Package: designit (via r-universe)

September 18, 2024

Title Blocking and Randomization for Experimental Design

Version 0.5.0.9000

Description Intelligently assign samples to batches in order to reduce batch effects. Batch effects can have a significant impact on data analysis, especially when the assignment of samples to batches coincides with the contrast groups being studied. By defining a batch container and a scoring function that reflects the contrasts, this package allows users to assign samples in a way that minimizes the potential impact of batch effects on the comparison of interest. Among other functionality, we provide an implementation for OSAT score by Yan et al. (2012, <doi:10.1186/1471-2164-13-689>).

License MIT + file LICENSE

URL `https://bedapub.github.io/designit/`, `https://github.com/BEDApub/designit/`

BugReports `https://github.com/BEDApub/designit/issues`

Depends R (>= 4.1.0)

Imports rlang (>= 0.4.0), dplyr (>= 1.0.0), purrr, ggplot2, scales, tibble, tidyr, assertthat, stringr, R6, data.table, stats

Suggests testthat, roxygen2, pkgdown, knitr, markdown, rmarkdown, gt, bench, OSAT, tidyverse, printr, devtools (>= 2.0.0), ggpattern, cowplot, bestNormalize, here

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

VignetteBuilder knitr

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

RemoteUrl https://github.com/bedapub/designit

RemoteRef HEAD

RemoteSha 61dbbbe73c7b952cfcda4798504702248f95a50c

Contents

`accept_leftmost_improvement` *Alternative acceptance function for multi-dimensional scores in which order (left to right, e.g. first to last) denotes relevance.*

Description

Alternative acceptance function for multi-dimensional scores in which order (left to right, e.g. first to last) denotes relevance.

Usage

```
accept_leftmost_improvement(current_score, best_score, ..., tolerance = 0)
```

Arguments

`current_score`	One- or multi-dimensional score from the current optimizing iteration (double or vector of doubles)
`best_score`	Best one- or multi-dimensional score found so far (double or vector of doubles)
`...`	Ignored arguments that may be used by alternative acceptance functions
`tolerance`	Tolerance value: When comparing score vectors from left to right, differences within +/- tol won't immediately shortcut the comparison at this point, allowing improvement in a less important score to exhibit some influence

Value

Boolean, TRUE if current score should be taken as the new optimal score, FALSE otherwise

`assign_from_table` *Distributes samples based on a sample sheet.*

Description

Distributes samples based on a sample sheet.

Usage

```
assign_from_table(batch_container, samples)
```

Arguments

`batch_container`	Instance of BatchContainer class
`samples`	`data.frame` with samples (a sample sheet). This `data.frame` (or `tibble::tibble()`) should contain samples together with their locations. No `.sample_id` column can be present in the sample sheet. In `batch_container` already has samples assigned, the function will check if samples in `batch_container` are identical to the ones in the `samples` argument.

Value

Returns a new `BatchContainer`.

Examples

```
bc <- BatchContainer$new(
  dimensions = list(
    plate = 2,
    column = list(values = letters[1:3]),
    row = 3
  )
)

sample_sheet <- tibble::tribble(
  ~plate, ~column, ~row, ~sampleID, ~group,
  1, "a", 1, 1, "TRT",
  1, "b", 2, 2, "CNTRL",
  2, "a", 1, 3, "TRT",
  2, "b", 2, 4, "CNTRL",
  2, "a", 3, 5, "TRT",
)
# assign samples from the sample sheet
bc <- assign_from_table(bc, sample_sheet)

bc$get_samples(remove_empty_locations = TRUE)
```

`assign_in_order` *Distributes samples in order.*

Description

First sample is assigned to the first location, second sample is assigned to the second location, etc.

Usage

```
assign_in_order(batch_container, samples = NULL)
```

Arguments

`batch_container`	Instance of BatchContainer class
`samples`	data.frame with samples.

Value

Returns a new `BatchContainer`.

Examples

```
samples <- data.frame(sampId = 1:3, sampName = letters[1:3])
samples

bc <- BatchContainer$new(dimensions = c("row" = 3, "column" = 2))
bc

set.seed(42)
# assigns samples randomly
bc <- assign_random(bc, samples)
bc$get_samples()

# assigns samples in order
bc <- assign_in_order(bc)
bc$get_samples()
```

`assign_random` *Assignment function which distributes samples randomly.*

Description

Assignment function which distributes samples randomly.

Usage

```
assign_random(batch_container, samples = NULL)
```

Arguments

`batch_container`	Instance of BatchContainer class
`samples`	data.frame with samples.

Value

Returns a new `BatchContainer`.

Examples

```
samples <- data.frame(sampId = 1:3, sampName = letters[1:3])
samples

bc <- BatchContainer$new(dimensions = c("row" = 3, "column" = 2))
bc

set.seed(42)
# assigns samples randomly
bc <- assign_random(bc, samples)
bc$get_samples()

# assigns samples in order
bc <- assign_in_order(bc)
bc$get_samples()
```

`BatchContainer` *R6 Class representing a batch container.*

Description

Describes container dimensions and samples to container location assignment.

Details

A typical workflow starts with creating a `BatchContainer`. Then samples can be assigned to locations in that container.

Public fields

`trace` Optimization trace, a `tibble::tibble()`

Active bindings

`scoring_f` Scoring functions used for optimization. Each scoring function should receive a BatchContainer. This function should return a floating point score value for the assignment. This a list of functions. Upon assignment a single function will be automatically converted to a list In the later case each function is called.

`has_samples` Returns TRUE if `BatchContainer` has samples.

`has_samples_attr` Returns TRUE if `BatchContainer` has sample atrributes assigned.

`n_locations` Returns number of locations in a `BatchContainer`.

`n_dimensions` Returns number of dimensions in a `BatchContainer`. This field cannot be assigned.

`dimension_names` character vector with dimension names. This field cannot be assigned.

