Package ‘groupdata2’

October 24, 2021

Title Creating Groups from Data

Version 2.0.0

Description Methods for dividing data into groups.
Create balanced partitions and cross-validation folds.
Perform time series windowing and general grouping and splitting of data.
Balance existing groups with up- and downsampling or collapse them to fewer groups.

Depends R (>= 3.5)

License MIT + file LICENSE

URL `https://github.com/ludvigolsen/groupdata2`

BugReports `https://github.com/ludvigolsen/groupdata2/issues`

Encoding UTF-8

Imports checkmate (>= 2.0.0),
dplyr (>= 0.8.4),
numbers (>= 0.7-5),
lifecycle,
plyr (>= 1.8.5),
purrr,
rearrr (>= 0.3.0),
rlang (>= 0.4.4),
stats,
tibble (>= 2.1.3),
tidyr,
utils

RoxygenNote 7.1.2

Suggests broom,
covr,
ggplot2,
hydroGOF,
knitr,
lmerTest,
rmarkdown,
testthat,
xpectr (>= 0.4.0)

RdMacros lifecycle

Roxygen list(markdown = TRUE)

VignetteBuilder knitr

R topics documented:

all_groups_identical *Test if two grouping factors contain the same groups*

Description

[Maturing]

Checks whether two grouping factors contain the same groups, looking only at the group members, allowing for different group names / identifiers.

Usage

```
all_groups_identical(x, y)
```

Arguments

`x, y`	Two grouping factors (`vectors`/`factors` with group identifiers) to compare. **N.B.** Both are converted to `character vectors`.

Details

Both factors are sorted by `` `x` ``. A grouping factor is created with new groups starting at the values in `` `y` `` which differ from the previous row (i.e. `group()` with `method = "l_starts"` and `n = "auto"`). A similar grouping factor is created for `` `x` ``, to have group identifiers range from `1` to the number of groups. The two generated grouping factors are tested for equality.

Value

Whether **all** groups in `` `x` `` are the same in `` `y` ``, *memberwise*. (logical)

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other grouping functions: collapse_groups_by, collapse_groups(), fold(), group_factor(), group(), partition(), splt()

Examples

```
# Attach groupdata2
library(groupdata2)

# Same groups, different identifiers
x1 <- c(1, 1, 2, 2, 3, 3)
x2 <- c(2, 2, 1, 1, 4, 4)
all_groups_identical(x1, x2) # TRUE

# Same groups, different identifier types
x1 <- c(1, 1, 2, 2, 3, 3)
x2 <- c("a", "a", "b", "b", "c", "c")
all_groups_identical(x1, x2) # TRUE

# Not same groups
# Note that all groups must be the same to return TRUE
x1 <- c(1, 1, 2, 2, 3, 3)
x2 <- c(1, 2, 2, 3, 3, 3)
all_groups_identical(x1, x2) # FALSE

# Different number of groups
x1 <- c(1, 1, 2, 2, 3, 3)
x2 <- c(1, 1, 1, 2, 2, 2)
all_groups_identical(x1, x2) # FALSE
```

balance *Balance groups by up- and downsampling*

Description

[Maturing]

Uses up- and/or downsampling to fix the group sizes to the min, max, mean, or median group size or to a specific number of rows. Has a range of methods for balancing on ID level.

Usage

```
balance(
  data,
  size,
  cat_col,
  id_col = NULL,
  id_method = "n_ids",
  mark_new_rows = FALSE,
  new_rows_col_name = ".new_row"
)
```

Arguments

`data` `data.frame`. Can be *grouped*, in which case the function is applied group-wise.

`size` Size to fix group sizes to. Can be a specific number, given as a whole number, or one of the following strings: `"min"`, `"max"`, `"mean"`, `"median"`.

number: Fix each group to have the size of the specified number of row. Uses downsampling for groups with too many rows and upsampling for groups with too few rows.

min: Fix each group to have the size of smallest group in the dataset. Uses downsampling on all groups that have too many rows.

max: Fix each group to have the size of largest group in the dataset. Uses upsampling on all groups that have too few rows.

mean: Fix each group to have the mean group size in the dataset. The mean is rounded. Uses downsampling for groups with too many rows and upsampling for groups with too few rows.

median: Fix each group to have the median group size in the dataset. The median is rounded. Uses downsampling for groups with too many rows and upsampling for groups with too few rows.

`cat_col` Name of categorical variable to balance by. (Character)

`id_col` Name of factor with IDs. (Character)

IDs are considered entities, e.g. allowing us to add or remove all rows for an ID. How this is used is up to the `` `id_method` ``.

E.g. If we have measured a participant multiple times and want make sure that we keep all these measurements. Then we would either remove/add all measurements for the participant or leave in all measurements for the participant.

N.B. When `` `data` `` is a *grouped* `data.frame` (see `dplyr::group_by()`), IDs that appear in multiple groupings are considered separate entities within those groupings.

`id_method` Method for balancing the IDs. (Character)

`"n_ids"`, `"n_rows_c"`, `"distributed"`, or `"nested"`.

n_ids (default): Balances on ID level only. It makes sure there are the same number of IDs for each category. This might lead to a different number of rows between categories.

n_rows_c: Attempts to level the number of rows per category, while only removing/adding entire IDs. This is done in 2 steps:

1. If a category needs to add all its rows one or more times, the data is repeated.
2. Iteratively, the ID with the number of rows closest to the lacking/excessive number of rows is added/removed. This happens until adding/removing the closest ID would lead to a size further from the target size than the current size. If multiple IDs are closest, one is randomly sampled.

distributed: Distributes the lacking/excess rows equally between the IDs. If the number to distribute can not be equally divided, some IDs will have 1 row more/less than the others.

nested: Calls `balance()` on each category with IDs as cat_col.
I.e. if size is `"min"`, IDs will have the size of the smallest ID in their category.

`mark_new_rows` Add column with `1`s for added rows, and `0`s for original rows. (Logical)

`new_rows_col_name`

Name of column marking new rows. Defaults to `".new_row"`.

Details

Without ‘id_col‘: Upsampling is done with replacement for added rows, while the original data remains intact. Downsampling is done without replacement, meaning that rows are not duplicated but only removed.

With ‘id_col‘: See `id_method` description.

Value

data.frame with added and/or deleted rows. Ordered by potential grouping variables, `cat_col` and (potentially) `id_col`.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other sampling functions: downsample(), upsample()

Examples

```
# Attach packages
library(groupdata2)

# Create data frame
df <- data.frame(
  "participant" = factor(c(1, 1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5, 5)),
  "diagnosis" = factor(c(0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0)),
  "trial" = c(1, 2, 1, 1, 2, 3, 4, 1, 2, 1, 2, 3, 4),
  "score" = sample(c(1:100), 13)
)

# Using balance() with specific number of rows
balance(df, 3, cat_col = "diagnosis")

# Using balance() with min
balance(df, "min", cat_col = "diagnosis")

# Using balance() with max
balance(df, "max", cat_col = "diagnosis")

# Using balance() with id_method "n_ids"
# With column specifying added rows
balance(df, "max",
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "n_ids",
  mark_new_rows = TRUE
)

# Using balance() with id_method "n_rows_c"
# With column specifying added rows
balance(df, "max",
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "n_rows_c",
  mark_new_rows = TRUE
)

# Using balance() with id_method "distributed"
# With column specifying added rows
balance(df, "max",
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "distributed",
  mark_new_rows = TRUE
)

# Using balance() with id_method "nested"
# With column specifying added rows
balance(df, "max",
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "nested",
  mark_new_rows = TRUE
)
```

`collapse_groups` *Collapse groups with categorical, numerical, ID, and size balancing*

Description

[Experimental]

Collapses a set of groups into a smaller set of groups.

Attempts to balance the new groups by specified numerical columns, categorical columns, level counts in ID columns, and/or the number of rows (size).

Note: The more of these you balance at a time, the less balanced each of them may become. While, *on average*, the balancing work better than without, this is **not guaranteed on every run**. Enabling `` `auto_tune` `` can yield a much better overall balance than without in most contexts. This generates a larger set of group columns using all combinations of the balancing columns and selects the most balanced group column(s). This is slower and we recommend enabling parallelization (see `` `parallel` ``).

While this balancing algorithm will not be *optimal* in all cases, it allows balancing a **large** number of columns at once. Especially with auto-tuning enabled, this can be very powerful.

Tip: Check the balances of the new groups with `summarize_balances()` and `ranked_balances()`.

Note: The categorical and ID balancing algorithms are different to those in `fold()` and `partition()`.

Usage

```
collapse_groups(
  data,
  n,
  group_cols,
  cat_cols = NULL,
  cat_levels = NULL,
  num_cols = NULL,
  id_cols = NULL,
  balance_size = TRUE,
  auto_tune = FALSE,
  weights = NULL,
  method = "balance",
  group_aggregation_fn = mean,
  num_new_group_cols = 1,
  unique_new_group_cols_only = TRUE,
  max_iters = 5,
  extreme_pairing_levels = 1,
  combine_method = "avg_standardized",
  col_name = ".coll_groups",
  parallel = FALSE,
  verbose = TRUE
)
```

Arguments

`data` — `data.frame`. Can be *grouped*, in which case the function is applied group-wise.

`n` — Number of new groups.

When \``num_new_group_cols`\` > 1, \``n`\` can also be a vector with one \``n`\` per new group column. This allows trying multiple \``n`\` settings at a time. Note that the generated group columns are not guaranteed to be in the order of \``n`\`.

`group_cols` — Names of factors in \``data`\` for identifying the *existing* groups that should be collapsed.

Multiple names are treated as in `dplyr::group_by()` (i.e., a hierarchy of groups), where each leaf group within each parent group is considered a unique group to be collapsed. Parent groups are not considered during collapsing, why leaf groups from different parent groups can be collapsed together.

Note: Do not confuse these group columns with potential columns that \``data`\` is grouped by. \``group_cols`\` identifies the groups to be collapsed. When \``data`\` is grouped with `dplyr::group_by()`, the function is applied separately to each of those subsets.

`cat_cols` — Names of categorical columns to balance the average frequency of one or more levels of.

`cat_levels` — Names of the levels in the \``cat_cols`\` columns to balance the average frequencies of. When \``NULL`\` (default), all levels are balanced. Can be weights indicating the balancing importance of each level (within each column).

The weights are automatically scaled to sum to 1.

Can be `".minority"` or `".majority"`, in which case the minority/majority level are found and used.

When ‘`cat_cols`‘ has single column name::
Either a `vector` with level names or a named `numeric vector` with weights:
E.g. `c("dog","pidgeon","mouse")` or `c("dog" = 5,"pidgeon" = 1,"mouse" = 3)`

When ‘`cat_cols`‘ has multiple column names::
A named `list` with `vector`s for each column name in \``cat_cols`\`. When not providing a `vector` for a \``cat_cols`\` column, all levels are balanced in that column.
E.g. `list("col1" = c("dog" = 5,"pidgeon" = 1,"mouse" = 3),"col2" = c("hydrated","dehy`

num_cols
: Names of numerical columns to balance between groups.

id_cols
: Names of factor columns with IDs to balance the counts of between groups.

E.g. useful to get a similar number of participants in each group.

balance_size
: Whether to balance the size of the collapsed groups. (logical)

auto_tune
: Whether to create a larger set of collapsed group columns from all combinations of the balancing dimensions and select the overall most balanced group column(s).

This tends to create much more balanced collapsed group columns.

Can be slow, why we recommend enabling parallelization (see `parallel`).

weights
: Named vector with balancing importance weights for each of the balancing columns. Besides the columns in `cat_cols`, `num_cols`, and `id_cols`, the *size* balancing weight can be given as "size".

The weights are automatically scaled to sum to 1.

Dimensions that are *not* given a weight is automatically given the weight 1.

E.g. c("size" = 1,"cat" = 1,"num1" = 4,"num2" = 7,"id" = 2).

method
: "balance", "ascending", or "descending":

After calculating a *combined balancing column* from each of the balancing columns (see Details >> Balancing columns):

- "balance" balances the combined balancing column between the groups.
- "ascending" orders the combined balancing column and groups from the lowest to highest value.
- "descending" orders the combined balancing column and groups from the highest to lowest value.

group_aggregation_fn
: Function for aggregating values in the `num_cols` columns for each group in `group_cols`.

Default is mean(), where the average value(s) are balanced across the new groups.

When using sum(), the groups will have similar sums across the new groups.

N.B. Only used when `num_cols` is specified.

num_new_group_cols
: Number of group columns to create.

When `num_new_group_cols` > 1, columns are named with a combination of `col_name` and "_1", "_2", etc. E.g. ”$.coll_groups_1$”, ”$.coll_groups_2$”, ...

N.B. When `unique_new_group_cols_only` is `TRUE`, we may end up with fewer columns than specified, see `max_iters`.

unique_new_group_cols_only
: Whether to only return unique new group columns.

