Package: bkmodel (via r-universe)

June 4, 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

Suggests tinytest, rmarkdown

RdMacros Rdpack

Language en-US

RoxygenNote 7.3.3

Roxygen list(markdown = TRUE)

Encoding UTF-8

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

Date/Publication 2026-06-04 03:47:02 UTC

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

RemoteRef HEAD

RemoteSha e2d22a4ca02f782f8cd09850014b1955293ef141

Contents

`design_bootstrap` *Create a design-aware bootstrap plan*

Description

`design_bootstrap()` creates a lazy design-aware resampling object for bootstrap validation. It computes the retained row universe from a `design_spec()` object and creates bootstrap split indices on those rows. It does not fit a statistical model and does not build a full-data design matrix.

Use `design_fold()` to materialize one bootstrap split. Each materialized split fits its design schema on the bootstrap analysis sample only and then encodes the full retained-row assessment set and out-of-bag rows with that analysis schema.

Usage

```
design_bootstrap(
  spec,
  data,
  n_boot = 1000L,
  strata = NULL,
  group = NULL,
  response_type = c("response", "ordinal_counts")
)
```

Arguments

`spec`	(design_spec) Unfitted design specification returned by `design_spec()`.
`data`	(data.frame) Analysis data used to create bootstrap samples. Objects that inherit from `data.frame`, including tibbles, are accepted.
`n_boot`	(numeric scalar: `1000L`; whole number in `[1, .Machine$integer.max]`) Number of bootstrap replicates. Whole-number numeric values such as `100` and `100.0` are accepted and stored as integers.

`strata` (formula: `NULL`)
Stratification specification. Use `NULL` for no stratification. Use `~ 1` to stratify by the raw response. Use a one-sided formula such as `~ treatment` or `~ site + treatment` to stratify by observed columns.

`group` (formula: `NULL`)
Grouping specification. Use `NULL` for row-level bootstrap sampling. Use a one-sided formula such as `~ patient_id` to draw whole groups with replacement.

`response_type` (Scalar character: `"response"`)
Response output mode used by `design_fold()`. Must be one of `"response"` or `"ordinal_counts"`.

Details

Workflow:

A typical bootstrap validation workflow starts with `design_spec()` to declare the design rules. Call `design_bootstrap()` after setting the random-number seed to create reproducible bootstrap samples. Then iterate over `seq_along(bs$splits)`, materialize each split with `design_fold()`, fit the model on `fold$analysis`, and evaluate predictions on `fold$assessment`, `fold$oob`, or both.

The full retained-row `assessment` input is useful for bootstrap optimism correction. The `oob` input is useful for out-of-bag error estimation.

Keeping split construction separate from fold materialization prevents assessment and out-of-bag rows from determining factor levels, dropped categorical terms, column order, or other fitted design-schema details.

Retained rows:

The retained row universe is computed once before bootstrap samples are drawn. The `na_action` value stored in `spec` controls whether rows with missing predictor or response values are kept, removed, or rejected. Weights and offsets are checked on retained rows using the same rules as `design_fit()`.

Each split stores row identities from the retained universe. When a split is materialized, `row_index` values in `analysis`, `assessment`, and `oob` refer to row numbers in the original `data`. Assessment and OOB row identities are preserved after split construction so prediction errors can be aligned back to the original data. Assessment and OOB predictors and responses are still encoded with the bootstrap-analysis schema.

Splits:

`n_boot` controls the number of bootstrap replicates. Each split contains:

- `analysis`: retained row identities drawn with replacement. Duplicate row identities are expected and their draw order is preserved.
- `assessment`: the full retained-row universe in retained-row order.
- `oob`: retained row identities not present in `unique(analysis)`. The OOB set can be empty for a bootstrap split.
- `seed`: single positive integer for reproducible model fits on this split.

Bootstrap assignment is over retained rows when `group` is `NULL` and over retained groups when `group` is supplied. When `group` is supplied, all rows from a drawn group enter `analysis` together. If retained group sizes differ, the length of `analysis` can vary across bootstrap splits. Each

bootstrap split advances the current random-number generator state. Call `base::set.seed()` immediately before `design_bootstrap()` for reproducible split assignment.

Each split carries a `seed`, a single positive integer for reproducible model fits on that split. `design_fold()` copies this seed onto the materialized fold. See `resample_bootstrap()` for the seed recipe and its rationale.

Operational variables:

`group` and `strata` are optional operational controls for drawing bootstrap samples. They are evaluated on the retained rows after the design specification computes the resampling row universe.

If the design formula uses `.`, variables used only for `group` or `strata` must not enter the predictor set through `.` expansion. Put those variables in `case_id`, `weights`, `offset`, or `exclude`, or write an explicit predictor formula. Operational variables may also be named explicitly as predictors when that is the intended statistical design.

Stratification:

Use `strata = NULL` for unstratified bootstrap sampling. Use a one-sided formula such as `strata = ~ treatment` to draw samples within observed covariate levels. Use `strata = ~ 1` to stratify by the raw response.

Response stratification evaluates the response before any `response_type` conversion. Each bootstrap stratum must contain at least two sampling units. See `resample_bootstrap()` for additional stratification and grouping details.

`response_type`**:**

`response_type = "response"` returns the ordinary evaluated response in each materialized split. `response_type = "ordinal_counts"` returns weighted ordinal-count response matrices in `analysis$y`, `assessment$y`, and `oob$y`. Ordinal-count splits require a two-sided bare-symbol factor or ordered-factor response with non-missing retained responses. They also require weights when splits are materialized because the response matrix stores weighted counts.

The response type is fixed when the bootstrap object is created. Every materialized split uses the same response representation for analysis, assessment, and OOB inputs. Create a separate design bootstrap object when a different response representation is needed.

Value

A list of class `c("design_bootstrap_resamples", "design_resamples")` with elements:

- `spec`: the original `design_spec()` object.
- `data`: the original analysis data.
- `row_index`: retained row identities used for bootstrap construction.
- `response_type`: response representation used by materialized splits.
- `response_levels`: declared response levels for ordinal-count output, or `NULL`.
- `resamples`: the underlying index-only `bootstrap_resamples` object.
- `splits`: a named list of bootstrap split indices.

Each split contains `analysis`, `assessment`, `oob`, and `seed`. Use `design_fold()` to convert a split into model-ready inputs.

See Also

design_spec(), design_fold(), design_fit(), design_data(), design_cv(), resample_bootstrap(), base::set.seed()

Examples

```
#--------------------------------------------------------------------------
# Estimate optimism-corrected prediction error
#--------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  patient_id = sprintf("P%02d", 1:36),
  age = round(rnorm(36, mean = 60, sd = 8)),
  treatment = factor(rep(c("control", "active"), 18)),
  stage = factor(rep(c("I", "II", "III"), each = 12)),
  exposure = runif(36, min = 0.5, max = 2),
  inverse_prob_weight = runif(36, min = 0.8, max = 1.2),
  chart_score = rnorm(36)
)
analysis$risk_group <- interaction(
  analysis$treatment,
  analysis$stage,
  drop = TRUE
)

analysis$outcome <- with(
  analysis,
  5 + 0.25 * age + 1.5 * (treatment == "active") +
    0.8 * (stage == "III") + log(exposure) + rnorm(36)
)

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

weighted_rmse <- function(observed, predicted, weights) {
  sqrt(stats::weighted.mean((observed - predicted)^2, weights))
}

fit_lm <- function(inputs) {
  stats::lm.wfit(
    x = inputs$x,
```

```
    y = inputs$y,
    w = inputs$weights,
    offset = inputs$offset
  )
}

predict_lm <- function(fit, inputs) {
  predicted <- drop(inputs$x %*% fit$coefficients)
  predicted + inputs$offset
}

full_fit <- design_fit(spec, analysis)
full_inputs <- design_data(full_fit)
full_model <- fit_lm(full_inputs)
apparent_rmse <- weighted_rmse(
  full_inputs$y,
  predict_lm(full_model, full_inputs),
  full_inputs$weights
)

set.seed(42L)
bs <- design_bootstrap(
  spec,
  analysis,
  n_boot = 25L,
  strata = ~ risk_group
)

optimism <- vapply(seq_along(bs$splits), function(i) {
  fold <- design_fold(bs, i)
  model <- fit_lm(fold$analysis)

  bootstrap_error <- weighted_rmse(
    fold$analysis$y,
    predict_lm(model, fold$analysis),
    fold$analysis$weights
  )

  original_error <- weighted_rmse(
    fold$assessment$y,
    predict_lm(model, fold$assessment),
    fold$assessment$weights
  )

  original_error - bootstrap_error
}, numeric(1))

apparent_rmse
apparent_rmse + mean(optimism)

#--------------------------------------------------------------------------
# Estimate out-of-bag prediction error
#--------------------------------------------------------------------------
```

```
oob_rmse <- vapply(seq_along(bs$splits), function(i) {
  fold <- design_fold(bs, i)
  model <- fit_lm(fold$analysis)

  weighted_rmse(
    fold$oob$y,
    predict_lm(model, fold$oob),
    fold$oob$weights
  )
}, numeric(1))

mean(oob_rmse)

#--------------------------------------------------------------------------
# Draw repeated patient observations as groups
#--------------------------------------------------------------------------
repeated <- data.frame(
  patient_id = rep(sprintf("P%02d", 1:18), each = 2),
  visit = rep(c("baseline", "follow_up"), times = 18),
  treatment = factor(rep(rep(c("control", "active"), each = 9), each = 2)),
  age = rep(round(rnorm(18, mean = 60, sd = 8)), each = 2),
  outcome = rnorm(36)
)

repeated_spec <- design_spec(
  outcome ~ age + treatment + visit,
  case_id = ~ patient_id,
  encoding = "dummy",
  intercept = TRUE
)

set.seed(7L)
grouped_bs <- design_bootstrap(
  repeated_spec,
  repeated,
  n_boot = 10L,
  group = ~ patient_id,
  strata = ~ treatment
)

grouped_fold <- design_fold(grouped_bs, 1L)
grouped_fold$analysis$case_id[duplicated(grouped_fold$analysis$case_id)]
```

`design_cv` *Create a design-aware K-fold cross-validation plan*

Description

`design_cv()` creates a lazy design-aware resampling object for K-fold cross-validation. It computes the retained row universe from a `design_spec()` object and creates K-fold split indices on

those rows. It does not fit a statistical model and does not build a full-data design matrix.

Use design_fold() to materialize one split. Each materialized fold fits its design schema on the analysis rows only and then encodes the assessment rows with that analysis schema.

Usage

```
design_cv(
  spec,
  data,
  n_folds = 5L,
  n_iters = 1L,
  strata = NULL,
  group = NULL,
  response_type = c("response", "ordinal_counts")
)
```

Arguments

spec	(design_spec) Unfitted design specification returned by design_spec().
data	(data.frame) Analysis data used to create folds. Objects that inherit from data.frame, including tibbles, are accepted.
n_folds	(numeric scalar: 5L; whole number in [2, unit_n]) Number of folds. Whole-number numeric values such as 5 and 5.0 are accepted and stored as integers. The value must be at least 2 and no larger than the number of sampling units.
n_iters	(numeric scalar: 1L; whole number in [1, .Machine$integer.max]) Number of independent K-fold partitions. Use values greater than one for repeated K-fold cross-validation.
strata	(formula: NULL) Stratification specification. Use NULL for no stratification. Use ~ 1 to stratify by the raw response. Use a one-sided formula such as ~ treatment or ~ site + treatment to stratify by observed columns.
group	(formula: NULL) Grouping specification. Use NULL for row-level resampling. Use a one-sided formula such as ~ patient_id to keep rows with the same group value together.
response_type	(Scalar character: "response") Response output mode used by design_fold(). Must be one of "response" or "ordinal_counts".

