also for continuous predictors, for instance one could also be interested in whether the difference at two extremes of a continuous predictor is significant.
- Finally, **marginal effects**, obtained via `estimate_slopes()`, are different in that their focus is not values on the response variable, but the model's parameters. The idea is to assess the effect of a predictor at a specific configuration of the other predictors. This is relevant in the case of interactions or non-linear relationships, when the effect of a predictor variable changes depending on the other predictors. Moreover, these effects can also be "averaged" over other predictors, to get for instance the "general trend" of a predictor over different factor levels.

Example: Let's imagine the following model `lm(y ~ condition * x)` where `condition` is a factor with 3 levels A, B and C and `x` a continuous variable (like age for example). One idea is to see how this model performs, and compare the actual response y to the one predicted by the model (using `estimate_expectation()`). Another idea is evaluate the average mean at each of the condition's levels (using `estimate_means()`), which can be useful to visualize them. Another possibility is to evaluate the difference between these levels (using `estimate_contrasts()`). Finally, one could also estimate the effect of x averaged over all conditions, or instead within each condition (`using [estimate_slopes]`).

Value

A data frame of estimated marginal means.

References

Dickerman, Barbra A., and Miguel A. Hernán. 2020. Counterfactual Prediction Is Not Only for Causal Inference. European Journal of Epidemiology 35 (7): 615–17. doi:10.1007/s10654020-006598

Heiss, A. (2022). Marginal and conditional effects for GLMMs with marginaleffects. Andrew Heiss. doi:10.59350/xwnfmx1827

Examples

```
library(modelbased)

# Frequentist models
# -------------------
model <- lm(Petal.Length ~ Sepal.Width * Species, data = iris)

estimate_means(model)

# the `length` argument is passed to `insight::get_datagrid()` and modulates
# the number of representative values to return for numeric predictors
estimate_means(model, by = c("Species", "Sepal.Width"), length = 2)

# an alternative way to setup your data grid is specify the values directly
estimate_means(model, by = c("Species", "Sepal.Width = c(2, 4)"))
```

```
# or use one of the many predefined "tokens" that help you creating a useful
# data grid - to learn more about creating data grids, see help in
# `?insight::get_datagrid`.
estimate_means(model, by = c("Species", "Sepal.Width = [fivenum]"))

## Not run:
# same for factors: filter by specific levels
estimate_means(model, by = "Species=c('versicolor', 'setosa')")
estimate_means(model, by = c("Species", "Sepal.Width=0"))

# estimate marginal average of response at values for numeric predictor
estimate_means(model, by = "Sepal.Width", length = 5)
estimate_means(model, by = "Sepal.Width=c(2, 4)")

# or provide the definition of the data grid as list
estimate_means(
  model,
  by = list(Sepal.Width = c(2, 4), Species = c("versicolor", "setosa"))
)

# Methods that can be applied to it:
means <- estimate_means(model, by = c("Species", "Sepal.Width=0"))

plot(means) # which runs visualisation_recipe()
standardize(means)

data <- iris
data$Petal.Length_factor <- ifelse(data$Petal.Length < 4.2, "A", "B")

model <- lme4::lmer(
  Petal.Length ~ Sepal.Width + Species + (1 | Petal.Length_factor),
  data = data
)
estimate_means(model)
estimate_means(model, by = "Sepal.Width", length = 3)

## End(Not run)
```

`estimate_slopes` *Estimate Marginal Effects*

Description

Estimate the slopes (i.e., the coefficient) of a predictor over or within different factor levels, or alongside a numeric variable. In other words, to assess the effect of a predictor *at* specific configurations data. It corresponds to the derivative and can be useful to understand where a predictor has a significant role when interactions or non-linear relationships are present.

Other related functions based on marginal estimations includes `estimate_contrasts()` and `estimate_means()`.

See the **Details** section below, and don't forget to also check out the Vignettes and README examples for various examples, tutorials and use cases.

