Package: bkmodel (via r-universe)

October 20, 2026

Title Statistical Modeling Utility Functions

Version 0.0.1

Description A collection of statistical modeling utility functions.

URL https://brettklamer.com/work/bkmodel,
https://bitbucket.org/bklamer/bkmodel

License MIT + file LICENSE

LazyData TRUE

Depends R (>= 4.5.0)

Imports Rdpack, bkbase, stats, utils

Suggests tinytest, rmarkdown, survival

RdMacros Rdpack

Language en-US

Roxygen list(markdown = TRUE)

Encoding UTF-8

Config/roxygen2/version 8.1.1

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

Date/Publication 2026-10-20 15:11:14 UTC

RemoteUrl https://bitbucket.org/bklamer/bkmodel

RemoteRef HEAD

RemoteSha 13b4222bcb81298f3bbaf0d7e85b28bca6469785

Contents

design_apply 2
design_bootstrap 6
design_cv 11
design_fold 15
design_inner_cv 20
design_learn 23

design_spec 26
design_splits 30
get_dep_data 34
get_ind_data 37
optim2 40
ordinal_counts 44
ordinal_matrix_to_factor 47
parse_formula 49
print.design_plan 50
tall_to_wide 52

Index **56**

`design_apply` *Apply a learned design schema to new data*

Description

`design_apply()` encodes new data with a frozen `design_learn()` schema. The encoded predictor matrix has the same columns, categorical encoding, factor levels, and column order as the learn-side design matrix. `design_apply()` never refits the schema and never learns new factor levels from `data`.

The result holds a model-ready `design_inputs` payload and a `design_report` that attributes missing, novel, and non-finite encoded values by row.

Usage

```
design_apply(
  schema,
  data,
  row_index = NULL,
  response = c("auto", "require", "ignore"),
  weights = c("auto", "require", "ignore"),
  na_action = NULL,
  novel_levels = NULL
)
```

Arguments

`schema`	(`design_schema`) A schema returned by `design_learn()`.
`data`	(`data.frame`) New data to encode. Objects that inherit from `data.frame`, including tibbles, are accepted.
`row_index`	(`integer(nrow(data))` or `NULL`; default `NULL`) Row identities to carry on the inputs. `NULL` means `seq_len(nrow(data))`. Values must be non-missing, unique whole numbers in `[1, .Machine$integer.max]`.

`response` (`character(1)`; default `"auto"`)
One of `"auto"`, `"require"`, or `"ignore"`. Response evaluation mode.

`weights` (`character(1)`; default `"auto"`)
One of `"auto"`, `"require"`, or `"ignore"`. Weight evaluation mode.

`na_action` (`character(1)` or `NULL`; default `NULL`)
One of `"na.pass"`, `"na.omit"`, `"na.fail"`, or `"na.exclude"`. Missing-value policy for row retention. `NULL` means the schema's stored policy.

`novel_levels` (`character(1)` or `NULL`; default `NULL`)
One of `"na"` or `"fail"`. Handling for categorical values not seen at learn time. `NULL` means the schema's stored policy.

Details

Validation phases:

Validation runs in three phases. Phase one checks column presence and reports every absent column in one error, grouped by role. Phase two checks class and level conformance. Evaluated predictor columns must match the schema's numeric, logical, or categorical class, and categorical values are constrained to the frozen levels. Phase three builds the encoded matrix.

Missing, novel, and non-finite encoded predictor values are reported in the `design_report`, not raised as errors. The two exceptions are the caller's own declared policies, `na_action = "na.fail"` and `novel_levels = "fail"`.

Evaluated weights and offsets are outside this guarantee. They are validated on retained rows, as described in "Retained rows" below.

Functions called by the schema's formula and role expressions must be visible on the standard search path, because the schema does not carry their defining environment. See the pointwise expressions section of `design_spec()`.

Predictor schema:

All predictor source columns required by the schema must be present in `data`. Character and factor predictors are constrained to the levels frozen at learn time. `novel_levels = "na"` converts unseen categorical values to `NA` before row retention and encoding, and the report catalogs them. `novel_levels = "fail"` errors and reports the affected variables and values. `novel_levels = "fail"` checks every row of `data`, before `na_action` removes any row.

The returned `x` matrix has the same column names and order as the learn-side matrix. A learned column whose level does not occur in `data` holds zeros.

Retained rows:

`na_action` controls row retention for evaluated predictors, responses, weights, and offsets. The default `NULL` uses the schema's stored policy. `"na.pass"` keeps rows with missing values, `"na.omit"` and `"na.exclude"` keep only complete evaluated rows, and `"na.fail"` errors if any evaluated value is missing. Only evaluated roles participate in retention. With `response = "ignore"`, missing response values do not drop rows.

Weights and offsets are stricter. On retained rows, evaluated weights must be non-missing, finite, and non-negative, and evaluated offsets must be non-missing and finite. `Inf`, `-Inf`, and (for weights) negative values error under every `na_action`, because row dropping keys on `is.na()` and none of those values are missing by that test. `NA` and `NaN` are missing by that test, so they error only when `na_action` retains their row. `"na.omit"` drops them first. Set `weights = "ignore"` to

skip weight evaluation entirely. Offsets have no ignore mode. Case ids are metadata and do not participate in row retention.

Response modes:

`response = "auto"` evaluates `y` when all response source columns are present in `data`, returns `y = NULL` when none are present, and errors on partial presence. `response = "require"` errors when any response source column is absent. `response = "ignore"` never evaluates the response. For one-sided schemas, `y = NULL` under every mode.

When the schema froze response levels, `design_apply()` recodes a factor or character response to those levels. A response value outside the frozen levels stops with an error. A response of another type also stops with an error.

Use `response = "ignore"` for pure prediction data without observed responses. Use `response = "require"` for assessment data where the response must be present.

Weight modes:

If the schema has no weights role, all modes return `weights = NULL`. Otherwise `weights = "auto"`, `"require"`, and `"ignore"` follow the same presence semantics as the response modes.

Offsets:

When the schema has an offset, it is always evaluated. There is no offset mode argument. Explicit role offsets and right-hand-side `offset()` terms are evaluated on `data` and summed into one numeric vector, as at learn time. Absent offset source columns join the column-presence error.

Case ids:

When all case_id source columns are present, `case_id` is returned aligned to the retained rows. When none are present, `case_id = NULL`. Partial presence errors.

Report:

The `design_report` has the same shape as the learn-side report from `design_learn()`. The `missing` and `novel` masks are aligned to `inputs$row_index`. Under `na_action = "na.pass"` they describe every input row, so `missing | novel` marks exactly the rows of `x` that contain `NA`. Under a row-dropping `na_action` the flagged rows are omitted, so both masks are all `FALSE`, while `novel_levels` still catalogs the novel values found in the input. The `nonfinite` mask and `nonfinite_cols` catalog attribute every `Inf`, `-Inf`, and `NaN` in the encoded matrix by row identity.

Value

A plain list with elements:

`inputs` A `design_inputs` with `x`, `y`, `weights`, `offset`, `case_id`, and `row_index`, all aligned to the retained rows.

`report` A `design_report` with `missing`, `novel`, and `nonfinite` logical masks aligned to `inputs$row_index`, plus the `novel_levels` and `nonfinite_cols` catalogs.

See Also

`design_learn()`, `design_spec()`, `design_fold()`

Examples

```
#-------------------------------------------------------------------------------
# Encode prediction data with a learned schema
#-------------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  patient_id = sprintf("P%02d", 1:12),
  outcome = rnorm(12),
  age = round(rnorm(12, mean = 60, sd = 8)),
  treatment = factor(rep(c("control", "active"), 6)),
  stage = factor(rep(c("I", "II", "III"), each = 4)),
  exposure = runif(12, min = 0.5, max = 2),
  inverse_prob_weight = runif(12, min = 0.8, max = 1.2)
)

spec <- design_spec(
  outcome ~ age + treatment + stage,
  weights = ~inverse_prob_weight,
  offset = ~ log(exposure),
  case_id = ~patient_id,
  encoding = "dummy",
  intercept = TRUE,
  na_action = "na.omit"
)

lr <- design_learn(spec, analysis)
schema <- lr$specs[[1L]]$schema
inputs <- lr$specs[[1L]]$inputs

model <- stats::lm.wfit(
  x = inputs$x,
  y = inputs$y,
  w = inputs$weights,
  offset = inputs$offset
)

prediction_data <- data.frame(
  patient_id = sprintf("V%02d", 1:3),
  age = c(55, 67, 61),
  treatment = factor(c("active", "control", "active")),
  stage = factor(c("II", "I", "III")),
  exposure = c(1.1, 0.9, 1.5)
)

ap <- design_apply(
  schema,
  prediction_data,
  response = "ignore",
  weights = "ignore"
)
```

```
linear_predictor <- drop(ap$inputs$x %*% model$coefficients)
linear_predictor <- linear_predictor + ap$inputs$offset
linear_predictor

#------------------------------------------------------------------------------
# Encode assessment data and read the report
#------------------------------------------------------------------------------
assessment_data <- data.frame(
  patient_id = sprintf("A%02d", 1:3),
  outcome = c(0.1, -0.4, 0.7),
  age = c(58, 63, 52),
  treatment = factor(c("control", "active", "control")),
  stage = factor(c("I", "III", "II")),
  exposure = c(1.0, 1.3, 0.8),
  inverse_prob_weight = c(1.0, 0.9, 1.1)
)

ap2 <- design_apply(
  schema,
  assessment_data,
  response = "require",
  weights = "require"
)

ap2$inputs$y
ap2$inputs$weights
ap2$report$missing
```

`design_bootstrap` *Build a bootstrap plan*

Description

`design_bootstrap()` plans bootstrap resampling for one or more `design_spec()` objects. Draws are taken over the shared row universe, the rows all specs retain, so every draw respects each spec's `na_action`. The returned `design_plan` holds the specs, the data, the universe, and the draws. It fits no model and builds no design matrix. Use `design_fold()` to materialize one draw at a time.

Optimism-corrected tuning is `design_bootstrap()` with an `inner` descriptor. There is no separate constructor.

Usage

```
design_bootstrap(
  specs,
  data,
  n_boot = 1000L,
  strata = NULL,
```

```
  group = NULL,
  inner = NULL,
  seed = NULL
)
```

Arguments

specs	(design_spec or list of design_spec) One or more design specifications from design_spec(). A bare spec is treated as a length-one list. All specs must share the same response expression, and factor responses must share declared levels.
data	(data.frame) Data to resample. Objects that inherit from data.frame, including tibbles, are accepted.
n_boot	(integer(1); default 1000L; range [1, .Machine$integer.max]) Number of bootstrap draws. Each draw is one repetition. Whole-number numeric values such as 500 and 500.0 are accepted and stored as integers.
strata	(formula or NULL; default NULL) Stratification specification. Use NULL for unstratified draws, ~ 1 to stratify by the raw response, or a one-sided formula such as ~ treatment for observed columns. Evaluated on universe rows only. NA values error.
group	(formula or NULL; default NULL) Grouping specification. Use NULL for row-level draws, or a one-sided formula such as ~ patient_id to draw whole groups together. Evaluated on universe rows only. NA values error.
inner	(design_inner_cv or NULL; default NULL) Inner cross-validation descriptor from design_inner_cv(). When present, every materialized fold carries an inner plan over its draw positions.
seed	(integer(1) or NULL; default NULL) Plan seed. NULL draws one integer from the ambient stream. An explicit value leaves the caller's stream untouched.

Details

Row universe:

specs is a design_spec or a list of them, as in design_learn(). The row universe is the sorted intersection of the per-spec retained rows, computed by the same internal helper as design_learn() and design_cv(). When the intersection drops rows for any spec, construction warns with condition class bkmodel_warning_universe. When the intersection is empty, construction errors.

Construction runs retention first, then the universe, then strata and group resolution, then draw assignment. strata and group are evaluated on universe rows only, and an NA value in either is a construction-time error. No draw contains a row any spec's na_action would drop.

Draws:

Each draw is one repetition, named in the "rep17" style (indices are zero-padded when the count needs it). Each split stores analysis as integer positions into plan$universe, in draw order with duplicates kept as draw multiplicities, and assessment as the full universe. At materialization,

design_fold() scores the full universe as the assessment part (orig scoring) and derives oob as the rows absent from the draw (.632 blends). plan$universe[split$analysis] recovers data row identities.

Stratification and grouping:

With strata, sampling runs within stratum and preserves stratum sizes, so draw order is stratum-major. Every bootstrap stratum must contain at least two sampling units, because singleton strata are deterministic under resampling with replacement. Construction warns when ~ 1 stratifies an integer or character response that looks high-cardinality, because many small strata leave little room for within-stratum draws to vary. The advisory carries condition class bkmodel_warning_strata. With group, whole groups are drawn together and all copies of a drawn group are contiguous in the draw.

When a spec's formula uses ., a group or strata variable may not enter the predictors through the expansion alone. The check reports an operational variable that . adds as a predictor with no sign from the user, for example an identifier or a stratifier derived from the outcome. Give a group variable the case_id or exclude role so . skips it. To use a strata variable as a predictor, name it after ., as in y ~ . + x, and otherwise give it the exclude role. A response variable is always allowed, because . never adds it.