Details

Workflow:

A typical validation workflow starts with design_spec() to declare the design rules. Call design_cv() after setting the random-number seed to create reproducible folds. Then iterate over seq_along(cv$splits),

materialize each split with design_fold(), fit the model on fold$analysis, and evaluate predictions on fold$assessment.

Keeping split construction separate from fold materialization prevents assessment rows from determining factor levels, dropped categorical terms, column order, or other fitted design-schema details.

Retained rows:

The retained row universe is computed once before folds are assigned. The na_action value stored in spec controls whether rows with missing predictor or response values are kept, removed, or rejected. Weights and offsets are checked on retained rows using the same rules as design_fit().

Each split stores row identities from the retained universe. When a fold is materialized, fold$analysis$row_index and fold$assessment$row_index refer to row numbers in the original data. Assessment row identities are preserved after split construction so prediction errors can be aligned back to the original data. Assessment predictors and responses are still encoded with the fold-specific analysis schema.

Splits:

n_folds controls the number of folds in each K-fold partition. n_iters controls the number of independent K-fold partitions. The total number of splits is n_folds * n_iters.

Within each iteration, every retained row appears in exactly one assessment set. Across different iterations, the same row can appear in different assessment folds. Call base::set.seed() immediately before design_cv() for reproducible split assignment.

Each split carries a seed, a single positive integer for a reproducible model fit on that split. design_fold() copies this seed onto the materialized fold. See resample_cv() for the seeding recipe and its rationale.

Operational variables:

group and strata are optional operational controls for assigning folds. They are evaluated on the retained rows after the design specification computes the resampling row universe.

If the design formula uses ., variables used only for group or strata must not enter the predictor set through . expansion. Put those variables in case_id, weights, offset, or exclude, or write an explicit predictor formula. Operational variables may also be named explicitly as predictors when that is the intended statistical design.

Response stratification:

Use strata = NULL for unstratified folds. Use a one-sided formula such as strata = ~ treatment to balance folds by observed covariate levels. Use strata = ~ 1 to stratify by the raw response.

Response stratification evaluates the response before any response_type conversion. For ordinal-count workflows with rare levels, strata = ~ 1 can reduce the chance that an analysis fold has an all-zero declared-level response column. See resample_cv() for additional stratification and grouping details.

response_type**:**

response_type = "response" returns the ordinary evaluated response in each materialized fold. response_type = "ordinal_counts" returns weighted ordinal-count response matrices in fold$analysis$y and fold$assessment$y. Ordinal-count folds require a two-sided bare-symbol factor or ordered-factor response with non-missing retained responses. They also require evaluated weights because the response matrix stores weighted counts.

The response type is fixed when the CV object is created. Every materialized fold uses the same response representation for analysis and assessment inputs. Create a separate design CV object when a different response representation is needed.

Value

A list of class `c("design_cv_resamples", "design_resamples")` with elements:

- `spec`: the original `design_spec()` object.
- `data`: the original analysis data.
- `row_index`: retained row identities used for split construction.
- `response_type`: response representation used by materialized folds.
- `response_levels`: declared response levels for ordinal-count output, or `NULL`.
- `resamples`: the underlying index-only `cv_resamples` object.
- `splits`: a named list of CV split indices.

Each split contains `analysis`, `assessment`, `iter`, `fold`, and `seed`. Use `design_fold()` to convert a split into model-ready inputs.

See Also

`design_spec()`, `design_fold()`, `design_fit()`, `design_data()`, `resample_cv()`, `base::set.seed()`

Examples

```
#------------------------------------------------------------------------------
# Estimate prediction error with stratified K-fold cross-validation
#------------------------------------------------------------------------------
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)),
  stage = factor(rep(c("I", "II", "III"), each = 8)),
  exposure = runif(24, min = 0.5, max = 2),
  inverse_prob_weight = runif(24, min = 0.8, max = 1.2),
  chart_score = rnorm(24)
)
analysis$risk_group <- interaction(
  analysis$treatment,
  analysis$stage,
  drop = TRUE
)

analysis$outcome <- with(
  analysis,
  5 + 0.25 * age + 1.5 * (treatment == "active") +
    0.8 * (stage == "III") + log(exposure) + rnorm(24)
```

```
)

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

set.seed(42L)
cv <- design_cv(
  spec,
  analysis,
  n_folds = 4L,
  strata = ~ risk_group
)

fold_rmse <- vapply(seq_along(cv$splits), function(i) {
  fold <- design_fold(cv, i)

  fit <- stats::lm.wfit(
    x = fold$analysis$x,
    y = fold$analysis$y,
    w = fold$analysis$weights,
    offset = fold$analysis$offset
  )

  predicted <- drop(fold$assessment$x %*% fit$coefficients)
  predicted <- predicted + fold$assessment$offset
  sqrt(mean((fold$assessment$y - predicted)^2))
}, numeric(1))

fold_rmse
mean(fold_rmse)

#-------------------------------------------------------------------------
# Keep repeated patient observations together
#-------------------------------------------------------------------------
repeated <- data.frame(
  patient_id = rep(sprintf("P%02d", 1:12), each = 2),
  visit = rep(c("baseline", "follow_up"), times = 12),
  treatment = factor(rep(rep(c("control", "active"), each = 6), each = 2)),
  age = rep(round(rnorm(12, mean = 60, sd = 8)), each = 2),
  outcome = rnorm(24)
)

repeated_spec <- design_spec(
  outcome ~ age + treatment + visit,
```

```
  case_id = ~ patient_id,
  encoding = "dummy",
  intercept = TRUE
)

set.seed(7L)
grouped_cv <- design_cv(
  repeated_spec,
  repeated,
  n_folds = 4L,
  group = ~ patient_id,
  strata = ~ treatment
)

grouped_fold <- design_fold(grouped_cv, 1L)
intersect(grouped_fold$analysis$case_id, grouped_fold$assessment$case_id)
```

`design_data` *Extract model-ready inputs from a fitted design schema*

Description

`design_data()` extracts the retained training inputs from a fitted `design_fit()` object. It reuses cached `x`, `y`, `weights`, `offset`, `case_id`, and `row_index`. It does not rebuild the design matrix and does not re-evaluate any formula.

Use this function after `design_fit()` when passing analysis data to a model fitter, scoring function, or diagnostic routine. Use `design_newdata()` when encoding validation or prediction data.

Usage

```
design_data(fit, response_type = c("response", "ordinal_counts"))
```

Arguments

`fit`	(design_fit) Fitted design schema returned by `design_fit()`.
`response_type`	(Scalar character: `"response"`) Response output mode. Must be one of `"response"` or `"ordinal_counts"`.

Details

Standard response output:

`response_type = "response"` returns the response and row roles as they were evaluated and retained by `design_fit()`. The returned `x` matrix already has the fitted predictor columns, categorical encoding, row filtering, and column order. The returned `y`, `weights`, `offset`, `case_id`, and `row_index` are aligned row-for-row with `x`.

Ordinal-count output:

`response_type = "ordinal_counts"` returns a weighted count matrix in `y` and `weights = NULL`. This output is useful for ordinal or categorical likelihoods that take a count matrix instead of a factor response plus case weights.

This mode requires the response formula to have a bare-symbol left-hand side that references a factor or ordered-factor column. It also requires non-`NULL` weights, at least two declared response levels, and non-missing retained responses. Ordinal-count rows place the row's observation weight in the observed response column and zero in all other columns. Declared but unobserved response levels are retained as zero-count columns. See `ordinal_matrix_to_factor()` for converting the count matrix back to factor responses and row weights.

Value

A list of class `"design_inputs"` with elements:

- `x`: retained training design matrix.
- `y`: retained response, weighted ordinal-count matrix, or `NULL` for one-sided designs.
- `weights`: retained analysis weights, or `NULL`.
- `offset`: retained offset values, or `NULL`.
- `case_id`: retained case identifiers, or `NULL`.
- `row_index`: retained row identities from the fitted design.

All non-`NULL` elements are aligned to the rows of `x`.

See Also

`design_fit()`, `design_newdata()`, `ordinal_matrix_to_factor()`

Examples

```
#-------------------------------------------------------------------------------
# Extract training inputs for a weighted linear model
#-------------------------------------------------------------------------------
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"
)

fit <- design_fit(spec, analysis)
inputs <- design_data(fit)

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

#-------------------------------------------------------------------------
# Extract weighted ordinal-count responses
#-------------------------------------------------------------------------
ordinal_data <- 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)
)

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

ordinal_fit <- design_fit(ordinal_spec, ordinal_data)
ordinal_inputs <- design_data(
  ordinal_fit,
  response_type = "ordinal_counts"
)
ordinal_inputs$y
rowSums(ordinal_inputs$y)
```

`design_fit` *Fit a design schema from a specification and data*

Description

`design_fit()` fits a design schema to analysis data. It does not fit a statistical model. It evaluates

the formula and role expressions from a design_spec() object, resolves ., learns categorical levels, applies the missing-value policy, and builds the training design matrix.

The fitted schema is the object to reuse when training data and validation or prediction data must have the same predictor columns and factor-level encoding.

Usage

```
design_fit(spec, data, row_index = NULL)
```

Arguments

spec	(design_spec) Unfitted design specification returned by design_spec().
data	(data.frame) Analysis data used to learn the design schema. Objects that inherit from data.frame, including tibbles, are accepted.
row_index	(integer: NULL) Optional row identity vector for data. If NULL, seq_len(nrow(data)) is used. Duplicate values are allowed. Values must be non-missing whole numbers and must have length nrow(data).

Details

Workflow:

A typical modeling workflow starts with design_spec() to declare the design. design_fit() then learns the concrete schema from one analysis dataset. Use design_data() to extract training inputs from the fitted design. Use design_newdata() to encode validation or prediction data with the same predictor terms, categorical levels, column order, and role handling.

Retained rows:

Retained rows are computed once from the evaluated predictor, response, weight, and offset expressions before degenerate categorical predictors are dropped. The na_action value stored in spec controls whether rows with missing predictor or response values are kept, removed, or rejected. If no rows remain after applying na_action, design_fit() errors.

Weights and offsets are stricter than predictors and responses. 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.

Degenerate categorical predictors:

After retained rows are fixed, categorical predictor columns with fewer than two observed retained levels are detected. Formula terms that reference any such categorical predictor are dropped in one pass. Interaction terms that reference a degenerate categorical predictor are dropped along with the main effect. Degenerate-term dropping does not add rows back and does not remove additional rows.

Constant numeric predictors are retained. They may produce rank-deficient predictor matrices. Downstream model functions are responsible for handling that rank deficiency.

Character predictors are treated as categorical predictors. Factor predictors store the observed retained levels in the fitted predictor schema. Declared but unobserved factor levels are not kept as design-matrix columns.

Response levels:

For a bare-symbol factor or ordered-factor response, `response_levels` stores the declared response levels from the analysis data. These levels are needed for ordinal-count output in `design_data()` and `design_newdata()`.

For one-sided specifications, transformed responses, and non-factor responses, `response_levels = NULL`.

Row index:

`row_index` is the row identity vector for the supplied `data`. `row_index = NULL` means `seq_len(nrow(data))`. When supplied, `row_index` must be a whole-number vector with `length(row_index) == nrow(data)`. It may contain duplicates, which is required for bootstrap analysis data. After NA filtering, returned row indices are `row_index[retained_positions]`.

Value

A list of class `"design_fit"`. Important elements include:

- `spec`: the original `design_spec()` object.
- `formula`, `predictor_terms`, and `response_terms`: fitted formula objects.
- `predictor_schema`: learned predictor roles and categorical levels.
- `row_index`: retained row identities after applying `na_action`.
- `dropped`: degenerate categorical variables and terms dropped during fitting.
- `x`: the retained training design matrix.
- `y`: the retained evaluated response, or `NULL` for one-sided designs.
- `weights`, `offset`, and `case_id`: retained role values, or `NULL` when the role was not specified.

