Package: modelbased (via r-universe)

November 23, 2026

Type Package

Title Estimation of Model-Based Predictions, Contrasts and Means

Version 0.19.0

Maintainer Dominique Makowski <officialeasystats@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 (>= 4.0)

Imports bayestestR (>= 0.19.0), datawizard (>= 1.4.0), insight (>= 1.5.4), parameters (>= 0.29.3), graphics, stats, tools, utils

Suggests afex, BH, betareg, boot, bootES, brglm2, brms, broom, car, carData, coda, collapse, correlation, coxme, curl, dagitty, discovr, easystats, effectsize (>= 1.0.0), emmeans (>= 1.10.2), Formula, gamm4, gganimate, ggdag, ggplot2, glmmTMB (>= 1.1.12), grid, httr2, knitr, lme4, lmerTest, logspline, MASS, MatchIt, Matrix, marginaleffects (>= 0.29.0), mice, mgcv, mvtnorm, nanoparquet, nestedLogit, nnet, ordinal, performance (>= 0.14.0), patchwork, pbkrtest, poorman, pscl, RcppEigen, Rdatasets, report, rmarkdown, rstanarm, rtdists, RWiener, sandwich, scales, see (>= 0.11.0), survival, testthat (>= 3.2.1), tinyplot (>= 0.6.0), tinytable, vdiffr, withr

VignetteBuilder knitr

Encoding UTF-8

Language en-US

Config/testthat/edition 3

Config/testthat/parallel false

Config/Needs/check stan-dev/cmdstanr

Config/Needs/website easystats/easystatstemplate

LazyData true

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Dominique Makowski [aut, cre] (ORCID: <https://orcid.org/0000-0001-5375-9967>), Daniel Lüdecke [aut] (ORCID: <https://orcid.org/0000-0002-8895-3206>), Mattan S. Ben-Shachar [aut] (ORCID: <https://orcid.org/0000-0002-4287-4801>), Indrajeet Patil [aut] (ORCID: <https://orcid.org/0000-0003-1995-6531>), Rémi Thériault [aut] (ORCID: <https://orcid.org/0000-0003-4315-6788>)

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

Date/Publication 2026-11-22 23:30:02 UTC

RemoteUrl https://github.com/cran/modelbased

RemoteRef HEAD

RemoteSha cae32fde46d37c5f1e304a985a54d814ae5fcecd

Contents

as.data.frame.estimate_contrasts . . . 3
coffee_data . . . 4
collapse_by_group . . . 4
describe_nonlinear . . . 5
display.estimate_contrasts . . . 6
efc . . . 9
estimate_contrasts . . . 9
estimate_expectation . . . 24
estimate_grouplevel . . . 30
estimate_means . . . 33
estimate_slopes . . . 42
fish . . . 51
get_emcontrasts . . . 51
modelbased-options . . . 60
plot.estimate_predicted . . . 61
pool_contrasts . . . 69
pool_predictions . . . 70
puppy_love . . . 71
residualize_over_grid . . . 72
smoothing . . . 73
stigma . . . 74
zero_crossings . . . 74

Index **76**

`as.data.frame.estimate_contrasts` *Converting modelbased-objects into raw data frames*

Description

`as.data.frame()` method for **modelbased** objects. Can be used to return a "raw" data frame without attributes and with standardized column names. By default, the original column names are preserved, to avoid unexpected changes, but this can be changed with the `preserve_names` argument.

Usage

```
## S3 method for class 'estimate_contrasts'
as.data.frame(
  x,
  row.names = NULL,
  optional = FALSE,
  ...,
  stringsAsFactors = FALSE,
  use_responsename = FALSE,
  preserve_names = TRUE
)
```

Arguments

`x`	An object returned by the different `estimate_*()` functions.
`row.names`	`NULL` or a character vector giving the row names for the data frame. Missing values are not allowed.
`optional`	logical. If `TRUE`, setting row names and converting column names (to syntactic names: see `make.names`) is optional. Note that all of R's **base** package `as.data.frame()` methods use `optional` only for column names treatment, basically with the meaning of `data.frame(*, check.names = !optional)`. See also the `make.names` argument of the `matrix` method.
`...`	Arguments passed to the `data.frame` method of `as.data.frame()`.
`stringsAsFactors`	logical: should the character vector be converted to a factor?
`use_responsename`	Logical, if `TRUE`, the response variable name is used as column name for the estimate column (if available). If `FALSE` (default), the column is named `"Coefficient"`.
`preserve_names`	Logical, if `TRUE` (default), the original column names are preserved. If `FALSE`, the estimate column is renamed to either the response name (if `use_responsename = TRUE`) or to `"Coefficient"`.

Value

A data frame.

Examples

```
model <- lm(Petal.Length ~ Species, data = iris)
out <- estimate_means(model, "Species")

# default
print(out)

as.data.frame(out)

as.data.frame(out, preserve_names = FALSE)

as.data.frame(out, preserve_names = FALSE, use_responsename = TRUE)
```

---

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.

---

collapse_by_group *Collapse raw data by random effect groups*

Description

This function extracts the raw data points (i.e. the data that was used to fit the model) and "averages" (i.e. "collapses") the response variable over the levels of the grouping factor given in `collapse_by`. Only works with mixed models.

Usage

```
collapse_by_group(grid, model, collapse_by = NULL, residuals = FALSE)
```

Arguments

grid	A data frame representing the data grid, or an object of class estimate_means or estimate_predicted, as returned by the different estimate_*() functions.
model	The model for which to compute partial residuals. The data grid grid should match to predictors in the model.
collapse_by	Name of the (random effects) grouping factor. Data is collapsed by the levels of this factor.
residuals	Logical, if TRUE, collapsed partial residuals instead of raw data by the levels of the grouping factor.

Value

A data frame with raw data points, averaged over the levels of the given grouping factor from the random effects. The group level of the random effect is saved in the column "random".

Examples

```
data(efc, package = "modelbased")
efc$e15relat <- as.factor(efc$e15relat)
efc$c161sex <- as.factor(efc$c161sex)
levels(efc$c161sex) <- c("male", "female")
model <- lme4::lmer(neg_c_7 ~ c161sex + (1 | e15relat), data = efc)
me <- estimate_means(model, "c161sex")
head(efc)
collapse_by_group(me, model, "e15relat")
```

---

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")
```

---

display.estimate_contrasts

Printing modelbased-objects

Description

print() method for **modelbased** objects. Can be used to tweak the output of tables.

Usage

```
## S3 method for class 'estimate_contrasts'
display(
  object,
  select = NULL,
  include_grid = NULL,
  full_labels = NULL,
  format = "markdown",
  ...
)

## S3 method for class 'estimate_contrasts'
format(
  x,
  format = NULL,
  select = getOption("modelbased_select", NULL),
  include_grid = getOption("modelbased_include_grid", FALSE),
```

```
  ...
)

## S3 method for class 'estimate_contrasts'
print(x, select = NULL, include_grid = NULL, full_labels = NULL, ...)
```

Arguments

`select` Determines which columns are printed and the table layout. There are two options for this argument:

- **A string expression with layout pattern**
  `select` is a string with "tokens" enclosed in braces. These tokens will be replaced by their associated columns, where the selected columns will be collapsed into one column. Following tokens are replaced by the related coefficients or statistics: `{estimate}`, `{se}`, `{ci}` (or `{ci_low}` and `{ci_high}`), `{p}`, `{pd}` and `{stars}`. The token `{ci}` will be replaced by `{ci_low}`, `{ci_high}`. Example: `select = "{estimate}{stars} ({ci})"` It is possible to create multiple columns as well. A `|` separates values into new cells/columns. Example: `select = "{estimate} ({ci})|{p}"`.
- **A string indicating a pre-defined layout**
  `select` can be one of the following string values, to create one of the following pre-defined column layouts:
  - `"minimal"`: Estimates, confidence intervals and numeric p-values, in two columns. This is equivalent to `select = "{estimate} ({ci})|{p}"`.
  - `"short"`: Estimate, standard errors and numeric p-values, in two columns. This is equivalent to `select = "{estimate} ({se})|{p}"`.
  - `"ci"`: Estimates and confidence intervals, no asterisks for p-values. This is equivalent to `select = "{estimate} ({ci})"`.
  - `"se"`: Estimates and standard errors, no asterisks for p-values. This is equivalent to `select = "{estimate} ({se})"`.
  - `"ci_p"`: Estimates, confidence intervals and asterisks for p-values. This is equivalent to `select = "{estimate}{stars} ({ci})"`.
  - `"se_p"`: Estimates, standard errors and asterisks for p-values. This is equivalent to `select = "{estimate}{stars} ({se})"`..

  Using `select` to define columns will re-order columns and remove all columns related to uncertainty (standard errors, confidence intervals), test statistics, and p-values (and similar, like `pd` or `BF` for Bayesian models), because these are assumed to be included or intentionally excluded when using `select`. The new column order will be: Parameter columns first, followed by the "glue" columns, followed by all remaining columns. If further columns should also be placed first, add those as `focal_terms` attributes to `x`. I.e., following columns are considers as "parameter columns" and placed first: `c(easystats_columns("parameter"), attributes(x)$focal_terms)`.

  Note: glue-like syntax is still experimental in the case of more complex models (like mixed models) and may not return expected results.

`include_grid` Logical, if `TRUE`, the data grid is included in the table output. Only applies to prediction-functions like `estimate_relation()` or `estimate_link()`. Default

	is NULL, which will set the value based on options(modelbased_include_grid), and use FALSE is no option is set.
full_labels	Logical, if TRUE (default), all labels for focal terms are shown. If FALSE, redundant (duplicated) labels are removed from rows. Default is NULL, which will set the value based on options(modelbased_full_labels), and use TRUE is no option is set.
format	String, indicating the output format. Can be "markdown" "html", or "tt". format = "html" create a HTML table using the *gt* package. format = "tt" creates a tinytable object, which is either printed as markdown or HTML table, depending on the environment. See insight::export_table() for details.
...	Arguments passed to insight::format_table() or insight::export_table().
x, object	An object returned by the different estimate_*() functions.

Value

Invisibly returns x.

Global Options to Customize Tables when Printing

Columns and table layout can be customized using options():

- modelbased_select: options(modelbased_select = <string>) will set a default value for the select argument and can be used to define a custom default layout for printing.
- modelbased_include_grid: options(modelbased_include_grid = TRUE) will set a default value for the include_grid argument and can be used to include data grids in the output by default or not.
- modelbased_full_labels: options(modelbased_full_labels = FALSE) will remove redundant (duplicated) labels from rows.
- easystats_display_format: options(easystats_display_format = <value>) will set the default format for the display() methods. Can be one of "markdown", "html", or "tt".
- options(easystats_ci_separator = <value>) defines the separator string used between the lower and upper CI limits. Defaults to ", ", resulting in an output like [0.3, 0.8].
- options(easystats_ci_brackets = <value>) determines whether brackets or parentheses are used to enclose formatted CIs. Can be FALSE to omit them, TRUE to include them (defaults to [and] for text output), or a character vector of length 2 to define custom opening and closing symbols (e.g., c("(", ")")).

Note

Use print_html() and print_md() to create tables in HTML or markdown format, respectively.

Examples

```
model <- lm(Petal.Length ~ Species, data = iris)
out <- estimate_means(model, "Species")

# default
print(out)
```

```
# smaller set of columns
print(out, select = "minimal")

# remove redundant labels
data(efc, package = "modelbased")
efc <- datawizard::to_factor(efc, c("c161sex", "c172code", "e16sex"))
levels(efc$c172code) <- c("low", "mid", "high")
fit <- lm(neg_c_7 ~ c161sex * c172code * e16sex, data = efc)
out <- estimate_means(fit, c("c161sex", "c172code", "e16sex"))
print(out, full_labels = FALSE, select = "{estimate} ({se})")
```

`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. For plotting, check the examples in `see::plot.see_estimate_contrasts()`. See also other related functions such as `estimate_means()` and `estimate_slopes()`.

Proper interpretation of the output requires understanding the assumptions behind your model predictions and which target quantity they refer to. We therefore strongly recommend thoroughly reading:

- the documentation for the `estimate` argument,
- the vignette on marginalization methods, and
- the specific considerations for mixed models.

Usage

```
estimate_contrasts(model, ...)

## Default S3 method:
estimate_contrasts(
  model,
  contrast = NULL,
```

```
  by = NULL,
  predict = NULL,
  ci = 0.95,
  comparison = "pairwise",
  estimate = NULL,
  p_adjust = "none",
  transform = NULL,
  post_process = NULL,
  keep_iterations = FALSE,
  effectsize = NULL,
  iterations = NULL,
  es_type = NULL,
  backend = NULL,
  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()`: Arguments such as `length`, `digits` or `range` can be used to control the (number of) representative values. For integer variables, `protect_integers` modulates whether these should also be treated as numerics, i.e. values can have fractions or not. • **marginaleffects**: Internally used functions are `avg_predictions()` for means and contrasts, and `avg_slope()` for slopes. Therefore, arguments for instance like `vcov`, `equivalence`, `df`, `slope`, `hypothesis` or even `newdata` can be passed to those functions (note that `data` is supported as an alias for `newdata` for consistency across the *easystats* ecosystem). A `weights` argument is passed to the `wts` argument in `avg_predictions()` or `avg_slopes()`, however, weights can only be applied when `estimate` is `"average"` or `"population"` (i.e. for those marginalization options that do not use data grids). Other arguments, such as `re.form` or `allow.new.levels`, may be passed to `predict()` (which is internally used by *marginaleffects*) if supported by that model class. • **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 `bayestestR::describe_post` thus, arguments like, for example, `centrality`, `rope_range`, or `test` are passed to that function. • Especially for `estimate_contrasts()` with integer focal predictors, for which contrasts should be calculated, use argument `integer_as_continuous` to set the maximum number of unique values in an integer predictor to treat that predictor as "discrete integer" or as numeric. For the first case, contrasts are calculated between values of the predictor, for the latter, contrasts of

slopes are calculated. If the integer has more than `integer_as_continuous` unique values, it is treated as numeric. Defaults to `5`. Set to `TRUE` to always treat integer predictors as continuous.