Inner cross-validation:

Pass inner = design_inner_cv(n_folds, n_reps) to make every materialized fold carry an inner plan over its draw. In a bootstrap plan, the draw is the training sample and each drawn copy is a distinct observation. Deduplicating before inner cross-validation would shrink the effective sample size and reweight duplicated rows, so inner selection would tune against a different data distribution than the fit it serves. Inner cross-validation therefore partitions draw positions. By default the inner plan inherits group and strata. Under group, the rows of one drawn copy of a group stay on one inner side. Two copies of one group can land on different inner sides, as two copies of one row can. Each draw position keeps the stratum of the row it copies, so the inner folds can stratify on it. Set strata or group on the descriptor to override the inherited value.

One caveat is inherent to this standard method. A data row or group drawn more than once can land on both inner sides. The inner score is a selection input, never a reported estimate, so this straddle produces no warning.

Seeds and reproducibility:

seed = NULL draws one integer from the ambient stream, so construction consumes exactly one draw and the set.seed()-before-the-call idiom reproduces exactly. An explicit seed leaves the caller's stream untouched. Draw assignment and every fit seed derive deterministically from the stored value, and all derivation is RNG-isolated. Growing n_boot with the same seed appends draws and seed blocks and leaves existing draws and seeds bit-identical. Only the values are stable, not the names: plan$splits names zero-pad to the current n_boot, so growth across a padding-width boundary (9 to 10, 99 to 100) renames an existing draw without changing it. Index plan$splits by position across incremental growth. All seeds a plan hands out are pairwise distinct, including every seed of every inner plan. See design_cv() for the shared seed model, its granularity, and its documented non-goals.

Construction warnings:

Documented warnings carry condition classes, so a consumer can muffle a known duplicate selectively with withCallingHandlers().

- bkmodel_warning_universe: the universe intersection dropped rows for any spec.
- bkmodel_warning_case_id: a spec carries a case_id role, duplicated ids exist among universe rows, and no group is given.
- bkmodel_warning_strata: with inner, a draw holds fewer than two sampling units of an inner stratum. An inner analysis set then has none of it. It also signals that strata = ~ 1 stratifies an integer or character response that looks high-cardinality.

Optimism workflow:

Optimism correction needs nothing beyond bootstrap folds. The apparent fit is the consumer's own full-data learn and tune, which precedes the plan and is universe-consistent via design_learn():

```
lr <- design_learn(spec, dat)              # apparent: full-data learn
# ... fit and tune on lr$specs[[1L]]$inputs ...
bplan <- design_bootstrap(spec, dat, n_boot = 500L,
                          inner = design_inner_cv(5L, 3L), seed = 23L)
# per draw: boot term on analysis, orig term on assessment,
# both under their scoreable masks; optimism = mean(boot - orig)
```

Value

A list of class design_plan with elements:

specs List of design_spec, length one or more.

data The data as given.

universe Sorted integer row identities this plan resamples over.

retained List with one integer vector per spec, the rows that spec's na_action retained before intersection.

response_levels Character vector of the response levels that design_learn() freezes on data, or NULL when the response is not a factor. Every fold uses these levels for the response.

splits Named list of splits, each list(analysis, assessment) of integer positions into universe. analysis keeps draw order and duplicates. assessment is the full universe.

method "bootstrap".

params List of the sampling parameters as given: n_boot, strata, group.

inner The design_inner_cv descriptor, or NULL.

seed Integer scalar, the resolved plan seed.

seeds The seed allocation table, one block per draw holding the rep sub-seed, the fold seed, per-spec entry seeds, and inner plan blocks.

See Also

design_spec(), design_learn(), design_inner_cv(), design_cv(), design_fold(), base::set.seed()

Examples

```
#--------------------------------------------------------------------------
# Build a stratified bootstrap plan
#--------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  patient_id = sprintf("P%02d", 1:24),
  age = round(rnorm(24, mean = 60, sd = 8)),
  treatment = factor(rep(c("control", "active"), 12)),
  outcome = rnorm(24)
)

spec <- design_spec(
  outcome ~ age + treatment,
  encoding = "dummy",
  intercept = TRUE
)

plan <- design_bootstrap(
  spec,
  analysis,
  n_boot = 20L,
  strata = ~treatment,
  seed = 42L
)

plan$method
names(plan$splits)

# A draw keeps duplicates; the assessment side is the full universe.
split <- plan$splits[["rep01"]]
table(table(split$analysis))
identical(split$assessment, seq_along(plan$universe))

#--------------------------------------------------------------------------
# Tuned bootstrap: inner cross-validation inside each draw
#--------------------------------------------------------------------------
tuned <- design_bootstrap(
  spec,
  analysis,
  n_boot = 5L,
  inner = design_inner_cv(n_folds = 3L),
  seed = 42L
)

tuned$inner
names(tuned$seeds[[1L]])
```

design_cv *Build a cross-validation plan*

Description

design_cv() plans K-fold cross-validation for one or more design_spec() objects. Splits are drawn over the shared row universe, the rows all specs retain, so every split respects each spec's na_action. The returned design_plan holds the specs, the data, the universe, and the splits. It fits no model and builds no design matrix. Use design_fold() to materialize one split at a time.

Nested cross-validation is design_cv() with an inner descriptor. There is no separate nested constructor.

Usage

```
design_cv(
  specs,
  data,
  n_folds = 5L,
  n_reps = 1L,
  strata = NULL,
  group = NULL,
  inner = NULL,
  seed = NULL
)
```

Arguments

specs	(design_spec or list of design_spec) One or more design specifications from design_spec(). A bare spec is treated as a length-one list. All specs must share the same response expression, and factor responses must share declared levels.
data	(data.frame) Data to resample. Objects that inherit from data.frame, including tibbles, are accepted.
n_folds	(integer(1); default 5L; range [2, sampling units]) Folds per repetition. The number of sampling units is the universe size, or the number of groups when group is given. Whole-number numeric values such as 5 and 5.0 are accepted and stored as integers.
n_reps	(integer(1); default 1L; range [1, .Machine$integer.max]) Independent K-fold partitions. Use values greater than one for repeated cross-validation.
strata	(formula or NULL; default NULL) Stratification specification. Use NULL for no stratification, ~ 1 to stratify by the raw response, or a one-sided formula such as ~ treatment for observed columns. Evaluated on universe rows only. NA values error.

`group` (formula or `NULL`; default `NULL`)
Grouping specification. Use `NULL` for row-level resampling, or a one-sided formula such as `~ patient_id` to keep rows with the same group value together. Evaluated on universe rows only. NA values error.

`inner` (design_inner_cv or `NULL`; default `NULL`)
Inner cross-validation descriptor from `design_inner_cv()`. When present, every materialized fold carries an inner plan over its analysis rows, which makes the plan nested cross-validation.

`seed` (`integer(1)` or `NULL`; default `NULL`)
Plan seed. `NULL` draws one integer from the ambient stream. An explicit value leaves the caller's stream untouched.

Details

Row universe:

`specs` is a `design_spec` or a list of them, as in `design_learn()`. Each spec's `na_action` resolves its retained rows, and the row universe is the sorted intersection of the per-spec retained rows. `design_cv()` and `design_learn()` compute the universe through the same internal helper, so both return identical universes from the same specs and data. When the intersection drops rows for any spec, construction warns with condition class `bkmodel_warning_universe`. When the intersection is empty, construction errors.

Ordering invariant:

Construction runs retention first, then the universe, then strata and group resolution, then fold assignment. `strata` and `group` are evaluated on universe rows only, and an NA value in either is a construction-time error, never a silent drop. No split contains a row any spec's `na_action` would drop.

Splits:

`n_folds` controls the number of folds per partition and `n_reps` the number of independent partitions, so the plan holds `n_folds * n_reps` splits named in the `"rep1.fold3"` style (indices are zero-padded when the count needs it). Within each repetition, the assessment sets partition the universe and every assessment set is disjoint from its analysis set.

Each split stores `analysis` and `assessment` as integer positions into `plan$universe`. `plan$universe[split$analysis]` recovers data row identities.

Stratification:

Use `strata = NULL` for unstratified folds, a one-sided formula such as `~ treatment` to balance folds by observed columns, or `~ 1` to stratify by the raw response. With `na_action = "na.pass"`, `strata = ~ 1` errors when the response contains NA on universe rows. Either clean the response or switch the spec to `"na.omit"`. For any strata source, construction warns when a stratum has only one sampling unit. The analysis set of the fold that holds this unit then has none of the stratum. A stratum with fewer units than folds raises no warning. Its units go to distinct assessment sets, and every analysis set keeps it. It warns separately when `~ 1` stratifies an integer or character response that looks high-cardinality. These advisories, including the inner-CV feasibility check, carry condition class `bkmodel_warning_strata`.

Grouping:

Use `group = ~ patient_id` style formulas to keep rows with the same group value together on one side of every split. With `group`, the sampling unit is the group, so `n_folds` may not exceed the number of groups.

When a spec's formula uses `.`, a `group` or `strata` variable may not enter the predictors through the expansion alone. The check reports an operational variable that `.` adds as a predictor with no sign from the user, for example an identifier or a stratifier derived from the outcome. Give a `group` variable the `case_id` or `exclude` role so `.` skips it. To use a `strata` variable as a predictor, name it after `.`, as in `y ~ . + x`, and otherwise give it the `exclude` role. A response variable is always allowed, because `.` never adds it.

Nested cross-validation:

Pass `inner = design_inner_cv(n_folds, n_reps)` to make every materialized fold carry an inner plan over its analysis rows. The inner plan is a complete `design_plan`, materialized with the same `design_fold()` verb, and never nested further. By default the inner plan inherits `strata` and `group`, re-resolved on the fold's analysis rows. Set either on the descriptor to override the inherited value. Inner splits partition only the outer fold's analysis rows, so tuning never sees held-out rows (Varma and Simon 2006; Cawley and Talbot 2010).

Seeds and reproducibility:

`seed = NULL` draws one integer from the ambient stream, so construction consumes exactly one draw and the `set.seed()`-before-the-call idiom reproduces exactly. An explicit `seed` leaves the caller's stream untouched. Split assignment and every fit seed derive deterministically from the stored value, and all derivation is RNG-isolated.

Growing `n_reps` with the same seed appends splits and seed blocks and leaves existing splits and seeds bit-identical, so Monte Carlo runs extend without refitting. Only the values are stable, not the names: `plan$splits` names zero-pad to the current `n_reps`, so growth across a padding-width boundary (9 to 10, 99 to 100) renames an existing split without changing it. Index `plan$splits` by position across incremental growth. All seeds a plan hands out are pairwise distinct, including every seed of every inner plan. Seeds are plain integers consumed via `base::set.seed()` immediately before a fit. Fold-level and per-(fold, spec) seeds make each fit reproducible alone and deterministic under parallel execution, although Mersenne-Twister streams seeded with different integers carry no formal independence guarantee. No stability is promised under changes to `n_folds`, `strata`, `group`, or the spec list. bkmodel derives seeds and assigns splits with a fixed generator, Mersenne-Twister with inversion and rejection sampling. Because the generator is fixed, a plan and its materialized folds do not change with the session `RNGkind()`. The exception is `seed = NULL`, which draws the plan seed from the session stream under the session `RNGkind()`.

Construction warnings:

Documented warnings carry condition classes, so a consumer can muffle a known duplicate selectively with `withCallingHandlers()` instead of blanket `suppressWarnings()`.

- `bkmodel_warning_universe`: the universe intersection dropped rows for any spec.
- `bkmodel_warning_case_id`: a spec carries a case_id role, duplicated ids exist among universe rows, and no `group` is given.
- `bkmodel_warning_strata`: a stratum has fewer than two sampling units in the data or in an outer analysis set, and an outer or inner analysis set then has none of it. It also signals that `strata = ~ 1` stratifies an integer or character response that looks high-cardinality.

Value

A list of class `design_plan` with elements:

`specs` List of `design_spec`, length one or more.

`data` The data as given.

`universe` Sorted integer row identities this plan resamples over.

`retained` List with one integer vector per spec, the rows that spec's `na_action` retained before intersection.

`response_levels` Character vector of the response levels that `design_learn()` freezes on `data`, or `NULL` when the response is not a factor. Every fold uses these levels for the response.

`splits` Named list of splits, each `list(analysis, assessment)` of integer positions into `universe`.

`method` `"cv"`.

`params` List of the sampling parameters as given: `n_folds`, `n_reps`, `strata`, `group`.

`inner` The `design_inner_cv` descriptor, or `NULL`.

`seed` Integer scalar, the resolved plan seed.

`seeds` The seed allocation table, one block per repetition holding the rep sub-seed, fold seeds, per-(fold, spec) entry seeds, and inner plan blocks.

References

Cawley GC, Talbot NLC (2010). "On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation." *Journal of Machine Learning Research*, **11**(70), 2079–2107. http://jmlr.org/papers/v11/cawley10a.html.

Varma S, Simon R (2006). "Bias in error estimation when using cross-validation for model selection." *BMC Bioinformatics*, **7**(1), 91. ISSN 1471-2105. doi:10.1186/14712105791.

