Package ‘stenR’

October 14, 2022

Title  Standardization of Raw Discrete Questionnaire Scores
Version  0.6.9
Description  An user-friendly framework to preprocess raw item scores of questionnaires into factors or scores and standardize them. Standardization can be made either by their normalization in representative sample, or by import of premade scoring table.
License  MIT + file LICENSE
Encoding  UTF-8
LazyData  true
RoxygenNote  7.1.2
Depends  R (>= 4.1)
Imports  cli, data.table, dplyr, moments, rlang, R6, stats
Suggests  covr, ggplot2, jsonlite, knitr, rmarkdown, SimMultiCorrData, testthat (>= 3.0.0)
VignetteBuilder  knitr
Config/testthat/edition  3
URL  https://statismike.github.io/stenR/
NeedsCompilation  no
Author  Michal Kosinski [aut, cre] (<https://orcid.org/0000-0002-8426-3654>)
Maintainer  Michal Kosinski <kosinski.mich@gmail.com>
Repository  CRAN
Date/Publication  2022-08-19 11:20:01 UTC

R topics documented:

  attach_scales .............................................................. 2
  CombScaleSpec ........................................................... 3
  CompScoreTable .......................................................... 4
  default_scales ........................................................... 7
  export_ScaleSpec ....................................................... 7
attach_scales

Attach additional StandardScale to already created ScoreTable

Description

Attach additional StandardScale to already created ScoreTable

Usage

attach_scales(x, scale)

Arguments

x  A ScoreTable object
scale  a StandardScale object or list of multiple StandardScale objects
Examples

# having a ScoreTable with one StandardScale attached
st <- ScoreTable(FrequencyTable(HEXACO_60$HEX_C), STEN)
st$scale
names(st$table)

# possibly attach more scales to ScoreTable
st <- attach_scales(st, list(STANINE, WECHSLER_IQ))
st$scale
names(st$table)

---

CombScaleSpec

Combined Scale Specification

Description

Combine multiple ScaleSpec objects into one in regards of `sum_items_to_scale()` function. Useful when one scale of factor contains items of different possible values or if there is hierarchy of scale or factors.

Also allows combining CombScaleSpec object if the factor structure have deeper hierarchy.

Usage

CombScaleSpec(name, ..., reverse = character(0))

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

## S3 method for class 'CombScaleSpec'
summary(object, ...)

Arguments

name           Name of the combined scale or factor
...            further arguments passed to or from other methods.
reverse        character vector containing names of the underlying subscales or factors that need to be reversed
x              a CombScaleSpec object
object         a CombScaleSpec object

Value

CombScaleSpec object

See Also

Other item preprocessing functions: `ScaleSpec()`, `sum_items_to_scale()`
Examples

# ScaleSpec objects to Combine

first_scale <- ScaleSpec(
  name = "First Scale",
  item_names = c("Item_1", "Item_2"),
  min = 1,
  max = 5
)

second_scale <- ScaleSpec(
  name = "Second Scale",
  item_names = c("Item_3", "Item_4"),
  min = 0,
  max = 7,
  reverse = "Item_3"
)

third_scale <- ScaleSpec(
  name = "Third Scale",
  item_names = c("Item_5", "Item_6"),
  min = 1,
  max = 5
)

# You can combine few ScaleSpec objects into CombScaleSpec

first_comb <- CombScaleSpec(
  name = "First Comb",
  first_scale,
  second_scale,
  reverse = "Second Scale"
)

print(first_comb)

# And also other CombScaleSpec objects!

second_comb <- CombScaleSpec(
  name = "Second Comb",
  first_comb,
  third_scale
)

print(second_comb)
**CompScoreTable**

**Description**

**[Experimental]** Computable ScoreTable class. It can compute and store ScoreTables for multiple variables containing raw score results.

After computation, it could be also used to compute new standardized scores for provided raw scores and integrate them into stored tables.

`summary()` function can be used to get general information about CompScoreTable object.

**Methods**

**Public methods:**

- `CompScoreTable$new()`
- `CompScoreTable$attach_StandardScale()`
- `CompScoreTable$attach_FrequencyTable()`
- `CompScoreTable$export_ScoreTable()`
- `CompScoreTable$standardize()`
- `CompScoreTable$clone()`

**Method `new()`**: Initialize a CompScoreTable object. You can attach one or many StandardScale and FrequencyTable objects

*Usage:*

`CompScoreTable$new(tables = NULL, scales = NULL)`

*Arguments:*

- `tables` Named list of FrequencyTable objects to be attached. Names will indicate the name of variable for which the table is calculated. Defaults to NULL, so no tables will be available at the beginning.
- `scales` StandardScale object or list of such objects to be attached. They will be used for calculation of ScoreTables. Defaults to NULL, so no scales will be available at the beginning.

*Details:* Both FrequencyTable and StandardScale objects can be attached with appropriate methods after object initialization.

*Returns:* CompScoreTable object

**Method `attach_StandardScale()`**: Attach new scale to the object. If there are any ScoreTables already computed, score for newly-attached scale will be computed automatically.

*Usage:*

`CompScoreTable$attach_StandardScale(scale, overwrite = FALSE)`

*Arguments:*

- `scale` StandardScale object defining a scale
- `overwrite` boolean indicating if the definition for a scale of the same name should be overwritten

**Method `attach_FrequencyTable()`**: Attach previously generated FrequencyTable for a given variable. ScoreTable containing every attached scale will be calculated automatically based on every new FrequencyTable.
Usage:
CompScoreTable$attach_FrequencyTable(
    ft,
    var,
    if_exists = c("stop", "append", "replace")
)

Arguments:
ft FrequencyTable to be attached
var String with the name of the variable
if_exists Action that should be taken if FrequencyTable for given variable already exists in the object.
• stop DEFAULT: don’t do anything
• append recalculates existing table
• replace replaces existing table

Method export_ScoreTable(): Export list of ScoreTables from the object
Usage:
CompScoreTable$export_ScoreTable(vars = NULL, strip = FALSE)

Arguments:
vars Names of the variables for which to get the tables. If left at NULL default - get all off them.
strip logical indicating if the ScoreTables should be stripped down to FrequencyTables during export. Defaults to FALSE

Returns: list of ScoreTable or FrequencyTable object

Method standardize(): Compute standardize scores for data.frame of raw scores. Additionally, the raw scores can be used to recalculate ScoreTables before computing (using calc = T).
Usage:
CompScoreTable$standardize(data, what, vars = names(data), calc = FALSE)

Arguments:
data data.frame containing raw scores.
what the values to get. One of either:
• quan - the quantile of raw score in the distribution
• Z - normalized Z score for the raw scores
• name of the scale attached to the CompScoreTable object
vars vector of variable names which will taken into account
calc should the ScoreTables be computed (or recalculated, if some are already provided?). Default to TRUE

Returns: data.frame with standardized values

Method clone(): The objects of this class are cloneable with this method.
Usage:
CompScoreTable$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.
**default_scales**

---

**Default Standard Scales**

**Description**

Few StandardScale objects pre-defined for usage. To create any other, use `StandardScale()` function.