- For count regression models that use an offset term, use `offset = <value>` to fix the offset at a specific value. Or use `estimate = "average"` or `estimate = "population"` without specifying the `offset`, to average predictions over the distribution of the offset (if appropriate).

`contrast` A character vector indicating the name of the variable(s) for which to compute the contrasts, optionally including representative values or levels at which contrasts are evaluated (e.g., `contrast="x=c('a','b')"`). **Note:** It is also possible to contrast average slopes, i.e. `contrast` can be the name of two numeric predictors. However, while it is possible to filter data for one numeric contrast (e.g., `contrast = c("num_pred=c(0, 1, 3)")`), it is not possible to "filter" at certain values for two numeric predictors. For contrasting slopes, the `comparison` will always be `"pairwise"`. It is possible to compute pairwise comparisons of two average slopes at the levels of a third variable, by also adding that variable to the `contrast` argument, e.g. `contrast = c("num1", "num2", "factor")`. See 'Examples'. For frequentist ordinal models and `backend = "emmeans"`, `contrast` can name the response with two or more categories, for example `contrast = "y = c('low', 'high')"`, to contrast only these response categories (with `predict = "prob"`, the default).

`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). `by` can be a character (vector) naming the focal predictors, optionally including representative values or levels at which focal predictors are evaluated (e.g., `by = "x = c(1, 2)"`). When `estimate` is *not* `"average"`, the `by` argument is used to create a "reference grid" or "data grid" with representative values for the focal predictors. In this case, `by` can also be list of named elements. See details in `insight::get_datagrid()` to learn more about how to create data grids for predictors of interest. For `estimate_means()`, `by` defaults to `"auto"`, which automatically selects the first focal predictor found in the model. If `by = NULL`, the grand mean of the response is predicted. For frequentist ordinal models (for example from `MASS::polr()`, `ordinal::clm()` or `ordinal::clmm()`), `by` can also name the response with values, for example `by = c("x", "y = c('low', 'high')")`, or as an element of a list or a column of a data frame, for example `by = list(x = c(1, 2), y = "low")`. `estimate_means()`, `estimate_contrasts()` and `estimate_slopes()` then keep only the rows of these response categories. This filter works with `backend = "emmeans"` and `predict = "prob"` (the default).

`predict` Is passed to the `type` argument in `emmeans::emmeans()` (when `backend = "emmeans"`) or in `marginaleffects::avg_predictions()` (when `backend = "marginaleffects"`). Valid options for `predict` are:

- `backend = "marginaleffects"`: `predict` can be `"response"`, `"link"`, `"inverse_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. The "inverse_link" is a special option, comparable to *marginaleffects*' invlink(link) option. It will calculate predictions on the link scale and then back-transform to the response scale.

- 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"). See also this vignette. For ordinal models (e.g. from MASS::polr() or ordinal::clm()), estimate_means(), estimate_contrasts() and estimate_slopes() also accept the ordinal modes of *emmeans*: "prob", "cum.prob", "exc.prob", "linear.predictor", "latent" and "mean.class" (see vignette("models", package = "emmeans")). For these models, the default is "prob", which returns one probability per response category (column Response). The modes "cum.prob", "exc.prob" and "linear.predictor" return one value per threshold (column Threshold). For modes with response categories or thresholds, contrasts compare the focal levels within each one, unless contrast names the response. For estimate_slopes(), the slopes for "prob" and "mean.class" are computed from the slopes of the cumulative probabilities.

See also section *Predictions on different scales*.

ci — Confidence Interval (CI) level. Must be numeric between 0 and 1. Defaults to 0.95. Use NULL to suppress calculation of standard errors and confidence intervals.

comparison — Specify the type of contrasts or tests that should be carried out. See also section *Comparison options* below for details.

- When backend = "emmeans", can be one of "pairwise", "revpairwise", "poly", "consec", "eff", "del.eff", "mean_chg", "trt.vs.ctrl", "dunnett", "wtcon" and some more. Defaults to "revpairwise", to return consistent results (regarding the sign) with the default backend = "marginaleffects". To test multiple hypotheses jointly (usually used for factorial designs), comparison can also be "joint". 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. For options not described below, see documentation of marginaleffects::comparisons, this website and section *Comparison options* below.
  - String: One of "pairwise", "ratio", "revpairwise", "reference", "sequential", "meandev" "meanotherdev", "poly", "helmert", or "trt_vs_ctrl". To test multiple hypotheses jointly (usually used for factorial designs), comparison can also be "joint" or "omnibus". In this case, use the test argument to specify which test should be conducted: "F" (default) or "Chi2", and use argument null to specify the null-hypothesis to test against. For "omnibus", null is set to the overall mean or prevalence/proportion.
  - String: Special string options are "inequality", "inequality_ratio", and "inequality_pairwise". comparison = "inequality" computes

the marginal effect inequality summary of categorical predictors' overall effects, respectively, the comprehensive effect of an independent variable across all outcome categories of a nominal or ordinal dependent variable (also called *absolute inequality*, or total marginal effect, see *Mize and Han, 2025*). `"inequality_ratio"` computes the ratio of marginal effect inequality measures, also known as *relative inequality*. This is useful to compare the relative effects of different predictors on the dependent variable. It provides a measure of how much more or less inequality one predictor has compared to another. `comparison = "inequality_pairwise"` computes pairwise differences of absolute inequality measures, while `"inequality_ratio_pairwise"` computes pairwise differences of relative inequality measures (ratios). See an overview of applications in the related case study in the vignettes.

- 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`, `pairwise`, `sequential`, `meandev`, etc., see string-options). Optionally, comparisons can be carried out within subsets by indicating the grouping variable after a vertical bar (`|`). If the left-hand side is missing, it defaults to `difference` (i.e. `comparison = ~pairs | group` is identical to `comparison = difference ~ pairs | group`).
- A custom function, e.g. `comparison = I(my_fun(x))`, or `<comparison> ~ I(my_fun(x)) | gr` (where `<comparison>` can be `difference` or `ratio`, or skipped).
- If contrasts should be calculated (or grouped by) factors, `comparison` can also be a matrix that specifies factor contrasts (see 'Examples').

`estimate` The `estimate` argument determines how predictions are averaged ("marginalized") over variables not specified in `by` or `contrast` (non-focal predictors). It controls whether predictions represent a "specific" or "typical" individual (*conditional*, *individual-level* predictions), or an "average" individual from the sample, or an "counterfactual" individual from a broader population (*marginal*, *population-level* predictions).

Conditional Predictions based on artificially constructed reference grids

- `"specific"`: The estimated means represent a rather "theoretical" view of the data. A prediction is made for a specific individual from the sample (i.e., a very specific combination of predictor values). This answers the question about the expected average value of the target variable for a *specific* observation.
- `"typical"` (Default): Calculates predictions for a balanced data grid representing all combinations of focal predictor levels (specified in `by`). For non-focal numeric predictors, it uses the mean; for non-focal categorical predictors, it marginalizes (averages) over the levels. This represents a "typical" observation based on the data grid and is useful for comparing groups. It answers: "What would the average outcome be for a 'typical' observation?". This is the default approach when estimating marginal means using the *emmeans* package.

Marginal predictions based on actual empirical observations

- `"average"`: Calculates predictions for each observation in the sample and then averages these predictions within each group defined by the focal predictors. This reflects the sample's actual distribution of non-focal predictors, not a balanced grid. It answers: "What is the predicted value for an average observation in my data?"
- `"population"`: "Clones" each observation, creating copies with all possible combinations of focal predictor levels. It then averages the predictions across these "counterfactual" observations (non-observed permutations) within each group. This extrapolates to a hypothetical broader population, considering "what if" scenarios. It answers: "What is the predicted response for the 'average' observation in a broader possible target population?" This approach mimics a "pseudo-randomization" and hence can be more apt to generalize, which should be used for **G-computation** (causal inference, see *Chatton and Rohrer 2024*). `"counterfactual"` is an alias for `"population"`.

You can set a default option for the `estimate` argument via `options()`, e.g. `options(modelbased_estimate = "average")`.

Note following limitations:

- Especially for continuous predictors, `estimate = "average"` may produce erratic-looking outputs, because it calculates the average based only on the data points that actually exist. This is in particular important for two or more focal predictors, because it doesn't generate a *complete* grid of all theoretical combinations of predictor values. Consequently, the output may not include all the values. To resolve this, you may provide a defined data grid via the `data` argument containing the values of interest.
- For the same reason, *conditional* predictions or `estimate = "population"` typically yield smoother curves in **visualizations** for continuous focal predictors, whereas *marginal* predictions with `estimate = "average"` may introduce noisy visual artifacts.
- Filtering the output at values of continuous predictors, e.g. `by = "x=1:5"`, in combination with `estimate = "average"` may result in returning an empty data frame because the requested values may not overlap with existing data. In such cases, you can use `estimate = "typical"` or use the `newdata` argument to provide a data grid of predictor values at which to evaluate predictions.
- `estimate = "population"` is not available for `estimate_slopes()`.

`p_adjust` The p-values adjustment method for frequentist multiple comparisons. Can be one of `"none"` (default), `"hochberg"`, `"hommel"`, `"bonferroni"`, `"BH"`, `"BY"`, `"fdr"`, `"tukey"`, `"sidak"`, `"sup-t"`, `"esarey"` or `"holm"`. The `"esarey"` option is specifically for the case of Johnson-Neyman intervals, i.e. when calling `estimate_slopes()` with two numeric predictors in an interaction term. `"sup-t"` computes simultaneous confidence bands, also called sup-t confidence band (Montiel Olea & Plagborg-Møller, 2019). Details for the other options can be found in the p-value adjustment section of the `emmeans::test` documentation or `?stats::p.adjust`. Note that certain options provided by the **emmeans** package are only available if you set `backend = "emmeans"`. The

Tukey adjustment is only appropriate for pairwise comparisons. With backend = "marginaleffects", "tukey" is therefore replaced by "sidak" (with a message) for comparisons that are not pairwise, for example comparison = "reference" or "sequential", as **emmeans** does. The Sidak adjustment counts the comparisons within each group of the by variables that are columns of the output. When the output has no such column, for example for contrasts of slopes without a group term in comparison, it counts all comparisons.

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. Note that no standard errors are returned when transformations are applied.

post_process — Optional formula, character string or function (see comparison), or a list of formulas, strings or functions, to process subsequent, multi-step comparisons. After the initial comparison in comparison is completed, the results are then post-processed using the specified post-process tests. See 'Examples'.

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, and so on. If keep_iterations is a positive number, only as many columns as indicated in keep_iterations will be added to the output. You can reshape them to a long format by running bayestestR::reshape_iterations().

effectsize — Desired measure of standardized effect size, one of "emmeans", "marginal", or "boot". Default is NULL, i.e. no effect size will be computed. Effect sizes are only available for Gaussian models without random effects, and (only for "emmeans") for ordinal models on the latent scale (see 'Details').

iterations — This argument has two distinct effects, depending on the context. When effectsize = "boot", it defines the number of bootstrap resamples used to calculate the effect size (via bootES::bootES()), and defaults to 200 in this case. For Bayesian models used in estimate_means(), estimate_slopes(), or estimate_contrasts() (and when backend = "marginaleffects"), it is passed to the ndraws argument of the related marginaleffects functions, and defines the number of posterior draws to sample from. If NULL (the default), all draws are used. Note that ndraws currently only has an effect for Bayesian models fit with **brms** or **MCMCglmm**.

es_type — Specifies the type of effect-size measure to estimate when using effectsize = "boot". One of "unstandardized", "cohens.d", "hedges.g", "cohens.d.sigma", "r", or "akp.robust.d". See effect.type argument of bootES::bootES() for details. If not specified, defaults to "cohens.d".

backend — Whether to use "marginaleffects" (default) or "emmeans" 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).

Another difference is that backend = "marginaleffects" will be slower than backend = "emmeans". For most models, this difference is negligible. However, in particular complex models or large data sets fitted with *glmmTMB* can be significantly slower.

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_relation()). 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 the 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.

Comparison options

- `comparison = "pairwise"` (default): This method computes all possible unique differences between pairs of levels of the focal predictor. For example, if a factor has levels A, B, and C, it would compute A-B, A-C, and B-C. The formula notation would be `comparison = difference ~ pairwise`.
- `comparison = "revpairwise"`: Like `"pairwise"`, but reverses the order of levels when comparing, e.g. B-A, C-A, and C-B.
- `comparison = "ratio"`: This method computes all possible unique ratios between pairs of levels of the focal predictor (not differences, like the default). It is basically a shortcut for `comparison = ratio ~ pairwise`.
- `comparison = "reference"`: This compares each level of the focal predictor to a specified reference level (by default, the first level). For example, if levels are A, B, C, and A is the reference, it computes B-A and C-A.
- `comparison = "sequential"`: This compares each level to the one immediately following it in the factor's order. For levels A, B, C, it would compute B-A and C-B.
- `comparison = "meandev"`: This contrasts each level's estimate against the grand mean of all estimates for the focal predictor.
- `comparison = "meanotherdev"`: Similar to `meandev`, but each level's estimate is compared against the mean of all *other* levels, excluding itself.
- `comparison = "poly"`: These are used for ordered categorical variables to test for linear, quadratic, cubic, etc., trends across the levels. They assume equal spacing between levels.
- `comparison = "helmert"`: Contrast 2nd level to the first, 3rd to the average of the first two, and so on. Each level (except the first) is compared to the mean of the preceding levels. For levels A, B, C, it would compute B-A and C-(A+B)/2.
- `comparison = "trt_vs_ctrl"`: This compares all levels (excluding the first, which is typically the control) against the first level. It's often used when comparing multiple treatment groups to a single control group.
- To test multiple hypotheses jointly (usually used for factorial designs), `comparison` can also be `"joint"`. This option runs sequential joint tests on the results of an initial pairwise comparison. To test for differences between groups from an global average, use `comparison = "omnibus"`, which conducts a global omnibus test. Use the `test` argument to specify which test should be conducted: `"F"` (default) or `"Chi2"`. Use the `null` argument to define a specific null-hypothesis to test against. For `"omnibus"`, `null` is set to the overall mean or prevalence/proportion.
- `comparison = "inequality"` computes the *absolute inequality* of groups, or in other words, the marginal effect inequality summary of categorical predictors' overall effects, respectively, the comprehensive effect of an independent variable across all outcome categories of a nominal or ordinal dependent variable (total marginal effect, see *Mize and Han, 2025*). The

marginal effect inequality focuses on the heterogeneity of the effects of a categorical *independent* variable. It helps understand how the effect of the variable differs across its categories or levels. When the *dependent* variable is categorical (e.g., logistic, ordinal or multinomial regression), marginal effect inequality provides a holistic view of how an independent variable affects a nominal or ordinal *dependent* variable. It summarizes the overall impact (absolute inequality, or total marginal effects) across all possible outcome categories.