See Also

`design_spec()`, `design_learn()`, `design_inner_cv()`, `design_bootstrap()`, `design_fold()`, `base::set.seed()`

Examples

```
#-------------------------------------------------------------------------
# Build a stratified, repeated cross-validation plan
#-------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  patient_id = sprintf("P%02d", 1:24),
  age = round(rnorm(24, mean = 60, sd = 8)),
  treatment = factor(rep(c("control", "active"), 12)),
  outcome = rnorm(24)
)
```

```
spec <- design_spec(
  outcome ~ age + treatment,
  encoding = "dummy",
  intercept = TRUE
)

plan <- design_cv(
  spec,
  analysis,
  n_folds = 4L,
  n_reps = 2L,
  strata = ~treatment,
  seed = 42L
)

plan$method
names(plan$splits)

# Splits store positions into the universe; identities are recovered as:
split <- plan$splits[["rep1.fold3"]]
plan$universe[split$analysis]

# The same seed reproduces the plan exactly.
plan2 <- design_cv(
  spec,
  analysis,
  n_folds = 4L,
  n_reps = 2L,
  strata = ~treatment,
  seed = 42L
)
identical(plan$splits, plan2$splits)

#-------------------------------------------------------------------------
# Nested cross-validation is design_cv() with inner
#-------------------------------------------------------------------------
nested <- design_cv(
  spec,
  analysis,
  n_folds = 4L,
  inner = design_inner_cv(n_folds = 3L, n_reps = 2L),
  seed = 42L
)

nested$inner
names(nested$seeds[[1L]])
```

`design_fold` *Materialize one fold of a resampling plan*

Description

design_fold() materializes one split of a design_plan from design_cv(), design_bootstrap(), or design_splits(). It learns every spec's schema on the split's analysis rows, encodes the held-out parts with those schemas, and flags the rows that every spec can score. It does not fit a statistical model.

Folds are lazy. A plan stores index-level splits only, and design_fold() is the one materialization path, one fold at a time on demand.

Usage

```
design_fold(plan, i, parts = NULL)
```

Arguments

plan	(design_plan) A resampling plan from design_cv(), design_bootstrap(), or design_splits(), including a fold's inner plan.
i	(integer(1) or character(1)) Split position or split name. Whole-number numeric values are accepted and stored as integers. Character values are matched against names(plan$splits).
parts	(character() or NULL; default NULL) Parts to materialize, a subset of c("analysis", "assessment", "oob"). NULL means every part the plan's method defines. "analysis" is mandatory, and parts the method does not define error.

Details

Fold anatomy:

A fold is spec-major and method-invariant, so consumers write one loop whatever the plan's method. fold$specs holds one entry per spec, positionally aligned with plan$specs. Each entry carries its own fit seed, the schema learned on the analysis rows, one design_inputs payload per materialized part, and one design_report per part. A fitting loop reads fold$specs[[j]]$analysis$x. Row-level attribution lives in the reports, for example fold$specs[[j]]$report$assessment$novel_levels. A single-spec plan keeps the list shape, so its entry is fold$specs[[1L]].

Schema learning and leakage:

Each fold's schema is learned on the analysis rows only, and assessment and oob rows are encoded with that schema. No schema is ever fitted on held-out rows, so held-out rows never determine factor levels, dropped degenerate terms, or column order. The response is the exception to this rule. The plan fixes the response levels at construction from the plan data, and every fold uses those levels. An analysis set that lacks a response level keeps that level with no rows. Because every fold re-learns its schema, the degenerate-drop and rank-deficiency warnings can re-fire per fold, with condition classes bkmodel_warning_degenerate and bkmodel_warning_rank_deficient. A loop can muffle these known duplicates by class via withCallingHandlers() instead of blanket suppressWarnings(). An encoded held-out matrix has exactly the analysis columns in the analysis order. An analysis column whose level does not occur in a held-out part holds zeros.

Fold-local novel levels always encode to NA, whatever the spec's stored novel_levels policy. A rare level landing only on a fold's held-out side is a resampling artifact, not a data-quality event,

so the fold path catalogs it in the part's report and excludes the affected rows via the scoreable masks instead of erroring. `novel_levels = "fail"` remains a deployment-time policy enforced by `design_apply()`. A consumer that wants fail-fast semantics inside resampling can raise its own error from `fold$specs[[j]]$report$assessment$novel_levels`.

Data-dependent predictor terms are refit on each fold's analysis subset and frozen there. The freeze covers terms whose functions provide a `stats::makepredictcall()` method, such as `scale()`, `poly()`, and the `splines` bases `ns()` and `bs()`. Their analysis-fit parameters, such as means, orthogonal bases, and knots, are carried into the held-out encoding. Any other predictor transform must be pointwise, computing row i from row i alone. A data-dependent expression without such a method, for example `I((x - mean(x)) / sd(x))` or `cut(x, 2)`, is silently re-evaluated on each part with that part's own parameters, matching the behavior of `predict()` on base R model fits.

Because refitting happens on a resampled subset, a data-dependent term can degenerate on a small or unlucky draw or fold in a way the full data does not, most often `scale()` on a zero-variance column. Under `na_action = "na.omit"`, `"na.exclude"`, or `"na.fail"` the fold then errors with a message that names the split and, for a predictor term, the term. Under `"na.pass"` the fold keeps the missing values, and the scoreable masks mark the affected rows. Avoid this with a pointwise transform, or remove the term.

Response, weights, and offset expressions are never frozen. They are re-evaluated on every part and must be pointwise. A data-dependent response transform such as `scale(y)` silently uses each part's own parameters. See the pointwise expressions section of `design_spec()`.

Row identity:

Payloads keep every split row. Held-out parts are encoded under `na.pass` semantics whatever the spec's `na_action`, so `row_index` matches the split exactly and fold-local `NA` lands in the report and the scoreable masks instead of dropping rows. A bootstrap fold's `analysis$row_index` keeps draw order with duplicates, its `assessment$row_index` is the full row universe, and its `oob$row_index` is the universe minus the distinct drawn rows.

Scoreable rows:

A row of a fold part is scoreable when its encoded `x` is complete and finite in every spec of the fold. The rule covers `NA` from missing values and novel levels plus `Inf` and `NaN` produced by formula terms, for example `log(0)`. The masks test only `x`. Under `na_action = "na.pass"`, a row with a missing response stays scoreable. `fold$scoreable` holds one logical mask per materialized part, analysis included, each aligned to that part's rows. One mask serves every spec because a part's rows are identical across specs.

The masks are exposed, never applied. The common-scoreable-row rule is a convention, not an enforced guarantee, so every workflow should subset explicitly in one line:

```
keep <- fold$scoreable$assessment
x <- fold$specs[[1L]]$assessment$x[keep, , drop = FALSE]
```

The analysis mask exists so optimism estimators score the bootstrap and original-sample terms under the same rule.

Part selection:

`parts` selects which parts to materialize. `NULL` means every part the method defines, `analysis` and `assessment` for CV plans, plus `oob` for bootstrap plans. Omitted parts are absent from the spec entries, `scoreable`, and the reports. The analysis part cannot be omitted because the schema

is learned on it. Requesting a part the plan's method does not define, for example `"oob"` on a CV plan, is an error.

`parts` is the opt-out for the bootstrap assessment part, which re-encodes the full row universe per draw. A .632-style consumer passes `c("analysis", "oob")`, and an apparent-versus-boot consumer in the style of `rms::validate()` passes `c("analysis", "assessment")`.

Coordinates:

`fold$rep` and `fold$fold` are first-class coordinates. On CV plans, `rep` is the repetition and `fold` the fold within it. On bootstrap plans, `rep` is the draw index, and on `design_splits()` plans it is the list position. `fold` is `NA_integer_` on both.

Seeds and reproducible fits:

All seeds are read from the plan's stored allocation. `fold$seed` is the fold-level fit seed, and each `fold$specs[[j]]$seed` is that (fold, spec) entry's own distinct seed, so every fit is reproducible alone, skippable, and parallel-safe. Call `set.seed(fold$specs[[j]]$seed)` immediately before fitting.

Materialization is pure. `design_fold()` never reads or advances the ambient random number stream, and returns an identical fold regardless of materialization order or subset, including its `inner` plan.

Inner plans:

When the plan was built with `inner = design_inner_cv(...)`, `fold$inner` is a complete `design_plan` over the fold's analysis rows, instantiated lazily from the plan's stored seed block. This holds on `design_splits()` plans too. It has the same class, is materialized with this same verb, and is never nested further. Inner splits partition only the outer fold's analysis rows, so tuning never sees held-out rows (Varma and Simon 2006; Cawley and Talbot 2010).

Under a CV outer, the inner plan inherits `strata` and `group` by default, re-resolved on the fold's analysis rows. The inner universe is the fold's analysis identities, so `fold$inner$universe` equals `fold$specs[[j]]$analysis$row_index` elementwise.

Under a bootstrap outer, the draw is the training sample and each drawn copy is a distinct observation, so the inner plan has the draw as its universe, in draw order with duplicates kept, and its inner CV partitions draw positions. `group` and `strata` are inherited by default. The rows of one drawn copy of a group stay on one inner side. Each draw position keeps the stratum of the row it copies. A descriptor value other than `"inherit"` replaces the inherited one. A data row or group drawn more than once can therefore land on both inner sides as distinct copies. This caveat is inherent to the standard method and produces no warning.

Cost:

Each `design_fold()` call re-learns every schema and re-encodes every requested part. Consumers that revisit folds should materialize each fold once and keep what they need.

Value

A list of class `design_fold` with elements:

`id` Character scalar, the split name.

`rep` Integer scalar, the repetition, bootstrap draw, or supplied-split position.

`fold` Integer scalar, the CV fold within `rep`, and `NA_integer_` on bootstrap and `design_splits()` folds.

`seed` Integer scalar, the fold-level fit seed.

`scoreable` Named list of logical masks, one per materialized part, each aligned to that part's rows and `TRUE` exactly where the encoded `x` row is complete and finite in every spec.

`inner` A `design_plan` over the analysis rows when the plan carries an inner descriptor, otherwise `NULL`.

`specs` List with one entry per spec, positionally aligned with `plan$specs`. Each entry holds `seed` (this entry's fit seed), `schema` (a `design_schema` learned on the analysis rows), one `design_inputs` per materialized part (`analysis`, `assessment`, and on bootstrap plans `oob`), and `report`, a list of one `design_report` per materialized part.

References

Cawley GC, Talbot NLC (2010). "On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation." *Journal of Machine Learning Research*, **11**(70), 2079–2107. http://jmlr.org/papers/v11/cawley10a.html.

Varma S, Simon R (2006). "Bias in error estimation when using cross-validation for model selection." *BMC Bioinformatics*, **7**(1), 91. ISSN 1471-2105. doi:10.1186/14712105791.

See Also

`design_cv()`, `design_bootstrap()`, `design_splits()`, `design_inner_cv()`, `design_learn()`, `design_apply()`, `ordinal_matrix_to_factor()`

Examples

```
#--------------------------------------------------------------------------
# Materialize a CV fold and score its scoreable assessment rows
#--------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  age = round(rnorm(24, mean = 60, sd = 8)),
  treatment = factor(rep(c("control", "active"), 12)),
  outcome = rnorm(24)
)

spec <- design_spec(
  outcome ~ age + treatment,
  encoding = "dummy",
  intercept = TRUE
)

plan <- design_cv(spec, analysis, n_folds = 4L, seed = 42L)
fold <- design_fold(plan, 1L)

entry <- fold$specs[[1L]]
```

```
fit <- stats::lm.fit(x = entry$analysis$x, y = entry$analysis$y)

# Subset to the fold's scoreable rows before scoring. The mask is exposed,
# never applied, so this one-line step belongs in every workflow.
keep <- fold$scoreable$assessment
x_test <- entry$assessment$x[keep, , drop = FALSE]
y_test <- entry$assessment$y[keep]
sqrt(mean((y_test - drop(x_test %*% fit$coefficients))^2))

#--------------------------------------------------------------------------
# Nested CV: fold$inner is an ordinary plan over the analysis rows
#--------------------------------------------------------------------------
nested <- design_cv(
  spec,
  analysis,
  n_folds = 4L,
  inner = design_inner_cv(n_folds = 3L),
  seed = 42L
)

outer <- design_fold(nested, 1L)
inner_fold <- design_fold(outer$inner, 1L)

# Inner splits partition only the outer analysis rows.
all(
  inner_fold$specs[[1L]]$assessment$row_index %in%
    outer$specs[[1L]]$analysis$row_index
)

#--------------------------------------------------------------------------
# Bootstrap: skip the full-universe assessment part when it is unused
#--------------------------------------------------------------------------
bplan <- design_bootstrap(spec, analysis, n_boot = 5L, seed = 42L)
bfold <- design_fold(bplan, 1L, parts = c("analysis", "oob"))

names(bfold$specs[[1L]])
names(bfold$scoreable)

# Out-of-bag scoring under the same one-line subset step.
keep <- bfold$scoreable$oob
oob_rows <- bfold$specs[[1L]]$oob$row_index[keep]
oob_rows
```

`design_inner_cv` *Describe an inner cross-validation design*

Description

`design_inner_cv()` describes an inner cross-validation design without instantiating it. Pass the

result as `inner` to `design_cv()`, `design_bootstrap()`, or `design_splits()` to plan tuning inside every fold or draw.

Nested cross-validation is `design_cv()` with an `inner` descriptor. There is no separate nested constructor.

