Package ‘junco’

February 11, 2026

Title Create Common Tables and Listings Used in Clinical Trials

Version 0.1.4

Date 2026-01-12

Description Structure and formatting requirements for clinical trial table and listing outputs vary between pharmaceutical companies. 'junco' provides additional tooling for use alongside the 'rtables', 'rlistings' and 'tern' packages when creating table and listing outputs. While motivated by the specifics of Johnson and Johnson Clinical and Statistical Programming's table and listing shells, 'junco' provides functionality that is general and reusable. Major features include a) alternative and extended statistical analyses beyond what 'tern' supports for use in standard safety and efficacy tables, b) a robust production-grade Rich Text Format (RTF) exporter for both tables and listings, c) structural support for spanning column headers and risk difference columns in tables, and d) robust font-aware automatic column width algorithms for both listings and tables.

License Apache License (>= 2)

URL https://github.com/johnsonandjohnson/junco, https://johnsonandjohnson.github.io/junco/

BugReports https://github.com/johnsonandjohnson/junco/issues

Depends R (>= 4.4), formatters (>= 0.5.12), rtables (>= 0.6.15)

Imports tidytlg (>= 0.11.0), tern (>= 0.9.10), rlistings (>= 0.2.13), checkmate (>= 2.1.0), rbmi (>= 1.6.0), broom, methods, dplyr, generics, stats, survival, tibble, utils, emmeans, mmrm, assertthat, vcdExtra (>= 0.8.7), rtables.officer (>= 0.1.0), flextable (>= 0.9.10), officer, xml2, ggplot2, stringi, systemfonts

Encoding UTF-8

Language en-US

RoxygenNote 7.3.3

Suggests knitr, rmarkdown, forcats (>= 1.0.0), testthat (>= 3.0.0), mockery, mvtnorm, parallel, readxl, rlang, tidyr, pharmaverseadamjnj (>= 0.0.2)

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation no

Author Gabriel Becker [cre, aut] (Original creator of the package, and author of included formatters functions),
Ilse Augustyns [aut],
Paul Jenkins [aut],
Daniel Hofstaedter [aut],
Joseph Kovach [aut],
David Munoz Tord [aut],
Daniel Sabanes Bove [aut],
Ezequiel Anokian [ctb],
Renfei Mao [ctb],
Mrinal Das [ctb],
Wojciech Wojciak [ctb],
Isaac Gravestock [cph] (Author of included rbmi functions),
Joe Zhu [cph] (Author of included tern and rtables.officer functions),
Johnson & Johnson Innovative Medicine [cph, fnd],
F. Hoffmann-La Roche AG [cph] (Copyright holder of included formatters, rtables.officer and tern functions)

Maintainer Gabriel Becker <gabembecker@gmail.com>

Repository CRAN

Date/Publication 2026-02-11 22:30:02 UTC

Contents

`analyze_values` *Shortcut Layout Function for Standard Continuous Variable Analysis*

Description

Shortcut Layout Function for Standard Continuous Variable Analysis

Usage

```
analyze_values(lyt, vars, ..., formats)
```

Arguments

`lyt`	(`layout`) input layout where analyses will be added to.
`vars`	(`character`) variable names for the primary analysis variable to be iterated over.
`...`	additional arguments for the lower level functions.
`formats`	(`list`) formats including `mean_sd`, `median` and `range` specifications.

Value

Modified layout.

a_eair100_j *Exposure-Adjusted Incidence Rate*

Description

Statistical/Analysis Function for presenting Exposure-Adjusted Incidence Rate summary data

Usage

```
a_eair100_j(
  df,
  labelstr = NULL,
  .var,
  .df_row,
  .spl_context,
  .alt_df_full = NULL,
  id = "USUBJID",
  drop_levels = FALSE,
  riskdiff = TRUE,
  ref_path = NULL,
  .stats = c("eair"),
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL,
  na_str = rep("NA", 3),
  conf_level = 0.95,
  fup_var,
  occ_var,
  occ_dy
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`labelstr`	(`string`) label string for the row.
`.var`	(`string`) variable name for analysis.
`.df_row`	(`data.frame`) data frame across all of the columns for the given row split.
`.spl_context`	(`data.frame`) gives information about ancestor split states.
`.alt_df_full`	(`dataframe`) denominator dataset for calculations.

`id`	(string) subject variable name.
`drop_levels`	(logical) if TRUE, non-observed levels will not be included.
`riskdiff`	(logical) if TRUE, risk difference calculations will be performed.
`ref_path`	(string) column path specifications for the control group.
`.stats`	(character) statistics to select for the table.
`.formats`	(named 'character' or 'list') formats for the statistics.
`.labels`	(named 'character') labels for the statistics.
`.indent_mods`	(named integer) indent modifiers for the labels.
`na_str`	(string) string used to replace all NA or empty values in the output.
`conf_level`	(proportion) confidence level of the interval.
`fup_var`	(string) variable name for follow-up time.
`occ_var`	(string) variable name for occurrence.
`occ_dy`	(string) variable name for occurrence day.

Value

- `a_eair100_j` returns the corresponding list with formatted `rtables::CellValue()`.

Functions

- `a_eair100_j()`: Formatted analysis function for exposure adjusted incidence rate summary which is used as `afun` in `analyze` or `cfun` in `summarize_row_groups`.

Examples

```
library(tern)
library(dplyr)
trtvar <- "ARM"
ctrl_grp <- "B: Placebo"
cutoffd <- as.Date("2023-09-24")


adexsum <- ex_adsl |>
```

```
    create_colspan_var(
      non_active_grp          = ctrl_grp,
      non_active_grp_span_lbl = " ",
      active_grp_span_lbl     = "Active Study Agent",
      colspan_var             = "colspan_trt",
      trt_var                 = trtvar
    ) |>
    mutate(
      rrisk_header = "Risk Difference (95% CI)",
      rrisk_label = paste(!!rlang::sym(trtvar), "vs", ctrl_grp),
      TRTDURY = case_when(
        !is.na(EOSDY) ~ EOSDY,
        TRUE ~ as.integer(cutoffd - as.Date(TRTSDTM) + 1)
      )
    ) |>
    select(USUBJID, !!rlang::sym(trtvar), colspan_trt, rrisk_header, rrisk_label, TRTDURY)

  adexsum$TRTDURY <- as.numeric(adexsum$TRTDURY)

  adae <- ex_adae |>
    group_by(USUBJID, AEDECOD) |>
    select(USUBJID, AEDECOD, ASTDY) |>
    mutate(rwnum = row_number()) |>
    mutate(AOCCPFL = case_when(
      rwnum == 1 ~ "Y",
      TRUE ~ NA
    )) |>
    filter(AOCCPFL == "Y")

  aefup <- left_join(adae, adexsum, by = "USUBJID")

  colspan_trt_map <- create_colspan_map(adexsum,
    non_active_grp = ctrl_grp,
    non_active_grp_span_lbl = " ",
    active_grp_span_lbl = "Active Study Agent",
    colspan_var = "colspan_trt",
    trt_var = trtvar
  )

  ref_path <- c("colspan_trt", " ", trtvar, ctrl_grp)

  ################################################################################
  # Define layout and build table:
  ################################################################################

  lyt <- basic_table(show_colcounts = TRUE, colcount_format = "N=xx", top_level_section_div = " ") |>
    split_cols_by("colspan_trt", split_fun = trim_levels_to_map(map = colspan_trt_map)) |>
    split_cols_by(trtvar) |>
    split_cols_by("rrisk_header", nested = FALSE) |>
    split_cols_by(trtvar, labels_var = "rrisk_label", split_fun = remove_split_levels(ctrl_grp)) |>
    analyze("TRTDURY",
      nested = FALSE,
      show_labels = "hidden",
      afun = a_patyrs_j
    ) |>
    analyze(
      vars = "AEDECOD",
      nested = FALSE,
      afun = a_eair100_j,
      extra_args = list(
        fup_var = "TRTDURY",
        occ_var = "AOCCPFL",
        occ_dy = "ASTDY",
        ref_path = ref_path,
        drop_levels = TRUE
      )
    )

  result <- build_table(lyt, aefup, alt_counts_df = adexsum)
  head(result, 5)
```

`a_freq_combos_j` *Analysis function count and percentage in column design controlled by combosdf*

Description

Analysis function count and percentage in column design controlled by combosdf

Usage

```
a_freq_combos_j(
  df,
  labelstr = NULL,
  .var = NA,
  val = NULL,
  combosdf = NULL,
  do_not_filter = NULL,
  filter_var = NULL,
  flag_var = NULL,
  .df_row,
  .spl_context,
  .N_col,
  id = "USUBJID",
  denom = c("N_col", "n_df", "n_altdf", "n_rowdf", "n_parentdf"),
  label = NULL,
  label_fstr = NULL,
  label_map = NULL,
  .alt_df_full = NULL,
  denom_by = NULL,
  .stats = "count_unique_denom_fraction",
  .formats = NULL,
  .labels_n = NULL,
  .indent_mods = NULL,
  na_str = rep("NA", 3)
)
```

Arguments

df	(data.frame) data set containing all analysis variables.
labelstr	(character) label of the level of the parent split currently being summarized (must be present as second argument in Content Row Functions). See rtables::summarize_row_groups() for more information.
.var	(string) single variable name that is passed by rtables when requested by a statistics function.
val	(character or NULL) When NULL, all levels of the incoming variable (variable used in the analyze call) will be considered. When a single string, only that current level/value of the incoming variable will be considered. When multiple levels, only those levels/values of the incoming variable will be considered. When no values are observed (eg zero row input df), a row with row-label No data reported will be included in the table.
combosdf	The df which provides the mapping of column facets to produce cumulative counts for .N_col. In the cell facet, these cumulative records must then be removed from the numerator, which can be done via the filter_var parameter to avoid unwanted counting of events.
do_not_filter	A vector of facets (i.e., column headers), identifying headers for which no filtering of records should occur. That is, the numerator should contain cumulative counts. Generally, this will be used for a "Total" column, or something similar.
filter_var	The variable which identifies the records to count in the numerator for any given column. Generally, this will contain text matching the column header for the column associated with a given record.
flag_var	Variable which identifies the occurrence (or first occurrence) of an event. The flag variable is expected to have a value of "Y" identifying that the event should be counted, or NA otherwise.
.df_row	(data.frame) data frame across all of the columns for the given row split.
.spl_context	(data.frame) gives information about ancestor split states that is passed by rtables.

`.N_col`
: (`integer`)
column-wise N (column count) for the full column being analyzed that is typically passed by `rtables`.

`id`
: (`string`)
subject variable name.

`denom`
: (`string`)
One of

- **N_col** Column count,
- **n_df** Number of patients (based upon the main input dataframe `df`),
- **n_altdf** Number of patients from the secondary dataframe (`.alt_df_full`),
Note that argument `denom_by` will perform a row-split on the `.alt_df_full` dataframe.
It is a requirement that variables specified in `denom_by` are part of the row split specifications.
- **n_rowdf** Number of patients from the current row-level dataframe (`.row_df` from the rtables splitting machinery).
- **n_parentdf** Number of patients from a higher row-level split than the current split.
This higher row-level split is specified in the argument `denom_by`.

`label`
: (`string`)
When `val` has length 1, the row label to be shown on the output can be specified using this argument.
When `val` is a `character vector`, the `label_map` argument can be specified to control the row-labels.

`label_fstr`
: (`string`)
a sprintf style format string. It can contain up to one `"%s"`, which takes the current split value and generates the row/column label.
It will be combined with the `labelstr` argument, when utilizing this function as a `cfun` in a `summarize_row_groups` call.
It is recommended not to utilize this argument for other purposes. The label argument could be used instead (if `val` is a single string)

`label_map`
: (`tibble`)
A mapping tibble to translate levels from the incoming variable into a different row label to be presented on the table.

`.alt_df_full`
: (`dataframe`)
Denominator dataset for fraction and relative risk calculations.
this argument gets populated by the rtables split machinery (see rtables::additional_fun_params).

`denom_by`	(`character`) Variables from row-split to be used in the denominator derivation. This controls both `denom = "n_parentdf"` and `denom = "n_altdf"`. When `denom = "n_altdf"`, the denominator is derived from `.alt_df_full` in combination with `denom_by` argument
`.stats`	(`character`) statistics to select for the table.
`.formats`	(named 'character' or 'list') formats for the statistics.
`.labels_n`	(named `character`) String to control row labels for the 'n'-statistics. Only useful when more than one 'n'-statistic is requested (rare situations only).
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.
`na_str`	(`string`) string used to replace all `NA` or empty values in the output.

Value

list of requested statistics with formatted `rtables::CellValue()`.

Examples