- `comparison = "inequality_ratio"` is comparable to `comparison = "inequality"`, but instead of calculating the absolute inequality, it computes the *relative inequality* of groups. This is useful to compare the relative effects of different predictors on the dependent variable. It provides a measure of how much more or less inequality one predictor has compared to another.
- `comparison = "inequality_pairwise"` computes pairwise differences of absolute inequality measures, while `"inequality_ratio_pairwise"` computes pairwise differences of relative inequality measures (ratios). Depending on the sign, this measure indicates which of the predictors has a stronger impact on the dependent variable in terms of inequalities.

Examples for analysing inequalities are shown in the related vignette. An introduction into pairwise comparisons and contrasts starts with this vignette.

Context Effects - contrasting average slopes

Calculating contrasts between average slopes can tell us about the "context" effects when modelling within- and between-effects. An example for within- and between effects is described in this vignette. A context effect describes the additional influence that the social or regional environment (e.g., place of residence) has on an individual, independent of their personal characteristics. It demonstrates that people with identical individual circumstances (such as the same income) face different opportunities or risks depending on the environment in which they live. This can be achieved by specifying both numeric predictors in the `contrast` argument, e.g. `contrast = c("x_within", "x_between")`. It is also possible to stratify context effects at different levels of another variable using `by`. If pairwise comparisons of context effects (i.e., the pairwise comparisons of the difference of between within- and between-effects, or the difference of average slopes) at different levels of another variable are required, add that variable to the `contrast` argument instead, e.g. `contrast = c("x_within", "x_between", "factor")`. These contrasts can additionally be stratified by another variable using `by` again, e.g. `contrast = c("x_within", "x_between", "factor1"), by = "factor2"`. Note that when average slopes are contrasted, the `comparison` argument has no effect and is always set to `"pairwise"`. See also 'Examples'.

Effect Size

By default, `estimate_contrasts()` reports no standardized effect size on purpose. Should one request one, some things are to keep in mind. As the authors of *emmeans* write, "There is substantial disagreement among practitioners on what is the appropriate sigma to use in computing effect sizes; or, indeed, whether any effect-size measure is appropriate for some situations. The user is completely responsible for specifying appropriate parameters (or for failing to do so)."

In particular, effect size method `"boot"` does not correct for covariates in the model, so should probably only be used when there is just one categorical predictor (with however many levels). Some believe that if there are multiple predictors or any covariates, it is important to re-compute sigma adding back in the response variance associated with the variables that aren't part of the contrast.

`effectsize = "emmeans"` uses emmeans::eff_size with `sigma = stats::sigma(model)`, `edf = stats::df.residual(mode` and `method = "identity"`. This standardizes using the MSE (sigma). Some believe this works when the contrasts are the only predictors in the model, but not when there are covariates. The response variance accounted for by the covariates should not be removed from the SD used to standardize. Otherwise, *d* will be overestimated.

For `polr` and `clm` models with `predict = "latent"` and a logit or probit link, `effectsize = "emmeans"` uses the SD of the latent error as sigma, i.e. `pi / sqrt(3)` (logit) or `1` (probit), with `edf = Inf`. A `rescale` argument passed to *emmeans* also rescales sigma. `clm` models with a `scale` or `nominal` formula are not supported. The confidence intervals of the effect sizes are not adjusted for multiple comparisons.

`effectsize = "marginal"` uses the following formula to compute effect size: `d_adj <- difference * (1- R2)/ sigma`. This standardizes using the response SD with only the between-groups variance on the focal factor/contrast removed. This allows for groups to be equated on their covariates, but creates an appropriate scale for standardizing the response.

`effectsize = "boot"` uses bootstrapping (defaults to a low value of 200) through bootES::bootES. Adjusts for contrasts, but not for covariates.

Predictions and contrasts at meaningful values (data grids)

To define representative values for focal predictors (specified in `by`, `contrast`, and `slope`), you can use several methods. These values are internally generated by `insight::get_datagrid()`, so consult its documentation for more details.

- You can directly specify values as strings or lists for `by`, `contrast`, and `slope`.
  - For numeric focal predictors, use examples like `by = "gear = c(4, 8)"`, `by = list(gear = c(4, 8))`, `by = "gear = 5:10"` or `by = list(gear = 5:10)`
  - For factor or character predictors, use `by = "Species = c('setosa', 'virginica')"` or `by = list(Species = c('setosa', 'virginica'))`
- You can use "shortcuts" within square brackets, such as `by = "Sepal.Width = [sd]"` or `by = "Sepal.Width = [fivenum]"`
- For numeric focal predictors, if no representative values are specified (i.e., `by = "gear"` and *not* `by = "gear = c(4, 8)"`), `length` and `range` control the number and type of representative values for the focal predictors:
  - `length` determines how many equally spaced values are generated.
  - `range` specifies the type of values, like `"range"` or `"sd"`.
  - `length` and `range` apply to all numeric focal predictors.
  - If you have multiple numeric predictors, `length` and `range` can accept multiple elements, one for each predictor (see 'Examples').
- For integer variables, only values that appear in the data will be included in the data grid, independent from the `length` argument. This behaviour can be changed by setting `protect_integers = FALSE`, which will then treat integer variables as numerics (and possibly produce fractions).

See also this vignette for some examples, and `insight::get_datagrid()` to learn more about how to create data grids for predictors of interest.

Predictions on different scales

The `predict` argument allows to generate predictions on different scales of the response variable. The `"link"` option does not apply to all models, and usually not to Gaussian models. `"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.

Jensen's Inequality:

For non-Gaussian models with non-linear link functions, back-transforming aggregated predictions from the link scale does not equal aggregating predictions already transformed to the response scale - a phenomenon known as *Jensen's inequality*. `"response"` and `"inverse_link"` both return predictions on the response scale; however, `"response"` first calculates predictions on the response scale for each observation and *then* aggregates them by the groups or levels defined in `by`. By contrast, `"inverse_link"` first calculates predictions on the link scale for each observation, aggregates them by the groups or levels defined in `by`, and finally back-transforms the predictions to the response scale. Both approaches have advantages and disadvantages. `"response"` usually produces less biased predictions, but confidence intervals may fall outside reasonable bounds (for instance, becoming negative for count data). The `"inverse_link"` approach is more robust in terms of confidence intervals, though it may produce biased predictions. However, you can set `bias_correction = TRUE` to adjust for this bias (and optionally pass a sensible value to `residual_variance` to specify the assumed variance).

In particular for mixed models, using `"response"` is recommended, because averaging across random effects groups is then more accurate.

References

- Mize, T., & Han, B. (2025). Inequality and Total Effect Summary Measures for Nominal and Ordinal Variables. Sociological Science, 12, 115–157. doi:10.15195/v12.a7
- Montiel Olea, J. L., and Plagborg-Møller, M. (2019). Simultaneous confidence bands: Theory, implementation, and an application to SVARs. Journal of Applied Econometrics, 34(1), 1–17. doi:10.1002/jae.2656

See Also

There is a `plot()` method, which is documented here: `see::plot.see_estimate_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)

# contrasts of slopes ------------------------
# --------------------------------------------

data(qol_cancer, package = "parameters")
qol_cancer$ID <- as.numeric(qol_cancer$ID)
qol_cancer$grp <- as.factor(ifelse(qol_cancer$ID < 100, "Group 1", "Group 2"))
model <- lm(QoL ~ time * education * grp, data = qol_cancer)

# "time" only has integer values and few values, so it's treated like a factor
estimate_contrasts(model, "time", by = "education")

# Setting `integer_as_continuous = TRUE` treats "time" as a continuous
# variable. This allows us to compare its average slope rather than making
# discrete pairwise comparisons at each time point.
estimate_contrasts(
  model,
  contrast = "time",
  by = "education",
  integer_as_continuous = TRUE
)

# pairwise comparisons for multiple groups
estimate_contrasts(
  model,
  "time",
  by = c("education", "grp"),
  integer_as_continuous = TRUE
)

# if we want pairwise comparisons only for one factor, but group by another,
# we need the formula specification and define the grouping variable after
# the vertical bar
```

```
estimate_contrasts(
  model,
  "time",
  by = c("education", "grp"),
  comparison = ~pairwise | grp,
  integer_as_continuous = TRUE
)

# custom factor contrasts - contrasts the average effects of two levels
# against the remaining third level
# ------------------------------------------------------------------------

data(puppy_love, package = "modelbased")
cond_tx <- cbind("no treatment" = c(1, 0, 0), "treatment" = c(0, 0.5, 0.5))
model <- lm(happiness ~ puppy_love * dose, data = puppy_love)
estimate_slopes(model, "puppy_love", by = "dose", comparison = cond_tx)

# Note: for the emmeans-backend, we need to use `estimate_contrasts()` for
# the above example:
cond_tx <- list(`no treatment` = c(1, 0, 0), treatment = c(0, 0.5, 0.5))
estimate_contrasts(
  model,
  contrast = "puppy_love",
  by = "dose",
  comparison = cond_tx,
  backend = "emmeans"
)

# 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")

# Omnibus and Joint Tests -----------------------------------
#
```

```
# ----------------------------------------------------------
data(coffee_data, package = "modelbased")
m <- lm(alertness ~ time * coffee, data = coffee_data)
# joint test for simple effects of "time" for levels of "coffee".
estimate_contrasts(m, contrast = "time", by = "coffee", comparison = "joint")
# global omnibus test to test for differences between "time" levels
estimate_contrasts(m, contrast = "time", comparison = "omnibus")
# test against a different null-hypothesis
estimate_contrasts(m, contrast = "time", comparison = "omnibus", null = 15)

# Context effects ------------------------------------------
# This is the difference of within- and between-effects, which
# typically are two average slopes that are compared. It is
# possible to calculate the context effect at different levels
# of another variable.
# ----------------------------------------------------------
data("qol_cancer", package = "parameters")
qol_cancer <- datawizard::demean(qol_cancer, select = "phq4", by = "ID")
model <- lme4::lmer(
  QoL ~ time * (phq4_within + phq4_between) + (1 + time | ID),
  data = qol_cancer
)

# context effect (difference between within- and between-effect)
# at each time point - we calculate the contrast of two average slopes
# at different levels of "time"
estimate_contrasts(model, c("phq4_within", "phq4_between"), by = "time")

# is the trend of the context effect across time points statistically
# significant? In this case, we just want the contrasts of the overall
# average slopes (not stratfied nor contrasted by time).
estimate_contrasts(model, c("phq4_within", "phq4_between"))

# now we ask whether contexts effects are different for different educational
# levels. We now need to model a 3-way interaction between time, education
# and the centered phq4 variables.
model <- lme4::lmer(
  QoL ~ time * education * (phq4_within + phq4_between) + (1 + time | ID),
  data = qol_cancer
)

# how do time trends of context effects differ between education levels?
estimate_contrasts(model, c("phq4_within", "phq4_between"), by = "education")

# are differences in time trends of context effects statistically significant
# between education levels?
estimate_contrasts(model, c("phq4_within", "phq4_between", "education"))

# Post-processing of multiple comparisons ---------------------
# Caution! Don't expect this example to be meaningful! # It is
# just to demonstrate the usage of the `post_process` argument.
# ----------------------------------------------------------
data("qol_cancer", package = "parameters")
```

```
model <- lme4::lmer(QoL ~ time * education + (1 + time | ID), data = qol_cancer)

# contrasts (pairwise comparisons - the default) by timepoints
estimate_contrasts(model, "education=c('low', 'mid')", by = "time")

# contrasts (pairwise comparisons, the default for `comparison`) by timepoints
# additionally, we compare the differences of these contrasts across timepoints
# against the reference time point (contrasts at times 2 and 3 against
# contrasts at time 1)
estimate_contrasts(model,
  contrast = "education=c('low', 'mid')",
  by = "time",
  post_process = ~reference
)

# multiple post-processing steps - same as before, but calculates
# additional poly-contrasts applied to the reference contrasts (that
# were applied to the default pairwise comparisons of educational levels,
# grouped `by time`)
estimate_contrasts(model,
  contrast = "education=c('low', 'mid')",
  by = "time",
  post_process = list(~reference, ~poly)
)

## 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,
  iterations = NULL,
  keep_iterations = FALSE,
  ...
)

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

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

estimate_relation(
```

```
  model,
  data = "grid",
  by = NULL,
  predict = "expectation",
  ci = 0.95,
  transform = NULL,
  iterations = 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. Must be numeric between 0 and 1. Defaults to 0.95. Use NULL to suppress calculation of standard errors and confidence intervals.
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 that no standard errors are returned when transformations are applied.
iterations	For Bayesian models, this corresponds to the number of posterior draws. If NULL, will use all the draws (one for each iteration of the model). For frequentist models, if not NULL, will generate bootstrapped draws, from which bootstrapped CIs will be computed. Use keep_iterations to control if and how many draws

will be included in the returned output (data frame), which can be used, for instance, for plotting.

`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`, and so on. If `keep_iterations` is a positive number, only as many columns as indicated in `keep_iterations` will be added to the output. You can reshape them to a long format by running `bayestestR::reshape_iterations()`.

`...`
: You can add all the additional control arguments from `insight::get_datagrid()` (used when `data = "grid"`) and `insight::get_predicted()`. Furthermore, for count regression models that use an offset term, use `offset = <value>` to fix the offset at a specific value. For models of class `nestedLogit`, use the `submodel` argument to specify the component for which predictions should be returned (see `?insight::get_predicted` for details).