- **STEN:** M: 5.5, SD: 2, min: 1, max: 10
- **STANINE:** M: 5, SD: 2, min: 1, max: 9
- **TANINE:** M: 50, SD: 10, min: 1, max: 100
- **TETRONIC:** M: 10, SD: 4, min: 0, max: 20
- **WECHSLER_IQ:** M: 100, SD: 15, min: 40, max: 160

---

**export_ScaleSpec**

**Export scale specification**

**Description**

Function to export ScaleSpec or CombScaleSpec object into json file which can be imported by `import_ScaleSpec()`

**Usage**

`export_ScaleSpec(spec, out_file)`

**Arguments**

- `spec` ScaleSpec or CombScaleSpec object to export
- `out_file` path to output file

**See Also**

Other import/export functions: `export_ScoringTable()`, `import_ScaleSpec()`, `import_ScoringTable()`

**Examples**

```r
# create temp files
ScaleSpecJSON <- tempfile(fileext = ".json")
CombScaleJSON <- tempfile(fileext = ".json")

#### import/export ScaleSpec ####
# create scale spec for export
scaleSpec <- ScaleSpec(
  name = "First Scale",
  ...)

eval(parse(text = readLines(ScaleSpecJSON, fileEncoding = "UTF-8")))
```
```r
item_names = c("Item_1", "Item_2"),
min = 1, max = 5)

# export / import
export_ScaleSpec(scaleSpec, ScaleSpecJSON)

imported_scaleSpec <- import_ScaleSpec(ScaleSpecJSON)

# check if they are the same
all.equal(scaleSpec, imported_scaleSpec)

### import/export CombScaleSpec ###
# create second scale and CombScaleSpec object
second_scale <- ScaleSpec(
  name = "Second Scale",
  item_names = c("Item_3", "Item_4"),
  min = 0, max = 7,
  reverse = "Item_3"
)
combScale <- CombScaleSpec(
  name = "First Comb",
  scaleSpec, second_scale,
  reverse = "Second Scale")

# export / import
export_ScaleSpec(combScale, CombScaleJSON)
imported_CombScale <- import_ScaleSpec(CombScaleJSON)

# check if they are the same
all.equal(combScale, imported_CombScale)
```

---

table of Export ScoringTable

---

### Description

After creation of ScoringTable it can be handy to export it into universally recognized and readable format. Two formats are currently supported: `csv` and `json`. They can be imported back into ScoringTable using `import_ScoringTable()` function.

- **csv** format is universally readable - it can be opened, edited and altered (e.g. before publication) in any spreadsheet editor. In case of ScoringTable created from GroupedScoreTable, GroupConditions can be exported to another `csv` file, creating two different files.

- **json** format can be more obtuse, but it allows export of both ScoringTable itself and GroupConditions in the same `json` file.
export_ScoringTable

Usage

```r
export_ScoringTable(
  table,
  out_file,
  method = c("csv", "json", "object"),
  cond_file
)
```

Arguments

- **table**: A ScoringTable object to export
- **out_file**: Output file. Ignored if `method = "object"`
- **method**: Method for export, either "csv", "json" or "object"
- **cond_file**: Output file for GroupConditions. Used only if `method = csv` and table created with GroupedScoreTable.

Value

list containing ScoringTable as a tibble and GroupConditions if `method = "object"`. NULL for other methods

See Also

- import_ScoringTable
- Other import/export functions: `export_ScaleSpec()`, `import_ScaleSpec()`, `import_ScoringTable()`

Examples

```r
# Scoring table to export / import #

Consc_ST <-
  GroupedFrequencyTable(
    data = IPIP_NEO_300,
    conditions = GroupConditions("Sex", "M" ~ sex == "M", "F" ~ sex == "F"),
    var = "C") |
  GroupedScoreTable(scale = STEN) |
  to_ScoringTable(min_raw = 60, max_raw = 300)

#### Export/import method: csv ####

scoretable_csv <- tempfile(fileext = ".csv")
conditions_csv <- tempfile(fileext = ".csv")

export_ScoringTable(
  table = Consc_ST,
  out_file = scoretable_csv,
  method = "csv",
  cond_file = conditions_csv
)```
## check if these are regular csv files
writeLines(head(readLines(scoretable_csv)))
writeLines(head(readLines(conditions_csv)))

imported_from_csv <- import_ScoringTable(
  source = scoretable_csv,
  method = "csv",
  cond_file = conditions_csv
)

all.equal(Consc_ST, imported_from_csv)

### Export/import method: json ###
scoretable_json <- tempfile(fileext = ".json")

export_ScoringTable(
  table = Consc_ST,
  out_file = scoretable_json,
  method = "json"
)

## check if this is regular json file
writeLines(head(readLines(scoretable_json)))

imported_from_json <- import_ScoringTable(
  source = scoretable_json,
  method = "json"
)

all.equal(Consc_ST, imported_from_json)

---

**extract_observations**

Extract observations from data

**Description**

On basis of *GroupAssignment* extract one or many groups from provided data.frame

**Usage**