Usage

```
design_inner_cv(
  n_folds = 5L,
  n_reps = 1L,
  strata = "inherit",
  group = "inherit"
)
```

Arguments

`n_folds`	(`integer(1)`; default `5L`; range `[2, .Machine$integer.max]`) Inner folds per repetition. Whole-number numeric values such as `5` and `5.0` are accepted and stored as integers.
`n_reps`	(`integer(1)`; default `1L`; range `[1, .Machine$integer.max]`) Inner repetitions. Use values greater than one for repeated inner cross-validation.
`strata`	(`"inherit"`, `NULL`, or one-sided formula; default `"inherit"`) Inner fold stratification. `"inherit"` takes the outer plan's `strata` under every outer method. `NULL` builds explicitly unstratified inner folds. A one-sided formula sets the inner rule directly; `~ 1` stratifies by the outcome.
`group`	(`"inherit"`, `NULL`, or one-sided formula; default `"inherit"`) Inner fold grouping. `"inherit"` takes the outer plan's `group`. `NULL` builds explicitly ungrouped inner folds.

Details

The descriptor carries sampling parameters only. It holds no data, no indices, no seed, and no inner of its own, so recursion stops at depth one. The descriptor supports cross-validation only.

`n_reps` supports repeated inner cross-validation, so the inner protocol can reproduce a deployment tuning protocol exactly. Unbiased evaluation of a tuned procedure requires the inner protocol to match the protocol it evaluates (Varma and Simon 2006; Cawley and Talbot 2010).

The inner fold rule is a property of the procedure under assessment, not of the outer resampling design. `strata` and `group` therefore default to `"inherit"`, which takes the outer plan's operand under every outer method, a bootstrap outer included. The draw replaces the outer sampling frame. The draw does not control how its rows go into the inner folds. Each draw position keeps the stratum of the row it copies, so the inner folds can stratify on it. Pass `NULL` for an explicitly unstratified or ungrouped inner plan, or a one-sided formula to set the inner rule directly. `NULL` already means "no strata", so the inherit default must be a different value. The sentinel is therefore the string `"inherit"`. Use `"inherit"` unless the inner rule must differ from the outer rule.

`design_fold()` instantiates the descriptor lazily against each fold's analysis rows from the plan's stored seed allocation, so materialization never touches the ambient random number stream.

Value

A list of class `design_inner_cv` with elements:

`n_folds` Integer scalar, inner folds per repetition.

`n_reps` Integer scalar, inner repetitions.

`strata` `"inherit"`, `NULL`, or a one-sided formula.

`group` `"inherit"`, `NULL`, or a one-sided formula.

References

Cawley GC, Talbot NLC (2010). "On Over-fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation." *Journal of Machine Learning Research*, **11**(70), 2079–2107. http://jmlr.org/papers/v11/cawley10a.html.

Varma S, Simon R (2006). "Bias in error estimation when using cross-validation for model selection." *BMC Bioinformatics*, **7**(1), 91. ISSN 1471-2105. doi:10.1186/14712105791.

See Also

`design_cv()`, `design_bootstrap()`, `design_splits()`

Examples

```
#-------------------------------------------------------------------------
# Plan nested cross-validation with an inner descriptor
#-------------------------------------------------------------------------
library(bkmodel)

# Five inner folds, three inner repetitions.
inner <- design_inner_cv(n_folds = 5L, n_reps = 3L)
inner

# Nested cross-validation is design_cv() with this descriptor.
analysis <- data.frame(
  outcome = rnorm(30),
  age = rnorm(30, mean = 60, sd = 8)
)
spec <- design_spec(outcome ~ age)
plan <- design_cv(
  spec,
  analysis,
  n_folds = 3L,
  inner = design_inner_cv(n_folds = 2L, n_reps = 2L),
  seed = 7L
)
plan$inner
```

`design_learn` *Learn design schemas from data*

Description

`design_learn()` evaluates one or more `design_spec()` objects on one dataset and learns each spec's schema. Every spec is learned on the same retained rows, so fit statistics from competing models stay comparable. It does not fit a statistical model.

The result holds, per spec, a data-free `design_schema`, a model-ready `design_inputs` payload, and a `design_report` with row-level attribution.

Usage

```
design_learn(specs, data)
```

Arguments

`specs`	(`design_spec` or `list` of `design_spec`) One or more design specifications from `design_spec()`. A bare spec is treated as a length-one list.
`data`	(`data.frame`) Data used to learn the schemas. Objects that inherit from `data.frame`, including tibbles, are accepted.

Details

Specs:

`specs` is a `design_spec` or a list of them. A bare spec is lifted to a length-one list, so a single spec and a multi-spec call return the same shape. All specs must share the same response expression, absence included, and factor responses must share declared levels.

Row universe:

Each spec's `na_action` resolves the rows it retains. The row universe is the sorted intersection of the per-spec retained rows, so every spec is learned on identical rows. When the intersection drops rows for any spec, `design_learn()` warns with condition class `bkmodel_warning_universe`. Use `withCallingHandlers()` to muffle the warning by class when the drop is expected. When the intersection is empty, `design_learn()` errors. The `retained` element records each spec's pre-intersection rows, so `setdiff(retained[[j]], universe)` lists the rows the intersection removed for spec `j`.

Missing values:

`na_action` acts only through retention. Retention flags a row when a predictor, the response, a weight, or an offset is `NA` for that spec. `"na.pass"` keeps all rows, `"na.omit"` and `"na.exclude"` keep unflagged rows, and `"na.fail"` errors when any row is flagged. Case ids are metadata and never participate in `na_action`. On retained rows, weights must be non-missing, finite, and non-negative, and offsets must be non-missing and finite. Weight and offset validation runs per spec,

before the cross-spec intersection. A row can therefore fail validation for one spec even when a different spec's na_action would exclude that row from the returned universe.

Under "na.pass" the encoded matrix can contain NA. design_learn() never drops those rows. The report's missing mask attributes them.

Degenerate categorical predictors:

After the universe is fixed, categorical predictors with fewer than two observed levels on universe rows are degenerate. Every formula term referencing a degenerate predictor is dropped in one pass, with a warning of condition class bkmodel_warning_degenerate. Dropping never changes the universe. If dropping leaves no predictor terms and the spec has no intercept, design_learn() errors.

Schema learning also warns with condition class bkmodel_warning_rank_deficient when a spec combines encoding = "one_hot", intercept = TRUE, and categorical predictors, because that design matrix is rank deficient. See design_spec() for the encoding rules.

Reports:

The design_report has one shape from both design_learn() and design_apply(). On the learn side the novel mask is all FALSE and novel_levels is empty, because the schema is learned from the same rows it encodes. The nonfinite mask and nonfinite_cols catalog attribute every Inf, -Inf, and NaN the encoded matrix contains, for example from log(0). bkmodel reports non-finite encoded predictors and never errors on them. The missing mask reflects NA in the columns of the final encoded schema. A predictor dropped for degeneracy carries its NA out of the mask along with the column.

Response levels:

For a response that evaluates to a factor or an ordered factor on data, the schema freezes its declared levels. The rule includes transformed responses such as factor(y). For one-sided and non-factor responses, response_levels = NULL.

Value

A plain list with elements:

universe Sorted integer row identities shared by all specs.

retained List with one integer vector per spec, the rows that spec's na_action retained before intersection.

specs List with one entry per spec, each a list(schema, inputs, report).

Each specs entry holds:

schema A design_schema, the data-free encoding description. It holds the predictor terms, per-column schema, the column map, encoding and intercept flags, the response as R source text and its frozen levels, role expressions, and the NA and novel-level policies. Formula environments are replaced, so serialized size does not grow with nrow(data).

inputs A design_inputs with x, y, weights, offset, case_id, and row_index, all aligned to universe rows.

report A design_report with missing, novel, and nonfinite logical masks aligned to inputs$row_index, plus the novel_levels and nonfinite_cols catalogs.

See Also

design_spec(), design_apply(), design_fold()

Examples

```
#--------------------------------------------------------------------------
# Learn one schema and fit a weighted linear model
#--------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  outcome = rnorm(12),
  age = round(rnorm(12, mean = 60, sd = 8)),
  treatment = factor(rep(c("control", "active"), 6)),
  exposure = runif(12, min = 0.5, max = 2),
  inverse_prob_weight = runif(12, min = 0.8, max = 1.2)
)

spec <- design_spec(
  outcome ~ age + treatment,
  weights = ~inverse_prob_weight,
  offset = ~ log(exposure),
  encoding = "dummy",
  intercept = TRUE
)

lr <- design_learn(spec, analysis)
lr$universe

inputs <- lr$specs[[1L]]$inputs
model <- stats::lm.wfit(
  x = inputs$x,
  y = inputs$y,
  w = inputs$weights,
  offset = inputs$offset
)
model$coefficients

#--------------------------------------------------------------------------
# Learn two specs on a shared row universe
#--------------------------------------------------------------------------
analysis$marker <- analysis$age
analysis$marker[1L] <- NA

spec_a <- design_spec(outcome ~ age, na_action = "na.omit")
spec_b <- design_spec(outcome ~ marker, na_action = "na.omit")

# Spec B retains fewer rows, so the intersection warns by condition class.
lr2 <- withCallingHandlers(
  design_learn(list(spec_a, spec_b), analysis),
  bkmodel_warning_universe = function(w) invokeRestart("muffleWarning")
```

```
)
lr2$universe
lapply(lr2$retained, length)
```

`design_spec` *Define a design specification*

Description

`design_spec()` defines how analysis data should be turned into model-ready inputs. It records the model formula, optional row roles, and design-matrix options. It does not inspect data, expand `.`, encode factors, remove missing rows, or fit a statistical model.

Use the returned object with `design_learn()` to learn a reproducible design schema from training data. Use it with `design_cv()` or `design_bootstrap()` to build resampling plans that use the same design rules.

Usage

```
design_spec(
  formula,
  weights = NULL,
  offset = NULL,
  case_id = NULL,
  exclude = NULL,
  encoding = "one_hot",
  intercept = FALSE,
  na_action = "na.pass",
  novel_levels = "na"
)
```

Arguments

`formula`	(formula) One-sided or two-sided model formula. Use a two-sided formula when the design has a response. Use a one-sided formula for predictor-only designs. `.` is resolved later, when the specification is fitted to data.
`weights`	(formula or `NULL`; default `NULL`) Optional one-sided formula for analysis weights. The right-hand side must evaluate to one numeric vector with one value per row.
`offset`	(formula or `NULL`; default `NULL`) Optional one-sided formula for an offset. The right-hand side is evaluated as one R expression, so `+` and `-` are arithmetic operators, not formula-term operators. It must evaluate to one numeric vector with one value per row. Offset components may also be supplied as `offset()` terms in `formula`. All components are summed into one numeric offset vector.

`case_id` (formula or `NULL`; default `NULL`)
Optional one-sided formula for row or subject identifiers. The right-hand side must evaluate to one atomic vector with one value per row.

`exclude` (formula or `NULL`; default `NULL`)
Optional one-sided formula listing variables to remove from `.` expansion. The right-hand side may contain only bare variable names separated by `+`. Explicit formula terms are not removed.

`encoding` (`character(1)`; default `"one_hot"`)
One of `"one_hot"` or `"dummy"`. Categorical predictor encoding used when the fitted design matrix is built.

`intercept` (`logical(1)`; default `FALSE`)
Whether to include a numeric `"(Intercept)"` column in fitted design matrices.

`na_action` (`character(1)`; default `"na.pass"`)
One of `"na.pass"`, `"na.omit"`, `"na.fail"`, or `"na.exclude"`. Missing-value policy for evaluated predictors and responses.

`novel_levels` (`character(1)`; default `"na"`)
One of `"na"` or `"fail"`. Handling for categorical levels in new data that were not observed during fitting.

Details

Workflow:

A typical workflow has three steps. First, call `design_spec()` to declare the statistical design. Second, call `design_learn()` on the analysis data to evaluate formulas, expand `.`, learn factor levels, and build the training design matrix. Third, use `design_apply()` to encode validation or prediction data with the same columns and factor levels.

Keeping these steps separate makes the analysis reproducible. The specification can be reused across cross-validation folds, bootstrap samples, and final model fitting without silently changing the predictor schema.

Roles:

The `weights`, `offset`, and `case_id` arguments declare columns or expressions with special roles. Each role may be `NULL` or a one-sided formula. Role formulas may not use `.`. Source variables used by these roles are removed from `.` expansion automatically and may not also be named as explicit predictors.

Response source variables may not also be used as predictors, weights, offsets, or case IDs. This prevents outcome leakage during training, assessment, and resampling. If an overlap is intentional, create a separate response column before calling `design_spec()`.

`exclude` may be `NULL` or a one-sided formula whose right-hand side contains only bare variable names separated by `+`. The exclusion set is `all.vars(exclude)`. Variables in `exclude` are removed from `.` expansion only. Explicitly named predictor terms are kept even when they also appear in `exclude`.

Right-hand-side `offset()` terms inside `formula` are collected as offset source variables. Variables referenced inside those calls are removed from `.` expansion and may not appear as explicit predictors. If `formula` contains `offset()` terms and the `offset` argument is also supplied, the evaluated components are summed into one numeric offset vector.

Pointwise expressions:

Response, weights, and offset expressions are re-evaluated on every data subset they meet. These subsets are the learn rows, each fold's analysis and assessment sets, and new data passed to `design_apply()`. No learn-time parameters are stored for them, so each expression must be pointwise, computing row i from row i alone. Pointwise transforms such as a `log(y)` response or a `log(exposure)` offset are supported. A data-dependent expression such as a `scale(y)` response computes its parameters from whichever subset it is evaluated on. Under resampling this silently puts the assessment response on a different scale than the response the model was trained on. If the model needs a transformed response, weights, or offset whose definition depends on other rows, create that column in the data before calling `design_spec()`.

