Package 'groupdata2'

July 3, 2021

Title Creating Groups from Data

Version 1.5.0

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.

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 (>= 0.2.0),
rlang (>= 0.4.4),
stats,
tibble (>= 2.1.3),
tidyr,
utils

RoxygenNote 7.1.1

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 . . . 2
balance . . . 3
differs_from_previous . . . 6
downsample . . . 8
find_missing_starts . . . 10
find_starts . . . 12
fold . . . 14
group . . . 20
groupdata2 . . . 23
group_factor . . . 24
partition . . . 27
splt . . . 31
summarize_group_cols . . . 33
upsample . . . 34
%primes% . . . 37
%staircase% . . . 38

Index **39**

`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: `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
)
```

`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 `vectors`. 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 vectors 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 `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`	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,
  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 `num_fold_cols` > 1.

`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.
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 folded, and the fold identifiers are transferred to the 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.

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()`, `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::group_by(.folds) %>%
  dplyr::summarise(
    sum_score = sum(score),
    mean_score = mean(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::group_by(.folds) %>%
  dplyr::summarise(
    sum_score = sum(score),
    mean_score = mean(score)
  )
```

group *Create groups from your data*

Description

[Stable]

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.

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.

There are **four** 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 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, 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", "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.
$E.g. n = list(0.2, 0.3) outputsgroupswithsizes(11, 17, 29)$.
`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".
$E.g. n = "auto" forc(10, 10, 7, 8, 8, 9) wouldstartgroupsatthefirst 10, 7, 8 and 9, andgivec(1, 1, 2$

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(), 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 1s for *group 1*, 2s 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.

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

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 1s for group 1, 2s 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()`, `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()`, `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)
  )
```

`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, 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"`, `"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.
$E.g. n = list(0.2, 0.3) outputsgroupswithsizes(11, 17, 29)$.
`` `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".

$E.g. n = "auto" forc(10, 10, 7, 8, 8, 9) wouldstartgroupsatthefirst 10, 7, 8 and 9, andgivec(1, 1, 2$

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(), 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_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 (factors) 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>

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

%primes%, *22*, *26*, 37, *38*
%staircase%, *22*, *26*, *37*, 38

all_groups_identical, 2, *17*, *22*, *26*, *29*, *33*

balance, 3, *8, 9*, *24*, *34*, *36*
binning *(*group*)*, 20

create_balanced_groups *(*fold*)*, 14

differs_from_previous, 6, *11–13*, *21, 22*, *26*, *32*
downsample, *5*, 8, *36*
dplyr::group_by(), *4*, *9*, *15*, *28*, *35*
dplyr::group_indices(), *7, 8*, *11–13*, *33*

find_missing_starts, *8*, 10, *13*, *22*, *26*
find_starts, *8*, *11*, 12, *21, 22*, *26*, *32*
fold, *3*, 14, *22*, *24*, *26*, *29*, *33*
fold(), *20*, *24*

group, *2, 3*, *8*, *11*, *13*, *17*, 20, *23*, *26*, *29*, *33*, *37, 38*
group(), *31*
group_factor, *3*, *8*, *11*, *13*, *17*, *22, 23*, 24, *29*, *33*, *37, 38*
group_factor(), *11*
groupdata2, 23

not_previous *(*differs_from_previous*)*, 6

partition, *3*, *17*, *22, 23*, *26*, 27, *33*
partition(), *20*, *24*
primes *(*%primes%*)*, 37

split *(*group*)*, 20
split(), *31*
splt, *3*, *17*, *22, 23*, *26*, *29*, 31
staircase *(*%staircase%*)*, 38
summarize_group_cols, 33

upsample, *5*, *9*, 34

window *(*group*)*, 20