```r
extract_observations(
  data,
  groups,
  group_names = NULL,
  extract_mode = c("list", "data.frame"),
  strict_names = TRUE,
  simplify = FALSE,
  id
)
```
extract_observations

Arguments

data  
data.frame from which to extract data

groups  
GroupAssignment object on basis of which extract the data.

group_names  
character vector of group names which to extract. If kept as default NULL, all groups are extracted.

extract_mode  
character: either list or data.frame. When kept as default: list, data is extracted as named list: where the name of list is name of the groups, and each one contains data.frame with observations. When data.frame is used, then assigned data is returned as one data.frame with new column named: GroupAssignment, declaring the group.

strict_names  
boolean If TRUE, then intersected groups are extracted using strict strategy: group_names need to be provided in form: "group1:group2". If FALSE, then intersected groups will be taken into regard separately, so eg. when "group1" is provided to group_names, all of: "group1:group2", "group1:group3", "group1:groupN" will be extracted. Defaults to TRUE

simplify  
boolean If TRUE, then when only one group is to be returned, it returns as data.frame without taking into account value of group_name argument. Defaults to FALSE

id  
If GroupAssignment mode is id, and you want to overwrite the original id_col, provide a name of the column there. If none is provided, then the default id_col will be used.

Value

either:

- named list of data.frames if extract_mode = 'list'
- data.frame if extract_mode = 'data.frame' or if only one group is to be returned and simplify = TRUE

See Also

Other observation grouping functions: GroupAssignment(), intersect_GroupAssignment()

Examples

#### Create Group Conditions ####

sex_grouping <- GroupConditions(
    conditions_category = "Sex",
    "M" ~ sex == "M",
    "F" ~ sex == "F",
    "O" ~ !sex %in% c("M", "F")
)

age_grouping <- GroupConditions(
    conditions_category = "Age",
    "to 20" ~ age < 20,
    "20 to 40" ~ age >= 20 & age <= 40,
"41 to 60" ~ age > 40 & age <= 60,
"above 60" ~ age > 60
)

#### Create Group Assignment ####
# can be done both with indices, so later this can be used only on the same data
# or with IDs - so later it can be done with only subset or transformed original data

sex_assignment <- GroupAssignment(HEXACO_60, sex_grouping, id = "user_id")
age_assignment <- GroupAssignment(HEXACO_60, age_grouping, id = "user_id")

#### Intersect two Group Assignment ####
# with additional forcing set
intersected <- intersect_GroupAssignment(
  sex_assignment,
age_assignment,
  force_exhaustive = TRUE,
  force_disjoint = FALSE
)

extracted <- extract_observations(
  HEXACO_60,
  groups = intersected,
group_names = c("M"),
extract_mode = "data.frame",
strict_names = FALSE)

# only groups created from "M" group were extracted
# groups without observations were dropped
table(extracted$GroupAssignment)

---

**Create a FrequencyTable**

**Description**

Normalizes the distribution of raw scores. It can be used to construct `ScoreTable()` with the use of some `StandardScale()` to normalize and standardize the raw discrete scores.

plot.FrequencyTable method requires `ggplot2` package to be installed.

**Usage**

FrequencyTable(data)

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

## S3 method for class 'FrequencyTable'
plot(x, ...)

## S3 method for class 'FrequencyTable'

 summary(object, ...)

### Arguments

- **data**: vector of raw scores. Double values are coerced to integer
- **x**: A FrequencyTable object
- **...**: further arguments passed to or from other methods.
- **object**: A FrequencyTable object

### Value

FrequencyTable object. Consists of:

- **table**: data.frame with number of observations (n), frequency in sample (freq), quantile (quan) and normalized Z-score (Z) for each point in raw score
- **status**: list containing the total number of simulated observations (n) and information about raw scores range completion (range): complete or incomplete

data.frame of descriptive statistics

### See Also

SimFrequencyTable()

---

### GroupAssignment

Assign to groups based on GroupConditions

### Description

Using GroupConditions object, assign observations to one of the groups. It can export either indices of the observations, or their unique ID: if column name is provided in id argument. Mostly used internally by more complex functions and R6 classes, but could also be useful on its own.

### Usage

GroupAssignment(
  data,
  conditions,
  id,
  force_disjoint,
  force_exhaustive,
  skip_faulty = FALSE,
  .all = FALSE,
  ...
)

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

## S3 method for class 'GroupAssignment'
summary(object, ...)

### Arguments

- **data**
  - data.frame containing observations

- **conditions**
  - GroupConditions object

- **id**
  - character name of the column containing unique ID of the observations to assign to each group. If not provided, indices will be used instead.

- **force_disjoint**
  - boolean indicating if groups disjointedness should be forced in case when one observation would pass conditions for more than one group. If TRUE, the first condition which will be met will indicate the group the observation will be assigned to. If not provided, the default from conditions will be used

- **force_exhaustive**
  - boolean indicating if groups exhausiveness should be forced in case when there are observations that don’t pass any of the provided conditions. If TRUE, then they will be assigned to .NA group. If not provided, the default from conditions will be used

- **skip_faulty**
  - boolean should the faulty condition be skipped? If FALSE as in default, error will be produced. Faultiness of seemingly correct condition may be caused by variable names to not be present in the data.

- **.all**
  - boolean. If TRUE, then additional group named .all will be created, which will contain all observations. Useful when object will be used for creation of GroupedFrequencyTable()

- **...**
  - additional arguments to be passed to or from method

- **x**
  - object

- **object**
  - GroupAssignment object

### Value

- GroupAssignment object
  - list of summaries, invisibly

### See Also

Other observation grouping functions: extract_observations(), intersect_GroupAssignment()

### Examples

```r
age_grouping <- GroupConditions(
  conditions_category = "Age",
  "to 20" ~ age < 20,
  "20 to 40" ~ age >= 20 & age <= 40,
```
"40 to 60" ~ age >= 40 & age < 60

# on basis of GroupConditions create GroupAssignment

age_assignment <- GroupAssignment(
  data = HEXACO_60,
  age_grouping)

print(age_assignment)

# overwrite the default settings imposed by `GroupConditions`

age_assignment_forced <- GroupAssignment(
  data = HEXACO_60,
  age_grouping,
  force_exhaustive = TRUE)

summary(age_assignment_forced)

# you can also use other unique identifier from your data

age_assignment_forced_w_id <- GroupAssignment(
  data = HEXACO_60,
  age_grouping,
  id = "user_id",
  force_exhaustive = TRUE)

summary(age_assignment_forced_w_id)

---

**GroupConditions**

**Conditions for observation grouping**

**Description**

With help of this function you can create GroupingConditions object, holding the basis of observation grouping. Objects of this class can be provided to complex functions to automatically group observations accordingly.

**Usage**

GroupConditions(
  conditions_category,
  ...,
  force_disjoint = TRUE,
  force_exhaustive = FALSE,
  .dots = list()
)
## S3 method for class 'GroupConditions'
print(x, ...)

## S3 method for class 'GroupConditions'
as.data.frame(x, ...)

### Arguments

- **conditions_category**
  - character value describing character of the group conditions. Mainly informative.

- **...**
  - additional arguments to be passed to or from methods.

- **force_disjoint**
  - boolean indicating if the condition formulas by default should be handled with `force_disjoint` strategy. By default `TRUE`. If `TRUE`, the first condition which will be met will indicate the group the observation will be assigned to.

- **force_exhaustive**
  - boolean indicating if groups exhaustiveness should be forced in case when there are observations that don’t pass any of the provided conditions. If `TRUE`, then they will be assigned to .NA group. Defaults to `FALSE`.

- **.dots**
  - formulas in form of a list

- **x**
  - GroupConditions object

### Value

- GroupConditions object

### Examples

```r
# create GroupConditions with formula-style conditions per each group
sex_grouping <- GroupConditions(
  conditions_category = "Sex",
  "M" ~ sex == "M",
  "F" ~ sex == "F",
  "O" ~ !sex %in% c("M", "F")
)
print(sex_grouping)

# GroupConditions can also mark if the groups should be handled by default
# with forced disjoint (default `TRUE`) and exhaustiveness (default `FALSE`)
age_grouping <- GroupConditions(
  conditions_category = "Age",
  "to 20" ~ age < 20,
  "20 to 40" ~ age >= 20 & age <= 40,
  "40 to 60" ~ age >= 40 & age < 60,
  force_disjoint = FALSE,
  force_exhaustive = TRUE
)
print(age_grouping)
```
**GroupedFrequencyTable**  
Create **GroupedFrequencyTable**

**Description**

Using `GroupConditions()` object and source `data.frame` compute a set of `FrequencyTable()`s for single variable.

**Usage**

```r
GroupedFrequencyTable(
  data, 
  conditions, 
  var, 
  force_disjoint = FALSE, 
  .all = TRUE
)
```

## S3 method for class 'GroupedFrequencyTable'

```r
print(x, ...)
```

## S3 method for class 'GroupedFrequencyTable'