As the number of column comparisons can be quite time consuming, we recommend enabling parallelization. See `parallel`.

N.B. We can end up with fewer columns than specified in `num_new_group_cols`, see `max_iters`.

N.B. Only used when `num_new_group_cols` > 1.

max_iters
: Maximum number of attempts at reaching `num_new_group_cols` *unique* new group columns.

When only keeping unique new group columns, we risk having fewer columns than expected. Hence, we repeatedly create the missing columns and remove those that are not unique. This is done until we have `num_new_group_cols` unique group columns or we have attempted `max_iters` times.

In some cases, it is not possible to create `num_new_group_cols` unique combinations of the dataset. `max_iters` specifies when to stop trying. Note that we can end up with fewer columns than specified in `num_new_group_cols`.

N.B. Only used when `num_new_group_cols` > `1`.

`extreme_pairing_levels`

How many levels of extreme pairing to do when balancing the groups by the combined balancing column (see `Details`).

Extreme pairing: Rows/pairs are ordered as smallest, largest, second smallest, second largest, etc. If `extreme_pairing_levels` > `1`, this is done "recursively" on the extreme pairs.

N.B. Larger values work best with large datasets. If set too high, the result might not be stochastic. Always check if an increase actually makes the groups more balanced.

`combine_method` Method to combine the balancing columns by. One of `"avg_standardized"` or `"avg_min_max_scaled"`.

For each balancing column (all columns in `num_cols`, `cat_cols`, and `id_cols`, plus *size*), we calculate a normalized, numeric group summary column, which indicates the "size" of each group in that dimension. These are then combined to a single *combined balancing column*.

The three steps are:

1. Calculate a numeric representation of the balance for each column. E.g. the number of unique levels within each group of an ID column (see `Details` > `Balancing columns` for more on this).
2. Normalize each column separately with standardization (`"avg_standardized"`; Default) or MinMax scaling to the [0, 1] range (`"avg_min_max_scaled"`).
3. Average the columns *rowwise* to get a single column with one value per group. The averaging is weighted by `weights`, which is useful when one of the dimensions is more important to get a good balance of.

`combine_method` chooses whether to use standardization or MinMax scaling in step 2.

`col_name` Name of the new group column. When creating multiple new group columns (`num_new_group_cols`>`1`), this is the prefix for the names, which will be suffixed with an underscore and a number (_1, _2, _3, etc.).

`parallel` Whether to parallelize the group column comparisons when `unique_new_group_cols_only` is `TRUE`.

Especially highly recommended when `auto_tune` is enabled.

Requires a registered parallel backend. Like `doParallel::registerDoParallel`.

`verbose` Whether to print information about the process. May make the function slightly slower.

N.B. Currently only used during auto-tuning.

Details

The goal of `collapse_groups()` is to combine existing groups to a lower number of groups while (optionally) balancing one or more *numeric*, *categorical* and/or *ID* columns, along with the group *size*.

For each of these columns (and size), we calculate a normalized, numeric *"balancing column"* that when balanced between the groups lead to its original column being balanced as well.

To balance multiple columns at once, we combine their balancing columns with weighted averaging (see `` `combine_method` `` and `` `weights` ``) to a single *combined balancing column*.

Finally, we create groups where this combined balancing column is balanced between the groups, using the numerical balancing in `fold()`.

Auto-tuning:

This strategy is not guaranteed to produce balanced groups in all contexts, e.g. when the balancing columns cancel out. To increase the probability of balanced groups, we can produce multiple group columns with all combinations of the balancing columns and select the overall most balanced group column(s). We refer to this as auto-tuning (see `` `auto_tune` ``).

We find the overall most balanced group column by ranking the across-group standard deviations for each of the balancing columns, as found with `summarize_balances()`.

Example of finding the overall most balanced group column(s):

Given a group column with the following average *age* per group: `` `c(16,18,25,21)` ``, the standard deviation hereof (`3.92`) is a measure of how balanced the *age* column is. Another group column can thus have a lower/higher standard deviation and be considered more/less balanced.

We find the rankings of these standard deviations for all the balancing columns and average them (again weighted by `` `weights` ``). We select the group column(s) with the, on average, highest rank (i.e. lowest standard deviations).

Checking balances:

We highly recommend using `summarize_balances()` and `ranked_balances()` to check how balanced the created groups are on the various dimensions. When applying `ranked_balances()` to the output of `summarize_balances()`, we get a `data.frame` with the standard deviations for each balancing dimension (lower means more balanced), ordered by the average rank (see `Examples`).

Balancing columns:

The following describes the creation of the balancing columns for each of the supported column types:

cat_cols: For each column in `` `cat_cols` ``:

- **Count each level** within each group. This creates a `data.frame` with one count column per level, with one row per group.
- **Standardize** the count columns.
- **Average** the standardized counts rowwise to create one combined column representing the balance of the levels for each group. When `cat_levels` contains weights for each of the levels, we apply weighted averaging.

Example: Consider a factor column with the levels `c("A","B","C")`. We count each level per group, **n**ormalize the counts and combine them with weighted averaging:

Group	**A**	**B**	**C**	**->**	**nA**	**nB**	**nC**	**->**	**Combined**
1	5	57	1	|	0.24	0.55	-0.77	|	0.007
2	7	69	2	|	0.93	0.64	-0.77	|	0.267
3	2	34	14	|	-1.42	0.29	1.34	|	0.07
4	5	0	4	|	0.24	-1.48	0.19	|	-0.35
...	...	...	...	|	...	...	...	|	...

id_cols: For each column in `` `id_cols` ``:

- **Count** the unique IDs (levels) within each group. (Note: The same ID can be counted in multiple groups.)

num_cols: For each column in `num_cols`:

- **Aggregate** the numeric columns by group using the `group_aggregation_fn`.

size:

- **Count** the number of rows per group.

Combining balancing columns:

- Apply standardization or MinMax scaling to each of the balancing columns (see `combine_method`).
- Perform weighted averaging to get a single balancing column (see `weights`).

Example: We apply standardization and perform weighted averaging:

Group	Size	Num	Cat	ID	->	nSize	nNum	nCat	nID	->	Combined
1	34	1.3	0.007	3	|	-0.33	-0.82	0.03	-0.46	|	-0.395
2	23	4.6	0.267	4	|	-1.12	0.34	1.04	0.0	|	0.065
3	56	7.2	0.07	7	|	1.27	1.26	0.28	1.39	|	1.05
4	41	1.4	-0.35	2	|	0.18	-0.79	-1.35	-0.93	|	-0.723
...	...	...	...	...	|	...	...	...	...	|	...

Creating the groups:

Finally, we get to the group creation. There are three methods for creating groups based on the combined balancing column: `"balance"` (default), `"ascending"`, and `"descending"`.

`method` *is "balance":* To create groups that are balanced by the combined balancing column, we use the numerical balancing in `fold()`.
The following describes the numerical balancing in broad terms:

1. Rows are shuffled. **Note** that this will only affect rows with the same value in the combined balancing column.
2. Extreme pairing 1: Rows are ordered as *smallest, largest, second smallest, second largest*, etc. Each small+large pair get an *extreme-group* identifier. (See `rearrr::pair_extremes()`)
3. If `extreme_pairing_levels` > 1: These extreme-group identifiers are reordered as *smallest, largest, second smallest, second largest*, etc., by the `sum` of the combined balancing column in the represented rows. These pairs (of pairs) get a new set of extreme-group identifiers, and the process is repeated `extreme_pairing_levels`-2 times. Note that the extreme-group identifiers at the last level will represent 2^`extreme_pairing_levels` rows, why you should be careful when choosing a larger setting.
4. The extreme-group identifiers from the last pairing are randomly divided into the final groups and these final identifiers are transferred to the original rows.

N.B. When doing extreme pairing of an unequal number of rows, the row with the smallest value is placed in a group by itself, and the order is instead: (smallest), *(second smallest, largest)*, *(third smallest, second largest)*, etc.
A similar approach with *extreme triplets* (i.e. smallest, closest to median, largest, second smallest, second closest to median, second largest, etc.) may also be utilized in some scenarios. (See `rearrr::triplet_extremes()`)
Example: We order the `data.frame` by smallest *"Num"* value, largest *"Num"* value, second smallest, and so on. We *could* further (when `extreme_pairing_levels` > 1) find the sum of *"Num"* for each pair and perform extreme pairing on the pairs. Finally, we group the `data.frame`:

Group	Num	->	Group	Num	Pair	->	New group
1	-0.395	|	5	-1.23	1	|	3
2	0.065	|	3	1.05	1	|	3

3	1.05	|	4	-0.723	2	|	1
4	-0.723	|	2	0.065	2	|	1
5	-1.23	|	1	-0.395	3	|	2
6	-0.15	|	6	-0.15	3	|	2
...	...	|	...	...	...	|	...

`method` *is "ascending" or "descending":* These methods order the data by the combined balancing column and creates groups such that the sums get increasingly larger (`` `ascending` ``) or smaller (`` `descending` ``). This will in turn lead to a *pattern* of increasing/decreasing sums in the balancing columns (e.g. increasing/decreasing counts of the categorical levels, counts of IDs, number of rows and sums of numeric columns).

Value

`data.frame` with one or more new grouping factors.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

`fold()` for creating balanced folds/groups.

`partition()` for creating balanced partitions.

Other grouping functions: `all_groups_identical()`, `collapse_groups_by`, `fold()`, `group_factor()`, `group()`, `partition()`, `splt()`

Examples

```
# Attach packages
library(groupdata2)
library(dplyr)

# Set seed
xpectr::set_test_seed(42)

# Create data frame
df <- data.frame(
  "participant" = factor(rep(1:20, 3)),
  "age" = rep(sample(c(1:100), 20), 3),
  "answer" = factor(sample(c("a", "b", "c", "d"), 60, replace = TRUE)),
  "score" = sample(c(1:100), 20 * 3)
)
df <- df %>% dplyr::arrange(participant)
df$session <- rep(c("1", "2", "3"), 20)

# Sample rows to get unequal sizes per participant
df <- dplyr::sample_n(df, size = 53)

# Create the initial groups (to be collapsed)
df <- fold(
  data = df,
  k = 8,
  method = "n_dist",
  id_col = "participant"
)

# Ungroup the data frame
# Otherwise `collapse_groups()` would be
# applied to each fold separately!
df <- dplyr::ungroup(df)

# NOTE: Make sure to check the examples with `auto_tune`
# in the end, as this is where the magic lies

# Collapse to 3 groups with size balancing
# Creates new `.coll_groups` column
df_coll <- collapse_groups(
  data = df,
  n = 3,
  group_cols = ".folds",
  balance_size = TRUE # enabled by default
)

# Check balances
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = ".coll_groups",
  cat_cols = 'answer',
  num_cols = c('score', 'age'),
  id_cols = 'participant'
))

# Get ranked balances
# NOTE: When we only have a single new group column
# we don't get ranks - but this is good to use
# when comparing multiple group columns!
# The scores are standard deviations across groups
ranked_balances(coll_summary)

# Collapse to 3 groups with size + *categorical* balancing
# We create 2 new `.coll_groups_1/2` columns
df_coll <- collapse_groups(
  data = df,
  n = 3,
  group_cols = ".folds",
  cat_cols = "answer",
  balance_size = TRUE,
  num_new_group_cols = 2
)

# Check balances
# To simplify the output, we only find the
# balance of the `answer` column
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = paste0(".coll_groups_", 1:2),
  cat_cols = 'answer'
))

# Get ranked balances
# All scores are standard deviations across groups or (average) ranks
# Rows are ranked by most to least balanced
# (i.e. lowest average SD rank)
ranked_balances(coll_summary)

# Collapse to 3 groups with size + categorical + *numerical* balancing
# We create 2 new `.coll_groups_1/2` columns
df_coll <- collapse_groups(
  data = df,
  n = 3,
  group_cols = ".folds",
  cat_cols = "answer",
  num_cols = "score",
  balance_size = TRUE,
  num_new_group_cols = 2
)

# Check balances
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = paste0(".coll_groups_", 1:2),
  cat_cols = 'answer',
  num_cols = 'score'
))

# Get ranked balances
# All scores are standard deviations across groups or (average) ranks
ranked_balances(coll_summary)

# Collapse to 3 groups with size and *ID* balancing
# We create 2 new `.coll_groups_1/2` columns
df_coll <- collapse_groups(
  data = df,
  n = 3,
  group_cols = ".folds",
  id_cols = "participant",
  balance_size = TRUE,
  num_new_group_cols = 2
)

# Check balances
# To simplify the output, we only find the
# balance of the `participant` column
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = paste0(".coll_groups_", 1:2),
  id_cols = 'participant'
))

# Get ranked balances
# All scores are standard deviations across groups or (average) ranks
ranked_balances(coll_summary)

###################
#### Auto-tune ####

# As you might have seen, the balancing does not always
# perform as optimal as we might want or need
# To get a better balance, we can enable `auto_tune`
# which will create a larger set of collapsings
# and select the most balanced new group columns
# While it is not required, we recommend
# enabling parallelization

## Not run:
# Uncomment for parallelization
# library(doParallel)
# doParallel::registerDoParallel(7) # use 7 cores

# Collapse to 3 groups with lots of balancing
# We enable `auto_tune` to get a more balanced set of columns
# We create 10 new `.coll_groups_1/2/...` columns
df_coll <- collapse_groups(
  data = df,
  n = 3,
  group_cols = ".folds",
  cat_cols = "answer",
  num_cols = "score",
  id_cols = "participant",
  balance_size = TRUE,
  num_new_group_cols = 10,
  auto_tune = TRUE,
  parallel = FALSE # Set to TRUE for parallelization!
)

# Check balances
# To simplify the output, we only find the
# balance of the `participant` column
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = paste0(".coll_groups_", 1:10),
  cat_cols = "answer",
  num_cols = "score",
  id_cols = 'participant'
))

# Get ranked balances
# All scores are standard deviations across groups or (average) ranks
ranked_balances(coll_summary)

# Now we can choose the .coll_groups_* column(s)
# that we favor the balance of
# and move on with our lives!

## End(Not run)
```

collapse_groups_by *Collapse groups balanced by a single attribute*

Description

[Experimental]

Collapses a set of groups into a smaller set of groups.