```
library(dplyr)
ADSL <- ex_adsl |> select(USUBJID, ARM, EOSSTT, EOSDT, EOSDY, TRTSDTM)

cutoffd <- as.Date("2023-09-24")

ADSL <- ADSL |>
  mutate(
    TRTDURY = case_when(
      !is.na(EOSDY) ~ EOSDY,
      TRUE ~ as.integer(cutoffd - as.Date(TRTSDTM) + 1)
    )
  ) |>
  mutate(ACAT1 = case_when(
    TRTDURY < 183 ~ "0-6 Months",
    TRTDURY < 366 ~ "6-12 Months",
    TRUE ~ "+12 Months"
  )) |>
  mutate(ACAT1 = factor(ACAT1, levels = c("0-6 Months", "6-12 Months", "+12 Months")))


ADAE <- ex_adae |> select(USUBJID, ARM, AEBODSYS, AEDECOD, ASTDY)

ADAE <- ADAE |>
  mutate(TRTEMFL = "Y") |>
  mutate(ACAT1 = case_when(
    ASTDY < 183 ~ "0-6 Months",
    ASTDY < 366 ~ "6-12 Months",
    TRUE ~ "+12 Months"
  )) |>
  mutate(ACAT1 = factor(ACAT1, levels = c("0-6 Months", "6-12 Months", "+12 Months")))

combodf <- tribble(
  ~valname, ~label, ~levelcombo, ~exargs,
  "Tot", "Total", c("0-6 Months", "6-12 Months", "+12 Months"), list(),
  "A_0-6 Months", "0-6 Months", c("0-6 Months", "6-12 Months", "+12 Months"), list(),
  "B_6-12 Months", "6-12 Months", c("6-12 Months", "+12 Months"), list(),
  "C_+12 Months", "+12 Months", c("+12 Months"), list()
)


lyt <- basic_table(show_colcounts = TRUE) |>
  split_cols_by("ARM") |>
  split_cols_by("ACAT1",
   split_fun = add_combo_levels(combosdf = combodf, trim = FALSE, keep_levels = combodf$valname)
  ) |>
  analyze("TRTEMFL",
    show_labels = "hidden",
    afun = a_freq_combos_j,
    extra_args = list(
      val = "Y",
      label = "Subjects with >= 1 AE",
      combosdf = combodf,
      filter_var = "ACAT1",
      do_not_filter = "Tot"
    )
  )


result <- build_table(lyt, df = ADAE, alt_counts_df = ADSL)

result
```

`a_freq_j` *Analysis/statistical function for count and percentage in core columns and (optional) relative risk columns*

Description

Analysis/statistical function for count and percentage in core columns and (optional) relative risk columns

Usage

```
s_freq_j(
  df,
  .var,
  .df_row,
  val = NULL,
  drop_levels = FALSE,
  excl_levels = NULL,
  alt_df,
  parent_df,
  id = "USUBJID",
  denom = c("n_df", "n_altdf", "N_col", "n_rowdf", "n_parentdf"),
  .N_col,
  countsource = c("df", "altdf")
)

a_freq_j(
  df,
  labelstr = NULL,
  .var = NA,
  val = NULL,
  drop_levels = FALSE,
  excl_levels = NULL,
  new_levels = NULL,
  new_levels_after = FALSE,
  addstr2levs = NULL,
  .df_row,
  .spl_context,
  .N_col,
  id = "USUBJID",
  denom = c("N_col", "n_df", "n_altdf", "N_colgroup", "n_rowdf", "n_parentdf"),
  riskdiff = TRUE,
  ref_path = NULL,
  variables = list(strata = NULL),
  conf_level = 0.95,
 method = c("wald", "waldcc", "cmh", "ha", "newcombe", "newcombecc", "strat_newcombe",
    "strat_newcombecc"),
  weights_method = "cmh",
  label = NULL,
  label_fstr = NULL,
  label_map = NULL,
  .alt_df_full = NULL,
  denom_by = NULL,
  .stats = c("count_unique_denom_fraction"),
  .formats = NULL,
  .indent_mods = NULL,
  na_str = rep("NA", 3),
  .labels_n = NULL,
  extrablankline = FALSE,
  extrablanklineafter = NULL,
  restr_columns = NULL,
  colgroup = NULL,
  countsource = c("df", "altdf")
)

a_freq_j_with_exclude(
  df,
  labelstr = NULL,
  exclude_levels,
  .var = NA,
  .spl_context,
  .df_row,
  .N_col,
  .alt_df_full = NULL,
  .stats = "count_unique_denom_fraction",
  .formats = NULL,
  .indent_mods = NULL,
  .labels_n = NULL,
  ...
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`.df_row`	(`data.frame`) data frame across all of the columns for the given row split.
`val`	(`character` or NULL) When NULL, all levels of the incoming variable (variable used in the `analyze` call) will be considered. When a single `string`, only that current level/value of the incoming variable will be considered. When multiple levels, only those levels/values of the incoming variable will be considered. When no values are observed (eg zero row input df), a row with row-label `No data reported` will be included in the table.
`drop_levels`	(`logical`) If `TRUE` non-observed levels (based upon .df_row) will not be included. Cannot be used together with `val`.
`excl_levels`	(`character` or NULL) When NULL, no levels of the incoming variable (variable used in the `analyze` call) will be excluded. When multiple levels, those levels/values of the incoming variable will be excluded. Cannot be used together with `val`.

`alt_df`	(`dataframe`) Will be derived based upon alt_df_full and denom_by within a_freq_j.
`parent_df`	(`dataframe`) Will be derived within a_freq_j based upon the input dataframe that goes into build_table (df) and denom_by. It is a data frame in the higher row-space than the current input df (which underwent row-splitting by the rtables splitting machinery).
`id`	(`string`) subject variable name.
`denom`	(`string`) See Details.
`.N_col`	(`integer`) column-wise N (column count) for the full column being analyzed that is typically passed by `rtables`.
`countsource`	Either `df` or `alt_df`. When `alt_df` the counts will be based upon the alternative dataframe `alt_df`. This is useful for subgroup processing, to present counts of subjects in a subgroup from the alternative dataframe.
`labelstr`	An argument to ensure this function can be used as a `cfun` in a `summarize_row_groups` call. It is recommended not to utilize this argument for other purposes. The label argument could be used instead (if `val` is a single string) An another approach could be to utilize the `label_map` argument to control the row labels of the incoming analysis variable.
`new_levels`	(list(2) or NULL) List of length 2. First element : names of the new levels Second element: list with values of the new levels.
`new_levels_after`	(`logical`) If `TRUE` new levels will be added after last level.
`addstr2levs`	string, if not NULL will be appended to the rowlabel for that level, eg to add ",n (percent)" at the end of the rowlabels
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`riskdiff`	(`logical`) When `TRUE`, risk difference calculations will be performed and presented (if required risk difference column splits are included). When `FALSE`, risk difference columns will remain blank (if required risk difference column splits are included).
`ref_path`	(`string`) Column path specifications for the control group for the relative risk derivation.
`variables`	Will be passed onto the relative risk function (internal function s_rel_risk_val_j), which is based upon `tern::s_proportion_diff()`. See `?tern::s_proportion_diff` for details.

`conf_level`	(`proportion`) confidence level of the interval.
`method`	Will be passed onto the relative risk function (internal function s_rel_risk_val_j).
`weights_method`	Will be passed onto the relative risk function (internal function s_rel_risk_val_j).
`label`	(`string`) When `val` has length 1, the row label to be shown on the output can be specified using this argument. When `val` is a `character vector`, the `label_map` argument can be specified to control the row-labels.
`label_fstr`	(`string`) a sprintf style format string. It can contain up to one `"%s"`, which takes the current split value and generates the row/column label. It will be combined with the `labelstr` argument, when utilizing this function as a `cfun` in a `summarize_row_groups` call. It is recommended not to utilize this argument for other purposes. The label argument could be used instead (if `val` is a single string)
`label_map`	(`tibble`) A mapping tibble to translate levels from the incoming variable into a different row label to be presented on the table.
`.alt_df_full`	(`dataframe`) Denominator dataset for fraction and relative risk calculations. this argument gets populated by the rtables split machinery (see rtables::additional_fun_params).
`denom_by`	(`character`) Variables from row-split to be used in the denominator derivation. This controls both `denom = "n_parentdf"` and `denom = "n_altdf"`. When `denom = "n_altdf"`, the denominator is derived from `.alt_df_full` in combination with `denom_by` argument
`.stats`	(`character`) statistics to select for the table. See Value for list of available statistics.
`.formats`	(named 'character' or 'list') formats for the statistics.
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.
`na_str`	(`string`) string used to replace all `NA` or empty values in the output.
`.labels_n`	(named `character`) String to control row labels for the 'n'-statistics. Only useful when more than one 'n'-statistic is requested (rare situations only).
`extrablankline`	(`logical`) When `TRUE`, an extra blank line will be added after the last value.

