

Package ‘rbmi’

July 22, 2026

Title Reference Based Multiple Imputation

Version 1.6.1

Description Implements standard and reference based multiple imputation methods for continuous longitudinal endpoints (Gower-Page et al. (2022) <[doi:10.21105/joss.04251](https://doi.org/10.21105/joss.04251)>). In particular, this package supports deterministic conditional mean imputation and jackknifing as described in Wolbers et al. (2022) <[doi:10.1002/pst.2234](https://doi.org/10.1002/pst.2234)>, Bayesian multiple imputation as described in Carpenter et al. (2013) <[doi:10.1080/10543406.2013.834911](https://doi.org/10.1080/10543406.2013.834911)>, and bootstrapped maximum likelihood imputation as described in von Hippel and Bartlett (2021) <[doi:10.1214/20-STS793](https://doi.org/10.1214/20-STS793)>.

Encoding UTF-8

LazyData true

URL <https://openpharma.github.io/rbmi/>,
<https://github.com/openpharma/rbmi>

BugReports <https://github.com/openpharma/rbmi/issues>

RoxygenNote 8.0.0

Suggests dplyr, tidyr, nlme, testthat, emmeans, tibble, mvtnorm,
knitr, rmarkdown, bookdown, lubridate, purrr, ggplot2, rstan
(>= 2.26.0), R.rsp, withr, spelling

Config/testthat/edition 3

Imports mrmr, pkgload, Matrix, tools, methods, R6, assertthat, jinjar,
fs, stringr, lifecycle

Depends R (>= 3.4.0)

License Apache License (>= 2)

VignetteBuilder R.rsp

Language en-GB

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Lukas A. Widmer [aut, cre] (ORCID:
<<https://orcid.org/0000-0003-1471-3493>>),
Craig Gower-Page [aut],
Isaac Gravestock [aut] (ORCID: <<https://orcid.org/0000-0003-0283-2065>>),

Alessandro Noci [aut],
 Marcel Wolbers [aut] (ORCID: <<https://orcid.org/0000-0003-4915-9015>>),
 Daniel Sabanes Bove [aut] (ORCID:
 <<https://orcid.org/0000-0002-0176-9239>>),
 F. Hoffmann-La Roche AG [cph, fnd]

Maintainer Lukas A. Widmer <lukas_andreas.widmer@novartis.com>

Repository CRAN

Date/Publication 2026-07-22 21:30:02 UTC

Contents

adjust_trajectories	5
adjust_trajectories_single	6
analyse	7
ancova	11
ancova_single	13
antidepressant_data	15
apply_delta	16
assert_variables_exist	17
as_analysis	17
as_ascii_table	18
as_cropped_char	18
as_dataframe	19
as_draws	19
as_imputation	20
as_indices	21
as_mmrn_df	21
as_mmrn_formula	22
as_model_df	22
as_simple_formula	23
as_stan_array	23
as_strata	24
char2fct	24
check_ESS	25
check_hmc_diagn	25
check_mcmc	26
compute_sigma	27
control	27
convert_to_imputation_list_df	29
delta_template	30
draws	33
d_lagscale	39
eval_mmrn	39
expand	40
extract_covariates	42
extract_data_mnar_as_na	43
extract_draws	43

extract_imputed_df	44
extract_imputed_dfs	45
extract_params	46
fit_mcmc	46
fit_mmrn	47
format_method_descriptions	48
generate_data_single	49
getStrategies	50
get_bootstrap_stack	51
get_conditional_parameters	51
get_delta_template	52
get_draws_mle	52
get_ESS	54
get_ests_bmlmi	54
get_example_data	55
get_jackknife_stack	56
get_mmrn_sample	56
get_pattern_groups	57
get_pattern_groups_unique	58
get_pool_components	58
get_visit_distribution_parameters	59
ife	59
imputation_df	60
imputation_list_df	60
imputation_list_single	61
imputation_single	61
impute	62
impute_data_individual	64
impute_internal	66
impute_outcome	67
invert	67
invert_indexes	68
is_absent	68
is_char_fact	69
is_char_one	69
is_in_rbmi_development	69
is_num_char_fact	70
locf	70
lsmeans	71
ls_design	72
make_rbmi_cluster	73
method	74
parametric_ci	77
par_lapply	78
pool	78
pool_bootstrap_normal	81
pool_bootstrap_percentile	82
prepare_stan_data	83

print.analysis	84
print.draws	84
print.imputation	85
progressLogger	85
pval_percentile	87
QR_decomp	87
random_effects_expr	88
rbmi-settings	88
record	89
recursive_reduce	90
remove_if_all_missing	91
rubin_df	91
rubin_rules	92
sample_ids	93
sample_list	93
sample_mvnorm	94
sample_single	94
scalerConstructor	95
set_simul_pars	97
set_vars	99
simulate_data	100
simulate_dropout	103
simulate_ice	104
simulate_test_data	105
sort_by	106
split_dim	107
split_imputations	108
STAN_BLOCKS	109
strategies	109
string_pad	111
str_contains	111
transpose_imputations	112
transpose_results	112
transpose_samples	113
validate.analysis	114
validate.draws	114
validate.is_mar	115
validate.ivars	115
validate.references	116
validate.sample_list	116
validate.sample_single	117
validate.simul_pars	117
validate.stan_data	118
validate_datalong	118
validate_strategies	119

adjust_trajectories *Adjust trajectories due to the intercurrent event (ICE)*

Description

Adjust trajectories due to the intercurrent event (ICE)

Usage

```
adjust_trajectories(
  distr_pars_group,
  outcome,
  ids,
  ind_ice,
  strategy_fun,
  distr_pars_ref = NULL
)
```

Arguments

distr_pars_group	Named list containing the simulation parameters of the multivariate normal distribution assumed for the given treatment group. It contains the following elements: • mu: Numeric vector indicating the mean outcome trajectory. It should include the outcome at baseline. • sigma Covariance matrix of the outcome trajectory.
outcome	Numeric variable that specifies the longitudinal outcome.
ids	Factor variable that specifies the id of each subject.
ind_ice	A binary variable that takes value 1 if the corresponding outcome is affected by the ICE and 0 otherwise.
strategy_fun	Function implementing trajectories after the intercurrent event (ICE). Must be one of getStrategies() . See getStrategies() for details.
distr_pars_ref	Optional. Named list containing the simulation parameters of the reference arm. It contains the following elements: • mu: Numeric vector indicating the mean outcome trajectory assuming no ICEs. It should include the outcome at baseline. • sigma Covariance matrix of the outcome trajectory assuming no ICEs.

Value

A numeric vector containing the adjusted trajectories.

See Also

[adjust_trajectories_single\(\)](#).

adjust_trajectories_single

Adjust trajectory of a subject's outcome due to the intercurrent event (ICE)

Description

Adjust trajectory of a subject's outcome due to the intercurrent event (ICE)

Usage

```
adjust_trajectories_single(
  distr_pars_group,
  outcome,
  strategy_fun,
  distr_pars_ref = NULL
)
```

Arguments

- | | |
|------------------|--|
| distr_pars_group | Named list containing the simulation parameters of the multivariate normal distribution assumed for the given treatment group. It contains the following elements: <ul style="list-style-type: none"> • mu: Numeric vector indicating the mean outcome trajectory. It should include the outcome at baseline. • sigma Covariance matrix of the outcome trajectory. |
| outcome | Numeric variable that specifies the longitudinal outcome. |
| strategy_fun | Function implementing trajectories after the intercurrent event (ICE). Must be one of getStrategies() . See getStrategies() for details. |
| distr_pars_ref | Optional. Named list containing the simulation parameters of the reference arm. It contains the following elements: <ul style="list-style-type: none"> • mu: Numeric vector indicating the mean outcome trajectory assuming no ICEs. It should include the outcome at baseline. • sigma Covariance matrix of the outcome trajectory assuming no ICEs. |

Details

outcome should be specified such that all-and-only the post-ICE observations (i.e. the observations to be adjusted) are set to NA.

Value

A numeric vector containing the adjusted trajectory for a single subject.

analyse

Analyse Multiple Imputed Datasets

Description

This function takes multiple imputed datasets (as generated by the [impute\(\)](#) function) and runs an analysis function on each of them.

Usage

```
analyse(  
  imputations,  
  fun = ancova,  
  delta = NULL,  
  ...,  
  ncores = 1,  
  .validate = TRUE  
)
```

Arguments

<code>imputations</code>	An imputations object as created by impute() .
<code>fun</code>	An analysis function to be applied to each imputed dataset. See details.
<code>delta</code>	A <code>data.frame</code> containing the delta transformation to be applied to the imputed datasets prior to running <code>fun</code> . See details.
<code>...</code>	Additional arguments passed onto <code>fun</code> .
<code>ncores</code>	The number of parallel processes to use when running this function. Can also be a cluster object created by make_rbmi_cluster() . See the parallelisation section below.
<code>.validate</code>	Should imputations be checked to ensure it conforms to the required format (default = TRUE) ? Can gain a small performance increase if this is set to FALSE when analysing a large number of samples.

Details

This function works by performing the following steps:

1. Extract a dataset from the `imputations` object.
2. Apply any delta adjustments as specified by the `delta` argument.
3. Run the analysis function `fun` on the dataset.
4. Repeat steps 1-3 across all of the datasets inside the `imputations` object.
5. Collect and return all of the analysis results.

The analysis function `fun` must take a `data.frame` as its first argument. All other options to `analyse()` are passed onto `fun` via `...`. `fun` must return a named list with each element itself being a list containing a single numeric element called `est` (or additionally `se` and `df` if you had originally specified `method_bayes()` or `method_approxbayes()`) i.e.:

```
myfun <- function(dat, ...) {
  mod_1 <- lm(data = dat, outcome ~ group)
  mod_2 <- lm(data = dat, outcome ~ group + covar)
  x <- list(
    trt_1 = list(
      est = coef(mod_1)[[group]],
      se = sqrt(vcov(mod_1)[group, group]),
      df = df.residual(mod_1)
    ),
    trt_2 = list(
      est = coef(mod_2)[[group]],
      se = sqrt(vcov(mod_2)[group, group]),
      df = df.residual(mod_2)
    )
  )
  return(x)
}
```

Please note that the `vars$subjid` column (as defined in the original call to `draws()`) will be scrambled in the `data.frames` that are provided to `fun`. This is to say they will not contain the original subject values and as such any hard coding of subject ids is strictly to be avoided.

By default `fun` is the `ancova()` function. Please note that this function requires that a `vars` object, as created by `set_vars()`, is provided via the `vars` argument e.g. `analyse(imputeObj, vars = set_vars(...))`. Please see the documentation for `ancova()` for full details. Please also note that the theoretical justification for the conditional mean imputation method (`method = method_condmean()` in `draws()`) relies on the fact that ANCOVA is a linear transformation of the outcomes. Thus care is required when applying alternative analysis functions in this setting.

The `delta` argument can be used to specify offsets to be applied to the outcome variable in the imputed datasets prior to the analysis. This is typically used for sensitivity or tipping point analyses. The `delta` dataset must contain columns `vars$subjid`, `vars$visit` (as specified in the original call to `draws()`) and `delta`. Essentially this `data.frame` is merged onto the imputed dataset by `vars$subjid` and `vars$visit` and then the outcome variable is modified by:

```
imputed_data[[vars$outcome]] <- imputed_data[[vars$outcome]] + imputed_data[["delta"]]
```

Please note that in order to provide maximum flexibility, the `delta` argument can be used to modify any/all outcome values including those that were not imputed. Care must be taken when defining offsets. It is recommended that you use the helper function `delta_template()` to define the `delta` datasets as this provides utility variables such as `is_missing` which can be used to identify exactly which visits have been imputed.

Value

An analysis object; a list of class "analysis" containing the analysis results (one entry per imputed dataset, each a named list of parameter estimates), the name of the analysis function (fun_name), the applied delta data.frame, the analysis function (fun) and the imputation method. This object is normally passed on to [pool\(\)](#).

Parallelisation

To speed up the evaluation of [analyse\(\)](#) you can use the `ncores` argument to enable parallelisation. Simply providing an integer will get `rbmi` to automatically spawn that many background processes to parallelise across. If you are using a custom analysis function then you need to ensure that any libraries or global objects required by your function are available in the sub-processes. To do this you need to use the [make_rbmi_cluster\(\)](#) function for example:

```
my_custom_fun <- function(...) <some analysis code>
cl <- make_rbmi_cluster(
  4,
  objects = list("my_custom_fun" = my_custom_fun),
  packages = c("dplyr", "nlme")
)
analyse(
  imputations = imputeObj,
  fun = my_custom_fun,
  ncores = cl
)
parallel::stopCluster(cl)
```

Note that there is significant overhead both with setting up the sub-processes and with transferring data back-and-forth between the main process and the sub-processes. As such parallelisation of the [analyse\(\)](#) function tends to only be worth it when you have > 2000 samples generated by [draws\(\)](#). Conversely using parallelisation if your samples are smaller than this may lead to longer run times than just running it sequentially.

It is important to note that the implementation of parallel processing within [analyse\(\)](#) has been optimised around the assumption that the parallel processes will be spawned on the same machine and not a remote cluster. One such optimisation is that the required data is saved to a temporary file on the local disk from which it is then read into each sub-process. This is done to avoid the overhead of transferring the data over the network. Our assumption is that if you are at the stage where you need to be parallelising your analysis over a remote cluster then you would likely be better off parallelising across multiple `rbmi` runs rather than within a single `rbmi` run.

Finally, if you are doing a tipping point analysis you can get a reasonable performance improvement by re-using the cluster between each call to [analyse\(\)](#) e.g.

```
cl <- make_rbmi_cluster(4)
ana_1 <- analyse(
  imputations = imputeObj,
  delta = delta_plan_1,
  ncores = cl
```

```
)  
ana_2 <- analyse(  
  imputations = imputeObj,  
  delta = delta_plan_2,  
  ncores = cl  
)  
ana_3 <- analyse(  
  imputations = imputeObj,  
  delta = delta_plan_3,  
  ncores = cl  
)  
parallel::clusterStop(cl)
```

See Also

[extract_imputed_dfs\(\)](#) for manually extracting imputed datasets.

[delta_template\(\)](#) for creating delta data.frames.

[ancova\(\)](#) for the default analysis function.

Examples

```
## Not run:  
vars <- set_vars(  
  subjid = "subjid",  
  visit = "visit",  
  outcome = "outcome",  
  group = "group",  
  covariates = c("sex", "age", "sex*age")  
)  
  
analyse(  
  imputations = imputeObj,  
  vars = vars  
)  
  
deltadf <- data.frame(  
  subjid = c("Pt1", "Pt1", "Pt2"),  
  visit = c("Visit_1", "Visit_2", "Visit_2"),  
  delta = c( 5, 9, -10)  
)  
  
analyse(  
  imputations = imputeObj,  
  delta = deltdf,  
  vars = vars  
)  
  
## End(Not run)
```

Description

Performs an analysis of covariance between two groups returning the estimated "treatment effect" (i.e. the contrast between the two treatment groups) and the least square means estimates in each group.

Usage

```
ancova(
  data,
  vars,
  visits = NULL,
  weights = c("counterfactual", "equal", "proportional_em", "proportional")
)
```

Arguments

<code>data</code>	A <code>data.frame</code> containing the data to be used in the model.
<code>vars</code>	A <code>vars</code> object as generated by <code>set_vars()</code> . Only the group, visit, outcome and covariates elements are required. See details.
<code>visits</code>	An optional character vector specifying which visits to fit the ANCOVA model at. If <code>NULL</code> , a separate ANCOVA model will be fit to the outcomes for each visit (as determined by <code>unique(data[[vars\$visit]])</code>). See details.
<code>weights</code>	Character, either "counterfactual" (default), "equal", "proportional_em" or "proportional". Specifies the weighting strategy to be used when calculating the lsmeans. See the weighting section for more details.

Details

The function works as follows:

1. Select the first value from `visits`.
2. Subset the data to only the observations that occurred on this visit.
3. Fit a linear model as `vars$outcome ~ vars$group + vars$covariates`.
4. Extract the "treatment effect" & least square means for each treatment group.
5. Repeat points 2-3 for all other values in `visits`.

If no value for `visits` is provided then it will be set to `unique(data[[vars$visit]])`.

In order to meet the formatting standards set by `analyse()` the results will be collapsed into a single list suffixed by the visit name, e.g.:

```
list(
  trt_visit_1 = list(est = ...),
  lsm_ref_visit_1 = list(est = ...),
  lsm_alt_visit_1 = list(est = ...),
  trt_visit_2 = list(est = ...),
  lsm_ref_visit_2 = list(est = ...),
  lsm_alt_visit_2 = list(est = ...),
  ...
)
```

Please note that `ref` refers to the first factor level of `vars$group` which does not necessarily coincide with the control arm. Analogously, `alt` refers to the second factor level of `vars$group`. `trt` refers to the model contrast translating the mean difference between the second level and first level.

If you want to include interaction terms in your model this can be done by providing them to the `covariates` argument of `set_vars()` e.g. `set_vars(covariates = c("sex*age"))`.

Value

A named list with one set of entries per visit. For each visit the list contains the estimated treatment effect (`trt_<visit>`) and the least square means for the reference and alternative groups (`lsm_ref_<visit>` and `lsm_alt_<visit>`). Each of these elements is itself a list holding the estimate (`est`), standard error (`se`) and degrees of freedom (`df`).

Weighting

Counterfactual:

For `weights = "counterfactual"` (the default) the lsmeans are obtained by taking the average of the predicted values for each patient after assigning all patients to each arm in turn. This approach is equivalent to standardization or g-computation. In comparison to `emmeans` this approach is equivalent to:

```
emmeans::emmeans(model, specs = "<treatment>", counterfactual = "<treatment>")
```

Note that to ensure backwards compatibility with previous versions of `rbmi` `weights = "proportional"` is an alias for `weights = "counterfactual"`. To get results consistent with `emmeans`'s `weights = "proportional"` please use `weights = "proportional_em"`.

Equal:

For `weights = "equal"` the lsmeans are obtained by taking the model fitted value of a hypothetical patient whose covariates are defined as follows:

- Continuous covariates are set to $\text{mean}(X)$
- Dummy categorical variables are set to $1/N$ where N is the number of levels
- Continuous * continuous interactions are set to $\text{mean}(X) * \text{mean}(Y)$
- Continuous * categorical interactions are set to $\text{mean}(X) * 1/N$
- Dummy categorical * categorical interactions are set to $1/N * 1/M$

In comparison to `emmeans` this approach is equivalent to:

```
emmeans::emmeans(model, specs = "<treatment>", weights = "equal")
```

Proportional:

For `weights = "proportional_em"` the lsmeans are obtained as per `weights = "equal"` except instead of weighting each observation equally they are weighted by the proportion in which the given combination of categorical values occurred in the data. In comparison to `emmeans` this approach is equivalent to:

```
emmeans::emmeans(model, specs = "<treatment>", weights = "proportional")
```

Note that this is not to be confused with `weights = "proportional"` which is an alias for `weights = "counterfactual"`.

See Also

```
analyse()
stats::lm()
set_vars()
```

Examples

```
# Simulate a small dataset with a single visit, a treatment group and a
# baseline covariate to adjust for.
set.seed(101)
dat <- data.frame(
  visit = factor("visit_1"),
  group = factor(rep(c("Control", "Intervention"), each = 50)),
  basval = rnorm(100)
)
dat$outcome <- 5 + 2 * (dat$group == "Intervention") + dat$basval + rnorm(100)

vars <- set_vars(
  outcome = "outcome",
  group = "group",
  visit = "visit",
  covariates = "basval"
)

# Estimated treatment effect and least square means for the single visit.
# In a full `rbmi` analysis, `ancova()` is passed to `analyse()` rather than
# called directly; see [analyse()].
ancova(dat, vars)
```

Description

Performance analysis of covariance. See [ancova\(\)](#) for full details.

Usage

```
ancova_single(
  data,
  outcome,
  group,
  covariates,
  weights = c("counterfactual", "equal", "proportional_em", "proportional")
)
```

Arguments

<code>data</code>	A <code>data.frame</code> containing the data to be used in the model.
<code>outcome</code>	Character, the name of the outcome variable in data.
<code>group</code>	Character, the name of the group variable in data.
<code>covariates</code>	Character vector containing the name of any additional covariates to be included in the model as well as any interaction terms.
<code>weights</code>	Character, either "counterfactual" (default), "equal", "proportional_em" or "proportional". Specifies the weighting strategy to be used when calculating the lsmeans. See the weighting section for more details.

Details

- `group` must be a factor variable with only 2 levels.
- `outcome` must be a continuous numeric variable.

Weighting**Counterfactual:**

For `weights = "counterfactual"` (the default) the lsmeans are obtained by taking the average of the predicted values for each patient after assigning all patients to each arm in turn. This approach is equivalent to standardization or g-computation. In comparison to `emmeans` this approach is equivalent to:

```
emmeans::emmeans(model, specs = "<treatment>", counterfactual = "<treatment>")
```

Note that to ensure backwards compatibility with previous versions of `rbmi` `weights = "proportional"` is an alias for `weights = "counterfactual"`. To get results consistent with `emmeans`'s `weights = "proportional"` please use `weights = "proportional_em"`.