```r
summary(object, ...)
```

**Arguments**

- `data`  
  source `data.frame`

- `conditions`  
  up to two `GroupConditions` objects. These objects will be passed along during creation of higher-level objects and used when `normalize_scores_grouped()` will be called. If two objects are provided, then intersection of groups will be made.

- `var`  
  name of variable to compute `GroupedFrequencyTable` for

- `force_disjoint`  
  It is recommended to keep it as default FALSE, unless the sample size is very big and it is completely mandatory to have the groups disjointed.

- `.all`  
  should `.all` or `.all1` and `.all2` groups be generated. If they are not generated, all score normalization procedures will fail if the observation can’t be assigned to any of the provided conditions (eg. because of missing data), leaving it’s score as NA. Defaults to TRUE

- `x`  
  A `GroupedFrequencyTable` object

- `...`  
  further arguments passed to or from other methods.

- `object`  
  A `GroupedFrequencyTable` object

**Details**

`force_exhaustive` will always be checked as FALSE during the calculations. It is mandatory for validity of the created `FrequencyTables`
Value

data.frame of descriptive statistics

See Also

plot.GroupedFrequencyTable

Description

Create GroupedScoreTable

Usage

GroupedScoreTable(table, scale)

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

Arguments

table GroupedFrequencyTable object
scale a StandardScale object or list of multiple StandardScale objects
x A GroupedScoreTable object
... further arguments passed to or from other methods.

Value

GroupedScoreTable object, which consists of named list of ScoreTable objects and GroupConditions object used for grouping

See Also

plot.GroupedScoreTable
HEXACO_60

Sample data of HEXACO-60 questionnaire results

Description

Dataset containing summed scale scores of HEXACO-60 questionnaire. They were obtained during 2020 study on Polish incidental sample.

Usage

HEXACO_60

Format

A data frame with 204 rows and 9 variables

- **user_id**: identity anonymized with 'ids::adjective_animal'
- **sex**: sex of the participant ('M'ale, 'F'emale or 'O'ther)
- **age**: age of the participant (15–62)
- **HEX_H**: Honesty-Humility raw score (14–50)
- **HEX_E**: Emotionality raw score (10–47)
- **HEX_X**: eXtraversion raw score (11–46)
- **HEX_A**: Agreeableness raw score (12–45)
- **HEX_C**: Consciousness raw score (17–50)
- **HEX_O**: Openness to Experience raw score (18–50)

Details

All HEXACO scales consist of 10 items with responses as numeric values 1-5 (so the absolute min and max are 10-50)

---

import_ScaleSpec

Import scale specification

Description

Function to import ScaleSpec or CombScaleSpec object from json file that have been exported with `export_ScaleSpec()`

Usage

`import_ScaleSpec(source)`
import_ScaleSpec

Arguments

source  path to JSON file containing exported object

See Also

Other import/export functions: export_ScaleSpec(), export_ScoringTable(), import_ScoringTable()

Examples

# create temp files
ScaleSpecJSON <- tempfile(fileext = "json")
CombScaleJSON <- tempfile(fileext = "json")

#### import/export ScaleSpec ####

# create scale spec for export
scaleSpec <- ScaleSpec(
  name = "First Scale",
  item_names = c("Item_1", "Item_2"),
  min = 1, max = 5)

# export / import
export_ScaleSpec(scaleSpec, ScaleSpecJSON)
imported_scaleSpec <- import_ScaleSpec(ScaleSpecJSON)

# check if they are the same
all.equal(scaleSpec, imported_scaleSpec)

#### import/export CombScaleSpec ####

# create second scale and CombScaleSpec object
second_scale <- ScaleSpec(
  name = "Second Scale",
  item_names = c("Item_3", "Item_4"),
  min = 0, max = 7,
  reverse = "Item_3"
)
combScale <- CombScaleSpec(
  name = "First Comb",
  scaleSpec, second_scale,
  reverse = "Second Scale"
)

# export / import
export_ScaleSpec(combScale, CombScaleJSON)
imported_CombScale <- import_ScaleSpec(CombScaleJSON)

# check if they are the same
all.equal(combScale, imported_CombScale)
import_ScoringTable

import_ScoringTable  Import ScoringTable

Description

ScoringTable can be imported from csv, json file or tibble. Source file or object can be either an output of export_ScoringTable() function, or created by hand - though it needs to be created following the correct format.

Usage

import_ScoringTable(
  source,  
  method = c("csv", "json", "object"),
  cond_file,
  conditions
)

Arguments

source  Path to the file to import the ScoringTable from (for csv and json methods) or ScoringTable in form of data.frame (for object method)
method  Method for import, either csv, json or object
cond_file  File to import the GroupConditions from, if using csv method
conditions  GroupCondition object or list of up to two of them. Mandatory for object method and csv method if no cond_file is provided. If provided while using json method, original GroupConditions will be ignored.

Value

ScoringTable object

See Also

export_ScoringTable

Other import/export functions: export_ScaleSpec(), export_ScoringTable(), import_ScaleSpec()

Examples

# Scoring table to export / import #

Consc_ST <-
  GroupedFrequencyTable(
    data = IPIP_NEO_300,
    conditions = GroupConditions("Sex", "M" ~ sex == "M", "F" ~ sex == "F"),
    var = "C" ) |>
  GroupedScoreTable(scale = STEN) |>
to_ScoringTable(min_raw = 60, max_raw = 300)

#### Export/import method: csv ####

scoretable_csv <- tempfile(fileext = "csv")
conditions_csv <- tempfile(fileext = "csv")

export_ScoringTable(
  table = Consc_ST,
  out_file = scoretable_csv,
  method = "csv",
  cond_file = conditions_csv
)

## check if these are regular csv files
writeLines(head(readLines(scoretable_csv)))
writeLines(head(readLines(conditions_csv)))

imported_from_csv <- import_ScoringTable(
  source = scoretable_csv,
  method = "csv",
  cond_file = conditions_csv
)

all.equal(Consc_ST, imported_from_csv)

#### Export/import method: json ####

scoretable_json <- tempfile(fileext = "json")

export_ScoringTable(
  table = Consc_ST,
  out_file = scoretable_json,
  method = "json"
)

## check if this is regular json file
writeLines(head(readLines(scoretable_json)))

imported_from_json <- import_ScoringTable(
  source = scoretable_json,
  method = "json"
)

all.equal(Consc_ST, imported_from_json)

intersect_GroupAssignment

Intersect two GroupAssignment
**intersect_GroupAssignment**

**Description**

You can intersect two GroupAssignment with this function.

**Usage**

```r
def intersect_GroupAssignment(
    GA1,
    GA2,
    force_disjoint = TRUE,
    force_exhaustive = FALSE
)
```

**Arguments**

- **GA1, GA2**: `GroupAssignment` objects to intersect. No previously intersected objects can be intersected again.
- **force_disjoint**: `boolean` indicating if groups disjointedness should be forced in case when one observation would end in multiple intersections. If `TRUE`, observation will remain only in the first intersection to which it will be assigned. Default to `TRUE`.
- **force_exhaustive**: `boolean` indicating if elements that are not assigned to any of the intersecting groups should be gathered together in `.NA:.NA` group

**Value**

`GroupAssignment` object with intersected groups.

**See Also**

Other observation grouping functions: `GroupAssignment()`, `extract_observations()`

**Examples**

```r
sex_grouping <- GroupConditions(
    conditions_category = "Sex",
    "M" ~ sex == "M",
    "F" ~ sex == "F",
    "O" ~ !sex %in% c("M", "F")
)

age_grouping <- GroupConditions(
    conditions_category = "Age",
    "to 20" ~ age < 20,
    "20 to 40" ~ age >= 20 & age <= 40,
    "40 to 60" ~ age >= 40 & age < 60,
    force_exhaustive = TRUE,
    force_disjoint = FALSE
)