`samples` Samples in the batch container. When assigning data.frame should not have column named .sample_id column.

`samples_attr` Extra attributes of samples. If set, this is included into `BatchContainer$get_samples()` output.

`assignment` Sample assignment vector. Should contain NAs for empty locations.

Assigning this field is deprecated, please use `$move_samples()` instead.

Methods

Public methods:

- `BatchContainer$new()`
- `BatchContainer$get_samples()`
- `BatchContainer$get_locations()`
- `BatchContainer$move_samples()`
- `BatchContainer$score()`
- `BatchContainer$copy()`
- `BatchContainer$print()`
- `BatchContainer$scores_table()`
- `BatchContainer$plot_trace()`

Method `new()`**:** Create a new BatchContainer object.

Usage:

```
BatchContainer$new(locations_table, dimensions, exclude = NULL)
```

Arguments:

`locations_table` A table with available locations.

`dimensions` A vector or list of dimensions. Every dimension should have a name. Could be an integer vector of dimensions or a named list. Every value of a list could be either dimension size or parameters for BatchContainerDimension$new(). Can be used as an alternative to passing `locations_table`.

`exclude` data.frame with excluded locations of a container. Only used together with dimensions.

Examples:

```
bc <- BatchContainer$new(
  dimensions = list(
    "plate" = 3,
    "row" = list(values = letters[1:3]),
    "column" = list(values = c(1, 3))
  ),
  exclude = data.frame(plate = 1, row = "a", column = c(1, 3), stringsAsFactors = FALSE)
)

bc
```

Method `get_samples()`**:** Return table with samples and sample assignment.

Usage:

```
BatchContainer$get_samples(
  assignment = TRUE,
  include_id = FALSE,
  remove_empty_locations = FALSE,
  as_tibble = TRUE
)
```

Arguments:

`assignment` Return sample assignment. If FALSE, only samples table is returned, with out batch assignment.

`include_id` Keep .sample_id in the table. Use `TRUE` for lower overhead.

`remove_empty_locations` Removes empty locations from the result tibble.

`as_tibble` Return `tibble`. If `FALSE` returns `data.table`. This should have lower overhead, as internally there is a cached `data.table`.

Returns: table with samples and sample assignment.

Method `get_locations()`**:** Get a table with all the locations in a `BatchContainer`.

Usage:

```
BatchContainer$get_locations()
```

Returns: A `tibble` with all the available locations.

Method `move_samples()`**:** Move samples between locations

This method can receive either `src` and `dst` or `locations_assignment`.

Usage:

```
BatchContainer$move_samples(src, dst, location_assignment)
```

Arguments:

`src` integer vector of source locations

`dst` integer vector of destination locations (the same length as `src`).

`location_assignment` integer vector with location assignment. The length of the vector should match the number of locations, `NA` should be used for empty locations.

Returns: `BatchContainer`, invisibly

Method `score()`**:** Score current sample assignment,

Usage:

```
BatchContainer$score(scoring)
```

Arguments:

`scoring` a function or a names list of scoring functions. Each function should return a numeric vector.

Returns: Returns a named vector of all scoring functions values.

Method `copy()`**:** Create an independent copy (clone) of a `BatchContainer`

Usage:

```
BatchContainer$copy()
```

Returns: Returns a new `BatchContainer`

Method `print()`**:** Prints information about `BatchContainer`.

Usage:

`BatchContainer$print(...)`

Arguments:

`...` not used.

Method `scores_table()`**:** Return a table with scores from an optimization.

Usage:

`BatchContainer$scores_table(index = NULL, include_aggregated = FALSE)`

Arguments:

`index` optimization index, all by default

`include_aggregated` include aggregated scores

Returns: a `tibble::tibble()` with scores

Method `plot_trace()`**:** Plot trace

Usage:

`BatchContainer$plot_trace(index = NULL, include_aggregated = FALSE, ...)`

Arguments:

`index` optimization index, all by default

`include_aggregated` include aggregated scores

`...` not used.

Returns: a `ggplot2::ggplot()` object List of scoring functions. Tibble with batch container locations. Tibble with sample information and sample ids. Sample attributes, a data.table. Vector with assignment of sample ids to locations. Cached data.table with samples assignment. Validate sample assignment.

Examples

```
## ------------------------------------------------
## Method `BatchContainer$new`
## ------------------------------------------------

bc <- BatchContainer$new(
  dimensions = list(
    "plate" = 3,
    "row" = list(values = letters[1:3]),
    "column" = list(values = c(1, 3))
  ),
  exclude = data.frame(plate = 1, row = "a", column = c(1, 3), stringsAsFactors = FALSE)
)

bc
```

`BatchContainerDimension` *R6 Class representing a batch container dimension.*

Description

R6 Class representing a batch container dimension.

R6 Class representing a batch container dimension.

Public fields

`name` dimension name.

`values` vector of dimension values.

Active bindings

`size` Returns size of a dimension.

`short_info` Returns a string summarizing the dimension. E.g., "mydim<size=10>".

Methods

Public methods:

- `BatchContainerDimension$new()`
- `BatchContainerDimension$clone()`

Method `new()`**:** Create a new BatchContainerDimension object.
This is usually used implicitly via `BatchContainer$new()`.

Usage:

```
BatchContainerDimension$new(name, size = NULL, values = NULL)
```

Arguments:

`name` Dimension name, a character string. Required.

`size` Dimension size. Setting this implies that dimension values are 1:`size`.

`values` Explicit list of dimension values. Could be numeric, character or factor.
It is required to provide dimension namd and either size of values.