Equal:

For `weights = "equal"` the lsmeans are obtained by taking the model fitted value of a hypothetical patient whose covariates are defined as follows:

- Continuous covariates are set to $\text{mean}(X)$
- Dummy categorical variables are set to $1/N$ where N is the number of levels
- Continuous * continuous interactions are set to $\text{mean}(X) * \text{mean}(Y)$
- Continuous * categorical interactions are set to $\text{mean}(X) * 1/N$
- Dummy categorical * categorical interactions are set to $1/N * 1/M$

In comparison to emmeans this approach is equivalent to:

```
emmeans::emmeans(model, specs = "<treatment>", weights = "equal")
```

Proportional:

For weights = "proportional_em" the lsmeans are obtained as per weights = "equal" except instead of weighting each observation equally they are weighted by the proportion in which the given combination of categorical values occurred in the data. In comparison to emmeans this approach is equivalent to:

```
emmeans::emmeans(model, specs = "<treatment>", weights = "proportional")
```

Note that this is not to be confused with weights = "proportional" which is an alias for weights = "counterfactual".

See Also

[ancova\(\)](#)

Examples

```
## Not run:
iris2 <- iris[ iris$Species %in% c("versicolor", "virginica"), ]
iris2$Species <- factor(iris2$Species)
ancova_single(iris2, "Sepal.Length", "Species", c("Petal.Length * Petal.Width"))

## End(Not run)
```

antidepressant_data	<i>Antidepressant trial data</i>
---------------------	----------------------------------

Description

A dataset containing data from a publicly available example data set from an antidepressant clinical trial. The dataset is available on the website of the [Drug Information Association Scientific Working Group on Estimands and Missing Data](#). As per that website, the original data are from an antidepressant clinical trial with four treatments; two doses of an experimental medication, a positive control, and placebo and was published in Goldstein et al (2004). To mask the real data, week 8 observations were removed and two arms were created: the original placebo arm and a "drug arm" created by randomly selecting patients from the three non-placebo arms.

Usage

```
antidepressant_data
```

Format

A data.frame with 608 rows and 11 variables:

- PATIENT: patients IDs.
- HAMATOTL: total score Hamilton Anxiety Rating Scale.
- PGIIMP: patient's Global Impression of Improvement Rating Scale.
- RELDAYS: number of days between visit and baseline.
- VISIT: post-baseline visit. Has levels 4,5,6,7.
- THERAPY: the treatment group variable. It is equal to PLACEBO for observations from the placebo arm, or DRUG for observations from the active arm.
- GENDER: patient's gender.
- POOLINV: pooled investigator.
- BASVAL: baseline outcome value.
- HAMD17: Hamilton 17-item rating scale value.
- CHANGE: change from baseline in the Hamilton 17-item rating scale.

Details

The relevant endpoint is the Hamilton 17-item rating scale for depression (HAMD17) for which baseline and weeks 1, 2, 4, and 6 assessments are included. Study drug discontinuation occurred in 24% subjects from the active drug and 26% from placebo. All data after study drug discontinuation are missing and there is a single additional intermittent missing observation.

References

Goldstein, Lu, Detke, Wiltse, Mallinckrodt, Demitrack. Duloxetine in the treatment of depression: a double-blind placebo-controlled comparison with paroxetine. J Clin Psychopharmacol 2004;24: 389-399.

apply_delta	<i>Applies delta adjustment</i>
-------------	---------------------------------

Description

Takes a delta dataset and adjusts the outcome variable by adding the corresponding delta.

Usage

```
apply_delta(data, delta = NULL, group = NULL, outcome = NULL)
```

Arguments

data	data.frame which will have its outcome column adjusted.
delta	data.frame (must contain a column called delta).
group	character vector of variables in both data and delta that will be used to merge the 2 data.frames together by.
outcome	character, name of the outcome variable in data.

 assert_variables_exist

Assert that all variables exist within a dataset

Description

Performs an assertion check to ensure that a vector of variable exists within a data.frame as expected.

Usage

```
assert_variables_exist(data, vars)
```

Arguments

data	a data.frame
vars	a character vector of variable names

 as_analysis

Construct an analysis object

Description

Creates an analysis object ensuring that all components are correctly defined.

Usage

```
as_analysis(results, method, delta = NULL, fun = NULL, fun_name = NULL)
```

Arguments

results	A list of lists contain the analysis results for each imputation See analyse() for details on what this object should look like.
method	The method object as specified in draws() .
delta	The delta dataset used. See analyse() for details on how this should be specified.
fun	The analysis function that was used.
fun_name	The character name of the analysis function (used for printing) purposes.

as_ascii_table	<i>as_ascii_table</i>
----------------	-----------------------

Description

This function takes a data.frame and attempts to convert it into a simple ascii format suitable for printing to the screen. It is assumed all variable values have a as.character() method in order to cast them to character.

Usage

```
as_ascii_table(dat, line_prefix = " ", pcol = NULL)
```

Arguments

dat	Input dataset to convert into a ascii table
line_prefix	Symbols to prefix in front of every line of the table
pcol	name of column to be handled as a p-value. Sets the value to <0.001 if the value is 0 after rounding

as_cropped_char	<i>as_cropped_char</i>
-----------------	------------------------

Description

Makes any character string above x chars Reduce down to a x char string with ...

Usage

```
as_cropped_char(inval, crop_at = 30, ndp = 3)
```

Arguments

inval	a single element value
crop_at	character limit
ndp	Number of decimal places to display

as_dataframe	<i>Convert object to dataframe</i>
--------------	------------------------------------

Description

Convert object to dataframe

Usage

```
as_dataframe(x)
```

Arguments

x	a data.frame like object Utility function to convert a "data.frame-like" object to an actual data.frame to avoid issues with inconsistency on methods (such as <code>[]()</code> and <code>dplyr</code> 's grouped dataframes)
---	---

as_draws	<i>Creates a draws object</i>
----------	-------------------------------

Description

Creates a draws object which is the final output of a call to `draws()`.

Usage

```
as_draws(method, samples, data, formula, n_failures = NULL, fit = NULL)
```

Arguments

method	A method object as generated by either <code>method_bayes()</code> , <code>method_approxbayes()</code> , <code>method_condmean()</code> or <code>method_bmlmi()</code> .
samples	A list of <code>sample_single</code> objects. See <code>sample_single()</code> .
data	R6 longdata object containing all relevant input data information.
formula	Fixed effects formula object used for the model specification.
n_failures	Absolute number of failures of the model fit.
fit	If <code>method_bayes()</code> is chosen, returns the MCMC Stan fit object. Otherwise NULL.

Value

A draws object which is a named list containing the following:

- data: R6 longdata object containing all relevant input data information.
- method: A method object as generated by either `method_bayes()`, `method_approxbayes()` or `method_condmean()`.
- samples: list containing the estimated parameters of interest. Each element of samples is a named list containing the following:
 - ids: vector of characters containing the ids of the subjects included in the original dataset.
 - beta: numeric vector of estimated regression coefficients.
 - sigma: list of estimated covariance matrices (one for each level of vars\$group).
 - theta: numeric vector of transformed covariances.
 - failed: Logical. TRUE if the model fit failed.
 - ids_samp: vector of characters containing the ids of the subjects included in the given sample.
- fit: if `method_bayes()` is chosen, returns the MCMC Stan fit object. Otherwise NULL.
- n_failures: absolute number of failures of the model fit. Relevant only for `method_condmean(type = "bootstrap")`, `method_approxbayes()` and `method_bmlmi()`.
- formula: fixed effects formula object used for the model specification.

as_imputation

Create an imputation object

Description

This function creates the object that is returned from `impute()`. Essentially it is a glorified wrapper around `list()` ensuring that the required elements have been set and that the class is added as expected.

Usage

```
as_imputation(imputations, data, method, references)
```

Arguments

imputations	A list of imputations_list's as created by <code>imputation_df()</code>
data	A longdata object as created by <code>longDataConstructor()</code>
method	A method object as created by <code>method_condmean()</code> , <code>method_bayes()</code> or <code>method_approxbayes()</code>
references	A named vector. Identifies the references to be used when generating the imputed values. Should be of the form <code>c("Group" = "Reference", "Group" = "Reference")</code> .

as_indices	<i>Convert indicator to index</i>
------------	-----------------------------------

Description

Converts a string of 0's and 1's into index positions of the 1's padding the results by 0's so they are all the same length

Usage

```
as_indices(x)
```

Arguments

x a character vector whose values are all either "0" or "1". All elements of the vector must be the same length

Details

i.e.

```
patmap(c("1101", "0001")) -> list(c(1,2,4,999), c(4,999, 999, 999))
```

as_mmrn_df	<i>Creates a "MMRM" ready dataset</i>
------------	---------------------------------------

Description

Converts a design matrix + key variables into a common format In particular this function does the following:

- Renames all covariates as V1, V2, etc to avoid issues of special characters in variable names
- Ensures all key variables are of the right type
- Inserts the outcome, visit and subjid variables into the data.frame naming them as outcome, visit and subjid
- If provided will also insert the group variable into the data.frame named as group

Usage

```
as_mmrn_df(designmat, outcome, visit, subjid, group = NULL)
```

Arguments

designmat	a <code>data.frame</code> or <code>matrix</code> containing the covariates to use in the MMRM model. Dummy variables must already be expanded out, i.e. via <code>stats::model.matrix()</code> . Cannot contain any missing values
outcome	a numeric vector. The outcome value to be regressed on in the MMRM model.
visit	a character / factor vector. Indicates which visit the outcome value occurred on.
subjid	a character / factor vector. The subject identifier used to link separate visits that belong to the same subject.
group	a character / factor vector. Indicates which treatment group the patient belongs to. Will internally be converted to a factor if it is a character vector.

as_mmr_formula	<i>Create MMRM formula</i>
----------------	----------------------------

Description

Derives the MMRM model formula from the structure of `mmrm_df`. returns a formula object of the form:

Usage

```
as_mmr_formula(mmrm_df, cov_struct)
```

Arguments

mmrm_df	an mmrm <code>data.frame</code> as created by <code>as_mmr_df()</code>
cov_struct	Character - The covariance structure to be used, must be one of "us" (default), "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", or "toeph")

Details

```
outcome ~ 0 + V1 + V2 + V4 + ... + us(visit | group / subjid)
```

as_model_df	<i>Expand data.frame into a design matrix</i>
-------------	---

Description

Expands out a `data.frame` using a formula to create a design matrix. Key details are that it will always place the outcome variable into the first column of the return object.

Usage

```
as_model_df(dat, frm)
```

Arguments

dat	a data.frame
frm	a formula

Details

The outcome column may contain NA, but none of the other variables listed in the formula should contain missing values

as_simple_formula	<i>Creates a simple formula object from a string</i>
-------------------	--

Description

Converts a string list of variables into a formula object

Usage

```
as_simple_formula(outcome, covars)
```

Arguments

outcome	character (length 1 vector). Name of the outcome variable
covars	character (vector). Name of covariates

Value

A formula

as_stan_array	<i>As array</i>
---------------	-----------------

Description

Converts a numeric value of length 1 into a 1 dimension array. This is to avoid type errors that are thrown by Stan when length 1 numeric vectors are provided by R for `stan::vector` inputs

Usage

```
as_stan_array(x)
```

Arguments

x	a numeric vector
---	------------------

as_strata	<i>Create vector of strata</i>
-----------	--------------------------------

Description

Collapse multiple categorical variables into distinct unique categories. e.g.

```
as_strata(c(1,1,2,2,2,1), c(5,6,5,5,6,5))
```

would return

```
c(1,2,3,3,4,1)
```

Usage

```
as_strata(...)
```

Arguments

...	numeric/character/factor vectors of the same length
-----	---

Examples

```
## Not run:
as_strata(c(1,1,2,2,2,1), c(5,6,5,5,6,5))

## End(Not run)
```

char2fct	<i>Convert character variables to factor</i>
----------	--

Description

Provided a vector of variable names this function converts any character variables into factors. Has no affect on numeric or existing factor variables

Usage

```
char2fct(data, vars = NULL)
```

Arguments

data	A data.frame
vars	a character vector of variables in data

check_ESS

Diagnostics of the MCMC based on ESS

Description

Check the quality of the MCMC draws from the posterior distribution by checking whether the relative ESS is sufficiently large.

Usage

```
check_ESS(stan_fit, n_draws, threshold_lowESS = 0.4)
```

Arguments

stan_fit	A stanfit object.
n_draws	Number of MCMC draws.
threshold_lowESS	A number in $[0,1]$ indicating the minimum acceptable value of the relative ESS. See details.

Details

check_ESS() works as follows:

1. Extract the ESS from stan_fit for each parameter of the model.
2. Compute the relative ESS (i.e. the ESS divided by the number of draws).
3. Check whether for any of the parameter the ESS is lower than threshold. If for at least one parameter the relative ESS is below the threshold, a warning is thrown.

Value

A warning message in case of detected problems.

check_hmc_diagn

Diagnostics of the MCMC based on HMC-related measures.

Description

Check that:

1. There are no divergent iterations.
2. The Bayesian Fraction of Missing Information (BFMI) is sufficiently low.
3. The number of iterations that saturated the max treedepth is zero.

Please see `rstan::check_hmc_diagnostics()` for details.

Usage

```
check_hmc_diagn(stan_fit)
```

Arguments

`stan_fit` A stanfit object.

Value

A warning message in case of detected problems.

check_mcmc	<i>Diagnostics of the MCMC</i>
------------	--------------------------------

Description

Diagnostics of the MCMC

Usage

```
check_mcmc(stan_fit, n_draws, threshold_lowESS = 0.4)
```

Arguments

`stan_fit` A stanfit object.

`n_draws` Number of MCMC draws.

`threshold_lowESS` A number in $[0,1]$ indicating the minimum acceptable value of the relative ESS. See details.

Details

Performs checks of the quality of the MCMC. See [check_ESS\(\)](#) and [check_hmc_diagn\(\)](#) for details.

Value

A warning message in case of detected problems.

compute_sigma	<i>Compute covariance matrix for some reference-based methods (JR, CIR)</i>
---------------	---

Description

Adapt covariance matrix in reference-based methods. Used for Copy Increments in Reference (CIR) and Jump To Reference (JTR) methods, to adapt the covariance matrix to different pre-deviation and post deviation covariance structures. See Carpenter et al. (2013)

Usage

```
compute_sigma(sigma_group, sigma_ref, index_mar)
```

Arguments

sigma_group	the covariance matrix with dimensions equal to index_mar for the subjects original group
sigma_ref	the covariance matrix with dimensions equal to index_mar for the subjects reference group
index_mar	A logical vector indicating which visits meet the MAR assumption for the subject. I.e. this identifies the observations that after a non-MAR intercurrent event (ICE).

References

Carpenter, James R., James H. Roger, and Michael G. Kenward. "Analysis of longitudinal trials with protocol deviation: a framework for relevant, accessible assumptions, and inference via multiple imputation." *Journal of Biopharmaceutical statistics* 23.6 (2013): 1352-1371.

control	<i>Control the computational details of the imputation methods</i>
---------	--

Description

These functions control lower level computational details of the imputation methods.

Usage

```
control_bayes(
  warmup = 200,
  thin = 50,
  chains = 1,
  init = ifelse(chains > 1, "random", "mrmr"),
  seed = sample.int(.Machine$integer.max, 1),
  ...
)
```

Arguments

warmup	a numeric, the number of warmup iterations for the MCMC sampler.
thin	a numeric, the thinning rate of the MCMC sampler.
chains	a numeric, the number of chains to run in parallel.
init	a character string, the method used to initialise the MCMC sampler, see the details.
seed	a numeric, the seed used to initialise the MCMC sampler.
...	additional arguments to be passed to <code>rstan::sampling()</code> .

Details

Currently only the Bayesian imputation via `method_bayes()` uses a control function:

- The `init` argument can be set to "random" to randomly initialise the sampler with `rstan` default values or to "mmrm" to initialise the sampler with the maximum likelihood estimate values of the MMRM.
- The `seed` argument is used to set the seed for the MCMC sampler. By default, a random seed is generated, such that outside invocation of the `set.seed()` call can effectively set the seed.
- The samples are split across the chains, such that each chain produces `n_samples / chains` (rounded up) samples. The total number of samples that will be returned across all chains is `n_samples` as specified in `method_bayes()`.
- Therefore, the additional parameters passed to `rstan::sampling()` must not contain `n_samples` or `iter`. Instead, the number of samples must only be provided directly via the `n_samples` argument of `method_bayes()`. Similarly, the `refresh` argument is also not allowed here, instead use the `quiet` argument directly in `draws()`.

Value

A named list of control parameters (`warmup`, `thin`, `chains`, `init`, `seed` and any additional arguments passed via ...) that control the MCMC sampler used by `method_bayes()`.

Note

For full reproducibility of the imputation results, it is required to use a `set.seed()` call before defining the control list, and calling the `draws()` function. It is not sufficient to merely set the `seed` argument in the control list.

Examples

```
# Default control settings
control_bayes()

# Customise the warmup, thinning and seed used by the MCMC sampler
control_bayes(warmup = 200, thin = 5, seed = 1821)

# Pass the control settings on to method_bayes()
method_bayes(
```

```

n_samples = 150,
control = control_bayes(warmup = 200, thin = 5, seed = 1821)
)

```

```
convert_to_imputation_list_df
```

Convert list of `imputation_list_single()` objects to an `imputation_list_df()` object (i.e. a list of `imputation_df()` objects)

Description

Convert list of `imputation_list_single()` objects to an `imputation_list_df()` object (i.e. a list of `imputation_df()` objects)

Usage

```
convert_to_imputation_list_df(imputes, sample_ids)
```

Arguments

`imputes` a list of `imputation_list_single()` objects

`sample_ids` A list with 1 element per required `imputation_df`. Each element must contain a vector of "ID"s which correspond to the `imputation_single()` ID's that are required for that dataset. The total number of ID's must be equal to the total number of rows within all of `imputes$imputations`

To accommodate for `method_bmlmi()` the `impute_data_individual()` function returns a list of `imputation_list_single()` objects with 1 object per each subject.

`imputation_list_single()` stores the subjects imputations as a matrix where the columns of the matrix correspond to the D of `method_bmlmi()`. Note that all other methods (i.e. `methods_*`()) are a special case of this with `D = 1`. The number of rows in the matrix varies for each subject and is equal to the number of times the patient was selected for imputation (for non-conditional mean methods this should be 1 per subject per imputed dataset).

This function is best illustrated by an example:

```

imputes = list(
  imputation_list_single(
    id = "Tom",
    imputations = matrix(
      imputation_single_t_1_1, imputation_single_t_1_2,
      imputation_single_t_2_1, imputation_single_t_2_2,
      imputation_single_t_3_1, imputation_single_t_3_2
    )
  ),
)

```

```

      imputation_list_single(
        id = "Tom",
        imputations = matrix(
          imputation_single_h_1_1, imputation_single_h_1_2,
        )
      )
    )
  )
sample_ids <- list(
  c("Tom", "Harry", "Tom"),
  c("Tom")
)

```

Then `convert_to_imputation_df(imputes, sample_ids)` would result in:

```

imputation_list_df(
  imputation_df(
    imputation_single_t_1_1,
    imputation_single_h_1_1,
    imputation_single_t_2_1
  ),
  imputation_df(
    imputation_single_t_1_2,
    imputation_single_h_1_2,
    imputation_single_t_2_2
  ),
  imputation_df(
    imputation_single_t_3_1
  ),
  imputation_df(
    imputation_single_t_3_2
  )
)

```

Note that the different repetitions (i.e. the value set for D) are grouped together sequentially.

delta_template	Create a delta data.frame template
----------------	------------------------------------

Description

Creates a data.frame in the format required by `analyse()` for the use of applying a delta adjustment.

Usage

```
delta_template(imputations, delta = NULL, dlag = NULL, missing_only = TRUE)
```

Arguments

imputations	an imputation object as created by <code>impute()</code> .
delta	NULL or a numeric vector. Determines the baseline amount of delta to be applied to each visit. See details. If a numeric vector it must have the same length as the number of unique visits in the original dataset.
dlag	NULL or a numeric vector. Determines the scaling to be applied to delta based upon which visit the ICE occurred on. See details. If a numeric vector it must have the same length as the number of unique visits in the original dataset.
missing_only	Logical, if TRUE then non-missing post-ICE data will have a delta value of 0 assigned. Note that the calculation (as described in the details section) is performed first and then overwritten with 0's at the end (i.e. the delta values for missing post-ICE visits will stay the same regardless of this option).

Details

To apply a delta adjustment the `analyse()` function expects a delta data.frame with 3 variables: `vars$subjid`, `vars$visit` and `delta` (where `vars` is the object supplied in the original call to `draws()` as created by the `set_vars()` function).