Predictor terms follow a different rule. Terms whose functions provide a `stats::makepredictcall()` method, such as `scale()`, `poly()`, `ns()`, and `bs()`, are frozen at their learn-row parameters and reused when new rows are encoded. Any other predictor transform must also be pointwise. See the schema learning section of `design_fold()` for the full set of rules.

Functions called in `formula`, `weights`, `offset`, and `case_id` must be defined in the global environment or provided by an attached package. The learned schema replaces each formula's defining environment with the global environment so that the schema serializes without capturing local data. A function defined only in a local scope therefore resolves during `design_learn()`, which still evaluates in the defining environment, but fails in `design_apply()` and `design_fold()` with a could-not-find-function error. Define helper functions at top level, or attach the package that provides them, wherever the schema is applied.

Encoding:

`encoding` sets how categorical predictors enter the design matrix. With `encoding = "dummy"`, categorical predictors use treatment contrasts and the reference level is represented by zeros in the treatment columns. This is the usual choice for linear and generalized linear model workflows that include an intercept. With `encoding = "one_hot"`, categorical predictors use one indicator column per level. Combined with the default `intercept = FALSE`, this gives a design matrix similar to `glmnet::makeX()` factor encoding, useful when a downstream fitter supplies its own intercept. bkmodel encodes an ordered factor like an unordered factor under both encodings. It does not use polynomial contrasts.

`intercept` controls only whether fitted design matrices contain a numeric `"(Intercept)"` column. It is separate from any intercept option supplied to a downstream model fitter. If the formula suppresses the intercept with `-1` or `+0`, `intercept = TRUE` errors when the schema is learned.

Some combinations produce rank-deficient design matrices. `encoding = "one_hot"` with `intercept = TRUE` is rank deficient when categorical predictors are present, and schema learning in `design_learn()` and `design_fold()` warns with condition class `bkmodel_warning_rank_deficient`. Additive formulas with multiple one-hot categorical predictors can be rank deficient even when `intercept = FALSE`.

Missing values:

`na_action` is applied when `design_learn()`, `design_apply()`, and design-aware resampling functions evaluate predictors and responses. `"na.pass"` keeps rows with missing predictor or response values. `"na.omit"` and `"na.exclude"` keep only complete rows. `"na.fail"` errors if any evaluated predictor or response value is missing.

Weights and offsets are validated more strictly. On every retained row, weights must be non-missing, finite, and non-negative. On every retained row, offsets must be non-missing and finite. Case IDs are metadata and do not participate in `na_action`.

Novel levels:

`novel_levels` controls categorical values in new data that were not observed when the design schema was fitted. `"na"` converts those values to `NA` before encoding. `"fail"` errors and reports the affected variables and values. The policy applies when scoring new data via `design_apply()`. Fold materialization via `design_fold()` always encodes fold-local novel levels to `NA`, because a rare level landing on a fold's held-out side is a resampling artifact, not a data-quality event.

Value

An object of class `"design_spec"`, a list that stores each argument unchanged under the same name. The elements are `formula`, `weights`, `offset`, `case_id`, `exclude`, `encoding`, `intercept`, `na_action`, and `novel_levels`. The returned object does not contain a design matrix, response vector, factor schema, retained-row index, or fitted model.

See Also

`design_learn()`, `design_apply()`, `design_fold()`, `design_cv()`, `design_bootstrap()`

Examples

```
#-------------------------------------------------------------------------
# Declare a design, fit its schema, and model the analysis data
#-------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  patient_id = sprintf("P%02d", 1:12),
  outcome = rnorm(12),
  age = round(rnorm(12, mean = 60, sd = 8)),
  treatment = factor(rep(c("control", "active"), 6)),
  stage = factor(rep(c("I", "II", "III"), each = 4)),
  exposure = runif(12, min = 0.5, max = 2),
  inverse_prob_weight = runif(12, min = 0.8, max = 1.2),
  chart_score = rnorm(12)
)

spec <- design_spec(
  outcome ~ .,
  weights = ~ inverse_prob_weight,
  offset = ~ log(exposure),
  case_id = ~ patient_id,
  exclude = ~ chart_score,
  encoding = "dummy",
  intercept = TRUE,
  na_action = "na.omit",
  novel_levels = "fail"
)

lr <- design_learn(spec, analysis)
inputs <- lr$specs[[1L]]$inputs
```

```
model <- stats::lm.wfit(
  x = inputs$x,
  y = inputs$y,
  w = inputs$weights,
  offset = inputs$offset
)
model$coefficients

#------------------------------------------------------------------------------
# Apply the same columns and factor levels to validation data
#------------------------------------------------------------------------------
validation <- data.frame(
  patient_id = sprintf("V%02d", 1:3),
  age = c(55, 67, 61),
  treatment = factor(c("active", "control", "active")),
  stage = factor(c("II", "I", "III")),
  exposure = c(1.1, 0.9, 1.5)
)

ap <- design_apply(
  lr$specs[[1L]]$schema,
  validation,
  response = "ignore",
  weights = "ignore"
)

linear_predictor <- drop(ap$inputs$x %*% model$coefficients)
linear_predictor <- linear_predictor + ap$inputs$offset
linear_predictor
```

`design_splits` *Build a plan from user-supplied splits*

Description

`design_splits()` builds a `design_plan` from a plain list of splits the caller supplies. It is the escape hatch for schemes the native constructors cannot draw: time-series, blocked, or external-package splits. Supplied splits are validated against the shared row universe, the rows all specs retain, so every split respects each spec's `na_action`. The returned `design_plan` holds the specs, the data, the universe, and the splits. It fits no model and builds no design matrix. Use `design_fold()` to materialize one split at a time.

Usage

```
design_splits(
  specs,
  data,
  splits,
```

```
  method = c("cv", "bootstrap"),
  inner = NULL,
  seed = NULL
)
```

Arguments

specs	(design_spec or list of design_spec) One or more design specifications from design_spec(). A bare spec is treated as a length-one list. All specs must share the same response expression, and factor responses must share declared levels.
data	(data.frame) Data the split row numbers refer to. Objects that inherit from data.frame, including tibbles, are accepted.
splits	(list) Plain unclassed list of splits. Each element is list(analysis, assessment) of data row numbers, with assessment optional.
method	(character(1); default "cv") "cv" or "bootstrap". Fixes the held-out parts, the assessment default, and duplicate handling.
inner	(design_inner_cv or NULL; default NULL) Inner cross-validation descriptor from design_inner_cv(). When present, every materialized fold carries an inner plan over its analysis rows.
seed	(integer(1) or NULL; default NULL) Plan seed. NULL draws one integer from the ambient stream. An explicit value leaves the caller's stream untouched.

Details

Entry shape:

splits is a plain unclassed list. Each element is list(analysis, assessment) of data row numbers. analysis is required and non-empty. Duplicated analysis rows are allowed only under method = "bootstrap", where they are draw multiplicities. assessment is optional. Under "cv" it defaults to the universe rows absent from analysis, and under "bootstrap" to the full universe. The out-of-bag rows are always derived at materialization as the universe rows absent from the draw, so splits never carry an oob component. Classed split objects are not accepted, and components other than analysis and assessment are errors. In particular, user-supplied fit seeds are not accepted. Names on the supplied list are ignored. Splits are named in the "split4" style by list position, zero-padded when the count needs it.

Row universe:

specs is a design_spec or a list of them, as in design_learn(). The row universe is the sorted intersection of the per-spec retained rows, computed by the same internal helper as design_learn(), design_cv(), and design_bootstrap(). When the intersection drops rows for any spec, construction warns with condition class bkmodel_warning_universe. When the intersection is empty, construction errors.

Trimming:

Row numbers outside the computed universe are trimmed from both sides of every split, with one warning of class `bkmodel_warning_splits_trimmed` reporting the trimmed row numbers per split. Callers build splits before knowing what the specs retain, so rejecting out-of-universe rows would force them to pre-resolve NAs. A split whose `analysis` side becomes empty after trimming is an error. Surviving row numbers convert internally to positions into `plan$universe`, so stored splits have one shape on every construction path and `plan$universe[split$analysis]` recovers data row numbers. CV sides are stored sorted, matching the native constructors, and a bootstrap `analysis` keeps its supplied order as the draw order.

Validation:

Row numbers must be positive whole numbers no larger than `nrow(data)`, and `assessment` rows must be unique. After trimming, a `"cv"` split's `assessment` must be disjoint from its `analysis`, and a supplied `"bootstrap"` `assessment` must be set-equal to the universe, so supplying it only spells out the default. Violations error. A `"cv"` split's `assessment` does not have to cover the universe, so forward-window and blocked schemes are expressible. Beyond these rules, index provenance is the user's responsibility.

Coordinates and seeds:

A split's coordinate is its list position: `rep` is the position and `fold` is `NA_integer_`. Appending splits therefore preserves earlier seeds: rebuilding with the same `seed` and extra splits appended leaves the existing splits and all existing seeds bit-identical. Only the values are stable, not the names: `plan$splits` names zero-pad to the current split count, so growth across a padding-width boundary (9 to 10, 99 to 100) renames an existing split without changing it. Index `plan$splits` by position across incremental growth. Fit seeds follow the allocation recipe of the native constructors, with one fold-level seed and one per-spec entry seed per split. The seed values are not the same. A native constructor draws the split assignment before the fit seeds, and `design_cv()` allocates one seed block per repetition, not per split. For these reasons, `design_splits()` on the splits of a native plan does not replay its fit seeds or its inner splits, even with the same `seed`. To replay a native plan, call its constructor again with the same arguments. See `design_cv()` for the shared seed model, its granularity, and its documented non-goals.

Inner cross-validation:

Pass `inner = design_inner_cv(n_folds, n_reps)` to make every materialized fold carry an inner plan. bkmodel generates the inner splits over each supplied split's analysis rows, so tuning never sees held-out rows even with external splits. Users never supply nested split lists. Under `method = "bootstrap"` the inner plan partitions draw positions with each drawn copy carried as a distinct observation, as in `design_bootstrap()`.

Construction warnings:

Documented warnings carry condition classes, so a consumer can muffle a known duplicate selectively with `withCallingHandlers()`.

- `bkmodel_warning_universe`: the universe intersection dropped rows for any spec.
- `bkmodel_warning_case_id`: a spec carries a case_id role and duplicated ids exist among universe rows. `design_splits()` has no `group` argument, so rows sharing a case id stay together only when the supplied splits keep them together.
- `bkmodel_warning_splits_trimmed`: out-of-universe row numbers were trimmed from the supplied splits.

- bkmodel_warning_strata: with inner, the analysis set of a supplied split holds fewer than two sampling units of an inner stratum. An inner analysis set then has none of it.

Value

A list of class design_plan with elements:

specs List of design_spec, length one or more.

data The data as given.

universe Sorted integer row identities this plan resamples over.

retained List with one integer vector per spec, the rows that spec's na_action retained before intersection.

response_levels Character vector of the response levels that design_learn() freezes on data, or NULL when the response is not a factor. Every fold uses these levels for the response.

splits Named list of splits, each list(analysis, assessment) of integer positions into universe.

method "cv" or "bootstrap", as given.

params List of n_splits plus the resolved strata and group, both NULL on this construction path.

inner The design_inner_cv descriptor, or NULL.

seed Integer scalar, the resolved plan seed.

seeds The seed allocation table, one block per split holding the rep sub-seed, the fold seed, per-spec entry seeds, and inner plan blocks.

See Also

design_spec(), design_learn(), design_cv(), design_bootstrap(), design_inner_cv(), design_fold()

Examples

```
#--------------------------------------------------------------------------
# Rolling-origin CV splits on ordered data
#--------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  month = 1:24,
  demand = rnorm(24, mean = 100, sd = 10),
  temperature = rnorm(24, mean = 15, sd = 5)
)

spec <- design_spec(demand ~ temperature)

splits <- list(
  list(analysis = 1:12, assessment = 13:16),
  list(analysis = 1:16, assessment = 17:20),
  list(analysis = 1:20, assessment = 21:24)
)
```

```
plan <- design_splits(spec, analysis, splits, method = "cv", seed = 42L)
plan$method
names(plan$splits)

# Splits store positions into the universe.
plan$universe[plan$splits[["split3"]]$analysis]

#--------------------------------------------------------------------------
# Externally drawn bootstrap splits
#--------------------------------------------------------------------------
set.seed(7L)
draws <- lapply(1:5, function(i) {
  list(analysis = sample(24L, size = 24L, replace = TRUE))
})

bplan <- design_splits(spec, analysis, draws, method = "bootstrap", seed = 42L)

# The assessment default is the full universe; oob is derived later.
identical(
  bplan$splits[["split1"]]$assessment,
  seq_along(bplan$universe)
)
```

`get_dep_data` *Get dependent two-sample or one-sample test data*

Description

Prepare paired data or one-sample data from a formula. The result is ready for a paired test or a one-sample test, such as `stats::t.test()`. The function accepts three formula forms:

- `y ~ x | z` for paired data in tall format.
- `y ~ x` for paired data in wide format.
- `~x` for one sample, or for paired differences that are already calculated.