Usage

```
estimate_slopes(
  model,
  trend = NULL,
  by = NULL,
  ci = 0.95,
  backend = getOption("modelbased_backend", "marginaleffects"),
  verbose = TRUE,
  ...
)
```

Arguments

model	A statistical model.
trend	A character indicating the name of the variable for which to compute the slopes.
by	The (focal) predictor variable(s) at which to evaluate the desired effect / mean / contrasts. Other predictors of the model that are not included here will be collapsed and "averaged" over (the effect will be estimated across them). The by argument is used to create a "reference grid" or "data grid" with representative values for the focal predictors. by can be a character (vector) naming the focal predictors (and optionally, representative values or levels), or a list of named elements. See details in insight::get_datagrid() to learn more about how to create data grids for predictors of interest.
ci	Confidence Interval (CI) level. Default to 0.95 (95%).
backend	Whether to use "emmeans" or "marginaleffects" as a backend. Results are usually very similar. The major difference will be found for mixed models, where backend = "marginaleffects" will also average across random effects levels, producing "marginal predictions" (instead of "conditional predictions", see Heiss 2022). You can set a default backend via options(), e.g. use options(modelbased_backend = "emmeans") to use the **emmeans** package or options(modelbased_backend = "marginaleffects") to set **marginaleffects** as default backend.
verbose	Use FALSE to silence messages and warnings.
...	Other arguments passed, for instance, to insight::get_datagrid(), to functions from the **emmeans** or **marginaleffects** package, or to process Bayesian models via bayestestR::describe_posterior(). Examples: • insight::get_datagrid(): Argument such as length or range can be used to control the (number of) representative values. • **marginaleffects**: Internally used functions are avg_predictions() for means and contrasts, and avg_slope() for slopes. Therefore, arguments for instance like vcov, transform, equivalence, slope or even newdata can be passed to those functions.

- **emmeans**: Internally used functions are `emmeans()` and `emtrends()`. Additional arguments can be passed to these functions.
- Bayesian models: For Bayesian models, parameters are cleaned using `describe_posterior()`, thus, arguments like, for example, `centrality`, `rope_range`, or `test` are passed to that function.

Details

The `estimate_slopes()`, `estimate_means()` and `estimate_contrasts()` functions are forming a group, as they are all based on *marginal* estimations (estimations based on a model). All three are built on the **emmeans** or **marginaleffects** package (depending on the `backend` argument), so reading its documentation (for instance `emmeans::emmeans()`, `emmeans::emtrends()` or this website) is recommended to understand the idea behind these types of procedures.

- Model-based **predictions** is the basis for all that follows. Indeed, the first thing to understand is how models can be used to make predictions (see `estimate_link()`). This corresponds to the predicted response (or "outcome variable") given specific predictor values of the predictors (i.e., given a specific data configuration). This is why the concept of `reference grid()` is so important for direct predictions.
- **Marginal "means"**, obtained via `estimate_means()`, are an extension of such predictions, allowing to "average" (collapse) some of the predictors, to obtain the average response value at a specific predictors configuration. This is typically used when some of the predictors of interest are factors. Indeed, the parameters of the model will usually give you the intercept value and then the "effect" of each factor level (how different it is from the intercept). Marginal means can be used to directly give you the mean value of the response variable at all the levels of a factor. Moreover, it can also be used to control, or average over predictors, which is useful in the case of multiple predictors with or without interactions.
- **Marginal contrasts**, obtained via `estimate_contrasts()`, are themselves at extension of marginal means, in that they allow to investigate the difference (i.e., the contrast) between the marginal means. This is, again, often used to get all pairwise differences between all levels of a factor. It works also for continuous predictors, for instance one could also be interested in whether the difference at two extremes of a continuous predictor is significant.
- Finally, **marginal effects**, obtained via `estimate_slopes()`, are different in that their focus is not values on the response variable, but the model's parameters. The idea is to assess the effect of a predictor at a specific configuration of the other predictors. This is relevant in the case of interactions or non-linear relationships, when the effect of a predictor variable changes depending on the other predictors. Moreover, these effects can also be "averaged" over other predictors, to get for instance the "general trend" of a predictor over different factor levels.