This function will return a data.frame with the aforementioned variables with one row per subject per visit. If the `delta` argument to this function is NULL then the `delta` column in the returned data.frame will be 0 for all observations. If the `delta` argument is not NULL then `delta` will be calculated separately for each subject as the accumulative sum of `delta` multiplied by the scaling coefficient `dlag` based upon how many visits after the subject's intercurrent event (ICE) the visit in question is. This is best illustrated with an example:

Let `delta = c(5,6,7,8)` and `dlag=c(1,2,3,4)` (i.e. assuming there are 4 visits) and lets say that the subject had an ICE on visit 2. The calculation would then be as follows:

```
v1  v2  v3  v4
-----
 5   6   7   8 # delta assigned to each visit
0   1   2   3 # lagged scaling starting from the first visit after the subjects ICE
-----
0   6  14  24 # delta * lagged scaling
-----
0   6  20  44 # accumulative sum of delta to be applied to each visit
```

That is to say the subject would have a delta offset of 0 applied for visit-1, 6 for visit-2, 20 for visit-3 and 44 for visit-4. As a comparison, lets say that the subject instead had their ICE on visit 3, the calculation would then be as follows:

```
v1  v2  v3  v4
-----
 5   6   7   8 # delta assigned to each visit
0   0   1   2 # lagged scaling starting from the first visit after the subjects ICE
-----
0   0   7  16 # delta * lagged scaling
-----
0   0   7  23 # accumulative sum of delta to be applied to each visit
```

In terms of practical usage, let's say that you wanted a delta of 5 to be used for all post ICE visits regardless of their proximity to the ICE visit. This can be achieved by setting $\text{delta} = c(5, 5, 5, 5)$ and $\text{dlag} = c(1, 0, 0, 0)$. For example let's say a subject had their ICE on visit-1, then the calculation would be as follows:

```
v1  v2  v3  v4
-----
5   5   5   5 # delta assigned to each visit
1   0   0   0 # lagged scaling starting from the first visit after the subjects ICE
-----
5   0   0   0 # delta * lagged scaling
-----
5   5   5   5 # accumulative sum of delta to be applied to each visit
```

Another way of using these arguments is to set delta to be the difference in time between visits and dlag to be the amount of delta per unit of time. For example let's say that we have a visit on weeks 1, 5, 6 & 9 and that we want a delta of 3 to be applied for each week after an ICE. This can be achieved by setting $\text{delta} = c(0, 4, 1, 3)$ (the difference in weeks between each visit) and $\text{dlag} = c(3, 3, 3, 3)$. For example let's say we have a subject who had their ICE on week-5 (i.e. visit-2) then the calculation would be:

```
v1  v2  v3  v4
-----
0   4   1   3 # delta assigned to each visit
0   0   3   3 # lagged scaling starting from the first visit after the subjects ICE
-----
0   0   3   9 # delta * lagged scaling
-----
0   0   3  12 # accumulative sum of delta to be applied to each visit
```

i.e. on week-6 (1 week after the ICE) they have a delta of 3 and on week-9 (4 weeks after the ICE) they have a delta of 12.

Please note that this function also returns several utility variables so that the user can create their own custom logic for defining what delta should be set to. These additional variables include:

- `is_mar` - If the observation was missing would it be regarded as MAR? This variable is set to FALSE for observations that occurred after a non-MAR ICE, otherwise it is set to TRUE.
- `is_missing` - Is the outcome variable for this observation missing.
- `is_post_ice` - Does the observation occur after the patient's ICE as defined by the `data_ice` dataset supplied to `draws()`.
- `strategy` - What imputation strategy was assigned to for this subject.

The design and implementation of this function is largely based upon the same functionality as implemented in the so called "five macros" by James Roger. See Roger (2021).

Value

A data.frame with one row per subject per visit containing the variables `vars$subjid`, `vars$visit` and `delta` (the delta offset to apply), together with the utility variables `is_mar`, `is_missing`, `is_post_ice` and `strategy` which can be used to define custom delta logic. If `delta` is `NULL` the delta column is 0 for all observations.

References

Roger, James. Reference-based MI via Multivariate Normal RM (the “five macros” and MIWithD), 2021. URL <https://www.lshtm.ac.uk/research/centres-projects-groups/missing-data#dia-missing-data>.

See Also

`analyse()`

Examples

```
## Not run:
delta_template(imputeObj)
delta_template(imputeObj, delta = c(5,6,7,8), dlag = c(1,2,3,4))

## End(Not run)
```

draws

Fit the base imputation model and get parameter estimates

Description

`draws` fits the base imputation model to the observed outcome data according to the given multiple imputation methodology. According to the user’s method specification, it returns either draws from the posterior distribution of the model parameters as required for Bayesian multiple imputation or frequentist parameter estimates from the original data and bootstrapped or leave-one-out datasets as required for conditional mean imputation. The purpose of the imputation model is to estimate model parameters in the absence of intercurrent events (ICEs) handled using reference-based imputation methods. For this reason, any observed outcome data after ICEs, for which reference-based imputation methods are specified, are removed and considered as missing for the purpose of estimating the imputation model, and for this purpose only. The imputation model is a mixed model for repeated measures (MMRM) that is valid under a missing-at-random (MAR) assumption. It can be fit using maximum likelihood (ML) or restricted ML (REML) estimation, a Bayesian approach, or an approximate Bayesian approach according to the user’s method specification. The ML/REML approaches and the approximate Bayesian approach support several possible covariance structures, while the Bayesian approach based on MCMC sampling supports only an unstructured covariance structure. In any case the covariance matrix can be assumed to be the same or different across each group.

Usage

```
draws(data, data_ice = NULL, vars, method, ncores = 1, quiet = FALSE)

## S3 method for class 'approxbayes'
draws(data, data_ice = NULL, vars, method, ncores = 1, quiet = FALSE)

## S3 method for class 'condmean'
draws(data, data_ice = NULL, vars, method, ncores = 1, quiet = FALSE)

## S3 method for class 'bm1mi'
draws(data, data_ice = NULL, vars, method, ncores = 1, quiet = FALSE)

## S3 method for class 'bayes'
draws(data, data_ice = NULL, vars, method, ncores = 1, quiet = FALSE)
```

Arguments

<code>data</code>	A <code>data.frame</code> containing the data to be used in the model. See details.
<code>data_ice</code>	A <code>data.frame</code> that specifies the information related to the ICEs and the imputation strategies. See details.
<code>vars</code>	A <code>vars</code> object as generated by <code>set_vars()</code> . See details.
<code>method</code>	A method object as generated by either <code>method_bayes()</code> , <code>method_approxbayes()</code> , <code>method_condmean()</code> or <code>method_bm1mi()</code> . It specifies the multiple imputation methodology to be used. See details.
<code>ncores</code>	A single numeric specifying the number of cores to use in creating the draws object. Note that this parameter is ignored for <code>method_bayes()</code> (Default = 1). Can also be a cluster object generated by <code>make_rbmi_cluster()</code>
<code>quiet</code>	Logical, if TRUE will suppress printing of progress information that is printed to the console.

Details

`draws` performs the first step of the multiple imputation (MI) procedure: fitting the base imputation model. The goal is to estimate the parameters of interest needed for the imputation phase (i.e. the regression coefficients and the covariance matrices from a MMRM model).

The function distinguishes between the following methods:

- Bayesian MI based on MCMC sampling: `draws` returns the draws from the posterior distribution of the parameters using a Bayesian approach based on MCMC sampling. This method can be specified by using `method = method_bayes()`.
- Approximate Bayesian MI based on bootstrapping: `draws` returns the draws from the posterior distribution of the parameters using an approximate Bayesian approach, where the sampling from the posterior distribution is simulated by fitting the MMRM model on bootstrap samples of the original dataset. This method can be specified by using `method = method_approxbayes()`.
- Conditional mean imputation with bootstrap re-sampling: `draws` returns the MMRM parameter estimates from the original dataset and from `n_samples` bootstrap samples. This method can be specified by using `method = method_condmean()` with argument `type = "bootstrap"`.

- Conditional mean imputation with jackknife re-sampling: `draws` returns the MMRM parameter estimates from the original dataset and from each leave-one-subject-out sample. This method can be specified by using `method = method_condmean()` with argument `type = "jackknife"`.
- Bootstrapped Maximum Likelihood MI: `draws` returns the MMRM parameter estimates from a given number of bootstrap samples needed to perform random imputations of the bootstrapped samples. This method can be specified by using `method = method_bmlmi()`.

Bayesian MI based on MCMC sampling has been proposed in Carpenter, Roger, and Kenward (2013) who first introduced reference-based imputation methods. Approximate Bayesian MI is discussed in Little and Rubin (2002). Conditional mean imputation methods are discussed in Wolbers et al (2022). Bootstrapped Maximum Likelihood MI is described in Von Hippel & Bartlett (2021).

The argument `data` contains the longitudinal data. It must have at least the following variables:

- `subjid`: a factor vector containing the subject ids.
- `visit`: a factor vector containing the visit the outcome was observed on.
- `group`: a factor vector containing the group that the subject belongs to.
- `outcome`: a numeric vector containing the outcome variable. It might contain missing values. Additional baseline or time-varying covariates must be included in `data`.

`data` must have one row per visit per subject. This means that incomplete outcome data must be set as NA instead of having the related row missing. Missing values in the covariates are not allowed. If `data` is incomplete then the `expand_locf()` helper function can be used to insert any missing rows using Last Observation Carried Forward (LOCF) imputation to impute the covariates values. Note that LOCF is generally not a principled imputation method and should only be used when appropriate for the specific covariate.

Please note that there is no special provisioning for the baseline outcome values. If you do not want baseline observations to be included in the model as part of the response variable then these should be removed in advance from the outcome variable in `data`. At the same time if you want to include the baseline outcome as covariate in the model, then this should be included as a separate column of `data` (as any other covariate).

Character covariates will be explicitly cast to factors. If you use a custom analysis function that requires specific reference levels for the character covariates (for example in the computation of the least square means computation) then you are advised to manually cast your character covariates to factor in advance of running `draws()`.

The argument `data_ice` contains information about the occurrence of ICEs. It is a `data.frame` with 3 columns:

- **Subject ID**: a character vector containing the ids of the subjects that experienced the ICE. This column must be named as specified in `vars$subjid`.
- **Visit**: a character vector containing the first visit after the occurrence of the ICE (i.e. the first visit affected by the ICE). The visits must be equal to one of the levels of `data[[vars$visit]]`. If multiple ICEs happen for the same subject, then only the first non-MAR visit should be used. This column must be named as specified in `vars$visit`.
- **Strategy**: a character vector specifying the imputation strategy to address the ICE for this subject. This column must be named as specified in `vars$strategy`. Possible imputation strategies are:
 - "MAR": Missing At Random.

- "CIR": Copy Increments in Reference.
- "CR": Copy Reference.
- "JR": Jump to Reference.
- "LMCF": Last Mean Carried Forward. For explanations of these imputation strategies, see Carpenter, Roger, and Kenward (2013), Cro et al (2021), and Wolbers et al (2022). Please note that user-defined imputation strategies can also be set.

The `data_ice` argument is necessary at this stage since (as explained in Wolbers et al (2022)), the model is fitted after removing the observations which are incompatible with the imputation model, i.e. any observed data on or after `data_ice[[vars$visit]]` that are addressed with an imputation strategy different from MAR are excluded for the model fit. However such observations will not be discarded from the data in the imputation phase (performed with the function `impute()`). To summarize, **at this stage only pre-ICE data and post-ICE data that is after ICEs for which MAR imputation is specified are used.**

If the `data_ice` argument is omitted, or if a subject doesn't have a record within `data_ice`, then it is assumed that all of the relevant subject's data is pre-ICE and as such all missing visits will be imputed under the MAR assumption and all observed data will be used to fit the base imputation model. Please note that the ICE visit cannot be updated via the `update_strategy` argument in `impute()`; this means that subjects who didn't have a record in `data_ice` will always have their missing data imputed under the MAR assumption even if their strategy is updated.

The `vars` argument is a named list that specifies the names of key variables within `data` and `data_ice`. This list is created by `set_vars()` and contains the following named elements:

- `subjid`: name of the column in `data` and `data_ice` which contains the subject ids variable.
- `visit`: name of the column in `data` and `data_ice` which contains the visit variable.
- `group`: name of the column in `data` which contains the group variable.
- `outcome`: name of the column in `data` which contains the outcome variable.
- `covariates`: vector of characters which contains the covariates to be included in the model (including interactions which are specified as "`covariateName1*covariateName2`"). If no covariates are provided the default model specification of `outcome ~ 1 + visit + group` will be used. Please note that the `group*visit` interaction is **not** included in the model by default.
- `strata`: covariates used as stratification variables in the bootstrap sampling. By default only the `vars$group` is set as stratification variable. Needed only for `method_condmean(type = "bootstrap")` and `method_apprxbayes()`.
- `strategy`: name of the column in `data_ice` which contains the subject-specific imputation strategy.

In our experience, Bayesian MI (`method = method_bayes()`) with a relatively low number of samples (e.g. `n_samples` below 100) frequently triggers STAN warnings about R-hat such as "The largest R-hat is X.XX, indicating chains have not mixed". In many instances, this warning might be spurious, i.e. standard diagnostics analysis of the MCMC samples do not indicate any issues and results look reasonable. Increasing the number of samples to e.g. above 150 usually gets rid of the warning.

Value

A `draws` object which is a named list containing the following:

- `data`: R6 longdata object containing all relevant input data information.
- `method`: A method object as generated by either `method_bayes()`, `method_approxbayes()` or `method_condmean()`.
- `samples`: list containing the estimated parameters of interest. Each element of `samples` is a named list containing the following:
 - `ids`: vector of characters containing the ids of the subjects included in the original dataset.
 - `beta`: numeric vector of estimated regression coefficients.
 - `sigma`: list of estimated covariance matrices (one for each level of `vars$group`).
 - `theta`: numeric vector of transformed covariances.
 - `failed`: Logical. TRUE if the model fit failed.
 - `ids_samp`: vector of characters containing the ids of the subjects included in the given sample.
- `fit`: if `method_bayes()` is chosen, returns the MCMC Stan fit object. Otherwise NULL.
- `n_failures`: absolute number of failures of the model fit. Relevant only for `method_condmean(type = "bootstrap")`, `method_approxbayes()` and `method_bmlmi()`.
- `formula`: fixed effects formula object used for the model specification.

References

James R Carpenter, James H Roger, and Michael G Kenward. Analysis of longitudinal trials with protocol deviation: a framework for relevant, accessible assumptions, and inference via multiple imputation. *Journal of Biopharmaceutical Statistics*, 23(6):1352–1371, 2013.

Suzie Cro, Tim P Morris, Michael G Kenward, and James R Carpenter. Sensitivity analysis for clinical trials with missing continuous outcome data using controlled multiple imputation: a practical guide. *Statistics in Medicine*, 39(21):2815–2842, 2020.

Roderick J. A. Little and Donald B. Rubin. *Statistical Analysis with Missing Data*, Second Edition. John Wiley & Sons, Hoboken, New Jersey, 2002. [Section 10.2.3]

Marcel Wolbers, Alessandro Noci, Paul Delmar, Craig Gower-Page, Sean Yiu, Jonathan W. Bartlett. Standard and reference-based conditional mean imputation. *Pharmaceutical Statistics*. 21(6): 1246-1257, 2022. doi:[10.1002/pst.2234](https://doi.org/10.1002/pst.2234)

Von Hippel, Paul T and Bartlett, Jonathan W. Maximum likelihood multiple imputation: Faster imputations and consistent standard errors without posterior draws. 2021.

See Also

`method_bayes()`, `method_approxbayes()`, `method_condmean()`, `method_bmlmi()` for setting method.

`set_vars()` for setting vars.

`expand_locf()` for expanding data in case of missing rows.

For more details see the quickstart vignette: `vignette("quickstart", package = "rbmi")`.

Examples

```
# Prepare the data: expand to one row per subject and visit
dat <- expand_locf(
  antidepressant_data,
  PATIENT = levels(antidepressant_data$PATIENT),
  VISIT = levels(antidepressant_data$VISIT),
  vars = c("BASVAL", "THERAPY"),
  group = c("PATIENT"),
  order = c("PATIENT", "VISIT")
)

# Derive data_ice: the first visit with a missing outcome for each patient is
# imputed under the Jump To Reference (JR) strategy. Subject 3618 has only an
# intermittent missing value and is removed (imputed under MAR by default).
is_missing <- is.na(dat$CHANGE)
dat_ice <- dat[is_missing, c("PATIENT", "VISIT")]
dat_ice <- dat_ice[!duplicated(dat_ice$PATIENT), ]
dat_ice$strategy <- "JR"
dat_ice <- dat_ice[dat_ice$PATIENT != 3618, ]

vars <- set_vars(
  outcome = "CHANGE",
  visit = "VISIT",
  subjid = "PATIENT",
  group = "THERAPY",
  covariates = c("BASVAL*VISIT", "THERAPY*VISIT")
)

# Conditional mean imputation with jackknife resampling (no rstan required)

drawObj <- draws(
  data = dat,
  data_ice = dat_ice,
  vars = vars,
  method = method_condmean(type = "jackknife"),
  quiet = TRUE
)
drawObj

# Bayesian imputation instead requires the rstan package
if (requireNamespace("rstan", quietly = TRUE)) {
  set.seed(987)
  drawObjBayes <- draws(
    data = dat,
    data_ice = dat_ice,
    vars = vars,
    method = method_bayes(
      n_samples = 150,
      control = control_bayes(warmup = 200, thin = 5, seed = 1821)
    ),
    quiet = TRUE
  )
}
```

```
    drawObjBayes  
  }
```

d_lagscale*Calculate delta from a lagged scale coefficient*

Description

Calculates a delta value based upon a baseline delta value and a post ICE scaling coefficient.

Usage

```
d_lagscale(delta, dlag, is_post_ice)
```

Arguments

delta	a numeric vector. Determines the baseline amount of delta to be applied to each visit.
dlag	a numeric vector. Determines the scaling to be applied to delta based upon with visit the ICE occurred on. Must be the same length as delta.
is_post_ice	logical vector. Indicates whether a visit is "post-ICE" or not.

Details

See [delta_template\(\)](#) for full details on how this calculation is performed.

eval_mmrn*Evaluate a call to mmrm*

Description

This is a utility function that attempts to evaluate a call to mmrm managing any warnings or errors that are thrown.

Usage

```
eval_mmrn(expr)
```

Arguments

expr	An expression to be evaluated. Should be a call to <code>mmrm::mmrm()</code> .
------	--

Details

This function was originally developed for use with `glmmTMB` which needed more hand-holding and dropping of false-positive warnings. It is not as important now but is kept around encase we need to catch false-positive warnings again in the future.

See Also

[record\(\)](#)

Examples

```
## Not run:
eval_mmrn({
  mmrm::mmrm(formula, data)
})

## End(Not run)
```

expand

Expand and fill in missing data.frame rows

Description

These functions are essentially wrappers around `base::expand.grid()` to ensure that missing combinations of data are inserted into a `data.frame` with imputation/fill methods for updating covariate values of newly created rows.

Usage

```
expand(data, ...)
```

```
fill_locf(data, vars, group = NULL, order = NULL)
```

```
expand_locf(data, ..., vars, group, order)
```

Arguments

<code>data</code>	dataset to expand or fill in.
<code>...</code>	variables and the levels that should be expanded out (note that duplicate entries of levels will result in multiple rows for that level).
<code>vars</code>	character vector containing the names of variables that need to be filled in.
<code>group</code>	character vector containing the names of variables to group by when performing LOCF imputation of var.
<code>order</code>	character vector containing the names of additional variables to sort the <code>data.frame</code> by before performing LOCF.

Details

The `draws()` function makes the assumption that all subjects and visits are present in the `data.frame` and that all covariate values are non missing; `expand()`, `fill_locf()` and `expand_locf()` are utility functions to support users in ensuring that their `data.frame`'s conform to these assumptions.

`expand()` takes vectors for expected levels in a `data.frame` and expands out all combinations inserting any missing rows into the `data.frame`. Note that all "expanded" variables are cast as factors.

`fill_locf()` applies LOCF imputation to named covariates to fill in any NAs created by the insertion of new rows by `expand()` (though do note that no distinction is made between existing NAs and newly created NAs). Note that the `data.frame` is sorted by `c(group, order)` before performing the LOCF imputation; the `data.frame` will be returned in the original sort order however.

`expand_locf()` a simple composition function of `fill_locf()` and `expand()` i.e. `fill_locf(expand(...))`.

Missing First Values:

The `fill_locf()` function performs last observation carried forward imputation. A natural consequence of this is that it is unable to impute missing observations if the observation is the first value for a given subject / grouping. These values are deliberately not imputed as doing so risks silent errors in the case of time varying covariates. One solution is to first use `expand_locf()` on just the visit variable and time varying covariates and then merge on the baseline covariates afterwards i.e.

```
library(dplyr)

dat_expanded <- expand(
  data = dat,
  subject = c("pt1", "pt2", "pt3", "pt4"),
  visit = c("vis1", "vis2", "vis3")
)

dat_filled <- dat_expanded %>%
  left_join(baseline_covariates, by = "subject")
```

Value

A `data.frame` (of the same class as the input data). `expand()` returns data with all missing combinations of the expanded variables inserted as new rows, with the expanded variables cast to factors. `fill_locf()` returns data with the named vars imputed by last observation carried forward (in the original row order). `expand_locf()` returns the result of applying `fill_locf()` to the output of `expand()`.

Examples

```
## Not run:
dat_expanded <- expand(
  data = dat,
  subject = c("pt1", "pt2", "pt3", "pt4"),
  visit = c("vis1", "vis2", "vis3")
)
```

```

dat_filled <- fill_locf(
  data = dat_expanded,
  vars = c("Sex", "Age"),
  group = "subject",
  order = "visit"
)

## Or

dat_filled <- expand_locf(
  data = dat,
  subject = c("pt1", "pt2", "pt3", "pt4"),
  visit = c("vis1", "vis2", "vis3"),
  vars = c("Sex", "Age"),
  group = "subject",
  order = "visit"
)

## End(Not run)

```

extract_covariates	<i>Extract Variables from string vector</i>
--------------------	---

Description

Takes a string including potentially model terms like * and : and extracts out the individual variables

Usage

```
extract_covariates(x)
```

Arguments

x string of variable names potentially including interaction terms

Details

i.e. c("v1", "v2", "v2*v3", "v1:v2") becomes c("v1", "v2", "v3")

```
extract_data_mnar_as_na
```

Set to NA outcome values that would be MNAR if they were missing (i.e. which occur after an ICE handled using a reference-based imputation strategy)

Description

Set to NA outcome values that would be MNAR if they were missing (i.e. which occur after an ICE handled using a reference-based imputation strategy)

Usage

```
extract_data_mnar_as_na(longdata)
```

Arguments

longdata R6 longdata object containing all relevant input data information.