Balance the new groups by:

- The **number of rows** with `collapse_groups_by_size()`
- **Numerical columns** with `collapse_groups_by_numeric()`
- One or more levels of **categorical columns** with `collapse_groups_by_levels()`
- Level counts in **ID columns** with `collapse_groups_by_ids()`
- **Any combination** of these with `collapse_groups()`

These functions wrap `collapse_groups()` to provide a simpler interface. To balance more than one of the attributes at a time and/or create multiple new unique grouping columns at once, use `collapse_groups()` directly.

While, *on average*, the balancing work better than without, this is **not guaranteed on every run**. `` `auto_tune` `` (enabled by default) can yield a much better overall balance than without in most contexts. This generates a larger set of group columns using all combinations of the balancing columns and selects the most balanced group column(s). This is slower and can be speeded up by enabling parallelization (see `` `parallel` ``).

Tip: When speed is more important than balancing, disable `` `auto_tune` ``.

Tip: Check the balances of the new groups with `summarize_balances()` and `ranked_balances()`.

Note: The categorical and ID balancing algorithms are different to those in `fold()` and `partition()`.

Usage

```
collapse_groups_by_size(
  data,
  n,
  group_cols,
  auto_tune = TRUE,
  method = "balance",
  col_name = ".coll_groups",
  parallel = FALSE,
  verbose = FALSE
)

collapse_groups_by_numeric(
  data,
  n,
  group_cols,
  num_cols,
  balance_size = FALSE,
  auto_tune = TRUE,
  method = "balance",
  group_aggregation_fn = mean,
  col_name = ".coll_groups",
  parallel = FALSE,
  verbose = FALSE
)
```

```
collapse_groups_by_levels(
  data,
  n,
  group_cols,
  cat_cols,
  cat_levels = NULL,
  balance_size = FALSE,
  auto_tune = TRUE,
  method = "balance",
  col_name = ".coll_groups",
  parallel = FALSE,
  verbose = FALSE
)

collapse_groups_by_ids(
  data,
  n,
  group_cols,
  id_cols,
  balance_size = FALSE,
  auto_tune = TRUE,
  method = "balance",
  col_name = ".coll_groups",
  parallel = FALSE,
  verbose = FALSE
)
```

Arguments

`data` — `data.frame`. Can be *grouped*, in which case the function is applied group-wise.

`n` — Number of new groups.

`group_cols` — Names of factors in `` `data` `` for identifying the *existing* groups that should be collapsed.

Multiple names are treated as in `dplyr::group_by()` (i.e., a hierarchy of groups), where each leaf group within each parent group is considered a unique group to be collapsed. Parent groups are not considered during collapsing, why leaf groups from different parent groups can be collapsed together.

Note: Do not confuse these group columns with potential columns that `` `data` `` is grouped by. `` `group_cols` `` identifies the groups to be collapsed. When `` `data` `` is grouped with `dplyr::group_by()`, the function is applied separately to each of those subsets.

`auto_tune` — Whether to create a larger set of collapsed group columns from all combinations of the balancing dimensions and select the overall most balanced group column(s).

This tends to create much more balanced collapsed group columns.

Can be slow, why we recommend enabling parallelization (see `` `parallel` ``).

`method` — `"balance"`, `"ascending"`, or `"descending"`.

- `"balance"` balances the attribute between the groups.
- `"ascending"` orders by the attribute and groups from the lowest to highest value.

- "descending" orders by the attribute and groups from the highest to lowest value.

`col_name` Name of the new group column. When creating multiple new group columns (`num_new_group_cols`>1), this is the prefix for the names, which will be suffixed with an underscore and a number (_1, _2, _3, etc.).

`parallel` Whether to parallelize the group column comparisons when `auto_tune` is enabled.

Requires a registered parallel backend. Like `doParallel::registerDoParallel`.

`verbose` Whether to print information about the process. May make the function slightly slower.

N.B. Currently only used during auto-tuning.

`num_cols` Names of numerical columns to balance between groups.

`balance_size` Whether to balance the size of the collapsed groups. (logical)

`group_aggregation_fn`

Function for aggregating values in the `num_cols` columns for each group in `group_cols`.

Default is `mean()`, where the average value(s) are balanced across the new groups.

When using `sum()`, the groups will have similar sums across the new groups.

N.B. Only used when `num_cols` is specified.

`cat_cols` Names of categorical columns to balance the average frequency of one or more levels of.

`cat_levels` Names of the levels in the `cat_cols` columns to balance the average frequencies of. When `NULL` (default), all levels are balanced. Can be weights indicating the balancing importance of each level (within each column).

The weights are automatically scaled to sum to 1.

Can be ".minority" or ".majority", in which case the minority/majority level are found and used.

When ‘cat_cols‘ has single column name::
Either a `vector` with level names or a named `numeric vector` with weights:
E.g. `c("dog","pidgeon","mouse")` or `c("dog" = 5,"pidgeon" = 1,"mouse" = 3)`

When ‘cat_cols‘ has multiple column names::
A named `list` with `vector`s for each column name in `cat_cols`. When not providing a `vector` for a `cat_cols` column, all levels are balanced in that column.
E.g. `list("col1" = c("dog" = 5,"pidgeon" = 1,"mouse" = 3),"col2" = c("hydrated","dehy`

`id_cols` Names of factor columns with IDs to balance the counts of between groups.

E.g. useful to get a similar number of participants in each group.

Details

See details in `collapse_groups()`.

Value

`data` with a new grouping factor column.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other grouping functions: `all_groups_identical()`, `collapse_groups()`, `fold()`, `group_factor()`, `group()`, `partition()`, `splt()`

Examples

```
# Attach packages
library(groupdata2)
library(dplyr)

# Set seed
xpectr::set_test_seed(42)

# Create data frame
df <- data.frame(
  "participant" = factor(rep(1:20, 3)),
  "age" = rep(sample(c(1:100), 20), 3),
  "answer" = factor(sample(c("a", "b", "c", "d"), 60, replace = TRUE)),
  "score" = sample(c(1:100), 20 * 3)
)
df <- df %>% dplyr::arrange(participant)
df$session <- rep(c("1", "2", "3"), 20)

# Sample rows to get unequal sizes per participant
df <- dplyr::sample_n(df, size = 53)

# Create the initial groups (to be collapsed)
df <- fold(
  data = df,
  k = 8,
  method = "n_dist",
  id_col = "participant"
)

# Ungroup the data frame
# Otherwise `collapse_groups*()` would be
# applied to each fold separately!
df <- dplyr::ungroup(df)

# When `auto_tune` is enabled for larger datasets
# we recommend enabling parallelization
# This can be done with:
# library(doParallel)
# doParallel::registerDoParallel(7) # use 7 cores

## Not run:

# Collapse to 3 groups with size balancing
# Creates new `.coll_groups` column
df_coll <- collapse_groups_by_size(
  data = df,
  n = 3,
  group_cols = ".folds"
)

# Check balances
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = ".coll_groups"
))

# Get ranked balances
# This is most useful when having created multiple
# new group columns with `collapse_groups()`
# The scores are standard deviations across groups
ranked_balances(coll_summary)

# Collapse to 3 groups with *categorical* balancing
df_coll <- collapse_groups_by_levels(
  data = df,
  n = 3,
  group_cols = ".folds",
  cat_cols = "answer"
)

# Check balances
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = ".coll_groups",
  cat_cols = 'answer'
))

# Collapse to 3 groups with *numerical* balancing
# Also balance size to get similar sums
# as well as means
df_coll <- collapse_groups_by_numeric(
  data = df,
  n = 3,
  group_cols = ".folds",
  num_cols = "score",
  balance_size = TRUE
)

# Check balances
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = ".coll_groups",
  num_cols = 'score'
))

# Collapse to 3 groups with *ID* balancing
# This should give us a similar number of IDs per group
df_coll <- collapse_groups_by_ids(
  data = df,
  n = 3,
  group_cols = ".folds",
  id_cols = "participant"
)
```

```
# Check balances
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = ".coll_groups",
  id_cols = 'participant'
))

# Collapse to 3 groups with balancing of ALL attributes
# We create 5 new grouping factors and compare them
# The latter is in-general a good strategy even if you
# only need a single collapsed grouping factor
# as you can choose your preferred balances
# based on the summary
# NOTE: This is slow (up to a few minutes)
# consider enabling parallelization
df_coll <- collapse_groups(
  data = df,
  n = 3,
  num_new_group_cols = 5,
  group_cols = ".folds",
  cat_cols = "answer",
  num_cols = 'score',
  id_cols = "participant",
  auto_tune = TRUE   # Disabled by default in `collapse_groups()`
  # parallel = TRUE  # Add comma above and uncomment
)

# Check balances
(coll_summary <- summarize_balances(
  data = df_coll,
  group_cols = paste0(".coll_groups_", 1:5),
  cat_cols = "answer",
  num_cols = 'score',
  id_cols = 'participant'
))

# Compare the new grouping columns
# The lowest across-group standard deviation
# is the most balanced
ranked_balances(coll_summary)


## End(Not run)
```

differs_from_previous *Find values in a vector that differ from the previous value*

Description

[Maturing]

Finds values, or indices of values, that differ from the previous value by some threshold(s).

Operates with both a positive and a negative threshold. Depending on `direction`, it checks if the difference to the previous value is:

- greater than or equal to the positive threshold.
- less than or equal to the negative threshold.

Usage

```
differs_from_previous(
  data,
  col = NULL,
  threshold = NULL,
  direction = "both",
  return_index = FALSE,
  include_first = FALSE,
  handle_na = "ignore",
  factor_conversion_warning = TRUE
)
```

Arguments

`data`

`data.frame` or `vector`.

N.B. If checking a `factor`, it is converted to a `character vector`. This means that factors can only be used when `` `threshold` `` is `NULL`. Conversion will generate a warning, which can be turned off by setting `` `factor_conversion_warning` `` to `FALSE`.

N.B. If `` `data` `` is a *grouped* `data.frame`, the function is applied group-wise and the output is a `list` of `vector`s. The names are based on the group indices (see `dplyr::group_indices()`).

`col`

Name of column to find values that differ in. Used when `` `data` `` is `data.frame`. (Character)

`threshold`

Threshold to check difference to previous value to.

`NULL`, *numeric scalar* or *numeric vector with length* `2`.

NULL: Checks if the value is different from the previous value.
Ignores `` `direction` ``.
N.B. Works for both numeric and character vectors.

Numeric scalar: Positive number.
Negative threshold is the negated number.
N.B. Only works for numeric vectors.

Numeric vector with length 2: Given as `c(negative threshold,positive threshold)`.
Negative threshold must be a negative number and positive threshold must be a positive number.
N.B. Only works for numeric vectors.

`direction`

`both`, `positive` or `negative`. (character)

both: Checks whether the difference to the previous value is

- greater than or equal to the positive threshold.
- less than or equal to the negative threshold.

positive: Checks whether the difference to the previous value is

- greater than or equal to the positive threshold.

negative: Checks whether the difference to the previous value is

- less than or equal to the negative threshold.

`return_index` Return indices of values that differ. (Logical)

`include_first` Whether to include the first element of the vector in the output. (Logical)

`handle_na` How to handle `NA`s in the column.

"ignore": Removes the `NA`s before finding the differing values, ensuring that the first value after an `NA` will be correctly identified as new, if it differs from the value before the `NA`(s).