Examples:

```
plate_dimension <- BatchContainerDimension$new("plate", size=3)
row_dimension <- BatchContainerDimension$new("row", values = letters[1:3])
column_dimension <- BatchContainerDimension$new("column", values = 1:3)

bc <- BatchContainer$new(
  dimensions = list(plate_dimension, row_dimension, column_dimension),
 exclude = data.frame(plate = 1, row = "a", column = c(1, 3), stringsAsFactors = FALSE)
)

bc
```

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

Usage:

`BatchContainerDimension$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `BatchContainerDimension$new`
## ------------------------------------------------

plate_dimension <- BatchContainerDimension$new("plate", size=3)
row_dimension <- BatchContainerDimension$new("row", values = letters[1:3])
column_dimension <- BatchContainerDimension$new("column", values = 1:3)

bc <- BatchContainer$new(
  dimensions = list(plate_dimension, row_dimension, column_dimension),
  exclude = data.frame(plate = 1, row = "a", column = c(1, 3), stringsAsFactors = FALSE)
)

bc
```

`batch_container_from_table`

Creates a BatchContainer from a table (data.frame/tibble::tibble) containing sample and location information.

Description

Creates a BatchContainer from a table (data.frame/tibble::tibble) containing sample and location information.

Usage

```
batch_container_from_table(tab, location_cols)
```

Arguments

`tab`	A table with location and sample information. Table rows with all `NA`s in sample information columns are treated as empty locations.
`location_cols`	Names of columns containing information about locations.

Value

A BatchContainer assigned samples.

Examples

```
tab <- data.frame(
  row = rep(1:3, each = 3),
  column = rep(1:3, 3),
  sample_id = c(1, 2, 3, NA, 5, 6, 7, NA, 9)
)
bc <- batch_container_from_table(tab, location_cols = c("row", "column"))
```

---

compile_possible_subgroup_allocation

Compile list of all possible ways to assign levels of the allocation variable to a given set of subgroups

---

Description

All information needed to perform this function (primarily the number and size of subgroups plus the levels of the allocation variable) are contained in and extracted from the subgroup object.

Usage

```
compile_possible_subgroup_allocation(
  subgroup_object,
  fullTree = FALSE,
  maxCalls = 1e+06
)
```

Arguments

`subgroup_object`	A subgrouping object as returned by `form_homogeneous_subgroups()`
`fullTree`	Boolean: Enforce full search of the possibility tree, independent of the value of `maxCalls`
`maxCalls`	Maximum number of recursive calls in the search tree, to avoid long run times with very large trees

Value

List of possible allocations; Each allocation is an integer vector of allocation levels that are assigned in that order to the subgroups with given sizes

`complete_random_shuffling` *Reshuffle sample indices completely randomly*

Description

This function was just added to test early on the functionality of optimize_design() to accept a permutation vector rather than a list with src and dst indices.

Usage

```
complete_random_shuffling(batch_container, ...)
```

Arguments

`batch_container`	The batch-container.
`...`	Other params that are passed to a generic shuffling function (like the iteration number).

Value

A random permutation of the sample assignment in the container.

Examples

```
data("invivo_study_samples")
bc <- BatchContainer$new(
  dimensions = c("plate" = 2, "column" = 5, "row" = 6)
)
scoring_f <- osat_score_generator("plate", "Sex")
bc <- optimize_design(
  bc, scoring = scoring_f, invivo_study_samples,
  max_iter = 100,
  shuffle_proposal_func = complete_random_shuffling
)
```

`drop_order` *Drop highest order interactions*

Description

Drop highest order interactions

Usage

```
drop_order(.terms, m = -1)
```

Arguments

`.terms`	`terms.object`
`m`	order of interaction (highest available if -1)

`first_score_only` *Aggregation of scores: take first (primary) score only*

Description

This function enables comparison of the results of two scoring functions by just basing the decision on the first element. This reflects the original behavior of the optimization function, just evaluating the 'auxiliary' scores for the user's information.

Usage

```
first_score_only(scores, ...)
```

Arguments

`scores`	A score or multiple component score vector
`...`	Parameters to be ignored by this aggregation function

Value

The aggregated score, i.e. the first element of a multiple-component score vector.

Examples

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

`form_homogeneous_subgroups` *Form groups and subgroups of 'homogeneous' samples as defined by certain variables and size constraints*

Description

Form groups and subgroups of 'homogeneous' samples as defined by certain variables and size constraints

Usage

```
form_homogeneous_subgroups(
  batch_container,
  allocate_var,
  keep_together_vars = c(),
  n_min = NA,
  n_max = NA,
  n_ideal = NA,
  subgroup_var_name = NULL,
  prefer_big_groups = TRUE,
  strict = TRUE
)
```

Arguments

`batch_container`	Batch container with all samples assigned that are to be grouped and sub-grouped
`allocate_var`	Name of a variable in the `samples` table to inform possible groupings, as (sub)group sizes must add up to the correct totals
`keep_together_vars`	Vector of column names in sample table; groups are formed by pooling samples with identical values of all those variables
`n_min`	Minimal number of samples in one sub(!)group; by default 1
`n_max`	Maximal number of samples in one sub(!)group; by default the size of the biggest group
`n_ideal`	Ideal number of samples in one sub(!)group; by default the floor or ceiling of `mean(n_min,n_max)`, depending on the setting of `prefer_big_groups`
`subgroup_var_name`	An optional column name for the subgroups which are formed (or NULL)
`prefer_big_groups`	Boolean; indicating whether or not bigger subgroups should be preferred in case of several possibilities
`strict`	Boolean; if TRUE, subgroup size constraints have to be met strictly, implying the possibility of finding no solution at all

Value

Subgroup object to be used in subsequent calls to `compile_possible_subgroup_allocation()`

generate_terms *Generate `terms.object` (formula with attributes)*

Description

Generate `terms.object` (formula with attributes)

Usage

```
generate_terms(.tbl, ...)
```

Arguments

`.tbl`	data
`...`	columns to skip (unquoted)

Value

`terms.object`

get_order *Get highest order interaction*

Description

Get highest order interaction

Usage