Value

A data.frame containing longdata\$get_data(longdata\$ids), but MNAR outcome values are set to NA.

```
extract_draws            Extract draws from a stanfit object
```

Description

Extract draws from a stanfit object and convert them into lists.

The function `rstan::extract()` returns the draws for a given parameter as an array. This function calls `rstan::extract()` to extract the draws from a stanfit object and then convert the arrays into lists.

Usage

```
extract_draws(stan_fit, n_samples)
```

Arguments

stan_fit A stanfit object.
n_samples Number of MCMC draws.

Value

A named list of length 2 containing:

- `beta`: a list of length equal to `n_samples` containing the draws from the posterior distribution of the regression coefficients.
- `sigma`: a list of length equal to `n_samples` containing the draws from the posterior distribution of the covariance matrices. Each element of the list is a list with length equal to 1 if `same_cov = TRUE` or equal to the number of groups if `same_cov = FALSE`.

<code>extract_imputed_df</code>	<i>Extract imputed dataset</i>
---------------------------------	--------------------------------

Description

Takes an imputation object as generated by `imputation_df()` and uses this to extract a completed dataset from a `longdata` object as created by `longDataConstructor()`. Also applies a delta transformation if a `data.frame` is provided to the `delta` argument. See `analyse()` for details on the structure of this `data.frame`.

Subject IDs in the returned `data.frame` are scrambled i.e. are not the original values.

Usage

```
extract_imputed_df(imputation, ld, delta = NULL, idmap = FALSE)
```

Arguments

<code>imputation</code>	An imputation object as generated by <code>imputation_df()</code> .
<code>ld</code>	A <code>longdata</code> object as generated by <code>longDataConstructor()</code> .
<code>delta</code>	Either <code>NULL</code> or a <code>data.frame</code> . Is used to offset outcome values in the imputed dataset.
<code>idmap</code>	Logical. If <code>TRUE</code> an attribute called <code>idmap</code> is attached to the return object which contains a list that maps the old subject ids the new subject ids.

Value

A `data.frame`.

extract_imputed_dfs	<i>Extract imputed datasets</i>
---------------------	---------------------------------

Description

Extracts the imputed datasets contained within an imputations object generated by [impute\(\)](#).

Usage

```
extract_imputed_dfs(  
  imputations,  
  index = seq_along(imputations$imputations),  
  delta = NULL,  
  idmap = FALSE  
)
```

Arguments

imputations	An imputations object as created by impute() .
index	The indexes of the imputed datasets to return. By default, all datasets within the imputations object will be returned.
delta	A <code>data.frame</code> containing the delta transformation to be applied to the imputed dataset. See analyse() for details on the format and specification of this <code>data.frame</code> .
idmap	Logical. The subject IDs in the imputed <code>data.frame</code> 's are replaced with new IDs to ensure they are unique. Setting this argument to <code>TRUE</code> attaches an attribute, called <code>idmap</code> , to the returned <code>data.frame</code> 's that will provide a map from the new subject IDs to the old subject IDs.

Value

A list of `data.frames` equal in length to the `index` argument.

See Also

[delta_template\(\)](#) for creating delta `data.frames`.
[analyse\(\)](#).

Examples

```
## Not run:  
extract_imputed_dfs(imputeObj)  
extract_imputed_dfs(imputeObj, c(1:3))  
  
## End(Not run)
```

extract_params	<i>Extract parameters from a MMRM model</i>
----------------	---

Description

Extracts the beta and sigma coefficients from an MMRM model created by `mmrm::mmrm()`.

Usage

```
extract_params(fit)
```

Arguments

`fit` an object created by `mmrm::mmrm()`

Details

For structured covariance models, additional parameter estimates will be returned, based on the type of the covariance model.

fit_mcmc	<i>Fit the base imputation model using a Bayesian approach</i>
----------	--

Description

`fit_mcmc()` fits the base imputation model using a Bayesian approach. This is done through a MCMC method that is implemented in Stan and is run by using the function `rstan::sampling()`. The function returns the draws from the posterior distribution of the model parameters and the `stanfit` object. Additionally it performs multiple diagnostics checks of the chain and returns warnings in case of any detected issues.

Usage

```
fit_mcmc(designmat, outcome, group, subjid, visit, method, quiet = FALSE)
```

Arguments

<code>designmat</code>	The design matrix of the fixed effects.
<code>outcome</code>	The response variable. Must be numeric.
<code>group</code>	Character vector containing the group variable.
<code>subjid</code>	Character vector containing the subjects IDs.
<code>visit</code>	Character vector containing the visit variable.
<code>method</code>	A method object as generated by <code>method_bayes()</code> .
<code>quiet</code>	Specify whether the Stan sampling log should be printed to the console.

Details

The Bayesian model assumes a multivariate normal likelihood function and weakly-informative priors for the model parameters: in particular, uniform priors are assumed for the regression coefficients and inverse-Wishart priors for the covariance matrices. The chain is initialized using the REML parameter estimates from MMRM as starting values.

The function performs the following steps:

1. Fit MMRM using a REML approach.
2. Prepare the input data for the MCMC fit as described in the `data{}` block of the Stan file. See [prepare_stan_data\(\)](#) for details.
3. Run the MCMC according the input arguments and using as starting values the REML parameter estimates estimated at point 1.
4. Performs diagnostics checks of the MCMC. See [check_mcmc\(\)](#) for details.
5. Extract the draws from the model fit.

The chains perform `method$n_samples` draws by keeping one every `method$burn_between` iterations. Additionally the first `method$burn_in` iterations are discarded. The total number of iterations will then be `method$burn_in + method$burn_between*method$n_samples`. The purpose of `method$burn_in` is to ensure that the samples are drawn from the stationary distribution of the Markov Chain. The `method$burn_between` aims to keep the draws uncorrelated each from other.

Value

A named list composed by the following:

- `samples`: a named list containing the draws for each parameter. It corresponds to the output of [extract_draws\(\)](#).
- `fit`: a `stanfit` object.

fit_mmrn

Fit a MMRM model

Description

Fits a MMRM model allowing for different covariance structures using `mmrm::mmrm()`. Returns a list of key model parameters `beta`, `sigma` and an additional element `failed` indicating whether or not the fit failed to converge. If the fit did fail to converge `beta` and `sigma` will not be present.

Usage

```
fit_mmrn(
  designmat,
  outcome,
  subjid,
  visit,
  group,
```

```

cov_struct = c("us", "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", "toeph"),
REML = TRUE,
same_cov = TRUE
)

```

Arguments

designmat	a data.frame or matrix containing the covariates to use in the MMRM model. Dummy variables must already be expanded out, i.e. via <code>stats::model.matrix()</code> . Cannot contain any missing values
outcome	a numeric vector. The outcome value to be regressed on in the MMRM model.
subjid	a character / factor vector. The subject identifier used to link separate visits that belong to the same subject.
visit	a character / factor vector. Indicates which visit the outcome value occurred on.
group	a character / factor vector. Indicates which treatment group the patient belongs to. Will internally be converted to a factor if it is a character vector.
cov_struct	a character value. Specifies which covariance structure to use. Must be one of "us" (default), "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", or "toeph")
REML	logical. Specifies whether restricted maximum likelihood should be used
same_cov	logical. Used to specify if a shared or individual covariance matrix should be used per group

format_method_descriptions

Format method descriptions

Description

This function formats method descriptions by combining method names and their descriptions.

Usage

```
format_method_descriptions(method)
```

Arguments

method	A named list of methods and their descriptions.
--------	---

Details

If any non-atomic elements are present in the method list, they are converted to a string representation using `dput()`.

Value

A character vector of formatted method descriptions.

generate_data_single *Generate data for a single group*

Description

Generate data for a single group

Usage

```
generate_data_single(pars_group, strategy_fun = NULL, distr_pars_ref = NULL)
```

Arguments

- | | |
|----------------|---|
| pars_group | A simul_pars object as generated by set_simul_pars() . It specifies the simulation parameters of the given group. |
| strategy_fun | Function implementing trajectories after the intercurrent event (ICE). Must be one of getStrategies() . See getStrategies() for details. If NULL then post-ICE outcomes are untouched. |
| distr_pars_ref | Optional. Named list containing the simulation parameters of the reference arm. It contains the following elements: <ul style="list-style-type: none"> • mu: Numeric vector indicating the mean outcome trajectory assuming no ICEs. It should include the outcome at baseline. • sigma Covariance matrix of the outcome trajectory assuming no ICEs. If NULL, then these parameters are inherited from pars_group. |

Value

A data.frame containing the simulated data. It includes the following variables:

- id: Factor variable that specifies the id of each subject.
- visit: Factor variable that specifies the visit of each assessment. Visit 0 denotes the baseline visit.
- group: Factor variable that specifies which treatment group each subject belongs to.
- outcome_bl: Numeric variable that specifies the baseline outcome.
- outcome_noICE: Numeric variable that specifies the longitudinal outcome assuming no ICEs.
- ind_ice1: Binary variable that takes value 1 if the corresponding visit is affected by ICE1 and 0 otherwise.
- dropout_ice1: Binary variable that takes value 1 if the corresponding visit is affected by the drop-out following ICE1 and 0 otherwise.
- ind_ice2: Binary variable that takes value 1 if the corresponding visit is affected by ICE2.
- outcome: Numeric variable that specifies the longitudinal outcome including ICE1, ICE2 and the intermittent missing values.

See Also

[simulate_data\(\)](#).

getStrategies	<i>Get imputation strategies</i>
---------------	----------------------------------

Description

Returns a list defining the imputation strategies to be used to create the multivariate normal distribution parameters by merging those of the source group and reference group per patient.

Usage

```
getStrategies(...)
```

Arguments

... User defined methods to be added to the return list. Input must be a function.

Details

By default Jump to Reference (JR), Copy Reference (CR), Copy Increments in Reference (CIR), Last Mean Carried Forward (LMCF) and Missing at Random (MAR) are defined.

The user can define their own strategy functions (or overwrite the pre-defined ones) by specifying a named input to the function i.e. `NEW = function(...) ...`. Only exception is MAR which cannot be overwritten.

All user defined functions must take 3 inputs: `pars_group`, `pars_ref` and `index_mar`. `pars_group` and `pars_ref` are both lists with elements `mu` and `sigma` representing the multivariate normal distribution parameters for the subject's current group and reference group respectively. `index_mar` will be a logical vector specifying which visits the subject met the MAR assumption at. The function must return a list with elements `mu` and `sigma`. See the implementation of `strategy_JR()` for an example.

Value

A named list of imputation strategy functions. By default this contains the built-in strategies JR, CR, CIR, LMCF and MAR, together with (or overwritten by) any user-defined strategies supplied via ... Each element is a function taking `pars_group`, `pars_ref` and `index_mar`.

Examples

```
## Not run:
getStrategies()
getStrategies(
  NEW = function(pars_group, pars_ref, index_mar) code ,
  JR = function(pars_group, pars_ref, index_mar) more_code
)

## End(Not run)
```

get_bootstrap_stack	<i>Creates a stack object populated with bootstrapped samples</i>
---------------------	---

Description

Function creates a `Stack()` object and populated the stack with bootstrap samples based upon `method$n_samples`

Usage

```
get_bootstrap_stack(longdata, method, stack = Stack$new())
```

Arguments

longdata	A <code>longDataConstructor()</code> object
method	A method object
stack	A <code>Stack()</code> object (this is only exposed for unit testing purposes)

get_conditional_parameters	<i>Derive conditional multivariate normal parameters</i>
----------------------------	--

Description

Takes parameters for a multivariate normal distribution and observed values to calculate the conditional distribution for the unobserved values.

Usage

```
get_conditional_parameters(pars, values)
```

Arguments

pars	a list with elements <code>mu</code> and <code>sigma</code> defining the mean vector and covariance matrix respectively.
values	a vector of observed values to condition on, must be same length as <code>pars\$mu</code> . Missing values must be represented by an <code>NA</code> .

Value

A list with the conditional distribution parameters:

- `mu` - The conditional mean vector.
- `sigma` - The conditional covariance matrix.

get_delta_template	<i>Get delta utility variables</i>
--------------------	------------------------------------

Description

This function creates the default delta template (1 row per subject per visit) and extracts all the utility information that users need to define their own logic for defining delta. See [delta_template\(\)](#) for full details.

Usage

```
get_delta_template(imputations)
```

Arguments

`imputations` an imputations object created by [impute\(\)](#).

get_draws_mle	<i>Fit the base imputation model on bootstrap samples</i>
---------------	---

Description

Fit the base imputation model using a ML/REML approach on a given number of bootstrap samples as specified by `method$n_samples`. Returns the parameter estimates from the model fit.

Usage

```
get_draws_mle(
  longdata,
  method,
  sample_stack,
  n_target_samples,
  first_sample_orig,
  use_samp_ids,
  failure_limit = 0,
  ncores = 1,
  quiet = FALSE
)
```

Arguments

<code>longdata</code>	R6 longdata object containing all relevant input data information.
<code>method</code>	A method object as generated by either method_approxbayes() or method_condmean() with argument <code>type = "bootstrap"</code> .
<code>sample_stack</code>	A stack object containing the subject ids to be used on each mmrm iteration.

n_target_samples	Number of samples needed to be created
first_sample_orig	Logical. If TRUE the function returns method\$n_samples + 1 samples where the first sample contains the parameter estimates from the original dataset and method\$n_samples samples contain the parameter estimates from bootstrap samples. If FALSE the function returns method\$n_samples samples containing the parameter estimates from bootstrap samples.
use_samp_ids	Logical. If TRUE, the sampled subject ids are returned. Otherwise the subject ids from the original dataset are returned. These values are used to tell <code>impute()</code> what subjects should be used to derive the imputed dataset.
failure_limit	Number of failed samples that are allowed before throwing an error
ncores	Number of processes to parallelise the job over
quiet	Logical, If TRUE will suppress printing of progress information that is printed to the console.

Details

This function takes a Stack object which contains multiple lists of patient ids. The function takes this Stack and pulls a set ids and then constructs a dataset just consisting of these patients (i.e. potentially a bootstrap or a jackknife sample).

The function then fits a MMRM model to this dataset to create a sample object. The function repeats this process until n_target_samples have been reached. If more than failure_limit samples fail to converge then the function throws an error.

After reaching the desired number of samples the function generates and returns a draws object.

Value

A draws object which is a named list containing the following:

- data: R6 longdata object containing all relevant input data information.
- method: A method object as generated by either `method_bayes()`, `method_approxbayes()` or `method_condmean()`.
- samples: list containing the estimated parameters of interest. Each element of samples is a named list containing the following:
 - ids: vector of characters containing the ids of the subjects included in the original dataset.
 - beta: numeric vector of estimated regression coefficients.
 - sigma: list of estimated covariance matrices (one for each level of vars\$group).
 - theta: numeric vector of transformed covariances.
 - failed: Logical. TRUE if the model fit failed.
 - ids_samp: vector of characters containing the ids of the subjects included in the given sample.
- fit: if method_bayes() is chosen, returns the MCMC Stan fit object. Otherwise NULL.
- n_failures: absolute number of failures of the model fit. Relevant only for method_condmean(type = "bootstrap"), method_approxbayes() and method_bmlmi().
- formula: fixed effects formula object used for the model specification.

get_ESS	<i>Extract the Effective Sample Size (ESS) from a stanfit object</i>
---------	--

Description

Extract the Effective Sample Size (ESS) from a stanfit object

Usage

```
get_ESS(stan_fit)
```

Arguments

stan_fit A stanfit object.

Value

A named vector containing the ESS for each parameter of the model.

get_est_bmlmi	<i>Von Hippel and Bartlett pooling of BMLMI method</i>
---------------	--

Description

Compute pooled point estimates, standard error and degrees of freedom according to the Von Hippel and Bartlett formula for Bootstrapped Maximum Likelihood Multiple Imputation (BMLMI).

Usage

```
get_est_bmlmi(ests, D)
```

Arguments

ests numeric vector containing estimates from the analysis of the imputed datasets.
 D numeric representing the number of imputations between each bootstrap sample in the BMLMI method.

Details

ests must be provided in the following order: the firsts D elements are related to analyses from random imputation of one bootstrap sample. The second set of D elements (i.e. from D+1 to 2*D) are related to the second bootstrap sample and so on.

Value

a list containing point estimate, standard error and degrees of freedom.

References

Von Hippel, Paul T and Bartlett, Jonathan W8. Maximum likelihood multiple imputation: Faster imputations and consistent standard errors without posterior draws. 2021

`get_example_data`*Simulate a realistic example dataset*

Description

Simulate a realistic example dataset using `simulate_data()` with hard-coded values of all the input arguments.

Usage

```
get_example_data()
```

Details

`get_example_data()` simulates a 1:1 randomized trial of an active drug (intervention) versus placebo (control) with 100 subjects per group and 6 post-baseline assessments (bi-monthly visits until 12 months). One intercurrent event corresponding to treatment discontinuation is also simulated. Specifically, data are simulated under the following assumptions:

- The mean outcome trajectory in the placebo group increases linearly from 50 at baseline (visit 0) to 60 at visit 6, i.e. the slope is 10 points/year.
- The mean outcome trajectory in the intervention group is identical to the placebo group up to visit 2. From visit 2 onward, the slope decreases by 50% to 5 points/year.
- The covariance structure of the baseline and follow-up values in both groups is implied by a random intercept and slope model with a standard deviation of 5 for both the intercept and the slope, and a correlation of 0.25. In addition, an independent residual error with standard deviation 2.5 is added to each assessment.
- The probability of study drug discontinuation after each visit is calculated according to a logistic model which depends on the observed outcome at that visit. Specifically, a visit-wise discontinuation probability of 2% and 3% in the control and intervention group, respectively, is specified in case the observed outcome is equal to 50 (the mean value at baseline). The odds of a discontinuation is simulated to increase by +10% for each +1 point increase of the observed outcome.
- Study drug discontinuation is simulated to have no effect on the mean trajectory in the placebo group. In the intervention group, subjects who discontinue follow the slope of the mean trajectory from the placebo group from that time point onward. This is compatible with a copy increments in reference (CIR) assumption.
- Study drop-out at the study drug discontinuation visit occurs with a probability of 50% leading to missing outcome data from that time point onward.

Value

A `data.frame` of simulated longitudinal trial data (as produced by `simulate_data()`) with 100 subjects per group across 7 visits, suitable for demonstrating the `rbmi` workflow. It contains the subject id, group, visit, outcome and covariate columns together with the intercurrent-event and missingness indicators generated by the simulation.

See Also

`simulate_data()`, `set_simul_pars()`

Examples

```
set.seed(1974)
dat <- get_example_data()
head(dat)
```

<code>get_jackknife_stack</code>	<i>Creates a stack object populated with jackknife samples</i>
----------------------------------	--

Description

Function creates a `Stack()` object and populated the stack with jackknife samples based upon

Usage

```
get_jackknife_stack(longdata, method, stack = Stack$new())
```

Arguments

<code>longdata</code>	A <code>longDataConstructor()</code> object
<code>method</code>	A method object
<code>stack</code>	A <code>Stack()</code> object (this is only exposed for unit testing purposes)

<code>get_mmrn_sample</code>	<i>Fit MMRM and returns parameter estimates</i>
------------------------------	---

Description

`get_mmrn_sample` fits the base imputation model using a ML/REML approach. Returns the parameter estimates from the fit.

Usage

```
get_mmrn_sample(ids, longdata, method)
```

Arguments

ids	vector of characters containing the ids of the subjects.
longdata	R6 longdata object containing all relevant input data information.
method	A method object as generated by either <code>method_approxbayes()</code> or <code>method_condmean()</code> .

Value

A named list of class `sample_single`. It contains the following:

- ids vector of characters containing the ids of the subjects included in the original dataset.
- beta numeric vector of estimated regression coefficients.
- sigma list of estimated covariance matrices (one for each level of `vars$group`).
- theta numeric vector of transformed covariances.
- failed logical. TRUE if the model fit failed.
- ids_samp vector of characters containing the ids of the subjects included in the given sample.

get_pattern_groups	<i>Determine patients missingness group</i>
--------------------	---

Description

Takes a design matrix with multiple rows per subject and returns a dataset with 1 row per subject with a new column `pgroup` indicating which group the patient belongs to (based upon their missingness pattern and treatment group)

Usage

```
get_pattern_groups(ddat)
```

Arguments

ddat	a <code>data.frame</code> with columns <code>subjid</code> , <code>visit</code> , <code>group</code> , <code>is_avail</code>
------	--

Details

- The column `is_avail` must be a character or numeric 0 or 1

get_pattern_groups_unique

Get Pattern Summary

Description

Takes a dataset of pattern information and creates a summary dataset of it with just 1 row per pattern

Usage

```
get_pattern_groups_unique(patterns)
```

Arguments

patterns A data.frame with the columns pgroup, pattern and group

Details

- The column pgroup must be a numeric vector indicating which pattern group the patient belongs to
- The column pattern must be a character string of 0's or 1's. It must be identical for all rows within the same pgroup
- The column group must be a character / numeric vector indicating which covariance group the observation belongs to. It must be identical within the same pgroup

get_pool_components *Expected Pool Components*

Description

Returns the elements expected to be contained in the analyse object depending on what analysis method was specified.