# intersect two distinct GroupAssignments
```
interacted <- intersect_GroupAssignment(
  GA1 = GroupAssignment HEXACO_60, sex_grouping),
  GA2 = GroupAssignment HEXACO_60, age_grouping),
  force_exhaustive = TRUE,
  force_disjoint = FALSE
)

summary(interacted)

### IPIP_NEO_300

**Sample data of IPIP-NEO-300 questionnaire results**

**Description**

Dataset containing sample of 13198 results of IPIP-NEO-300 results from Johnson J.A. study published at 2014, preprocessed using `sum_items_to_scale()` function. It contains many observations of different ages and sexes, also including NA values, whenever at least one of the underlying item scores were missing.

**Usage**

IPIP_NEO_300

**Format**

A data frame with 13198 rows and 7 variables

- **sex** sex of the participant ('M'ale or 'F'emale)
- **age** age of the participant (10–98)
- **N** Raw score for Neuroticism scale (63–292)
- **E** Raw score for Extraversion scale (80–296)
- **O** Raw score for Openness to Experience (76–298)
- **A** Raw score for Agreeableness (66–292)
- **C** Raw score for Consciousness (81–299)

**References**

is_stenR_classes

Checkers for stenR S3 and R6 classes

Description

Various functions to check if given \texttt{R} object is of given class. Additionally:

- \texttt{is.intersected()} checks if the \texttt{GroupAssignment} object have been created with \texttt{intersect_GroupAssignment()} and \texttt{GroupedFrequencyTable}, \texttt{GroupedScoreTable} or \texttt{ScoringTable} have been created with two \texttt{GroupConditions} objects.
- \texttt{is.Simulated()} checks if the \texttt{FrequencyTable} or \texttt{ScoreTable} have been created on basis of simulated distribution (based on \texttt{SimFrequencyTable()})

Usage

\begin{verbatim}
  is.GroupConditions(x)
  is.GroupAssignment(x)
  is.intersected(x)
  is.ScaleSpec(x)
  is.CombScaleSpec(x)
  is.FrequencyTable(x)
  is.GroupedFrequencyTable(x)
  is.Simulated(x)
  is.ScoreTable(x)
  is.GroupedScoreTable(x)
  is.ScoringTable(x)
  is.StandardScale(x)
\end{verbatim}

Arguments

\begin{verbatim}
  x        any \texttt{R} object
\end{verbatim}
normalize_score  Normalize raw scores

Description

Use computed FrequencyTable or ScoreTable to normalize the provided raw scores.

Usage

normalize_score(x, table, what)

Arguments

x  vector of raw scores to normalize

table  FrequencyTable or ScoreTable object

what  the values to get. One of either:

• quan - the quantile of x in the raw score distribution
• Z - normalized Z score for the x raw score
• name of the scale calculated in ScoreTable provided to table argument

Value

Numeric vector with values specified in what argument

See Also

Other score-normalization functions: normalize_scores_df(), normalize_scores_grouped(), normalize_scores_scoring()

Examples

# normalize with FrequencyTable
suppressMessages(
  ft <- FrequencyTable(HEXACO_60$HEX_H)
)
normalize_score(HEXACO_60$HEX_H[1:5], ft, what = "Z")

# normalize with ScoreTable
st <- ScoreTable(ft, list(STEN, STANINE))
normalize_score(HEXACO_60$HEX_H[1:5], st, what = "sten")
normalize_score(HEXACO_60$HEX_H[1:5], st, what = "stanine")
normalize_scores_df

Normalize raw scores for multiple variables

Description

Wrapper for normalize_score() that works on data frame and multiple variables

Usage

normalize_scores_df(data, vars, ..., what, retain = FALSE, .dots = list())

Arguments

data
data.frame containing raw scores

vars
names of columns to normalize. Length of vars need to be the same as number of tables provided to either ... or .dots

... ScoreTable or FrequencyTable objects to be used for normalization

what
the values to get. One of either:
  • quan - the quantile of x in the raw score distribution
  • Z - normalized Z score for the x raw score
  • name of the scale calculated in ScoreTables provided to ... or .dots argument

retain
either boolean: TRUE if all columns in the data are to be retained, FALSE if none; or character vector with names of columns to be retained

.dots ScoreTable or FrequencyTable objects provided as a list, instead of individually in ....

Value

data.frame with normalized scores

See Also

Other score-normalization functions: normalize_scores_grouped(), normalize_scores_scoring(), normalize_score()

Examples

# normalize multiple variables with FrequencyTable
suppressMessages({
  ft_H <- FrequencyTable(HEXACO_60$HEX_H)
  ft_E <- FrequencyTable(HEXACO_60$HEX_E)
  ft_X <- FrequencyTable(HEXACO_60$HEX_X)
})

normalize_scores_df(data = head(HEXACO_60),
normalize_scores_grouped

Normalize scores using GroupedFrequencyTables or GroupedScoreTables

Description

Normalize scores using either GroupedFrequencyTable or GroupedScoreTable for one or more variables. Given data.frame should also contain columns used in GroupingConditions attached to the table.

Usage

normalize_scores_grouped(
  data,
  vars,
  ..., what,
  retain = FALSE,
  group_col = NULL,
  .dots = list()
)

Arguments

data data.frame object containing raw scores

vars names of columns to normalize. Length of vars need to be the same as number of tables provided to either ... or .dots

... GroupedFrequencyTable or GroupedScoreTable objects to be used for normalization. They should be provided in the same order as vars
normalize_scores_grouped

what the values to get. One of either:
  • quan - the quantile of x in the raw score distribution
  • Z - normalized Z score for the x raw score
  • name of the scale calculated in GroupedScoreTables provided to ... or .dots argument

retain either boolean: TRUE if all columns in the data are to be retained, FALSE if none; or character vector with names of columns to be retained

group_col name of the column for name of the group each observation was qualified into. If left as default NULL, they won’t be returned.

Value data.frame with normalized scores

See Also
Other score-normalization functions: normalize_scores_df(), normalize_scores_scoring(), normalize_score()

Examples

# setup - create necessary objects # suppressMessages({
  age_grouping <- GroupConditions(
    conditions_category = "Age",
    "below 22" ~ age < 22,
    "23-60" ~ age >= 23 & age <= 60,
    "above 60" ~ age > 60
  )
  sex_grouping <- GroupConditions(
    conditions_category = "Sex",
    "Male" ~ sex == "M",
    "Female" ~ sex == "F"
  )
  NEU_gft <- GroupedFrequencyTable(
    data = IPIP_NEO_300, 
    conditions = list(age_grouping, sex_grouping),
    var = "N"
  )
  NEU_gst <- GroupedScoreTable(
    NEU_gft,
    scale = list(STEN, STANINE)
  )
})

##### normalize scores #####
# to Z score or quantile using GroupedFrequencyTable
normalized_to_quan <- normalize_scores_grouped(
normalize_scores_scoring

Normalize scores using ScoringTables

Description

Normalize scores using either ScoringTable objects for one or more variables. Given data.frame should also contain columns used in GroupingConditions attached to the table (if any).

Usage

normalize_scores_scoring(
  data,
  vars,
  ..., retain = FALSE,
  group_col = NULL,
  .dots = list()
)

Arguments

data = data.frame containing raw scores
vars = names of columns to normalize. Length of vars need to be the same as number of tables provided to either ... or .dots
ScoringTable objects to be used for normalization. They should be provided in the same order as vars.

retain: either boolean: TRUE if all columns in the data are to be retained, FALSE if none; or names of columns to be retained.

group_col: name of the column for name of the group each observation was qualified into. If left as default NULL, they won’t be returned. Ignored if no conditions are available.

dots: ScoringTable objects provided as a list, instead of individually.

Value
data.frame with normalized scores

See Also
Other score-normalization functions: `normalize_scores_df()`, `normalize_scores_grouped()`, `normalize_score()`

Examples