```
get_order(.terms)
```

Arguments

`.terms`	`terms.object`

Value

highest order (numeric).

`invivo_study_samples` *A sample list from an in vivo experiment with multiple treatments and 2 strains*

Description

This sample list is intended to be used in connection with the `"invivo_study_treatments"` data object

Usage

```
data(invivo_study_samples)
```

Format

An object of class `"tibble"`

AnimalID The animal IDs, i.e. unique identifiers for each animal

Strain Strain (A or B)

Sex Female (F) or Male (M)

BirthDate Date of birth, not available for all the animals

Earmark Markings to distinguish individual animals, applied on the left (L), right (R) or both(B) ears

ArrivalWeight Initial body weight of the animal

Arrival weight Unit Unit of the body weight, here: grams

Litter The litter IDs, grouping offspring from one set of parents

Author(s)

Guido Steiner

`invivo_study_treatments` *A treatment list together with additional constraints on the strain and sex of animals*

Description

This treatment list is intended to be used in connection with the `"invivo_study_samples"` data object

Usage

```
data(invivo_study_treatments)
```

Format

An object of class `"tibble"`

Treatment The treatment to be given to an individual animal (1-3, plus a few untreated cases)

Strain Strain (A or B) - a constraint which kind of animal may receive the respective treatment

Sex Female (F) or Male (M) - a constraint which kind of animal may receive the respective treatment

Author(s)

Guido Steiner

`L1_norm` *Aggregation of scores: L1 norm*

Description

This function enables comparison of the results of two scoring functions by calculating an L1 norm (Manhattan distance from origin).

Usage

```
L1_norm(scores, ...)
```

Arguments

`scores`	A score or multiple component score vector
`...`	Parameters to be ignored by this aggregation function

Value

The L1 norm as an aggregated score.

Examples

```
L1_norm(c(2, 2))
```

`L2s_norm` *Aggregation of scores: L2 norm squared*

Description

This function enables comparison of the results of two scoring functions by calculating an L2 norm (euclidean distance from origin). Since this is only used for ranking solutions, the squared L2 norm is returned.

Usage

```
L2s_norm(scores, ...)
```

Arguments

`scores`	A score or multiple component score vector
`...`	Parameters to be ignored by this aggregation function

Value

The squared L2 norm as an aggregated score.

Examples

```
L2s_norm(c(2, 2))
```

`locations_table_from_dimensions` *Create locations table from dimensions and exclude table*

Description

Create locations table from dimensions and exclude table

Usage

```
locations_table_from_dimensions(dimensions, exclude)
```

Arguments

`dimensions`	A vector or list of dimensions. Every dimension should have a name. Could be an integer vector of dimensions or a named list. Every value of a list could be either dimension size or parameters for BatchContainerDimension$new().
`exclude`	data.frame with excluded locations of a container.

Value

a `tibble::tibble()` with all the available locations.

`longitudinal_subject_samples` *Subject sample list with group and time plus controls*

Description

A sample list with 9 columns as described below. There are 3 types of records (rows) indicated by the `SampleType` variable. Patient samples, controls and spike-in standards. Patient samples were collected over up to 7 time points. Controls and SpikeIns are QC samples for distribution of the samples on 96 well plates.

Usage

```
data(longitudinal_subject_samples)
```

Format

An object of class `"tibble"`

SampleID A unique sample identifier.

SampleType Indicates whether the sample is a patient sample, control oder spike-in.

SubjectID The subject identifier.

Group Indicates the treatment group of a subject.

Week Sampling time points in weeks of study.

Sex Subject Sex, Female (F) or Male (M).

Age Subject age.

BMI Subject Body Mass Index.

SamplesPerSubject Look up variable for the number of samples per subject. This varies as not subject have samples from all weeks.

Author(s)

Juliane Siebourg

`mk_exponentially_weighted_acceptance_func`

Alternative acceptance function for multi-dimensional scores with exponentially downweighted score improvements from left to right

Description

Alternative acceptance function for multi-dimensional scores with exponentially downweighted score improvements from left to right

Usage

```
mk_exponentially_weighted_acceptance_func(
  kappa = 0.5,
  simulated_annealing = FALSE,
  temp_function = mk_simanneal_temp_func(T0 = 500, alpha = 0.8)
)
```

Arguments

`kappa`	Coefficient that determines how quickly the weights for the individual score improvements drop when going from left to right (i.e. first to last score). Weight for the first score's delta is 1, then the original delta multiplied with kappa^(p-1) for the p'th score
`simulated_annealing`	Boolean; if TRUE, simulated annealing (SA) will be used to minimize the weighted improved score
`temp_function`	In case SA is used, a temperature function that returns the annealing temperature for a certain iteration number

Value

Acceptance function which returns TRUE if current score should be taken as the new optimal score, FALSE otherwise

`mk_plate_scoring_functions`

Create a list of scoring functions (one per plate) that quantify the spatially homogeneous distribution of conditions across the plate

Description

Create a list of scoring functions (one per plate) that quantify the spatially homogeneous distribution of conditions across the plate

Usage

```
mk_plate_scoring_functions(
  batch_container,
  plate = NULL,
  row,
  column,
  group,
  p = 2,
  penalize_lines = "soft"
)
```

Arguments

`batch_container`	Batch container (bc) with all columns that denote plate related information
`plate`	Name of the bc column that holds the plate identifier (may be missing or NULL in case just one plate is used)
`row`	Name of the bc column that holds the plate row number (integer values starting at 1)
`column`	Name of the bc column that holds the plate column number (integer values starting at 1)
`group`	Name of the bc column that denotes a group/condition that should be distributed on the plate
`p`	p parameter for minkowski type of distance metrics. Special cases: p=1 - Manhattan distance; p=2 - Euclidean distance
`penalize_lines`	How to penalize samples of the same group in one row or column of the plate. Valid options are: 'none' - there is no penalty and the pure distance metric counts, 'soft' - penalty will depend on the well distance within the shared plate row or column, 'hard' - samples in the same row/column will score a zero distance

Value

List of scoring functions, one per plate, that calculate a real valued measure for the quality of the group distribution (the lower the better).

Examples