Example: Let's imagine the following model `lm(y ~ condition * x)` where `condition` is a factor with 3 levels A, B and C and `x` a continuous variable (like age for example). One idea is to see how this model performs, and compare the actual response `y` to the one predicted by the model (using `estimate_expectation()`). Another idea is evaluate the average mean at each of the condition's levels (using `estimate_means()`), which can be useful to visualize them. Another possibility is to evaluate the difference between these levels (using `estimate_contrasts()`). Finally, one could also estimate the effect of `x` averaged over all conditions, or instead within each condition (`using [estimate_slopes]`).

Value

A data.frame of class `estimate_slopes`.

Examples

```
library(ggplot2)
# Get an idea of the data
ggplot(iris, aes(x = Petal.Length, y = Sepal.Width)) +
  geom_point(aes(color = Species)) +
  geom_smooth(color = "black", se = FALSE) +
  geom_smooth(aes(color = Species), linetype = "dotted", se = FALSE) +
  geom_smooth(aes(color = Species), method = "lm", se = FALSE)

# Model it
model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)
# Compute the marginal effect of Petal.Length at each level of Species
slopes <- estimate_slopes(model, trend = "Petal.Length", by = "Species")
slopes

## Not run:
# Plot it
plot(slopes)
standardize(slopes)

model <- mgcv::gam(Sepal.Width ~ s(Petal.Length), data = iris)
slopes <- estimate_slopes(model, by = "Petal.Length", length = 50)
summary(slopes)
plot(slopes)

model <- mgcv::gam(Sepal.Width ~ s(Petal.Length, by = Species), data = iris)
slopes <- estimate_slopes(model,
  trend = "Petal.Length",
  by = c("Petal.Length", "Species"), length = 20
)
summary(slopes)
plot(slopes)

## End(Not run)
```

`fish` *Sample data set*

Description

A sample data set, used in tests and some examples. Useful for demonstrating count models (with or without zero-inflation component). It consists of nine variables from 250 observations.

get_emcontrasts *Consistent API for 'emmeans' and 'marginaleffects'*

Description

These functions are convenient wrappers around the **emmeans** and the **marginaleffects** packages. They are mostly available for developers who want to leverage a unified API for getting model-based estimates, and regular users should use the `estimate_*` set of functions.

The `get_emmeans()`, `get_emcontrasts()` and `get_emtrends()` functions are wrappers around `emmeans::emmeans()` and `emmeans::emtrends()`.

Usage

```
get_emcontrasts(
  model,
  contrast = NULL,
  by = NULL,
  predict = NULL,
  comparison = "pairwise",
  transform = NULL,
  verbose = TRUE,
  ...
)

get_emmeans(
  model,
  by = "auto",
  predict = NULL,
  transform = NULL,
  verbose = TRUE,
  ...
)

get_emtrends(model, trend = NULL, by = NULL, verbose = TRUE, ...)

get_marginalcontrasts(
  model,
  contrast = NULL,
  by = NULL,
  predict = NULL,
  ci = 0.95,
  comparison = "pairwise",
  estimate = "average",
  p_adjust = "none",
  transform = NULL,
  verbose = TRUE,
```

```
  ...
)

get_marginalmeans(
  model,
  by = "auto",
  predict = NULL,
  ci = 0.95,
  estimate = "average",
  transform = NULL,
  verbose = TRUE,
  ...
)

get_marginaltrends(model, trend = NULL, by = NULL, verbose = TRUE, ...)
```

Arguments

model	A statistical model.
contrast	A character vector indicating the name of the variable(s) for which to compute the contrasts.
by	The (focal) predictor variable(s) at which to evaluate the desired effect / mean / contrasts. Other predictors of the model that are not included here will be collapsed and "averaged" over (the effect will be estimated across them). The by argument is used to create a "reference grid" or "data grid" with representative values for the focal predictors. by can be a character (vector) naming the focal predictors (and optionally, representative values or levels), or a list of named elements. See details in `insight::get_datagrid()` to learn more about how to create data grids for predictors of interest.
predict	Is passed to the type argument in `emmeans::emmeans()` (when backend = "emmeans") or in `marginaleffects::avg_predictions()` (when backend = "marginaleffects"). For emmeans, see also this vignette. Valid options for 'predict" are: • backend = "emmeans": predict can be "response", "link", "mu", "unlink", or "log". If predict = NULL (default), the most appropriate transformation is selected (which usually is "response"). • backend = "marginaleffects": predict can be "response", "link" or any valid type option supported by model's class predict() method (e.g., for zero-inflation models from package **glmmTMB**, you can choose predict = "zprob" or predict = "conditional" etc., see glmmTMB::predict.glmmTMB). By default, when predict = NULL, the most appropriate transformation is selected, which usually returns predictions or contrasts on the response-scale. "link" will leave the values on scale of the linear predictors. "response" (or NULL) will transform them on scale of the response variable. Thus for a logistic model, "link" will give estimations expressed in log-odds (probabilities on logit scale) and "response" in terms of probabilities. To predict distributional parameters (called "dpar" in other packages), for instance when using complex