	Avoid using this in template scripts, use section_div = " " instead (once PR for rtables is available)
`extrablanklineafter`	(`string`) When the row-label matches the string, an extra blank line will be added after that value.
`restr_columns`	`character` If not NULL, columns not defined in `restr_columns` will be blanked out.
`colgroup`	The name of the column group variable that is used as source for denominator calculation. Required to be specified when `denom = "N_colgroup"`.
`exclude_levels`	(`list`) A named list where names correspond to split variables and values are vectors of levels to exclude.
`...`	additional arguments for the lower level functions.

Details

`denom` controls the denominator used to calculate proportions/percents. It must be one of

- **N_col** Column count,
- **n_df** Number of patients (based upon the main input dataframe `df`),
- **n_altdf** Number of patients from the secondary dataframe (`.alt_df_full`),
  Note that argument `denom_by` will perform a row-split on the `.alt_df_full` dataframe.
  It is a requirement that variables specified in `denom_by` are part of the row split specifications.
- **N_colgroup** Number of patients from the column group variable (note that this is based upon the input .alt_df_full dataframe).
  Note that the argument `colgroup` (column variable) needs to be provided, as it cannot be retrieved directly from the column layout definition.
- **n_rowdf** Number of patients from the current row-level dataframe (`.row_df` from the rtables splitting machinery).
- **n_parentdf** Number of patients from a higher row-level split than the current split.
  This higher row-level split is specified in the argument `denom_by`.

Value

- `s_freq_j`: returns a list of following statistics
  - n_df

  - n_rowdf
  - n_parentdf
  - n_altdf
  - denom
  - count
  - count_unique
  - count_unique_fraction
  - count_unique_denom_fraction

- `a_freq_j`: returns a list of requested statistics with formatted `rtables::CellValue()`. Within the relative risk difference columns, the following stats are blanked out:
  - any of the n-statistics (n_df, n_altdf, n_parentdf, n_rowdf, denom)
  - count
  - count_unique

  For the others (count_unique_fraction, count_unique_denom_fraction), the statistic is replaced by the relative risk difference + confidence interval.

Functions

- `a_freq_j_with_exclude()`: Wrapper for the `afun` which can exclude row split levels from producing the analysis. These have to be specified in the `exclude_levels` argument, see `?do_exclude_split` for details.

Examples

```
library(dplyr)

adsl <- ex_adsl |> select("USUBJID", "SEX", "ARM")
adae <- ex_adae |> select("USUBJID", "AEBODSYS", "AEDECOD")
adae[["TRTEMFL"]] <- "Y"

trtvar <- "ARM"
ctrl_grp <- "B: Placebo"
adsl$colspan_trt <- factor(ifelse(adsl[[trtvar]] == ctrl_grp, " ", "Active Study Agent"),
  levels = c("Active Study Agent", " ")
)

adsl$rrisk_header <- "Risk Difference (%) (95% CI)"
adsl$rrisk_label <- paste(adsl[[trtvar]], paste("vs", ctrl_grp))

adae <- adae |> left_join(adsl)

colspan_trt_map <- create_colspan_map(adsl,
  non_active_grp = "B: Placebo",
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Study Agent",
  colspan_var = "colspan_trt",
  trt_var = trtvar
)
```

```
ref_path <- c("colspan_trt", " ", trtvar, ctrl_grp)

lyt <- basic_table(show_colcounts = TRUE) |>
  split_cols_by("colspan_trt", split_fun = trim_levels_to_map(map = colspan_trt_map)) |>
  split_cols_by(trtvar) |>
  split_cols_by("rrisk_header", nested = FALSE) |>
  split_cols_by(trtvar, labels_var = "rrisk_label", split_fun = remove_split_levels(ctrl_grp))

lyt1 <- lyt |>
  analyze("TRTEMFL",
    show_labels = "hidden",
    afun = a_freq_j,
    extra_args = list(
      method = "wald",
      .stats = c("count_unique_denom_fraction"),
      ref_path = ref_path
    )
  )

result1 <- build_table(lyt1, adae, alt_counts_df = adsl)

result1

x_drug_x <- list(length(unique(subset(adae, adae[[trtvar]] == "A: Drug X")[["USUBJID"]])))
N_x_drug_x <- length(unique(subset(adsl, adsl[[trtvar]] == "A: Drug X")[["USUBJID"]]))
y_placebo <- list(length(unique(subset(adae, adae[[trtvar]] == ctrl_grp)[["USUBJID"]])))
N_y_placebo <- length(unique(subset(adsl, adsl[[trtvar]] == ctrl_grp)[["USUBJID"]]))

tern::stat_propdiff_ci(
  x = x_drug_x,
  N_x = N_x_drug_x,
  y = y_placebo,
  N_y = N_y_placebo
)

x_combo <- list(length(unique(subset(adae, adae[[trtvar]] == "C: Combination")[["USUBJID"]])))
N_x_combo <- length(unique(subset(adsl, adsl[[trtvar]] == "C: Combination")[["USUBJID"]]))

tern::stat_propdiff_ci(
  x = x_combo,
  N_x = N_x_combo,
  y = y_placebo,
  N_y = N_y_placebo
)


extra_args_rr <- list(
  denom = "n_altdf",
  denom_by = "SEX",
  riskdiff = FALSE,
  .stats = c("count_unique")
)
```

```
extra_args_rr2 <- list(
  denom = "n_altdf",
  denom_by = "SEX",
  riskdiff = TRUE,
  ref_path = ref_path,
  method = "wald",
  .stats = c("count_unique_denom_fraction"),
  na_str = rep("NA", 3)
)

lyt2 <- basic_table(
  top_level_section_div = " ",
  colcount_format = "N=xx"
) |>
  split_cols_by("colspan_trt", split_fun = trim_levels_to_map(map = colspan_trt_map)) |>
  split_cols_by(trtvar, show_colcounts = TRUE) |>
  split_cols_by("rrisk_header", nested = FALSE) |>
  split_cols_by(trtvar,
    labels_var = "rrisk_label", split_fun = remove_split_levels("B: Placebo"),
    show_colcounts = FALSE
  ) |>
  split_rows_by("SEX", split_fun = drop_split_levels) |>
  summarize_row_groups("SEX",
    cfun = a_freq_j,
    extra_args = append(extra_args_rr, list(label_fstr = "Gender: %s"))
  ) |>
  split_rows_by("TRTEMFL",
    split_fun = keep_split_levels("Y"),
    indent_mod = -1L,
    section_div = c(" ")
  ) |>
  summarize_row_groups("TRTEMFL",
    cfun = a_freq_j,
    extra_args = append(extra_args_rr2, list(
      label =
        "Subjects with >=1 AE", extrablankline = TRUE
    ))
  ) |>
  split_rows_by("AEBODSYS",
    split_label = "System Organ Class",
    split_fun = trim_levels_in_group("AEDECOD"),
    label_pos = "topleft",
    section_div = c(" "),
    nested = TRUE
  ) |>
  summarize_row_groups("AEBODSYS",
    cfun = a_freq_j,
    extra_args = extra_args_rr2
  ) |>
  analyze("AEDECOD",
    afun = a_freq_j,
    extra_args = extra_args_rr2
  )

result2 <- build_table(lyt2, adae, alt_counts_df = adsl)
```

---

a_freq_resp_var_j *Analysis Function for Response Variables*

Description

This function calculates counts and percentages for response variables (Y/N values), with optional risk difference calculations.

Usage

```
a_freq_resp_var_j(
  df,
  .var,
  .df_row,
  .N_col,
  .spl_context,
  resp_var = NULL,
  id = "USUBJID",
  drop_levels = FALSE,
  riskdiff = TRUE,
  ref_path = NULL,
  variables = formals(s_proportion_diff)$variables,
  conf_level = formals(s_proportion_diff)$conf_level,
 method = c("wald", "waldcc", "cmh", "ha", "newcombe", "newcombecc", "strat_newcombe",
    "strat_newcombecc"),
  weights_method = formals(s_proportion_diff)$weights_method,
  ...
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.var`	(`string`) variable name that is passed by `rtables`.
`.df_row`	(`data.frame`) data frame across all of the columns for the given row split.
`.N_col`	(`integer`) column-wise N (column count) for the full column being analyzed.
`.spl_context`	(`data.frame`) gives information about ancestor split states.

`resp_var`	(`string`) response variable name containing Y/N values.
`id`	(`string`) subject variable name.
`drop_levels`	(`logical`) if TRUE, non-observed levels will not be included.
`riskdiff`	(`logical`) if TRUE, risk difference calculations will be performed.
`ref_path`	(`string`) column path specifications for the control group.
`variables`	(`list`) variables to include in the analysis.
`conf_level`	(`proportion`) confidence level of the interval.
`method`	(`character`) method for calculating confidence intervals.
`weights_method`	(`character`) method for calculating weights.
`...`	Additional arguments passed to other functions.

Value

Formatted analysis function which is used as `afun` in `analyze_vars()` and as `cfun` in `summarize_row_groups()`.

Examples

```
library(dplyr)
ADSL <- ex_adsl |> select(USUBJID, ARM, SEX)

ADAE <- ex_adae |> select(USUBJID, ARM, SEX, AEBODSYS, AEDECOD)

ADAE <- ADAE |>
  mutate(TRTEMFL = "Y")

lyt <- basic_table(show_colcounts = TRUE) |>
  split_cols_by("ARM") |>
  analyze("SEX",
    show_labels = "visible",
    afun = a_freq_resp_var_j,
    extra_args = list(resp_var = "TRTEMFL", riskdiff = FALSE)
  )

result <- build_table(lyt, df = ADAE, alt_counts_df = ADSL)

result
```

a_freq_subcol_j — *Analysis function count and percentage with extra column-subsetting in selected columns (controlled by subcol_* arguments)*

Description

Analysis function count and percentage with extra column-subsetting in selected columns (controlled by subcol_* arguments)

Usage

```
a_freq_subcol_j(
  df,
  labelstr = NULL,
  .var = NA,
  val = NULL,
  subcol_split = NULL,
  subcol_var = NULL,
  subcol_val = NULL,
  .df_row,
  .spl_context,
  .N_col,
  id = "USUBJID",
  denom = c("N_col", "n_df", "n_altdf", "n_rowdf", "n_parentdf"),
  label = NULL,
  label_fstr = NULL,
  label_map = NULL,
  .alt_df_full = NULL,
  denom_by = NULL,
  .stats = c("count_unique_denom_fraction"),
  .formats = NULL,
  .labels_n = NULL,
  .indent_mods = NULL,
  na_str = rep("NA", 3)
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`labelstr`	(`character`) label of the level of the parent split currently being summarized (must be present as second argument in Content Row Functions). See `rtables::summarize_row_groups()` for more information.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.