Usage

```
get_pool_components(x)
```

Arguments

x Character name of the analysis method, must one of either "rubin", "jackknife", "bootstrap" or "bmlmi".

get_visit_distribution_parameters

Derive visit distribution parameters

Description

Takes patient level data and beta coefficients and expands them to get a patient specific estimate for the visit distribution parameters mu and sigma. Returns the values in a specific format which is expected by downstream functions in the imputation process (namely `list(list(mu = ..., sigma = ...), list(mu = ..., sigma = ...))`).

Usage

```
get_visit_distribution_parameters(dat, beta, sigma)
```

Arguments

dat	Patient level dataset, must be 1 row per visit. Column order must be in the same order as beta. The number of columns must match the length of beta
beta	List of model beta coefficients. There should be 1 element for each sample e.g. if there were 3 samples and the models each had 4 beta coefficients then this argument should be of the form <code>list(c(1,2,3,4) , c(5,6,7,8) , c(9,10,11,12))</code> . All elements of beta must be the same length and must be the same length and order as dat.
sigma	List of sigma. Must have the same number of entries as beta.

ife	<i>if else</i>
-----	----------------

Description

A wrapper around `if() else()` to prevent unexpected interactions between `ifelse()` and factor variables

Usage

```
ife(x, a, b)
```

Arguments

x	True / False
a	value to return if True
b	value to return if False

Details

By default `ifelse()` will convert factor variables to their numeric values which is often undesirable. This convenience function avoids that problem

<code>imputation_df</code>	<i>Create a valid imputation_df object</i>
----------------------------	--

Description

Create a valid `imputation_df` object

Usage

```
imputation_df(...)
```

Arguments

... a list of `imputation_single`.

<code>imputation_list_df</code>	<i>List of imputation_dfs</i>
---------------------------------	-------------------------------

Description

A container for multiple [imputation_df](#)'s

Usage

```
imputation_list_df(...)
```

Arguments

... objects of class `imputation_df`

imputation_list_single

A collection of imputation_singles() grouped by a single subjid ID

Description

A collection of imputation_singles() grouped by a single subjid ID

Usage

```
imputation_list_single(imputations, D = 1)
```

Arguments

- | | |
|-------------|--|
| imputations | a list of <code>imputation_single()</code> objects ordered so that repetitions are grouped sequentially |
| D | <p>the number of repetitions that were performed which determines how many columns the imputation matrix should have</p> <p>This is a constructor function to create a <code>imputation_list_single</code> object which contains a matrix of <code>imputation_single()</code> objects grouped by a single id. The matrix is split so that it has D columns (i.e. for non-BMLMI methods this will always be 1)</p> <p>The id attribute is determined by extracting the id attribute from the contributing <code>imputation_single()</code> objects. An error is throw if multiple id are detected</p> |

imputation_single *Create a valid imputation_single object*

Description

Create a valid imputation_single object

Usage

```
imputation_single(id, values)
```

Arguments

- | | |
|--------|---|
| id | a character string specifying the subject id. |
| values | a numeric vector indicating the imputed values. |

impute

*Create imputed datasets***Description**

`impute()` creates imputed datasets based upon the data and options specified in the call to `draws()`. One imputed dataset is created per each "sample" created by `draws()`.

Usage

```
impute(
  draws,
  references = NULL,
  update_strategy = NULL,
  strategies = getStrategies()
)

## S3 method for class 'random'
impute(
  draws,
  references = NULL,
  update_strategy = NULL,
  strategies = getStrategies()
)

## S3 method for class 'condmean'
impute(
  draws,
  references = NULL,
  update_strategy = NULL,
  strategies = getStrategies()
)
```

Arguments

<code>draws</code>	A draws object created by <code>draws()</code> .
<code>references</code>	A named vector. Identifies the references to be used for reference-based imputation methods. Should be of the form <code>c("Group1" = "Reference1", "Group2" = "Reference2")</code> . If <code>NULL</code> (default), the references are assumed to be of the form <code>c("Group1" = "Group1", "Group2" = "Group2")</code> . This argument cannot be <code>NULL</code> if an imputation strategy (as defined by <code>data_ice[[vars\$strategy]]</code> in the call to <code>draws()</code>) other than MAR is set.
<code>update_strategy</code>	An optional <code>data.frame</code> . Updates the imputation method that was originally set via the <code>data_ice</code> option in <code>draws()</code> . See the details section for more information.

strategies A named list of functions. Defines the imputation functions to be used. The names of the list should mirror the values specified in `strategy` column of `data_ice`. Default = `getStrategies()`. See `getStrategies()` for more details.

Details

`impute()` uses the imputation model parameter estimates, as generated by `draws()`, to first calculate the marginal (multivariate normal) distribution of a subject's longitudinal outcome variable depending on their covariate values. For subjects with intercurrent events (ICEs) handled using non-MAR methods, this marginal distribution is then updated depending on the time of the first visit affected by the ICE, the chosen imputation strategy and the chosen reference group as described in Carpenter, Roger, and Kenward (2013). The subject's imputation distribution used for imputing missing values is then defined as their marginal distribution conditional on their observed outcome values. One dataset is being generated per set of parameter estimates provided by `draws()`.

The exact manner in how missing values are imputed from this conditional imputation distribution depends on the method object that was provided to `draws()`, in particular:

- Bayes & Approximate Bayes: each imputed dataset contains 1 row per subject & visit from the original dataset with missing values imputed by taking a single random sample from the conditional imputation distribution.
- Conditional Mean: each imputed dataset contains 1 row per subject & visit from the bootstrapped or jackknife dataset that was used to generate the corresponding parameter estimates in `draws()`. Missing values are imputed by using the mean of the conditional imputation distribution. Please note that the first imputed dataset refers to the conditional mean imputation on the original dataset whereas all subsequent imputed datasets refer to conditional mean imputations for bootstrap or jackknife samples, respectively, of the original data.
- Bootstrapped Maximum Likelihood MI (BMLMI): it performs D random imputations of each bootstrapped dataset that was used to generate the corresponding parameter estimates in `draws()`. A total number of B*D imputed datasets is provided, where B is the number of bootstrapped datasets. Missing values are imputed by taking a random sample from the conditional imputation distribution.

The `update_strategy` argument can be used to update the imputation strategy that was originally set via the `data_ice` option in `draws()`. This avoids having to re-run the `draws()` function when changing the imputation strategy in certain circumstances (as detailed below). The `data.frame` provided to `update_strategy` argument must contain two columns, one for the subject ID and another for the imputation strategy, whose names are the same as those defined in the `vars` argument as specified in the call to `draws()`. Please note that this argument only allows you to update the imputation strategy and not other arguments such as the time of the first visit affected by the ICE. A key limitation of this functionality is that one can only switch between a MAR and a non-MAR strategy (or vice versa) for subjects without observed post-ICE data. The reason for this is that such a change would affect whether the post-ICE data is included in the base imputation model or not (as explained in the help to `draws()`). As an example, if a subject had their ICE on "Visit 2" but had observed/known values for "Visit 3" then the function will throw an error if one tries to switch the strategy from MAR to a non-MAR strategy. In contrast, switching from a non-MAR to a MAR strategy, whilst valid, will raise a warning as not all usable data will have been utilised in the imputation model.

Value

An imputation object; a list of class "imputation" containing the original longitudinal data, a list of imputations (one entry per imputed dataset, each recording the imputed values needed to reconstruct a completed dataset) and the method used. This object is typically passed on to [analyse\(\)](#); the completed datasets can be extracted with [extract_imputed_dfs\(\)](#).

References

James R Carpenter, James H Roger, and Michael G Kenward. Analysis of longitudinal trials with protocol deviation: a framework for relevant, accessible assumptions, and inference via multiple imputation. *Journal of Biopharmaceutical Statistics*, 23(6):1352–1371, 2013. [Section 4.2 and 4.3]

Examples

```
## Not run:

impute(
  draws = drawobj,
  references = c("Trt" = "Placebo", "Placebo" = "Placebo")
)

new_strategy <- data.frame(
  subjid = c("Pt1", "Pt2"),
  strategy = c("MAR", "JR")
)

impute(
  draws = drawobj,
  references = c("Trt" = "Placebo", "Placebo" = "Placebo"),
  update_strategy = new_strategy
)

## End(Not run)
```

impute_data_individual

Impute data for a single subject

Description

This function performs the imputation for a single subject at a time implementing the process as detailed in [impute\(\)](#).

Usage

```
impute_data_individual(
  id,
```

```

    index,
    beta,
    sigma,
    data,
    references,
    strategies,
    condmean,
    n_imputations = 1
  )

```

Arguments

id	Character string identifying the subject.
index	The sample indexes which the subject belongs to e.g c(1,1,1,2,2,4).
beta	A list of beta coefficients for each sample, i.e. beta[[1]] is the set of beta coefficients for the first sample.
sigma	A list of the sigma coefficients for each sample split by group i.e. sigma[[1]][["A"]] would give the sigma coefficients for group A for the first sample.
data	A longdata object created by longDataConstructor()
references	A named vector. Identifies the references to be used when generating the imputed values. Should be of the form c("Group" = "Reference", "Group" = "Reference").
strategies	A named list of functions. Defines the imputation functions to be used. The names of the list should mirror the values specified in method column of data_ice. Default = getStrategies(). See getStrategies() for more details.
condmean	Logical. If TRUE will impute using the conditional mean values, if FALSE will impute by taking a random draw from the multivariate normal distribution.
n_imputations	When condmean = FALSE numeric representing the number of random imputations to be performed for each sample. Default is 1 (one random imputation per sample).

Details

Note that this function performs all of the required imputations for a subject at the same time. I.e. if a subject is included in samples 1,3,5,9 then all imputations (using sample-dependent imputation model parameters) are performed in one step in order to avoid having to look up a subjects' covariates and expanding them to a design matrix multiple times (which would be more computationally expensive). The function also supports subject belonging to the same sample multiple times, i.e. 1,1,2,3,5,5, as will typically occur for bootstrapped datasets.

impute_internal	Create imputed datasets
-----------------	-------------------------

Description

This is the work horse function that implements most of the functionality of impute. See the user level function [impute\(\)](#) for further details.

Usage

```
impute_internal(
  draws,
  references = NULL,
  update_strategy,
  strategies,
  condmean
)
```

Arguments

draws	A draws object created by draws() .
references	A named vector. Identifies the references to be used for reference-based imputation methods. Should be of the form <code>c("Group1" = "Reference1", "Group2" = "Reference2")</code> . If NULL (default), the references are assumed to be of the form <code>c("Group1" = "Group1", "Group2" = "Group2")</code> . This argument cannot be NULL if an imputation strategy (as defined by <code>data_ice[[vars\$strategy]]</code> in the call to draws()) other than MAR is set.
update_strategy	An optional data.frame. Updates the imputation method that was originally set via the <code>data_ice</code> option in draws() . See the details section for more information.
strategies	A named list of functions. Defines the imputation functions to be used. The names of the list should mirror the values specified in <code>strategy</code> column of <code>data_ice</code> . Default = getStrategies() . See getStrategies() for more details.
condmean	logical. If TRUE will impute using the conditional mean values, if values will impute by taking a random draw from the multivariate normal distribution.

impute_outcome	<i>Sample outcome value</i>
----------------	-----------------------------

Description

Draws a random sample from a multivariate normal distribution.

Usage

```
impute_outcome(conditional_parameters, n_imputations = 1, condmean = FALSE)
```

Arguments

conditional_parameters	a list with elements mu and sigma which contain the mean vector and covariance matrix to sample from.
n_imputations	numeric representing the number of random samples from the multivariate normal distribution to be performed. Default is 1.
condmean	should conditional mean imputation be performed (as opposed to random sampling)

invert	<i>invert</i>
--------	---------------

Description

Utility function used to replicate `purrr::transpose`. Turns a list inside out.

Usage

```
invert(x)
```

Arguments

x	list
---	------

invert_indexes	<i>Invert and derive indexes</i>
----------------	----------------------------------

Description

Takes a list of elements and creates a new list containing 1 entry per unique element value containing the indexes of which original elements it occurred in.

Usage

```
invert_indexes(x)
```

Arguments

x list of elements to invert and calculate index from (see details).

Details

This functions purpose is best illustrated by an example:

input:

```
list( c("A", "B", "C"), c("A", "A", "B")) }
```

becomes:

```
list( "A" = c(1,2,2), "B" = c(1,2), "C" = 1 )
```

is_absent	<i>Is value absent</i>
-----------	------------------------

Description

Returns true if a value is either NULL, NA or "". In the case of a vector all values must be NULL/NA/"" for x to be regarded as absent.

Usage

```
is_absent(x, na = TRUE, blank = TRUE)
```

Arguments

x	a value to check if it is absent or not
na	do NAs count as absent
blank	do blanks i.e. "" count as absent

is_char_fact	<i>Is character or factor</i>
--------------	-------------------------------

Description

returns true if x is character or factor vector

Usage

```
is_char_fact(x)
```

Arguments

x	a character or factor vector
---	------------------------------

is_char_one	<i>Is single character</i>
-------------	----------------------------

Description

returns true if x is a length 1 character vector

Usage

```
is_char_one(x)
```

Arguments

x	a character vector
---	--------------------

is_in_rbmi_development	<i>Is package in development mode?</i>
------------------------	--

Description

Returns TRUE if the package is being developed on i.e. you have a local copy of the source code which you are actively editing Returns FALSE otherwise

Usage

```
is_in_rbmi_development()
```

Details

Main use of this function is in parallel processing to indicate whether the sub-processes need to load the current development version of the code or whether they should load the main installed package on the system

is_num_char_fact	<i>Is character, factor or numeric</i>
------------------	--

Description

returns true if x is a character, numeric or factor vector

Usage

```
is_num_char_fact(x)
```

Arguments

x	a character, numeric or factor vector
---	---------------------------------------

locf	<i>Last Observation Carried Forward</i>
------	---

Description

Returns a vector after applied last observation carried forward imputation.

Usage

```
locf(x)
```

Arguments

x	a vector.
---	-----------

Value

A vector of the same length as x in which each NA has been replaced by the most recent preceding non-NA value. Leading NAs (those with no earlier value to carry forward) are left unchanged.

Examples

```
## Not run:
locf(c(NA, 1, 2, 3, NA, 4)) # Returns c(NA, 1, 2, 3, 3, 4)

## End(Not run)
```

lsmeans

*Least Square Means***Description**

Estimates the least square means from a linear model. The exact implementation / interpretation depends on the weighting scheme; see the weighting section for more information.

Usage

```
lsmeans(
  model,
  ...,
  .weights = c("counterfactual", "equal", "proportional_em", "proportional")
)
```

Arguments

<code>model</code>	A model created by <code>lm</code> .
<code>...</code>	Fixes specific variables to specific values i.e. <code>trt = 1</code> or <code>age = 50</code> . The name of the argument must be the name of the variable within the dataset.
<code>.weights</code>	Character, either "counterfactual" (default), "equal", "proportional_em" or "proportional". Specifies the weighting strategy to be used when calculating the lsmeans. See the weighting section for more details.

Weighting**Counterfactual:**

For `weights = "counterfactual"` (the default) the lsmeans are obtained by taking the average of the predicted values for each patient after assigning all patients to each arm in turn. This approach is equivalent to standardization or g-computation. In comparison to `emmeans` this approach is equivalent to:

```
emmeans::emmeans(model, specs = "<treatment>", counterfactual = "<treatment>")
```

Note that to ensure backwards compatibility with previous versions of `rbmi` `weights = "proportional"` is an alias for `weights = "counterfactual"`. To get results consistent with `emmeans`'s `weights = "proportional"` please use `weights = "proportional_em"`.

Equal:

For `weights = "equal"` the lsmeans are obtained by taking the model fitted value of a hypothetical patient whose covariates are defined as follows:

- Continuous covariates are set to $\text{mean}(X)$
- Dummy categorical variables are set to $1/N$ where N is the number of levels
- Continuous * continuous interactions are set to $\text{mean}(X) * \text{mean}(Y)$
- Continuous * categorical interactions are set to $\text{mean}(X) * 1/N$

- Dummy categorical * categorical interactions are set to $1/N * 1/M$

In comparison to emmeans this approach is equivalent to:

```
emmeans::emmeans(model, specs = "<treatment>", weights = "equal")
```

Proportional:

For weights = "proportional_em" the lsmeans are obtained as per weights = "equal" except instead of weighting each observation equally they are weighted by the proportion in which the given combination of categorical values occurred in the data. In comparison to emmeans this approach is equivalent to:

```
emmeans::emmeans(model, specs = "<treatment>", weights = "proportional")
```

Note that this is not to be confused with weights = "proportional" which is an alias for weights = "counterfactual".

Fixing

Regardless of the weighting scheme any named arguments passed via ... will fix the value of the covariate to the specified value. For example, lsmeans(model, trt = "A") will fix the dummy variable trtA to 1 for all patients (real or hypothetical) when calculating the lsmeans.

See the references for similar implementations as done in SAS and in R via the emmeans package.

References

<https://CRAN.R-project.org/package=emmeans>

https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.3/statug/statug_glm_details41.htm

Examples

```
## Not run:
mod <- lm(Sepal.Length ~ Species + Petal.Length, data = iris)
lsmeans(mod)
lsmeans(mod, Species = "virginica")
lsmeans(mod, Species = "versicolor")
lsmeans(mod, Species = "versicolor", Petal.Length = 1)

## End(Not run)
```

ls_design

Calculate design vector for the lsmeans

Description

Calculates the design vector as required to generate the lsmean and standard error. ls_design_equal calculates it by applying an equal weight per covariate combination whilst ls_design_proportional applies weighting proportional to the frequency in which the covariate combination occurred in the actual dataset.

Usage

```
ls_design_equal(data, frm, fix)

ls_design_counterfactual(data, frm, fix)

ls_design_proportional(data, frm, fix)
```

Arguments

data	A data.frame
frm	Formula used to fit the original model
fix	A named list of variables with fixed values

make_rbmi_cluster	<i>Create a rbmi ready cluster</i>
-------------------	------------------------------------

Description

Create a rbmi ready cluster

Usage

```
make_rbmi_cluster(ncores = 1, objects = NULL, packages = NULL)
```

Arguments

ncores	Number of parallel processes to use or an existing cluster to make use of
objects	a named list of objects to export into the sub-processes
packages	a character vector of libraries to load in the sub-processes

This function is a wrapper around `parallel::makePSOCKcluster()` but takes care of configuring rbmi to be used in the sub-processes as well as loading user defined objects and libraries and setting the seed for reproducibility.

If ncores is 1 this function will return NULL.

If ncores is a cluster created via `parallel::makeCluster()` then this function just takes care of inserting the relevant rbmi objects into the existing cluster.

Value

A parallel socket cluster object (as created by `parallel::makePSOCKcluster()`) configured for use with rbmi, or NULL if ncores = 1. When ncores is an existing cluster it is returned with the required rbmi objects and libraries loaded into it.

Examples

```
## Not run:
# Basic usage
make_rbmi_cluster(5)

# User objects + libraries
VALUE <- 5
myfun <- function(x) {
  x + day(VALUE) # From lubridate::day()
}
make_rbmi_cluster(5, list(VALUE = VALUE, myfun = myfun), c("lubridate"))

# Using a already created cluster
cl <- parallel::makeCluster(5)
make_rbmi_cluster(cl)

## End(Not run)
```

method	<i>Set the multiple imputation methodology</i>
--------	--

Description

These functions determine what methods rbmi should use when creating the imputation models, generating imputed values and pooling the results.

Usage

```
method_bayes(
  covariance = c("us", "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", "toeph"),
  same_cov = TRUE,
  n_samples = 20,
  prior_cov = c("default", "lkj"),
  control = control_bayes(),
  burn_in = NULL,
  burn_between = NULL
)

method_apprxbayes(
  covariance = c("us", "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", "toeph"),
  threshold = 0.01,
  same_cov = TRUE,
  REML = TRUE,
  n_samples = 20
)

method_condmean(
  covariance = c("us", "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", "toeph"),
```

```

    threshold = 0.01,
    same_cov = TRUE,
    REML = TRUE,
    n_samples = NULL,
    type = c("bootstrap", "jackknife")
  )

method_bmlmi(
  covariance = c("us", "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", "toeph"),
  threshold = 0.01,
  same_cov = TRUE,
  REML = TRUE,
  B = 20,
  D = 2
)

```

Arguments

covariance	a character string that specifies the structure of the covariance matrix to be used in the imputation model. Must be one of "us" (default), "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", or "toeph". See details.
same_cov	a logical, if TRUE the imputation model will be fitted using a single shared covariance matrix for all observations. If FALSE a separate covariance matrix will be fit for each group as determined by the group argument of <code>set_vars()</code> .
n_samples	a numeric that determines how many imputed datasets are generated. In the case of <code>method_condmean(type = "jackknife")</code> this argument must be set to NULL. See details.
prior_cov	a character string that specifies the prior used for the covariance model parameters. Must be one of "default" (default) or "lkj" (for the unstructured covariance model). See the Statistical Specifications vignette for details.
control	a list which specifies further lower level details of the computations. Currently only used by <code>method_bayes()</code> , please see control_bayes() for details and default settings.
burn_in	deprecated. Please use the warmup argument in control_bayes() instead.
burn_between	deprecated. Please use the thin argument in control_bayes() instead.
threshold	a numeric between 0 and 1, specifies the proportion of bootstrap datasets that can fail to produce valid samples before an error is thrown. See details.
REML	a logical indicating whether to use REML estimation rather than maximum likelihood.
type	a character string that specifies the resampling method used to perform inference when a conditional mean imputation approach (set via <code>method_condmean()</code>) is used. Must be one of "bootstrap" or "jackknife".
B	a numeric that determines the number of bootstrap samples for <code>method_bmlmi</code> .
D	a numeric that determines the number of random imputations for each bootstrap sample. Needed for <code>method_bmlmi()</code> .