formulae in brms models, the predict argument can take the value of the parameter you want to estimate, for instance "sigma", "kappa", etc.

comparison — Specify the type of contrasts or tests that should be carried out.

- When backend = "emmeans", can be one of "pairwise", "poly", "consec", "eff", "del.eff", "mean_chg", "trt.vs.ctrl", "dunnett", "wtcon" and some more. See also method argument in emmeans::contrast and the ?emmeans::emmc-functions.
- For backend = "marginaleffects", can be a numeric value, vector, or matrix, a string equation specifying the hypothesis to test, a string naming the comparison method, a formula, or a function. Strings, string equations and formula are probably the most common options and described below. For other options and detailed descriptions of those options, see also marginaleffects::comparisons and this website.
  - String: One of "pairwise", "reference", "sequential", "meandev" "meanotherdev", "poly", "helmert", or "trt_vs_ctrl".
  - String equation: To identify parameters from the output, either specify the term name, or "b1", "b2" etc. to indicate rows, e.g.:"hp = drat", "b1 = b2", or "b1 + b2 + b3 = 0".
  - Formula: A formula like comparison ~ pairs | group, where the left-hand side indicates the type of comparison (difference or ratio), the right-hand side determines the pairs of estimates to compare (reference, sequential, meandev, etc., see string-options). Optionally, comparisons can be carried out within subsets by indicating the grouping variable after a vertical bar (|).

transform — A function applied to predictions and confidence intervals to (back-) transform results, which can be useful in case the regression model has a transformed response variable (e.g., lm(log(y) ~ x)). For Bayesian models, this function is applied to individual draws from the posterior distribution, before computing summaries. Can also be TRUE, in which case insight::get_transformation() is called to determine the appropriate transformation-function.

verbose — Use FALSE to silence messages and warnings.

... — Other arguments passed, for instance, to insight::get_datagrid(), to functions from the **emmeans** or **marginaleffects** package, or to process Bayesian models via bayestestR::describe_posterior(). Examples:

- insight::get_datagrid(): Argument such as length or range can be used to control the (number of) representative values.
- **marginaleffects**: Internally used functions are avg_predictions() for means and contrasts, and avg_slope() for slopes. Therefore, arguments for instance like vcov, transform, equivalence, slope or even newdata can be passed to those functions.
- **emmeans**: Internally used functions are emmeans() and emtrends(). Additional arguments can be passed to these functions.
- Bayesian models: For Bayesian models, parameters are cleaned using describe_posterior(), thus, arguments like, for example, centrality, rope_range, or test are passed to that function.

trend — A character indicating the name of the variable for which to compute the slopes.

ci — Confidence Interval (CI) level. Default to 0.95 (95%).

estimate — Character string, indicating the type of target population predictions refer to. This dictates how the predictions are "averaged" over the non-focal predictors, i.e. those variables that are not specified in by or contrast.

- "average" (default): Takes the mean value for non-focal numeric predictors and marginalizes over the factor levels of non-focal terms, which computes a kind of "weighted average" for the values at which these terms are hold constant. These predictions are a good representation of the sample, because all possible values and levels of the non-focal predictors are considered. It answers the question, "What is the predicted value for an 'average' observation in *my data*?". Cum grano salis, it refers to randomly picking a subject of your sample and the result you get on average. This approach is the one taken by default in the emmeans package.
- "population": Non-focal predictors are marginalized over the observations in the sample, where the sample is replicated multiple times to produce "counterfactuals" and then takes the average of these predicted values (aggregated/grouped by the focal terms). It can be considered as extrapolation to a hypothetical target population. Counterfactual predictions are useful, insofar as the results can also be transferred to other contexts (Dickerman and Hernan, 2020). It answers the question, "What is the predicted response value for the 'average' observation in *the broader target population*?". It does not only refer to the actual data in your observed sample, but also "what would be if" we had more data, or if we had data from a different sample.