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.

Finite mixture models

For finite mixture models (currently, only the `brms::mixture()` family from package *brms* is supported), use `predict = "classification"` with `data = NULL` to predict the class membership for each observation (e.g., `estimate_prediction(model, predict = "classification")`). To return predicted values stratified by class membership, use `predict = "link"` (possibly in combination with `data` or `by`, e.g. `estimate_link(model, by = "predictor")`). Other `predict` options will return predicted values of the outcome for the full data, not stratified by class membership.

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
data(mtcars)
```

```
mtcars$gear <- as.factor(mtcars$gear)
model <- glmmTMB::glmmTMB(mpg ~ wt + (1 | gear), data = mtcars)
estimate_expectation(model)
estimate_relation(model)

# Predict random effects and calculate contrasts
estim <- estimate_relation(model, by = "gear")
estim

estimate_contrasts(estim)

# 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, commonly referred to as BLUPs (Best Linear Unbiased Predictors). 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, ...)

## Default S3 method:
estimate_grouplevel(model, type = "random", ...)

## S3 method for class 'brmsfit'
estimate_grouplevel(
  model,
  type = "random",
  dispersion = TRUE,
  test = NULL,
  diagnostic = NULL,
  ...
)

reshape_grouplevel(x, ...)
```

```
## S3 method for class 'estimate_grouplevel'
reshape_grouplevel(x, indices = "all", group = NULL, ...)
```

Arguments

model	A mixed model with random effects.
...	Other arguments passed to parameters::model_parameters() or estimate_means().
type	String, describing the type of estimates that should be returned. Can be "random", "total", or "marginal" (experimental). • If "random" (default), the coefficients correspond to the conditional estimates of the random effects (as they are returned by lme4::ranef()). They typically correspond to the deviation of each individual group from their fixed effect (assuming the random effect is also included as a 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"). These estimates are typically the same as the "total" ones, but are simply expressed on a different scale: *relative* to the fixed effect vs. *absolute* coefficient values. • If "total", it will return the sum of the random effect and its corresponding fixed effects, which internally relies on the coef() method (see ?coef.merMod). Note that type = "total" yet does not return uncertainty indices (such as SE and CI) for models from *lme4* or *glmmTMB*, as the necessary information to compute them is not yet available. However, for Bayesian models, it is possible to compute them. • If "marginal" (experimental), it returns marginal group-level estimates. The random intercepts are computed using marginal means (see estimate_means()), and the random slopes using marginal effects (see estimate_slopes()). This method does not directly extract the parameters estimated by the model, but recomputes them using model predictions. By default, the function uses "typical" marginalization (estimate = "typical"). This results in estimates closely comparable to that when type = "total" (and identical to it when continuous predictors are mean-centered). Setting estimate = "average" instead returns each group's *actual* average predicted outcome, based on that group's own observed predictor distribution rather than a shared reference point (see below for details on this distinction). One benefit of "marginal" over "total"/"random" is interpretability for non-linear models (e.g. logistic or count GLMMs), since predictions are expressed on the response scale rather than the link scale. Note that group-level estimates here are not technically "intercepts" or model parameters, but marginal average levels and effects.
dispersion, test, diagnostic	Arguments passed to parameters::model_parameters() for Bayesian models. By default, it won't return significance or diagnostic indices (as it is not typically very useful).
x	The output of estimate_grouplevel().

indices	A character vector containing the indices (i.e., which columns) to extract (e.g., "Coefficient", "Median").
group	The name of the random factor to select as string value (e.g., "Participant", if the model was y ~ x + (1|Participant).

Details

Benefits of model-based group-level estimates:

Unlike raw group means, BLUPs apply shrinkage: they are a compromise between the group estimate and the population estimate. This improves generalizability and prevents overfitting.

What group-level estimates tell you:

Imagine the two following scenarios:

- Scenario 1: A reaction time task where each trial is assigned to a difficulty condition. It is a manipulated, balanced experimental condition. We have model like rt ~ Condition + (Condition | Participant), where Condition (1-5) is assigned by the experiment, so every participant completes the same set of levels in the same proportions.
- Scenario 2: A reaction time task where participants rate each trial according to their own self-reported confidence - e.g. rt ~ Confidence + (Confidence | Participant), where Confidence (1-5) is generated by the participant, and so varies by participant both in its average level (some participants report high confidence overall, some low) and in how strongly it relates to RT (some have strong metacognition - a steep slope - others don't).

In **Scenario 1**, interpretation is straightforward and largely insensitive to which method is used ("random", "total", or "marginal", with either estimate option). Because every participant is exposed to the same balanced set of Condition levels, each participant's own mean Condition already equals the grand mean. There is no meaningful difference between evaluating at a shared reference point versus at each participant's own observed values, so all approaches converge on essentially the same answer: the random intercept values tell us how fast or slow each participant is overall, and the random slopes of Condition tell us how strongly each participant's RT is affected by increasing difficulty, relative to the population-level effect of Condition.

In **Scenario 2**, this is no longer true, and the choice of method changes what the random intercepts and slopes mean. Two families of estimates emerge:

- type = "total"/"random", and type = "marginal" with estimate = "typical" (default), all evaluate every participant at one *shared* reference point (the model's intercept, or the grand mean of Confidence, respectively). These target each participant's *intrinsic* characteristics - their personal regression line itself - independently of how often they happened to report a given confidence level. The trade-off is that this can be a counterfactual quantity: for a participant whose confidence ratings rarely approach the grand mean, "their predicted RT at average confidence" extrapolates beyond what they actually reported. The random intercept tells us how fast or slow this participant would intrinsically be at the same (hypothetical) confidence level, and the random slope of Confidence tells us how strongly their RT is intrinsically tied to their own confidence ratings - both independent of how often they happen to report high or low confidence.
- type = "marginal" with estimate = "average" instead evaluates each participant using *their own* observed distribution of Confidence. This avoids extrapolation, but it now conflates the participant's intrinsic regression parameters with their personal response style: a chronically

high-confidence participant with a steep slope will get a marginal intercept reflecting *both* their baseline speed *and* their tendency to report high confidence, entangled into a single number. It is no longer a controlled, like-for-like comparison across participants. The random intercept tells us how fast or slow this participant actually was, on average, across the confidence levels they actually reported while the random slope tells us how strongly their RT tracks their own confidence ratings.

As a rule of thumb: use `"total"`, `"random"`, or `"marginal"` with `estimate = "typical"` when you want each group's *intrinsic* characteristics, comparable on equal footing across groups. Use `"marginal"` with `estimate = "average"` when you want each group's *actual, as-observed* outcome, accepting that it may reflect a mix of intrinsic effect and exposure or response style, especially when the predictors values vary across random factor levels.

Examples

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

# Show group-specific effects
random

# Visualize random effects
plot(random)

# Reshape to wide data...
reshaped <- reshape_grouplevel(random, group = "carb", indices = c("Coefficient", "SE"))

# ...and can be easily combined with the original data
alldata <- merge(mtcars, reshaped)

# overall coefficients
r_tot <- estimate_grouplevel(model, type = "total")
cor(r_tot$Coefficient, random$Coefficient) # r = 1 (just a scale difference)

# marginal coefficients
r_mar <- estimate_grouplevel(model, type = "marginal")
cor(r_mar$Coefficient, r_tot$Coefficient)
```

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

Description

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

Proper interpretation of the output requires understanding the assumptions behind your model predictions and which target quantity they refer to. We therefore strongly recommend thoroughly reading:

- the documentation for the `estimate` argument,
- the vignette on marginalization methods, and
- the specific considerations for mixed models.