Details

In the case of `method_condmean(type = "bootstrap")` there will be `n_samples + 1` imputation models and datasets generated as the first sample will be based on the original dataset whilst the other `n_samples` samples will be bootstrapped datasets. Likewise, for `method_condmean(type = "jackknife")` there will be `length(unique(data$subjid)) + 1` imputation models and datasets generated. In both cases this is represented by `n + 1` being displayed in the print message. In the case that `method_bayes()` is used, and with the `control` argument the number of chains is set to more than 1, then the `n_samples` samples will be distributed across the chains. The total number of returned samples will still be `n_samples`.

The user is able to specify different covariance structures using the `covariance` argument. Currently supported structures include:

- Unstructured ("us") (default)
- Ante-dependence ("ad")
- Heterogeneous ante-dependence ("adh")
- First-order auto-regressive ("ar1")
- Heterogeneous first-order auto-regressive ("ar1h")
- Compound symmetry ("cs")
- Heterogeneous compound symmetry ("csh")
- Toeplitz ("toep")
- Heterogeneous Toeplitz ("toeph")

For full details please see `mmrm:cov_types()`.

In the case of `method_condmean(type = "bootstrap")`, `method_approxbayes()` and `method_bmlmi()` repeated bootstrap samples of the original dataset are taken with an MMRM fitted to each sample. Due to the randomness of these sampled datasets, as well as limitations in the optimisers used to fit the models, it is not uncommon that estimates for a particular dataset can't be generated. In these instances `rbmi` is designed to throw out that bootstrapped dataset and try again with another. However to ensure that these errors are due to chance and not due to some underlying misspecification in the data and/or model a tolerance limit is set on how many samples can be discarded. Once the tolerance limit has been reached an error will be thrown and the process aborted. The tolerance limit is defined as `ceiling(threshold * n_samples)`. Note that for the jackknife method estimates need to be generated for all leave-one-out datasets and as such an error will be thrown if any of them fail to fit.

Please note that at the time of writing (September 2021) Stan is unable to produce reproducible samples across different operating systems even when the same seed is used. As such care must be taken when using Stan across different machines. For more information on this limitation please consult the Stan documentation https://mc-stan.org/docs/2_27/reference-manual/reproducibility-chapter.html

Value

A method object; a named list of class `c("method", <type>)` where `<type>` is one of "bayes", "approxbayes", "condmean" or "bmlmi". It stores the chosen imputation methodology and its settings and is passed to the `method` argument of `draws()`.

Examples

```
# Bayesian multiple imputation with 150 imputed datasets
method_bayes(
  n_samples = 150,
  control = control_bayes(warmup = 200, thin = 5)
)

# Approximate Bayesian multiple imputation with 20 imputed datasets
method_approxbayes(n_samples = 20)

# Conditional mean imputation with jackknife resampling
method_condmean(type = "jackknife")

# Conditional mean imputation with 100 bootstrap samples
method_condmean(type = "bootstrap", n_samples = 100)

# Bootstrapped maximum likelihood multiple imputation with D = 2 random
# imputations for each bootstrap sample and B = 20 bootstrap samples
method_bmlmi(B = 20, D = 2)
```

parametric_ci

Calculate parametric confidence intervals

Description

Calculates confidence intervals based upon a parametric distribution.

Usage

```
parametric_ci(point, se, alpha, alternative, qfun, pfun, ...)
```

Arguments

point	The point estimate.
se	The standard error of the point estimate. If using a non-"normal" distribution this should be set to 1.
alpha	The type 1 error rate, should be a value between 0 and 1.
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less".
qfun	The quantile function for the assumed distribution i.e. qnorm.
pfun	The CDF function for the assumed distribution i.e. pnorm.
...	additional arguments passed on qfun and pfun i.e. df = 102.

par_lapply	<i>Parallelise lapply</i>
------------	---------------------------

Description

Simple wrapper around `lapply` and `parallel::clusterApplyLB` to abstract away the logic of deciding which one to use

Usage

```
par_lapply(cl, fun, x, ...)
```

Arguments

<code>cl</code>	Cluster created by <code>parallel::makeCluster()</code> or <code>NULL</code>
<code>fun</code>	Function to be run
<code>x</code>	object to be looped over
<code>...</code>	extra arguments passed to <code>fun</code>

pool	<i>Pool analysis results obtained from the imputed datasets</i>
------	---

Description

Pool analysis results obtained from the imputed datasets

Usage

```
pool(
  results,
  conf.level = 0.95,
  alternative = c("two.sided", "less", "greater"),
  type = c("percentile", "normal")
)

## S3 method for class 'pool'
as.data.frame(x, ...)

## S3 method for class 'pool'
print(x, ...)

mcse(x, results)

## S3 method for class 'mcse'
```

```
as.data.frame(x, ...)

## S3 method for class 'mcse'
print(x, ..., pval_digits = 2, pval_eps = 1e-06, pval_nsmall = 5)
```

Arguments

<code>results</code>	an analysis object created by analyse() .
<code>conf.level</code>	confidence level of the returned confidence interval. Must be a single number between 0 and 1. Default is 0.95.
<code>alternative</code>	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less".
<code>type</code>	a character string of either "percentile" (default) or "normal". Determines what method should be used to calculate the bootstrap confidence intervals. See details. Only used if <code>method_condmean(type = "bootstrap")</code> was specified in the original call to draws() .
<code>x</code>	a pool object generated by pool() .
<code>...</code>	not used.
<code>pval_digits</code>	number of significant digits to print for p-values' MCSE.
<code>pval_eps</code>	the minimum p-values' MCSE to print.
<code>pval_nsmall</code>	the minimum number of digits to print for p-values' MCSE.

Details

The calculation used to generate the point estimate, standard errors and confidence interval depends upon the method specified in the original call to [draws\(\)](#); In particular:

- `method_approxbayes()` & `method_bayes()` both use Rubin's rules to pool estimates and variances across multiple imputed datasets, and the Barnard-Rubin rule to pool degree's of freedom; see Little & Rubin (2002). Here, the `mcse()` function can compute the Monte Carlo standard error (MCSE) of the pooled estimates, via a Jackknife variance estimator for all parameters; see Efron & Gong (1983) and Royston, Carlin & White (2009).
- `method_condmean(type = "bootstrap")` uses percentile or normal approximation; see Efron & Tibshirani (1994). Note that for the percentile bootstrap, no standard error is calculated, i.e. the standard errors will be NA in the object / `data.frame`.
- `method_condmean(type = "jackknife")` uses the standard jackknife variance formula; see Efron & Tibshirani (1994).
- `method_bmlmi` uses pooling procedure for Bootstrapped Maximum Likelihood MI (BMLMI). See Von Hippel & Bartlett (2021).

Value

A pool object; a list of class "pool" containing the pooled analysis results with the following elements:

- `pars`: a named list with one entry per parameter, each itself a list containing the pooled point estimate (`est`), confidence interval (`ci`), standard error (`se`) and p-value (`pvalue`).

- `conf.level`: the confidence level used for the confidence intervals.
- `alternative`: the alternative hypothesis used to derive the p-values.
- `N`: the number of analysis results that were combined.
- `method`: the pooling method that was used.

The `as.data.frame()` method returns a `data.frame` with one row per parameter (columns `parameter`, `est`, `se`, `lci`, `uci`, `pval`) and the `print()` method returns its input invisibly.

`mcse()` returns an `mcse` object; a list of class "mcse" containing `pars` (the Monte Carlo standard errors of the pooled estimates, in the same structure as the `pars` element of a `pool` object) and `N` (the number of results combined).

References

Bradley Efron and Robert J Tibshirani. An introduction to the bootstrap. CRC press, 1994. [Section 11]

Bradley Efron and Gail Gong. A leisurely look at the bootstrap, the jackknife, and cross-validation. The American Statistician, 37(1):36-48, 1983.

Roderick J. A. Little and Donald B. Rubin. Statistical Analysis with Missing Data, Second Edition. John Wiley & Sons, Hoboken, New Jersey, 2002. [Section 5.4]

Royston, P., Carlin, J. B., & White, I. R. Multiple imputation of missing values: New features for `mim`. Stata Journal, 9(2): 252-264, 2009.

Von Hippel, Paul T and Bartlett, Jonathan W. Maximum likelihood multiple imputation: Faster imputations and consistent standard errors without posterior draws. 2021.

Examples

```
# Prepare the data: expand to one row per subject and visit
dat <- expand_locf(
  antidepressant_data,
  PATIENT = levels(antidepressant_data$PATIENT),
  VISIT = levels(antidepressant_data$VISIT),
  vars = c("BASVAL", "THERAPY"),
  group = c("PATIENT"),
  order = c("PATIENT", "VISIT")
)

# Derive data_ice: first missing-outcome visit per patient, imputed under JR.
# Subject 3618 has only an intermittent missing value and is removed.
is_missing <- is.na(dat$CHANGE)
dat_ice <- dat[is_missing, c("PATIENT", "VISIT")]
dat_ice <- dat_ice[!duplicated(dat_ice$PATIENT), ]
dat_ice$strategy <- "JR"
dat_ice <- dat_ice[dat_ice$PATIENT != 3618, ]

vars <- set_vars(
  outcome = "CHANGE",
  visit = "VISIT",
  subjid = "PATIENT",
  group = "THERAPY",
```

```

    covariates = c("BASVAL*VISIT", "THERAPY*VISIT")
  )

  # Approximate Bayesian imputation, so that pooling uses Rubin's rules
  set.seed(987)
  drawObj <- draws(
    data = dat,
    data_ice = dat_ice,
    vars = vars,
    method = method_approxbayes(n_samples = 20),
    quiet = TRUE
  )
  imputeObj <- impute(
    drawObj,
    references = c("DRUG" = "PLACEBO", "PLACEBO" = "PLACEBO")
  )
  anaObj <- analyse(
    imputeObj,
    ancova,
    vars = set_vars(
      subjid = "PATIENT",
      outcome = "CHANGE",
      visit = "VISIT",
      group = "THERAPY",
      covariates = "BASVAL"
    )
  )
)

# Pool the analysis results across the imputed datasets
poolObj <- pool(anaObj)
poolObj

# Monte Carlo standard errors of the pooled estimates
mcse(poolObj, anaObj)

```

pool_bootstrap_normal *Bootstrap Pooling via normal approximation*

Description

Get point estimate, confidence interval and p-value using the normal approximation.

Usage

```
pool_bootstrap_normal(est, conf.level, alternative)
```

Arguments

<code>est</code>	a numeric vector of point estimates from each bootstrap sample.
<code>conf.level</code>	confidence level of the returned confidence interval. Must be a single number between 0 and 1. Default is 0.95.
<code>alternative</code>	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less".

Details

The point estimate is taken to be the first element of `est`. The remaining `n-1` values of `est` are then used to generate the confidence intervals.

`pool_bootstrap_percentile`

Bootstrap Pooling via Percentiles

Description

Get point estimate, confidence interval and p-value using percentiles. Note that quantile "type=6" is used, see [stats::quantile\(\)](#) for details.

Usage

```
pool_bootstrap_percentile(est, conf.level, alternative)
```

Arguments

<code>est</code>	a numeric vector of point estimates from each bootstrap sample.
<code>conf.level</code>	confidence level of the returned confidence interval. Must be a single number between 0 and 1. Default is 0.95.
<code>alternative</code>	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less".

Details

The point estimate is taken to be the first element of `est`. The remaining `n-1` values of `est` are then used to generate the confidence intervals.

prepare_stan_data	<i>Prepare input data to run the Stan model</i>
-------------------	---

Description

Prepare input data to run the Stan model. Creates / calculates all the required inputs as required by the `data{}` block of the MMRM Stan program.

Usage

```
prepare_stan_data(ddat, subjid, visit, outcome, group)
```

Arguments

<code>ddat</code>	A design matrix
<code>subjid</code>	Character vector containing the subjects IDs.
<code>visit</code>	Vector containing the visits.
<code>outcome</code>	Numeric vector containing the outcome variable.
<code>group</code>	Vector containing the group variable.

Details

- The `group` argument determines which covariance matrix group the subject belongs to. If you want all subjects to use a shared covariance matrix then set `group` to "1" for everyone.

Value

A `stan_data` object. A named list as per `data{}` block of the related Stan file. In particular it returns:

- `N` - The number of rows in the design matrix
- `P` - The number of columns in the design matrix
- `G` - The number of distinct covariance matrix groups (i.e. `length(unique(group))`)
- `n_visit` - The number of unique outcome visits
- `n_pat` - The total number of pattern groups (as defined by missingness patterns & covariance group)
- `pat_G` - Index for which Sigma each pattern group should use
- `pat_n_pt` - number of patients within each pattern group
- `pat_n_visit` - number of non-missing visits in each pattern group
- `pat_sigma_index` - rows/cols from Sigma to subset on for the pattern group (padded by 0's)
- `y` - The outcome variable
- `Q` - design matrix (after QR decomposition)
- `R` - R matrix from the QR decomposition of the design matrix

print.analysis	<i>Print analysis object</i>
----------------	------------------------------

Description

Print analysis object

Usage

```
## S3 method for class 'analysis'  
print(x, ...)
```

Arguments

x	An analysis object generated by analyse() .
...	Not used.

print.draws	<i>Print draws object</i>
-------------	---------------------------

Description

Print draws object

Usage

```
## S3 method for class 'draws'  
print(x, ...)
```

Arguments

x	A draws object generated by draws() .
...	not used.

print.imputation	<i>Print imputation object</i>
------------------	--------------------------------

Description

Print imputation object

Usage

```
## S3 method for class 'imputation'
print(x, ...)
```

Arguments

x	An imputation object generated by <code>impute()</code> .
...	Not used.

progressLogger	<i>R6 Class for printing current sampling progress</i>
----------------	--

Description

Object is initialised with total number of iterations that are expected to occur. User can then update the object with the add method to indicate how many more iterations have just occurred. Every time $\text{step} * 100\%$ of iterations have occurred a message is printed to the console. Use the quiet argument to prevent the object from printing anything at all

Public fields

step	real, percentage of iterations to allow before printing the progress to the console
step_current	integer, the total number of iterations completed since progress was last printed to the console
n	integer, the current number of completed iterations
n_max	integer, total number of expected iterations to be completed acts as the denominator for calculating progress percentages
quiet	logical holds whether or not to print anything

Methods

Public methods:

- `progressLogger$new()`
- `progressLogger$add()`
- `progressLogger$print_progress()`
- `progressLogger$clone()`

`progressLogger$new()`: Create progressLogger object

Usage:

```
progressLogger$new(n_max, quiet = FALSE, step = 0.1)
```

Arguments:

`n_max` integer, sets field `n_max`

`quiet` logical, sets field `quiet`

`step` real, sets field `step`

`progressLogger$add()`: Records that `n` more iterations have been completed this will add that number to the current step count (`step_current`) and will print a progress message to the log if the step limit (`step`) has been reached. This function will do nothing if `quiet` has been set to `TRUE`

Usage:

```
progressLogger$add(n)
```

Arguments:

`n` the number of successfully complete iterations since `add()` was last called

`progressLogger$print_progress()`: method to print the current state of progress

Usage:

```
progressLogger$print_progress()
```

`progressLogger$clone()`: The objects of this class are cloneable with this method.

Usage:

```
progressLogger$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

pval_percentile	<i>P-value of percentile bootstrap</i>
-----------------	--

Description

Determines the (not necessarily unique) quantile (type=6) of "est" which gives a value of 0. From this, derive the p-value corresponding to the percentile bootstrap via inversion.

Usage

```
pval_percentile(est)
```

Arguments

est a numeric vector of point estimates from each bootstrap sample.

Details

The p-value for $H_0: \theta=0$ vs $H_A: \theta>0$ is the value α for which $q_\alpha = 0$. If there is at least one estimate equal to zero it returns the largest α such that $q_\alpha = 0$. If all bootstrap estimates are > 0 it returns 0; if all bootstrap estimates are < 0 it returns 1. Analogous reasoning is applied for the p-value for $H_0: \theta=0$ vs $H_A: \theta<0$.

Value

A named numeric vector of length 2 containing the p-value for $H_0: \theta=0$ vs $H_A: \theta>0$ ("pval_greater") and the p-value for $H_0: \theta=0$ vs $H_A: \theta<0$ ("pval_less").

QR_decomp	<i>QR decomposition</i>
-----------	-------------------------

Description

QR decomposition as defined in the [Stan user's guide \(section 1.2\)](#).

Usage

```
QR_decomp(mat)
```

Arguments

mat A matrix to perform the QR decomposition on.

random_effects_expr	<i>Construct random effects formula</i>
---------------------	---

Description

Constructs a character representation of the random effects formula for fitting a MMRM for subject by visit in the format required for `mmrm::mmrm()`.

Usage

```
random_effects_expr(
  cov_struct = c("us", "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", "toeph"),
  cov_by_group = FALSE
)
```

Arguments

cov_struct	Character - The covariance structure to be used, must be one of "us" (default), "ad", "adh", "ar1", "ar1h", "cs", "csh", "toep", or "toeph")
cov_by_group	Boolean - Whenever or not to use separate covariances per each group level

Details

For example assuming the user specified a covariance structure of "us" and that no groups were provided this will return

```
us(visit | subjid)
```

If cov_by_group is set to FALSE then this indicates that separate covariance matrices are required per group and as such the following will be returned:

```
us( visit | group / subjid )
```

rbmi-settings	<i>rbmi settings</i>
---------------	----------------------

Description

Define settings that modify the behaviour of the rbmi package

Each of the following are the name of options that can be set via:

```
options(<option_name> = <value>)
```

`rbmi.cache_dir:`

Default = `tempfile()`

Directory to store compiled Stan models in to avoid having to re-compile. If the environment variable `RBMI_CACHE_DIR` has been set this will be used as the default value. Note that if you are running `rbmi` in multiple R processes at the same time (that is say multiple calls to `Rscript` at once) then there is a theoretical risk of the processes breaking each other as they attempt to read/write to the same cache folder at the same time. To avoid this potential issue it is recommended to leave this value at the default which will result in a unique cache for each process

`rbmi.enable_cache:`

Default = `TRUE`

If `TRUE` then the package will attempt to cache compiled Stan models to the `rbmi.cache_dir` directory. If `FALSE` then the package will re-compile the Stan model each time it is required. If the environment variable `RBMI_ENABLE_CACHE` has been set this will be used as the default value.

Usage

```
set_options()
```

Examples

```
## Not run:
options(rbmi.cache_dir = "some/directory/path")
options(rbmi.enable_cache = FALSE)

## End(Not run)
```

record

Capture all Output

Description

This function silences all warnings, errors & messages and instead returns a list containing the results (if it didn't error) + the warning and error messages as character vectors.

Usage

```
record(expr)
```

Arguments

`expr` An expression to be executed

Value

A list containing

- **results** - The object returned by `expr` or `list()` if an error was thrown
- **warnings** - NULL or a character vector if warnings were thrown
- **errors** - NULL or a string if an error was thrown
- **messages** - NULL or a character vector if messages were produced

Examples

```
## Not run:
record({
  x <- 1
  y <- 2
  warning("something went wrong")
  message("O nearly done")
  x + y
})

## End(Not run)
```

recursive_reduce	<i>recursive_reduce</i>
------------------	-------------------------

Description

Utility function used to replicate `purrr::reduce`. Recursively applies a function to a list of elements until only 1 element remains

Usage

```
recursive_reduce(.l, .f)
```

Arguments

<code>.l</code>	list of values to apply a function to
<code>.f</code>	function to apply to each element of the list in turn i.e. <code>.l[[1]] <- .f(.l[[1]] , .l[[2]]) ; .l[[2]]</code>

remove_if_all_missing	<i>Remove subjects from dataset if they have no observed values</i>
-----------------------	---

Description

This function takes a `data.frame` with variables `visit`, `outcome` & `subjid`. It then removes all rows for a given `subjid` if they don't have any non-missing values for `outcome`.

Usage

```
remove_if_all_missing(dat)
```

Arguments

<code>dat</code>	a <code>data.frame</code>
------------------	---------------------------

<code>rubin_df</code>	<i>Barnard and Rubin degrees of freedom adjustment</i>
-----------------------	--

Description

Compute degrees of freedom according to the Barnard-Rubin formula.

Usage

```
rubin_df(v_com, var_b, var_t, M)
```

Arguments

<code>v_com</code>	Positive number representing the degrees of freedom in the complete-data analysis.
<code>var_b</code>	Between-variance of point estimate across multiply imputed datasets.
<code>var_t</code>	Total-variance of point estimate according to Rubin's rules.
<code>M</code>	Number of imputations.

Details

The computation takes into account limit cases where there is no missing data (i.e. the between-variance `var_b` is zero) or where the complete-data degrees of freedom is set to `Inf`. Moreover, if `v_com` is given as `NA`, the function returns `Inf`.

Value

Degrees of freedom according to Barnard-Rubin formula. See Barnard-Rubin (1999).

References

Barnard, J. and Rubin, D.B. (1999). Small sample degrees of freedom with multiple imputation. *Biometrika*, 86, 948-955.

rubin_rules

Combine estimates using Rubin's rules

Description

Pool together the results from M complete-data analyses according to Rubin's rules. See details.

Usage