The object also stores the role formulas, role source variables, `encoding`, `intercept`, `na_action`, `novel_levels`, `response_name`, and `response_levels` so the same schema can be applied to new data.

See Also

`design_spec()`, `design_data()`, `design_newdata()`

Examples

```
#-------------------------------------------------------------------------
# Fit a design schema and use it for weighted linear regression
#-------------------------------------------------------------------------
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"
)

fit <- design_fit(spec, analysis)
colnames(fit$x)
fit$row_index

inputs <- design_data(fit)

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

#------------------------------------------------------------------------------
# Encode validation data with the fitted training schema
#------------------------------------------------------------------------------
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)
)

validation_inputs <- design_newdata(
  fit,
  validation,
  outcome = "ignore",
  weights = "ignore"
)

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

design_fold *Materialize one fold of a design-aware resampling object*

Description

`design_fold()` returns model-ready inputs for one split of a design-aware resampling object. It fits the split-specific design schema on the split's analysis rows and encodes non-analysis rows with that fitted schema. It does not fit a statistical model.

For ordinary CV folds, the result contains `analysis` and `assessment`. For nested-CV outer folds, the result contains `analysis`, `assessment`, and `inner`. For bootstrap splits, the result contains `analysis`, `assessment`, and `oob`.

Usage

```
design_fold(resample, i)
```

Arguments

`resample`	(design_resamples) A design-aware resampling object returned by `design_cv()`, `design_nested_cv()`, `design_bootstrap()`, or `design_resample()`.
`i`	(integer or character scalar) Split position or split name. Numeric values are coerced to integer with `base::as.integer()`. Character values are matched against `names(resample$splits)`.

Details

Workflow:

Use `design_fold()` after creating a design-aware resampling object with `design_cv()`, `design_nested_cv()`, `design_bootstrap()`, or `design_resample()`. Select a split by integer position or by split name. Fit the statistical model on `fold$analysis`. Evaluate predictions on `fold$assessment`, `fold$oob`, or an inner fold depending on the resampling method.

Design schemas:

Each split's design schema is fitted on its analysis rows only. Assessment and OOB rows are encoded with that fitted schema. No assessment or OOB schema is fitted.

This prevents assessment and OOB rows from determining factor levels, dropped categorical terms, column order, or other fitted design-schema details. Fold-local novel levels are handled by the `novel_levels` value stored in the original `design_spec()` object.

Row identity:

The `row_index` element in each returned `design_inputs` object refers to row numbers in the original data supplied to the resampling constructor or wrapper. For CV and nested-CV folds, `analysis$row_index` and `assessment$row_index` are disjoint. For bootstrap splits, `analysis$row_index` can contain duplicates and preserves bootstrap draw order. Bootstrap `assessment$row_index` is the full retained-row universe. Bootstrap `oob$row_index` contains rows not present in `unique(analysis$row_index)`.

Assessment and OOB row identities are preserved even when fold-local novel levels encode to NA.

Nested CV:

Materializing a nested-CV outer fold returns an inner design-aware CV object. Its splits are restricted to the outer analysis rows. Call design_fold(outer$inner, j) to materialize an inner fold for model selection. The outer fold's seed is used for the final refit, and each inner fold's seed is used for its candidate sweep.

Reproducible fits:

Each fold carries the split's seed, so a fold is self-contained. Call set.seed(fold$seed) immediately before fitting a random-using model on fold$analysis to make the fit reproducible regardless of run order. Applying the seed sets the global random-number state. See resample_cv() for the seeding recipe and its rationale.

Response output:

The response representation is fixed when the design-aware resampling object is created. With response_type = "response", materialized inputs contain the ordinary evaluated response. With response_type = "ordinal_counts", materialized y components contain weighted ordinal-count response matrices.

Value

A list of class c("design_<method>_fold", "design_fold"). The first class is one of "design_cv_fold", "design_nested_cv_fold", or "design_bootstrap_fold".

All returned folds contain:

- id: split name.
- seed: the split's single positive integer seed.
- analysis: a "design_inputs" object for model fitting.
- assessment: a "design_inputs" object for assessment.

Nested-CV outer folds also contain:

- inner: a design-aware CV object restricted to the outer analysis rows.

Bootstrap folds also contain:

- oob: a "design_inputs" object for out-of-bag assessment.

See Also

design_cv(), design_nested_cv(), design_bootstrap(), design_resample(), design_data(), design_newdata()

Examples

```
#--------------------------------------------------------------------------------
# Materialize a CV fold and estimate assessment error
#--------------------------------------------------------------------------------
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)),
  stage = factor(rep(c("I", "II", "III"), each = 8)),
  exposure = runif(24, min = 0.5, max = 2),
  inverse_prob_weight = runif(24, min = 0.8, max = 1.2),
  chart_score = rnorm(24)
)
analysis$risk_group <- interaction(
  analysis$treatment,
  analysis$stage,
  drop = TRUE
)

analysis$outcome <- with(
  analysis,
  5 + 0.25 * age + 1.5 * (treatment == "active") +
    0.8 * (stage == "III") + log(exposure) + rnorm(24)
)

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

set.seed(42L)
cv <- design_cv(
  spec,
  analysis,
  n_folds = 4L,
  strata = ~ risk_group
)

fold <- design_fold(cv, 1L)

fit <- stats::lm.wfit(
  x = fold$analysis$x,
```

```
  y = fold$analysis$y,
  w = fold$analysis$weights,
  offset = fold$analysis$offset
)

predicted <- drop(fold$assessment$x %*% fit$coefficients)
predicted <- predicted + fold$assessment$offset
sqrt(mean((fold$assessment$y - predicted)^2))

#-------------------------------------------------------------------------
# Materialize an inner fold for nested model selection
#-------------------------------------------------------------------------
set.seed(7L)
ncv <- design_nested_cv(
  spec,
  analysis,
  n_outer_folds = 3L,
  n_inner_folds = 2L,
  strata = ~ risk_group
)

outer <- design_fold(ncv, 1L)
inner <- design_fold(outer$inner, 1L)
all(inner$analysis$row_index %in% outer$analysis$row_index)
all(inner$assessment$row_index %in% outer$analysis$row_index)

#-------------------------------------------------------------------------
# Materialize a bootstrap split for out-of-bag assessment
#-------------------------------------------------------------------------
set.seed(11L)
bs <- design_bootstrap(
  spec,
  analysis,
  n_boot = 10L,
  strata = ~ risk_group
)

boot_fold <- design_fold(bs, 1L)
c(
  analysis_rows = length(boot_fold$analysis$row_index),
  oob_rows = length(boot_fold$oob$row_index)
)
```

`design_nested_cv` *Create a design-aware nested K-fold cross-validation plan*

Description

`design_nested_cv()` creates a lazy design-aware resampling object for nested K-fold cross-validation. It computes the retained row universe from a `design_spec()` object and creates outer and inner K-

fold split indices on those rows. It does not fit a statistical model and does not build a full-data design matrix.

Use design_fold() to materialize one outer split. The materialized outer fold contains analysis, assessment, and an inner design-aware CV object. Use design_fold() again on outer$inner to materialize inner folds for model selection within the outer analysis rows.

Usage

```
design_nested_cv(
  spec,
  data,
  n_outer_folds = 5L,
  n_inner_folds = 5L,
  n_iters = 1L,
  strata = NULL,
  group = NULL,
  response_type = c("response", "ordinal_counts")
)
```

Arguments

spec	(design_spec) Unfitted design specification returned by design_spec().
data	(data.frame) Analysis data used to create folds. Objects that inherit from data.frame, including tibbles, are accepted.
n_outer_folds	(numeric scalar: 5L; whole number in [2, unit_n]) Number of outer folds. unit_n is the number of retained rows when group is NULL and the number of retained groups otherwise. Whole-number numeric values such as 5 and 5.0 are accepted and stored as integers.
n_inner_folds	(numeric scalar: 5L; whole number in [2, unit_n - ceiling(unit_n / n_outer_folds)]) Number of inner folds. The upper bound ensures that every outer-analysis set has enough sampling units for inner cross-validation. Whole-number numeric values such as 5 and 5.0 are accepted and stored as integers.
n_iters	(numeric scalar: 1L; whole number in [1, .Machine$integer.max]) Number of independent outer K-fold partitions. Use values greater than one for repeated nested cross-validation.
strata	(formula: NULL) Stratification specification. Use NULL for no stratification. Use ~ 1 to stratify by the raw response. Use a one-sided formula such as ~ treatment or ~ site + treatment to stratify by observed columns.
group	(formula: NULL) Grouping specification. Use NULL for row-level resampling. Use a one-sided formula such as ~ patient_id to keep rows with the same group value together.
response_type	(Scalar character: "response") Response output mode used by design_fold(). Must be one of "response" or "ordinal_counts".

Details

Workflow:

A typical nested validation workflow starts with `design_spec()` to declare the design rules. Call `design_nested_cv()` after setting the random-number seed to create reproducible outer and inner folds. For each outer split, use the inner folds to choose a candidate model or tuning parameter. Then fit the selected candidate on the full outer analysis input and evaluate it on the outer assessment input.

Keeping split construction separate from fold materialization prevents assessment rows from determining factor levels, dropped categorical terms, column order, or other fitted design-schema details. Outer assessment rows are not used during inner model selection.

Retained rows:

The retained row universe is computed once before folds are assigned. The `na_action` value stored in `spec` controls whether rows with missing predictor or response values are kept, removed, or rejected. Weights and offsets are checked on retained rows using the same rules as `design_fit()`.

Each outer and inner split stores row identities from the retained universe. When a fold is materialized, `row_index` values in `analysis` and `assessment` refer to row numbers in the original `data`. Assessment row identities are preserved after split construction so prediction errors can be aligned back to the original data. Assessment predictors and responses are still encoded with the fold-specific analysis schema.

Splits:

`n_outer_folds` controls the number of outer folds in each outer K-fold partition. `n_inner_folds` controls the number of inner folds within each outer analysis set. `n_iters` controls the number of independent outer K-fold partitions. The total number of outer splits is `n_outer_folds * n_iters`. Fold assignment is over retained rows when `group` is `NULL` and over retained groups when `group` is supplied.

Within each iteration, every retained row appears in exactly one outer assessment set. Within each outer split, every outer-analysis row appears in exactly one inner assessment set. Across different iterations, the same row can appear in different outer and inner folds. Call `base::set.seed()` immediately before `design_nested_cv()` for reproducible split assignment.

Each outer split carries a `seed` for its final refit, and each inner split carries a `seed` for its candidate sweep. `design_fold()` copies the relevant seed onto the materialized outer and inner folds. See `resample_nested_cv()` for the seeding recipe and its rationale.

Operational variables:

`group` and `strata` are optional operational controls for assigning folds. They are evaluated on the retained rows after the design specification computes the resampling row universe.

If the design formula uses `.`, variables used only for `group` or `strata` must not enter the predictor set through `.` expansion. Put those variables in `case_id`, `weights`, `offset`, or `exclude`, or write an explicit predictor formula. Operational variables may also be named explicitly as predictors when that is the intended statistical design.

Response stratification:

Use `strata = NULL` for unstratified folds. Use a one-sided formula such as `strata = ~ treatment` to balance folds by observed covariate levels. Use `strata = ~ 1` to stratify by the raw response.

Response stratification evaluates the response before any `response_type` conversion. For ordinal-count workflows with rare levels, `strata = ~ 1` can reduce the chance that an analysis fold has an all-zero declared-level response column. See `resample_nested_cv()` for additional stratification and grouping details.

`response_type`**:**

`response_type = "response"` returns the ordinary evaluated response in each materialized fold. `response_type = "ordinal_counts"` returns weighted ordinal-count response matrices in `analysis$y` and `assessment$y`. Ordinal-count folds require a two-sided bare-symbol factor or ordered-factor response with non-missing retained responses. They also require weights when folds are materialized because the response matrix stores weighted counts.

The response type is fixed when the nested CV object is created. Every materialized outer and inner fold uses the same response representation for analysis and assessment inputs. Create a separate design nested-CV object when a different response representation is needed.

Value

A list of class `c("design_nested_cv_resamples", "design_resamples")` with elements:

- `spec`: the original `design_spec()` object.
- `data`: the original analysis data.
- `row_index`: retained row identities used for split construction.
- `response_type`: response representation used by materialized folds.
- `response_levels`: declared response levels for ordinal-count output, or `NULL`.
- `resamples`: the underlying index-only `nested_cv_resamples` object.
- `splits`: a named list of outer split indices.

Each outer split contains `analysis`, `assessment`, `iter`, `outer_fold`, `seed`, and `inner`. Each inner split contains `analysis`, `assessment`, `iter`, `outer_fold`, `inner_fold`, and `seed`. Use `design_fold()` to convert outer or inner splits into model-ready inputs.

See Also

`design_spec()`, `design_fold()`, `design_fit()`, `design_data()`, `design_cv()`, `resample_nested_cv()`, `base::set.seed()`

Examples