Usage

```
estimate_means(
  model,
  by = "auto",
  predict = NULL,
  ci = 0.95,
  estimate = NULL,
  transform = NULL,
  keep_iterations = FALSE,
  backend = NULL,
  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). `by` can be a character (vector) naming the focal predictors, optionally including representative values or levels at which focal predictors are evaluated (e.g., `by = "x = c(1, 2)"`). When `estimate` is *not* `"average"`, the `by` argument is used to create a "reference grid" or "data grid" with representative values for the focal predictors. In this case, `by` can also be list of named elements. See details in `insight::get_datagrid()` to learn more about how to create data grids for predictors of interest. For `estimate_means()`, `by` defaults to `"auto"`, which automatically selects the first focal predictor found in the model. If `by = NULL`, the grand mean of the response is predicted. For frequentist ordinal models (for example from `MASS::polr()`, `ordinal::clm()` or `ordinal::clmm()`), `by` can also name the response with values, for example `by = c("x", "y = c('low', 'high')")`, or as an element of a list or a column of a data frame, for example `by = list(x = c(1, 2), y = "low")`. `estimate_means()`, `estimate_contrasts()` and `estimate_slopes()` then keep only the rows of these response categories. This filter works with `backend = "emmeans"` and `predict = "prob"` (the default).
`predict`	Is passed to the `type` argument in `emmeans::emmeans()` (when `backend = "emmeans"`) or in `marginaleffects::avg_predictions()` (when `backend = "marginaleffects"`). Valid options for `predict` are:

- backend = "marginaleffects": predict can be "response", "link", "inverse_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. The "inverse_link" is a special option, comparable to *marginaleffects*' invlink(link) option. It will calculate predictions on the link scale and then back-transform to the response scale.
- 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"). See also this vignette. For ordinal models (e.g. from MASS::polr() or ordinal::clm()), estimate_means(), estimate_contrasts() and estimate_slopes() also accept the ordinal modes of *emmeans*: "prob", "cum.prob", "exc.prob", "linear.predictor", "latent" and "mean.class" (see vignette("models", package = "emmeans")). For these models, the default is "prob", which returns one probability per response category (column Response). The modes "cum.prob", "exc.prob" and "linear.predictor" return one value per threshold (column Threshold). For modes with response categories or thresholds, contrasts compare the focal levels within each one, unless contrast names the response. For estimate_slopes(), the slopes for "prob" and "mean.class" are computed from the slopes of the cumulative probabilities.

See also section *Predictions on different scales*.

ci — Confidence Interval (CI) level. Must be numeric between 0 and 1. Defaults to 0.95. Use NULL to suppress calculation of standard errors and confidence intervals.

estimate — The estimate argument determines how predictions are averaged ("marginalized") over variables not specified in by or contrast (non-focal predictors). It controls whether predictions represent a "specific" or "typical" individual (*conditional*, *individual-level* predictions), or an "average" individual from the sample, or an "counterfactual" individual from a broader population (*marginal*, *population-level* predictions).

Conditional Predictions based on artificially constructed reference grids

- "specific": The estimated means represent a rather "theoretical" view of the data. A prediction is made for a specific individual from the sample (i.e., a very specific combination of predictor values). This answers the question about the expected average value of the target variable for a *specific* observation.
- "typical" (Default): Calculates predictions for a balanced data grid representing all combinations of focal predictor levels (specified in by). For non-focal numeric predictors, it uses the mean; for non-focal categorical predictors, it marginalizes (averages) over the levels. This represents a "typical" observation based on the data grid and is useful for comparing groups. It answers: "What would the average outcome be for a 'typical' observation?". This is the default approach when estimating marginal means using the *emmeans* package.

Marginal predictions based on actual empirical observations

- `"average"`: Calculates predictions for each observation in the sample and then averages these predictions within each group defined by the focal predictors. This reflects the sample's actual distribution of non-focal predictors, not a balanced grid. It answers: "What is the predicted value for an average observation in my data?"
- `"population"`: "Clones" each observation, creating copies with all possible combinations of focal predictor levels. It then averages the predictions across these "counterfactual" observations (non-observed permutations) within each group. This extrapolates to a hypothetical broader population, considering "what if" scenarios. It answers: "What is the predicted response for the 'average' observation in a broader possible target population?" This approach mimics a "pseudo-randomization" and hence can be more apt to generalize, which should be used for **G-computation** (causal inference, see *Chatton and Rohrer 2024*). `"counterfactual"` is an alias for `"population"`.

You can set a default option for the `estimate` argument via `options()`, e.g. `options(modelbased_estimate = "average")`.

Note following limitations:

- Especially for continuous predictors, `estimate = "average"` may produce erratic-looking outputs, because it calculates the average based only on the data points that actually exist. This is in particular important for two or more focal predictors, because it doesn't generate a *complete* grid of all theoretical combinations of predictor values. Consequently, the output may not include all the values. To resolve this, you may provide a defined data grid via the `data` argument containing the values of interest.
- For the same reason, *conditional* predictions or `estimate = "population"` typically yield smoother curves in **visualizations** for continuous focal predictors, whereas *marginal* predictions with `estimate = "average"` may introduce noisy visual artifacts.
- Filtering the output at values of continuous predictors, e.g. `by = "x=1:5"`, in combination with `estimate = "average"` may result in returning an empty data frame because the requested values may not overlap with existing data. In such cases, you can use `estimate = "typical"` or use the `newdata` argument to provide a data grid of predictor values at which to evaluate predictions.
- `estimate = "population"` is not available for `estimate_slopes()`.

`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. Note that no standard errors are returned when transformations are applied.

`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`, and so on. If

keep_iterations is a positive number, only as many columns as indicated in keep_iterations will be added to the output. You can reshape them to a long format by running bayestestR::reshape_iterations().

backend — Whether to use "marginaleffects" (default) or "emmeans" 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).

Another difference is that backend = "marginaleffects" will be slower than backend = "emmeans". For most models, this difference is negligible. However, in particular complex models or large data sets fitted with *glmmTMB* can be significantly slower.

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(): Arguments such as length, digits or range can be used to control the (number of) representative values. For integer variables, protect_integers modulates whether these should also be treated as numerics, i.e. values can have fractions or not.
- **marginaleffects**: Internally used functions are avg_predictions() for means and contrasts, and avg_slope() for slopes. Therefore, arguments for instance like vcov, equivalence, df, slope, hypothesis or even newdata can be passed to those functions (note that data is supported as an alias for newdata for consistency across the *easystats* ecosystem). A weights argument is passed to the wts argument in avg_predictions() or avg_slopes(), however, weights can only be applied when estimate is "average" or "population" (i.e. for those marginalization options that do not use data grids). Other arguments, such as re.form or allow.new.levels, may be passed to predict() (which is internally used by *marginaleffects*) if supported by that model class.
- **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 bayestestR::describe_post thus, arguments like, for example, centrality, rope_range, or test are passed to that function.
- Especially for estimate_contrasts() with integer focal predictors, for which contrasts should be calculated, use argument integer_as_continuous to set the maximum number of unique values in an integer predictor to treat that predictor as "discrete integer" or as numeric. For the first case, contrasts are calculated between values of the predictor, for the latter, contrasts of slopes are calculated. If the integer has more than integer_as_continuous unique values, it is treated as numeric. Defaults to 5. Set to TRUE to always treat integer predictors as continuous.

- For count regression models that use an offset term, use `offset = <value>` to fix the offset at a specific value. Or use `estimate = "average"` or `estimate = "population"` without specifying the `offset`, to average predictions over the distribution of the offset (if appropriate).

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_relation()`). 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 the 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.

Predictions and contrasts at meaningful values (data grids)

To define representative values for focal predictors (specified in `by`, `contrast`, and `slope`), you can use several methods. These values are internally generated by `insight::get_datagrid()`, so consult its documentation for more details.

- You can directly specify values as strings or lists for `by`, `contrast`, and `slope`.
  - For numeric focal predictors, use examples like `by = "gear = c(4, 8)"`, `by = list(gear = c(4, 8))`, `by = "gear = 5:10"` or `by = list(gear = 5:10)`
  - For factor or character predictors, use `by = "Species = c('setosa', 'virginica')"` or `by = list(Species = c('setosa', 'virginica'))`
- You can use "shortcuts" within square brackets, such as `by = "Sepal.Width = [sd]"` or `by = "Sepal.Width = [fivenum]"`
- For numeric focal predictors, if no representative values are specified (i.e., `by = "gear"` and *not* `by = "gear = c(4, 8)"`), `length` and `range` control the number and type of representative values for the focal predictors:
  - `length` determines how many equally spaced values are generated.
  - `range` specifies the type of values, like `"range"` or `"sd"`.
  - `length` and `range` apply to all numeric focal predictors.
  - If you have multiple numeric predictors, `length` and `range` can accept multiple elements, one for each predictor (see 'Examples').
- For integer variables, only values that appear in the data will be included in the data grid, independent from the `length` argument. This behaviour can be changed by setting `protect_integers = FALSE`, which will then treat integer variables as numerics (and possibly produce fractions).

See also this vignette for some examples, and `insight::get_datagrid()` to learn more about how to create data grids for predictors of interest.

Predictions on different scales

The `predict` argument allows to generate predictions on different scales of the response variable. The `"link"` option does not apply to all models, and usually not to Gaussian models. `"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.

Jensen's Inequality:

For non-Gaussian models with non-linear link functions, back-transforming aggregated predictions from the link scale does not equal aggregating predictions already transformed to the response scale - a phenomenon known as *Jensen's inequality*. `"response"` and `"inverse_link"` both return predictions on the response scale; however, `"response"` first calculates predictions on

the response scale for each observation and *then* aggregates them by the groups or levels defined in by. By contrast, "inverse_link" first calculates predictions on the link scale for each observation, aggregates them by the groups or levels defined in by, and finally back-transforms the predictions to the response scale. Both approaches have advantages and disadvantages. "response" usually produces less biased predictions, but confidence intervals may fall outside reasonable bounds (for instance, becoming negative for count data). The "inverse_link" approach is more robust in terms of confidence intervals, though it may produce biased predictions. However, you can set bias_correction = TRUE to adjust for this bias (and optionally pass a sensible value to residual_variance to specify the assumed variance).

In particular for mixed models, using "response" is recommended, because averaging across random effects groups is then more accurate.

Finite mixture models

For finite mixture models (currently, only the brms::mixture() family from package *brms* is supported), use predict = "link" to return predicted values stratified by class membership. To predict the class membership, use predict = "classification". See also this vignette.

Equivalence tests (smallest effect size of interest)

There are two ways of performing equivalence tests with **modelbased**.

- Using the *marginaleffects* machinery
  The first is by specifying the equivalence argument. It takes a numeric vector of length two, defining the lower and upper bounds of the region of equivalence (ROPE). The output then includes an additional column p_Equivalence. A high p-value (non-significant result) means we reject the assumption of practical equivalence (and that a minimal important difference can be assumed, or that the estimate of the predicted value, slope or contrast is likely outside the ROPE).
- Using the equivalence_test() function
  The second option is to use the parameters::equivalence_test.lm() function from the **parameters** package on the output of estimate_means(), estimate_slopes() or estimate_contrasts(). This method is more flexible and implements different "rules" to calculate practical equivalence. Furthermore, the rule decisions of accepting, rejecting, or undecided regarding the null hypothesis of the equivalence test are also provided. Thus, resulting p-values may differ from those p-values returned when using the equivalence argument.

The output from equivalence_test() returns a column SGPV, the "second generation p-value", which is equivalent to the p (Equivalence) column when using the equivalence argument. It is basically representative of the ROPE coverage from the confidence interval of the estimate (i.e. the proportion of the confidence intervals that lies within the region of practical equivalence).

Global Options to Customize Estimation of Marginal Means

- modelbased_backend: options(modelbased_backend = <string>) will set a default value for the backend argument and can be used to set the package used by default to calculate marginal means. Can be "marginaleffects" or "emmeans".
- modelbased_estimate: options(modelbased_estimate = <string>) will set a default value for the estimate argument.

- `modelbased_integer`: `options(modelbased_integer = <value>)` will set the minimum number of unique values in an integer predictor to treat that predictor as a "discrete integer" or as continuous. If the integer has more than `modelbased_integer` unique values, it is treated as continuous. Set to `TRUE` to always treat integer predictors as continuous.

References

Chatton, A. and Rohrer, J.M. 2024. The Causal Cookbook: Recipes for Propensity Scores, G-Computation, and Doubly Robust Standardization. Advances in Methods and Practices in Psychological Science. 2024;7(1). doi:10.1177/25152459241236149

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

See Also

There is a `plot()` method, which is documented here: `visualisation_recipe.estimate_predicted()`

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"))
)

# equivalence test: the null-hypothesis is that the estimate is outside
# the equivalence bounds [-4.5, 4.5]
estimate_means(model, by = "Species", equivalence = c(-4.5, 4.5))

# 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)

# grids for numeric predictors, combine range and length
model <- lm(Sepal.Length ~ Sepal.Width * Petal.Length, data = iris)

# create a "grid": value range for first numeric predictor, mean +/-1 SD
# for remaining numeric predictors.
estimate_means(model, c("Sepal.Width", "Petal.Length"), range = "grid")

# range from minimum to maximum spread over four values,
# and mean +/- 1 SD (a total of three values)
estimate_means(
  model,
  by = c("Sepal.Width", "Petal.Length"),
  range = c("range", "sd"),
  length = c(4, 3)
)

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,
  slope = NULL,
  by = NULL,
  predict = NULL,
  ci = 0.95,
  estimate = NULL,
  transform = NULL,
  p_adjust = "none",
  keep_iterations = FALSE,
  backend = NULL,
  verbose = TRUE,
  trend = NULL,
  ...
)
```

Arguments

`model`	A statistical model.
`slope`, `trend`	A character indicating the name of the variable for which to compute the slopes. To get marginal effects at specific values, use `slope="<variable>"` along with the by argument, e.g. `by="<variable> = c(1, 3, 5)"` (or `by=list(<variable> = c(1, 3, 5))`), or a combination of by and `length`, for instance, `by="<variable>", length=30`. To calculate average marginal effects over a range of values, use `slope="<variable> = seq(1, 3, 0.1)"` or similar, or `slope=list(<variable> = seq(1, 3, 0.1))`), and omit the variable provided in `slope` from the by argument. `trend` is an alias for `slope`, for backward compatibility. See 'Examples'.
`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). `by` can be a character (vector) naming the focal predictors, optionally including representative values or levels at which focal predictors are evaluated (e.g., `by = "x = c(1, 2)"`). When `estimate` is *not* `"average"`, the `by` argument is used to create a "reference grid" or "data grid" with representative values for the focal predictors. In this case, `by` can also be list of named elements. See details in `insight::get_datagrid()` to learn more about how to create data grids for

predictors of interest. For `estimate_means()`, by defaults to `"auto"`, which automatically selects the first focal predictor found in the model. If `by = NULL`, the grand mean of the response is predicted. For frequentist ordinal models (for example from `MASS::polr()`, `ordinal::clm()` or `ordinal::clmm()`), by can also name the response with values, for example `by = c("x", "y = c('low', 'high')")`, or as an element of a list or a column of a data frame, for example `by = list(x = c(1, 2), y = "low")`. `estimate_means()`, `estimate_contrasts()` and `estimate_slopes()` then keep only the rows of these response categories. This filter works with `backend = "emmeans"` and `predict = "prob"` (the default).

`predict` Is passed to the `type` argument in `emmeans::emmeans()` (when `backend = "emmeans"`) or in `marginaleffects::avg_predictions()` (when `backend = "marginaleffects"`). Valid options for `predict` are:

- `backend = "marginaleffects"`: `predict` can be `"response"`, `"link"`, `"inverse_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. The `"inverse_link"` is a special option, comparable to *marginaleffects*' `invlink(link)` option. It will calculate predictions on the link scale and then back-transform to the response scale.
- `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"`). See also this vignette. For ordinal models (e.g. from `MASS::polr()` or `ordinal::clm()`), `estimate_means()`, `estimate_contrasts()` and `estimate_slopes()` also accept the ordinal modes of *emmeans*: `"prob"`, `"cum.prob"`, `"exc.prob"`, `"linear.predictor"`, `"latent"` and `"mean.class"` (see `vignette("models", package = "emmeans")`). For these models, the default is `"prob"`, which returns one probability per response category (column `Response`). The modes `"cum.prob"`, `"exc.prob"` and `"linear.predictor"` return one value per threshold (column `Threshold`). For modes with response categories or thresholds, contrasts compare the focal levels within each one, unless `contrast` names the response. For `estimate_slopes()`, the slopes for `"prob"` and `"mean.class"` are computed from the slopes of the cumulative probabilities.

See also section *Predictions on different scales*.

`ci` Confidence Interval (CI) level. Must be numeric between 0 and 1. Defaults to `0.95`. Use `NULL` to suppress calculation of standard errors and confidence intervals.

`estimate` The `estimate` argument determines how predictions are averaged ("marginalized") over variables not specified in `by` or `contrast` (non-focal predictors). It controls whether predictions represent a "specific" or "typical" individual (*conditional*, *individual-level* predictions), or an "average" individual from the sample, or an "counterfactual" individual from a broader population (*marginal*, *population-level* predictions).

Conditional Predictions based on artificially constructed reference grids

- `"specific"`: The estimated means represent a rather "theoretical" view of the data. A prediction is made for a specific individual from the sample (i.e., a very specific combination of predictor values). This answers the question about the expected average value of the target variable for a *specific* observation.
- `"typical"` (Default): Calculates predictions for a balanced data grid representing all combinations of focal predictor levels (specified in `by`). For non-focal numeric predictors, it uses the mean; for non-focal categorical predictors, it marginalizes (averages) over the levels. This represents a "typical" observation based on the data grid and is useful for comparing groups. It answers: "What would the average outcome be for a 'typical' observation?". This is the default approach when estimating marginal means using the *emmeans* package.

Marginal predictions based on actual empirical observations

- `"average"`: Calculates predictions for each observation in the sample and then averages these predictions within each group defined by the focal predictors. This reflects the sample's actual distribution of non-focal predictors, not a balanced grid. It answers: "What is the predicted value for an average observation in my data?"
- `"population"`: "Clones" each observation, creating copies with all possible combinations of focal predictor levels. It then averages the predictions across these "counterfactual" observations (non-observed permutations) within each group. This extrapolates to a hypothetical broader population, considering "what if" scenarios. It answers: "What is the predicted response for the 'average' observation in a broader possible target population?" This approach mimics a "pseudo-randomization" and hence can be more apt to generalize, which should be used for **G-computation** (causal inference, see *Chatton and Rohrer 2024*). `"counterfactual"` is an alias for `"population"`.

You can set a default option for the `estimate` argument via `options()`, e.g. `options(modelbased_estimate = "average")`.

Note following limitations:

- Especially for continuous predictors, `estimate = "average"` may produce erratic-looking outputs, because it calculates the average based only on the data points that actually exist. This is in particular important for two or more focal predictors, because it doesn't generate a *complete* grid of all theoretical combinations of predictor values. Consequently, the output may not include all the values. To resolve this, you may provide a defined data grid via the `data` argument containing the values of interest.
- For the same reason, *conditional* predictions or `estimate = "population"` typically yield smoother curves in **visualizations** for continuous focal predictors, whereas *marginal* predictions with `estimate = "average"` may introduce noisy visual artifacts.
- Filtering the output at values of continuous predictors, e.g. `by = "x=1:5"`, in combination with `estimate = "average"` may result in returning an empty data frame because the requested values may not overlap with existing data.

In such cases, you can use `estimate = "typical"` or use the `newdata` argument to provide a data grid of predictor values at which to evaluate predictions.

- `estimate = "population"` is not available for `estimate_slopes()`.

`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. Note that no standard errors are returned when transformations are applied.