`val`	(`character` or NULL) When NULL, all levels of the incoming variable (variable used in the `analyze` call) will be considered. When a single `string`, only that current level/value of the incoming variable will be considered. When multiple levels, only those levels/values of the incoming variable will be considered. When no values are observed (eg zero row input df), a row with row-label `No data reported` will be included in the table.
`subcol_split`	(`string`) text to search colid to determine whether further subsetting should be performed.
`subcol_var`	(`string`) name of variable containing to be searched for the text identified in subcol_val argument.
`subcol_val`	(`string`) value to use to perform further data sub-setting.
`.df_row`	(`data.frame`) data frame across all of the columns for the given row split.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`.N_col`	(`integer`) column-wise N (column count) for the full column being analyzed that is typically passed by `rtables`.
`id`	(`string`) subject variable name.
`denom`	(`string`) One of

- **N_col** Column count,
- **n_df** Number of patients (based upon the main input dataframe `df`),
- **n_altdf** Number of patients from the secondary dataframe (`.alt_df_full`), Note that argument `denom_by` will perform a row-split on the `.alt_df_full` dataframe.
  It is a requirement that variables specified in `denom_by` are part of the row split specifications.
- **n_rowdf** Number of patients from the current row-level dataframe (`.row_df` from the rtables splitting machinery).
- **n_parentdf** Number of patients from a higher row-level split than the current split.
  This higher row-level split is specified in the argument `denom_by`.

`label`	(`string`) When `val` has length 1, the row label to be shown on the output can be specified using this argument. When `val` is a `character vector`, the `label_map` argument can be specified to control the row-labels.
`label_fstr`	(`string`) a sprintf style format string. It can contain up to one `"%s"`, which takes the current split value and generates the row/column label. It will be combined with the `labelstr` argument, when utilizing this function as a `cfun` in a `summarize_row_groups` call. It is recommended not to utilize this argument for other purposes. The label argument could be used instead (if `val` is a single string)
`label_map`	(`tibble`) A mapping tibble to translate levels from the incoming variable into a different row label to be presented on the table.
`.alt_df_full`	(`dataframe`) Denominator dataset for fraction and relative risk calculations. this argument gets populated by the rtables split machinery (see rtables::additional_fun_params).
`denom_by`	(`character`) Variables from row-split to be used in the denominator derivation. This controls both `denom = "n_parentdf"` and `denom = "n_altdf"`. When `denom = "n_altdf"`, the denominator is derived from `.alt_df_full` in combination with `denom_by` argument
`.stats`	(`character`) statistics to select for the table.
`.formats`	(named 'character' or 'list') formats for the statistics.
`.labels_n`	(named `character`) String to control row labels for the 'n'-statistics. Only useful when more than one 'n'-statistic is requested (rare situations only).
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.
`na_str`	(`string`) string used to replace all `NA` or empty values in the output.

Value

list of requested statistics with formatted `rtables::CellValue()`.

Examples

```
library(dplyr)
```

```
ADSL <- ex_adsl |>
  select(USUBJID, ARM)

ADSL$COLSPAN_REL <- "AEs"

ADAE <- ex_adae |>
  select(USUBJID, ARM, AEDECOD, AREL)

ADAE <- ADAE |>
  mutate(
    AEREL = case_when(
      AREL == "Y" ~ "RELATED",
      AREL == "N" ~ "NOT RELATED"
    ),
    AEREL = factor(AEREL),
    COLSPAN_REL = "AEs"
  )

combodf <- tribble(
  ~valname,  ~label,        ~levelcombo, ~exargs,
  "RELATED", "Related AEs", c("AEs"),    list()
)

lyt <- basic_table(show_colcounts = TRUE) |>
  split_cols_by("COLSPAN_REL", split_fun = add_combo_levels(combodf, trim = TRUE)) |>
  split_cols_by("ARM") |>
  analyze("AEDECOD",
    afun = a_freq_subcol_j,
    extra_args = list(
      subcol_split = "RELATED",
      subcol_var = "AEREL",
      subcol_val = "RELATED"
    )
  )

result <- build_table(lyt, ADAE, alt_counts_df = ADSL)

result
```

---

a_maxlev   *Calculate Count and Percentage of the Maximum Level of an Ordered Factor per Subject.*

Description

A formatted analysis function used as an `afun` in analyze and as a `cfun` in summarize_row_groups.

It computes count and proportion statistics for the maximum level of an ordered factor, `df[[.var]]`, for each unique subject in `df[[id]]`. Specifically, for each subject, the function identifies the highest level of `df[[.var]]`, producing one value per subject. Then, if `any_level = TRUE`, the function reports the total number of maximum values, excluding those specified in `any_level_exclude`. Otherwise, it tabulates the frequency of each maximum level across all subjects.

This function is particularly useful for identifying the maximum severity of adverse events in a treatment sequence, where the most severe event experienced by a subject is used for reporting.

Usage

```
a_maxlev(
  df,
  labelstr = NULL,
  .var,
  .spl_context,
  id = "USUBJID",
  .alt_df_full = NULL,
  any_level = FALSE,
  any_level_exclude = "Missing",
  ...
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`labelstr`	(`character`) label of the level of the parent split currently being summarized (must be present as second argument in Content Row Functions). See `rtables::summarize_row_groups()` for more information.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`id`	(`string`) subject variable name.
`.alt_df_full`	(`dataframe`) A dataset used to compute the denominator for proportions. This is required when the same subject appears multiple times in the dataset due to treatment sequences. `colnames(.alt_df_full)` must be a superset of `id`. This argument gets populated by the rtables split machinery (see rtables::additional_fun_params).
`any_level`	(`flag`) Should be set to `TRUE` when the function is used as a `cfun`.
`any_level_exclude`	(`character`) Applicable only when `any_level = TRUE`. Specifies levels of `df[[.var]]` to exclude from the statistic (default = "Missing").
`...`	additional arguments for the lower level functions.

Details

For each unique subject, only the maximum level of the ordered factor `df[[.var]]` is included in the final count and percentage statistics.

Value

A `RowsVerticalSection` object.

Note

The denominator for proportions is computed using the `denom_df` argument. This serves as a temporary workaround until the next version of `rtables` is released, which will support `.alt_count_df` for use in `afun`/`cfun`.

Examples

```
treatments <- factor(c("a", "b", "c"))
ae_severities <- c("Missing", "Mild", "Moderate", "Severe")
ae_severities <- ordered(ae_severities, levels = ae_severities)
my_adae <- data.frame(
  ID = c(1, 1, 1, 2, 2, 3, 3, 3, 3, 4),
  TRT = factor(c("a", "b", "b", "b", "c", "c", "a", "c", "b", "b")),
  AESEV = ae_severities[c(4L, 1L, 2L, 1L, 2L, 1L, 2L, 3L, 1L, 2L)]
)
my_adsl <- data.frame(
  ID = rep(1:5, each = 3),
  TRT = factor(rep(c("a", "b", "c"), times = 5))
)

aesevall_spf <- make_combo_splitfun(
  nm = "AESEV_ALL",
  label = "Any AE",
  levels = NULL,
)

lyt <- basic_table() |>
  split_cols_by("TRT") |>
  add_overall_col("Total") |>
  split_rows_by("AESEV", split_fun = aesevall_spf) |>
  summarize_row_groups(
    "AESEV",
    cfun = a_maxlev,
    extra_args = list(id = "ID", any_level = TRUE)
  ) |>
  analyze(
    "AESEV",
    afun = a_maxlev,
    extra_args = list(id = "ID")
  )
build_table(lyt, my_adae, alt_counts_df = my_adsl)
```

a_proportion_ci_factor
Formatted Analysis Function For Proportion Confidence Interval for Factor

Description

Formatted Analysis Function For Proportion Confidence Interval for Factor.

Usage

```
a_proportion_ci_factor(df, .var, ...)
```

Arguments

`df`	(`data.frame`) including factor `.var`.
`.var`	(`string`) name of the factor variable.
`...`	see `a_proportion_ci_logical()` for additionally required arguments.

Value

The `rtables::rcell()` result.

Examples

```
a_proportion_ci_factor(
  df = DM,
  .var = "SEX",
  .alt_df = DM,
  conf_level = 0.95,
  formats = list(prop_ci = jjcsformat_xx("xx.x%, xx.x%")),
  method = "clopper-pearson"
)
```

a_proportion_ci_logical
Formatted Analysis Function For Proportion Confidence Interval for Logical

Description

Formatted Analysis Function For Proportion Confidence Interval for Logical.

Usage

```
a_proportion_ci_logical(x, .alt_df, conf_level, method, formats)
```

Arguments

`x`	(`logical`) including binary response values.
`.alt_df`	(`data.frame`) alternative data frame used for denominator calculation.
`conf_level`	(`numeric`) confidence level for the confidence interval.
`method`	(`string`) please see `tern::s_proportion()` for possible methods.
`formats`	(`list`) including element `prop_ci` with the required format. Note that the value is in percent already.

Value

The `rtables::rcell()` result.

Examples

```
a_proportion_ci_logical(
  x = DM$SEX == "F",
  .alt_df = DM,
  conf_level = 0.95,
  formats = list(prop_ci = jjcsformat_xx("xx.xx% - xx.xx%")),
  method = "wald"
)
```

`a_relative_risk` *Relative risk estimation*

Description

The analysis function `a_relative_risk()` is used to create a layout element to estimate the relative risk for response within a studied population. Only the CMH method is available currently. The primary analysis variable, `vars`, is a logical variable indicating whether a response has occurred for each record. A stratification variable must be supplied via the `strata` element of the `variables` argument.

Usage