```r
# Scoring table to export / import #
suppressMessages(
  Consc_ST <-
    GroupedFrequencyTable(
      data = IPIP_NEO_300,
      conditions = GroupConditions("Sex", "M" ~ sex == "M", "F" ~ sex == "F"),
      var = "C") |>
    GroupedScoreTable(scale = STEN) |>
    to_ScoringTable(min_raw = 60, max_raw = 300)
)

# normalize scores
Consc_norm <-
  normalize_scores_scoring(
    data = IPIP_NEO_300,
    vars = "C",
    Consc_ST,
    group_col = "Group"  
  )

str(Consc_norm)
```

---

**plot.GroupedFrequencyTable**

*Generic plot of the GroupedFrequencyTable*
Description

Generic plot using ggplot2. It plots FrequencyTables for all groups by default, or only chosen ones using when group_names argument is specified.

Usage

```r
## S3 method for class 'GroupedFrequencyTable'
plot(
  x,
  group_names = NULL,
  strict_names = TRUE,
  plot_grid = is.intersected(x),
  ...
)
```

Arguments

- **x**: A GroupedFrequencyTable object
- **group_names**: vector specifying which groups should appear in the plots
- **strict_names**: If TRUE, then intersected groups are filtered using strict strategy: group_names need to be provided in form: "group1:group2". If FALSE, then intersected groups will be taken into regard separately, so eg. when "group1" is provided to group_names, all of: "group1:group2", "group1:group3", "group1:groupN" will be plotted. Defaults to TRUE
- **plot_grid**: boolean indicating if the `ggplot2::facet_grid()` should be used. If FALSE, then `ggplot2::facet_wrap()` is used. If groups are not intersected, then it will be ignored and facet_wrap will be used.
- **...**: named list of additional arguments passed to facet function used.

---

**plot.GroupedScoreTable**

Generic plot of the GroupedScoreTable

Description

Generic plot using ggplot2. It plots ScoreTables for all groups by default, or only chosen ones using when group_names argument is specified.

Usage

```r
## S3 method for class 'GroupedScoreTable'
plot(
  x,
  scale_name = NULL,
  group_names = NULL,
  ...
)
```
strict_names = TRUE,
plot_grid = is.intersected(x),
...
)

Arguments

- **x**: A GroupedScoreTable object
- **scale_name**: if scores for multiple scales available, provide the name of the scale for plotting.
- **group_names**: names specifying which groups should appear in the plots
- **strict_names**: If TRUE, then intersected groups are filtered using strict strategy: group_names need to be provided in form: "group1:group2". If FALSE, then intersected groups will be taken into regard separately, so eg. when "group1" is provided to group_names, all of: "group1:group2", "group1:group3", "group1:groupN" will be plotted. Defaults to TRUE
- **plot_grid**: boolean indicating if the ggplot2::facet_grid() should be used. If FALSE, then ggplot2::facet_wrap() is used. If groups are not intersected, then it will be ignored and facet_wrap will be used.
- **...**: named list of additional arguments passed to facet function.

ScaleSpec

Scale Specification object

Description

Object containing scale or factor specification data. It describes the scale or factor, with regard to which items from the source data are part of it, which need to be summed with reverse scoring, and how to handle NAs. To be used with sum_items_to_scale() function to preprocess item data.

Usage

ScaleSpec(
  name,
  item_names,
  min,
  max,
  reverse = character(0),
  na_strategy = c("asis", "mean", "median", "mode"),
  na_value = as.integer(NA),
  na_value_custom
)

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

## S3 method for class 'ScaleSpec'
summary(object, ...)
Arguments

name character with name of the scale/factor
item_names character vector containing names of the items that the scale/factor consists of.
min, max integer containing the default minimal/maximal value that the answer to the item can be scored as.
reverse character vector containing names of the items that need to be reversed during scale/factor summing. Reversed using the default "min" and "max" values.
na_strategy character vector specifying which strategy should be taken during filling of NA. Defaults to "asis" and, other options are "mean", "median" and "mode". Strategies are explained in the details section.
na_value integer value to be input in missing values as default. Defaults to as.integer(NA).
na_value_custom if there are any need for specific questions be gives specific values in place of NAs, provide a named integer vector there. Names should be the names of the questions.
x a ScaleSpec object
... further arguments passed to or from other methods.
object a ScaleSpec object

Details

NA imputation:
it specifies how NA values should be treated during sum_items_to_scale() function run. asis strategy is literal: the values specified in na_value or na_value_custom will be used without any changes. mean, median and mode are functional strategies. They work on a rowwise basis, so the appropriate value for every observation will be used. If there are no values provided to check for the mean, median or mode, the value provided in na_value or na_value_custom will be used. The values of mean and median will be rounded before imputation.

Order of operations:
• item reversion
• functional NAs imputation
• literal NAs imputation

Value

object of ScaleSpec class
data.frame of item names, if they are reversed, and custom NA value if available, invisibly

See Also

Other item preprocessing functions: CombScaleSpec(), sum_items_to_scale()
Examples

# simple scale specification

```r
simple_scaleSpec <- ScaleSpec(
  name = "simple",
  # scale consists of 5 items
  item_names = c("item_1", "item_2", "item_3", "item_4", "item_5"),
  # item scores can take range of values: 1-5
  min = 1,
  max = 5,
  # item 2 and 5 need to be reversed
  reverse = c("item_2", "item_5"))

print(simple_scaleSpec)
```

# scale specification with literal NA imputation strategy

```r
asis_scaleSpec <- ScaleSpec(
  name = "w_asis",
  item_names = c("item_1", "item_2", "item_3", "item_4", "item_5"),
  min = 1,
  max = 5,
  reverse = "item_2",
  # na values by default will be filled with '3'
  na_value = 3,
  # except for item_4, where they will be filled with '2'
  na_value_custom = c(item_4 = 2)
)

print(asis_scaleSpec)
```

# scale specification with functional NA imputation strategy

```r
func_scaleSpec <- ScaleSpec(
  name = "w_func",
  item_names = c("item_1", "item_2", "item_3", "item_4", "item_5"),
  min = 1,
  max = 5,
  reverse = "item_2",
  # strategies available are 'mean', 'median' and 'mode'
  na_strategy = "mean"
)

print(func_scaleSpec)
```
Description

Creates a table to calculate scores in specified standardized scale for each discrete raw score. Uses normalization provided by `FrequencyTable()` and scale definition created with `StandardScale()`. After creation it can be used to normalize and standardize raw scores with `normalize_score()` or `normalize_scores_df()`.

`plot.ScoreTable()` method requires `ggplot2` package to be installed.

Usage