```
data("invivo_study_samples")
bc <- BatchContainer$new(
  dimensions = c("column" = 6, "row" = 10)
)
bc <- assign_random(bc, invivo_study_samples)
scoring_f <- mk_plate_scoring_functions(
  bc,
  row = "row", column = "column", group = "Sex"
)
bc <- optimize_design(bc, scoring = scoring_f, max_iter = 100)
```

```
plot_plate(bc$get_samples(), .col = Sex)
```

---

mk_simanneal_acceptance_func *Generate acceptance function for an optimization protocol based on simulated annealing*

Description

Generate acceptance function for an optimization protocol based on simulated annealing

Usage

```
mk_simanneal_acceptance_func(
  temp_function = mk_simanneal_temp_func(T0 = 500, alpha = 0.8)
)
```

Arguments

`temp_function`	A temperature function that returns the annealing temperature for a certain cycle k

Value

A function that takes parameters (`current_score`, `best_score`, `iteration`) for an optimization step and return a Boolean indicating whether the current solution should be accepted or dismissed. Acceptance probability of a worse solution decreases with annealing temperature.

---

mk_simanneal_temp_func *Create a temperature function that returns the annealing temperature at a given step (iteration)*

Description

Supported annealing types are currently "Exponential multiplicative", "Logarithmic multiplicative", "Quadratic multiplicative" and "Linear multiplicative", each with dedicated constraints on alpha. For information, see http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/

Usage

```
mk_simanneal_temp_func(T0, alpha, type = "Quadratic multiplicative")
```

Arguments

`T0`	Initial temperature at step 1 (when k=0)
`alpha`	Rate of cooling
`type`	Type of annealing protocol. Defaults to the quadratic multiplicative method which seems to perform well.

Value

Temperature at cycle `k`.

`mk_subgroup_shuffling_function` *Created a shuffling function that permutes samples within certain subgroups of the container locations*

Description

If length(n_swaps)==1, the returned function may be called an arbitrary number of times. If length(n_swaps)>1 the returned function may be called length(n_swaps) timed before returning NULL, which would be the stopping criterion if all requested swaps have been exhausted.

Usage

```
mk_subgroup_shuffling_function(
  subgroup_vars,
  restrain_on_subgroup_levels = c(),
  n_swaps = 1
)
```

Arguments

`subgroup_vars`	Column names of the variables that together define the relevant subgroups
`restrain_on_subgroup_levels`	Permutations can be forced to take place only within a level of the factor of the subgrouping variable. In this case, the user must pass only one subgrouping variable and a number of levels that together define the permuted subgroup.
`n_swaps`	Vector with number of swaps to be proposed in successive calls to the returned function (each value should be in valid range from 1..floor(n_locations/2))

Value

Function to return a list with length `n` vectors `src` and `dst`, denoting source and destination index for the swap operation, or `NULL` if the user provided a defined protocol for the number of swaps and the last iteration has been reached

Examples

```
set.seed(42)

bc <- BatchContainer$new(
  dimensions = c(
    plate = 2,
    row = 4, col = 4
  )
)

bc <- assign_in_order(bc, samples = tibble::tibble(
  Group = c(rep(c("Grp 1", "Grp 2", "Grp 3", "Grp 4"), each = 8)),
  ID = 1:32
))

# here we use a 2-step approach:
# 1. Assign samples to plates.
# 2. Arrange samples within plates.

# overview of sample assagnment before optimization
plot_plate(bc,
  plate = plate, row = row, column = col, .color = Group
)

# Step 1, assign samples to plates
scoring_f <- osat_score_generator(
  batch_vars = c("plate"), feature_vars = c("Group")
)
bc <- optimize_design(
  bc,
  scoring = scoring_f,
  max_iter = 10, # the real number of iterations should be bigger
  n_shuffle = 2,
  quiet = TRUE
)
plot_plate(
  bc,
  plate = plate, row = row, column = col, .color = Group
)

# Step 2, distribute samples within plates
scoring_f <- mk_plate_scoring_functions(
  bc,
  plate = "plate", row = "row", column = "col", group = "Group"
)
bc <- optimize_design(
  bc,
  scoring = scoring_f,
  max_iter = 50,
  shuffle_proposal_func = mk_subgroup_shuffling_function(subgroup_vars = c("plate")),
  aggregate_scores_func = L2s_norm,
  quiet = TRUE
)
plot_plate(bc,
  plate = plate, row = row, column = col, .color = Group
)
```

`mk_swapping_function` *Create function to propose swaps of samples on each call, either with a constant number of swaps or following a user defined protocol*

Description

If `length(n_swaps)==1`, the returned function may be called an arbitrary number of times. If `length(n_swaps)>1` and called without argument, the returned function may be called length(n_swaps) timed before returning NULL, which would be the stopping criterion if all requested swaps have been exhausted. Alternatively, the function may be called with an iteration number as the only argument, giving the user some freedom how to iterate over the sample swapping protocol.

Usage

```
mk_swapping_function(n_swaps = 1)
```

Arguments

`n_swaps`	Vector with number of swaps to be proposed in successive calls to the returned function (each value should be in valid range from 1..`floor(n_samples/2)`)

Value

Function to return a list with length n vectors `src` and `dst`, denoting source and destination index for the swap operation, or NULL if the user provided a defined protocol for the number of swaps and the last iteration has been reached.

Examples