```
a_relative_risk(
  df,
  .var,
  ref_path,
  .spl_context,
  ...,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL
)

s_relative_risk(
  df,
  .var,
  .ref_group,
  .in_ref_col,
  variables = list(strata = NULL),
  conf_level = 0.95,
  method = "cmh",
  weights_method = "cmh"
)
```

Arguments

`df`	(`data.frame`) input data frame.
`.var`	(`string`) name of the response variable.
`ref_path`	(`character`) path to the reference group.
`.spl_context`	(`environment`) split context environment.
`...`	Additional arguments passed to the statistics function.
`.stats`	(`character`) statistics to calculate.
`.formats`	(`list`) formats for the statistics.
`.labels`	(`list`) labels for the statistics.
`.indent_mods`	(`list`) indentation modifications for the statistics.
`.ref_group`	(`data.frame`) reference group data frame.

`.in_ref_col`	(`logical`) whether the current column is the reference column.
`variables`	(`list`) list with strata variable names.
`conf_level`	(`numeric`) confidence level for the confidence interval.
`method`	(`string`) method to use for relative risk calculation.
`weights_method`	(`string`) method to use for weights calculation in stratified analysis.

Details

The variance of the CMH relative risk estimate is calculated using the Greenland and Robins (1985) variance estimation.

Value

- `a_relative_risk()` returns the corresponding list with formatted `rtables::CellValue()`.
- `s_relative_risk()` returns a named list of elements `rel_risk_ci` and `pval`.

Functions

- `a_relative_risk()`: Formatted analysis function which is used as `afun`. Note that the junco specific `ref_path` and `.spl_context` arguments are used for reference column information.
- `s_relative_risk()`: Statistics function estimating the relative risk for response.

Note

This has been adapted from the `odds_ratio` functions in the `tern` package.

Examples

```
nex <- 100
dta <- data.frame(
  "rsp" = sample(c(TRUE, FALSE), nex, TRUE),
  "grp" = sample(c("A", "B"), nex, TRUE),
  "f1" = sample(c("a1", "a2"), nex, TRUE),
  "f2" = sample(c("x", "y", "z"), nex, TRUE),
  stringsAsFactors = TRUE
)

l <- basic_table() |>
  split_cols_by(var = "grp") |>
  analyze(
    vars = "rsp",
    afun = a_relative_risk,
    extra_args = list(
      conf_level = 0.90,
      variables = list(strata = "f1"),
      ref_path = c("grp", "B")
    )
  )

build_table(l, df = dta)
nex <- 100
dta <- data.frame(
  "rsp" = sample(c(TRUE, FALSE), nex, TRUE),
  "grp" = sample(c("A", "B"), nex, TRUE),
  "f1" = sample(c("a1", "a2"), nex, TRUE),
  "f2" = sample(c("x", "y", "z"), nex, TRUE),
  stringsAsFactors = TRUE
)

s_relative_risk(
  df = subset(dta, grp == "A"),
  .var = "rsp",
  .ref_group = subset(dta, grp == "B"),
  .in_ref_col = FALSE,
  variables = list(strata = c("f1", "f2")),
  conf_level = 0.90
)
```

a_summarize_ancova_j *ANCOVA Summary Function*

Description

Combination of tern::s_summary, and ANCOVA based estimates for mean and diff between columns, based on ANCOVA function s_ancova_j.

Usage

```
a_summarize_ancova_j(
  df,
  .var,
  .df_row,
  ref_path,
  .spl_context,
  ...,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL
)

s_summarize_ancova_j(df, .var, .df_row, .ref_group, .in_ref_col, ...)
```

Arguments

df	(data.frame) data set containing all analysis variables.
.var	(string) single variable name that is passed by rtables when requested by a statistics function.
.df_row	(data.frame) data set that includes all the variables that are called in .var and variables.
ref_path	(character) path to the reference group.
.spl_context	(environment) split context environment.
...	Additional arguments passed to s_ancova_j.
.stats	(character) statistics to calculate.
.formats	(list) formats for the statistics.
.labels	(list) labels for the statistics.
.indent_mods	(list) indentation modifications for the statistics.
.ref_group	(data.frame or vector) the data corresponding to the reference group.
.in_ref_col	(flag) TRUE when working with the reference level, FALSE otherwise.

Details

Combination of tern::s_summary, and ANCOVA based estimates for mean and diff between columns, based on ANCOVA function s_ancova_j

Value

- a_summarize_ancova_j() returns the corresponding list with formatted rtables::CellValue().

returns the statistics from tern::s_summary(x), appended with a new statistics based upon ANCOVA

Functions

- a_summarize_ancova_j(): Formatted analysis function which is used as afun. Note that the junco specific ref_path and .spl_context arguments are used for reference column information.

See Also

Other Inclusion of ANCOVA Functions: a_summarize_aval_chg_diff_j(), s_ancova_j()

Examples

```
basic_table() |>
  split_cols_by("Species") |>
  add_colcounts() |>
  analyze(
    vars = "Petal.Length",
    afun = a_summarize_ancova_j,
    show_labels = "hidden",
    na_str = tern::default_na_str(),
    table_names = "unadj",
    var_labels = "Unadjusted comparison",
    extra_args = list(
      variables = list(arm = "Species", covariates = NULL),
      conf_level = 0.95,
      .labels = c(lsmean = "Mean", lsmean_diff = "Difference in Means"),
      ref_path = c("Species", "setosa")
    )
  ) |>
  analyze(
    vars = "Petal.Length",
    afun = a_summarize_ancova_j,
    show_labels = "hidden",
    na_str = tern::default_na_str(),
    table_names = "adj",
    var_labels = "Adjusted comparison (covariates: Sepal.Length and Sepal.Width)",
    extra_args = list(
      variables = list(
        arm = "Species",
        covariates = c("Sepal.Length", "Sepal.Width")
      ),
      conf_level = 0.95,
      ref_path = c("Species", "setosa")
    )
  ) |>
  build_table(iris)

library(dplyr)
library(tern)

df <- iris |> filter(Species == "virginica")
.df_row <- iris
.var <- "Petal.Length"
variables <- list(arm = "Species", covariates = "Sepal.Length * Sepal.Width")
.ref_group <- iris |> filter(Species == "setosa")
conf_level <- 0.95
s_summarize_ancova_j(
  df,
  .var = .var,
  .df_row = .df_row,
  variables = variables,
  .ref_group = .ref_group,
  .in_ref_col = FALSE,
  conf_level = conf_level
)
```

a_summarize_aval_chg_diff_j

Analysis function 3-column presentation

Description

Analysis functions to produce a 1-row summary presented in a 3-column layout in the columns (column 1 = N, column 2 = Value, column 3 = Change).
In the difference columns, only 1 column will be presented : difference + CI
When ancova = `TRUE`, the presented statistics will be based on ANCOVA method (`s_summarize_ancova_j`).
mean and ci (both for Value (column 2) and CHG (column 3)) using statistic `lsmean_ci`
mean and ci for the difference column are based on same ANCOVA model using statistic `lsmean_diffci`
When ancova = `FALSE`, descriptive statistics will be used instead.
In the difference column, the 2-sample t-test will be used.

Usage

```
a_summarize_aval_chg_diff_j(
  df,
  .df_row,
  .spl_context,
  ancova = FALSE,
  comp_btw_group = TRUE,
  ref_path = NULL,
  .N_col,
  denom = c("N", ".N_col"),
  indatavar = NULL,
  d = 0,
  id = "USUBJID",
  interaction_y = FALSE,
  interaction_item = NULL,
  conf_level = 0.95,
  variables = list(arm = "TRT01A", covariates = NULL),
  format_na_str = "",
  .stats = list(col1 = "count_denom_frac", col23 = "mean_ci_3d", coldiff =
    "meandiff_ci_3d"),
  .formats = list(col1 = NULL, col23 = "xx.dx (xx.dx, xx.dx)", coldiff =
    "xx.dx (xx.dx, xx.dx)"),
  .formats_fun = list(col1 = jjcsformat_count_denom_fraction, col23 = jjcsformat_xx,
    coldiff = jjcsformat_xx),
  multivars = c("AVAL", "AVAL", "CHG")
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.df_row`	(`data.frame`) data frame across all of the columns for the given row split.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`ancova`	(`logical`) If FALSE, only descriptive methods will be used. If TRUE, ANCOVA methods will be used for each of the columns : AVAL, CHG, DIFF.
`comp_btw_group`	(`logical`) If TRUE, comparison between groups will be performed. When ancova = FALSE, the estimate of between group difference (on CHG) will be based upon a two-sample t-test. When ancova = TRUE, the same ANCOVA model will be used for the estimate of between group difference (on CHG).
`ref_path`	(`character`) global reference group specification, see `get_ref_info()`.
`.N_col`	(`integer`) column-wise N (column count) for the full column being analyzed that is typically passed by `rtables`.
`denom`	(`string`) choice of denominator for proportions. Options are: • `N`: number of records in this column/row split. There is no check in place that the current split only has one record per subject. Users should be careful with this. • `.N_col`: number of records in this column intersection (based on alt_counts_df dataset) (when alt_counts_df is a single record per subjects, this will match number of subjects)
`indatavar`	(`string`) If not null, variable name to extra subset incoming df to non-missing values of this variable.
`d`	(default = 1) choice of Decimal precision. Note that one extra precision will be added, as means are presented. Options are: • numerical(1) • variable name containing information on the precision, this variable should be available on input dataset. The content of this variable should then be an integer.
`id`	(`string`) subject variable name.

`interaction_y`	(`character`) Will be passed onto the `tern` function `s_ancova`, when ancova = TRUE.
`interaction_item`	(`character`) Will be passed onto the `tern` function `s_ancova`, when ancova = TRUE.
`conf_level`	(`proportion`) Confidence level of the interval
`variables`	(named list of strings) list of additional analysis variables, with expected elements: • arm (string) group variable, for which the covariate adjusted means of multiple groups will be summarized. Specifically, the first level of arm variable is taken as the reference group. • covariates (character) a vector that can contain single variable names (such as 'X1'), and/or interaction terms indicated by 'X1 * X2'.
`format_na_str`	(`string`)
`.stats`	(named `list`) column statistics to select for the table. The following column names are to be used: `col1`, `col23`, `coldiff`. For `col1`, the following stats can be specified. For `col23`, only `mean_ci_3d` is available. When ancova = `TRUE` these are LS Means, otherwise, arithmetic means. For `coldiff`, only `meandiff_ci_3d` is available. When ancova = `TRUE` these are LS difference in means, otherwise, difference in means based upon 2-sample t-test.
`.formats`	(named `list`) formats for the column statistics. `xx.d` style formats can be used.
`.formats_fun`	(named `list`) formatting functions for the column statistics, to be applied after the conversion of `xx.d` style to the appropriate precision.
`multivars`	(`string(3)`) Variables names to use in 3-col layout.