"as_element": Treats all `NA`s as the string `"NA"`. This means, that `threshold` must be `NULL` when using this method.

Numeric scalar: A numeric value to replace `NA`s with.

`factor_conversion_warning`

Whether to throw a warning when converting a `factor` to a `character`. (Logical)

Value

`vector` with either the differing values or the indices of the differing values.

N.B. If `` `data` `` is a *grouped* `data.frame`, the output is a `list` of `vector`s with the differing values. The names are based on the group indices (see `dplyr::group_indices()`).

Author(s)

Ludvig Renbo Olsen, <`r-pkgs@ludvigolsen.dk`>

See Also

Other l_starts tools: `find_missing_starts()`, `find_starts()`, `group_factor()`, `group()`

Examples

```
# Attach packages
library(groupdata2)

# Create a data frame
df <- data.frame(
  "a" = factor(c("a", "a", "b", "b", "c", "c")),
  "n" = c(1, 3, 6, 2, 2, 4)
)

# Get differing values in column 'a' with no threshold.
# This will simply check, if it is different to the previous value or not.
differs_from_previous(df, col = "a")

# Get indices of differing values in column 'a' with no threshold.
differs_from_previous(df, col = "a", return_index = TRUE)

# Get values, that are 2 or more greater than the previous value
differs_from_previous(df, col = "n", threshold = 2, direction = "positive")

# Get values, that are 4 or more less than the previous value
differs_from_previous(df, col = "n", threshold = 4, direction = "negative")
```

```
# Get values, that are either 2 or more greater than the previous value
# or 4 or more less than the previous value
differs_from_previous(df, col = "n", threshold = c(-4, 2), direction = "both")
```

downsample *Downsampling of rows in a data frame*

Description

[Maturing]

Uses random downsampling to fix the group sizes to the smallest group in the `data.frame`.

Wraps `balance()`.

Usage

```
downsample(data, cat_col, id_col = NULL, id_method = "n_ids")
```

Arguments

`data` `data.frame`. Can be *grouped*, in which case the function is applied group-wise.

`cat_col` Name of categorical variable to balance by. (Character)

`id_col` Name of factor with IDs. (Character)

IDs are considered entities, e.g. allowing us to add or remove all rows for an ID. How this is used is up to the `` `id_method` ``.

E.g. If we have measured a participant multiple times and want make sure that we keep all these measurements. Then we would either remove/add all measurements for the participant or leave in all measurements for the participant.

N.B. When `` `data` `` is a *grouped* `data.frame` (see `dplyr::group_by()`), IDs that appear in multiple groupings are considered separate entities within those groupings.

`id_method` Method for balancing the IDs. (Character)

`"n_ids"`, `"n_rows_c"`, `"distributed"`, or `"nested"`.

n_ids (default): Balances on ID level only. It makes sure there are the same number of IDs for each category. This might lead to a different number of rows between categories.

n_rows_c: Attempts to level the number of rows per category, while only removing/adding entire IDs. This is done in 2 steps:

1. If a category needs to add all its rows one or more times, the data is repeated.
2. Iteratively, the ID with the number of rows closest to the lacking/excessive number of rows is added/removed. This happens until adding/removing the closest ID would lead to a size further from the target size than the current size. If multiple IDs are closest, one is randomly sampled.

distributed: Distributes the lacking/excess rows equally between the IDs. If the number to distribute can not be equally divided, some IDs will have 1 row more/less than the others.

nested: Calls `balance()` on each category with IDs as cat_col.
I.e. if size is `"min"`, IDs will have the size of the smallest ID in their category.

Details

Without ‘id_col‘: Downsampling is done without replacement, meaning that rows are not duplicated but only removed.

With ‘id_col‘: See `id_method` description.

Value

data.frame with some rows removed. Ordered by potential grouping variables, `cat_col` and (potentially) `id_col`.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other sampling functions: balance(), upsample()

Examples

```
# Attach packages
library(groupdata2)

# Create data frame
df <- data.frame(
  "participant" = factor(c(1, 1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5, 5)),
  "diagnosis" = factor(c(0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0)),
  "trial" = c(1, 2, 1, 1, 2, 3, 4, 1, 2, 1, 2, 3, 4),
  "score" = sample(c(1:100), 13)
)

# Using downsample()
downsample(df, cat_col = "diagnosis")

# Using downsample() with id_method "n_ids"
# With column specifying added rows
downsample(df,
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "n_ids"
)

# Using downsample() with id_method "n_rows_c"
# With column specifying added rows
downsample(df,
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "n_rows_c"
)

# Using downsample() with id_method "distributed"
downsample(df,
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "distributed"
)

# Using downsample() with id_method "nested"
downsample(df,
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "nested"
)
```

find_missing_starts *Find start positions that cannot be found in* ‘data‘

Description

[Maturing]

Tells you which values and (optionally) skip-to-numbers that are recursively removed when using the "l_starts" method with `remove_missing_starts` set to TRUE.

Usage

```
find_missing_starts(data, n, starts_col = NULL, return_skip_numbers = TRUE)
```

Arguments

data
: data.frame or vector.

  N.B. If `data` is a *grouped* data.frame, the function is applied group-wise and the output is a list of either vectors or lists. The names are based on the group indices (see dplyr::group_indices()).

n
: List of starting positions.

  Skip values by c(value,skip_to_number) where skip_to_number is the nth appearance of the value in the vector.

  See group_factor() for explanations and examples of using the "l_starts" method.

starts_col
: Name of column with values to match when `data` is a data.frame. Pass 'index' to use row names. (Character)

return_skip_numbers
: Return skip-to-numbers along with values (Logical).

Value

List of start values and skip-to-numbers or a vector with the start values. Returns NULL if no values were found.

N.B. If `data` is a *grouped* data.frame, the function is applied group-wise and the output is a list of either vectors or lists. The names are based on the group indices (see dplyr::group_indices()).

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other l_starts tools: differs_from_previous(), find_starts(), group_factor(), group()

Examples

```
# Attach packages
library(groupdata2)

# Create a data frame
df <- data.frame(
  "a" = c("a", "a", "b", "b", "c", "c"),
  stringsAsFactors = FALSE
)

# Create list of starts
starts <- c("a", "e", "b", "d", "c")

# Find missing starts with skip_to numbers
find_missing_starts(df, starts, starts_col = "a")

# Find missing starts without skip_to numbers
find_missing_starts(df, starts,
  starts_col = "a",
  return_skip_numbers = FALSE
)
```

find_starts *Find start positions of groups in data*

Description

[Maturing]

Finds values or indices of values that are not the same as the previous value.

E.g. to use with the "l_starts" method.

Wraps differs_from_previous().

Usage

```
find_starts(
  data,
  col = NULL,
  return_index = FALSE,
  handle_na = "ignore",
  factor_conversion_warning = TRUE
)
```

Arguments

data — data.frame or vector.

N.B. If checking a factor, it is converted to a character vector. Conversion will generate a warning, which can be turned off by setting `factor_conversion_warning` to FALSE.

N.B. If `data` is a *grouped* data.frame, the function is applied group-wise and the output is a list of vectors. The names are based on the group indices (see dplyr::group_indices()).

col — Name of column to find starts in. Used when `data` is a data.frame. (Character)

return_index — Whether to return indices of starts. (Logical)

handle_na — How to handle NAs in the column.

"ignore": Removes the NAs before finding the differing values, ensuring that the first value after an NA will be correctly identified as new, if it differs from the value before the NA(s).

"as_element": Treats all NAs as the string "NA". This means, that threshold must be NULL when using this method.

Numeric scalar: A numeric value to replace NAs with.

factor_conversion_warning — Throw warning when converting factor to character. (Logical)

Value

vector with either the start values or the indices of the start values.

N.B. If `data` is a *grouped* data.frame, the output is a list of vectors. The names are based on the group indices (see dplyr::group_indices()).

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other l_starts tools: differs_from_previous(), find_missing_starts(), group_factor(), group()

Examples

```
# Attach packages
library(groupdata2)

# Create a data frame
df <- data.frame(
  "a" = c("a", "a", "b", "b", "c", "c"),
  stringsAsFactors = FALSE
)

# Get start values for new groups in column 'a'
find_starts(df, col = "a")

# Get indices of start values for new groups
# in column 'a'
find_starts(df,
  col = "a",
  return_index = TRUE
)
```

```
## Use found starts with l_starts method
# Notice: This is equivalent to n = 'auto'
# with l_starts method

# Get start values for new groups in column 'a'
starts <- find_starts(df, col = "a")

# Use starts in group() with 'l_starts' method
group(df,
  n = starts, method = "l_starts",
  starts_col = "a"
)

# Similar but with indices instead of values

# Get indices of start values for new groups
# in column 'a'
starts_ind <- find_starts(df,
  col = "a",
  return_index = TRUE
)

# Use starts in group() with 'l_starts' method
group(df,
  n = starts_ind, method = "l_starts",
  starts_col = "index"
)
```

fold *Create balanced folds for cross-validation*

Description

[Stable]

Divides data into groups by a wide range of methods. Balances a given categorical variable and/or numerical variable between folds and keeps (if possible) all data points with a shared ID (e.g. participant_id) in the same fold. Can create multiple unique fold columns for repeated cross-validation.

Usage

```
fold(
  data,
  k = 5,
  cat_col = NULL,
  num_col = NULL,
  id_col = NULL,
  method = "n_dist",
  id_aggregation_fn = sum,
  extreme_pairing_levels = 1,
  num_fold_cols = 1,
  unique_fold_cols_only = TRUE,
  max_iters = 5,
  use_of_triplets = "fill",
  handle_existing_fold_cols = "keep_warn",
  parallel = FALSE
)
```

Arguments

data	data.frame. Can be *grouped*, in which case the function is applied group-wise.

k	*Depends on* ‘method‘.

Number of folds (default), fold size, with more (see `method`).

When `num_fold_cols` > 1, `k` can also be a vector with one `k` per fold column. This allows trying multiple `k` settings at a time. Note that the generated fold columns are not guaranteed to be in the order of `k`.

Given as whole number or percentage (0 < `k` < 1).

cat_col	Name of categorical variable to balance between folds.

E.g. when predicting a binary variable (a or b), we usually want both classes represented in every fold.

N.B. If also passing an `id_col`, `cat_col` should be constant within each ID.

num_col	Name of numerical variable to balance between folds.

N.B. When used with `id_col`, values for each ID are aggregated using `id_aggregation_fn` before being balanced.

N.B. When passing `num_col`, the `method` parameter is ignored.

id_col	Name of factor with IDs. This will be used to keep all rows that share an ID in the same fold (if possible).

E.g. If we have measured a participant multiple times and want to see the effect of time, we want to have all observations of this participant in the same fold.

N.B. When `data` is a *grouped* data.frame (see dplyr::group_by()), IDs that appear in multiple groupings might end up in different folds in those groupings.

method	"n_dist", "n_fill", "n_last", "n_rand", "greedy", or "staircase".

Notice: examples are sizes of the generated groups based on a vector with 57 elements.

n_dist (default): Divides the data into a specified number of groups and distributes excess data points across groups $(e.g. 11, 11, 12, 11, 12)$.
`k` is number of groups

n_fill: Divides the data into a specified number of groups and fills up groups with excess data points from the beginning $(e.g. 12, 12, 11, 11, 11)$.
`k` is number of groups

n_last: Divides the data into a specified number of groups. It finds the most equal group sizes possible, using all data points. Only the last group is able to differ in size $(e.g. 11, 11, 11, 11, 13)$.
`k` is number of groups

n_rand: Divides the data into a specified number of groups. Excess data points are placed randomly in groups (only 1 per group) $(e.g. 12, 11, 11, 11, 12)$.
`k` is number of groups

greedy: Divides up the data greedily given a specified group size $(e.g. 10, 10, 10, 10, 10, 7)$.
`k` is group size

staircase: Uses step size to divide up the data. Group size increases with 1 step for every group, until there is no more data $(e.g. 5, 10, 15, 20, 7)$.
`k` is step size

`id_aggregation_fn`

Function for aggregating values in `num_col` for each ID, before balancing `num_col`.

N.B. Only used when `num_col` and `id_col` are both specified.

`extreme_pairing_levels`

How many levels of extreme pairing to do when balancing folds by a numerical column (i.e. `num_col` is specified).

Extreme pairing: Rows/pairs are ordered as smallest, largest, second smallest, second largest, etc. If `extreme_pairing_levels > 1`, this is done "recursively" on the extreme pairs. See `Details/num_col` for more.

N.B. Larger values work best with large datasets. If set too high, the result might not be stochastic. Always check if an increase actually makes the folds more balanced. See example.

`num_fold_cols` Number of fold columns to create. Useful for repeated cross-validation.

If `num_fold_cols > 1`, columns will be named $".folds_1"$, $".folds_2"$, etc. Otherwise simply $".folds"$.