```
#-------------------------------------------------------------------------
# Estimate prediction error after inner-fold model selection
#-------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
analysis <- data.frame(
  patient_id = sprintf("P%02d", 1:36),
  age = round(rnorm(36, mean = 60, sd = 8)),
  treatment = factor(rep(c("control", "active"), 18)),
  stage = factor(rep(c("I", "II", "III"), each = 12)),
```

```
  exposure = runif(36, min = 0.5, max = 2),
  inverse_prob_weight = runif(36, min = 0.8, max = 1.2),
  chart_score = rnorm(36)
)
analysis$risk_group <- interaction(
  analysis$treatment,
  analysis$stage,
  drop = TRUE
)

analysis$outcome <- with(
  analysis,
  5 + 0.25 * age + 1.5 * (treatment == "active") +
    0.8 * (stage == "III") + log(exposure) + rnorm(36)
)

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

candidate_models <- c("age_only", "full")

candidate_columns <- function(inputs, candidate) {
  if (candidate == "age_only") {
    return(c("(Intercept)", "age"))
  }
  colnames(inputs$x)
}

candidate_rmse <- function(fold, candidate) {
  cols <- candidate_columns(fold$analysis, candidate)

  fit <- stats::lm.wfit(
    x = fold$analysis$x[, cols, drop = FALSE],
    y = fold$analysis$y,
    w = fold$analysis$weights,
    offset = fold$analysis$offset
  )

  predicted <- drop(
    fold$assessment$x[, cols, drop = FALSE] %*% fit$coefficients
  )
  predicted <- predicted + fold$assessment$offset
  sqrt(mean((fold$assessment$y - predicted)^2))
}
```

```
set.seed(42L)
ncv <- design_nested_cv(
  spec,
  analysis,
  n_outer_folds = 3L,
  n_inner_folds = 2L,
  strata = ~ risk_group
)

nested_results <- lapply(seq_along(ncv$splits), function(i) {
  outer <- design_fold(ncv, i)

  inner_rmse <- vapply(candidate_models, function(candidate) {
    fold_rmse <- vapply(seq_along(outer$inner$splits), function(j) {
      inner <- design_fold(outer$inner, j)
      candidate_rmse(inner, candidate)
    }, numeric(1))
    mean(fold_rmse)
  }, numeric(1))

  selected <- names(which.min(inner_rmse))

  data.frame(
    outer_split = outer$id,
    selected_model = selected,
    rmse = candidate_rmse(outer, selected)
  )
})

nested_results <- do.call(rbind, nested_results)
nested_results
mean(nested_results$rmse)

#-------------------------------------------------------------------------
# Keep repeated patient observations together
#-------------------------------------------------------------------------
repeated <- data.frame(
  patient_id = rep(sprintf("P%02d", 1:18), each = 2),
  visit = rep(c("baseline", "follow_up"), times = 18),
  treatment = factor(rep(rep(c("control", "active"), each = 9), each = 2)),
  age = rep(round(rnorm(18, mean = 60, sd = 8)), each = 2),
  outcome = rnorm(36)
)

repeated_spec <- design_spec(
  outcome ~ age + treatment + visit,
  case_id = ~ patient_id,
  encoding = "dummy",
  intercept = TRUE
)

set.seed(7L)
```

```
grouped_ncv <- design_nested_cv(
  repeated_spec,
  repeated,
  n_outer_folds = 3L,
  n_inner_folds = 2L,
  group = ~ patient_id,
  strata = ~ treatment
)

outer_fold <- design_fold(grouped_ncv, 1L)
inner_fold <- design_fold(outer_fold$inner, 1L)
intersect(inner_fold$analysis$case_id, inner_fold$assessment$case_id)
```

`design_newdata` *Apply a fitted design schema to new data*

Description

`design_newdata()` applies a fitted `design_fit()` schema to validation, assessment, or prediction data. It builds a `design_inputs` object whose predictor matrix has the same columns, categorical encoding, factor levels, and column order as the training design. It does not refit the schema or learn new factor levels from `newdata`.

Use this function after `design_fit()` when new data must be encoded in the same way as the analysis data used for model training.

Usage

```
design_newdata(
  fit,
  newdata,
  row_index = NULL,
  outcome = c("auto", "require", "ignore"),
  weights = c("auto", "require", "ignore"),
  response_type = c("response", "ordinal_counts"),
  na_action = fit$na_action,
  novel_levels = fit$novel_levels
)
```

Arguments

`fit`	(design_fit) Fitted design schema returned by `design_fit()`.
`newdata`	(data.frame) New data to encode. Objects that inherit from `data.frame`, including tibbles, are accepted.

`row_index` (integer: `NULL`)
Optional row identity vector for `newdata`. If `NULL`, `seq_len(nrow(newdata))` is used. Values must be non-missing, unique whole numbers and must have length `nrow(newdata)`.

`outcome` (Scalar character: `"auto"`)
Response evaluation mode. Must be one of `"auto"`, `"require"`, or `"ignore"`.

`weights` (Scalar character: `"auto"`)
Weight evaluation mode. Must be one of `"auto"`, `"require"`, or `"ignore"`.

`response_type` (Scalar character: `"response"`)
Response output mode. Must be one of `"response"` or `"ordinal_counts"`.

`na_action` (Scalar character: `fit$na_action`)
Missing-value policy used when retaining new-data rows. Must be one of `"na.pass"`, `"na.omit"`, `"na.fail"`, or `"na.exclude"`.

`novel_levels` (Scalar character: `fit$novel_levels`)
Handling for categorical levels in `newdata` that were not observed when `fit` was created. Must be one of `"na"` or `"fail"`.

Details

Predictor schema:

All predictor source variables required by the fitted schema must be present in `newdata`. Character and factor predictors are constrained to the levels learned by `design_fit()`. `novel_levels = "na"` converts new categorical levels to `NA` before row filtering and encoding. `novel_levels = "fail"` errors and reports the affected variables and values.

The returned `x` matrix has the same column names and order as `fit$x`. Missing one-hot columns are filled with zeros when a fitted training level is absent from `newdata`.

Retained rows:

`na_action` controls row retention for evaluated predictors, responses, weights, and offsets. The default is `fit$na_action`. `na_action = "na.pass"` keeps rows with missing predictor or response values. `na_action = "na.omit"` and `na_action = "na.exclude"` keep only complete evaluated rows. `na_action = "na.fail"` errors if any evaluated predictor or response value is missing.

Weights and offsets are stricter. On retained rows, evaluated weights must be non-missing, finite, and non-negative. On retained rows, evaluated offsets must be non-missing and finite. Case IDs are metadata and do not participate in row filtering.

Outcome modes:

`outcome = "auto"` evaluates `y` when all response source variables are present in `newdata` and returns `NULL` only when none are present. `outcome = "require"` errors when any response source variable is absent from `newdata`. `outcome = "ignore"` never evaluates the response.

For one-sided designs, `y = NULL` for all `outcome` modes. Use `outcome = "ignore"` for pure prediction data without observed outcomes. Use `outcome = "require"` for assessment data where the response should be present.

Weight modes:

If the fitted design has no weights, all `weights` modes return `weights = NULL`. Otherwise, `weights = "auto"` evaluates weights when all weight source variables are present and returns `NULL` when none are present. `weights = "require"` errors when any weight source variable is absent. `weights = "ignore"` never evaluates weights and returns `NULL`.

Use `weights = "ignore"` for prediction data where analysis weights are not needed. Use `weights = "require"` for assessment data where weighted scoring or weighted likelihoods require weights.

Offsets:

Offsets are required whenever the fitted design has an offset. If any required offset source variable is absent from `newdata`, `design_newdata()` errors. Offset values must be non-missing and finite on retained rows.

Case IDs:

When all fitted case ID source variables are present, `case_id` is returned aligned to the retained rows. When none are present, `case_id = NULL`. When only some are present, `design_newdata()` errors.

Ordinal-count output:

Ordinal-count output requires evaluated responses and evaluated weights. Therefore `outcome = "ignore"` and `weights = "ignore"` error. `"auto"` behaves like `"require"` for both response and weight source variables. Missing retained responses error. Response values must be contained in the fitted declared response levels.

Value

A list of class `"design_inputs"` with elements:

- `x`: encoded new-data design matrix with the same columns as `fit$x`.
- `y`: evaluated response, weighted ordinal-count matrix, or `NULL`.
- `weights`: evaluated weights, or `NULL`.
- `offset`: evaluated offsets, or `NULL`.
- `case_id`: evaluated case identifiers, or `NULL`.
- `row_index`: retained row identities from `newdata`.

All non-`NULL` elements are aligned to the rows of `x`.

See Also

`design_fit()`, `design_data()`, `ordinal_matrix_to_factor()`

Examples

```
#-------------------------------------------------------------------------
# Encode prediction data with the training design 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),
  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"
)

fit <- design_fit(spec, analysis)
inputs <- design_data(fit)

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)
)

prediction_inputs <- design_newdata(
  fit,
  prediction_data,
  outcome = "ignore",
  weights = "ignore"
)

linear_predictor <- drop(prediction_inputs$x %*% model$coefficients)
linear_predictor <- linear_predictor + prediction_inputs$offset
linear_predictor

#------------------------------------------------------------------------------
# Encode assessment data with observed outcomes and weights
```

```
#--------------------------------------------------------------------------
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)
)

assessment_inputs <- design_newdata(
  fit,
  assessment_data,
  outcome = "require",
  weights = "require"
)

assessment_inputs$y
assessment_inputs$weights
```

`design_resample` *Wrap a user-supplied package resampling object with a design specification*

Description

`design_resample()` validates a user-supplied `cv_resamples`, `nested_cv_resamples`, or `bootstrap_resamples` object against a design spec and returns a design-aware resampling object. It computes the retained row universe from a `design_spec()` object and checks that the supplied split indices are compatible with those rows. It does not create new splits, fit a statistical model, or build a full-data design matrix.

Use `design_fold()` to materialize one validated split. Materialized CV and bootstrap splits fit their design schema on their analysis rows only and then encode assessment or out-of-bag rows with that analysis schema. Materialized nested-CV outer splits also return an `inner` design-aware CV object whose inner folds are materialized separately.

Usage