In other words, the distinction between estimate types resides in whether the prediction are made for:

- A specific "individual" from the sample (i.e., a specific combination of predictor values): this is what is obtained when using estimate_relation() and the other prediction functions.
- An average individual from the sample: obtained with estimate_means(..., estimate = "average")
- The broader, hypothetical target population: obtained with estimate_means(..., estimate = "population")

p_adjust — The p-values adjustment method for frequentist multiple comparisons. Can be one of "none" (default), "hochberg", "hommel", "bonferroni", "BH", "BY", "fdr", "tukey" or "holm". See the p-value adjustment section in the emmeans::test documentation or ?stats::p.adjust.

Examples

```
# Basic usage
model <- lm(Sepal.Width ~ Species, data = iris)
get_emcontrasts(model)

## Not run:
# Dealing with interactions
model <- lm(Sepal.Width ~ Species * Petal.Width, data = iris)
```

```
# By default: selects first factor
get_emcontrasts(model)
# Can also run contrasts between points of numeric
get_emcontrasts(model, contrast = "Petal.Width", length = 3)
# Or both
get_emcontrasts(model, contrast = c("Species", "Petal.Width"), length = 2)
# Or with custom specifications
estimate_contrasts(model, contrast = c("Species", "Petal.Width=c(1, 2)"))
# Or modulate it
get_emcontrasts(model, by = "Petal.Width", length = 4)

## End(Not run)


model <- lm(Sepal.Length ~ Species + Petal.Width, data = iris)

# By default, 'by' is set to "Species"
get_emmeans(model)

## Not run:
# Overall mean (close to 'mean(iris$Sepal.Length)')
get_emmeans(model, by = NULL)

# One can estimate marginal means at several values of a 'modulate' variable
get_emmeans(model, by = "Petal.Width", length = 3)

# Interactions
model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)

get_emmeans(model)
get_emmeans(model, by = c("Species", "Petal.Length"), length = 2)
get_emmeans(model, by = c("Species", "Petal.Length = c(1, 3, 5)"), length = 2)

## End(Not run)


## Not run:
model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)

get_emtrends(model)
get_emtrends(model, by = "Species")
get_emtrends(model, by = "Petal.Length")
get_emtrends(model, by = c("Species", "Petal.Length"))

## End(Not run)

model <- lm(Petal.Length ~ poly(Sepal.Width, 4), data = iris)
get_emtrends(model)
get_emtrends(model, by = "Sepal.Width")


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

```
# By default, 'by' is set to "Species"
get_marginalmeans(model)

# Overall mean (close to 'mean(iris$Sepal.Length)')
get_marginalmeans(model, by = NULL)

## Not run:
# One can estimate marginal means at several values of a 'modulate' variable
get_marginalmeans(model, by = "Petal.Width", length = 3)

# Interactions
model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)

get_marginalmeans(model)
get_marginalmeans(model, by = c("Species", "Petal.Length"), length = 2)
get_marginalmeans(model, by = c("Species", "Petal.Length = c(1, 3, 5)"), length = 2)

## End(Not run)


model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)

get_marginaltrends(model, trend = "Petal.Length", by = "Species")
get_marginaltrends(model, trend = "Petal.Length", by = "Petal.Length")
get_marginaltrends(model, trend = "Petal.Length", by = c("Species", "Petal.Length"))
```

`smoothing` *Smoothing a vector or a time series*

Description

Smoothing a vector or a time series. For data.frames, the function will smooth all numeric variables stratified by factor levels (i.e., will smooth within each factor level combination).

Usage