Usage

```
get_dep_data(data, formula, allow_one_sample = TRUE, agg_fun = "error")
```

Arguments

`data`	(data.frame) Data frame that holds the formula variables.
`formula`	(formula) Formula of form `y ~ x | z`, `y ~ x`, or `~x`. See Details for the accepted terms.

allow_one_sample
: (logical(1); default TRUE)
Whether to accept a formula of form ~x.

agg_fun
: (character(1) or function; default "error")
One of "error", "first", "last", "sum", "mean", "median", "min", or "max", or a function. Rule that computes each block and group cell from its rows in a y ~ x | z formula. The value "error" stops with an error when such rows occur. The other character values ignore NA values of y. A function runs on every non-empty cell, including a cell with one row. A function receives all y values of one cell, NA included, and must return one number. For example, agg_fun = mean gives NA for a cell that holds an NA, and agg_fun = "mean" ignores the NA. A function is slower than the built-in rules. For a formula without a block term, a value other than "error" has no effect, and the function emits a warning of class bkmodel_warning_agg_fun_ignored.

Details

Tall format:

The formula y ~ x | z has three terms:

- y is a numeric vector for the outcome.
- x is a factor with exactly two observed levels, which defines the two groups.
- z is a factor that identifies the block, such as the subject or the item.

The function reshapes the data to one row for each observed level of z, with one column for each group. The first observed level of x is the reference group, and the second observed level is the focal group. For example, data$x <- factor(data$x, levels = c("pre", "post")) makes pre the reference group and post the focal group. A downstream comparison then has the form mean(post - pre). The function ignores declared levels of x and z that have no rows.

The function converts a non-factor x or z with as.factor() and emits a warning of class bkmodel_warning_factor_conve
The reference group is then the smallest value of x. Numeric and logical values sort in increasing order. Character values sort in the collation order of the current locale, as in the formula method of stats::t.test(). Because the collation order depends on the locale, the reference group of a character x can change between machines. Pass x as a factor to set the reference group.

Two or more rows with the same block and group cause an error, unless agg_fun combines them.

Wide format:

The formula y ~ x has two terms:

- y is a numeric vector for the focal group.
- x is a numeric vector for the reference group.

Row i of y and row i of x are from the same subject or item. For the returned list out, a downstream difference has the form out$data[[out$focal_group]] - out$data[[out$ref_group]].

One sample:

In the formula ~x, x is a numeric vector for the outcome or for the paired differences. Set allow_one_sample = FALSE to reject this form.

Terms:

Each term must reference a different column. The function compares the terms as written, and accepts `I(y) ~ y`. Each term must have one value for each row of `data`. A one-column matrix, such as the result of `base::scale()`, meets this rule. A bare name must be a column of `data`.

Operators such as `+` and `*` have special meanings in a formula. To use a different expression as a term, such as `log(y)` or `x > 0`, wrap it in `I()`. The function evaluates the expression inside `I()` in `data`, then in the environment of `formula`. On the left hand side, `I()` transforms the outcome and must give a numeric result. On the group side and on the block side, `I()` derives the term.

For example, `I(log(y)) ~ x | z` uses the log-transformed outcome. The formula `y ~ x | I(interaction(subject, session))` uses each subject and session pair as one block.

Special values:

For `~x` and `y ~ x`, the function returns `NA`, `NaN`, and `Inf` values unchanged. For a `data` with zero rows, these two forms give zero-length vectors.

For `y ~ x | z`, the function drops a row with a missing value of `x` or `z` before it reshapes the data. A level of `x` counts as observed when at least one row has non-missing `x` and `z`. This rule applies even when all `y` values of the level are missing. Without duplicate rows, a missing value of `y` stays in its cell. A cell with no row becomes `NA_real_`. See `tall_to_wide()` for the handling of `NA` when `agg_fun` combines rows.

An observed level of `x` that is `""` or `NA` (from `base::addNA()`) cannot name an output column, and the function stops with an error. For `y ~ x | z`, a `data` with zero rows has no observed levels, and the function stops with an error.

Value

A list with the elements below.

`data` The outcome values by group. For `~x`, a named list of length 1 that holds the outcome. For `y ~ x`, a named list of length 2, with the focal group first and the reference group second. For `y ~ x | z`, a data frame with one row for each observed level of `z`, in `levels(z)` order. Its columns are the focal group, the reference group, and the block factor.

`focal_group` `character(1)`. The response term as written for `y ~ x`, the single term for `~x`, or the second observed level of `x` for `y ~ x | z`.

`ref_group` `character(1)` or `NULL`. The group term as written for `y ~ x`, the first observed level of `x` for `y ~ x | z`, or `NULL` for `~x`.

`block` `character(1)` or `NULL`. The block term as written for `y ~ x | z`, or `NULL` for the other forms.

`parse_formula` The list that `parse_formula()` returns for `formula`.

See Also

`parse_formula()`, `get_ind_data()`

Examples

```
#-------------------------------------------------------------------------
# get_dep_data() example
#-------------------------------------------------------------------------
library(bkmodel)
```

```
# A pre/post intervention study where each subject is measured twice
# pre is the first factor level, so it is the reference group
tall <- data.frame(
  score = c(5.1, 6.3, 5.8, 7.2, 6.4, 7.1, 6.9, 8.0),
  time = factor(
    rep(c("pre", "post"), times = 4),
    levels = c("pre", "post")
  ),
  subject = factor(rep(1:4, each = 2))
)

# Reshape tall data to one row per subject with focal and reference columns
res <- get_dep_data(data = tall, formula = score ~ time | subject)
res

# Extract the paired vectors with the returned group names
focal <- res$data[[res$focal_group]]
reference <- res$data[[res$ref_group]]

# Estimate the intervention effect as the mean within-subject difference
mean(focal - reference)

# The same pairing feeds a paired t-test
t.test(focal, reference, paired = TRUE)

# The same study already pivoted to one row per subject
wide <- data.frame(
  post = c(6.3, 7.2, 7.1, 8.0),
  pre = c(5.1, 5.8, 6.4, 6.9)
)
get_dep_data(data = wide, formula = post ~ pre)

# Pre-computed within-subject differences for a one-sample test
diffs <- data.frame(
  change = c(1.2, 1.4, 0.7, 1.1)
)
get_dep_data(data = diffs, formula = ~change)
```

`get_ind_data` *Get independent two-sample test data*

Description

Split a numeric outcome into two independent groups that a formula of form `y ~ x` defines. The result is ready for an independent two-sample test, such as `stats::t.test()`.

Usage

```
get_ind_data(data, formula)
```

Arguments

data	(data.frame) Data frame that holds the formula variables.
formula	(formula) Formula of form y ~ x. See Details for the accepted terms.

Details

The formula y ~ x has two terms:

- y is a numeric vector for the outcome.
- x is a factor with exactly two observed levels, which defines the two groups.

Reference group:

The first observed level of x is the reference group, and the second observed level is the focal group. For example, data$x <- factor(data$x, levels = c("pre", "post")) makes pre the reference group and post the focal group. A downstream comparison then has the form mean(post) - mean(pre). The function ignores declared levels of x that have no rows.

The function converts a non-factor x with as.factor() and emits a warning of class bkmodel_warning_factor_converte The reference group is then the smallest value of x. Numeric and logical values sort in increasing order. Character values sort in the collation order of the current locale, as in the formula method of stats::t.test(). Because the collation order depends on the locale, the reference group of a character x can change between machines. For example, "Treat" sorts after "control" in en_US.UTF-8 and before it in C.

Pass x as a factor to set the reference group.

Terms:

The two terms must reference different columns. Each term must have one value for each row of data. A one-column matrix, such as the result of base::scale(), meets this rule. A bare name must be a column of data.

Operators such as + and * have special meanings in a formula. To use a different expression as a term, such as log(y) or x > 0, wrap it in I(). The function evaluates the expression inside I() in data, then in the environment of formula. On the left hand side, I() transforms the outcome and must give a numeric result. For example, I(log(y)) ~ x uses the log-transformed outcome. On the right hand side, I() derives the grouping variable. For example, y ~ I(x > 0) groups the outcome by whether x is positive.

Special values:

The function drops a row with a missing value of x from both groups. A level of x counts as observed when at least one row has a non-missing x. This rule applies even when all y values of the level are missing. The function keeps NA, NaN, and Inf values of y unchanged in their groups.

An observed level of x that is "" or NA (from base::addNA()) cannot name a group, and the function stops with an error. A data with zero rows has no observed levels, and the function stops with an error.

Value

A list with the elements below.

`data` Named list of length 2 with the outcome values of each group in row order. The focal group is first and the reference group is second. Each element has the name of its level of `x`.

`focal_group` `character(1)`. The second observed level of `x`.

`ref_group` `character(1)`. The first observed level of `x`.

`parse_formula` The list that `parse_formula()` returns for `formula`.

See Also

`parse_formula()`, `get_dep_data()`

Examples

```
#-------------------------------------------------------------------------
# get_ind_data() example
#-------------------------------------------------------------------------
library(bkmodel)

# A two-arm study comparing an outcome under control and treatment
# control is the first factor level, so it is the reference group
data <- data.frame(
  score = c(5.1, 6.3, 5.8, 7.2, 8.1, 7.6, 8.4, 7.9),
  group = factor(
    rep(c("control", "treatment"), each = 4),
    levels = c("control", "treatment")
  )
)

# Split the outcome into the focal and reference groups
res <- get_ind_data(data = data, formula = score ~ group)
res

# Extract each group with the returned group names
focal <- res$data[[res$focal_group]]
reference <- res$data[[res$ref_group]]

# Estimate the treatment effect as the focal minus reference mean
mean(focal) - mean(reference)

# The same split feeds an independent two-sample t-test
t.test(focal, reference)

# Wrap the response in I() to compare on the log scale
get_ind_data(data = data, formula = I(log(score)) ~ group)
```

optim2 *General-purpose optimization*

Description

optim2() minimizes the function fun over the vector parameters. It runs one of four optimizers from base R and returns the same list elements for every method. Use it to fit a model by maximum likelihood or to minimize a smooth objective. Box constraints and a numerical Hessian are available.

Usage

```
optim2(
  fun,
  parameters,
  ...,
  gradient = NULL,
  method = "nlminb",
  lower = -Inf,
  upper = Inf,
  hessian = FALSE,
  control = list()
)
```

Arguments

fun	(function) Objective function to minimize. Its first argument is the parameter vector, and the arguments in ... follow. It must return a numeric scalar, and the value at parameters must be finite.
parameters	(numeric()) Start values, one for each parameter. The length must be one or more, and all values must be finite. Names of parameters go to fun, gradient, and the result.
...	Named arguments for fun and gradient. See the section "Arguments to fun" in Details.
gradient	(function or NULL; default NULL) Gradient of fun. It gets the same arguments as fun and returns a numeric vector of length length(parameters). If NULL, the methods that use derivatives approximate them by finite differences. Method "Nelder-Mead" does not accept a gradient.
method	(character(1); default "nlminb") One of "nlminb", "nlm", "Nelder-Mead", or "L-BFGS-B". Optimization algorithm. See the section "Methods" in Details.

`lower` (`numeric()`; default `-Inf`; range `[-Inf, Inf)`)
Lower bound for each parameter. The length must be 1 or `length(parameters)`. A scalar applies to all parameters. A finite value needs method `"nlminb"` or `"L-BFGS-B"`.

`upper` (`numeric()`; default `Inf`; range `(-Inf, Inf]`)
Upper bound for each parameter. The length must be 1 or `length(parameters)`. A scalar applies to all parameters. A finite value needs method `"nlminb"` or `"L-BFGS-B"`.

`hessian` (`logical(1)`; default `FALSE`)
Whether to compute the Hessian of `fun` at `estimate`. See the section "Hessian" in Details.

`control` (list; default `list()`)
Options for the optimizer. See the section "Control" in Details.

Details

Methods:

`method` selects the algorithm:

- `"nlminb"`, the default, is a quasi-Newton method from the PORT library, through `stats::nlminb()`. It accepts bounds.
- `"nlm"` is a Newton-type method, through `stats::nlm()`. It does not accept bounds.
- `"Nelder-Mead"` is a simplex method that uses no derivatives, through `stats::optim()`. It does not accept bounds or `gradient`, and it needs two or more parameters.
- `"L-BFGS-B"` is a limited-memory quasi-Newton method, through `stats::optim()`. It accepts bounds.

Nash (2014) discusses how to select among the optimizers in R.

Arguments to `fun`:

`fun` gets the parameter vector as its first argument, with the names of `parameters`, for every method. `stats::nlm()` removes these names, and `optim2()` restores them. `gradient` gets the same arguments as `fun`.

The arguments in `...` go only to `fun` and `gradient`, never to the optimizer. Every argument in `...` must have a name, or the function stops with an error. For example, a name such as `p` partially matches `parameters`, and the start values then go into `...` without a name. A name in `...` that is identical to an argument of `optim2()`, such as `hessian`, goes to that argument and not to `fun`. Give the argument of `fun` a different name in that case.

Checks:

`optim2()` checks its arguments before the optimizer starts. Each of these conditions stops the function with an error:

- `parameters` is `NULL`, has length zero, or contains `NA`, `NaN`, `Inf`, or `-Inf`.
- `lower` or `upper` contains `NA` or `NaN`.
- `lower` contains `Inf`, `upper` contains `-Inf`, or `lower` is greater than `upper`.
- `parameters` is outside the interval from `lower` to `upper`.
- A bound is finite and `method` is `"nlm"` or `"Nelder-Mead"`.

- `fun(parameters, ...)` does not return a finite numeric scalar.