N.B. If `unique_fold_cols_only` is `TRUE`, we can end up with fewer columns than specified, see `max_iters`.

N.B. If `data` has existing fold columns, see `handle_existing_fold_cols`.

`unique_fold_cols_only`

Check if fold columns are identical and keep only unique columns.

As the number of column comparisons can be time consuming, we can run this part in parallel. See `parallel`.

N.B. We can end up with fewer columns than specified in `num_fold_cols`, see `max_iters`.

N.B. Only used when `num_fold_cols` > `1` or `data` has existing fold columns.

`max_iters` Maximum number of attempts at reaching `num_fold_cols` *unique* fold columns.

When only keeping unique fold columns, we risk having fewer columns than expected. Hence, we repeatedly create the missing columns and remove those that are not unique. This is done until we have `num_fold_cols` unique fold columns or we have attempted `max_iters` times.

In some cases, it is not possible to create `num_fold_cols` unique combinations of the dataset, e.g. when specifying `cat_col`, `id_col` and `num_col`. `max_iters` specifies when to stop trying. Note that we can end up with fewer columns than specified in `num_fold_cols`.

N.B. Only used when `num_fold_cols` > `1`.

`use_of_triplets`

`"fill"`, `"instead"` or `"never"`.

When to use extreme triplet grouping in numerical balancing (when `num_col` is specified).

fill (default): When extreme pairing cannot create enough unique fold columns, use extreme triplet grouping to create additional unique fold columns.

instead: Use extreme triplet grouping instead of extreme pairing. For some datasets, grouping in triplets give better balancing than grouping in pairs. This can be worth exploring when numerical balancing is important.
Tip: Compare the balances with `summarize_balances()` and `ranked_balances()`.

never: Never use extreme triplet grouping.

Extreme triplet grouping: Similar to extreme pairing (see `Details >> num_col`), extreme triplet grouping orders the rows as *smallest, closest to the median, largest, second smallest, second closest to the median, second largest,* etc. Each triplet gets a group identifier and we either perform recursive extreme triplet grouping on the identifiers or fold the identifiers and transfer the fold IDs to the original rows.

For some datasets, this can be give more balanced groups than extreme pairing, but on average, extreme pairing works better. Due to the grouping into triplets instead of pairs they tend to create different groupings though, so when creating many fold columns and extreme pairing cannot create enough unique fold columns, we can create the remaining (or at least some additional number) with extreme triplet grouping.

Extreme triplet grouping is implemented in `rearrr::triplet_extremes()`.

`handle_existing_fold_cols`

How to handle existing fold columns. Either `"keep_warn"`, `"keep"`, or `"remove"`.

To **add** extra fold columns, use `"keep"` or `"keep_warn"`. Note that existing fold columns might be renamed.

To **replace** the existing fold columns, use `"remove"`.

`parallel` Whether to parallelize the fold column comparisons, when `` `unique_fold_cols_only` `` is `TRUE`.

Requires a registered parallel backend. Like `doParallel::registerDoParallel`.

Details

cat_col:

1. `` `data` `` is subset by `` `cat_col` ``.
2. Subsets are grouped and merged.

id_col:

1. Groups are created from unique IDs.

num_col:

1. Rows are shuffled. **Note** that this will only affect rows with the same value in `` `num_col` ``.
2. Extreme pairing 1: Rows are ordered as *smallest, largest, second smallest, second largest*, etc. Each pair get a group identifier. (See `rearrr::pair_extremes()`)
3. If `` `extreme_pairing_levels` `` > `1`: These group identifiers are reordered as *smallest, largest, second smallest, second largest*, etc., by the sum of `` `num_col` `` in the represented rows. These pairs (of pairs) get a new set of group identifiers, and the process is repeated `` `extreme_pairing_levels`-2 `` times. Note that the group identifiers at the last level will represent `` 2^`extreme_pairing_levels` `` rows, why you should be careful when choosing that setting.
4. The group identifiers from the last pairing are folded (randomly divided into groups), and the fold identifiers are transferred to the original rows.

N.B. When doing extreme pairing of an unequal number of rows, the row with the smallest value is placed in a group by itself, and the order is instead: smallest, *second smallest, largest, third smallest, second largest*, etc.

N.B. When `` `num_fold_cols` `` > `1` and fewer than `` `num_fold_cols` `` fold columns have been created after `` `max_iters` `` attempts, we try with *extreme triplets* instead (see `rearrr::triplet_extremes()`). It groups the elements as *smallest, closest to the median, largest, second smallest, second closest to the median, second largest*, etc. We can also choose to never/only use extreme triplets via `` `use_of_triplets` ``.

cat_col AND id_col:

1. `data` is subset by `cat_col`.
2. Groups are created from unique IDs in each subset.
3. Subsets are merged.

cat_col AND num_col:

1. `data` is subset by `cat_col`.
2. Subsets are grouped by `num_col`.
3. Subsets are merged such that the largest group (by sum of `num_col`) from the first category is merged with the smallest group from the second category, etc.

num_col AND id_col:

1. Values in `num_col` are aggregated for each ID, using `id_aggregation_fn`.
2. The IDs are grouped, using the aggregated values as "num_col".
3. The groups of the IDs are transferred to the rows.

cat_col AND num_col AND id_col:

1. Values in `num_col` are aggregated for each ID, using `id_aggregation_fn`.
2. IDs are subset by `cat_col`.
3. The IDs in each subset are grouped, by using the aggregated values as "num_col".
4. The subsets are merged such that the largest group (by sum of the aggregated values) from the first category is merged with the smallest group from the second category, etc.
5. The groups of the IDs are transferred to the rows.

Value

data.frame with grouping factor for subsetting in cross-validation.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

partition for balanced partitions

Other grouping functions: all_groups_identical(), collapse_groups_by, collapse_groups(), group_factor(), group(), partition(), splt()

Examples

```
# Attach packages
library(groupdata2)
library(dplyr)

# Create data frame
df <- data.frame(
  "participant" = factor(rep(c("1", "2", "3", "4", "5", "6"), 3)),
  "age" = rep(sample(c(1:100), 6), 3),
  "diagnosis" = factor(rep(c("a", "b", "a", "a", "b", "b"), 3)),
  "score" = sample(c(1:100), 3 * 6)
)
df <- df %>% arrange(participant)
```

```
df$session <- rep(c("1", "2", "3"), 6)

# Using fold()

## Without balancing
df_folded <- fold(data = df, k = 3, method = "n_dist")

## With cat_col
df_folded <- fold(
  data = df,
  k = 3,
  cat_col = "diagnosis",
  method = "n_dist"
)

## With id_col
df_folded <- fold(
  data = df,
  k = 3,
  id_col = "participant",
  method = "n_dist"
)

## With num_col
# Note: 'method' would not be used in this case
df_folded <- fold(data = df, k = 3, num_col = "score")

# With cat_col and id_col
df_folded <- fold(
  data = df,
  k = 3,
  cat_col = "diagnosis",
  id_col = "participant", method = "n_dist"
)

## With cat_col, id_col and num_col
df_folded <- fold(
  data = df,
  k = 3,
  cat_col = "diagnosis",
  id_col = "participant", num_col = "score"
)

# Order by folds
df_folded <- df_folded %>% arrange(.folds)

## Multiple fold columns
# Useful for repeated cross-validation
# Note: Consider running in parallel
df_folded <- fold(
  data = df,
  k = 3,
  cat_col = "diagnosis",
  id_col = "participant",
  num_fold_cols = 5,
  unique_fold_cols_only = TRUE,
  max_iters = 4
)

# Different `k` per fold column
# Note: `length(k) == num_fold_cols`
df_folded <- fold(
  data = df,
  k = c(2, 3),
  cat_col = "diagnosis",
  id_col = "participant",
  num_fold_cols = 2,
  unique_fold_cols_only = TRUE,
  max_iters = 4
)

# Check the generated columns
# with `summarize_group_cols()`
summarize_group_cols(
  data = df_folded,
  group_cols = paste0('.folds_', 1:2)
)

## Check if additional `extreme_pairing_levels`
## improve the numerical balance
set.seed(2) # try with seed 1 as well
df_folded_1 <- fold(
  data = df,
  k = 3,
  num_col = "score",
  extreme_pairing_levels = 1
)
df_folded_1 %>%
  dplyr::ungroup() %>%
  summarize_balances(group_cols = '.folds', num_cols = 'score')

set.seed(2)  # Try with seed 1 as well
df_folded_2 <- fold(
  data = df,
  k = 3,
  num_col = "score",
  extreme_pairing_levels = 2
)
df_folded_2 %>%
  dplyr::ungroup() %>%
  summarize_balances(group_cols = '.folds', num_cols = 'score')

# We can directly compare how balanced the 'score' is
# in the two fold columns using a combination of
# `summarize_balances()` and `ranked_balances()`
# We see that the second fold column (made with `extreme_pairing_levels = 2`)
# has a lower standard deviation of its mean scores - meaning that they
# are more similar and thus more balanced
df_folded_1$.folds_2 <- df_folded_2$.folds
df_folded_1 %>%
  dplyr::ungroup() %>%
  summarize_balances(group_cols = c('.folds', '.folds_2'), num_cols = 'score') %>%
  ranked_balances()
```

group *Create groups from your data*

Description

[Stable]

Divides data into groups by a wide range of methods. Creates a grouping factor with `1`s for group 1, `2`s for group 2, etc. Returns a `data.frame` grouped by the grouping factor for easy use in `magrittr` `` `%>%` `` pipelines.

By default*, the data points in a group are connected sequentially (e.g. `c(1, 1, 2, 2, 3, 3)`) and splitting is done from top to bottom. *Except in the `"every"` method.

There are **five** types of grouping methods:

The `"n_*"` methods split the data into a given *number of groups*. They differ in how they handle excess data points.

The `"greedy"` method uses a *group size* to split the data into groups, greedily grabbing `` `n` `` data points from the top. The last group may thus differ in size (e.g. `c(1, 1, 2, 2, 3)`).

The `"l_*"` methods use a *list* of either starting points (`"l_starts"`) or group sizes (`"l_sizes"`). The `"l_starts"` method can also auto-detect group starts (when a value differs from the previous value).

The `"every"` method puts every `` `n` ``th data point into the same group (e.g. `c(1, 2, 3, 1, 2, 3)`).

The step methods `"staircase"` and `"primes"` increase the group size by a step for each group.

Note: To create groups balanced by a categorical and/or numerical variable, see the `fold()` and `partition()` functions.

Usage

```
group(
  data,
  n,
  method = "n_dist",
  starts_col = NULL,
  force_equal = FALSE,
  allow_zero = FALSE,
  return_factor = FALSE,
  descending = FALSE,
  randomize = FALSE,
  col_name = ".groups",
  remove_missing_starts = FALSE
)
```

Arguments

`data` `data.frame` or `vector`. When a *grouped* `data.frame`, the function is applied group-wise.

`n` *Depends on* ‘`method`‘.

Number of groups (default), group size, list of group sizes, list of group starts, number of data points between group members, step size or prime number to start at. See `` `method` ``.

Passed as whole number(s) and/or percentage(s) (`0` < `n` < `1`) and/or character.

Method `"l_starts"` allows `'auto'`.

`method` `"greedy"`, `"n_dist"`, `"n_fill"`, `"n_last"`, `"n_rand"`, `"l_sizes"`, `"l_starts"`, `"every"`, `"staircase"`, or `"primes"`.

Note: examples are sizes of the generated groups based on a vector with `57` elements.

greedy: Divides up the data greedily given a specified group size $(e.g. 10, 10, 10, 10, 10, 7)$.
`` `n` `` is group size.

n_dist (default): Divides the data into a specified number of groups and distributes excess data points across groups $(e.g. 11, 11, 12, 11, 12)$.
`` `n` `` is number of groups.

n_fill: Divides the data into a specified number of groups and fills up groups with excess data points from the beginning $(e.g. 12, 12, 11, 11, 11)$.
`` `n` `` is number of groups.

n_last: Divides the data into a specified number of groups. It finds the most equal group sizes possible, using all data points. Only the last group is able to differ in size $(e.g. 11, 11, 11, 11, 13)$.
`` `n` `` is number of groups.

n_rand: Divides the data into a specified number of groups. Excess data points are placed randomly in groups (max. 1 per group) $(e.g. 12, 11, 11, 11, 12)$.
`` `n` `` is number of groups.

l_sizes: Divides up the data by a `list` of group sizes. Excess data points are placed in an extra group at the end.
$E.g. n = list(0.2, 0.3) outputs groups with sizes (11, 17, 29)$.
`` `n` `` is a `list` of group sizes.

l_starts: Starts new groups at specified values in the `` `starts_col` `` vector.
`n` is a `list` of starting positions. Skip values by `c(value,skip_to_number)` where `skip_to_number` is the nth appearance of the value in the vector after the previous group start. The first data point is automatically a starting position.
$E.g. n = c(1, 3, 7, 25, 50) outputs groups with sizes (2, 4, 18, 25, 8)$.
To skip: $given vector c("a", "e", "o", "a", "e", "o"), n = list("a", "e", c("o", 2)) outputs groups$

If passing $n =' auto'$ the starting positions are automatically found such that a group is started whenever a value differs from the previous value (see `find_starts()`). Note that all `NA`s are first replaced by a single unique value, meaning that they will also cause group starts. See `differs_from_previous()` to set a threshold for what is considered "different".