```
smoothing(x, method = "loess", strength = 0.25, ...)
```

Arguments

`x`	A numeric vector.
`method`	Can be "loess" (default) or "smooth". A loess smoothing can be slow.
`strength`	This argument only applies when `method = "loess"`. Degree of smoothing passed to `span` (see `loess()`).
`...`	Arguments passed to or from other methods.

Value

A smoothed vector or data frame.

Examples

```
x <- sin(seq(0, 4 * pi, length.out = 100)) + rnorm(100, 0, 0.2)
plot(x, type = "l")
lines(smoothing(x, method = "smooth"), type = "l", col = "blue")
lines(smoothing(x, method = "loess"), type = "l", col = "red")

x <- sin(seq(0, 4 * pi, length.out = 10000)) + rnorm(10000, 0, 0.2)
plot(x, type = "l")
lines(smoothing(x, method = "smooth"), type = "l", col = "blue")
lines(smoothing(x, method = "loess"), type = "l", col = "red")
```

---

visualisation_matrix *Create a reference grid*

Description

This function is an alias (another name) for the `insight::get_datagrid()` function. Same arguments apply.

Usage

```
visualisation_matrix(x, ...)

## S3 method for class 'data.frame'
visualisation_matrix(
  x,
  by = "all",
  factors = "reference",
  numerics = "mean",
  preserve_range = FALSE,
  reference = x,
  ...
)

## S3 method for class 'numeric'
visualisation_matrix(x, ...)

## S3 method for class 'factor'
visualisation_matrix(x, ...)
```

Arguments

x	An object from which to construct the reference grid.
...	Arguments passed to or from other methods (for instance, `length` or `range` to control the spread of numeric variables.).
by	Indicates the *focal predictors* (variables) for the reference grid and at which values focal predictors should be represented. If not specified otherwise, representative values for numeric variables or predictors are evenly distributed from the minimum to the maximum, with a total number of `length` values covering that range (see 'Examples'). Possible options for by are:

- `"all"`, which will include all variables or predictors.
- a character vector of one or more variable or predictor names, like `c("Species", "Sepal.Width")`, which will create a grid of all combinations of unique values. For factors, will use all levels, for numeric variables, will use a range of length `length` (evenly spread from minimum to maximum) and for character vectors, will use all unique values.
- a list of named elements, indicating focal predictors and their representative values, e.g. by = `list(Sepal.Length = c(2, 4), Species = "setosa")`.
- a string with assignments, e.g. by = `"Sepal.Length = 2"` or by = `c("Sepal.Length = 2", "Species = 'setosa'")` - note the usage of single and double quotes to assign strings within strings.

There is a special handling of assignments with *brackets*, i.e. values defined inside `[` and `]`.For **numeric** variables, the value(s) inside the brackets should either be

- two values, indicating minimum and maximum (e.g. by = `"Sepal.Length = [0, 5]"`), for which a range of length `length` (evenly spread from given minimum to maximum) is created.
- more than two numeric values by = `"Sepal.Length = [2,3,4,5]"`, in which case these values are used as representative values.
- a "token" that creates pre-defined representative values:
  - for mean and -/+ 1 SD around the mean: `"x = [sd]"`
  - for median and -/+ 1 MAD around the median: `"x = [mad]"`
  - for Tukey's five number summary (minimum, lower-hinge, median, upper-hinge, maximum): `"x = [fivenum]"`
  - for terciles, including minimum and maximum: `"x = [terciles]"`
  - for terciles, excluding minimum and maximum: `"x = [terciles2]"`
  - for quartiles, including minimum and maximum: `"x = [quartiles]"` (same as `"x = [fivenum]"`)
  - for quartiles, excluding minimum and maximum: `"x = [quartiles2]"`
  - for a pretty value range: `"x = [pretty]"`
  - for minimum and maximum value: `"x = [minmax]"`
  - for 0 and the maximum value: `"x = [zeromax]"`

For **factor** variables, the value(s) inside the brackets should indicate one or more factor levels, like by = `"Species = [setosa, versicolor]"`. **Note**: the `length` argument will be ignored when using brackets-tokens.

	The remaining variables not specified in by will be fixed (see also arguments factors and numerics).
factors	Type of summary for factors. Can be "reference" (set at the reference level), "mode" (set at the most common level) or "all" to keep all levels.
numerics	Type of summary for numeric values. Can be "all" (will duplicate the grid for all unique values), any function ("mean", "median", ...) or a value (e.g., numerics = 0).
preserve_range	In the case of combinations between numeric variables and factors, setting preserve_range = TRUE will drop the observations where the value of the numeric variable is originally not present in the range of its factor level. This leads to an unbalanced grid. Also, if you want the minimum and the maximum to closely match the actual ranges, you should increase the length argument.
reference	The reference vector from which to compute the mean and SD. Used when standardizing or unstandardizing the grid using effectsize::standardize.

Value

Reference grid data frame.

Examples