```
data("invivo_study_samples")
bc <- BatchContainer$new(
  dimensions = c("plate" = 2, "column" = 5, "row" = 6)
)
scoring_f <- osat_score_generator("plate", "Sex")
optimize_design(
  bc, scoring = scoring_f, invivo_study_samples,
  max_iter = 100,
  shuffle_proposal_func = mk_swapping_function(1)
)
```

`multi_trt_day_samples` *Unbalanced treatment and time sample list*

Description

A sample list with 4 columns SampleName, Well, Time and Treatment Not all treatments are available at all time points. All samples are placed on the same plate.

Usage

```
data(multi_trt_day_samples)
```

Format

An object of class `"tibble"`

Author(s)

siebourj

`optimize_design` *Generic optimizer that can be customized by user provided functions for generating shuffles and progressing towards the minimal score*

Description

Generic optimizer that can be customized by user provided functions for generating shuffles and progressing towards the minimal score

Usage

```
optimize_design(
  batch_container,
  samples = NULL,
  scoring = NULL,
  n_shuffle = NULL,
  shuffle_proposal_func = NULL,
  acceptance_func = accept_strict_improvement,
  aggregate_scores_func = identity,
  check_score_variance = TRUE,
  autoscale_scores = FALSE,
  autoscaling_permutations = 100,
  autoscale_useboxcox = TRUE,
  sample_attributes_fixed = FALSE,
  max_iter = 10000,
  min_delta = NA,
  quiet = FALSE
)
```

Arguments

`batch_container`
: An instance of `BatchContainer`.

`samples`
: A `data.frame` with sample information. Should be `NULL` if the `BatchContainer` already has samples in it.

`scoring`
: Scoring function or a named `list()` of scoring functions.

`n_shuffle`
: Vector of length 1 or larger, defining how many random sample swaps should be performed in each iteration. If `length(n_shuffle)==1`, this sets no limit to the number of iterations. Otherwise, the optimization stops if the swapping protocol is exhausted.

`shuffle_proposal_func`
: A user defined function to propose the next shuffling of samples. Takes priority over n_shuffle if both are provided. The function is called with a BatchContainer `bc` and an integer parameter `iteration` for the current iteration number, allowing very flexible shuffling strategies. Mapper syntax is supported (see `purrr::as_mapper()`). The returned function must either return a list with fields `src`and `dst` (for pairwise sample swapping) or a numeric vector with a complete re-assigned sample order.

`acceptance_func`
: Alternative function to select a new score as the best one. Defaults to strict improvement rule, i.e. all elements of a score have to be smaller or equal in order to accept the solution as better. This may be replaced with an alternative acceptance function included in the package (e.g. `mk_simanneal_acceptance_func()`) or a user provided function. Mapper syntax is supported (see `purrr::as_mapper()`).

`aggregate_scores_func`
: A function to aggregate multiple scores AFTER (potential) auto-scaling and BEFORE acceptance evaluation. If a function is passed, (multi-dimensional) scores will be transformed (often to a single double value) before calling the acceptance function. E.g., see `first_score_only()` or `worst_score()`. Note that particular acceptance functions may require aggregation of a score to a single scalar in order to work, see for example those generated by `mk_simanneal_acceptance_func()`. Mapper syntax is supported (see `purrr::as_mapper()`).

`check_score_variance`
: Logical: if TRUE, scores will be checked for variability under sample permutation and the optimization is not performed if at least one subscore appears to have a zero variance.

`autoscale_scores`
: Logical: if TRUE, perform a transformation on the fly to equally scale scores to a standard normal. This makes scores more directly comparable and easier to aggregate.

`autoscaling_permutations`
: How many random sample permutations should be done to estimate autoscaling parameters. (Note: minimum will be 20, regardless of the specified value)

`autoscale_useboxcox`
: Logical; if TRUE, use a boxcox transformation for the autoscaling if possible at all. Requires installation of the `bestNormalize` package.

`sample_attributes_fixed`	Logical; if TRUE, sample shuffle function may generate altered sample attributes at each iteration. This affects estimation of score distributions. (Parameter only relevant if shuffle function does introduce attributes!)
`max_iter`	Stop optimization after a maximum number of iterations, independent from other stopping criteria (user defined shuffle proposal or min_delta).
`min_delta`	If not NA, optimization is stopped as soon as successive improvement (i.e. euclidean distance between score vectors from current best and previously best solution) drops below min_delta.
`quiet`	If TRUE, suppress non-critical warnings or messages.

Value

A trace object

Examples

```
data("invivo_study_samples")
bc <- BatchContainer$new(
  dimensions = c("plate" = 2, "column" = 5, "row" = 6)
)
bc <- optimize_design(bc, invivo_study_samples,
  scoring = osat_score_generator("plate", "Sex"),
  max_iter = 100
)
plot_plate(bc$get_samples(), .col = Sex)
```

---

optimize_multi_plate_design

Convenience wrapper to optimize a typical multi-plate design

Description

The batch container will in the end contain the updated experimental layout

Usage

```
optimize_multi_plate_design(
  batch_container,
  across_plates_variables = NULL,
  within_plate_variables = NULL,
  plate = "plate",
  row = "row",
  column = "column",
  n_shuffle = 1,
  max_iter = 1000,
  quiet = FALSE
)
```

Arguments

`batch_container`	Batch container (bc) with all columns that denote plate related information
`across_plates_variables`	Vector with bc column name(s) that denote(s) groups/conditions to be balanced across plates, sorted by relative importance of the factors
`within_plate_variables`	Vector with bc column name(s) that denote(s) groups/conditions to be spaced out within each plate, sorted by relative importance of the factors
`plate`	Name of the bc column that holds the plate identifier
`row`	Name of the bc column that holds the plate row number (integer values starting at 1)
`column`	Name of the bc column that holds the plate column number (integer values starting at 1)
`n_shuffle`	Vector of length 1 or larger, defining how many random sample swaps should be performed in each iteration. See `optimize_design()`.
`max_iter`	Stop any of the optimization runs after this maximum number of iterations. See `optimize_design()`.
`quiet`	If TRUE, suppress informative messages.

Value

A list with named traces, one for each optimization step

`osat_score` *Compute OSAT score for sample assignment.*

Description

The OSAT score is intended to ensure even distribution of samples across batches and is closely related to the chi-square test contingency table (Yan et al. (2012) doi:10.1186/1471216413689).

Usage