$E.g. n = "auto" for c(10, 10, 7, 8, 8, 9) would start groups at the first 10, 7, 8 and 9, and give c(1, 1, 2$

every: Combines every `` `n` ``th data point into a group. $(e.g. 12, 12, 11, 11, 11 with n = 5)$.
`` `n` `` is the number of data points between group members ("every n").

staircase: Uses step size to divide up the data. Group size increases with 1 step for every group, until there is no more data $(e.g. 5, 10, 15, 20, 7)$.
`` `n` `` is step size.

primes: Uses prime numbers as group sizes. Group size increases to the next prime number until there is no more data. $(e.g. 5, 7, 11, 13, 17, 4)$.
`` `n` `` is the prime number to start at.

starts_col
: Name of column with values to match in method "l_starts" when `data` is a data.frame. Pass 'index' to use row names. (Character)

force_equal
: Create equal groups by discarding excess data points. Implementation varies between methods. (Logical)

allow_zero
: Whether `n` can be passed as 0. Can be useful when programmatically finding n. (Logical)

return_factor
: Only return the grouping factor. (Logical)

descending
: Change the direction of the method. (Not fully implemented) (Logical)

randomize
: Randomize the grouping factor. (Logical)

col_name
: Name of the added grouping factor.

remove_missing_starts
: Recursively remove elements from the list of starts that are not found. For method "l_starts" only. (Logical)

Value

data.frame grouped by existing grouping variables and the new grouping factor.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other grouping functions: all_groups_identical(), collapse_groups_by, collapse_groups(), fold(), group_factor(), partition(), splt()

Other staircase tools: %primes%(), %staircase%(), group_factor()

Other l_starts tools: differs_from_previous(), find_missing_starts(), find_starts(), group_factor()

Examples

```
# Attach packages
library(groupdata2)
library(dplyr)

# Create data frame
df <- data.frame(
  "x" = c(1:12),
  "species" = factor(rep(c("cat", "pig", "human"), 4)),
  "age" = sample(c(1:100), 12)
)

# Using group()
df_grouped <- group(df, n = 5, method = "n_dist")

# Using group() in pipeline to get mean age
df_means <- df %>%
  group(n = 5, method = "n_dist") %>%
  dplyr::summarise(mean_age = mean(age))

# Using group() with `l_sizes`
df_grouped <- group(
  data = df,
  n = list(0.2, 0.3),
  method = "l_sizes"
)

# Using group_factor() with `l_starts`
# `c('pig', 2)` skips to the second appearance of
# 'pig' after the first appearance of 'cat'
df_grouped <- group(
  data = df,
  n = list("cat", c("pig", 2), "human"),
  method = "l_starts",
  starts_col = "species"
)
```

---

groupdata2 *groupdata2: A package for creating groups from data*

Description

Methods for dividing data into groups. Create balanced partitions and cross-validation folds. Perform time series windowing and general grouping and splitting of data. Balance existing groups with up- and downsampling.

Details

The groupdata2 package provides six main functions: group(), group_factor(), splt(), partition(), fold(), and balance().

group

Create groups from your data.

Divides data into groups by a wide range of methods. Creates a grouping factor with 1s for group 1, 2s for group 2, etc. Returns a data.frame grouped by the grouping factor for easy use in magrittr pipelines.

Go to group()

group_factor

Create grouping factor for subsetting your data.

Divides data into groups by a wide range of methods. Creates and returns a grouping factor with 1s for group 1, 2s for group 2, etc.

Go to group_factor()

splt

Split data by a wide range of methods.

Divides data into groups by a wide range of methods. Splits data by these groups.

Go to splt()

partition

Create balanced partitions (e.g. training/test sets).

Splits data into partitions. Balances a given categorical variable between partitions and keeps (if possible) all data points with a shared ID (e.g. participant_id) in the same partition.

Go to `partition()`

fold

Create balanced folds for cross-validation.

Divides data into groups (folds) by a wide range of methods. Balances a given categorical variable between folds and keeps (if possible) all data points with the same ID (e.g. participant_id) in the same fold.

Go to `fold()`

balance

Balance the sizes of your groups with up- and downsampling.

Uses up- and/or downsampling to fix the group sizes to the `min`, `max`, `mean`, or `median` group size or to a specific number of rows. Has a set of methods for balancing on ID level.

Go to `balance()`

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

`group_factor` *Create grouping factor for subsetting your data*

Description

[Stable]

Divides data into groups by a wide range of methods. Creates and returns a grouping factor with `1`s for *group 1*, `2`s for *group 2*, etc.

By default*, the data points in a group are connected sequentially (e.g. `c(1,1,2,2,3,3)`) and splitting is done from top to bottom. *Except in the `"every"` method.

There are **five** types of grouping methods:

The `"n_*"` methods split the data into a given *number of groups*. They differ in how they handle excess data points.

The `"greedy"` method uses a *group size* to split the data into groups, greedily grabbing `` `n` `` data points from the top. The last group may thus differ in size (e.g. `c(1,1,2,2,3)`).

The `"l_*"` methods use a *list* of either starting points (`"l_starts"`) or group sizes (`"l_sizes"`). The `"l_starts"` method can also auto-detect group starts (when a value differs from the previous value).

The `"every"` method puts every `` `n` ``th data point into the same group (e.g. `c(1,2,3,1,2,3)`).

The step methods `"staircase"` and `"primes"` increase the group size by a step for each group.

Note: To create groups balanced by a categorical and/or numerical variable, see the `fold()` and `partition()` functions.

Usage

```
group_factor(
  data,
  n,
  method = "n_dist",
  starts_col = NULL,
  force_equal = FALSE,
  allow_zero = FALSE,
  descending = FALSE,
  randomize = FALSE,
  remove_missing_starts = FALSE
)
```

Arguments

`data` — `data.frame` or `vector`. When a *grouped* `data.frame`, the function is applied group-wise.

`n` — *Depends on* '`method`'.

Number of groups (default), group size, list of group sizes, list of group starts, number of data points between group members, step size or prime number to start at. See `` `method` ``.

Passed as whole number(s) and/or percentage(s) (`0` < `n` < `1`) and/or character.

Method `"l_starts"` allows `'auto'`.

`method` — `"greedy"`, `"n_dist"`, `"n_fill"`, `"n_last"`, `"n_rand"`, `"l_sizes"`, `"l_starts"`, `"every"`, `"staircase"`, or `"primes"`.

Note: examples are sizes of the generated groups based on a vector with `57` elements.

greedy: Divides up the data greedily given a specified group size $(e.g. 10, 10, 10, 10, 10, 7)$.
`` `n` `` is group size.

n_dist (default): Divides the data into a specified number of groups and distributes excess data points across groups $(e.g. 11, 11, 12, 11, 12)$.
`` `n` `` is number of groups.

n_fill: Divides the data into a specified number of groups and fills up groups with excess data points from the beginning $(e.g. 12, 12, 11, 11, 11)$.
`` `n` `` is number of groups.

n_last: Divides the data into a specified number of groups. It finds the most equal group sizes possible, using all data points. Only the last group is able to differ in size $(e.g. 11, 11, 11, 11, 13)$.
`` `n` `` is number of groups.

n_rand: Divides the data into a specified number of groups. Excess data points are placed randomly in groups (max. 1 per group) $(e.g. 12, 11, 11, 11, 12)$.
`` `n` `` is number of groups.

l_sizes: Divides up the data by a `list` of group sizes. Excess data points are placed in an extra group at the end.
$E.g. n = list(0.2, 0.3) outputs groups with sizes (11, 17, 29)$.
`` `n` `` is a `list` of group sizes.

l_starts: Starts new groups at specified values in the `` `starts_col` `` vector.

`n` is a `list` of starting positions. Skip values by `c(value,skip_to_number)` where `skip_to_number` is the nth appearance of the value in the vector after the previous group start. The first data point is automatically a starting position.
$E.g.n = c(1, 3, 7, 25, 50)outputsgroupswithsizes(2, 4, 18, 25, 8)$.
To skip: $givenvectorc(”a”, ”e”, ”o”, ”a”, ”e”, ”o”), n = list(”a”, ”e”, c(”o”, 2))outputsgroups$

If passing $n =' auto'$ the starting positions are automatically found such that a group is started whenever a value differs from the previous value (see `find_starts()`). Note that all `NA`s are first replaced by a single unique value, meaning that they will also cause group starts. See `differs_from_previous()` to set a threshold for what is considered "different".
$E.g.n = ”auto” forc(10, 10, 7, 8, 8, 9)wouldstartgroupsatthefirst10, 7, 8and9, andgivec(1, 1, 2$

every: Combines every `` `n` ``th data point into a group. $(e.g.12, 12, 11, 11, 11withn = 5)$.
`` `n` `` is the number of data points between group members ("every n").

staircase: Uses step size to divide up the data. Group size increases with 1 step for every group, until there is no more data $(e.g.5, 10, 15, 20, 7)$.
`` `n` `` is step size.

primes: Uses prime numbers as group sizes. Group size increases to the next prime number until there is no more data. $(e.g.5, 7, 11, 13, 17, 4)$.
`` `n` `` is the prime number to start at.

`starts_col` Name of column with values to match in method `"l_starts"` when `` `data` `` is a `data.frame`. Pass `'index'` to use row names. (Character)

`force_equal` Create equal groups by discarding excess data points. Implementation varies between methods. (Logical)

`allow_zero` Whether `` `n` `` can be passed as `0`. Can be useful when programmatically finding `n`. (Logical)

`descending` Change the direction of the method. (Not fully implemented) (Logical)

`randomize` Randomize the grouping factor. (Logical)

`remove_missing_starts`
Recursively remove elements from the list of starts that are not found. For method `"l_starts"` only. (Logical)

Value

Grouping factor with `1`s for group 1, `2`s for group 2, etc.

N.B. If `` `data` `` is a *grouped* `data.frame`, the output is a `data.frame` with the existing groupings and the generated grouping factor. The row order from `` `data` `` is maintained.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other grouping functions: `all_groups_identical()`, `collapse_groups_by`, `collapse_groups()`, `fold()`, `group()`, `partition()`, `splt()`

Other staircase tools: `%primes%()`, `%staircase%()`, `group()`

Other l_starts tools: `differs_from_previous()`, `find_missing_starts()`, `find_starts()`, `group()`

Examples

```
# Attach packages
library(groupdata2)
library(dplyr)

# Create a data frame
df <- data.frame(
  "x" = c(1:12),
  "species" = factor(rep(c("cat", "pig", "human"), 4)),
  "age" = sample(c(1:100), 12)
)

# Using group_factor() with n_dist
groups <- group_factor(df, 5, method = "n_dist")
df$groups <- groups

# Using group_factor() with greedy
groups <- group_factor(df, 5, method = "greedy")
df$groups <- groups

# Using group_factor() with l_sizes
groups <- group_factor(df, list(0.2, 0.3), method = "l_sizes")
df$groups <- groups

# Using group_factor() with l_starts
groups <- group_factor(df, list("cat", c("pig", 2), "human"),
  method = "l_starts", starts_col = "species"
)
df$groups <- groups
```

---

partition *Create balanced partitions*

Description

[Stable]

Splits data into partitions. Balances a given categorical variable and/or numerical variable between partitions and keeps (if possible) all data points with a shared ID (e.g. participant_id) in the same partition.

Usage

```
partition(
  data,
  p = 0.2,
  cat_col = NULL,
  num_col = NULL,
  id_col = NULL,
  id_aggregation_fn = sum,
  extreme_pairing_levels = 1,
  force_equal = FALSE,
  list_out = TRUE
)
```

Arguments

`data`
: `data.frame`. Can be *grouped*, in which case the function is applied group-wise.

`p`
: List or vector of partition sizes. Given as whole number(s) and/or percentage(s) (`0` < `` `p` `` < `1`).

  E.g. $c(0.2, 3, 0.1)$.

`cat_col`
: Name of categorical variable to balance between partitions.

  E.g. when training and testing a model for predicting a binary variable (a or b), we usually want both classes represented in both the training set and the test set.

  N.B. If also passing an `` `id_col` ``, `` `cat_col` `` should be constant within each ID.

`num_col`
: Name of numerical variable to balance between partitions.

  N.B. When used with `` `id_col` ``, values in `` `num_col` `` for each ID are aggregated using `` `id_aggregation_fn` `` before being balanced.