```
design_resample(
  spec,
  data,
  resamples,
  response_type = c("response", "ordinal_counts")
)
```

Arguments

spec	(design_spec) Unfitted design specification returned by design_spec().
data	(data.frame) Analysis data referenced by resamples. Objects that inherit from data.frame, including tibbles, are accepted.
resamples	(resamples) A user-supplied cv_resamples, nested_cv_resamples, or bootstrap_resamples object. Its split indices must refer to rows of data.
response_type	(Scalar character: "response") Response output mode used by design_fold(). Must be one of "response" or "ordinal_counts".

Details

Workflow:

A typical workflow starts with an index-only resampling object created by resample_cv(), resample_nested_cv(), resample_bootstrap(), or another package function that returns the same classes and split structure. Call design_resample() to attach a design_spec() object to those split indices. Then iterate over seq_along(resample$splits), materialize each split with design_fold(), fit the model on fold$analysis, and evaluate predictions on the assessment or OOB inputs supplied by that resampling method.

Use design_resample() when the split indices already exist. Use design_cv(), design_nested_cv(), or design_bootstrap() when the design-aware object should create the split indices.

Retained rows:

The retained row universe is computed once before the supplied split indices are checked. The na_action value stored in spec controls whether rows with missing predictor or response values are kept, removed, or rejected. Weights and offsets are checked on retained rows using the same rules as design_fit().

All split indices must be contained in the design-retained row universe. If the design drops rows, create the resampling object on the retained rows or use one of the design-aware split constructors.

Validation:

The supplied resampling object's n must equal nrow(data). Split indices must be positive integer vectors.

For CV splits, analysis and assessment must be non-empty, unique, and disjoint within every split.

For nested-CV splits, outer and inner analysis and assessment vectors must be non-empty, unique, and disjoint. Every inner analysis and inner assessment vector must be contained in the corresponding outer analysis vector.

For bootstrap splits, assessment must equal the retained row universe. oob must be set-equal to the retained-row complement of unique(analysis). Bootstrap analysis may contain duplicates. Ungrouped bootstrap analysis must have the same length as the retained row universe.

response_type:

`response_type = "response"` returns the ordinary evaluated response in each materialized split. `response_type = "ordinal_counts"` returns weighted ordinal-count response matrices in the materialized `y` components supplied by the resampling method. Ordinal-count splits require a two-sided bare-symbol factor or ordered-factor response with non-missing retained responses. They also require weights when splits are materialized because the response matrix stores weighted counts.

The response type is fixed when the design-aware resampling object is created. Every materialized split uses the same response representation. Create a separate design-aware resampling object when a different response representation is needed.

Per-split seeds:

Each split may carry a `seed`, a single positive integer for reproducible model fits on that split. Objects built by `resample_cv()`, `resample_nested_cv()`, and `resample_bootstrap()` already carry seeds. `design_resample()` preserves the seeds on the wrapped object and does not create or modify them, so `design_fold()` propagates whatever seed each split carries. See `resample_cv()` for the seeding recipe and its rationale.

Value

A list of class `c("design_<method>_resamples", "design_resamples")` with elements:

- `spec`: the original `design_spec()` object.
- `data`: the original analysis data.
- `row_index`: retained row identities used to validate split indices.
- `response_type`: response representation used by materialized splits.
- `response_levels`: declared response levels for ordinal-count output, or `NULL`.
- `resamples`: the index-only resampling object.
- `splits`: the named list of split indices from `resamples`. Each split carries a `seed` when the wrapped object carried one.

The first class is one of `"design_cv_resamples"`, `"design_nested_cv_resamples"`, or `"design_bootstrap_resamples"`. Use `design_fold()` to convert a split into model-ready inputs.

See Also

`design_cv()`, `design_nested_cv()`, `design_bootstrap()`, `design_fold()`, `resample_cv()`, `resample_nested_cv()`, `resample_bootstrap()`

Examples

```
#-------------------------------------------------------------------------
# Wrap existing CV splits and estimate prediction error
#-------------------------------------------------------------------------
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)),
  stage = factor(rep(c("I", "II", "III"), each = 8)),
  exposure = runif(24, min = 0.5, max = 2),
  inverse_prob_weight = runif(24, min = 0.8, max = 1.2),
  chart_score = rnorm(24)
)
analysis$risk_group <- interaction(
  analysis$treatment,
  analysis$stage,
  drop = TRUE
)

analysis$outcome <- with(
  analysis,
  5 + 0.25 * age + 1.5 * (treatment == "active") +
    0.8 * (stage == "III") + log(exposure) + rnorm(24)
)

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

set.seed(42L)
raw_cv <- resample_cv(
  analysis,
  outcome ~ age,
  n_folds = 4L,
  strata = ~ risk_group
)

cv <- design_resample(spec, analysis, raw_cv)

fold_rmse <- vapply(seq_along(cv$splits), function(i) {
  fold <- design_fold(cv, i)

  fit <- stats::lm.wfit(
    x = fold$analysis$x,
    y = fold$analysis$y,
    w = fold$analysis$weights,
    offset = fold$analysis$offset
  )

  predicted <- drop(fold$assessment$x %*% fit$coefficients)
  predicted <- predicted + fold$assessment$offset
  sqrt(mean((fold$assessment$y - predicted)^2))
```

```
}, numeric(1))

fold_rmse
mean(fold_rmse)

#-------------------------------------------------------------------------------
# Wrap existing bootstrap splits for out-of-bag scoring
#-------------------------------------------------------------------------------
set.seed(7L)
raw_bs <- resample_bootstrap(
  analysis,
  outcome ~ age,
  n_boot = 10L,
  strata = ~ risk_group
)

bs <- design_resample(spec, analysis, raw_bs)
boot_fold <- design_fold(bs, 1L)
c(
  analysis_rows = length(boot_fold$analysis$row_index),
  oob_rows = length(boot_fold$oob$row_index)
)
```

`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_fit()` to learn a reproducible design schema from training data. Use it with `design_cv()`, `design_nested_cv()`, `design_bootstrap()`, or `design_resample()` to build resampling plans that use the same design rules.

Usage

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

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: 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: NULL) Optional one-sided formula for an offset. The right-hand side must evaluate to one numeric vector with one value per row. This argument must contain a single top-level expression. Use offset() terms in formula or create one offset column if multiple components are needed.
case_id	(formula or NULL: 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: 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	(Scalar character: "one_hot") Categorical predictor encoding used when the fitted design matrix is built. Must be one of "one_hot" or "dummy".
intercept	(Scalar logical: FALSE) Whether to include a numeric "(Intercept)" column in fitted design matrices.
na_action	(Scalar character: "na.pass") Missing-value policy for evaluated predictors and responses. Must be one of "na.pass", "na.omit", "na.fail", or "na.exclude".
novel_levels	(Scalar character: "na") Handling for categorical levels in new data that were not observed during fitting. Must be one of "na" or "fail".

Details

Workflow:

A typical workflow has three steps. First, call design_spec() to declare the statistical design. Second, call design_fit() on the analysis data to evaluate formulas, expand ., learn factor levels, and build the training design matrix. Third, use design_data() for training inputs and design_newdata() 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.

Encoding:

encoding = "dummy" uses treatment contrasts. encoding = "one_hot" creates one indicator column per categorical level. 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 explicitly suppresses the intercept with -1 or +0, intercept = TRUE errors.

With encoding = "dummy", categorical predictors are treatment-coded. The reference level is represented by zeros in the treatment columns. This is the usual choice for ordinary linear and generalized linear model workflows that include an intercept.

With encoding = "one_hot", categorical predictors use one column per level. The default intercept = FALSE gives a design matrix similar to glmnet::makeX() factor encoding. This is useful when a downstream fitter supplies its own intercept or when full level indicators are desired.

Some choices can produce rank-deficient design matrices. For example, encoding = "one_hot" with intercept = TRUE is rank deficient when categorical predictors are present. Additive formulas with multiple one-hot categorical predictors can also be rank deficient when intercept = FALSE. Choose encoding = "dummy" for reference-level contrasts. Choose encoding = "one_hot" for full level indicators.

Missing values:

na_action is one of "na.pass", "na.omit", "na.exclude", or "na.fail". It is applied when design_fit(), design_newdata(), 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.

Value

A list of class "design_spec". It stores the unevaluated design contract and contains these elements:

- formula
- weights, offset, case_id, exclude
- encoding, intercept, na_action, novel_levels

The returned object does not contain a design matrix, response vector, factor schema, retained-row index, or fitted model.

See Also

design_fit(), design_data(), design_newdata(), design_cv(), design_nested_cv(), design_bootstrap(), design_resample()

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"
)

fit <- design_fit(spec, analysis)
inputs <- design_data(fit)

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)
)

validation_inputs <- design_newdata(
  fit,
  validation,
  outcome = "ignore",
  weights = "ignore"
)

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

`get_dep_data` *Get dependent test data*

Description

Get data for one-sample or dependent two-sample tests using a formula. Three forms are supported.

- `y ~ x | z`
- `y ~ x`
- `~x`

Usage

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

Arguments

`data`	(data.frame) The data frame containing the formula variables.

formula (formula)
The formula to be parsed and evaluated on the data.

allow_one_sample
(scalar logical: TRUE)
Whether or not a formula of form ~x is allowed.

agg_fun (Scalar character or function: "error")
Used for aggregating duplicate cases of grouping/blocking combinations when formula of form y ~ x | z is used. Ignored with a warning when the formula has no block term. Select one of "first", "last", "sum", "mean", "median", "min", or "max" for built-in aggregation handling (each applies na.rm = TRUE). Or define your own function. User-defined functions are routed through a generic aggregator that is slower than the built-in cases. "error" (default) will return an error if duplicate grouping/blocking combinations are encountered.

Details

Dependent two-sample tests:

For dependent two-sample tests with data in tall format, the formula with form y ~ x | z is defined by

- y is a numeric vector for the outcome.
- x is a factor with exactly two observed levels representing the binary group variable.
- z is a factor with n levels representing the blocking variable (subject/item index).

For factor x, the first observed level will be the reference 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 takes the form mean(post - pre).

A non-factor x or z is converted with as.factor() and a warning is emitted. as.factor() then orders the levels alphabetically or numerically, so the reference group for x becomes its smallest value. Pass x as a factor to control which group is the reference. Declared-but-unobserved levels of x and z are dropped before the two-level check, so they cannot displace observed levels or seed all-NA rows or columns in the wide output.

For dependent two-sample tests with data in wide format, the formula with form y ~ x is defined by

- y is a numeric vector for the group of interest.
- x is a numeric vector for the reference group.
- Row i of y and row i of x correspond to the same subject/item.

For returned object out, downstream differences would take the form out$data[[out$focal_group]] - out$data[[out$ref_group]].

One-sample (or paired) tests:

For one-sample (or paired) tests, the formula has form ~x where

- x is a numeric vector for the outcome (or differences).

NA handling:

For ~x and y ~ x, missing values are retained and returned as-is. For y ~ x | z, a row with a missing value of x or z is dropped before reshaping. Missing values of x are ignored when checking that

it has exactly two observed levels. See tall_to_wide() for missing value handling during cell-aggregation. A missing value of y is retained. A cell with no observation in the wide output becomes NA_real_.

Wrapping terms in I():

Any term in the formula may be wrapped in I(). Operators such as + and * carry special meaning inside a formula. Any other expression, such as log(y) or x > 0, must be wrapped in I() so it is evaluated as ordinary R code against data. On the left hand side, I() transforms the outcome and must yield a numeric result. For example, I(log(y)) ~ x | z uses the log-transformed outcome. On the group side, I() derives the group term. For example, y ~ I(x > 0) | z groups by whether x is positive. On the block side, I() derives the block term. For example, y ~ x | I(interaction(subject, session)) blocks by a subject-session combination.

Value

A named list with components:

1. data

   The response values split by group. For ~x a length-1 named list holding the outcome vector. For y ~ x a length-2 named list with the focal group first and the reference group second. For y ~ x | z a data.frame with the focal group in column 1, the reference group in column 2, and the block identifier in column 3, with one row per level of the block factor in levels(z) order.

2. focal_group

   A scalar character naming the focal group. The name of the response variable for y ~ x, the single term for ~ x, or the second factor level for y ~ x | z.

3. ref_group

   A scalar character naming the reference group, or NULL for a ~x formula. The name of the grouping variable for y ~ x, or the first factor level for y ~ x | z.

4. block

   A scalar character naming the block column for y ~ x | z, or NULL otherwise.

5. parse_formula

   The list returned by parse_formula() applied to 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 test data*

Description

Get data for independent two-sample tests when defined by a formula of form `y ~ x`.

Usage

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

Arguments

`data`	(data.frame) The data frame containing the formula variables.

`formula` (formula)
The formula to be parsed and evaluated on the data.

Details

The formula with form `y ~ x` is defined by

- `y` is a numeric vector for the outcome.
- `x` is a factor with exactly two observed levels representing the binary group variable.

For factor `x`, the first observed level will be the reference 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 takes the form `mean(post) - mean(pre)`.

A non-factor `x` is converted with `as.factor()` and a warning is emitted. The reference group is then the smallest value of `x`. `as.factor()` orders the levels alphabetically or numerically. Pass `x` as a factor to control which group is the reference.

A row with a missing value of `x` is dropped from both groups. Missing values of `x` are ignored when checking that it has exactly two observed levels. A missing value of `y` is kept in its group.

Either side of the formula may wrap a term in `I()`. Operators such as `+` and `*` carry special meaning inside a formula. Any other expression, such as `log(y)` or `x > 0`, must be wrapped in `I()` so it is evaluated as ordinary R code against `data`. On the left hand side, `I()` transforms the outcome and must yield 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.

Value

A named list with components:

1. `data`
   A length-2 named list of numeric response vectors, the focal group first and the reference group second, each named by its factor level.
2. `focal_group`
   A scalar character giving the focal group name, the second level with observations in the grouping factor.
3. `ref_group`
   A scalar character giving the reference group name, the first level with observations in the grouping factor.
4. `parse_formula`
   The list returned by `parse_formula()` applied to `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

A unified interface to four optimization methods.

Usage