`p_adjust` The p-values adjustment method for frequentist multiple comparisons, when several slopes are tested, or when slopes are compared. For Johnson-Neyman intervals, i.e. in case of interactions with two numeric predictors (one specified in `slope`, one in `by`), the `"esarey"` or `"sup-t"` options are recommended, but `p_adjust` can also be one of `"none"` (default), `"hochberg"`, `"hommel"`, `"bonferroni"`, `"BH"`, `"BY"`, `"fdr"`, `"tukey"`, `"sidak"`, or `"holm"`. `"sup-t"` computes simultaneous confidence bands, also called sup-t confidence band (Montiel Olea & Plagborg-Møller, 2019). The Tukey adjustment is only appropriate for pairwise comparisons. It is used when slopes are compared pairwise, for example with `comparison = ~pairwise` or `post_process = ~pairwise`. For slopes that are each tested against 0 or against another value (for example `hypothesis = 1`), and for slopes that are not compared pairwise, `"tukey"` is replaced by `"sidak"` (with a message), as `emmeans::test()` does. The Sidak adjustment counts the tests (rows) and the Tukey adjustment the compared slopes, in the whole output or, for comparisons with a group term (for example `comparison = ~pairwise | group`), in each group. In `estimate_slopes()` with `backend = "emmeans"`, only the `stats::p.adjust()` methods, `"tukey"` and `"sidak"` are applied to the slopes, and other methods are ignored.

`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`, and so on. If `keep_iterations` is a positive number, only as many columns as indicated in `keep_iterations` will be added to the output. You can reshape them to a long format by running `bayestestR::reshape_iterations()`.

`backend` Whether to use `"marginaleffects"` (default) or `"emmeans"` 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).

Another difference is that `backend = "marginaleffects"` will be slower than `backend = "emmeans"`. For most models, this difference is negligible. However, in particular complex models or large data sets fitted with *glmmTMB* can be significantly slower.

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(): Arguments such as length, digits or range can be used to control the (number of) representative values. For integer variables, protect_integers modulates whether these should also be treated as numerics, i.e. values can have fractions or not.
- **marginaleffects**: Internally used functions are avg_predictions() for means and contrasts, and avg_slope() for slopes. Therefore, arguments for instance like vcov, equivalence, df, slope, hypothesis or even newdata can be passed to those functions (note that data is supported as an alias for newdata for consistency across the *easystats* ecosystem). A weights argument is passed to the wts argument in avg_predictions() or avg_slopes(), however, weights can only be applied when estimate is "average" or "population" (i.e. for those marginalization options that do not use data grids). Other arguments, such as re.form or allow.new.levels, may be passed to predict() (which is internally used by *marginaleffects*) if supported by that model class.
- **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 bayestestR::describe_post thus, arguments like, for example, centrality, rope_range, or test are passed to that function.
- Especially for estimate_contrasts() with integer focal predictors, for which contrasts should be calculated, use argument integer_as_continuous to set the maximum number of unique values in an integer predictor to treat that predictor as "discrete integer" or as numeric. For the first case, contrasts are calculated between values of the predictor, for the latter, contrasts of slopes are calculated. If the integer has more than integer_as_continuous unique values, it is treated as numeric. Defaults to 5. Set to TRUE to always treat integer predictors as continuous.
- For count regression models that use an offset term, use offset = <value> to fix the offset at a specific value. Or use estimate = "average" or estimate = "population" without specifying the offset, to average predictions over the distribution of the offset (if appropriate).

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_relation()). 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 the 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`.

Predictions and contrasts at meaningful values (data grids)

To define representative values for focal predictors (specified in `by`, `contrast`, and `slope`), you can use several methods. These values are internally generated by `insight::get_datagrid()`, so consult its documentation for more details.

- You can directly specify values as strings or lists for `by`, `contrast`, and `slope`.
  - For numeric focal predictors, use examples like `by = "gear = c(4, 8)"`, `by = list(gear = c(4, 8))`, `by = "gear = 5:10"` or `by = list(gear = 5:10)`
  - For factor or character predictors, use `by = "Species = c('setosa', 'virginica')"` or `by = list(Species = c('setosa', 'virginica'))`
- You can use "shortcuts" within square brackets, such as `by = "Sepal.Width = [sd]"` or `by = "Sepal.Width = [fivenum]"`

- For numeric focal predictors, if no representative values are specified (i.e., by = "gear" and *not* by = "gear = c(4, 8)"), length and range control the number and type of representative values for the focal predictors:
  - length determines how many equally spaced values are generated.
  - range specifies the type of values, like "range" or "sd".
  - length and range apply to all numeric focal predictors.
  - If you have multiple numeric predictors, length and range can accept multiple elements, one for each predictor (see 'Examples').
- For integer variables, only values that appear in the data will be included in the data grid, independent from the length argument. This behaviour can be changed by setting protect_integers = FALSE, which will then treat integer variables as numerics (and possibly produce fractions).

See also this vignette for some examples, and insight::get_datagrid() to learn more about how to create data grids for predictors of interest.

References

Montiel Olea, J. L., and Plagborg-Møller, M. (2019). Simultaneous confidence bands: Theory, implementation, and an application to SVARs. Journal of Applied Econometrics, 34(1), 1–17. doi:10.1002/jae.2656

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, slope = "Petal.Length", by = "Species")
slopes

# What is the *average* slope of Petal.Length? This can be calculated by
# taking the average of the slopes across all Species, using `comparison`.
# We pass a function to `comparison` that calculates the mean of the slopes.
estimate_slopes(
  model,
  slope = "Petal.Length",
  by = "Species",
  comparison = ~I(mean(x))
)

## 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,
  slope = "Petal.Length",
  by = c("Petal.Length", "Species"), length = 20
)
summary(slopes)
plot(slopes)

# marginal effects, grouped by Species, at different values of Petal.Length
estimate_slopes(model,
  slope = "Petal.Length",
  by = c("Petal.Length", "Species"), length = 10
)

# marginal effects at different values of Petal.Length
estimate_slopes(model, slope = "Petal.Length", by = "Petal.Length", length = 10)

# marginal effects at very specific values of Petal.Length
estimate_slopes(model, slope = "Petal.Length", by = "Petal.Length=c(1, 3, 5)")
# or in "classic" R syntax, using lists
estimate_slopes(model, slope = "Petal.Length", by = list(Petal.Length = c(1, 3, 5)))

# average marginal effects of Petal.Length,
# just for the trend within a certain range
estimate_slopes(model, slope = "Petal.Length=seq(2, 4, 0.01)")
# "classic" R syntax, using lists
estimate_slopes(model, slope = list(Petal.Length = seq(2, 4, 0.01)))

## End(Not run)


## Not run:
# marginal effects with different `estimate` options
data(penguins, package = "datasets")
penguins$long_bill <- factor(datawizard::categorize(penguins$bill_len), labels = c("short", "long"))
m <- glm(long_bill ~ sex + species + island * bill_dep, data = penguins, family = "binomial")

# the emmeans default
estimate_slopes(m, "bill_dep", by = "island")
emmeans::emtrends(m, "island", var = "bill_dep", regrid = "response")

# the marginaleffects default
estimate_slopes(m, "bill_dep", by = "island", estimate = "average")
marginaleffects::avg_slopes(m, variables = "bill_dep", by = "island")

# filter by groups
estimate_slopes(m, "bill_dep", by = "island=c('Biscoe','Dream')")
```

```
# or in "classic" R syntax, using lists
estimate_slopes(m, "bill_dep", by = list(island = c('Biscoe', 'Dream')))

## 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 = "revpairwise",
  keep_iterations = FALSE,
  verbose = TRUE,
  ...
)

get_emmeans(
  model,
  by = "auto",
  predict = NULL,
  keep_iterations = FALSE,
  verbose = TRUE,
  ...
```

```
)

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

get_marginalcontrasts(
  model,
  contrast = NULL,
  by = NULL,
  predict = NULL,
  ci = 0.95,
  comparison = "pairwise",
  estimate = NULL,
  transform = NULL,
  post_process = NULL,
  p_adjust = "none",
  iterations = NULL,
  keep_iterations = FALSE,
  verbose = TRUE,
  ...
)

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

get_marginaltrends(
  model,
  trend = NULL,
  by = NULL,
  predict = NULL,
  ci = 0.95,
```