The last check calls `fun` one time. `evaluations` does not count this call.

Control:

`control` goes to the optimizer unchanged. For `"nlm"`, each element of `control` becomes an argument of `stats::nlm()`, such as `iterlim` or `stepmax`. For the other methods, `control` is the `control` argument of `stats::nlminb()` or `stats::optim()`. An unknown name in `control` gives an error for `"nlm"` and a warning for the other methods.

For `"nlm"` with `gradient`, `stats::nlm()` compares `gradient` with a finite-difference gradient at `parameters` before the first iteration. This check calls `fun` and `gradient` one time for each parameter. `evaluations` counts these calls. If the two gradients do not agree, `stats::nlm()` stops with an error. If you know that `gradient` is correct, `control = list(check.analytical = FALSE)` removes the check.

Convergence:

`converged` is `TRUE` when the optimizer reports success and the absolute value of `minimum` is less than `.Machine$double.xmax`. Success is code 1 or 2 for `"nlm"` and code 0 for the other methods. The size test is necessary because `stats::nlm()` replaces `-Inf` with `-.Machine$double.xmax` and can report success at that value. When `converged` is `FALSE`, `optim2()` gives a warning of class `bkmodel_warning_not_converged`.

`converged` shows the result of the stopping rule of the optimizer. It does not prove that `estimate` is a minimum. For example, `"Nelder-Mead"` reports success on the Wood test function at a point far from the minimum.

Hessian:

When `hessian = TRUE`, `stats::optimHess()` computes the Hessian of `fun` at `estimate` by central differences. Without `gradient`, it evaluates `fun` at points up to `2e-3` from `estimate` in each parameter. With `gradient`, it evaluates `gradient` at points up to `1e-3` from `estimate`. These points can be outside a bound. `evaluations` does not count these calls.

Without `gradient`, `stats::optimHess()` calls `fun` $4p^2$ times, where p is `length(parameters)`. With `gradient`, it calls `gradient` $2p$ times and does not call `fun`. For an objective that is slow to evaluate, supply `gradient`.

If `fun` or `gradient` fails or is not finite at one of these points, `hessian` contains `NA`. If the true parameter is on a bound, the estimate is not asymptotically normal (Self and Liang 1987). Standard errors from the inverse Hessian are then not valid. For this reason, `optim2()` also checks whether an element of `estimate` is equal to a bound. Both cases give a warning of class `bkmodel_warning_hessian`, and `optim2()` returns the fit.

Maximization:

To maximize a function, minimize its negative. `control` does not go to `stats::optimHess()`. For example, for `"Nelder-Mead"` and `"L-BFGS-B"`, `control = list(fnscale = -1)` makes `stats::optim()` maximize `fun`. Then `minimum` holds the maximum, and `hessian` is negative definite.

Value

A list with seven elements:

`estimate` `numeric(length(parameters))`, the parameter values at the end of the optimization, with the names of `parameters`.

`minimum` `numeric(1)`, the value of `fun` at `estimate`.

`converged` `logical(1)`, the convergence result. See the section "Convergence" in Details.

`code` `integer(1)`, the convergence code of the optimizer, unchanged.

`message` `character(1)`, the convergence message of the optimizer. For `"nlm"` and `"Nelder-Mead"`, which return no message, `optim2()` gives a fixed text for each code.

`evaluations` `integer(1)`, the number of calls to `fun` by the optimizer, including calls for finite-difference gradients.

`hessian` `NULL` when `hessian = FALSE`. Otherwise a square numeric matrix with `length(parameters)` rows and the names of `parameters` as row and column names. It contains `NA` when the Hessian cannot be computed.

References

Nash JC (2014). "On Best Practice Optimization Methods in R." *Journal of Statistical Software*, **60**(2), 1–14. doi:10.18637/jss.v060.i02.

Self SG, Liang K (1987). "Asymptotic Properties of Maximum Likelihood Estimators and Likelihood Ratio Tests under Nonstandard Conditions." *Journal of the American Statistical Association*, **82**(398), 605–610. doi:10.1080/01621459.1987.10478472.

See Also

`stats::nlminb()`, `stats::nlm()`, `stats::optim()`, `stats::optimHess()`, `stats4::mle()`

Examples

```
#-------------------------------------------------------------------------
# optim2() example
#-------------------------------------------------------------------------
library(bkmodel)

# Normal linear model by maximum likelihood. Fitting log(sigma) keeps sigma
# positive without bounds.
set.seed(1)
x1 <- 1:100
y <- 4 + 2 * x1 + rnorm(100)

nll <- function(par, y, x1) {
  mu <- par[["alpha"]] + par[["beta1"]] * x1
  -sum(dnorm(y, mean = mu, sd = exp(par[["log_sigma"]]), log = TRUE))
}
start <- c(alpha = 0, beta1 = 0, log_sigma = log(sd(y)))

fit <- optim2(nll, start, y = y, x1 = x1, hessian = TRUE)
fit$estimate
fit$converged

# Standard errors from the inverse Hessian. Because lm() divides by n - 2,
# not n, its standard errors are larger by sqrt(n / (n - 2)).
sqrt(diag(solve(fit$hessian)))
```

```
coef(summary(lm(y ~ x1)))[, "Std. Error"]

# Other methods. Nelder-Mead uses no derivatives. nlm needs a smaller step
# limit here because large steps in log(sigma) overflow exp().
optim2(nll, start, y = y, x1 = x1, method = "Nelder-Mead")$estimate
optim2(
  nll,
  start,
  y = y,
  x1 = x1,
  method = "nlm",
  control = list(stepmax = 10)
)$estimate

# Bounds. The unconstrained minimum is at (3, 5), and the bound x >= 4 is
# active.
f <- function(par) (par[["x"]] - 3)^2 + (par[["y"]] - 5)^2
optim2(f, c(x = 6, y = 6), lower = c(4, -Inf))$estimate
optim2(f, c(x = 6, y = 6), method = "L-BFGS-B", lower = c(4, -Inf))$estimate
```

`ordinal_counts` *Convert a weighted categorical response to an ordinal-count matrix*

Description

`ordinal_counts()` converts the factor response and case weights of an inputs payload into a count matrix with one column per response level. The columns use the levels that the schema froze at learn time. Use it when a model or likelihood takes a count matrix instead of a factor response plus case weights.

The function accepts an inputs payload from `design_learn()`, `design_apply()`, or a fold part from `design_fold()`. `ordinal_matrix_to_factor()` converts the count matrix back to a factor response and row weights.

Usage

```
ordinal_counts(inputs, schema, warn = TRUE)
```

Arguments

`inputs`	(`design_inputs`) Inputs payload from `design_learn()`, `design_apply()`, or a fold part from `design_fold()`.
`schema`	(`design_schema`) Schema that supplies the frozen response levels and the weights role. Use the schema that produced `inputs`. For `design_apply()`, that is the schema passed to it.
`warn`	(`logical(1)`; default `TRUE`) Whether to warn on frozen levels with zero counts.

Details

Ordinal prerequisites:

The transformation requires:

- A schema learned on a factor or ordered-factor response with at least two frozen levels. A transformed response such as `factor(cyl)` is valid.
- A payload response `inputs$y` that is a factor with levels identical to the frozen levels. Every payload from `design_learn()`, `design_apply()`, and `design_fold()` meets this requirement.
- Non-missing `y` on the payload rows.
- Finite, non-negative `weights`, one per payload row, when the schema has a weights role.

A payload that violates a prerequisite stops with an error. When the schema has no weights role, each row has a count of one. Unit counts give the standard unweighted likelihood. When the schema has a weights role and `inputs$weights` is `NULL`, the function stops with an error. A call to `design_apply(weights = "ignore")` gives such a payload. To count each row once, set `inputs$weights` to `rep(1, length(inputs$y))`.

For a plain (non-ordered) factor, the frozen level order is the order that `levels()` returned at learn time. For a factor made from a character vector, that order is the sorted order by default. Set the level order explicitly, or use an `ordered()` factor, when the order matters.

A payload has missing responses only under `na.pass`. A spec with `na_action = "na.pass"` keeps such rows in the learn payload and in every fold part. A call to `design_apply(na_action = "na.pass")` keeps them in its payload. Subset such a payload to its complete rows before you transform it.

Count construction:

Each payload row puts its full observation weight in the column of its observed response level and zero in every other column. Frozen but unobserved levels stay as zero-count columns. The matrix always has one column per frozen level, in frozen-level order. A row with weight zero gives an all-zero row. Some fitters reject all-zero rows. For example, `nnet::multinom()` stops with an error. For such a fitter, remove the zero-weight rows first.

Zero-count warning:

When a frozen level has zero total counts, `ordinal_counts()` warns with condition class `bkmodel_warning_zero_counts`. A model fit to such counts is degenerate. In a cumulative-link model, an empty first or last level sends a threshold estimate to an infinite value. An empty interior level makes two threshold estimates equal. Rare levels can give empty levels in resampled analysis sets. To reduce that chance, use `strata = ~ 1` at plan construction, or merge rare levels before `design_spec()`.

Keep `warn = TRUE` for a payload that you fit a model to, such as an analysis part. Set `warn = FALSE` for a payload that you only score, such as an assessment or out-of-bag part. An empty level in a scored payload has no effect on the fit. Also set `warn = FALSE` when your code removes the zero-count columns before the fit and reports them.

Value

A `design_inputs` object with the elements below.

`x` Passed through from `inputs`.

`y` Numeric count matrix, one row per payload row and one column per frozen response level, columns named by the frozen levels.

`weights` `NULL`, because the weights are in the counts.

`offset` Passed through from `inputs`.

`case_id` Passed through from `inputs`.

`row_index` Passed through from `inputs`.

See Also

`ordinal_matrix_to_factor()`, `design_learn()`, `design_apply()`, `design_fold()`

Examples

```
#--------------------------------------------------------------------------
# Transform a learn payload
#--------------------------------------------------------------------------
library(bkmodel)

analysis <- data.frame(
  response = ordered(
    c("low", "medium", "high", "low", "medium", "high"),
    levels = c("low", "medium", "high")
  ),
  marker = c(0.2, 0.4, 1.1, 0.3, 0.8, 1.4),
  sampling_weight = c(1.0, 0.5, 1.5, 1.2, 0.8, 1.1)
)

spec <- design_spec(
  response ~ marker,
  weights = ~sampling_weight
)

lr <- design_learn(spec, analysis)
counts <- ordinal_counts(lr$specs[[1L]]$inputs, lr$specs[[1L]]$schema)

counts$y
rowSums(counts$y)
counts$weights

#--------------------------------------------------------------------------
# Round trip back to a factor response and weights
#--------------------------------------------------------------------------
back <- ordinal_matrix_to_factor(counts$y)
back$y
back$weights

#--------------------------------------------------------------------------
# Zero-count frozen levels warn by condition class
#--------------------------------------------------------------------------
analysis$response <- ordered(
  as.character(analysis$response),
```

```
  levels = c("low", "medium", "high", "extreme")
)

lr2 <- design_learn(spec, analysis)

# 'extreme' is frozen but unobserved, so its column has zero counts.
counts2 <- withCallingHandlers(
  ordinal_counts(lr2$specs[[1L]]$inputs, lr2$specs[[1L]]$schema),
  bkmodel_warning_zero_counts = function(w) invokeRestart("muffleWarning")
)
colSums(counts2$y)

# An assessment part is only scored. Its empty levels do not matter.
plan <- design_cv(spec, analysis, n_folds = 2L, seed = 1L)
part <- design_fold(plan, 1L)$specs[[1L]]
scored <- ordinal_counts(part$assessment, part$schema, warn = FALSE)
scored$y
```

`ordinal_matrix_to_factor` *Convert an ordinal-count matrix to a factor response and row weights*

Description

`ordinal_matrix_to_factor()` converts a weighted ordinal-count response matrix into a factor response and row weights. Use it when a model, metric, or diagnostic needs a factor response plus case weights instead of a count matrix.

The function undoes `ordinal_counts()` on its `y` matrix, with three losses:

- The factor is a plain factor. Use `as.ordered()` to restore an ordered response.
- A zero-weight row returns `NA`, because an all-zero row has no level.
- Unit weights return as a vector of ones, not as `NULL`.

Usage

```
ordinal_matrix_to_factor(y)
```

Arguments

`y`	(numeric matrix) Ordinal-count response matrix. Rows are observations and columns are ordinal response levels.

Details

The matrix column names define the factor levels, in column order. Column names must be non-missing, non-empty, and unique.

Values must be non-missing, finite, and non-negative. Each row must have zero positive cells or exactly one positive cell. A row with one positive cell returns that column as its level and the cell value as its weight. A row with all zeros returns `NA` and weight `0`. A row with more than one positive cell stops with an error. A matrix with zero rows returns a zero-length factor and zero-length weights.

The function does not require integer counts. It accepts fractional values because `ordinal_counts()` puts case weights in the cells.

Value

A plain list with the elements below, aligned row for row with `y`.

`y` Factor with `levels = colnames(y)`, `NA` on all-zero rows.

`weights` Unnamed numeric vector equal to `rowSums(y)`.