```
osat_score(bc, batch_vars, feature_vars, expected_dt = NULL, quiet = FALSE)
```

Arguments

`bc`	BatchContainer with samples or `data.table`/data.frame where every row is a location in a container and a sample in this location.
`batch_vars`	character vector with batch variable names to take into account for the score computation.
`feature_vars`	character vector with sample variable names to take into account for score computation.

`expected_dt`	A `data.table` with expected number of samples sample variables and batch variables combination. This is not required, however it does not change during the optimization process. So it is a good idea to cache this value.
`quiet`	Do not warn about `NA`s in feature columns.

Value

a list with two attributes: `$score` (numeric score value), `$expected_dt` (expected counts `data.table` for reuse)

Examples

```
sample_assignment <- tibble::tribble(
  ~ID, ~SampleType, ~Sex, ~plate,
  1, "Case", "Female", 1,
  2, "Case", "Female", 1,
  3, "Case", "Male", 2,
  4, "Control", "Female", 2,
  5, "Control", "Female", 1,
  6, "Control", "Male", 2,
  NA, NA, NA, 1,
  NA, NA, NA, 2,
)

osat_score(sample_assignment,
  batch_vars = "plate",
  feature_vars = c("SampleType", "Sex")
)
```

`osat_score_generator` *Convenience wrapper for the OSAT score*

Description

This function wraps `osat_score()` in order to take full advantage of the speed gain without managing the buffered objects in the user code.

Usage

```
osat_score_generator(batch_vars, feature_vars, quiet = FALSE)
```

Arguments

`batch_vars`	character vector with batch variable names to take into account for the score computation.
`feature_vars`	character vector with sample variable names to take into account for score computation.
`quiet`	Do not warn about `NA`s in feature columns.

Value

A function that returns the OSAT score for a specific sample arrangement

Examples

```
sample_assignment <- tibble::tribble(
  ~ID, ~SampleType, ~Sex, ~plate,
  1, "Case", "Female", 1,
  2, "Case", "Female", 1,
  3, "Case", "Male", 2,
  4, "Control", "Female", 2,
  5, "Control", "Female", 1,
  6, "Control", "Male", 2,
  NA, NA, NA, 1,
  NA, NA, NA, 2,
)

osat_scoring_function <- osat_score_generator(
  batch_vars = "plate",
  feature_vars = c("SampleType", "Sex")
)

osat_scoring_function(sample_assignment)
```

`plate_effect_example` *Example dataset with a plate effect*

Description

Here top and bottom row were both used as controls (in dilutions). The top row however was affected differently than the bottom one. This makes normalization virtually impossible.

Usage

```
data(plate_effect_example)
```

Format

An object of class "`tibble`"

row Plate row

column Plate column

conc Sample concentration

log_conc Logarithm of sample concentration

treatment Sample treatment

readout Readout from experiment

Author(s)

Balazs Banfai

`plot_plate` *Plot plate layouts*

Description

Plot plate layouts

Usage

```
plot_plate(
  .tbl,
  plate = plate,
  row = row,
  column = column,
  .color,
  .alpha = NULL,
  .pattern = NULL,
  title = paste("Layout by", rlang::as_name(rlang::enquo(plate))),
  add_excluded = FALSE,
  rename_empty = FALSE
)
```

Arguments

`.tbl`	a `tibble` (or `data.frame`) with the samples assigned to locations. Alternatively a BatchContainter with samples can be supplied here.
`plate`	optional dimension variable used for the plate ids
`row`	the dimension variable used for the row ids
`column`	the dimension variable used for the column ids
`.color`	the continuous or discrete variable to color by
`.alpha`	a continuous variable encoding transparency
`.pattern`	a discrete variable encoding tile pattern (needs ggpattern)
`title`	string for the plot title
`add_excluded`	flag to add excluded wells (in bc$exclude) to the plot. A BatchContainer must be provided for this.
`rename_empty`	whether NA entries in sample table should be renamed to 'empty'.

Value

the ggplot object

Author(s)

siebourj

Examples

```
nPlate <- 3
nColumn <- 4
nRow <- 6

treatments <- c("CTRL", "TRT1", "TRT2")
timepoints <- c(1, 2, 3)


bc <- BatchContainer$new(
  dimensions = list(
    plate = nPlate,
    column = list(values = letters[1:nColumn]),
    row = nRow
  )
)

sample_sheet <- tibble::tibble(
  sampleID = 1:(nPlate * nColumn * nRow),
 Treatment = rep(treatments, each = floor(nPlate * nColumn * nRow) / length(treatments)),
  Timepoint = rep(timepoints, floor(nPlate * nColumn * nRow) / length(treatments))
)

# assign samples from the sample sheet
bc <- assign_random(bc, samples = sample_sheet)

plot_plate(bc$get_samples(),
  plate = plate, column = column, row = row,
  .color = Treatment, .alpha = Timepoint
)


plot_plate(bc$get_samples(),
  plate = plate, column = column, row = row,
  .color = Treatment, .pattern = Timepoint
)
```

`shuffle_grouped_data` *Generate in one go a shuffling function that produces permutations with specific constraints on multiple sample variables and group sizes fitting one specific allocation variable*

Description

Generate in one go a shuffling function that produces permutations with specific constraints on multiple sample variables and group sizes fitting one specific allocation variable

Usage