```
# See `?insight::get_datagrid`
```

---

visualisation_recipe.estimate_predicted

Automated plotting for 'modelbased' objects

Description

Most 'modelbased' objects can be visualized using the plot() function, which internally calls the visualisation_recipe() function. See the **examples** below for more information and examples on how to create and customize plots.

Usage

```
## S3 method for class 'estimate_predicted'
visualisation_recipe(
  x,
  show_data = FALSE,
  point = NULL,
  line = NULL,
  pointrange = NULL,
  ribbon = NULL,
  facet = NULL,
  grid = NULL,
  join_dots = getOption("modelbased_join_dots", TRUE),
```

```
  ...
)

## S3 method for class 'estimate_slopes'
visualisation_recipe(
  x,
  line = NULL,
  pointrange = NULL,
  ribbon = NULL,
  facet = NULL,
  grid = NULL,
  ...
)

## S3 method for class 'estimate_grouplevel'
visualisation_recipe(
  x,
  line = NULL,
  pointrange = NULL,
  ribbon = NULL,
  facet = NULL,
  grid = NULL,
  ...
)
```

Arguments

x	A modelbased object.
show_data	Logical, if TRUE, display the "raw" data as a background to the model-based estimation.
point, line, pointrange, ribbon, facet, grid	Additional aesthetics and parameters for the geoms (see customization example).
join_dots	Logical, if TRUE and for categorical focal terms in by, dots (estimates) are connected by lines, i.e. plots will be a combination of dots with error bars and connecting lines. If FALSE, only dots and error bars are shown. It is possible to set a global default value using options(), e.g. options("modelbased_join_dots" = FALSE).
...	Not used.

Details

The plotting works by mapping any predictors from the by argument to the x-axis, colors, alpha (transparency) and facets. Thus, the appearance of the plot depends on the order of the variables that you specify in the by argument. For instance, the plots corresponding to `estimate_relation(model, by=c("Species", "Sepal.Length"))` and `estimate_relation(model, by=c("Sepal.Length", "Species"))` will look different.

The automated plotting is primarily meant for convenient visual checks, but for publication-ready figures, we recommend re-creating the figures using the `ggplot2` package directly.

There are two options to remove the confidence bands or errors bars from the plot. To remove error bars, simply set the `pointrange` geom to `point`, e.g. `plot(..., pointrange = list(geom = "point"))`. To remove the confidence bands from line geoms, use `ribbon = "none"`.

Examples

```
library(ggplot2)
library(see)
# ============================================
# estimate_relation, estimate_expectation, ...
# ============================================
# Simple Model ---------------
x <- estimate_relation(lm(mpg ~ wt, data = mtcars))
layers <- visualisation_recipe(x)
layers
plot(layers)

# visualization_recipe() is called implicitly when you call plot()
plot(estimate_relation(lm(mpg ~ qsec, data = mtcars)))

## Not run:
# And can be used in a pipe workflow
lm(mpg ~ qsec, data = mtcars) |>
  estimate_relation(ci = c(0.5, 0.8, 0.9)) |>
  plot()

# Customize aesthetics ----------

plot(x,
  point = list(color = "red", alpha = 0.6, size = 3),
  line = list(color = "blue", size = 3),
  ribbon = list(fill = "green", alpha = 0.7)
) +
  theme_minimal() +
  labs(title = "Relationship between MPG and WT")

# Customize raw data -------------

plot(x, point = list(geom = "density_2d_filled"), line = list(color = "white")) +
  scale_x_continuous(expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  theme(legend.position = "none")