```
  estimate = NULL,
  transform = NULL,
  p_adjust = "none",
  iterations = NULL,
  keep_iterations = FALSE,
  verbose = TRUE,
  ...
)
```

Arguments

model	A statistical model.
contrast	A character vector indicating the name of the variable(s) for which to compute the contrasts, optionally including representative values or levels at which contrasts are evaluated (e.g., `contrast="x=c('a','b')"`). **Note:** It is also possible to contrast average slopes, i.e. `contrast` can be the name of two numeric predictors. However, while it is possible to filter data for one numeric contrast (e.g., `contrast = c("num_pred=c(0, 1, 3)")`), it is not possible to "filter" at certain values for two numeric predictors. For contrasting slopes, the `comparison` will always be `"pairwise"`. It is possible to compute pairwise comparisons of two average slopes at the levels of a third variable, by also adding that variable to the `contrast` argument, e.g. `contrast = c("num1", "num2", "factor")`. See 'Examples'. For frequentist ordinal models and `backend = "emmeans"`, `contrast` can name the response with two or more categories, for example `contrast = "y = c('low', 'high')"`, to contrast only these response categories (with `predict = "prob"`, the default).
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). `by` can be a character (vector) naming the focal predictors, optionally including representative values or levels at which focal predictors are evaluated (e.g., `by = "x = c(1, 2)"`). When `estimate` is *not* `"average"`, the `by` argument is used to create a "reference grid" or "data grid" with representative values for the focal predictors. In this case, `by` can also be list of named elements. See details in `insight::get_datagrid()` to learn more about how to create data grids for predictors of interest. For `estimate_means()`, `by` defaults to `"auto"`, which automatically selects the first focal predictor found in the model. If `by = NULL`, the grand mean of the response is predicted. For frequentist ordinal models (for example from `MASS::polr()`, `ordinal::clm()` or `ordinal::clmm()`), `by` can also name the response with values, for example `by = c("x", "y = c('low', 'high')")`, or as an element of a list or a column of a data frame, for example `by = list(x = c(1, 2), y = "low")`. `estimate_means()`, `estimate_contrasts()` and `estimate_slopes()` then keep only the rows of these response categories. This filter works with `backend = "emmeans"` and `predict = "prob"` (the default).
predict	Is passed to the `type` argument in `emmeans::emmeans()` (when `backend = "emmeans"`) or in `marginaleffects::avg_predictions()` (when `backend = "marginaleffects"`). Valid options for `predict` are:

- backend = "marginaleffects": predict can be "response", "link", "inverse_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. The "inverse_link" is a special option, comparable to *marginaleffects*' invlink(link) option. It will calculate predictions on the link scale and then back-transform to the response scale.
- 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"). See also this vignette. For ordinal models (e.g. from MASS::polr() or ordinal::clm()), estimate_means(), estimate_contrasts() and estimate_slopes() also accept the ordinal modes of *emmeans*: "prob", "cum.prob", "exc.prob", "linear.predictor", "latent" and "mean.class" (see vignette("models", package = "emmeans")). For these models, the default is "prob", which returns one probability per response category (column Response). The modes "cum.prob", "exc.prob" and "linear.predictor" return one value per threshold (column Threshold). For modes with response categories or thresholds, contrasts compare the focal levels within each one, unless contrast names the response. For estimate_slopes(), the slopes for "prob" and "mean.class" are computed from the slopes of the cumulative probabilities.

See also section *Predictions on different scales*.

comparison Specify the type of contrasts or tests that should be carried out. See also section *Comparison options* below for details.

- When backend = "emmeans", can be one of "pairwise", "revpairwise", "poly", "consec", "eff", "del.eff", "mean_chg", "trt.vs.ctrl", "dunnett", "wtcon" and some more. Defaults to "revpairwise", to return consistent results (regarding the sign) with the default backend = "marginaleffects". To test multiple hypotheses jointly (usually used for factorial designs), comparison can also be "joint". 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. For options not described below, see documentation of marginaleffects::comparisons, this website and section *Comparison options* below.
  - String: One of "pairwise", "ratio", "revpairwise", "reference", "sequential", "meandev" "meanotherdev", "poly", "helmert", or "trt_vs_ctrl". To test multiple hypotheses jointly (usually used for factorial designs), comparison can also be "joint" or "omnibus". In this case, use the test argument to specify which test should be conducted: "F" (default) or "Chi2", and use argument null to specify the null-hypothesis to test against. For "omnibus", null is set to the overall mean or prevalence/proportion.
  - String: Special string options are "inequality", "inequality_ratio", and "inequality_pairwise". comparison = "inequality" computes

the marginal effect inequality summary of categorical predictors' overall effects, respectively, the comprehensive effect of an independent variable across all outcome categories of a nominal or ordinal dependent variable (also called *absolute inequality*, or total marginal effect, see *Mize and Han, 2025*). `"inequality_ratio"` computes the ratio of marginal effect inequality measures, also known as *relative inequality*. This is useful to compare the relative effects of different predictors on the dependent variable. It provides a measure of how much more or less inequality one predictor has compared to another. `comparison = "inequality_pairwise"` computes pairwise differences of absolute inequality measures, while `"inequality_ratio_pairwise"` computes pairwise differences of relative inequality measures (ratios). See an overview of applications in the related case study in the vignettes.

- 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`, `pairwise`, `sequential`, `meandev`, etc., see string-options). Optionally, comparisons can be carried out within subsets by indicating the grouping variable after a vertical bar (`|`). If the left-hand side is missing, it defaults to `difference` (i.e. `comparison = ~pairs | group` is identical to `comparison = difference ~ pairs | group`).
- A custom function, e.g. `comparison = I(my_fun(x))`, or `<comparison> ~ I(my_fun(x)) | gr` (where `<comparison>` can be `difference` or `ratio`, or skipped).
- If contrasts should be calculated (or grouped by) factors, `comparison` can also be a matrix that specifies factor contrasts (see 'Examples').

`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`, and so on. If `keep_iterations` is a positive number, only as many columns as indicated in `keep_iterations` will be added to the output. You can reshape them to a long format by running `bayestestR::reshape_iterations()`.

`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()`: Arguments such as `length`, `digits` or `range` can be used to control the (number of) representative values. For integer variables, `protect_integers` modulates whether these should also be treated as numerics, i.e. values can have fractions or not.
- **marginaleffects**: Internally used functions are `avg_predictions()` for means and contrasts, and `avg_slope()` for slopes. Therefore, arguments for instance like `vcov`, `equivalence`, `df`, `slope`, `hypothesis` or even `newdata` can be passed to those functions (note that `data` is supported as an alias for

newdata for consistency across the *easystats* ecosystem). A weights argument is passed to the wts argument in avg_predictions() or avg_slopes(), however, weights can only be applied when estimate is "average" or "population" (i.e. for those marginalization options that do not use data grids). Other arguments, such as re.form or allow.new.levels, may be passed to predict() (which is internally used by *marginaleffects*) if supported by that model class.

- **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 bayestestR::describe_post thus, arguments like, for example, centrality, rope_range, or test are passed to that function.
- Especially for estimate_contrasts() with integer focal predictors, for which contrasts should be calculated, use argument integer_as_continuous to set the maximum number of unique values in an integer predictor to treat that predictor as "discrete integer" or as numeric. For the first case, contrasts are calculated between values of the predictor, for the latter, contrasts of slopes are calculated. If the integer has more than integer_as_continuous unique values, it is treated as numeric. Defaults to 5. Set to TRUE to always treat integer predictors as continuous.
- For count regression models that use an offset term, use offset = <value> to fix the offset at a specific value. Or use estimate = "average" or estimate = "population" without specifying the offset, to average predictions over the distribution of the offset (if appropriate).

trend — A character indicating the name of the variable for which to compute the slopes. To get marginal effects at specific values, use trend="<variable>" along with the by argument, e.g. by="<variable> = c(1, 3, 5)", or a combination of by and length, for instance, by="<variable>", length=30. To calculate average marginal effects over a range of values, use trend="<variable> = seq(1, 3, 0.1)" (or similar) and omit the variable provided in trend from the by argument.

ci — Confidence Interval (CI) level. Must be numeric between 0 and 1. Defaults to 0.95. Use NULL to suppress calculation of standard errors and confidence intervals.

estimate — The estimate argument determines how predictions are averaged ("marginalized") over variables not specified in by or contrast (non-focal predictors). It controls whether predictions represent a "specific" or "typical" individual (*conditional*, *individual-level* predictions), or an "average" individual from the sample, or an "counterfactual" individual from a broader population (*marginal*, *population-level* predictions).

Conditional Predictions based on artificially constructed reference grids

- "specific": The estimated means represent a rather "theoretical" view of the data. A prediction is made for a specific individual from the sample (i.e., a very specific combination of predictor values). This answers the question about the expected average value of the target variable for a *specific* observation.

- `"typical"` (Default): Calculates predictions for a balanced data grid representing all combinations of focal predictor levels (specified in `by`). For non-focal numeric predictors, it uses the mean; for non-focal categorical predictors, it marginalizes (averages) over the levels. This represents a "typical" observation based on the data grid and is useful for comparing groups. It answers: "What would the average outcome be for a 'typical' observation?". This is the default approach when estimating marginal means using the *emmeans* package.

Marginal predictions based on actual empirical observations

- `"average"`: Calculates predictions for each observation in the sample and then averages these predictions within each group defined by the focal predictors. This reflects the sample's actual distribution of non-focal predictors, not a balanced grid. It answers: "What is the predicted value for an average observation in my data?"
- `"population"`: "Clones" each observation, creating copies with all possible combinations of focal predictor levels. It then averages the predictions across these "counterfactual" observations (non-observed permutations) within each group. This extrapolates to a hypothetical broader population, considering "what if" scenarios. It answers: "What is the predicted response for the 'average' observation in a broader possible target population?" This approach mimics a "pseudo-randomization" and hence can be more apt to generalize, which should be used for **G-computation** (causal inference, see *Chatton and Rohrer 2024*). `"counterfactual"` is an alias for `"population"`.

You can set a default option for the `estimate` argument via `options()`, e.g. `options(modelbased_estimate = "average")`.

Note following limitations:

- Especially for continuous predictors, `estimate = "average"` may produce erratic-looking outputs, because it calculates the average based only on the data points that actually exist. This is in particular important for two or more focal predictors, because it doesn't generate a *complete* grid of all theoretical combinations of predictor values. Consequently, the output may not include all the values. To resolve this, you may provide a defined data grid via the `data` argument containing the values of interest.
- For the same reason, *conditional* predictions or `estimate = "population"` typically yield smoother curves in **visualizations** for continuous focal predictors, whereas *marginal* predictions with `estimate = "average"` may introduce noisy visual artifacts.
- Filtering the output at values of continuous predictors, e.g. `by = "x=1:5"`, in combination with `estimate = "average"` may result in returning an empty data frame because the requested values may not overlap with existing data. In such cases, you can use `estimate = "typical"` or use the `newdata` argument to provide a data grid of predictor values at which to evaluate predictions.
- `estimate = "population"` is not available for `estimate_slopes()`.

`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. Note that no standard errors are returned when transformations are applied.

`post_process` Optional formula, character string or function (see `comparison`), or a list of formulas, strings or functions, to process subsequent, multi-step comparisons. After the initial comparison in `comparison` is completed, the results are then post-processed using the specified post-process tests. See 'Examples'.

`p_adjust` The p-values adjustment method for frequentist multiple comparisons, when several slopes are tested, or when slopes are compared. For Johnson-Neyman intervals, i.e. in case of interactions with two numeric predictors (one specified in `slope`, one in `by`), the `"esarey"` or `"sup-t"` options are recommended, but `p_adjust` can also be one of `"none"` (default), `"hochberg"`, `"hommel"`, `"bonferroni"`, `"BH"`, `"BY"`, `"fdr"`, `"tukey"`, `"sidak"`, or `"holm"`. `"sup-t"` computes simultaneous confidence bands, also called sup-t confidence band (Montiel Olea & Plagborg-Møller, 2019). The Tukey adjustment is only appropriate for pairwise comparisons. It is used when slopes are compared pairwise, for example with `comparison = ~pairwise` or `post_process = ~pairwise`. For slopes that are each tested against 0 or against another value (for example `hypothesis = 1`), and for slopes that are not compared pairwise, `"tukey"` is replaced by `"sidak"` (with a message), as `emmeans::test()` does. The Sidak adjustment counts the tests (rows) and the Tukey adjustment the compared slopes, in the whole output or, for comparisons with a group term (for example `comparison = ~pairwise | group`), in each group. In `estimate_slopes()` with `backend = "emmeans"`, only the `stats::p.adjust()` methods, `"tukey"` and `"sidak"` are applied to the slopes, and other methods are ignored.

`iterations` This argument has two distinct effects, depending on the context. When `effectsize = "boot"`, it defines the number of bootstrap resamples used to calculate the effect size (via `bootES::bootES()`), and defaults to `200` in this case. For Bayesian models used in `estimate_means()`, `estimate_slopes()`, or `estimate_contrasts()` (and when `backend = "marginaleffects"`), it is passed to the `ndraws` argument of the related `marginaleffects` functions, and defines the number of posterior draws to sample from. If `NULL` (the default), all draws are used. Note that `ndraws` currently only has an effect for Bayesian models fit with **brms** or **MCMCglmm**.

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)
# Or both
```

```
get_emcontrasts(model, contrast = c("Species", "Petal.Width"), length = 2)
# Or with custom specifications
get_emcontrasts(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"))
```

`modelbased-options` *Global options from the modelbased package*

Description

Global options from the modelbased package

Global options to set defaults for function arguments

For calculating marginal means

- `options(modelbased_backend = <string>)` will set a default value for the `backend` argument and can be used to set the package used by default to calculate marginal means. Can be `"marginaleffects"` or `"emmeans"`.
- `options(modelbased_estimate = <string>)` will set a default value for the `estimate` argument, which modulates the type of target population predictions refer to.
- `options(modelbased_integer = <value>)` will set the minimum number of unique values in an integer predictor to treat that predictor as a "discrete integer" or as continuous. If the integer has more than `modelbased_integer` unique values, it is treated as continuous. Set to `TRUE` to always treat integer predictors as continuous.

For printing

- `options(modelbased_select = <string>)` will set a default value for the `select` argument and can be used to define a custom default layout for printing.
- `options(modelbased_include_grid = TRUE)` will set a default value for the `include_grid` argument and can be used to include data grids in the output by default or not.

- `options(modelbased_full_labels = FALSE)` will remove redundant (duplicated) labels from rows.
- `options(easystats_display_format = <value>)` will set the default format for the `display()` methods. Can be one of `"markdown"`, `"html"`, or `"tt"`. See `display.estimate_contrasts()` for details.
- `options(easystats_ci_separator = <value>)` defines the separator string used between the lower and upper CI limits. Defaults to `", "`, resulting in an output like `[0.3, 0.8]`.
- `options(easystats_ci_brackets = <value>)` determines whether brackets or parentheses are used to enclose formatted CIs. Can be `FALSE` to omit them, `TRUE` to include them (defaults to `[` and `]` for text output), or a character vector of length 2 to define custom opening and closing symbols (e.g., `c("(", ")")`).

For plotting

- `options(modelbased_join_dots = <logical>)` will set a default value for the `join_dots`.
- `options(modelbased_numeric_as_discrete = <number>)` will set a default value for the `modelbased_numeric_as_discrete` argument. Can also be `FALSE`.
- `options(modelbased_ribbon_alpha = <number>)` will set a default value for the `alpha` argument of the `ribbon` geom. Should be a number between `0` and `1`.
- `options(modelbased_tinyplot_dodge = <number>)` will set a default value for the `dodge` argument (spacing between geoms) when using `tinyplot::plt()`. Should be a number between `0` and `1`.

`plot.estimate_predicted` *Automated plotting for 'modelbased' objects*

Description

Most **modelbased** objects can be visualized using either the `plot()` function, which internally calls the `visualisation_recipe()` function and relies on `{ggplot2}`. There is also a `tinyplot()` method, which uses the `{tinyplot}` package and relies on the core R graphic system. See the examples below for more information and examples on how to create and customize plots.

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.

Usage

```
## S3 method for class 'estimate_predicted'
plot(x, ...)
```

```
## S3 method for class 'estimate_means'
plot(x, ...)

## S3 method for class 'estimate_means'
tinyplot(
  x,
  type = NULL,
  dodge = NULL,
  show_data = FALSE,
  collapse_group = NULL,
  numeric_as_discrete = NULL,
  colors = NULL,
  size_title = NULL,
  size_axis_title = NULL,
  size_axis_text = NULL,
  size_point = NULL,
  size_line = NULL,
  ...
)

## S3 method for class 'estimate_predicted'
visualisation_recipe(
  x,
  show_data = FALSE,
  show_residuals = FALSE,
  collapse_group = NULL,
  point = NULL,
  line = NULL,
  pointrange = NULL,
  ribbon = NULL,
  facet = NULL,
  grid = NULL,
  join_dots = NULL,
  numeric_as_discrete = NULL,
  ...
)

## 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.
...	Arguments passed from plot() to visualisation_recipe(), or to tinyplot() if you use that method.
type	The type of tinyplot visualization. It is recommended that users leave as NULL (the default), in which case the plot type will be determined automatically by the underlying modelbased object.
dodge	Dodge value for grouped plots. If NULL (the default), then the dodging behavior is determined by the number of groups and getOption("modelbased_tinyplot_dodge").
show_data	Logical, if TRUE, display the "raw" data as a background to the model-based estimation. For mixed models, you can additionally use the collapse_group argument to "collapse" data points by random effects grouping factors. Argument show_data will be ignored for plotting objects returned by estimate_slopes() or estimate_grouplevel().
collapse_group	This argument only takes effect when either show_data or show_residuals is TRUE. For mixed effects models, name of the grouping variable of random effects. If collapse_group = TRUE, data points "collapsed" by the first random effect groups are added to the plot. Else, if collapse_group is a name of a group factor, data is collapsed by that specific random effect. See collapse_by_group() for further details.
numeric_as_discrete	Maximum number of unique values in a numeric predictor to treat that predictor as discrete. Defaults to 8. Numeric predictors are usually mapped to a continuous color scale, unless they have only few unique values. In the latter case, numeric predictors are assumed to represent "categories", e.g. when only the mean value and +/- 1 standard deviation around the mean are chosen as representative values for that predictor. Use FALSE to always use continuous color scales for numeric predictors. It is possible to set a global default value using options(), e.g. options(modelbased_numeric_as_discrete = 10).
colors	Colors or color palette used for plotting. Following options are allowed: • A string corresponding to one of the many palettes listed by either palette.pals() or hcl.pals(). • The palette.colors() function, e.g. palette.colors(palette = "Okabe-Ito", alpha = 0.5).

- A vector or list of colours, e.g. `c("darkorange", "purple", "cyan4")`. If too few colours are provided, they will be recycled (for discrete palettes) or a gradient palette will be interpolated for continuous palettes. If the `palette` argument is used, `colors` will be ignored.

`size_title`, `size_axis_title`, `size_axis_text`
: Numeric, set the size of plot title, axis title or axis labels. If not `NULL`, `par()` is called temporarily to set `cex.main`, `cex.axis` and `cex.lab`. The original values are restored afterwards. The default size is `1`. Larger values increase text sizes and vice versa.

`size_point`, `size_line`
: Size of points and lines in the plot. Default is `1`. Larger values increase point/line sizes and vice versa. If argument `cex` is used, `size_point` will be ignored. Same for argument `lwd`, which overrides `size_line`.

`show_residuals`
: Logical, if `TRUE`, display residuals of the model as a background to the model-based estimation. Residuals will be computed for the predictors in the data grid, using `residualize_over_grid()`. For mixed models, you can additionally use the `collapse_group` argument to "collapse" data points from residuals by random effects grouping factors.

`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` (default), only dots and error bars are shown. It is possible to set a global default value using `options()`, e.g. `options(modelbased_join_dots = TRUE)`.

Details

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"`.

For the `tinyplot()` method, the x-axis automatically adjusts its limits when categorical predictors are used (by setting `xlim` to `c(0.5, n + 0.5)`, the geoms are moved closer together, resulting in a more compact appearance). If appearance is too compact, specify different values for `xlim`, for instance, `xlim = c(1, n)` (where `n` is the number of unique categories).

Value

An object of class `visualisation_recipe` that describes the layers used to create a plot based on `{ggplot2}`. The related `plot()` method is in the `{see}` package. For `tinyplot()` (or its alias `plt()`), a base graphics R plot is created, which can be modified further using classic R graphics code.

Global Options to Customize Plots

Some arguments for `plot()` can get global defaults using `options()`:

- `modelbased_join_dots`: `options(modelbased_join_dots = <logical>)` will set a default value for the `join_dots`.
- `modelbased_numeric_as_discrete`: `options(modelbased_numeric_as_discrete = <number>)` will set a default value for the `modelbased_numeric_as_discrete` argument. Can also be `FALSE`.
- `modelbased_ribbon_alpha`: `options(modelbased_ribbon_alpha = <number>)` will set a default value for the `alpha` argument of the `ribbon` geom. Should be a number between `0` and `1`.
- `modelbased_tinyplot_dodge`: `options(modelbased_tinyplot_dodge = <number>)` will set a default value for the dodge argument (spacing between geoms) when using `tinyplot::plt()`. Should be a number between `0` and `1`.

Examples

```
# ==============================================
# tinyplot
# ==============================================

library(tinyplot)
data(efc, package = "modelbased")
efc <- datawizard::to_factor(efc, c("e16sex", "c172code", "e42dep"))
m <- lm(neg_c_7 ~ e16sex + c172code + barthtot, data = efc)

em <- estimate_means(m, "c172code")
plt(em)

# pass additional tinyplot arguments for customization, e.g.
plt(em, theme = "classic")
plt(em, theme = "classic", flip = TRUE)
# etc.