```
rubin_rules(ests, ses, v_com)
```

Arguments

<code>ests</code>	Numeric vector containing the point estimates from the complete-data analyses.
<code>ses</code>	Numeric vector containing the standard errors from the complete-data analyses.
<code>v_com</code>	Positive number representing the degrees of freedom in the complete-data analysis.

Details

`rubin_rules` applies Rubin's rules (Rubin, 1987) for pooling together the results from a multiple imputation procedure. The pooled point estimate `est_point` is the average across the point estimates from the complete-data analyses (given by the input argument `ests`). The total variance `var_t` is the sum of two terms representing the within-variance and the between-variance (see Little-Rubin (2002)). The function also returns `df`, the estimated pooled degrees of freedom according to Barnard-Rubin (1999) that can be used for inference based on the t-distribution.

Value

A list containing:

- `est_point`: the pooled point estimate according to Little-Rubin (2002).
- `var_t`: total variance according to Little-Rubin (2002).
- `df`: degrees of freedom according to Barnard-Rubin (1999).

References

Barnard, J. and Rubin, D.B. (1999). Small sample degrees of freedom with multiple imputation. *Biometrika*, 86, 948-955

Roderick J. A. Little and Donald B. Rubin. *Statistical Analysis with Missing Data*, Second Edition. John Wiley & Sons, Hoboken, New Jersey, 2002. [Section 5.4]

See Also

[rubin_df\(\)](#) for the degrees of freedom estimation.

sample_ids	<i>Sample Patient Ids</i>
------------	---------------------------

Description

Performs a stratified bootstrap sample of IDS ensuring the return vector is the same length as the input vector

Usage

```
sample_ids(ids, strata = rep(1, length(ids)))
```

Arguments

ids	vector to sample from
strata	strata indicator, ids are sampled within each strata ensuring the that the numbers of each strata are maintained

Examples

```
## Not run:  
sample_ids( c("a", "b", "c", "d"), strata = c(1,1,2,2))  
  
## End(Not run)
```

sample_list	<i>Create and validate a sample_list object</i>
-------------	---

Description

Given a list of sample_single objects generate by [sample_single\(\)](#), creates a sample_list objects and validate it.

Usage

```
sample_list(...)
```

Arguments

...	A list of sample_single objects.
-----	----------------------------------

sample_mvnorm

Sample random values from the multivariate normal distribution

Description

Sample random values from the multivariate normal distribution

Usage

```
sample_mvnorm(mu, sigma)
```

Arguments

mu	mean vector
sigma	covariance matrix

Samples multivariate normal variables by multiplying univariate random normal variables by the Cholesky decomposition of the covariance matrix.
If mu is length 1 then just uses rnorm instead.

sample_single

Create object of sample_single class

Description

Creates an object of class sample_single which is a named list containing the input parameters and validate them.

Usage

```
sample_single(
  ids,
  beta = NA,
  sigma = NA,
  theta = NA,
  failed = any(is.na(beta)),
  ids_samp = ids
)
```

Arguments

ids	Vector of characters containing the ids of the subjects included in the original dataset.
beta	Numeric vector of estimated regression coefficients.
sigma	List of estimated covariance matrices (one for each level of vars\$group).

theta	Numeric vector of transformed covariances.
failed	Logical. TRUE if the model fit failed.
ids_samp	Vector of characters containing the ids of the subjects included in the given sample.

Value

A named list of class `sample_single`. It contains the following:

- `ids` vector of characters containing the ids of the subjects included in the original dataset.
- `beta` numeric vector of estimated regression coefficients.
- `sigma` list of estimated covariance matrices (one for each level of `vars$group`).
- `theta` numeric vector of transformed covariances.
- `failed` logical. TRUE if the model fit failed.
- `ids_samp` vector of characters containing the ids of the subjects included in the given sample.

scalerConstructor	<i>R6 Class for scaling (and un-scaling) design matrices</i>
-------------------	--

Description

Scales a design matrix so that all non-categorical columns have a mean of 0 and an standard deviation of 1.

Details

The object initialisation is used to determine the relevant means and standard deviations to scale by and then the scaling (and un-scaling) itself is performed by the relevant object methods.

Un-scaling is done on linear model Beta and Sigma coefficients. For this purpose the first column on the dataset to be scaled is assumed to be the outcome variable with all other variables assumed to be post-transformation predictor variables (i.e. all dummy variables have already been expanded).

Public fields

`centre` Vector of column means. The first value is the outcome variable, all other variables are the predictors.

`scales` Vector of column standard deviations. The first value is the outcome variable, all other variables are the predictors.

Methods

Public methods:

- `scaler$new()`
- `scaler$scale()`
- `scaler$unscale_sigma()`
- `scaler$unscale_beta()`
- `scaler$clone()`

`scaler$new()`: Uses `dat` to determine the relevant column means and standard deviations to use when scaling and un-scaling future datasets. Implicitly assumes that new datasets have the same column order as `dat`

Usage:

```
scaler$new(dat)
```

Arguments:

`dat` A `data.frame` or matrix. All columns must be numeric (i.e dummy variables, must have already been expanded out).

Details: Categorical columns (as determined by those who's values are entirely 1 or 0) will not be scaled. This is achieved by setting the corresponding values of centre to 0 and scale to 1.

`scaler$scale()`: Scales a dataset so that all continuous variables have a mean of 0 and a standard deviation of 1.

Usage:

```
scaler$scale(dat)
```

Arguments:

`dat` A `data.frame` or matrix whose columns are all numeric (i.e. dummy variables have all been expanded out) and whose columns are in the same order as the dataset used in the initialization function.

`scaler$unscale_sigma()`: Unscales a sigma value (or matrix) as estimated by a linear model using a design matrix scaled by this object. This function only works if the first column of the initialisation `data.frame` was the outcome variable.

Usage:

```
scaler$unscale_sigma(sigma)
```

Arguments:

`sigma` A numeric value or matrix.

Returns: A numeric value or matrix

`scaler$unscale_beta()`: Unscales a beta value (or vector) as estimated by a linear model using a design matrix scaled by this object. This function only works if the first column of the initialization `data.frame` was the outcome variable.

Usage:

```
scaler$unscale_beta(beta)
```

Arguments:

beta A numeric vector of beta coefficients as estimated from a linear model.

Returns: A numeric vector.

scaler\$clone(): The objects of this class are cloneable with this method.

Usage:

```
scaler$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

set_simul_pars	<i>Set simulation parameters of a study group.</i>
----------------	--

Description

This function provides input arguments for each study group needed to simulate data with [simulate_data\(\)](#). [simulate_data\(\)](#) generates data for a two-arms clinical trial with longitudinal continuous outcomes and two intercurrent events (ICEs). ICE1 may be thought of as a discontinuation from study treatment due to study drug or condition related (SDCR) reasons. ICE2 may be thought of as discontinuation from study treatment due to uninformative study drop-out, i.e. due to not study drug or condition related (NSDCR) reasons and outcome data after ICE2 is always missing.

Usage

```
set_simul_pars(
  mu,
  sigma,
  n,
  prob_ice1 = 0,
  or_outcome_ice1 = 1,
  prob_post_ice1_dropout = 0,
  prob_ice2 = 0,
  prob_miss = 0
)
```

Arguments

mu	Numeric vector describing the mean outcome trajectory at each visit (including baseline) assuming no ICEs.
sigma	Covariance matrix of the outcome trajectory assuming no ICEs.
n	Number of subjects belonging to the group.
prob_ice1	Numeric vector that specifies the probability of experiencing ICE1 (discontinuation from study treatment due to SDCR reasons) after each visit for a subject with observed outcome at that visit equal to the mean at baseline (mu[1]). If a single numeric is provided, then the same probability is applied to each visit.

or_outcome_ice1	Numeric value that specifies the odds ratio of experiencing ICE1 after each visit corresponding to a +1 higher value of the observed outcome at that visit.
prob_post_ice1_dropout	Numeric value that specifies the probability of study drop-out following ICE1. If a subject is simulated to drop-out after ICE1, all outcomes after ICE1 are set to missing.
prob_ice2	Numeric that specifies an additional probability that a post-baseline visit is affected by study drop-out. Outcome data at the subject's first simulated visit affected by study drop-out and all subsequent visits are set to missing. This generates a second intercurrent event ICE2, which may be thought as treatment discontinuation due to NSDCR reasons with subsequent drop-out. If for a subject, both ICE1 and ICE2 are simulated to occur, then it is assumed that only the earlier of them counts. In case both ICEs are simulated to occur at the same time, it is assumed that ICE1 counts. This means that a single subject can experience either ICE1 or ICE2, but not both of them.
prob_miss	Numeric value that specifies an additional probability for a given post-baseline observation to be missing. This can be used to produce "intermittent" missing values which are not associated with any ICE.

Details

For the details, please see [simulate_data\(\)](#).

Value

A simul_pars object which is a named list containing the simulation parameters.

See Also

[simulate_data\(\)](#)

Examples

```
# Simulation parameters for a group with 5 visits (including baseline)
time <- c(0, 3, 6, 9, 12)

# Mean outcome trajectory
mu <- c(50.0, 52.5, 55.0, 57.5, 60.0)

# Covariance matrix implied by a random intercept and slope model
sd_error <- 2.5
covRE <- rbind(
  c(25.0, 6.25),
  c(6.25, 25.0)
)
sigma <- cbind(1, time / 12) %*% covRE %*% rbind(1, time / 12) +
  diag(sd_error^2, nrow = length(time))

set_simul_pars(
```

```

    mu = mu,
    sigma = sigma,
    n = 100,
    prob_ice1 = 0.03,
    or_outcome_ice1 = 1.10,
    prob_post_ice1_dropout = 0.5
  )

```

set_vars

Set key variables

Description

This function is used to define the names of key variables within the `data.frame`'s that are provided as input arguments to `draws()` and `ancova()`.

Usage

```

set_vars(
  subjid = "subjid",
  visit = "visit",
  outcome = "outcome",
  group = "group",
  covariates = character(0),
  strata = group,
  strategy = "strategy"
)

```

Arguments

subjid	The name of the "Subject ID" variable. A length 1 character vector.
visit	The name of the "Visit" variable. A length 1 character vector.
outcome	The name of the "Outcome" variable. A length 1 character vector.
group	The name of the "Group" variable. A length 1 character vector.
covariates	The name of any covariates to be used in the context of modelling. See details.
strata	The name of the any stratification variable to be used in the context of bootstrap sampling. See details.
strategy	The name of the "strategy" variable. A length 1 character vector.

Details

In both `draws()` and `ancova()` the `covariates` argument can be specified to indicate which variables should be included in the imputation and analysis models respectively. If you wish to include interaction terms these need to be manually specified i.e. `covariates = c("group*visit",`

"age*sex"). Please note that the use of the `I()` function to inhibit the interpretation/conversion of objects is not supported.

Currently `strata` is only used by `draws()` in combination with `method_condmean(type = "bootstrap")` and `method_approxbayes()` in order to allow for the specification of stratified bootstrap sampling. By default `strata` is set equal to the value of `group` as it is assumed most users will want to preserve the group size between samples. See `draws()` for more details.

Likewise, currently the `strategy` argument is only used by `draws()` to specify the name of the strategy variable within the `data_ice` data.frame. See `draws()` for more details.

Value

A vars object; a named list of class `ivars` recording the names of the key variables (`subjid`, `visit`, `outcome`, `group`, `covariates`, `strata` and `strategy`) used throughout `rbmi` by functions such as `draws()`, `ancova()` and `analyse()`.

See Also

`draws()`
`ancova()`

Examples

```
## Not run:

# Using CDISC variable names as an example
set_vars(
  subjid = "usubjid",
  visit = "avisit",
  outcome = "aval",
  group = "arm",
  covariates = c("bwt", "bht", "arm * avisit"),
  strategy = "strat"
)

## End(Not run)
```

simulate_data

Generate data

Description

Generate data for a two-arms clinical trial with longitudinal continuous outcome and two inter-current events (ICEs). ICE1 may be thought of as a discontinuation from study treatment due to study drug or condition related (SDCR) reasons. ICE2 may be thought of as discontinuation from study treatment due to uninformative study drop-out, i.e. due to not study drug or condition related (NSDCR) reasons and outcome data after ICE2 is always missing.

Usage

```
simulate_data(pars_c, pars_t, post_ice1_traj, strategies = getStrategies())
```

Arguments

<code>pars_c</code>	A <code>simul_pars</code> object as generated by <code>set_simul_pars()</code> . It specifies the simulation parameters of the control arm.
<code>pars_t</code>	A <code>simul_pars</code> object as generated by <code>set_simul_pars()</code> . It specifies the simulation parameters of the treatment arm.
<code>post_ice1_traj</code>	A string which specifies how observed outcomes occurring after ICE1 are simulated. Must target a function included in <code>strategies</code> . Possible choices are: Missing At Random "MAR", Jump to Reference "JR", Copy Reference "CR", Copy Increments in Reference "CIR", Last Mean Carried Forward "LMCF". User-defined strategies could also be added. See <code>getStrategies()</code> for details.
<code>strategies</code>	A named list of functions. Default equal to <code>getStrategies()</code> . See <code>getStrategies()</code> for details.

Details

The data generation works as follows:

- Generate outcome data for all visits (including baseline) from a multivariate normal distribution with parameters `pars_c$mu` and `pars_c$sigma` for the control arm and parameters `pars_t$mu` and `pars_t$sigma` for the treatment arm, respectively. Note that for a randomized trial, outcomes have the same distribution at baseline in both treatment groups, i.e. one should set `pars_c$mu[1]=pars_t$mu[1]` and `pars_c$sigma[1,1]=pars_t$sigma[1,1]`.
- Simulate whether ICE1 (study treatment discontinuation due to SDCR reasons) occurs after each visit according to parameters `pars_c$prob_ice1` and `pars_c$or_outcome_ice1` for the control arm and `pars_t$prob_ice1` and `pars_t$or_outcome_ice1` for the treatment arm, respectively.
- Simulate drop-out following ICE1 according to `pars_c$prob_post_ice1_dropout` and `pars_t$prob_post_ice1_dropout`.
- Simulate an additional uninformative study drop-out with probabilities `pars_c$prob_ice2` and `pars_t$prob_ice2` at each visit. This generates a second intercurrent event ICE2, which may be thought as treatment discontinuation due to NSDCR reasons with subsequent drop-out. The simulated time of drop-out is the subject's first visit which is affected by drop-out and data from this visit and all subsequent visits are consequently set to missing. If for a subject, both ICE1 and ICE2 are simulated to occur, then it is assumed that only the earlier of them counts. In case both ICEs are simulated to occur at the same time, it is assumed that ICE1 counts. This means that a single subject can experience either ICE1 or ICE2, but not both of them.
- Adjust trajectories after ICE1 according to the given assumption expressed with the `post_ice1_traj` argument. Note that only post-ICE1 outcomes in the intervention arm can be adjusted. Post-ICE1 outcomes from the control arm are not adjusted.
- Simulate additional intermittent missing outcome data as per arguments `pars_c$prob_miss` and `pars_t$prob_miss`.

The probability of the ICE after each visit is modelled according to the following logistic regression model: $\sim 1 + I(\text{visit} == 0) + \dots + I(\text{visit} == n_visits - 1) + I((x - \alpha))$ where:

- `n_visits` is the number of visits (including baseline).
- `alpha` is the baseline outcome mean. The term $I((x - \alpha))$ specifies the dependency of the probability of the ICE on the current outcome value. The corresponding regression coefficients of the logistic model are defined as follows: The intercept is set to 0, the coefficients corresponding to discontinuation after each visit for a subject with outcome equal to the mean at baseline are set according to parameters `pars_c$prob_ice1 (pars_t$prob_ice1)`, and the regression coefficient associated with the covariate $I((x - \alpha))$ is set to $\log(\text{pars_c\$or_outcome_ice1})$ ($\log(\text{pars_t\$or_outcome_ice1})$).

Please note that the baseline outcome cannot be missing nor be affected by any ICEs.

Value

A data.frame containing the simulated data. It includes the following variables:

- `id`: Factor variable that specifies the id of each subject.
- `visit`: Factor variable that specifies the visit of each assessment. Visit 0 denotes the baseline visit.
- `group`: Factor variable that specifies which treatment group each subject belongs to.
- `outcome_bl`: Numeric variable that specifies the baseline outcome.
- `outcome_noICE`: Numeric variable that specifies the longitudinal outcome assuming no ICEs.
- `ind_ice1`: Binary variable that takes value 1 if the corresponding visit is affected by ICE1 and 0 otherwise.
- `dropout_ice1`: Binary variable that takes value 1 if the corresponding visit is affected by the drop-out following ICE1 and 0 otherwise.
- `ind_ice2`: Binary variable that takes value 1 if the corresponding visit is affected by ICE2.
- `outcome`: Numeric variable that specifies the longitudinal outcome including ICE1, ICE2 and the intermittent missing values.

Examples

```
set.seed(1392)

time <- c(0, 3, 6, 9, 12)

# Mean outcome trajectory in the control and intervention groups
muC <- c(50.0, 52.5, 55.0, 57.5, 60.0)
muT <- c(50.0, 52.5, 55.0, 56.25, 57.50)

# Covariance matrix implied by a random intercept and slope model
sd_error <- 2.5
covRE <- rbind(
  c(25.0, 6.25),
  c(6.25, 25.0)
)
```

```

Sigma <- cbind(1, time / 12) %*% covRE %*% rbind(1, time / 12) +
  diag(sd_error^2, nrow = length(time))

# Simulation parameters of the control group
parsC <- set_simul_pars(
  mu = muC,
  sigma = Sigma,
  n = 100,
  prob_ice1 = 0.03,
  or_outcome_ice1 = 1.10,
  prob_post_ice1_dropout = 0.5
)

# Simulation parameters of the intervention group
parsT <- parsC
parsT$mu <- muT
parsT$prob_ice1 <- 0.04

# Simulate a dataset with a copy-increments-in-reference post-ICE1 trajectory
data <- simulate_data(
  pars_c = parsC,
  pars_t = parsT,
  post_ice1_traj = "CIR"
)
head(data)

```

simulate_dropout	<i>Simulate drop-out</i>
------------------	--------------------------

Description

Simulate drop-out

Usage

```
simulate_dropout(prob_dropout, ids, subset = rep(1, length(ids)))
```

Arguments

prob_dropout	Numeric that specifies the probability that a post-baseline visit is affected by study drop-out.
ids	Factor variable that specifies the id of each subject.
subset	Binary variable that specifies the subset that could be affected by drop-out. I.e. subset is a binary vector of length equal to the length of ids that takes value 1 if the corresponding visit could be affected by drop-out and 0 otherwise.

Details

subset can be used to specify outcome values that cannot be affected by the drop-out. By default subset will be set to 1 for all the values except the values corresponding to the baseline outcome, since baseline is supposed to not be affected by drop-out. Even if subset is specified by the user, the values corresponding to the baseline outcome are still hard-coded to be 0.

Value

A binary vector of length equal to the length of ids that takes value 1 if the corresponding outcome is affected by study drop-out.

simulate_ice	<i>Simulate intercurrent event</i>
--------------	------------------------------------

Description

Simulate intercurrent event

Usage

```
simulate_ice(outcome, visits, ids, prob_ice, or_outcome_ice, baseline_mean)
```

Arguments

outcome	Numeric variable that specifies the longitudinal outcome for a single group.
visits	Factor variable that specifies the visit of each assessment.
ids	Factor variable that specifies the id of each subject.
prob_ice	Numeric vector that specifies for each visit the probability of experiencing the ICE after the current visit for a subject with outcome equal to the mean at baseline. If a single numeric is provided, then the same probability is applied to each visit.
or_outcome_ice	Numeric value that specifies the odds ratio of the ICE corresponding to a +1 higher value of the outcome at the visit.
baseline_mean	Mean outcome value at baseline.

Details

The probability of the ICE after each visit is modelled according to the following logistic regression model: $\sim 1 + I(\text{visit} == 0) + \dots + I(\text{visit} == n_visits - 1) + I((x - \alpha))$ where:

- `n_visits` is the number of visits (including baseline).
- `alpha` is the baseline outcome mean set via argument `baseline_mean`. The term $I((x - \alpha))$ specifies the dependency of the probability of the ICE on the current outcome value. The corresponding regression coefficients of the logistic model are defined as follows: The intercept is set to 0, the coefficients corresponding to discontinuation after each visit for a subject with outcome equal to the mean at baseline are set according to parameter `or_outcome_ice`, and the regression coefficient associated with the covariate $I((x - \alpha))$ is set to $\log(\text{or_outcome_ice})$.

Value

A binary variable that takes value 1 if the corresponding outcome is affected by the ICE and 0 otherwise.

simulate_test_data	Create simulated datasets
--------------------	---------------------------

Description

Creates a longitudinal dataset in the format that rbmi was designed to analyse.

Usage

```
simulate_test_data(
  n = 200,
  sd = c(3, 5, 7),
  cor = c(0.1, 0.7, 0.4),
  mu = list(int = 10, age = 3, sex = 2, trt = c(0, 4, 8), visit = c(0, 1, 2))
)

as_vcov(sd, cor)
```

Arguments

n	the number of subjects to sample. Total number of observations returned is thus $n * \text{length}(\text{sd})$
sd	the standard deviations for the outcome at each visit. i.e. the square root of the diagonal of the covariance matrix for the outcome
cor	the correlation coefficients between the outcome values at each visit. See details.
mu	the coefficients to use to construct the mean outcome value at each visit. Must be a named list with elements int, age, sex, trt & visit. See details.

Details

The number of visits is determined by the size of the variance covariance matrix. i.e. if 3 standard deviation values are provided then 3 visits per patient will be created.