# Single predictors examples -----------

plot(estimate_relation(lm(Sepal.Length ~ Species, data = iris)))

# 2-ways interaction ------------

# Numeric * numeric
```

```
x <- estimate_relation(lm(mpg ~ wt * qsec, data = mtcars))
plot(x)

# Numeric * factor
x <- estimate_relation(lm(Sepal.Width ~ Sepal.Length * Species, data = iris))
plot(x)

# ============================================
# estimate_means
# ============================================
# Simple Model ---------------
x <- estimate_means(lm(Sepal.Width ~ Species, data = iris), by = "Species")
layers <- visualisation_recipe(x)
layers
plot(layers)

# Customize aesthetics
layers <- visualisation_recipe(x,
  point = list(width = 0.03, color = "red"),
  pointrange = list(size = 2, linewidth = 2),
  line = list(linetype = "dashed", color = "blue")
)
plot(layers)

# Two levels ---------------
data <- mtcars
data$cyl <- as.factor(data$cyl)

model <- lm(mpg ~ cyl * wt, data = data)

x <- estimate_means(model, by = c("cyl", "wt"))
plot(x)


# GLMs ---------------------
data <- data.frame(vs = mtcars$vs, cyl = as.factor(mtcars$cyl))
x <- estimate_means(glm(vs ~ cyl, data = data, family = "binomial"), by = c("cyl"))
plot(x)

## End(Not run)


# ============================================
# estimate_slopes
# ============================================
model <- lm(Sepal.Width ~ Species * Petal.Length, data = iris)
x <- estimate_slopes(model, trend = "Petal.Length", by = "Species")

layers <- visualisation_recipe(x)
layers
plot(layers)

## Not run:
```

```
# Customize aesthetics and add horizontal line and theme
layers <- visualisation_recipe(x, pointrange = list(size = 2, linewidth = 2))
plot(layers) +
  geom_hline(yintercept = 0, linetype = "dashed", color = "red") +
  theme_minimal() +
  labs(y = "Effect of Petal.Length", title = "Marginal Effects")

model <- lm(Petal.Length ~ poly(Sepal.Width, 4), data = iris)
x <- estimate_slopes(model, trend = "Sepal.Width", by = "Sepal.Width", length = 20)
plot(visualisation_recipe(x))

model <- lm(Petal.Length ~ Species * poly(Sepal.Width, 3), data = iris)
x <- estimate_slopes(model, trend = "Sepal.Width", by = c("Sepal.Width", "Species"))
plot(visualisation_recipe(x))

## End(Not run)


# ============================================
# estimate_grouplevel
# ============================================
## Not run:
data <- lme4::sleepstudy
data <- rbind(data, data)
data$Newfactor <- rep(c("A", "B", "C", "D"))

# 1 random intercept
model <- lme4::lmer(Reaction ~ Days + (1 | Subject), data = data)
x <- estimate_grouplevel(model)
layers <- visualisation_recipe(x)
layers
plot(layers)

# 2 random intercepts
model <- lme4::lmer(Reaction ~ Days + (1 | Subject) + (1 | Newfactor), data = data)
x <- estimate_grouplevel(model)
plot(x) +
  geom_hline(yintercept = 0, linetype = "dashed") +
  theme_minimal()
# Note: we need to use hline instead of vline because the axes is flipped

model <- lme4::lmer(Reaction ~ Days + (1 + Days | Subject) + (1 | Newfactor), data = data)
x <- estimate_grouplevel(model)
plot(x)

## End(Not run)
```

`zero_crossings` *Find zero-crossings and inversion points*

Description

Find zero crossings of a vector, i.e., indices when the numeric variable crosses 0. It is useful for finding the points where a function changes by looking at the zero crossings of its derivative.

Usage

```
zero_crossings(x)

find_inversions(x)
```

Arguments

x	A numeric vector.

Value

Vector of zero crossings or points of inversion.

See Also

Based on the `uniroot.all` function from the rootSolve package.

Examples

```
x <- sin(seq(0, 4 * pi, length.out = 100))
plot(x, type = "b")

zero_crossings(x)
find_inversions(x)
```

Index