# Aside: use tinyplot::tinytheme() to set a persistent theme
tinytheme("classic")

# continuous variable example
em <- estimate_means(m, "barthtot")
plt(em)

# grouped example
m <- lm(neg_c_7 ~ e16sex * c172code + e42dep, data = efc)
em <- estimate_means(m, c("e16sex", "c172code"))
plt(em)

# use plt_add (alias tinyplot_add) to add layers
plt_add(type = "l", lty = 2)

# Reset to default theme
tinytheme()

# facets
# ----------------------
```

```
# when using facets, the `tinyplot()` method checks if the legend is
# redundant (because it already appears in the facets), and if so, it
# removes the legend. Set `legend = TRUE` to add it back.

data(efc, package = "modelbased")
# convert to factors, assign labels. we use datawizard::to_factor() in
# the second row to automatically assign value labels as factor levels.
# because labels are too long for `c172code`, we assign new labels using
# `as.factor()`
efc$c172code <- factor(efc$c172code, labels = c("low", "mid", "high"))
efc$e42dep <- datawizard::to_factor(efc$e42dep)
# fit model
m <- lm(neg_c_7 ~ c172code * e42dep, data = efc)
em <- estimate_means(m, c("c172code", "e42dep"))

# for facets, it can be useful to remove dodging
plt(em, facet = ~e42dep, dodge = 0, theme = "float")

# remove x-axis limits adjustments with `xlim`
plt(
  em,
  facet = ~e42dep,
  dodge = 0,
  theme = "float",
  xlim = c(1, 3),
  grid = TRUE
)

# add back legend
plt(em, facet = ~e42dep, legend = TRUE)



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:
# It 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)

# ============================================
# Adding original data to the plot
# ============================================
data(efc, package = "modelbased")
efc$e15relat <- as.factor(efc$e15relat)
efc$c161sex <- as.factor(efc$c161sex)
levels(efc$c161sex) <- c("male", "female")
model <- lme4::lmer(neg_c_7 ~ c161sex + (1 | e15relat), data = efc)

me <- estimate_means(model, "c161sex")
plot(me, show_data = TRUE)

# data points: collapse by / average over random effects groups -------
plot(me, show_data = TRUE, collapse_group = "e15relat")

## 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)
```

`pool_contrasts` *Pool contrasts and comparisons from* `estimate_contrasts()`

Description

This function "pools" (i.e. combines) multiple `estimate_contrasts` objects, returned by `estimate_contrasts()`, in a similar fashion as `mice::pool()`.

Usage

```
pool_contrasts(x, ...)
```

Arguments

`x`	A list of `estimate_contrasts` objects, as returned by `estimate_contrasts()`.
`...`	Currently not used.

Details

Averaging of parameters follows Rubin's rules (*Rubin, 1987, p. 76*).

Value

A data frame with pooled comparisons or contrasts of predictions.

References

Rubin, D.B. (1987). Multiple Imputation for Nonresponse in Surveys. New York: John Wiley and Sons.

Examples

```
data("nhanes2", package = "mice")
imp <- mice::mice(nhanes2, printFlag = FALSE)
comparisons <- lapply(1:5, function(i) {
  m <- lm(bmi ~ age + hyp + chl, data = mice::complete(imp, action = i))
  estimate_contrasts(m, "age")
})
pool_contrasts(comparisons)
```

---

pool_predictions *Pool Predictions and Estimated Marginal Means*

Description

This function "pools" (i.e. combines) multiple `estimate_means` objects, in a similar fashion as `mice::pool()`.

Usage

```
pool_predictions(x, transform = NULL, ...)

pool_slopes(x, transform = NULL, ...)
```

Arguments

`x`	A list of `estimate_means` objects, as returned by `estimate_means()`, or `estimate_predicted` objects, as returned by `estimate_relation()` and related functions. For `pool_slopes()`, must be a list of `estimate_slopes` objects, as returned by `estimate_slopes()`.
`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. Note that no standard errors are returned when transformations are applied.

... Currently not used.

Details

Averaging of parameters follows Rubin's rules (*Rubin, 1987, p. 76*). Pooling is applied to the predicted values and based on the standard errors as they are calculated in the `estimate_means` or `estimate_predicted` objects provided in `x`. For objects of class `estimate_means`, the predicted values are on the response scale by default, and standard errors are calculated using the delta method. Then, pooling estimates and calculating standard errors for the pooled estimates based ob Rubin's rule is carried out. There is no back-transformation to the link-scale of predicted values before applying Rubin's rule.

Value

A data frame with pooled predictions.

References

Rubin, D.B. (1987). Multiple Imputation for Nonresponse in Surveys. New York: John Wiley and Sons.

Examples

```
# example for multiple imputed datasets
data("nhanes2", package = "mice")
imp <- mice::mice(nhanes2, printFlag = FALSE)

# estimated marginal means
predictions <- lapply(1:5, function(i) {
  m <- lm(bmi ~ age + hyp + chl, data = mice::complete(imp, action = i))
  estimate_means(m, "age")
})
pool_predictions(predictions)

# estimated slopes (marginal effects)
slopes <- lapply(1:5, function(i) {
  m <- lm(bmi ~ age + hyp + chl, data = mice::complete(imp, action = i))
  estimate_slopes(m, "chl")
})
pool_slopes(slopes)
```

---

puppy_love *More puppy therapy data*

Description

Fictitious data related to whether puppy therapy works when you adjust for a person's love of puppies, taken from the {discovr} package (Field 2025)

Details

Following variables are included in the dataset:

- `id`: Participant id
- `dose`: Treatment group to which the participant was randomly assigned (No puppies (control), 15 minutes of puppy therapy, 30 minutes of puppy therapy)
- `happiness`: Self-reported happiness from 0 (as unhappy as I can possibly imagine being) to 10 (as happy as I can possibly imagine being)
- `puppy_love`: Self-reported love of puppies from 0 (I am a weird person who hates puppies, please be deeply suspicious of me) to 7 (puppies are the best thing ever, one day I might marry one)

For further details, see `?discovr::puppy_love`.

References

Field, A. P. (2025). Discovering statistics using R and RStudio (2nd ed.). London: Sage.

`residualize_over_grid` *Compute partial residuals from a data grid*

Description

This function computes partial residuals based on a data grid, where the data grid is usually a data frame from all combinations of factor variables or certain values of numeric vectors. This data grid is usually used as `newdata` argument in `predict()`, and can be created with `insight::get_datagrid()`.

Usage

```
residualize_over_grid(grid, model, ...)

## S3 method for class 'data.frame'
residualize_over_grid(grid, model, predictor_name, ...)
```

Arguments

`grid`	A data frame representing the data grid, or an object of class `estimate_means` or `estimate_predicted`, as returned by the different `estimate_*()` functions.
`model`	The model for which to compute partial residuals. The data grid `grid` should match to predictors in the model.
`...`	Currently not used.
`predictor_name`	The name of the focal predictor, for which partial residuals are computed.

Value

A data frame with residuals for the focal predictor.

Partial Residuals

For **generalized linear models** (glms), residualized scores are computed as `inv.link(link(Y) + r)` where `Y` are the predicted values on the response scale, and `r` are the *working* residuals.

For (generalized) linear **mixed models**, the random effect are also partialled out.

References

Fox J, Weisberg S. Visualizing Fit and Lack of Fit in Complex Regression Models with Predictor Effect Plots and Partial Residuals. Journal of Statistical Software 2018;87.

Examples

```
set.seed(1234)
x1 <- rnorm(200)
x2 <- rnorm(200)
# quadratic relationship
y <- 2 * x1 + x1^2 + 4 * x2 + rnorm(200)

d <- data.frame(x1, x2, y)
model <- lm(y ~ x1 + x2, data = d)

pr <- estimate_means(model, c("x1", "x2"))
head(residualize_over_grid(pr, model))
```

`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")
```

---

stigma *Sample dataset from patients with somatic symptoms*

Description

An artificial, anonymized data set based on a project related to the development and persistence of somatic symptoms.

The data consists of nine variables from 1140 observations (380 patients at three time points):

- `ID`: A unique identifier for each participant
- `disease_group`: Classification of main disease from patient, e.g. chronic kidney disease (CKD), Ulcerative colitis, Irritable Bowel Syndrome (IBS) and others.
- `phq15`: Total score of the PHQ-15 questionnaire (Patient Health Questionnaire-15), measuring the somatic symptoms severity
- `symptoms_unreal`: Question regarding perceived stigma related to somatic symptoms, asking "People believe that my symptoms are not a real illness". Answer categories ranged from `"strongly disagree"`, `"disagree"`, `"neutral"`, `"agree"`, and `"strongly agree"`. `"I have no complaints"` was the residual category for patients without specific somatic symptoms.
- `time3`: Measurement point (baseline, and two follow-ups)
- `education`: Educational level, measured according to the CASMIN classification.
- `sex`: The participant's sex
- `age_z`: Age of patient, standardized

---

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")

modelbased::zero_crossings(x)
modelbased::find_inversions(x)
```

Index

∗ **data**
coffee_data, 4
efc, 9
fish, 51
puppy_love, 71
stigma, 74

as.data.frame.estimate_contrasts, 3

bayestestR::describe_posterior(), *10*, *37*, *47*, *55*
bayestestR::reshape_iterations(), *15*, *27*, *37*, *46*, *55*
bootES::bootES, *19*
bootES::bootES(), *15*, *58*
brms::mixture(), *29*, *40*

coffee_data, 4
collapse_by_group, 4
collapse_by_group(), *63*

data.frame, *3*
describe_nonlinear, 5
display.estimate_contrasts, 6
display.estimate_contrasts(), *61*

efc, 9
emmeans::contrast, *12*, *54*
emmeans::eff_size, *19*
emmeans::emmeans(), *16*, *38*, *47*
emmeans::emtrends(), *16*, *38*, *47*
estimate_contrasts, 9
estimate_contrasts(), *16*, *33*, *38*, *43*, *47*, *48*, *69*
estimate_expectation, 24
estimate_expectation(), *16*, *38*, *48*
estimate_grouplevel, 30
estimate_link *(*estimate_expectation*)*, 24
estimate_means, 33
estimate_means(), *9*, *16*, *31*, *38*, *43*, *47*, *48*, *70*
estimate_prediction *(*estimate_expectation*)*, 24
estimate_relation *(*estimate_expectation*)*, 24
estimate_relation(), *6*, *16*, *38*, *47*, *70*
estimate_slopes, 42
estimate_slopes(), *9*, *16*, *17*, *31*, *33*, *38*, *47*, *48*, *70*
estimate_smooth *(*describe_nonlinear*)*, 5

find_inversions *(*zero_crossings*)*, 74
fish, 51
format.estimate_contrasts *(*display.estimate_contrasts*)*, 6

get_emcontrasts, 51
get_emmeans *(*get_emcontrasts*)*, 51
get_emtrends *(*get_emcontrasts*)*, 51
get_marginalcontrasts *(*get_emcontrasts*)*, 51
get_marginalmeans *(*get_emcontrasts*)*, 51
get_marginaltrends *(*get_emcontrasts*)*, 51
glmmTMB::predict.glmmTMB, *11*, *35*, *44*, *54*

insight::export_table(), *8*
insight::get_data(), *28*
insight::get_datagrid(), *10*, *11*, *19*, *25–29*, *34*, *37*, *39*, *43*, *47*, *49*, *53*, *55*, *72*
insight::get_predicted(), *24*, *26*, *27*, *29*

loess(), *73*

make.names, *3*
marginaleffects::comparisons, *12*, *54*
mice::pool(), *69*, *70*
modelbased-options, 60

parameters::equivalence_test.lm(), *40*
parameters::model_parameters(), *31*
plot(), *25*, *27*

plot.estimate_means
(plot.estimate_predicted), 61
plot.estimate_predicted, 61
plotting examples, *25*
pool_contrasts, 69
pool_predictions, 70
pool_slopes (pool_predictions), 70
print.estimate_contrasts
(display.estimate_contrasts), 6
puppy_love, 71

reference grid, *16*, *38*, *48*
reshape_grouplevel
(estimate_grouplevel), 30
residualize_over_grid, 72
residualize_over_grid(), *64*

see::plot.see_estimate_contrasts(), *9*,
20
smoothing, 73
stigma, 74

tinyplot.estimate_means
(plot.estimate_predicted), 61

visualisation_recipe(), *27*, *29*
visualisation_recipe.estimate_grouplevel
(plot.estimate_predicted), 61
visualisation_recipe.estimate_predicted
(plot.estimate_predicted), 61
visualisation_recipe.estimate_predicted(),
33, *41*
visualisation_recipe.estimate_slopes
(plot.estimate_predicted), 61

zero_crossings, 74