```
optim2(fun, parameters, method = "optim", lower = -Inf, upper = Inf, ...)
```

Arguments

fun	(function) A function to be minimized.

`parameters` (numeric)
A numeric vector of initial values for each parameter to be optimized in `fun`.

`method` (string: `"optim"`)
A string for the optimization method. Must be one of:

- `"nlm"`: `stats::nlm()`, a Newton-type line-search algorithm.
- `"nlminb"`: `stats::nlminb()`, box-constrained optimization via PORT routines.
- `"optim"`: `stats::optim()` with Nelder-Mead (the default method).
- `"optim_constrained"`: `stats::optim()` with `method = "L-BFGS-B"`.

`lower` (numeric: `-Inf`)
Lower bound for each parameter. Either a scalar applied uniformly to all parameters, or a numeric vector of length `length(parameters)` with one bound per parameter. Used only by `"nlminb"` and `"optim_constrained"`; ignored by `"nlm"` and `"optim"`.

`upper` (numeric: `Inf`)
Upper bound for each parameter. Either a scalar applied uniformly to all parameters, or a numeric vector of length `length(parameters)` with one bound per parameter. Used only by `"nlminb"` and `"optim_constrained"`; ignored by `"nlm"` and `"optim"`.

`...` Additional arguments passed to the corresponding `method`. When `method = "optim_constrained"`, do not pass `method` via `...`; `"L-BFGS-B"` is forced automatically and a duplicate argument will cause an error.

Value

A named list with five elements:

1. `estimate`: numeric vector of optimized parameter values, with length equal to `length(parameters)`.
2. `minimum`: scalar numeric giving the value of `fun` at `estimate`.
3. `iterations`: scalar integer. For `"nlm"` and `"nlminb"`, the number of optimizer iterations. For `"optim"` and `"optim_constrained"`, the number of function evaluations (`counts["function"]` from `stats::optim()`); `optim()` does not report iteration counts, so this quantity is not directly comparable to the other methods.
4. `code`: integer convergence code. For `"nlm"`, codes 1–2 indicate probable convergence and codes 3–5 indicate potential failure; see `stats::nlm()` for definitions. For all other methods, 0 indicates success and a positive integer indicates failure; see `stats::nlminb()` and `stats::optim()` for method-specific codes.
5. `message`: character string describing the convergence outcome.

See Also

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

Examples

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

# Linear regression
# residuals should be normal with mean 0 and unknown standard deviation sigma
x1 <- 1:100
set.seed(1)
epsilon <- rnorm(100)
y <- 4 + x1*2 + epsilon

nll <- function(par, y, x1){
  alpha <- par[1]
  beta1 <- par[2]
  sigma <- par[3]
  resid <- y - alpha - beta1 * x1
  -sum(dnorm(resid, mean = 0, sigma, log = TRUE))
}

res_lm <- lm(y ~ x1)
res_lm

res_optim <- optim(
  fn = nll,
  par = c(alpha = mean(y),
          beta1 = 0,
          sigma = 1),
  y = y,
  x1 = x1
) |>
  suppressWarnings()
res_optim

res_optim2 <- optim2(
  fun = nll,
  parameters = c(alpha = mean(y),
                 beta1 = 0,
                 sigma = 1),
  method = "optim",
  y = y,
  x1 = x1
) |>
  suppressWarnings()
res_optim2

res_nlm <- optim2(
  fun = nll,
  parameters = c(alpha = mean(y),
                 beta1 = 0,
                 sigma = 1),
```

```
  method = "nlm",
  y = y,
  x1 = x1
) |>
  suppressWarnings()
res_nlm

res_nlminb <- optim2(
  fun = nll,
  parameters = c(alpha = mean(y),
                 beta1 = 0,
                 sigma = 1),
  method = "nlminb",
  lower = c(-Inf, -Inf, 1e-6),
  y = y,
  x1 = x1
)
res_nlminb

res_optim_constrained <- optim2(
  fun = nll,
  parameters = c(alpha = mean(y),
                 beta1 = 0,
                 sigma = 1),
  method = "optim_constrained",
  lower = c(-Inf, -Inf, 1e-6),
  y = y,
  x1 = x1
)
res_optim_constrained
```

`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. It is the inverse of the ordinal-count response representation returned by `design_data()` and `design_newdata()` when `response_type = "ordinal_counts"`.

Use this helper when a model, metric, or diagnostic needs a factor response plus case weights rather than a count matrix.

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. Column order is preserved as the factor's level order. Column names must be non-missing, non-empty, and unique.

Values must be non-missing, finite, and non-negative. Each row must have either zero positive cells or exactly one positive cell. Rows with one positive cell return that column as the factor level and the row's positive cell value as its weight. Rows with all zeros return `NA` and weight `0`. Rows with more than one positive cell error.

The function does not require integer counts. Fractional values are allowed because design-generated ordinal-count matrices use analysis weights.

Value

A list with elements:

- `y`: a factor with `levels = colnames(y)`.
- `weights`: a numeric vector equal to `rowSums(y)`.

The returned factor and weights are aligned row-for-row with the input matrix.

See Also

`design_data()`, `design_newdata()`

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 design_data(response_type = "ordinal_counts")
#-------------------------------------------------------------------------
set.seed(1L)
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
)

fit <- design_fit(spec, analysis)
ordinal_inputs <- design_data(
  fit,
  response_type = "ordinal_counts"
)

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

`parse_formula` *Parse formula*

Description

Parse a formula into its terms `lhs ~ rhs` where `rhs` can have two components separated by `|`. For example, `y ~ x | z` has `rhs = ~ x | z` where `x` is typically described as the grouping term and `z` is the blocking term. A fast, unsafe alternative to `modeltools::ParseFormula()`.

Usage

```
parse_formula(formula, specials = NULL, data = NULL, terms = FALSE)
```

Arguments

`formula`	(formula) The formula that should be parsed.
`specials`	(character: `NULL`) A character vector of functions in `formula` that should be flagged in the specials attribute when `terms.formula()` is applied. Has no effect unless expansion runs.

data	(data frame) Used to infer the meaning of the special symbol . in formula. Can be a data.frame or list.
terms	(scalar logical: FALSE) Whether or not to parse formula using stats::terms.formula(). Expansion is also performed when specials or data is non-NULL, even if terms is FALSE.

Value

A named list with components:

- formula: the (possibly terms.formula-expanded) input formula.
- lhs: a one-sided formula containing the response, or NULL if formula is one-sided.
- rhs: a one-sided formula containing the full right-hand side.
- rhs_group: a one-sided formula containing the term left of |, or rhs if no top-level | is present.
- rhs_block: a one-sided formula containing the term right of |, or NULL if no top-level | is present.

See Also

get_ind_data(), get_dep_data(), tall_to_wide()

Examples

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

# One-sided formula
parse_formula(~x)

# Two-sided formula
parse_formula(y ~ x)

# Two-sided formula with a top-level `|` splitting the rhs into group and block
parse_formula(y ~ x | z)

# Dot expansion against the names in `data`
parse_formula(y ~ ., data = data.frame(y = 1, a = 1, b = 1))

# Functions named in `specials` are preserved through term expansion
parse_formula(y ~ s(x) + z, specials = "s")
```

resample_bootstrap *Create bootstrap resampling indices*

Description

Create an index-only resampling plan for bootstrap resampling with replacement. Each split stores integer row positions from `data` for model fitting and assessment. The function supports stratified bootstrap samples and grouped bootstrap samples.

Usage