```r
ScoreTable(ft, scale)
```

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

## S3 method for class 'ScoreTable'
plot(x, scale_name = NULL, ...)

Arguments

- **ft**
  a `FrequencyTable` object

- **scale**
  a `StandardScale` object or list of multiple `StandardScale` objects

- **x**
  a `ScoreTable` object

- **...**
  further arguments passed to or from other methods

- **scale_name**
  if scores for multiple scales available, provide the name of the scale for plotting.

Value

object of class `ScoreTable`. Consists of:

- **table**: data.frame containing for each point in the raw score:
  - number of observations (n),
  - frequency in sample (freq),
  - quantile (quan),
  - normalized Z-score (Z),
  - score transformed to every of provided `StandardScales`

- **status**: list containing the total number of simulated observations (n) and information about raw scores range completion (range): complete or incomplete

- **scale**: named list of all attached `StandardScale` objects

Examples

```r
# firstly compute FrequencyTable for a variable
ft <- FrequencyTable(HEXACO_60$HEX_A)

# then create a ScoreTable
st <- ScoreTable(ft, STEN)
```
SimFrequencyTable

Generate FrequencyTable using simulated distribution

Description

It is always best to use raw scores for computing the FrequencyTable. They aren’t always available - in that case, this function can be used to simulate the distribution given its descriptive statistics. This simulation should be always treated as an estimate.

The distribution is generated using the Fleishmann method from SimMultiCorrData::nonnormvar1() function. The SimMultiCorrData package needs to be installed.

Usage

SimFrequencyTable(min, max, M, SD, skew = 0, kurt = 3, n = 10000, seed = NULL)

Arguments

- **min**: minimum value of raw score
- **max**: maximum value of raw score
- **M**: mean of the raw scores distribution
- **SD**: standard deviation of the raw scores distribution
- **skew**: skewness of the raw scores distribution. Defaults to 0 for normal distribution
- **kurt**: kurtosis of the raw scores distribution. Defaults to 3 for normal distribution
- **n**: number of observations to simulate. Defaults to 10000, but greater values could be used to generate better estimates. Final number of observations in the generated Frequency Table may be less - all values lower than min and higher than max are filtered out.
- **seed**: the seed value for random number generation

Value

FrequencyTable object created with simulated data. Consists of:

- **table**: data.frame with number of observations (n), frequency in sample (freq), quantile (quan) and normalized Z-score (Z) for each point in raw score
- **status**: list containing the total number of simulated observations (n) and information about raw scores range completion (range): complete or incomplete
### SLCS

**Sample data of SLCS questionnaire results**

**Description**

Dataset containing individual items answers of SLCS questionnaire. They were obtained during 2020 study on Polish incidental sample.

**Usage**

SLCS

**Format**

A data frame with 103 rows and 19 variables

- **user_id**: identity anonymized with ‘ids::adjective_animal’
- **sex**: sex of the participant (‘M’ale, ‘F’emale or ‘O’ther)
- **age**: age of the participant (15–68)
- **SLCS_1, SLCS_2, SLCS_3, SLCS_4, SLCS_5, SLCS_6, SLCS_7, SLCS_8, SLCS_9, SLCS_10, SLCS_11, SLCS_12, SLCS_13, SLCS_14, SLCS_15, SLCS_16**: Score for each of measure items. (1–5)

**Details**

All SLCS item responses can take integer values 1-5. The measure consists of two sub-scales: Self-Liking and Self-Competence, and the General Score can also be calculated. Below are the item numbers that are used for each sub-scale (R near the number means that the item need to be reversed.)

- **Self-Liking**: 1R, 3, 5, 6R, 7R, 9, 11, 15R
- **Self-Competence**: 2, 4, 8R, 10R, 12, 13R, 14, 16
- **General Score**: All of the above items (they need to be reversed as in sub-scales)

### StandardScale

**Specify standard scale**

**Description**

StandardScale objects are used with ScoreTable() or GroupedScoreTable() objects to recalculate FrequencyTable() or GroupedFrequencyTable() into some standardized scale score.

There are few StandardScale defaults available. Plot method requires ggplot2 package to be installed.
Usage

StandardScale(name, M, SD, min, max)

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

## S3 method for class 'StandardScale'
plot(x, n = 1000, ...)

Arguments

name | Name of the scale
M | Mean of the scale
SD | Standard deviation of the scale
min | Minimal value the scale takes
max | Maximal value the scale takes
x | a StandardScale object
... | further arguments passed to or from other methods.
n | Number of points the plot generates. The higher the number, the more detailed are the plots. Default to 1000 for nicely detailed plot.

Value

StandardScale object

---

strip_ScoreTable | Revert the ScoreTable back to FrequencyTable object.

Description

Revert the ScoreTable back to FrequencyTable object.

Usage

strip_ScoreTable(x)

Arguments

x | a ScoreTable object
Examples

```r
# having a ScoreTable object
st <- ScoreTable(FrequencyTable(HEXACO$HEX_X), TANINE)
class(st)

# revert it back to the FrequencyTable
ft <- strip_ScoreTable(st)
class(ft)
```

sum_items_to_scale  Sum up discrete raw data

Description

Helper function to sum-up and - if needed - automatically reverse discrete raw item values to scale or factor that they are measuring.

Usage

```r
sum_items_to_scale(data, ..., retain = FALSE, .dots = list())
```

Arguments

data  data.frame object containing numerical values of items data

...  objects of class ScaleSpec or CombScaleSpec. If all item names are found in data, summed items will be available in returned data.frame as column named as their name value.

retain  either boolean: TRUE if all columns in the data are to be retained, FALSE if none, or character vector with names of columns to be retained

.dots  ScaleSpec or CombScaleSpec objects provided as a list, instead of individually in ....

Details

All summing up of the raw discrete values into scale or factor score is done according to provided specifications utilizing ScaleSpec() objects. For more information refer to their constructor help page.

Value

object of class data.frame

See Also

Other item preprocessing functions: CombScaleSpec(), ScaleSpec()
Examples

# create the Scale Specifications for SLCS dataset
## Self-Liking specification
SL_spec <- ScaleSpec(
  name = "Self-Liking",
  item_names = paste("SLCS", c(1, 3, 5, 6, 7, 9, 11, 15), sep = " "),
  reverse = paste("SLCS", c(1, 6, 7, 15), sep = " "),
  min = 1,
  max = 5)

## Self-Competence specification
SC_spec <- ScaleSpec(
  name = "Self-Competence",
  item_names = paste("SLCS", c(2, 4, 8, 10, 12, 13, 14, 16), sep = " "),
  reverse = paste("SLCS", c(8, 10, 13), sep = " "),
  min = 1,
  max = 5)

## General Score specification
GS_spec <- CombScaleSpec(
  name = "General Score",
  SL_spec,
  SC_spec)

# Sum the raw item scores to raw scale scores
SLCS_summed <- sum_items_to_scale(SLCS, SL_spec, SC_spec, GS_spec, retain = "user_id")
summary(SLCS_summed)

---

Create ScoringTable

Description

ScoringTable is a simple version of ScoreTable() or GroupedScoreTable(), that don’t include the FrequencyTable internally. It can be easily saved to csv or json using export_ScoringTable() and loaded from these files using import_ScoringTable().

When using GroupedScoreTable, the columns will be named the same as the name of group. If it was created using two GroupCondition object, the names of columns will be names of the groups seperated by :

Usage

to_ScoringTable(table, ...)

## S3 method for class 'ScoreTable'
to_ScoringTable(
  table,
  scale = NULL,
to_ScoringTable