Details

See Description

Value

A function that can be used in an analyze function call

See Also

s_summarize_ancova_j

Other Inclusion of ANCOVA Functions: `a_summarize_ancova_j()`, `s_ancova_j()`

Examples

```
library(dplyr)

ADEG <- data.frame(
  STUDYID = c(
    "DUMMY", "DUMMY", "DUMMY", "DUMMY", "DUMMY",
    "DUMMY", "DUMMY", "DUMMY", "DUMMY", "DUMMY"
  ),
  USUBJID = c(
    "XXXXX01", "XXXXX02", "XXXXX03", "XXXXX04", "XXXXX05",
    "XXXXX06", "XXXXX07", "XXXXX08", "XXXXX09", "XXXXX10"
  ),
  TRT01A = c(
    "ARMA", "ARMA", "ARMA", "ARMA", "ARMA", "Placebo",
    "Placebo", "Placebo", "ARMA", "ARMA"
  ),
  PARAM = c("BP", "BP", "BP", "BP", "BP", "BP", "BP", "BP", "BP", "BP"),
  AVISIT = c(
    "Visit 1", "Visit 1", "Visit 1", "Visit 1", "Visit 1",
    "Visit 1", "Visit 1", "Visit 1", "Visit 1", "Visit 1"
  ),
  AVAL = c(56, 78, 67, 87, 88, 93, 39, 87, 65, 55),
  CHG = c(2, 3, -1, 9, -2, 0, 6, -2, 5, 2)
)

ADEG <- ADEG |>
  mutate(
    TRT01A = as.factor(TRT01A),
    STUDYID = as.factor(STUDYID)
  )

ADEG$colspan_trt <- factor(ifelse(ADEG$TRT01A == "Placebo", " ", "Active Study Agent"),
  levels = c("Active Study Agent", " ")
)
ADEG$rrisk_header <- "Risk Difference (%) (95% CI)"
ADEG$rrisk_label <- paste(ADEG$TRT01A, paste("vs", "Placebo"))

colspan_trt_map <- create_colspan_map(ADEG,
  non_active_grp = "Placebo",
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Study Agent",
  colspan_var = "colspan_trt",
  trt_var = "TRT01A"
)
ref_path <- c("colspan_trt", " ", "TRT01A", "Placebo")

lyt <- basic_table() |>
```

```
  split_cols_by(
    "colspan_trt",
    split_fun = trim_levels_to_map(map = colspan_trt_map)
  ) |>
  split_cols_by("TRT01A") |>
  split_rows_by(
    "PARAM",
    label_pos = "topleft",
    split_label = "Blood Pressure",
    section_div = " ",
    split_fun = drop_split_levels
  ) |>
  split_rows_by(
    "AVISIT",
    label_pos = "topleft",
    split_label = "Study Visit",
    split_fun = drop_split_levels,
    child_labels = "hidden"
  ) |>
  split_cols_by_multivar(
    c("AVAL", "AVAL", "CHG"),
    varlabels = c("n/N (%)", "Mean (CI)", "CFB (CI)")
  ) |>
  split_cols_by("rrisk_header", nested = FALSE) |>
  split_cols_by(
    "TRT01A",
    split_fun = remove_split_levels("Placebo"),
    labels_var = "rrisk_label"
  ) |>
  split_cols_by_multivar(c("CHG"), varlabels = c(" ")) |>
  analyze("STUDYID",
    afun = a_summarize_aval_chg_diff_j,
    extra_args = list(
      format_na_str = "-", d = 0,
      ref_path = ref_path, variables = list(arm = "TRT01A", covariates = NULL)
    )
  )

result <- build_table(lyt, ADEG)

result
```

a_summarize_ex_j   *Tabulation for Exposure Tables*

Description

A function to create the appropriate statistics needed for exposure table

Usage

```
s_summarize_ex_j(
  df,
  .var,
  .df_row,
  .spl_context,
  comp_btw_group = TRUE,
  ref_path = NULL,
  ancova = FALSE,
  interaction_y,
  interaction_item,
  conf_level,
  daysconv,
  variables
)

a_summarize_ex_j(
  df,
  .var,
  .df_row,
  .spl_context,
  comp_btw_group = TRUE,
  ref_path = NULL,
  ancova = FALSE,
  interaction_y = FALSE,
  interaction_item = NULL,
  conf_level = 0.95,
  variables,
  .stats = c("mean_sd", "median", "range", "quantiles", "total_subject_years"),
  .formats = c(diff_mean_est_ci = jjcsformat_xx("xx.xx (xx.xx, xx.xx)")),
  .labels = c(quantiles = "Interquartile range"),
  .indent_mods = NULL,
  na_str = rep("NA", 3),
  daysconv = 1
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`.df_row`	(`data.frame`) data frame across all of the columns for the given row split.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.

`comp_btw_group`	(`logical`) If TRUE, comparison between groups will be performed. When ancova = FALSE, the estimate of between group difference (on CHG) will be based upon two-sample t-test. When ancova = TRUE, the same ANCOVA model will be used for the estimate of between group difference (on CHG).
`ref_path`	(`character`) global reference group specification, see `get_ref_info()`.
`ancova`	(`logical`) If FALSE, only descriptive methods will be used. If TRUE, ANCOVA methods will be used for each of the columns : AVAL, CHG, DIFF.
`interaction_y`	(`character`) Will be passed onto the `tern` function `s_ancova`, when ancova = TRUE.
`interaction_item`	(`character`) Will be passed onto the `tern` function `s_ancova`, when ancova = TRUE.
`conf_level`	(`proportion`) Confidence level of the interval
`daysconv`	(`numeric`) conversion required to get the values into days (i.e 1 if original PARAMCD unit is days, 30.4375 if original PARAMCD unit is in months)
`variables`	(named list of strings) list of additional analysis variables, with expected elements: • arm (string) group variable, for which the covariate adjusted means of multiple groups will be summarized. Specifically, the first level of arm variable is taken as the reference group. • covariates (character) a vector that can contain single variable names (such as 'X1'), and/or interaction terms indicated by 'X1 * X2'.
`.stats`	(`character`) statistics to select for the table.
`.formats`	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.
`.labels`	(named `character`) labels for the statistics (without indent).
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.
`na_str`	(`string`) string used to replace all `NA` or empty values in the output.

Details

Creates statistics needed for standard exposure table. This includes differences and 95% CI and total treatment years. This is designed to be used as an analysis (afun in `analyze`) function.

Value

- `a_summarize_ex_j()` returns the corresponding list with formatted `rtables::CellValue()`.

Functions

- `s_summarize_ex_j()`: Statistics function needed for the exposure tables.
- `a_summarize_ex_j()`: Formatted analysis function which is used as `afun`.

Examples

```
library(dplyr)
ADEX <- ex_adsl |> select(USUBJID, ARM, TRTSDTM, EOSSTT, EOSDY)

trtvar <- "ARM"
ctrl_grp <- "B: Placebo"
cutoffd <- as.Date("2023-09-24")

ADEX <- ADEX |>
  create_colspan_var(
    non_active_grp          = ctrl_grp,
    non_active_grp_span_lbl = " ",
    active_grp_span_lbl     = "Active Study Agent",
    colspan_var             = "colspan_trt",
    trt_var                 = trtvar
  ) |>
  mutate(
    diff_header = "Difference in Means (95% CI)",
    diff_label = paste(!!rlang::sym(trtvar), "vs", ctrl_grp),
    TRTDURY = case_when(
      !is.na(EOSDY) ~ EOSDY,
      TRUE ~ as.integer(cutoffd - as.Date(TRTSDTM) + 1)
    )
  )

colspan_trt_map <- create_colspan_map(ADEX,
  non_active_grp = ctrl_grp,
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Study Agent",
  colspan_var = "colspan_trt",
  trt_var = trtvar
)

ref_path <- c("colspan_trt", "", trtvar, ctrl_grp)

lyt <- basic_table() |>
  split_cols_by(
    "colspan_trt",
    split_fun = trim_levels_to_map(map = colspan_trt_map)
  ) |>
  split_cols_by(trtvar) |>
  split_cols_by("diff_header", nested = FALSE) |>
  split_cols_by(
    trtvar,
    split_fun = remove_split_levels(ctrl_grp),
    labels_var = "diff_label"
  ) |>
  analyze("EOSDY",
    afun = a_summarize_ex_j, var_labels = "Duration of treatment (Days)",
    show_labels = "visible",
    indent_mod = 0L,
    extra_args = list(
      daysconv = 1,
      ref_path = ref_path,
      variables = list(arm = trtvar, covariates = NULL),
      ancova = TRUE,
      comp_btw_group = TRUE
    )
  )

result <- build_table(lyt, ADEX, alt_counts_df = ADEX)
result
```

a_two_tier *Two Tier Analysis Function*

Description

The analysis function used as an `afun` in analyze. This function simulates a final additional level of nesting with a traditional analyze call inside it.

This makes it possible to create what *appear to be* group summary or content rows and to *optionally or conditionally* generate one or more "detail" rows underneath it.

For example, in a disposition table, one might want counts for completed and ongoing patients with no further detail underneath, but a breakdown of specific reasons beneath the count of patients who discontinued treatment.

Usage

```
a_two_tier(
  df,
  labelstr = NULL,
  .var,
  .N_col,
  .df_row,
  inner_var,
  drill_down_levs,
  .spl_context,
  use_all_levels = FALSE,
  grp_fun,
  detail_fun,
  .alt_df_full = NULL,
  ...
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`labelstr`	(`character`) label of the level of the parent split currently being summarized (must be present as second argument in Content Row Functions). See `rtables::summarize_row_groups()` for more information.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`.N_col`	(`integer`) column-wise N (column count) for the full column being analyzed that is typically passed by `rtables`.
`.df_row`	(`data.frame`) data frame across all of the columns for the given row split.
`inner_var`	(`string`) single variable name to use when generating the detail rows.
`drill_down_levs`	(`character`) the level(s) of `.var` under which detail rows should be generated.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`use_all_levels`	(`flag`) controls which factor levels will be present for `inner_var` (both in `df`/`x` and in `.df_row`) when calling `detail_fun`. If `TRUE`, all levels (those present on the factor `.df_row[[inner_var]]`, *regardless if the level is observed in the row group or not) will be present when creating detail rows. Otherwise (the default), only the levels observed *anywhere in the row group, i.e., within* `.df_row` will be present.
`grp_fun`	(`function`) analysis function to be used when generating the "group summary" outer rows.
`detail_fun`	(`function`) analysis function to be used when generating "detail" inner rows.
`.alt_df_full`	(`dataframe`) denominator dataset for fraction and relative risk calculations. this argument gets populated by the rtables split machinery (see rtables::additional_fun_params).
`...`	additional arguments passed directly to `grp_fun` and `detail_fun`.