```
shuffle_grouped_data(
  batch_container,
  allocate_var,
  keep_together_vars = c(),
  keep_separate_vars = c(),
  n_min = NA,
  n_max = NA,
  n_ideal = NA,
  subgroup_var_name = NULL,
  report_grouping_as_attribute = FALSE,
  prefer_big_groups = FALSE,
  strict = TRUE,
  fullTree = FALSE,
  maxCalls = 1e+06
)
```

Arguments

`batch_container`	Batch container with all samples assigned that are to be grouped and sub-grouped
`allocate_var`	Name of a variable in the `samples` table to inform possible groupings, as (sub)group sizes must add up to the correct totals
`keep_together_vars`	Vector of column names in sample table; groups are formed by pooling samples with identical values of all those variables
`keep_separate_vars`	Vector of column names in sample table; items with identical values in those variables will not be put into the same subgroup if at all possible
`n_min`	Minimal number of samples in one sub(!)group; by default 1
`n_max`	Maximal number of samples in one sub(!)group; by default the size of the biggest group
`n_ideal`	Ideal number of samples in one sub(!)group; by default the floor or ceiling of `mean(n_min,n_max)`, depending on the setting of `prefer_big_groups`
`subgroup_var_name`	An optional column name for the subgroups which are formed (or NULL)
`report_grouping_as_attribute`	Boolean, if TRUE, add an attribute table to the permutation functions' output, to be used in scoring during the design optimization
`prefer_big_groups`	Boolean; indicating whether or not bigger subgroups should be preferred in case of several possibilities
`strict`	Boolean; if TRUE, subgroup size constraints have to be met strictly, implying the possibility of finding no solution at all
`fullTree`	Boolean: Enforce full search of the possibility tree, independent of the value of `maxCalls`

`maxCalls`	Maximum number of recursive calls in the search tree, to avoid long run times with very large trees

Value

Shuffling function that on each call returns an index vector for a valid sample permutation

---

`shuffle_with_constraints`

Shuffling proposal function with constraints.

Description

Can be used with `optimize_design` to improve convergence speed.

Usage

```
shuffle_with_constraints(src = TRUE, dst = TRUE)
```

Arguments

`src`	Expression to define possible source locations in the samples/locations table. Usually evaluated based on `BatchContainer$get_samples(include_id = TRUE, as_tibble = FALSE)` as an environment (see also `with()`). A single source location is selected from rows where the expression evaluates to`TRUE`.
`dst`	Expression to define possible destination locations in the samples/locations table. Usually evaluated based on `BatchContainer$get_samples()` as an environment. Additionally a special variable `.src` is available in this environment which describes the selected source row from the table.

Value

Returns a function which accepts a `BatchContainer` and an iteration number (`i`). This function returns a list with two names: `src` vector of length 2 and `dst` vector of length two. See `BatchContainer$move_samples()`.

Examples

```
set.seed(43)

samples <- data.frame(
  id = 1:100,
  sex = sample(c("F", "M"), 100, replace = TRUE),
  group = sample(c("treatment", "control"), 100, replace = TRUE)
)

bc <- BatchContainer$new(
  dimensions = c("plate" = 5, "position" = 25)
)

scoring_f <- function(samples) {
  osat_score(
    samples,
    "plate",
    c("sex", "group")
  )$score
}

# in this example we treat all the positions in the plate as equal.
# when shuffling we enforce that source location is non-empty,
# and destination location has a different plate number
bc <- optimize_design(
  bc,
  scoring = scoring_f,
  samples,
  shuffle_proposal = shuffle_with_constraints(
    # source is non-empty location
    !is.na(.sample_id),
    # destination has a different plate
    plate != .src$plate
  ),
  max_iter = 10
)
```

---

shuffle_with_subgroup_formation

Compose shuffling function based on already available subgrouping and allocation information

---

Description

Compose shuffling function based on already available subgrouping and allocation information

Usage

```
shuffle_with_subgroup_formation(
  subgroup_object,
  subgroup_allocations,
  keep_separate_vars = c(),
  report_grouping_as_attribute = FALSE
)
```

Arguments

`subgroup_object`
: A subgrouping object as returned by `form_homogeneous_subgroups()`

`subgroup_allocations`
: A list of possible assignments of the allocation variable as returned by `compile_possible_subgroup_all`

`keep_separate_vars`
: Vector of column names in sample table; items with identical values in those variables will not be put into the same subgroup if at all possible

`report_grouping_as_attribute`
: Boolean, if TRUE, add an attribute table to the permutation functions' output, to be used in scoring during the design optimization

Value

Shuffling function that on each call returns an index vector for a valid sample permutation

---

`sum_scores` *Aggregation of scores: sum up all individual scores*

Description

Aggregation of scores: sum up all individual scores

Usage

```
sum_scores(scores, na.rm = FALSE, ...)
```

Arguments

`scores`	A score or multiple component score vector
`na.rm`	Boolean. Should NA values be ignored when obtaining the maximum? FALSE by default as ignoring NA values may render the sum meaningless.
`...`	Parameters to be ignored by this aggregation function

Value

The aggregated score, i.e. the sum of all indicidual scores.

Examples

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

`validate_samples` *Validates sample data.frame.*

Description

Validates sample data.frame.

Usage

```
validate_samples(samples)
```

Arguments

`samples`	A `data.frame` having a sample annotation per row.

`worst_score` *Aggregation of scores: take the maximum (i.e. worst score only)*

Description

This function enables comparison of the results of two scoring functions by just basing the decision on the largest element. This corresponds to the infinity-norm in ML terms.

Usage

```
worst_score(scores, na.rm = FALSE, ...)
```

Arguments

`scores`	A score or multiple component score vector
`na.rm`	Boolean. Should NA values be ignored when obtaining the maximum? FALSE by default as ignoring NA values may hide some issues with the provided scoring functions and also the aggregated value cannot be seen as the proper infinity norm anymore.
`...`	Parameters to be ignored by this aggregation function

Value

The aggregated score, i.e. the value of the largest element in a multiple-component score vector.

Examples

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

Index