```
resample_bootstrap(data, formula, n_boot = 1000L, strata = NULL, group = NULL)
```

Arguments

`data`	(data.frame) Data set to resample. It must have at least one column and at least two rows.
`formula`	(formula) Two-sided model formula. The left-hand side must be a single untransformed column name in `data`. Calls such as `cbind(y1, y2)`, `Surv(time, status)`, `log(y)`, and `factor(y)` are rejected. The right-hand side is ignored when the resampling indices are created.
`n_boot`	(numeric scalar: `1000L`; whole number in `[1, .Machine$integer.max]`) Number of bootstrap replicates. Whole-number numeric values such as `10` and `10.0` are accepted and stored as integers.
`strata`	(formula: `NULL`) Stratification specification. Use `NULL` for no stratification. Use `~ 1` to stratify by the model outcome. Use a one-sided formula such as `~ a + b` to stratify by the interaction of columns `a` and `b`.
`group`	(formula: `NULL`) Grouping specification. Use `NULL` for row-level resampling. Use a one-sided formula such as `~ subject` to draw whole groups with replacement. The right-hand side must be a single bare column name. Create composite group keys in `data` before calling `resample_bootstrap()`.

Details

`resample_bootstrap()` does not copy `data`. It returns row indices that can be used to subset `data` later. The predictor side of `formula` is not used to build the splits. It is accepted so the same formula can be passed to a model-fitting function.

Splits:

Each element of `splits` contains:

- `analysis`: integer row indices drawn with replacement and used to fit a model. Duplicate row indices are expected. When `group` is `NULL`, length equals `nrow(data)`. When `group` is supplied, length equals the total row count of `n_groups` drawn groups, which can vary across bootstrap splits if group sizes differ.

- assessment: integer row indices for the full original data set, seq_len(nrow(data)). This supports bootstrap optimism correction.
- oob: integer out-of-bag row indices. These are computed as sort(setdiff(seq_len(nrow(data)), unique(analysis))).
- seed: single positive integer for reproducible model fits on this split.

Bootstrap assignment:

Rows are the sampling units when group is NULL. Groups are the sampling units when group is supplied. Each bootstrap split advances the current random-number generator state.

Stratification:

Stratification is optional. Pass NULL for no stratification. strata = ~ 1 stratifies by the model outcome. A formula such as strata = ~ a + b stratifies by interaction(data$a, data$b, drop = TRUE). Variables referenced by strata must be columns of data. NA stratum values are not allowed. Empty factor levels are dropped. Stratification variables are treated as categorical. Pre-bin continuous variables before passing them to strata. This also applies when strata = ~ 1 and the model outcome is continuous.

Expressions in strata are not evaluated as transformations. Create transformed or composite strata columns in data before calling resample_bootstrap().

For row-level bootstrap resampling, stratification samples rows with replacement within each stratum. It preserves the original row count of each stratum in each bootstrap split. The analysis vector concatenates per-stratum draws in factor-level order. Each bootstrap stratum must contain at least two rows.

Grouping:

Grouping is optional. Pass NULL for ordinary row-level resampling. Pass a one-sided formula such as group = ~ subject to draw whole groups with replacement. The right-hand side of group must be a single bare column name in data. NA group values are not allowed. Empty factor levels are dropped.

When group is supplied, groups are drawn with replacement. For each bootstrap split, n_groups groups are drawn and their rows are concatenated in draw order. assessment remains seq_len(nrow(data)). oob is the set of rows whose group was never drawn into analysis.

Stratified grouped bootstrap draws groups within each stratum to preserve group counts per stratum. Strata are processed in resolved factor-level order. Rows are concatenated in draw order within each stratum. This preserves group counts per stratum rather than row counts per stratum. When group sizes differ, total analysis length and per-stratum row counts can vary across bootstrap splits. Each grouped bootstrap stratum must contain at least two groups.

When strata and group are both supplied, the resolved stratum vector must be constant within each group. If strata = ~ 1 is used with group, the outcome must be constant within each group.

Split Names:

Split names are generated from n_boot. For fewer than 10 splits, names are split1, ..., split9. For 10 or more splits, names are zero-padded so lexical order matches numeric order.

Reproducibility:

Call base::set.seed() immediately before resample_bootstrap() for reproducible splits.

Split indices reproduce the resampled rows, but a downstream model fit that uses randomness depends on the global random-number state when it runs, so it does not reproduce on its own.

Each split therefore carries a `seed`, a single positive integer generated at construction. Set the split seed immediately before fitting that split to make the fit reproducible regardless of run order.

```
# Single fit per split.
set.seed(split$seed)
fit <- fit_model(split$analysis)

# Several candidate models on one analysis set, each reproducible alone.
set.seed(split$seed)
candidate_seeds <- sample.int(.Machine$integer.max, K, replace = FALSE)
for (k in seq_len(K)) {
  set.seed(candidate_seeds[k])
  fit_k <- fit_candidate_k(split$analysis)
}
```

Seeds are created from the global generator and the generator state is then restored, so split indices and the caller's random-number stream are unchanged. Reproducing a fit assumes the same `base::RNGkind()` on each run.

Value

An S3 list of class `c("bootstrap_resamples", "resamples")` with elements:

- `call`: the matched call.
- `method`: `"bootstrap"`.
- `n`: `nrow(data)`.
- `formula`: the input `formula`.
- `outcome_var`: the resolved outcome column name.
- `strata`: the input `strata` formula or `NULL`.
- `strata_vars`: the resolved strata column names or `NULL`.
- `group`: the input `group` formula or `NULL`.
- `group_vars`: the resolved group column name or `NULL`.
- `n_groups`: the number of unique groups when `group` is supplied, otherwise `NULL`.
- `n_boot`: the resolved bootstrap count.
- `splits`: a named list of split entries. Each split contains `analysis`, `assessment`, `oob`, and `seed`.

For ungrouped bootstrap resampling, `length(analysis)` equals `n`. For grouped bootstrap resampling, `length(analysis)` can differ from `n` when group sizes differ.

See Also

`resample_cv()`, `resample_nested_cv()`, `base::set.seed()`

Examples

```
#-------------------------------------------------------------------------
# resample_bootstrap() examples
#-------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)

data <- data.frame(
  outcome = c(rep("event", 12L), rep("none", 12L)),
  age = c(34L, 52L, 41L, 63L, 29L, 48L, 37L, 55L,
          46L, 31L, 60L, 43L, 39L, 50L, 28L, 44L,
          57L, 36L, 62L, 33L, 47L, 53L, 40L, 58L),
  clinic = rep(c("north", "south", "west"), each = 8L),
  patient = rep(seq_len(12L), each = 2L)
)

bs <- resample_bootstrap(data, outcome ~ age, n_boot = 10L)
bs$splits[[1L]]$analysis
bs$splits[[1L]]$oob

# Preserve the outcome class counts in each bootstrap sample.
resample_bootstrap(data, outcome ~ age, n_boot = 10L, strata = ~ outcome)

# Draw whole patients with replacement.
resample_bootstrap(data, outcome ~ age, n_boot = 10L, group = ~ patient)
```

---

resample_cv *Create K-fold cross-validation resampling indices*

Description

Create an index-only resampling plan for K-fold cross-validation. Each split stores integer row positions from `data` for model fitting and model assessment. The function supports repeated K-fold cross-validation, stratified folds, and grouped folds.

Usage

```
resample_cv(
  data,
  formula,
  n_folds = 5L,
  n_iters = 1L,
  strata = NULL,
  group = NULL
)
```

Arguments

data	(data.frame) Data set to resample. It must have at least one column and at least two rows.
formula	(formula) Two-sided model formula. The left-hand side must be a single untransformed column name in data. Calls such as cbind(y1, y2), Surv(time, status), log(y), and factor(y) are rejected. The right-hand side is ignored when the resampling indices are created.
n_folds	(numeric scalar: 5L; whole number in [2, unit_n]) Number of folds. unit_n is nrow(data) when group is NULL and the number of unique groups otherwise. Whole-number numeric values such as 5 and 5.0 are accepted and stored as integers.
n_iters	(numeric scalar: 1L; whole number in [1, .Machine$integer.max]) Number of independent K-fold partitions. Use values greater than one for repeated K-fold cross-validation.
strata	(formula: NULL) Stratification specification. Use NULL for no stratification. Use ~ 1 to stratify by the model outcome. Use a one-sided formula such as ~ a + b to stratify by the interaction of columns a and b.
group	(formula: NULL) Grouping specification. Use NULL for row-level resampling. Use a one-sided formula such as ~ subject to keep rows with the same group value together. The right-hand side must be a single bare column name. Create composite group keys in data before calling resample_cv().

Details

resample_cv() does not copy data. It returns row indices that can be used to subset data later. The predictor side of formula is not used to build the splits. It is accepted so the same formula can be passed to a model-fitting function.

Splits:

Each element of splits contains:

- analysis: integer row indices used to fit a model. These are all rows not in assessment. They are sorted in ascending order.
- assessment: integer row indices used to assess the fitted model. They are sorted in ascending order.
- iter: integer iteration number.
- fold: integer fold number within iter.
- seed: single positive integer for reproducible model fits on this split.

Within each iteration, the assessment indices partition seq_len(nrow(data)). A row appears in exactly one assessment set per iteration. Across different iterations, the same row can appear in different folds.

Fold assignment:

Folds are assigned over sampling units. Rows are the sampling units when group is NULL. Groups are the sampling units when group is supplied.

Without stratification, fold sizes are balanced over the sampling units. The target fold sizes are floor(unit_n / n_folds), with unit_n %% n_folds folds one unit larger. The larger folds are selected at random.

When strata is supplied, strata are processed from smallest to largest, so rare strata are allocated first. Within each stratum, sampling units are shuffled and distributed across folds as evenly as possible. Per-stratum fold counts differ by at most one. Ties are broken with the current random-number generator state.

Stratification:

Stratification is optional. Pass NULL for no stratification. strata = ~ 1 stratifies by the model outcome. A formula such as strata = ~ a + b stratifies by interaction(data$a, data$b, drop = TRUE). Variables referenced by strata must be columns of data. NA stratum values are not allowed. Empty factor levels are dropped. Stratification variables are treated as categorical. Pre-bin continuous variables before passing them to strata. This also applies when strata = ~ 1 and the model outcome is continuous.

Expressions in strata are not evaluated as transformations. Create transformed or composite strata columns in data before calling resample_cv().

Grouping:

Grouping is optional. Pass NULL for ordinary row-level resampling. Pass a one-sided formula such as group = ~ subject to keep all rows from each subject together. The right-hand side of group must be a single bare column name in data. NA group values are not allowed. Empty factor levels are dropped. Rows that share a group value are kept together. All rows of any one group land entirely in one fold's assessment per iteration. Because groups are the sampling units, n_folds must be less than or equal to the number of unique groups. Fold balance is defined over groups, not rows. When group sizes differ, row counts across folds may differ by more than one.

When strata and group are both supplied, the resolved stratum vector must be constant within each group. If strata = ~ 1 is used with group, the outcome must be constant within each group.

Repeated K-fold:

When n_iters > 1, the function creates n_iters independent K-fold partitions and concatenates the splits. The total number of splits is n_folds * n_iters. Each iteration advances the current random-number generator state.

Split Names:

Split names are generated from the total number of splits. For fewer than 10 splits, names are split1, ..., split9. For 10 or more splits, names are zero-padded so lexical order matches numeric order.

Reproducibility:

Call base::set.seed() immediately before resample_cv() for reproducible splits.

Split indices reproduce the fold rows, but a downstream model fit that uses randomness depends on the global random-number state when it runs, so it does not reproduce on its own. Each split therefore carries a seed, a single positive integer generated at construction. Set the split seed immediately before fitting that split to make the fit reproducible regardless of run order.

```
# Single fit per split.
set.seed(split$seed)
fit <- fit_model(split$analysis)

# Several candidate models on one analysis set, each reproducible alone.
set.seed(split$seed)
candidate_seeds <- sample.int(.Machine$integer.max, K, replace = FALSE)
for (k in seq_len(K)) {
  set.seed(candidate_seeds[k])
  fit_k <- fit_candidate_k(split$analysis)
}
```

Seeds are created from the global generator and the generator state is then restored, so split indices and the caller's random-number stream are unchanged. Reproducing a fit assumes the same `base::RNGkind()` on each run.

Value

An S3 list of class `c("cv_resamples", "resamples")` with elements:

- `call`: the matched call.
- `method`: `"cv"`.
- `n`: `nrow(data)`.
- `formula`: the input `formula`.
- `outcome_var`: the resolved outcome column name.
- `strata`: the input `strata` formula or `NULL`.
- `strata_vars`: the resolved strata column names or `NULL`.
- `group`: the input `group` formula or `NULL`.
- `group_vars`: the resolved group column name or `NULL`.
- `n_groups`: the number of unique groups when `group` is supplied, otherwise `NULL`.
- `n_folds`: the resolved fold count.
- `n_iters`: the resolved iteration count.
- `splits`: a named list of split entries. Each split contains `analysis`, `assessment`, `iter`, `fold`, and `seed`.

See Also

`resample_nested_cv()`, `resample_bootstrap()`, `base::set.seed()`

Examples

```
#--------------------------------------------------------------------------
# resample_cv() examples
#--------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)
```

```
data <- data.frame(
  outcome = c(rep("event", 12L), rep("none", 12L)),
  age = c(34L, 52L, 41L, 63L, 29L, 48L, 37L, 55L,
          46L, 31L, 60L, 43L, 39L, 50L, 28L, 44L,
          57L, 36L, 62L, 33L, 47L, 53L, 40L, 58L),
  clinic = rep(c("north", "south", "west"), each = 8L),
  patient = rep(seq_len(12L), each = 2L)
)

cv <- resample_cv(data, outcome ~ age, n_folds = 4L)
cv$splits[[1L]]$analysis
cv$splits[[1L]]$assessment