Details

Both the analysis variable and `inner_var` must be factors. Detail rows are differentiated by having an indent mod of one, causing them to hang indented under their corresponding group row.

Value

A `RowsVerticalSection` object including both the group row and all detail rows, if applicable, for the facet.

Note

In its current form, this function may not retain any formatting or labeling instructions added by `grp_fun` or `detail_fun`, and it will override any `.indent_mods` values specified by them. This behavior may change in future versions.

Author(s)

GB, WW.

Examples

```
# Example 1

lyt_obs_levels <- basic_table() |>
  split_cols_by("ARM") |>
  split_rows_by("EOSSTT", child_labels = "hidden") |>
  analyze("EOSSTT",
    afun = a_two_tier,
    extra_args = list(
      grp_fun = simple_analysis,
      detail_fun = simple_analysis,
      inner_var = "DCSREAS",
      drill_down_levs = "DISCONTINUED"
    )
  )

tbl <- build_table(lyt_obs_levels, ex_adsl)
tbl

lyt_all_levels <- basic_table() |>
  split_cols_by("ARM") |>
  split_rows_by("EOSSTT", child_labels = "hidden") |>
  analyze("EOSSTT",
    afun = a_two_tier,
    extra_args = list(
      grp_fun = simple_analysis,
      detail_fun = simple_analysis,
      inner_var = "DCSREAS",
      drill_down_levs = "DISCONTINUED",
      use_all_levels = TRUE
    )
  )

adsl_subset <- subset(ex_adsl, DCSREAS != "ADVERSE EVENT")
levels(adsl_subset$DCSREAS)

tbl_all_levels <- build_table(lyt_all_levels, adsl_subset)
tbl_all_levels

tbl_obs_levels <- build_table(lyt_obs_levels, adsl_subset)
tbl_obs_levels

# Example 2

library(dplyr)

trtvar <- "ARM"
ctrl_grp <- "B: Placebo"

adsl <- ex_adsl |> select(c("USUBJID", "STRATA1", "EOSSTT", "DCSREAS", all_of(trtvar)))
adsl$colspan_trt <- factor(
  ifelse(adsl[[trtvar]] == ctrl_grp, " ", "Active Study Agent"),
  levels = c("Active Study Agent", " ")
)
adsl$rrisk_header <- "Risk Difference (%) (95% CI)"
adsl$rrisk_label <- paste(adsl[[trtvar]], paste("vs", ctrl_grp))

colspan_trt_map <- create_colspan_map(
  df = adsl,
  non_active_grp = ctrl_grp,
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Study Agent",
  colspan_var = "colspan_trt",
  trt_var = trtvar
)

a_freq_j_args <- list(
  .stats = "count_unique_fraction",
  denom = "n_altdf",
  ref_path = c("colspan_trt", " ", trtvar, ctrl_grp)
)

two_tier_args <- list(
  grp_fun = a_freq_j,
  detail_fun = a_freq_j,
  inner_var = "DCSREAS",
  drill_down_levs = "DISCONTINUED"
)

lyt_rrisk <- basic_table() |>
  split_cols_by("colspan_trt", split_fun = trim_levels_to_map(map = colspan_trt_map)) |>
  split_cols_by(trtvar) |>
  split_cols_by("rrisk_header", nested = FALSE) |>
 split_cols_by(trtvar, labels_var = "rrisk_label", split_fun = remove_split_levels(ctrl_grp)) |>
```

```
  split_rows_by("STRATA1") |>
  split_rows_by("EOSSTT", child_labels = "hidden") |>
  analyze("EOSSTT", afun = a_two_tier, extra_args = c(two_tier_args, a_freq_j_args))

adsl_subset <- subset(
  adsl,
  EOSSTT != "COMPLETED" & (is.na(DCSREAS) | DCSREAS != "PROTOCOL VIOLATION")
)

tbl_rrisk <- build_table(lyt_rrisk, adsl_subset, alt_counts_df = adsl_subset)
tbl_rrisk
```

---

bspt_pruner *Pruning Function for pruning based on a fraction and/or a difference from the control arm*

Description

This is a pruning constructor function which identifies records to be pruned based on the the fraction from the percentages. In addition to just looking at a fraction within an arm, this function also allows further flexibility to also prune based on a comparison versus the control arm.

Usage

```
bspt_pruner(
  fraction = 0.05,
  keeprowtext = "Analysis set: Safety",
  reg_expr = FALSE,
  control = NULL,
  diff_from_control = NULL,
  only_more_often = TRUE,
  cols = c("TRT01A")
)
```

Arguments

`fraction`	`(proportion)` Fraction threshold. Function will keep all records strictly greater than this threshold.
`keeprowtext`	`(character)` Row to be excluded from pruning.
`reg_expr`	`(logical)` Apply keeprowtext as a regular expression (grepl with fixed = TRUE)
`control`	`(character)` Control Group

`diff_from_control`	(`numeric`) Difference from control threshold.
`only_more_often`	(`logical`) TRUE: Only consider when column pct is more often than control. FALSE: Also select a row where column pct is less often than control and abs(diff) above threshold
`cols`	(`character`) Column path.

Value

Function that can be utilized as pruning function in prune_table.

Examples

```
ADSL <- data.frame(
  USUBJID = c(
    "XXXXX01", "XXXXX02", "XXXXX03", "XXXXX04", "XXXXX05",
    "XXXXX06", "XXXXX07", "XXXXX08", "XXXXX09", "XXXXX10"
  ),
  TRT01P = c(
    "ARMA", "ARMB", "ARMA", "ARMB", "ARMB",
    "Placebo", "Placebo", "Placebo", "ARMA", "ARMB"
  ),
  FASFL = c("Y", "Y", "Y", "Y", "N", "Y", "Y", "Y", "Y", "Y"),
  SAFFL = c("N", "N", "N", "N", "N", "N", "N", "N", "N", "N"),
  PKFL = c("N", "N", "N", "N", "N", "N", "N", "N", "N", "N")
)

ADSL <- ADSL |>
  dplyr::mutate(TRT01P = as.factor(TRT01P)) |>
  dplyr::mutate(SAFFL = factor(SAFFL, c("Y", "N"))) |>
  dplyr::mutate(PKFL = factor(PKFL, c("Y", "N")))

lyt <- basic_table() |>
  split_cols_by("TRT01P") |>
  add_overall_col("Total") |>
  split_rows_by(
    "FASFL",
    split_fun = drop_and_remove_levels("N"),
    child_labels = "hidden"
  ) |>
  analyze("FASFL",
    var_labels = "Analysis set:",
    afun = a_freq_j,
    show_labels = "visible",
    extra_args = list(label = "Full", .stats = "count_unique_fraction")
  ) |>
  split_rows_by(
    "SAFFL",
    split_fun = remove_split_levels("N"),
    child_labels = "hidden"
  ) |>
  analyze("SAFFL",
    var_labels = "Analysis set:",
    afun = a_freq_j,
    show_labels = "visible",
    extra_args = list(label = "Safety", .stats = "count_unique_fraction")
  ) |>
  split_rows_by(
    "PKFL",
    split_fun = remove_split_levels("N"),
    child_labels = "hidden"
  ) |>
  analyze("PKFL",
    var_labels = "Analysis set:",
    afun = a_freq_j,
    show_labels = "visible",
    extra_args = list(label = "PK", .stats = "count_unique_fraction")
  )

result <- build_table(lyt, ADSL)

result

result <- prune_table(
  result,
  prune_func = bspt_pruner(
    fraction = 0.05,
    keeprowtext = "Safety",
    cols = c("Total")
  )
)

result
```

build_formula *Building Model Formula*

Description

This builds the model formula which is used inside `fit_mmrm_j()` and provided to `mmrm::mmrm()` internally. It can be instructive to look at the resulting formula directly sometimes.

Usage

```
build_formula(
  vars,
  cor_struct = c("unstructured", "toeplitz", "heterogeneous toeplitz", "ante-dependence",
    "heterogeneous ante-dependence", "auto-regressive", "heterogeneous auto-regressive",
    "compound symmetry", "heterogeneous compound symmetry")
)
```

Arguments

vars	(list) variables to use in the model.
cor_struct	(string) specify the covariance structure to use.

Value

Formula to use in `mmrm::mmrm()`.

Examples

```
vars <- list(
  response = "AVAL", covariates = c("RACE", "SEX"),
  id = "USUBJID", arm = "ARMCD", visit = "AVISIT"
)
build_formula(vars, "auto-regressive")
build_formula(vars)
```

---

check_wrap_nobreak   *Check Word Wrapping*

Description

Check a set of column widths for word-breaking wrap behavior.

Usage

```
check_wrap_nobreak(tt, colwidths, fontspec)
```

Arguments

tt	(TableTree) TableTree object
colwidths	(numeric) Column widths (in numbers of spaces under `fontspec`)
fontspec	(font_spec) Font specification object

Value

`TRUE` if the wrap is able to be done without breaking words, `FALSE` if wordbreaking is required to apply `colwidths`.

`cmhrms` *Cochran-Mantel-Haenszel Row Mean Scores test*

Description

See https://psiaims.github.io/CAMIS/Comp/r-sas_cmh.html for a general comparison overview between R and SAS.

Usage

```
a_cmhrms_j(
  df,
  .var,
  ref_path,
  .spl_context,
  .ref_group,
  .in_ref_col,
  .df_row,
  ...,
  variables,
  collapse_combo = TRUE,
  .stats = NULL,
  .formats = NULL,
  .indent_mods = NULL,
  .labels = NULL
)

s_cmhrms_j(
  df,
  .var,
  .ref_group,
  .in_ref_col,
  ...,
  .df_row,
  variables,
  collapse_combo = FALSE
)

a_cmhrms_j_with_exclude(
  df,
  exclude_levels,
  .var,
  .spl_context,
  .ref_group,
  .in_ref_col,
  .df_row,
  ...,
  .stats = NULL,
  .formats = NULL,
  .indent_mods = NULL,
  .labels = NULL
)
```

Arguments

df	(`data.frame`) data set containing all analysis variables.
.var	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
ref_path	(`character`) global reference group specification, see `get_ref_info()`.
.spl_context	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
.ref_group	(`data.frame` or `vector`) the data corresponding to the reference group.
.in_ref_col	(`logical`) `TRUE` when working with the reference level, `FALSE` otherwise.
.df_row	(`data.frame`) data frame across all of the columns for the given row split.
...	additional arguments for the lower level functions.
variables	(`list`) list with arm and strata variable names.
collapse_combo	(`logical`) If TRUE, multiple arm levels from df will be combined into 1 level.
.stats	(`character`) statistics to select for the table.
.formats	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.
.indent_mods	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.
.labels	(named `character`) labels for the statistics (without indent).
exclude_levels	(`list`) A named list where names correspond to split variables and values are vectors of levels to exclude.