See Also

`ordinal_counts()`

Examples

```
#-------------------------------------------------------------------------
# Convert a weighted ordinal-count matrix
#-------------------------------------------------------------------------
library(bkmodel)

counts <- matrix(
  c(0, 1.2, 0,
    0.8, 0, 0,
    0, 0, 1.5,
    0, 0, 0),
  ncol = 3L,
  byrow = TRUE,
  dimnames = list(NULL, c("low", "medium", "high"))
)

converted <- ordinal_matrix_to_factor(counts)
converted$y
converted$weights

#-------------------------------------------------------------------------
# Round-trip from ordinal_counts()
#-------------------------------------------------------------------------
analysis <- data.frame(
  response = ordered(
    c("low", "medium", "high", "low", "medium", "high"),
    levels = c("low", "medium", "high")
```

```
  ),
  marker = c(0.2, 0.4, 1.1, 0.3, 0.8, 1.4),
  sampling_weight = c(1.0, 0.5, 1.5, 1.2, 0.8, 1.1)
)

spec <- design_spec(
  response ~ marker,
  weights = ~ sampling_weight
)

lr <- design_learn(spec, analysis)
ordinal_inputs <- ordinal_counts(
  lr$specs[[1L]]$inputs,
  lr$specs[[1L]]$schema
)

factor_inputs <- ordinal_matrix_to_factor(ordinal_inputs$y)
factor_inputs$weights

# The factor is plain. Restore the ordered class explicitly.
as.ordered(factor_inputs$y)
```

`parse_formula` *Split a formula into response, group, and block terms*

Description

`parse_formula()` splits a formula of the form `y ~ x | z` into the response `y`, the group term `x`, and the block term `z`. `get_ind_data()`, `get_dep_data()`, and `tall_to_wide()` use it to find the column that each term names.

Usage

```
parse_formula(formula)
```

Arguments

`formula`	(formula) Formula to split, of the form `y ~ x`, `y ~ x | z`, `~x`, or `~x | z`.

Details

The function uses only the syntax of `formula`. It does not evaluate the terms and does not expand `.`. To expand `.`, call `stats::terms.formula()` with a `data` argument first.

Only a top-level `|` splits the right-hand side:

- `y ~ x | z | w` gives the group term `x | z` and the block term `w`, because `|` groups from the left.
- `y ~ (x | z)`, `y ~ x + (a | b)`, and `y ~ I(x | z)` do not split.

A subclass of "formula", such as a terms object, gives the same lhs, rhs_group, and rhs_block as the plain formula. A NULL element means that the part is absent. For this reason, the function stops with an error when the response or the block term is the literal NULL, as in NULL ~ x or y ~ x | NULL. The function stops with an error when the ~ call of formula does not have one or two arguments.

Value

A list of length 4 with the elements below. The elements lhs, rhs_group, and rhs_block are symbols, calls, constants such as 1, or NULL. These three elements are not formulas. To evaluate one, use the environment of the element formula.

formula The input formula, with its class and environment unchanged.

lhs The response, or NULL when formula has no left-hand side.

rhs_group The term left of a top-level |, or the whole right-hand side when there is no top-level |.

rhs_block The term right of a top-level |, or NULL when there is no top-level |.

See Also

get_ind_data(), get_dep_data(), tall_to_wide(), stats::terms.formula()

Examples

```
#-------------------------------------------------------------------------
# parse_formula() example
#-------------------------------------------------------------------------
library(bkmodel)

# Response, group term, and block term
parse_formula(y ~ x | z)

# One-sided formula without a block term
parse_formula(~x)

# A `|` inside parentheses does not split the right-hand side
parse_formula(y ~ x + (a | b))
```

---

print.design_plan *Print methods for design objects*

Description

Compact summaries for the five design classes. Each method replaces the default list printing, which dumps stored data, seed tables, and encoded matrices at full length.

Usage

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

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

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

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

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

Arguments

x	(design_plan, design_fold, design_schema, design_inputs, or design_report) Object to print.
...	Ignored. Present for compatibility with the print() generic.

Details

Each method prints a header line with the class name, then one aligned line per field. Specs, roles, inner operands, and novel levels continue on more lines under the value column:

- design_plan: split count and shape, universe size against the data, spec formulas, strata and group when set, the inner descriptor when set, and the plan seed. Plans built by design_splits() label their split count supplied. An inner plan under a bootstrap outer resamples draw positions, so its universe line gives the position count and the unique row count.
- design_fold: the fold seed, scoreable row counts per materialized part, the inner design when present, and per spec the learned formula with part dimensions. The header gives the split id: rep<r>.fold<k> for CV plans, rep<r> for bootstrap plans, and split<i> for plans built by design_splits(). The inner plan partitions the analysis rows, so its size is the analysis row count. The learned formula omits terms dropped as degenerate on the fold's analysis rows.
- design_schema: response with level count for factor responses, predictor types with level counts, encoding and encoded column count, non-NULL roles one per line, and the NA and novel-level policies.
- design_inputs: class and shape of x, y, weights, offset, and case_id, and the head of row_index.
- design_report: row count and one line per attribution channel (missing, novel, nonfinite), with the novel levels per predictor and the non-finite columns.

A missing response, an empty predictor or row_index list, a NULL element of design_inputs, and an unset inner operand print as none. Unset strata, group, and roles are left out.

Item lists and formulas are cut to fit getOption("width"). A cut list ends with the total item count. A list always shows its first item whole, so a very long name can make a line wider than the console.

A cut formula ends at a space with `...`, so a term that contains a space, such as `log(w + 1)`, can be cut. Outcome strata (`~ 1`) print as `~1 (outcome)`.

All methods return their input invisibly and print nothing recursive, so large stored objects (`plan$data`, `plan$seeds`, `inputs$x`) never reach the console.

Value

`x`, invisibly.

See Also

`design_cv()`, `design_bootstrap()`, `design_splits()`, `design_fold()`, `design_learn()`, `design_apply()`

Examples

```
#--------------------------------------------------------------------------
# Print a plan, a fold, and the learn-side objects
#--------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  age = round(rnorm(24, mean = 60, sd = 8)),
  treatment = factor(rep(c("control", "active"), 12)),
  outcome = rnorm(24)
)

spec <- design_spec(
  outcome ~ age + treatment,
  encoding = "dummy",
  intercept = TRUE
)

plan <- design_cv(spec, analysis, n_folds = 4L, seed = 42L)
plan

fold <- design_fold(plan, 1L)
fold

lr <- design_learn(spec, analysis)
lr$specs[[1L]]$schema
lr$specs[[1L]]$inputs
lr$specs[[1L]]$report
```

`tall_to_wide` *Reshape data from tall to wide*

Description

Reshape a tall data frame to a wide layout following `y ~ x | z`. Optionally aggregates rows that share a `(z, x)` cell.

Usage

```
tall_to_wide(data, formula, agg_fun = "error", drop_unused_levels = FALSE)

ftall_to_wide(data, y, x, z, agg_fun = "error", drop_unused_levels = FALSE)
```

Arguments

`data`	(data.frame) The tall data frame to convert to wide format.
`formula`	(formula) The formula that names the outcome (`y`), group (`x`), and block (`z`) columns. The formula must have the form `y ~ x | z`, with a bare column name for each term.
`agg_fun`	(`character(1)` or function; default `"error"`) How to compute each `(z, x)` cell. One of `"error"`, `"first"`, `"last"`, `"sum"`, `"mean"`, `"median"`, `"min"`, `"max"`, or a function. `"error"` raises an error on cells with two or more rows. The other character values act only on such cells. A function is applied to every non-empty cell.
`drop_unused_levels`	(`logical(1)`; default `FALSE`) Drop levels of `x` and `z` that have no observations after the `NA` filter. If `FALSE`, they appear as all-`NA` columns or rows.
`y`	(`character(1)`) The name of the numeric outcome column in `data`.
`x`	(`character(1)`) The name of the grouping column in `data`.
`z`	(`character(1)`) The name of the blocking column in `data`.

Details

Inputs are interpreted as:

- `y` is a numeric vector for the outcome.
- `x` is a factor representing the grouping variable.
- `z` is a factor representing the blocking variable (subject or item index).

Pre-processing rules:

- If `x` or `z` is not a factor, it is converted with `base::as.factor()`.
- Factor levels with no observations appear as all-`NA` columns (`x`) or rows (`z`) in the output. Set `drop_unused_levels = TRUE` to remove them.

Constraints:

- y, x, and z must name different columns.
- The name z must not equal a level of x, because both become output column names.
- A level of x must not be "" or NA (from base::addNA()), because it cannot name a column.
- nlevels(x) * nlevels(z) must not exceed .Machine$integer.max.

Cell aggregation:

- A cell with no observations becomes NA_real_.
- A cell with two or more observations raises an error when agg_fun = "error" (the default).
- Other character values of agg_fun act only on cells with two or more observations. A cell with one observation keeps its value, which may be NA.
- A function agg_fun is called once for every non-empty cell, including cells with one observation. It receives the y values of the cell as a double vector, NA included. It must return one numeric or logical value.

NA handling:

- Rows with NA in x or z are dropped before aggregation. A level from base::addNA() is not a missing value, so in z it becomes an output row.
- Rows with NA in y are kept and reach the aggregator.
- Built-in aggregators use na.rm = TRUE and return NA_real_ when every value in a cell is NA.
- "first" and "last" return the first or last non-NA value in row order.
- Any NaN in the output becomes NA_real_.

The operation is similar to:

```
reshape(
  data = data,
  direction = "wide",
  idvar = "z",
  timevar = "x",
  v.names = "y"
)
```

Value

A data.frame with a leading z column, then one column per level of x, and one row per level of z. Levels removed by drop_unused_levels are not counted.

- The first column is a factor named after z, holding the levels of z in levels(z) order. It is ordered if z is ordered.
- The remaining columns are numeric, one per level of x in levels(x) order, each named by its level.
- Each numeric cell holds the y value, aggregated if needed, for its (z, x) combination, or NA_real_ if the cell is empty.

See Also

stats::reshape(), get_dep_data()

Examples

```
#-------------------------------------------------------------------------
# Basic reshape: one observation per (z, x) cell
#-------------------------------------------------------------------------
library(bkmodel)

tall <- data.frame(
  y = c(1, 2, 3, 4),
  x = factor(c("A", "B", "A", "B")),
  z = factor(c("1", "1", "2", "2"))
)
tall

# Wide layout: one row per level of z, one column per level of x
tall_to_wide(tall, y ~ x | z)

# Equivalent (fast) character interface
ftall_to_wide(tall, y = "y", x = "x", z = "z")

# Similar base R call (columns are named `y.A`, `y.B`)
reshape(
  data = tall,
  direction = "wide",
  idvar = "z",
  timevar = "x",
  v.names = "y"
)

#-------------------------------------------------------------------------
# Aggregating duplicate (z, x) cells
#-------------------------------------------------------------------------
dup <- data.frame(
  y = c(10, 20, 3, 4),
  x = factor(c("A", "A", "A", "B")),
  z = factor(c("1", "1", "2", "2"))
)

# Built-in aggregator
tall_to_wide(dup, y ~ x | z, agg_fun = "mean")

# User-supplied aggregator, called once per non-empty cell
tall_to_wide(dup, y ~ x | z, agg_fun = function(v) max(abs(v)))
```

Index

as.ordered(), *47*

base::addNA(), *36*, *38*, *54*
base::as.factor(), *53*
base::scale(), *36*, *38*
base::set.seed(), *9*, *13*, *14*

design_apply, 2
design_apply(), *17*, *19*, *24*, *25*, *27–29*, *44–46*, *52*
design_bootstrap, 6
design_bootstrap(), *14*, *16*, *19*, *21*, *22*, *26*, *29*, *31–33*, *52*
design_cv, 11
design_cv(), *7–9*, *16*, *19*, *21*, *22*, *26*, *29*, *31–33*, *52*
design_fold, 15
design_fold(), *4*, *6*, *8*, *9*, *11*, *13*, *14*, *21*, *25*, *28–30*, *33*, *44–46*, *52*
design_inner_cv, 20
design_inner_cv(), *7*, *9*, *12*, *14*, *19*, *31*, *33*
design_learn, 23
design_learn(), *2*, *4*, *7*, *9*, *12*, *14*, *19*, *26–29*, *31*, *33*, *44–46*, *52*
design_spec, 26
design_spec(), *3*, *4*, *6*, *7*, *9*, *11*, *14*, *17*, *23–25*, *31*, *33*, *45*
design_splits, 30
design_splits(), *16*, *18*, *19*, *21*, *22*, *51*, *52*

ftall_to_wide *(*tall_to_wide*)*, 52

get_dep_data, 34
get_dep_data(), *39*, *49*, *50*, *55*
get_ind_data, 37
get_ind_data(), *36*, *49*, *50*

optim2, 40
ordinal_counts, 44
ordinal_counts(), *47*, *48*
ordinal_matrix_to_factor, 47
ordinal_matrix_to_factor(), *19*, *44*, *46*

parse_formula, 49
parse_formula(), *36*, *39*
print(), *51*
print.design_fold *(*print.design_plan*)*, 50
print.design_inputs *(*print.design_plan*)*, 50
print.design_plan, 50
print.design_report *(*print.design_plan*)*, 50
print.design_schema *(*print.design_plan*)*, 50

stats4::mle(), *43*
stats::makepredictcall(), *17*
stats::nlm(), *41–43*
stats::nlminb(), *41–43*
stats::optim(), *41–43*
stats::optimHess(), *42*, *43*
stats::reshape(), *55*
stats::t.test(), *34*, *35*, *37*, *38*
stats::terms.formula(), *49*, *50*

tall_to_wide, 52
tall_to_wide(), *36*, *49*, *50*