# Keep outcome classes balanced across folds.
resample_cv(data, outcome ~ age, n_folds = 4L, strata = ~ outcome)

# Repeat the full K-fold partition three times.
resample_cv(data, outcome ~ age, n_folds = 4L, n_iters = 3L)

# Keep repeated observations from the same patient together.
resample_cv(data, outcome ~ age, n_folds = 4L, group = ~ patient)
```

`resample_nested_cv` *Create nested K-fold cross-validation resampling indices*

Description

Create an index-only resampling plan for nested K-fold cross-validation. Each outer split stores integer row positions for final model assessment. Each outer split also contains inner cross-validation splits for model selection within the outer analysis set. The function supports repeated outer cross-validation, stratified folds, and grouped folds.

Usage

```
resample_nested_cv(
  data,
  formula,
  n_outer_folds = 5L,
  n_inner_folds = 5L,
  n_iters = 1L,
  strata = NULL,
  group = NULL
)
```

Arguments

data	(data.frame) Data set to resample. It must have at least one column and at least two rows.
formula	(formula) Two-sided model formula. The left-hand side must be a single untransformed column name in data. Calls such as cbind(y1, y2), Surv(time, status), log(y), and factor(y) are rejected. The right-hand side is ignored when the resampling indices are created.
n_outer_folds	(numeric scalar: 5L; whole number in [2, unit_n]) Number of outer folds. unit_n is nrow(data) when group is NULL and the number of unique groups otherwise. Whole-number numeric values such as 5 and 5.0 are accepted and stored as integers.
n_inner_folds	(numeric scalar: 5L; whole number in [2, unit_n - ceiling(unit_n / n_outer_folds)]) Number of inner folds. The upper bound ensures that every outer-analysis set has enough sampling units for inner cross-validation. Whole-number numeric values such as 5 and 5.0 are accepted and stored as integers.
n_iters	(numeric scalar: 1L; whole number in [1, .Machine$integer.max]) Number of independent outer K-fold partitions. Use values greater than one for repeated nested cross-validation.
strata	(formula: NULL) Stratification specification. Use NULL for no stratification. Use ~ 1 to stratify by the model outcome. Use a one-sided formula such as ~ a + b to stratify by the interaction of columns a and b.
group	(formula: NULL) Grouping specification. Use NULL for row-level resampling. Use a one-sided formula such as ~ subject to keep rows with the same group value together. The right-hand side must be a single bare column name. Create composite group keys in data before calling resample_nested_cv().

Details

resample_nested_cv() does not copy data. It returns row indices that can be used to subset data later. The predictor side of formula is not used to build the splits. It is accepted so the same formula can be passed to a model-fitting function.

Splits:

Each outer split separates the original data rows into:

- analysis: integer row indices used for model selection and final model fitting within that outer split. These are all rows not in the outer assessment. They are sorted in ascending order.
- assessment: integer row indices held out for final assessment of that outer split. They are sorted in ascending order.
- iter: integer iteration number.
- outer_fold: integer outer-fold number within iter.
- seed: single positive integer for reproducible model fits on this split.

- `inner`: a named list of inner cross-validation splits. Inner splits are created only from the outer `analysis` rows.

Each inner split contains:

- `analysis`: integer row indices used to fit a candidate model. They are restricted to the parent outer `analysis` set. They are sorted in ascending order.
- `assessment`: integer row indices used to compare candidate models. They are restricted to the parent outer `analysis` set. They are sorted in ascending order.
- `iter`: integer iteration number inherited from the parent outer split.
- `outer_fold`: integer outer-fold number inherited from the parent outer split.
- `inner_fold`: integer inner-fold number.
- `seed`: single positive integer for reproducible candidate model fits on this inner analysis set.

Within each iteration, the outer `assessment` indices partition `seq_len(nrow(data))`. Within each outer split, the inner `assessment` indices partition the parent outer `analysis` indices.

Fold assignment:

Outer and inner folds use the same fold-assignment rules as `resample_cv()`. Rows are the sampling units when `group` is `NULL`. Groups are the sampling units when `group` is supplied.

Without stratification, fold sizes are balanced over the sampling units. When `strata` is supplied, strata are processed from smallest to largest, so rare strata are allocated first. Within each stratum, sampling units are shuffled and distributed across folds as evenly as possible. Per-stratum fold counts differ by at most one. Ties are broken with the current random-number generator state.

Stratification:

Stratification is optional. Pass `NULL` for no stratification. `strata = ~ 1` stratifies by the model outcome. A formula such as `strata = ~ a + b` stratifies by `interaction(data$a, data$b, drop = TRUE)`. Variables referenced by `strata` must be columns of `data`. `NA` stratum values are not allowed. Empty factor levels are dropped. Stratification variables are treated as categorical. Pre-bin continuous variables before passing them to `strata`. This also applies when `strata = ~ 1` and the model outcome is continuous.

Expressions in `strata` are not evaluated as transformations. Create transformed or composite strata columns in `data` before calling `resample_nested_cv()`.

Grouping:

Grouping is optional. Pass `NULL` for ordinary row-level resampling. Pass a one-sided formula such as `group = ~ subject` to keep all rows from each subject together. The right-hand side of `group` must be a single bare column name in `data`. `NA` group values are not allowed. Empty factor levels are dropped. Rows that share a group value are kept together in both outer and inner splits. No group is split between `analysis` and `assessment`.

Because groups are the sampling units, `n_outer_folds` must be less than or equal to the number of unique groups. `n_inner_folds` must be no larger than the smallest possible outer-analysis sampling-unit count. This upper bound is `unit_n - ceiling(unit_n / n_outer_folds)`. Fold balance is defined over groups, not rows. When group sizes differ, row counts across folds may differ by more than one.

When `strata` and `group` are both supplied, the resolved stratum vector must be constant within each group. If `strata = ~ 1` is used with `group`, the outcome must be constant within each group.

Repeated nested K-fold:

When `n_iters > 1`, the function repeats the outer K-fold partitioning. Inner partitioning is generated separately for each outer split. The total number of outer splits is `n_outer_folds * n_iters`. Each iteration advances the current random-number generator state.

Split Names:

Outer split names are generated from the total number of outer splits. For fewer than 10 outer splits, names are `split1`, ..., `split9`. For 10 or more outer splits, names are zero-padded so lexical order matches numeric order. Inner split names are generated separately within each outer split from `n_inner_folds`.

Reproducibility:

Call `base::set.seed()` immediately before `resample_nested_cv()` for reproducible splits.

Split indices reproduce the fold rows, but a downstream model fit that uses randomness depends on the global random-number state when it runs, so it does not reproduce on its own. Each outer split carries a `seed` for its final refit, and each inner split carries a `seed` for its candidate sweep. Set the relevant seed immediately before fitting to make the fit reproducible regardless of run order.

```
# Inner candidate sweep on inner analysis set j of outer split i.
inner <- ncv$splits[[i]]$inner[[j]]
set.seed(inner$seed)
candidate_seeds <- sample.int(.Machine$integer.max, K, replace = FALSE)
for (k in seq_len(K)) {
  set.seed(candidate_seeds[k])
  fit_k <- fit_candidate_k(inner$analysis)
}

# Final refit on outer analysis set i.
set.seed(ncv$splits[[i]]$seed)
final <- fit_model(ncv$splits[[i]]$analysis)
```

Seeds are created from the global generator and the generator state is then restored, so split indices and the caller's random-number stream are unchanged. Reproducing a fit assumes the same `base::RNGkind()` on each run.

Value

An S3 list of class `c("nested_cv_resamples", "resamples")` with elements:

- `call`: the matched call.
- `method`: `"nested_cv"`.
- `n`: `nrow(data)`.
- `formula`: the input `formula`.
- `outcome_var`: the resolved outcome column name.
- `strata`: the input `strata` formula or `NULL`.
- `strata_vars`: the resolved strata column names or `NULL`.
- `group`: the input `group` formula or `NULL`.

- `group_vars`: the resolved group column name or `NULL`.
- `n_groups`: the number of unique groups when `group` is supplied, otherwise `NULL`.
- `n_outer_folds`: the resolved outer-fold count.
- `n_inner_folds`: the resolved inner-fold count.
- `n_iters`: the resolved iteration count.
- `splits`: a named list of outer split entries. Each outer split contains `analysis`, `assessment`, `iter`, `outer_fold`, `seed`, and `inner`. Each inner split contains `analysis`, `assessment`, `iter`, `outer_fold`, `inner_fold`, and `seed`.

See Also

`resample_cv()`, `resample_bootstrap()`, `base::set.seed()`

Examples

```
#--------------------------------------------------------------------------
# resample_nested_cv() examples
#--------------------------------------------------------------------------
library(bkmodel)

set.seed(1L)

data <- data.frame(
  outcome = c(rep("event", 12L), rep("none", 12L)),
  age = c(34L, 52L, 41L, 63L, 29L, 48L, 37L, 55L,
          46L, 31L, 60L, 43L, 39L, 50L, 28L, 44L,
          57L, 36L, 62L, 33L, 47L, 53L, 40L, 58L),
  clinic = rep(c("north", "south", "west"), each = 8L),
  patient = rep(seq_len(12L), each = 2L)
)

ncv <- resample_nested_cv(
  data,
  outcome ~ age,
  n_outer_folds = 4L,
  n_inner_folds = 3L
)
ncv$splits[[1L]]$assessment
ncv$splits[[1L]]$inner[[1L]]$assessment

# Keep outcome classes balanced in outer and inner folds.
resample_nested_cv(
  data,
  outcome ~ age,
  n_outer_folds = 4L,
  n_inner_folds = 3L,
  strata = ~ outcome
)

# Keep repeated observations from the same patient together.
```

```
resample_nested_cv(
  data,
  outcome ~ age,
  n_outer_folds = 4L,
  n_inner_folds = 3L,
  group = ~ patient
)
```

`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 duplicate `(z, x)` cells.

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 specifying outcome (`y`), group (`x`), and block (`z`) columns. Must have the form `y ~ x | z`.
`agg_fun`	(scalar character or function: `"error"`) Aggregator for duplicate `(z, x)` cells. One of `"error"`, `"first"`, `"last"`, `"sum"`, `"mean"`, `"median"`, `"min"`, `"max"`, or a user-supplied function applied per cell. `"error"` (default) raises an error on duplicate cells.
`drop_unused_levels`	(scalar logical: `FALSE`) Drop unobserved levels of `x` and `z` after the NA filter so they do not appear as all-`NA` columns or rows in the output.
`y`	(scalar character) The name of the numeric outcome column in `data`.
`x`	(scalar character) The name of the factor grouping column in `data`.
`z`	(scalar character) The name of the factor 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` are not factors, they are coerced via `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 drop factor levels with no observations after the NA filter so they do not appear in the output.

Cell aggregation:

- A cell with no observations becomes `NA_real_`.
- A cell with one observation returns that value (which may itself be `NA`).
- A cell with two or more observations requires `agg_fun` to be set to something other than `"error"`, otherwise an error is raised.

User-supplied `agg_fun`:

- Called once per non-empty cell with a length >= 1 numeric vector, which may contain `NA`.
- Must return a single numeric value.

NA handling:

- Rows with `NA` in `x` or `z` are dropped before aggregation.
- Rows with `NA` in `y` are retained and reach the aggregator.
- Built-in aggregators apply `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` produced by an aggregator (`Inf + (-Inf)` in `sum`, `(-Inf + Inf) / 2` in `median`, `0 / 0` in `mean`, or returned by a user function) is normalized to `NA_real_`.

The operation is analogous to:

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

Value

A data.frame with `nlevels(z)` rows and `nlevels(x) + 1` columns.

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

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")

# Equivalent base R interface
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