`id_col`
: Name of factor with IDs. Used to keep all rows that share an ID in the same partition (if possible).

  E.g. If we have measured a participant multiple times and want to see the effect of time, we want to have all observations of this participant in the same partition.

  N.B. When `` `data` `` is a *grouped* `data.frame` (see `dplyr::group_by()`), IDs that appear in multiple groupings might end up in different partitions in those groupings.

`id_aggregation_fn`
: Function for aggregating values in `` `num_col` `` for each ID, before balancing `` `num_col` ``.

  N.B. Only used when `` `num_col` `` and `` `id_col` `` are both specified.

`extreme_pairing_levels`
: How many levels of extreme pairing to do when balancing partitions by a numerical column (i.e. `` `num_col` `` is specified).

  Extreme pairing: Rows/pairs are ordered as *smallest, largest, second smallest, second largest*, etc. If `` `extreme_pairing_levels` `` > `1`, this is done "recursively" on the extreme pairs. See `` `Details/num_col` `` for more.

  N.B. Larger values work best with large datasets. If set too high, the result might not be stochastic. Always check if an increase actually makes the partitions more balanced. See `` `Examples` ``.

`force_equal`
: Whether to discard excess data. (Logical)

`list_out`
: Whether to return partitions in a `list`. (Logical)

  N.B. When `` `data` `` is a grouped `data.frame`, the output is always a `data.frame` with partition identifiers.

Details

cat_col:

1. `` `data` `` is subset by `` `cat_col` ``.
2. Subsets are partitioned and merged.

id_col:

1. Partitions are created from unique IDs.

num_col:

1. Rows are shuffled. **Note** that this will only affect rows with the same value in `num_col`.
2. Extreme pairing 1: Rows are ordered as *smallest, largest, second smallest, second largest*, etc. Each pair get a group identifier.
3. If `extreme_pairing_levels` > 1: The group identifiers are reordered as *smallest, largest, second smallest, second largest*, etc., by the sum of `num_col` in the represented rows. These pairs (of pairs) get a new set of group identifiers, and the process is repeated `extreme_pairing_levels`-2 times. Note that the group identifiers at the last level will represent 2^`extreme_pairing_levels` rows, why you should be careful when choosing that setting.
4. The final group identifiers are shuffled, and their order is applied to the full dataset.
5. The ordered dataset is split by the sizes in `p`.

N.B. When doing extreme pairing of an unequal number of rows, the row with the largest value is placed in a group by itself, and the order is instead: *smallest, second largest, second smallest, third largest, ...* , largest.

cat_col AND id_col:

1. `data` is subset by `cat_col`.
2. Partitions are created from unique IDs in each subset.
3. Subsets are merged.

cat_col AND num_col:

1. `data` is subset by `cat_col`.
2. Subsets are partitioned by `num_col`.
3. Subsets are merged.

num_col AND id_col:

1. Values in `num_col` are aggregated for each ID, using id_aggregation_fn.
2. The IDs are partitioned, using the aggregated values as "num_col".
3. The partition identifiers are transferred to the rows of the IDs.

cat_col AND num_col AND id_col:

1. Values in `num_col` are aggregated for each ID, using id_aggregation_fn.
2. IDs are subset by `cat_col`.
3. The IDs for each subset are partitioned, by using the aggregated values as "num_col".
4. The partition identifiers are transferred to the rows of the IDs.

Value

If `list_out` is TRUE:

A list of partitions where partitions are data.frames.

If `list_out` is FALSE:

A data.frame with grouping factor for subsetting.

N.B. When `data` is a grouped data.frame, the output is always a data.frame with a grouping factor.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other grouping functions: all_groups_identical(), collapse_groups_by, collapse_groups(), fold(), group_factor(), group(), splt()

Examples

```
# Attach packages
library(groupdata2)
library(dplyr)

# Create data frame
df <- data.frame(
  "participant" = factor(rep(c("1", "2", "3", "4", "5", "6"), 3)),
  "age" = rep(sample(c(1:100), 6), 3),
  "diagnosis" = factor(rep(c("a", "b", "a", "a", "b", "b"), 3)),
  "score" = sample(c(1:100), 3 * 6)
)
df <- df %>% arrange(participant)
df$session <- rep(c("1", "2", "3"), 6)

# Using partition()

# Without balancing
partitions <- partition(data = df, p = c(0.2, 0.3))

# With cat_col
partitions <- partition(data = df, p = 0.5, cat_col = "diagnosis")

# With id_col
partitions <- partition(data = df, p = 0.5, id_col = "participant")

# With num_col
partitions <- partition(data = df, p = 0.5, num_col = "score")

# With cat_col and id_col
partitions <- partition(
  data = df,
  p = 0.5,
  cat_col = "diagnosis",
  id_col = "participant"
)

# With cat_col, num_col and id_col
partitions <- partition(
  data = df,
  p = 0.5,
  cat_col = "diagnosis",
  num_col = "score",
  id_col = "participant"
)

# Return data frame with grouping factor
# with list_out = FALSE
partitions <- partition(df, c(0.5), list_out = FALSE)

# Check if additional extreme_pairing_levels
# improve the numerical balance
set.seed(2) # try with seed 1 as well
partitions_1 <- partition(
  data = df,
  p = 0.5,
  num_col = "score",
  extreme_pairing_levels = 1,
  list_out = FALSE
)
partitions_1 %>%
  dplyr::group_by(.partitions) %>%
  dplyr::summarise(
    sum_score = sum(score),
    mean_score = mean(score)
  )
set.seed(2) # try with seed 1 as well
partitions_2 <- partition(
  data = df,
  p = 0.5,
  num_col = "score",
  extreme_pairing_levels = 2,
  list_out = FALSE
)
partitions_2 %>%
  dplyr::group_by(.partitions) %>%
  dplyr::summarise(
    sum_score = sum(score),
    mean_score = mean(score)
  )
```

`ranked_balances` *Extract ranked standard deviations from summary*

Description

[Experimental]

Extract the standard deviations (default) from the `"Summary"` `data.frame` from the output of `summarize_balances()`, ordered by the `` `SD_rank` `` column.

See examples of usage in `summarize_balances()`.

Usage

```
ranked_balances(summary, measure = "SD")
```

Arguments

`summary` `"Summary"` `data.frame` from output of `summarize_balances()`.
Can also be the direct output list of `summarize_balances()`, in which case the `"Summary"` element is used.

`measure` The measure to extract rows for. One of: `"mean"`, `"median"`, `"SD"`, `"IQR"`, `"min"`, `"max"`.
The most meaningful measures to consider as metrics of balance are `` `SD` `` and `` `IQR` ``, as a smaller spread of variables across group summaries means they are more similar.
NOTE: Ranks are of standard deviations and not affected by this argument.

Value

The rows in `` `summary` `` where `` `measure` `` == "SD", ordered by the `` `SD_rank` `` column.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other summarization functions: summarize_balances(), summarize_group_cols()

---

splt *Split data by a range of methods*

Description

[Stable]

Divides data into groups by a wide range of methods. Splits data by these groups.

Wraps group() with split().

Usage

```
splt(
  data,
  n,
  method = "n_dist",
  starts_col = NULL,
  force_equal = FALSE,
  allow_zero = FALSE,
  descending = FALSE,
  randomize = FALSE,
  remove_missing_starts = FALSE
)
```

Arguments

data	data.frame or vector. When a *grouped* data.frame, the function is applied group-wise.

n	*Depends on* 'method'.

Number of groups (default), group size, list of group sizes, list of group starts, number of data points between group members, step size or prime number to start at. See `` `method` ``.

Passed as whole number(s) and/or percentage(s) ($0 < n < 1$) and/or character.

Method "l_starts" allows 'auto'.

method	"greedy", "n_dist", "n_fill", "n_last", "n_rand", "l_sizes", "l_starts", "every", "staircase", or "primes".

Note: examples are sizes of the generated groups based on a vector with 57 elements.

greedy: Divides up the data greedily given a specified group size $(e.g. 10, 10, 10, 10, 10, 7)$.
`n` is group size.

n_dist (default): Divides the data into a specified number of groups and distributes excess data points across groups $(e.g. 11, 11, 12, 11, 12)$.
`n` is number of groups.

n_fill: Divides the data into a specified number of groups and fills up groups with excess data points from the beginning $(e.g. 12, 12, 11, 11, 11)$.
`n` is number of groups.

n_last: Divides the data into a specified number of groups. It finds the most equal group sizes possible, using all data points. Only the last group is able to differ in size $(e.g. 11, 11, 11, 11, 13)$.
`n` is number of groups.

n_rand: Divides the data into a specified number of groups. Excess data points are placed randomly in groups (max. 1 per group) $(e.g. 12, 11, 11, 11, 12)$.
`n` is number of groups.

l_sizes: Divides up the data by a `list` of group sizes. Excess data points are placed in an extra group at the end.
$E.g. n = list(0.2, 0.3) outputs groups with sizes (11, 17, 29)$.
`n` is a `list` of group sizes.

l_starts: Starts new groups at specified values in the `starts_col` vector.
`n` is a `list` of starting positions. Skip values by `c(value,skip_to_number)` where `skip_to_number` is the nth appearance of the value in the vector after the previous group start. The first data point is automatically a starting position.
$E.g. n = c(1, 3, 7, 25, 50) outputs groups with sizes (2, 4, 18, 25, 8)$.
To skip: $given vector c("a", "e", "o", "a", "e", "o"), n = list("a", "e", c("o", 2)) outputs groups$

If passing $n = 'auto'$ the starting positions are automatically found such that a group is started whenever a value differs from the previous value (see `find_starts()`). Note that all `NA`s are first replaced by a single unique value, meaning that they will also cause group starts. See `differs_from_previous()` to set a threshold for what is considered "different".
$E.g. n = "auto" for c(10, 10, 7, 8, 8, 9) would start groups at the first 10, 7, 8 and 9, and give c(1, 1, 2$

every: Combines every `n`th data point into a group. $(e.g. 12, 12, 11, 11, 11 with n = 5)$.
`n` is the number of data points between group members ("every n").

staircase: Uses step size to divide up the data. Group size increases with 1 step for every group, until there is no more data $(e.g. 5, 10, 15, 20, 7)$.
`n` is step size.

primes: Uses prime numbers as group sizes. Group size increases to the next prime number until there is no more data. $(e.g. 5, 7, 11, 13, 17, 4)$.
`n` is the prime number to start at.

`starts_col` Name of column with values to match in method `"l_starts"` when `data` is a `data.frame`. Pass `'index'` to use row names. (Character)

`force_equal` Create equal groups by discarding excess data points. Implementation varies between methods. (Logical)

`allow_zero` Whether `n` can be passed as `0`. Can be useful when programmatically finding `n`. (Logical) `descending` Change the direction of the method. (Not fully implemented) (Logical)

`randomize` Randomize the grouping factor. (Logical)

`remove_missing_starts`
Recursively remove elements from the list of starts that are not found. For method `"l_starts"` only. (Logical)

Value

`list` of the split `` `data` ``.

N.B. If `` `data` `` is a *grouped* `data.frame`, there's an outer list for each group. The names are based on the group indices (see `dplyr::group_indices()`).

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other grouping functions: `all_groups_identical()`, `collapse_groups_by`, `collapse_groups()`, `fold()`, `group_factor()`, `group()`, `partition()`

Examples

```
# Attach packages
library(groupdata2)
library(dplyr)

# Create data frame
df <- data.frame(
  "x" = c(1:12),
  "species" = factor(rep(c("cat", "pig", "human"), 4)),
  "age" = sample(c(1:100), 12)
)

# Using splt()
df_list <- splt(df, 5, method = "n_dist")
```

`summarize_balances` *Summarize group balances*

Description

[Experimental]

Summarize the balances of numeric, categorical, and ID columns in and between groups in one or more group columns.

This tool allows you to quickly and thoroughly assess the balance of different columns between groups. This is for instance useful after creating groups with `fold()`, `partition()`, or `collapse_groups()` to check how well they did and to compare multiple groupings.

The output contains:

1. `` `Groups` ``: a summary per group (per grouping column).

2. `` `Summary` ``: statistical descriptors of the group summaries.
3. `` `Normalized Summary` ``: statistical descriptors of a set of "normalized" group summaries. (Disabled by default)

When comparing how balanced the grouping columns are, we can use the standard deviations of the group summary columns. The lower a standard deviation is, the more similar the groups are in that column. To quickly extract these standard deviations, ordered by an aggregated rank, use `ranked_balances()` on the `"Summary"` `data.frame` in the output.