The covariates in the simulated dataset are produced as follows:

- Patients age is sampled at random from a $N(0,1)$ distribution
- Patients sex is sampled at random with a 50/50 split
- Patients group is sampled at random but fixed so that each group has $n/2$ patients
- The outcome variable is sampled from a multivariate normal distribution, see below for details

The mean for the outcome variable is derived as:

```
outcome = Intercept + age + sex + visit + treatment
```

The coefficients for the intercept, age and sex are taken from `mu$int`, `mu$age` and `mu$sex` respectively, all of which must be a length 1 numeric.

Treatment and visit coefficients are taken from `mu$trt` and `mu$visit` respectively and must either be of length 1 (i.e. a constant affect across all visits) or equal to the number of visits (as determined by the length of `sd`). I.e. if you wanted a treatment slope of 5 and a visit slope of 1 you could specify:

```
mu = list(..., "trt" = c(0,5,10), "visit" = c(0,1,2))
```

The correlation matrix is constructed from `cor` as follows. Let `cor = c(a, b, c, d, e, f)` then the correlation matrix would be:

```
1  a  b  d
a  1  c  e
b  c  1  f
d  e  f  1
```

Value

`as_vcov()` returns a symmetric positive-definite covariance matrix (a numeric matrix) constructed from the vector of standard deviations `sd` and the correlation entries `cor`.

Examples

```
# Construct a 3x3 covariance matrix from standard deviations and correlations.
# `cor` gives the upper-triangular correlations in row-major order:
# cor(1,2), cor(1,3), cor(2,3).
as_vcov(
  sd = c(1, 3, 2),
  cor = c(0.4, 0.5, 0.45)
)
```

sort_by	Sort data.frame
---------	-----------------

Description

Sorts a `data.frame` (ascending by default) based upon variables within the dataset

Usage

```
sort_by(df, vars = NULL, decreasing = FALSE)
```

Arguments

df	data.frame
vars	character vector of variables
decreasing	logical whether sort order should be in descending or ascending (default) order. Can be either a single logical value (in which case it is applied to all variables) or a vector which is the same length as <code>vars</code>

Examples

```
## Not run:
sort_by(iris, c("Sepal.Length", "Sepal.Width"), decreasing = c(TRUE, FALSE))

## End(Not run)
```

split_dim	<i>Transform array into list of arrays</i>
-----------	--

Description

Transform an array into list of arrays where the listing is performed on a given dimension.

Usage

```
split_dim(a, n)
```

Arguments

- a Array with number of dimensions at least 2.
- n Positive integer. Dimension of a to be listed.

Details

For example, if a is a 3 dimensional array and n = 1, split_dim(a,n) returns a list of 2 dimensional arrays (i.e. a list of matrices) where each element of the list is a[i, ,], where i takes values from 1 to the length of the first dimension of the array.

Example:

inputs: a <- array(c(1,2,3,4,5,6,7,8,9,10,11,12), dim = c(3,2,2)), which means that:

a[1,,]	a[2,,]	a[3,,]
[,1] [,2]	[,1] [,2]	[,1] [,2]
-----	-----	-----
1 7	2 8	3 9
4 10	5 11	6 12

```
n <- 1
```

output of res <- split_dim(a,n) is a list of 3 elements:

res[[1]]	res[[2]]	res[[3]]
[,1] [,2]	[,1] [,2]	[,1] [,2]
-----	-----	-----
1 7	2 8	3 9
4 10	5 11	6 12

Value

A list of length n of arrays with number of dimensions equal to the number of dimensions of a minus 1.

split_imputations	<i>Split a flat list of <code>imputation_single()</code> into multiple <code>imputation_df()</code>'s by ID</i>
-------------------	---

Description

Split a flat list of `imputation_single()` into multiple `imputation_df()`'s by ID

Usage

```
split_imputations(list_of_singles, split_ids)
```

Arguments

list_of_singles	A list of <code>imputation_single()</code> 's
split_ids	A list with 1 element per required split. Each element must contain a vector of "ID"'s which correspond to the <code>imputation_single()</code> ID's that are required within that sample. The total number of ID's must by equal to the length of list_of_singles

Details

This function converts a list of imputations from being structured per patient to being structured per sample i.e. it converts

```
obj <- list(
  imputation_single("Ben", numeric(0)),
  imputation_single("Ben", numeric(0)),
  imputation_single("Ben", numeric(0)),
  imputation_single("Harry", c(1, 2)),
  imputation_single("Phil", c(3, 4)),
  imputation_single("Phil", c(5, 6)),
  imputation_single("Tom", c(7, 8, 9))
)

index <- list(
  c("Ben", "Harry", "Phil", "Tom"),
  c("Ben", "Ben", "Phil")
)
```

Into:

```
output <- list(
  imputation_df(
    imputation_single(id = "Ben", values = numeric(0)),
    imputation_single(id = "Harry", values = c(1, 2)),
    imputation_single(id = "Phil", values = c(3, 4)),
    imputation_single(id = "Tom", values = c(7, 8, 9))
  ),
  imputation_df(
    imputation_single(id = "Ben", values = numeric(0)),
    imputation_single(id = "Ben", values = numeric(0)),
    imputation_single(id = "Phil", values = c(5, 6))
  )
)
```

STAN_BLOCKS	<i>List of Stan Blocks</i>
-------------	----------------------------

Description

A list with 1 element per standard Stan program blocks. This object is mostly used internally as a reference for what blocks are parsed from a covariance / prior Stan definition file.

Usage

```
STAN_BLOCKS
```

strategies	<i>Strategies</i>
------------	-------------------

Description

These functions are used to implement various reference based imputation strategies by combining a subjects own distribution with that of a reference distribution based upon which of their visits failed to meet the Missing-at-Random (MAR) assumption.

Usage

```
strategy_MAR(pars_group, pars_ref, index_mar)

strategy_JR(pars_group, pars_ref, index_mar)

strategy_CR(pars_group, pars_ref, index_mar)

strategy_CIR(pars_group, pars_ref, index_mar)

strategy_LMCF(pars_group, pars_ref, index_mar)
```

Arguments

<code>pars_group</code>	A list of parameters for the subject's group. See details.
<code>pars_ref</code>	A list of parameters for the subject's reference group. See details.
<code>index_mar</code>	A logical vector indicating which visits meet the MAR assumption for the subject. I.e. this identifies the observations after a non-MAR intercurrent event (ICE).

Details

`pars_group` and `pars_ref` both must be a list containing elements `mu` and `sigma`. `mu` must be a numeric vector and `sigma` must be a square matrix symmetric covariance matrix with dimensions equal to the length of `mu` and `index_mar`. e.g.

```
list(
  mu = c(1,2,3),
  sigma = matrix(c(4,3,2,3,5,4,2,4,6), nrow = 3, ncol = 3)
)
```

Users can define their own strategy functions and include them via the `strategies` argument to `impute()` using `getStrategies()`. That being said the following strategies are available "out the box":

- Missing at Random (MAR)
- Jump to Reference (JR)
- Copy Reference (CR)
- Copy Increments in Reference (CIR)
- Last Mean Carried Forward (LMCF)

Value

A list with elements `mu` (a numeric vector of means) and `sigma` (a covariance matrix) giving the parameters of the subject's imputation distribution, formed by combining their own group's parameters (`pars_group`) with the reference group's parameters (`pars_ref`) according to the chosen strategy.

Examples

```
# Parameters of the subject's own group and of the reference group
pars_group <- list(
  mu = c(1, 2, 3),
  sigma = as_vcov(c(1, 3, 2), c(0.4, 0.5, 0.45))
)
pars_ref <- list(
  mu = c(5, 6, 7),
  sigma = as_vcov(c(2, 1, 1), c(0.7, 0.8, 0.5))
)

# The first two visits meet the MAR assumption, the third does not
```

```

index_mar <- c(TRUE, TRUE, FALSE)

# Combine the parameters according to the different strategies
strategy_MAR(pars_group, pars_ref, index_mar)
strategy_JR(pars_group, pars_ref, index_mar)
strategy_CR(pars_group, pars_ref, index_mar)
strategy_CIR(pars_group, pars_ref, index_mar)
strategy_LMCF(pars_group, pars_ref, index_mar)

```

string_pad	<i>string_pad</i>
------------	-------------------

Description

Utility function used to replicate str_pad. Adds white space to either end of a string to get it to equal the desired length

Usage

```
string_pad(x, width)
```

Arguments

x	string
width	desired length

str_contains	<i>Does a string contain a substring</i>
--------------	--

Description

Returns a vector of TRUE/FALSE for each element of x if it contains any element in subs
i.e.

```

str_contains( c("ben", "tom", "harry"), c("e", "y"))
[1] TRUE FALSE TRUE

```

Usage

```
str_contains(x, subs)
```

Arguments

x	character vector
subs	a character vector of substrings to look for

`transpose_imputations` *Transpose imputations*

Description

Takes an `imputation_df` object and transposes it e.g.

```
list(
  list(id = "a", values = c(1,2,3)),
  list(id = "b", values = c(4,5,6))
)
```

Usage

```
transpose_imputations(imputations)
```

Arguments

`imputations` An `imputation_df` object created by `imputation_df()`

Details

becomes

```
list(
  ids = c("a", "b"),
  values = c(1,2,3,4,5,6)
)
```

`transpose_results` *Transpose results object*

Description

Transposes a Results object (as created by `analyse()`) in order to group the same estimates together into vectors.

Usage

```
transpose_results(results, components)
```

Arguments

`results` A list of results.
`components` a character vector of components to extract (i.e. "est", "se").

Details

Essentially this function takes an object of the format:

```
x <- list(
  list(
    "trt1" = list(
      est = 1,
      se  = 2
    ),
    "trt2" = list(
      est = 3,
      se  = 4
    )
  ),
  list(
    "trt1" = list(
      est = 5,
      se  = 6
    ),
    "trt2" = list(
      est = 7,
      se  = 8
    )
  )
)
```

and produces:

```
list(
  trt1 = list(
    est = c(1,5),
    se  = c(2,6)
  ),
  trt2 = list(
    est = c(3,7),
    se  = c(4,8)
  )
)
```

transpose_samples	<i>Transpose samples</i>
-------------------	--------------------------

Description

Transposes samples generated by `draws()` so that they are grouped by `subjid` instead of by sample number.

Usage

```
transpose_samples(samples)
```

Arguments

`samples` A list of samples generated by `draws()`.

<code>validate.analysis</code>	<i>Validate analysis objects</i>
--------------------------------	----------------------------------

Description

Validates the return object of the `analyse()` function.

Usage

```
## S3 method for class 'analysis'
validate(x, ...)
```

Arguments

`x` An analysis results object (of class "jackknife", "bootstrap", "rubin").
`...` Not used.

<code>validate.draws</code>	<i>Validate draws object</i>
-----------------------------	------------------------------

Description

Validate draws object

Usage

```
## S3 method for class 'draws'
validate(x, ...)
```

Arguments

`x` A draws object generated by `as_draws()`.
`...` Not used.

validate.is_mar	<i>Validate is_mar for a given subject</i>
-----------------	--

Description

Checks that the longitudinal data for a patient is divided in MAR followed by non-MAR data; a non-MAR observation followed by a MAR observation is not allowed.

Usage

```
## S3 method for class 'is_mar'  
validate(x, ...)
```

Arguments

x	Object of class is_mar. Logical vector indicating whether observations are MAR.
...	Not used.

Value

Will error if there is an issue otherwise will return TRUE.

validate.ivars	<i>Validate inputs for vars</i>
----------------	---------------------------------

Description

Checks that the required variable names are defined within vars and are of appropriate data types

Usage

```
## S3 method for class 'ivars'  
validate(x, ...)
```

Arguments

x	named list indicating the names of key variables in the source dataset
...	not used

validate.references	<i>Validate user supplied references</i>
---------------------	--

Description

Checks to ensure that the user specified references are expect values (i.e. those found within the source data).

Usage

```
## S3 method for class 'references'  
validate(x, control, ...)
```

Arguments

x	named character vector.
control	factor variable (should be the group variable from the source dataset).
...	Not used.

Value

Will error if there is an issue otherwise will return TRUE.

validate.sample_list	<i>Validate sample_list object</i>
----------------------	------------------------------------

Description

Validate sample_list object

Usage

```
## S3 method for class 'sample_list'  
validate(x, ...)
```

Arguments

x	A sample_list object generated by sample_list() .
...	Not used.

```
validate.sample_single
```

Validate sample_single object

Description

Validate sample_single object

Usage

```
## S3 method for class 'sample_single'  
validate(x, ...)
```

Arguments

x	A sample_single object generated by sample_single() .
...	Not used.

```
validate.simul_pars
```

Validate a simul_pars object

Description

Validate a simul_pars object

Usage

```
## S3 method for class 'simul_pars'  
validate(x, ...)
```

Arguments

x	An simul_pars object as generated by set_simul_pars() .
...	Not used.

validate.stan_data	<i>Validate a stan_data object</i>
--------------------	------------------------------------

Description

Validate a stan_data object

Usage

```
## S3 method for class 'stan_data'  
validate(x, ...)
```

Arguments

x	A stan_data object.
...	Not used.

validate_datalong	<i>Validate a longdata object</i>
-------------------	-----------------------------------

Description

Validate a longdata object

Usage

```
validate_datalong(data, vars)  
  
validate_datalong_varExists(data, vars)  
  
validate_datalong_types(data, vars)  
  
validate_datalong_notMissing(data, vars)  
  
validate_datalong_complete(data, vars)  
  
validate_datalong_unifromStrata(data, vars)  
  
validate_dataice(data, data_ice, vars, update = FALSE)
```

Arguments

data	a data.frame containing the longitudinal outcome data + covariates for multiple subjects
vars	a vars object as created by <code>set_vars()</code>
data_ice	a data.frame containing the subjects ICE data. See <code>draws()</code> for details.
update	logical, indicates if the ICE data is being set for the first time or if an update is being applied

Details

These functions are used to validate various different parts of the longdata object to be used in `draws()`, `impute()`, `analyse()` and `pool()`. In particular:

- `validate_datalong_varExists` - Checks that each variable listed in vars actually exists in the data
- `validate_datalong_types` - Checks that the types of each key variable is as expected i.e. that visit is a factor variable
- `validate_datalong_notMissing` - Checks that none of the key variables (except the outcome variable) contain any missing values
- `validate_datalong_complete` - Checks that data is complete i.e. there is 1 row for each subject * visit combination. e.g. that `nrow(data) == length(unique(subjects)) * length(unique(visits))`
- `validate_datalong_unifromStrata` - Checks to make sure that any variables listed as stratification variables do not vary over time. e.g. that subjects don't switch between stratification groups.

validate_strategies	<i>Validate user specified strategies</i>
---------------------	---

Description

Compares the user provided strategies to those that are required (the reference). Will throw an error if not all values of reference have been defined.

Usage

```
validate_strategies(strategies, reference)
```

Arguments

strategies	named list of strategies.
reference	list or character vector of strategies that need to be defined.

Value

Will throw an error if there is an issue otherwise will return TRUE.

Index

* datasets

antidepressant_data, 15

adjust_trajectories, 5

adjust_trajectories_single, 6

adjust_trajectories_single(), 5

analyse, 7

analyse(), 8, 9, 11, 13, 17, 30, 31, 33, 44, 45, 64, 79, 84, 100, 112, 114, 119

ancova, 11

ancova(), 8, 10, 13, 15, 99, 100

ancova_single, 13

antidepressant_data, 15

apply_delta, 16

as.data.frame.mcse (pool), 78

as.data.frame.pool (pool), 78

as_analysis, 17

as_ascii_table, 18

as_cropped_char, 18

as_dataframe, 19

as_draws, 19

as_draws(), 114

as_imputation, 20

as_indices, 21

as_mmr_df, 21

as_mmr_df(), 22

as_mmr_formula, 22

as_model_df, 22

as_simple_formula, 23

as_stan_array, 23

as_strata, 24

as_vcov (simulate_test_data), 105

assert_variables_exist, 17

base::expand.grid(), 40

char2fct, 24

check_ESS, 25

check_ESS(), 26

check_hmc_diagn, 25

check_hmc_diagn(), 26

check_mcmc, 26

check_mcmc(), 47

compute_sigma, 27

control, 27

control_bayes (control), 27

control_bayes(), 75

convert_to_imputation_list_df, 29

d_lagscale, 39

delta_template, 30

delta_template(), 8, 10, 39, 45, 52

draws, 33, 62, 66

draws(), 8, 9, 17, 19, 28, 31, 32, 35, 41, 62, 63, 66, 76, 79, 84, 99, 100, 113, 114, 119

eval_mmr, 39

expand, 40

expand_locf (expand), 40

expand_locf(), 35, 37

extract_covariates, 42

extract_data_mnar_as_na, 43

extract_draws, 43

extract_draws(), 47

extract_imputed_df, 44

extract_imputed_dfs, 45

extract_imputed_dfs(), 10, 64

extract_params, 46

fill_locf (expand), 40

fit_mcmc, 46

fit_mmr, 47

format_method_descriptions, 48

generate_data_single, 49

get_bootstrap_stack, 51

get_conditional_parameters, 51

get_delta_template, 52

get_draws_mle, 52

get_ESS, 54
 get_estsbmlmi, 54
 get_example_data, 55
 get_example_data(), 55
 get_jackknife_stack, 56
 get_mmrmsample, 56
 get_pattern_groups, 57
 get_pattern_groups_unique, 58
 get_pool_components, 58
 get_visit_distribution_parameters, 59
 getStrategies, 50
 getStrategies(), 5, 6, 49, 63, 65, 66, 101, 110

 I(), 100
 ife, 59
 imputation_df, 60, 60
 imputation_df(), 20, 29, 44, 108, 112
 imputation_list_df, 60
 imputation_list_df(), 29
 imputation_list_single, 61
 imputation_list_single(), 29
 imputation_single, 61
 imputation_single(), 29, 61, 108
 impute, 62
 impute(), 7, 20, 31, 36, 45, 52, 53, 64, 66, 85, 110, 119
 impute_data_individual, 64
 impute_data_individual(), 29
 impute_internal, 66
 impute_outcome, 67
 invert, 67
 invert_indexes, 68
 is_absent, 68
 is_char_fact, 69
 is_char_one, 69
 is_in_rbmi_development, 69
 is_num_char_fact, 70

 list(), 20
 locf, 70
 longDataConstructor(), 20, 44, 51, 56, 65
 ls_design, 72
 ls_design_counterfactual(ls_design), 72
 ls_design_equal(ls_design), 72
 ls_design_proportional(ls_design), 72
 lsmeans, 71

 make_rbmi_cluster, 73

 make_rbmi_cluster(), 7, 9, 34
 mcse(pool), 78
 method, 74
 method_approxbayes(method), 74
 method_approxbayes(), 8, 19, 20, 34, 37, 52, 53, 57
 method_bayes(method), 74
 method_bayes(), 8, 19, 20, 28, 34, 37, 46, 53
 method_bmlmi(method), 74
 method_bmlmi(), 19, 29, 34, 37
 method_condmean(method), 74
 method_condmean(), 19, 20, 34, 37, 52, 53, 57
 mmrm::cov_types(), 76
 mmrm::mmrm(), 39, 46, 47, 88

 par_lapply, 78
 parallel::clusterApplyLB, 78
 parallel::makeCluster(), 78
 parametric_ci, 77
 pool, 78
 pool(), 9, 79, 119
 pool_bootstrap_normal, 81
 pool_bootstrap_percentile, 82
 prepare_stan_data, 83
 prepare_stan_data(), 47
 print.analysis, 84
 print.draws, 84
 print.imputation, 85
 print.mcse(pool), 78
 print.pool(pool), 78
 progressLogger, 85
 pval_percentile, 87

 QR_decomp, 87

 random_effects_expr, 88
 rbmi-settings, 88
 record, 89
 record(), 40
 recursive_reduce, 90
 remove_if_all_missing, 91
 rstan::sampling(), 28
 rubin_df, 91
 rubin_df(), 93
 rubin_rules, 92

 sample_ids, 93
 sample_list, 93
 sample_list(), 116

sample_mvnorm, 94
 sample_single, 94
 sample_single(), 19, 93, 117
 scalerConstructor, 95
 set_options(rbmi-settings), 88
 set_simul_pars, 97
 set_simul_pars(), 49, 56, 101, 117
 set_vars, 99
 set_vars(), 8, 11–13, 31, 34, 36, 37, 119
 simulate_data, 100
 simulate_data(), 49, 55, 56, 97, 98
 simulate_dropout, 103
 simulate_ice, 104
 simulate_test_data, 105
 sort_by, 106
 split_dim, 107
 split_imputations, 108
 Stack(), 51, 56
 STAN_BLOCKS, 109
 stats::lm(), 13
 stats::model.matrix(), 22, 48
 stats::quantile(), 82
 str_contains, 111
 strategies, 109
 strategy_CIR(strategies), 109
 strategy_CR(strategies), 109
 strategy_JR(strategies), 109
 strategy_LMCF(strategies), 109
 strategy_MAR(strategies), 109
 string_pad, 111

 transpose_imputations, 112
 transpose_results, 112
 transpose_samples, 113

 validate.analysis, 114
 validate.draws, 114
 validate.is_mar, 115
 validate.ivars, 115
 validate.references, 116
 validate.sample_list, 116
 validate.sample_single, 117
 validate.simul_pars, 117
 validate.stan_data, 118
 validate_dataice(validate_datalong),
 118
 validate_datalong, 118
 validate_datalong_complete
 (validate_datalong), 118

 validate_datalong_notMissing
 (validate_datalong), 118
 validate_datalong_types
 (validate_datalong), 118
 validate_datalong_unifromStrata
 (validate_datalong), 118
 validate_datalong_varExists
 (validate_datalong), 118
 validate_strategies, 119