```r

min_raw = NULL,
max_raw = NULL,
score_colname = "Score",
...
)

## S3 method for class 'GroupedScoreTable'
to_ScoringTable(table, scale = NULL, min_raw = NULL, max_raw = NULL, ...)

## S3 method for class 'ScoreTable'
summary(object, ...)

Arguments

- `table` ScoreTable or GroupedScoreTable object
- `...` further arguments passed to or from other methods.
- `scale` name of the scale attached in table. If only one scale is attached, it can be left as default NULL
- `min_raw, max_raw` absolute minimum/maximum score that can be received. If left as default NULL, the minimum/maximum available in the data will be used.
- `score_colname` Name of the column containing the raw scores
- `object` ScoringTable object

Value

ScoringTable object

Examples

```r

Extr_ST <-
  # create FrequencyTable
  FrequencyTable(data = IPIP_NEO_300$E) |>
  # create ScoreTable
  ScoreTable(scale = STEN) |>
  # and transform into ScoringTable
to_ScoringTable(
    min_raw = 60,
    max_raw = 300
  )

summary(Extr_ST)

#### GroupConditions creation ####

sex_grouping <- GroupConditions(
  conditions_category = "Sex",
  "Male" ~ sex == "M",
  "Female" ~ sex == "F"
)
```

```r
#### Creating ScoringTable ####
## based on grouped data ##

Neu_ST <-
  # create FrequencyTable
  GroupedFrequencyTable(
    data = IPIP_NEO_300,
    conditions = sex_grouping,
    var = "N") |>
  # create ScoreTable
  GroupedScoreTable(
    scale = STEN) |>
  # and transform into ScoringTable
  to_ScoringTable(
    min_raw = 60,
    max_raw = 300
  )

summary(Neu_ST)
```
Index

* datasets
  HEXACO_60, 19
  IPIP_NEO_300, 24
  SLCS, 38

* import/export functions
  export_ScaleSpec, 7
  export_ScoringTable, 8
  import_ScaleSpec, 19
  import_ScoringTable, 21

* item preprocessing functions
  CombScaleSpec, 3
  ScaleSpec, 33
  sum_items_to_scale, 40

* observation grouping functions
  extract_observations, 10
  GroupAssignment, 13
  intersect_GroupAssignment, 22

* score-normalization functions
  normalize_score, 26
  normalize_scores_df, 27
  normalize_scores_grouped, 28
  normalize_scores_scoring, 30
  as.data.frame.GroupConditions
    (GroupConditions), 15
  attach_scales, 2
  CombScaleSpec, 3, 34, 40
  CompScoreTable, 4
  default_scales, 7
  export_ScaleSpec, 7, 9, 20, 21
  export_ScaleSpec(), 19
  export_ScoringTable, 7, 8, 20, 21
  export_ScoringTable(), 21, 41
  extract_observations, 10, 14, 23
  FrequencyTable, 12
  FrequencyTable(), 17, 36, 38
  ggplot2::facet_grid(), 32, 33
  ggplot2::facet_wrap(), 32, 33
  GroupAssignment, 11, 13, 23
  GroupConditions, 15
  GroupConditions(), 17
  GroupedFrequencyTable, 17
  GroupedFrequencyTable(), 14, 38
  GroupedScoreTable, 18
  GroupedScoreTable(), 38, 41
  HEXACO_60, 19
  import_ScaleSpec, 7, 9, 19, 21
  import_ScaleSpec(), 7
  import_ScoringTable, 7, 9, 20, 21
  import_ScoringTable(), 8, 41
  intersect_GroupAssignment, 11, 14, 22
  intersect_GroupAssignment(), 25
  IPIP_NEO_300, 24
  is.CombScaleSpec(is_stenR_classes), 25
  is.FrequencyTable(is_stenR_classes), 25
  is.GroupAssignment(is_stenR_classes), 25
  is.GroupConditions(is_stenR_classes), 25
  is.GroupedFrequencyTable
    (is_stenR_classes), 25
  is.GroupedScoreTable
    (is_stenR_classes), 25
  is.intersected(is_stenR_classes), 25
  is.ScaleSpec(is_stenR_classes), 25
  is.ScoreTable(is_stenR_classes), 25
  is.ScoringTable(is_stenR_classes), 25
  is.Simulated(is_stenR_classes), 25
  is.StandardScale(is_stenR_classes), 25
  is_stenR_classes, 25
  normalize_score, 26, 27, 29, 31
  normalize_score(), 27, 36
  normalize_scores_df, 26, 27, 29, 31
normalize_scores_df(), 36
normalize_scores_grouped(), 26, 27, 28, 31
normalize_scores_grouped(), 17
normalize_scores_scoring(), 26, 27, 29, 30
plot.FrequencyTable(FrequencyTable), 12
plot.GroupedFrequencyTable, 31
plot.GroupedScoreTable, 32
plot.ScoreTable(ScoreTable), 35
plot.ScoreTable(), 36
plot.StandardScale(StandardScale), 38
print.CombScaleSpec(CombScaleSpec), 3
print.FrequencyTable(FrequencyTable), 12
print.GroupAssignment(GroupAssignment), 13
print.GroupConditions(GroupConditions), 15
print.GroupedFrequencyTable(GroupedFrequencyTable), 17
print.GroupedScoreTable(GroupedScoreTable), 18
print.ScaleSpec(ScaleSpec), 33
print.ScoreTable(ScoreTable), 35
print.StandardScale(StandardScale), 38
ScaleSpec, 3, 33, 40
ScaleSpec(), 40
ScoreTable, 5, 35
ScoreTable(), 12, 38, 41
SimFrequencyTable, 37
SimFrequencyTable(), 13, 25
SimMultiCorrData::nonnormvar1(), 37
SLCS, 38
StandardScale, 38
StandardScale(), 7, 12, 36
STANINE(default_scales), 7
STEN(default_scales), 7
strip_ScoreTable, 39
sum_items_to_scale, 3, 34, 40
sum_items_to_scale(), 3, 24, 33, 34
summary.CombScaleSpec(CombScaleSpec), 3
summary.FrequencyTable(FrequencyTable), 12
summary.GroupAssignment(GroupAssignment), 13
summary.GroupedFrequencyTable(GroupedFrequencyTable), 17
summary.ScoreTable(ScoreTable), 33
summary.ScoreTable(to_ScoringTable), 41
TANINE(default_scales), 7
TETRONIC(default_scales), 7
to_ScoringTable, 41
WECHSLER_IQ(default_scales), 7