Usage

```
summarize_balances(
  data,
  group_cols,
  cat_cols = NULL,
  num_cols = NULL,
  id_cols = NULL,
  summarize_size = TRUE,
  include_normalized = FALSE,
  rank_weights = NULL,
  cat_levels_rank_weights = NULL,
  num_normalize_fn = function(x) {     rearrr::min_max_scale(x, old_min = quantile(x,
    0.025), old_max = quantile(x, 0.975), new_min = 0, new_max = 1) }
)
```

Arguments

`data` `data.frame` with group columns to summarize by.

Can be *grouped* (see `dplyr::group_by()`), in which case the function is applied group-wise. This is not to be confused with `` `group_cols` ``.

`group_cols` Names of columns with group identifiers to summarize columns in `` `data` `` by.

`cat_cols` Names of categorical columns to summarize.

Each categorical level is counted per group.

To distinguish between levels with the same name from different `` `cat_col` `` columns, we prefix the count column name for each categorical level with parts of the name of the categorical column. This amount can be controlled with `` `max_cat_prefix_chars` ``.

Normalization when `` `include_normalized` `` is enabled: The counts of each categorical level is normalized with `log(1 + count)`.

`num_cols` Names of numerical columns to summarize.

For each column, the `mean` and `sum` is calculated per group.

Normalization when `` `include_normalized` `` is enabled: Each column is normalized with `` `num_normalize_fn` `` before calculating the `mean` and `sum` per group.

`id_cols` Names of `factor` columns with IDs to summarize.

The number of unique IDs are counted per group.

Normalization when `` `include_normalized` `` is enabled: The count of unique IDs is normalized with `log(1 + count)`.

`summarize_size` Whether to summarize the number of rows per group.

`include_normalized`
: Whether to calculate and include the normalized summary in the output.

`rank_weights`
: A named `vector` with weights for averaging the rank columns when calculating the `` `SD_rank` `` column. The name is one of the balancing columns and the number is its weight. Non-specified columns are given the weight `1`. The weights are automatically scaled to sum to 1.

  When summarizing size (see `` `summarize_size` ``), name its weight `"size"`.

  E.g. `c("size" = 1,"a_cat_col" = 2,"a_num_col" = 4,"an_id_col" = 2)`.

`cat_levels_rank_weights`
: Weights for averaging ranks of the categorical levels in `` `cat_cols` ``. Given as a named `list` with a named `vector` for each column in `` `cat_cols` ``. Non-specified levels are given the weight `1`. The weights are automatically scaled to sum to 1.

  E.g. `list("a_cat_col" = c("a" = 3,"b" = 5),"b_cat_col" = c("1" = 3,"2" = 9))`

`num_normalize_fn`
: Function for normalizing the `` `num_cols` `` columns before calculating normalized group summaries.

  Only used when `` `include_normalized` `` is enabled.

Value

`list` with two/three `data.frames`:

Groups: A summary per group.

`` `cat_cols` ``: Each level has its own column with the count of the level per group.

`` `num_cols` ``: The `mean` and `sum` per group.

`` `id_cols` ``: The count of unique IDs per group.

Summary: Statistical descriptors of the columns in `` `Groups` ``.

Contains the `mean`, `median`, standard deviation (`SD`), interquartile range (`IQR`), `min`, and `max` measures.

Especially the standard deviations and IQR measures can tell us about how balanced the groups are. When comparing multiple `` `group_cols` ``, the group column with the lowest `SD` and `IQR` can be considered the most balanced.

Normalized Summary: (Disabled by default)

Same statistical descriptors as in `` `Summary` `` but for a "normalized" version of the group summaries. The motivation is that these normalized measures can more easily be compared or combined to a single "balance score".

First, we normalize each balance column:

`` `cat_cols` ``: The level counts in the original group summaries are normalized with with `log(1 + count)`. This eases comparison of the statistical descriptors (especially standard deviations) of levels with very different count scales.

`` `num_cols` ``: The numeric columns are normalized prior to summarization by group, using the `` `num_normalize_fn` `` function. By default this applies MinMax scaling to columns such that ~95% of the values are expected to be in the `[0,1]` range.

`` `id_cols` ``: The counts of unique IDs in the original group summaries are normalized with `log(1 + count)`.

Contains the `mean`, `median`, standard deviation (`SD`), interquartile range (`IQR`), `min`, and `max` measures.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other summarization functions: ranked_balances(), summarize_group_cols()

Examples

```
# Attach packages
library(groupdata2)
library(dplyr)

set.seed(1)

# Create data frame
df <- data.frame(
  "participant" = factor(rep(c("1", "2", "3", "4", "5", "6"), 3)),
  "age" = rep(sample(c(1:100), 6), 3),
  "diagnosis" = factor(rep(c("a", "b", "a", "a", "b", "b"), 3)),
  "score" = sample(c(1:100), 3 * 6)
)
df <- df %>% arrange(participant)
df$session <- rep(c("1", "2", "3"), 6)

# Using fold()

## Without balancing
set.seed(1)
df_folded <- fold(data = df, k = 3)

# Check the balances of the various columns
# As we have not used balancing in `fold()`
# we should not expect it to be amazingly balanced
df_folded %>%
  dplyr::ungroup() %>%
  summarize_balances(
    group_cols = ".folds",
    num_cols = c("score", "age"),
    cat_cols = "diagnosis",
    id_cols = "participant"
  )

## With balancing
set.seed(1)
df_folded <- fold(
  data = df,
  k = 3,
  cat_col = "diagnosis",
  num_col = 'score',
  id_col = 'participant'
)

# Now the balance should be better
# although it may be difficult to get a good balance
# the 'score' column when also balancing on 'diagnosis'
# and keeping all rows per participant in the same fold
df_folded %>%
  dplyr::ungroup() %>%
  summarize_balances(
    group_cols = ".folds",
    num_cols = c("score", "age"),
    cat_cols = "diagnosis",
    id_cols = "participant"
  )

# Comparing multiple grouping columns
# Create 3 fold column that only balance "score"
set.seed(1)
df_folded <- fold(
  data = df,
  k = 3,
  num_fold_cols = 3,
  num_col = 'score'
)

# Summarize all three grouping cols at once
(summ <- df_folded %>%
  dplyr::ungroup() %>%
  summarize_balances(
    group_cols = paste0(".folds_", 1:3),
    num_cols = c("score")
  )
)

# Extract the across-group standard deviations
# The group column with the lowest standard deviation(s)
# is the most balanced group column
summ %>% ranked_balances()
```

summarize_group_cols *Summarize group columns*

Description

[Experimental]

Get the following summary statistics for each group column:

1. Number of groups
2. Mean, median, std., IQR, min, and max number of rows per group.

The output can be given in either *long* (default) or *wide* format.

Usage

```
summarize_group_cols(data, group_cols, long = TRUE)
```

Arguments

`data`	`data.frame` with one or more group columns (`factor`s) to summarize.
`group_cols`	Names of columns to summarize. These columns must be factors in `` `data` ``.
`long`	Whether the output should be in *long* or *wide* format.

Value

Data frame (`tibble`) with summary statistics for each column in `` `group_cols` ``.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other summarization functions: `ranked_balances()`, `summarize_balances()`

Examples

```
# Attach packages
library(groupdata2)

# Create data frame
df <- data.frame(
  "some_var" = runif(25),
  "grp_1" = factor(sample(1:5, size = 25, replace=TRUE)),
  "grp_2" = factor(sample(1:8, size = 25, replace=TRUE)),
  "grp_3" = factor(sample(LETTERS[1:3], size = 25, replace=TRUE)),
  "grp_4" = factor(sample(LETTERS[1:12], size = 25, replace=TRUE))
)

# Summarize the group columns (long format)
summarize_group_cols(
  data = df,
  group_cols = paste0("grp_", 1:4),
  long = TRUE
 )

# Summarize the group columns (wide format)
summarize_group_cols(
  data = df,
  group_cols = paste0("grp_", 1:4),
  long = FALSE
 )
```

`upsample` *Upsampling of rows in a data frame*

Description

[Maturing]

Uses random upsampling to fix the group sizes to the largest group in the data frame.

Wraps `balance()`.

Usage

```
upsample(
  data,
  cat_col,
  id_col = NULL,
  id_method = "n_ids",
  mark_new_rows = FALSE,
  new_rows_col_name = ".new_row"
)
```

Arguments

`data` `data.frame`. Can be *grouped*, in which case the function is applied group-wise.

`cat_col` Name of categorical variable to balance by. (Character)

`id_col` Name of factor with IDs. (Character)

IDs are considered entities, e.g. allowing us to add or remove all rows for an ID. How this is used is up to the `` `id_method` ``.

E.g. If we have measured a participant multiple times and want make sure that we keep all these measurements. Then we would either remove/add all measurements for the participant or leave in all measurements for the participant.

N.B. When `` `data` `` is a *grouped* `data.frame` (see `dplyr::group_by()`), IDs that appear in multiple groupings are considered separate entities within those groupings.

`id_method` Method for balancing the IDs. (Character)

`"n_ids"`, `"n_rows_c"`, `"distributed"`, or `"nested"`.

n_ids (default): Balances on ID level only. It makes sure there are the same number of IDs for each category. This might lead to a different number of rows between categories.

n_rows_c: Attempts to level the number of rows per category, while only removing/adding entire IDs. This is done in 2 steps:

1. If a category needs to add all its rows one or more times, the data is repeated.
2. Iteratively, the ID with the number of rows closest to the lacking/excessive number of rows is added/removed. This happens until adding/removing the closest ID would lead to a size further from the target size than the current size. If multiple IDs are closest, one is randomly sampled.

distributed: Distributes the lacking/excess rows equally between the IDs. If the number to distribute can not be equally divided, some IDs will have 1 row more/less than the others.

nested: Calls `balance()` on each category with IDs as cat_col.
I.e. if size is `"min"`, IDs will have the size of the smallest ID in their category.

`mark_new_rows` Add column with `1`s for added rows, and `0`s for original rows. (Logical)

`new_rows_col_name`

Name of column marking new rows. Defaults to `".new_row"`.

Details

Without ‘`id_col`‘: Upsampling is done with replacement for added rows, while the original data remains intact.

With ‘`id_col`‘: See `` `id_method` `` description.

Value

data.frame with added rows. Ordered by potential grouping variables, `` `cat_col` `` and (potentially) `` `id_col` ``.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other sampling functions: balance(), downsample()

Examples

```
# Attach packages
library(groupdata2)

# Create data frame
df <- data.frame(
  "participant" = factor(c(1, 1, 2, 3, 3, 3, 3, 4, 4, 5, 5, 5, 5)),
  "diagnosis" = factor(c(0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0)),
  "trial" = c(1, 2, 1, 1, 2, 3, 4, 1, 2, 1, 2, 3, 4),
  "score" = sample(c(1:100), 13)
)

# Using upsample()
upsample(df, cat_col = "diagnosis")

# Using upsample() with id_method "n_ids"
# With column specifying added rows
upsample(df,
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "n_ids",
  mark_new_rows = TRUE
)

# Using upsample() with id_method "n_rows_c"
# With column specifying added rows
upsample(df,
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "n_rows_c",
  mark_new_rows = TRUE
)

# Using upsample() with id_method "distributed"
# With column specifying added rows
upsample(df,
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "distributed",
  mark_new_rows = TRUE
)

# Using upsample() with id_method "nested"
# With column specifying added rows
upsample(df,
  cat_col = "diagnosis",
  id_col = "participant",
  id_method = "nested",
  mark_new_rows = TRUE
)
```

%primes% *Find remainder from 'primes' method*

Description

[Stable]

When using the "primes" method, the last group might not have the size of the associated prime number if there are not enough elements left. Use %primes% to find this remainder.

Usage

```
size %primes% start_at
```

Arguments

size	Size to group (Integer)
start_at	Prime to start at (Integer)

Value

Remainder (Integer). Returns 0 if the last group has the size of the associated prime number.

Author(s)

Ludvig Renbo Olsen, <r-pkgs@ludvigolsen.dk>

See Also

Other staircase tools: %staircase%(), group_factor(), group()

Other remainder tools: %staircase%()

Examples

```
# Attach packages
library(groupdata2)

100 %primes% 2
```

`%staircase%` *Find remainder from 'staircase' method*

Description

[Stable]

When using the `"staircase"` method, the last group might not have the size of the second last group + step size. Use `%staircase%` to find this remainder.

Usage

```
size %staircase% step_size
```

Arguments

`size`	Size to staircase (Integer)
`step_size`	Step size (Integer)

Value

Remainder (Integer). Returns `0` if the last group has the size of the second last group + step size.

Author(s)

Ludvig Renbo Olsen, `<r-pkgs@ludvigolsen.dk>`

See Also

Other staircase tools: `%primes%()`, `group_factor()`, `group()`

Other remainder tools: `%primes%()`

Examples

```
# Attach packages
library(groupdata2)

100 %staircase% 2

# Finding remainder with value 0
size = 150
for (step_size in c(1:30)){
 if(size %staircase% step_size == 0){
   print(step_size)
 }}
```

Index