Value

- `s_cmhrms_j` a single element list containing the p-value from row mean score test.
- `a_cmhrms_j` a `VerticalRowsSection` object (single row).

Functions

- `a_cmhrms_j()`: Formatted analysis function which is used as `afun`.
- `s_cmhrms_j()`: Statistics function for the calculation of the p-value based upon the row mean scores test.
- `a_cmhrms_j_with_exclude()`: Wrapper for the `afun` which can exclude row split levels from producing the analysis. These have to be specified in the `exclude_levels` argument, see `?do_exclude_split` for details.

`cmp_cfun` *Summary Analysis Function for Compliance Columns*

Description

A simple statistics function which prepares the numbers with percentages in the required format, for use in a split content row. The denominator here is from the expected visits column.

Usage

```
cmp_cfun(df, labelstr, .spl_context, variables, formats)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`labelstr`	(`character`) label of the level of the parent split currently being summarized (must be present as second argument in Content Row Functions). See `rtables::summarize_row_groups()` for more information.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`variables`	(`list`) with variable names of logical columns for `expected`, `received` and `missing` visits.
`formats`	(`list`) with the `count_percent` format to use for the received and missing visits columns.

Value

The `rtables::in_rows()` result with the counts and proportion statistics.

See Also

`cmp_post_fun()` for the corresponding split function.

`cmp_post_fun` *Split Function for Compliance Columns*

Description

Here we just split into 3 columns for expected, received and missing visits.

Usage

```
cmp_post_fun(ret, spl, fulldf, .spl_context)

cmp_split_fun(df, spl, vals = NULL, labels = NULL, trim = FALSE, .spl_context)
```

Arguments

`ret`	(`list`) result from previous split function steps.
`spl`	(`split`) split object.
`fulldf`	(`data.frame`) full data frame.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`df`	(`data.frame`) data set containing all analysis variables.
`vals`	(`character`) values to use for the split.
`labels`	(named `character`) labels for the statistics (without indent).
`trim`	(`logical`) whether to trim the values.

Value

a split function for use with rtables::split_rows_by when creating proportion-based tables with compliance columns.

See Also

`rtables::make_split_fun()` describing the requirements for this kind of post-processing function.

column_stats — *Statistics within the column space*

Description

A function factory used for obtaining statistics within the columns of your table. Used in change from baseline tables. This takes the visit names as its row labels.

Usage

```
column_stats(
  exclude_visits = c("Baseline (DB)"),
  var_names = c("AVAL", "CHG", "BASE"),
 stats = list(main = c(N = "N", mean = "Mean", SD = "SD", SE = "SE", Med = "Med", Min =
    "Min", Max = "Max"), base = c(mean = "Mean"))
)
```

Arguments

`exclude_visits`	`(character vector)` Vector of visit(s) for which you do not want the statistics displayed in the baseline mean or change from baseline sections of the table.
`var_names`	`(character vector)` Vector of variable names to use instead of the default AVAL, CHG, BASE. The first two elements are treated as main variables with full statistics, and the third element is treated as the base variable. By default, the function expects these specific variable names in your data, but you can customize them to match your dataset's column names.
`stats`	`(list)` A list with two components, `main` and `base`, that define the statistics to be calculated for the main variables (default: AVAL, CHG) and the base variable (default: BASE). Default for main variables: c(N = "N", mean = "Mean", SD = "SD", SE = "SE", Med = "Med", Min = "Min", Max = "Max"). Default for base variable: c(mean = "Mean"). You can customize these statistics by providing your own named vectors in the list. The names are used internally for calculations, and the values are used as display labels in the table.

Value

An analysis function (for use with rtables::analyze) implementing the specified statistics.

`cond_rm_facets` *Conditional Removal of Facets*

Description

Conditional Removal of Facets

Usage

```
cond_rm_facets(
  facets = NULL,
  facets_regex = NULL,
  ancestor_pos = 1,
  split = NULL,
  split_regex = NULL,
  value = NULL,
  value_regex = NULL,
  keep_matches = FALSE
)
```

Arguments

`facets`	(`character` or NULL) Vector of facet names to be removed if condition(s) are met
`facets_regex`	(`character`) Regular expression to identify facet names to be removed if condition(s) are met.
`ancestor_pos`	(`numeric`) Row in spl_context to check the condition within. E.g., 1 represents the first split, 2 represents the second split nested within the first, etc. NA specifies that the conditions should be checked at all split levels. Negative integers indicate position counting back from the current one, e.g., -1 indicates the direct parent (most recent split before this one). Negative and positive/NA positions cannot be mixed.
`split`	(`character` or NULL) If specified, name of the split at position `ancestor_pos` must be identical to this value for the removal condition to be met.
`split_regex`	(`character` or NULL) If specified, a regular expression the name of the split at position `ancestor_pos` must match for the removal condition to be met. Cannot be specified at the same time as `split`.
`value`	(`character` or NULL) If specified, split (facet) value at position `ancestor_pos` must be identical to this value for removal condition to be met.

`value_regex`	(`character` or NULL) If specified, a regular expression the value of the split at position `ancestor_pos` must match for the removal condition to be met. Cannot be specified at the same time as `value`.
`keep_matches`	(`logical`) Given the specified condition is met, should the facets removed be those matching `facets/facets_regex` (`FALSE`, the default), or those *not* matching (`TRUE`).

Details

Facet removal occurs when the specified condition(s) on the split(s) and or value(s) are met within at least one of the split_context rows indicated by `ancestor_pos`; otherwise the set of facets is returned unchanged.

If facet removal is performed, either *all* facets which match `facets` (or `facets_regex` will be removed (the default `keep_matches` == `FALSE` case), or all *non-matching* facets will be removed (when `keep_matches_only` == `TRUE`).

Value

A function suitable for use in `make_split_fun`'s `post` argument which encodes the specified condition.

Note

A degenerate table is likely to be returned if all facets are removed.

Examples

```
rm_a_from_placebo <- cond_rm_facets(
  facets = "A",
  ancestor_pos = NA,
  value_regex = "Placeb",
  split = "ARM"
)
mysplit <- make_split_fun(post = list(rm_a_from_placebo))

lyt <- basic_table() |>
  split_cols_by("ARM") |>
  split_cols_by("STRATA1", split_fun = mysplit) |>
  analyze("AGE", mean, format = "xx.x")
build_table(lyt, ex_adsl)

rm_bc_from_combo <- cond_rm_facets(
  facets = c("B", "C"),
  ancestor_pos = -1,
  value_regex = "Combi"
)
mysplit2 <- make_split_fun(post = list(rm_bc_from_combo))

lyt2 <- basic_table() |>
  split_cols_by("ARM") |>
  split_cols_by("STRATA1", split_fun = mysplit2) |>
  analyze("AGE", mean, format = "xx.x")
tbl2 <- build_table(lyt2, ex_adsl)
tbl2

rm_bc_from_combo2 <- cond_rm_facets(
  facets_regex = "^A$",
  ancestor_pos = -1,
  value_regex = "Combi",
  keep_matches = TRUE
)
mysplit3 <- make_split_fun(post = list(rm_bc_from_combo2))

lyt3 <- basic_table() |>
  split_cols_by("ARM") |>
  split_cols_by("STRATA1", split_fun = mysplit3) |>
  analyze("AGE", mean, format = "xx.x")
tbl3 <- build_table(lyt3, ex_adsl)

stopifnot(identical(cell_values(tbl2), cell_values(tbl3)))
```

count and fraction related formatting functions

Formatting functions for count and fraction, and for count denominator and fraction values

Description

Formats a count together with fraction (and/or denominator) with special consideration when count is 0, or fraction is 1.
See also: `tern::format_count_fraction_fixed_dp()`

Usage

```
jjcsformat_cnt_den_fract_fct(
  d = 1,
  type = c("count_fraction", "count_denom_fraction", "fraction_count_denom"),
  verbose = FALSE
)

jjcsformat_count_fraction(x, round_type = valid_round_type, output, ...)

jjcsformat_count_denom_fraction(x, round_type = valid_round_type, output, ...)

jjcsformat_fraction_count_denom(x, round_type = valid_round_type, output, ...)
```

Arguments

`d`	`(numeric(1))` Number of digits to round fraction to (default = 1)
`type`	`(character(1)` One of `count_fraction`, `count_denom_fraction`, `fraction_count_denom`, to specify the type of format the function will represent.
`verbose`	`(logical)` Whether to print verbose output
`x`	`(numeric vector)` Vector with elements `num` and `fraction` or `num`, `denom` and `fraction`.
`round_type`	`(character(1))` the type of rounding to perform. See `formatters::format_value()` for more details.
`output`	`(string)` output type. See `formatters::format_value()` for more details.
`...`	Additional arguments passed to other methods.

Value

A formatting function to format input into string in the format `count / denom (ratio percent)`. If `count` is 0, the format is `0`. If fraction is >0.99, the format is `count / denom (>99.9 percent)`

See Also

Other JJCS formatting functions: `jjcsformat_xx()`

Examples

```
jjcsformat_count_fraction(c(7, 0.7))
jjcsformat_count_fraction(c(70000, 70000 / 70001))
jjcsformat_count_fraction(c(235, 235 / 235))
fmt <- jjcsformat_cnt_den_fract_fct(type = "count_fraction", d = 2)
fmt(c(23, 23 / 235))

jjcsformat_count_denom_fraction(c(7, 10, 0.7))
jjcsformat_count_denom_fraction(c(70000, 70001, 70000 / 70001))
jjcsformat_count_denom_fraction(c(235, 235, 235 / 235))
fmt <- jjcsformat_cnt_den_fract_fct(type = "count_denom_fraction", d = 2)
fmt(c(23, 235, 23 / 235))

jjcsformat_fraction_count_denom(c(7, 10, 0.7))
jjcsformat_fraction_count_denom(c(70000, 70001, 70000 / 70001))
jjcsformat_fraction_count_denom(c(235, 235, 235 / 235))
fmt <- jjcsformat_cnt_den_fract_fct(type = "fraction_count_denom", d = 2)
fmt(c(23, 235, 23 / 235))
```

count_pruner *Count Pruner*

Description

This is a pruning constructor function which identifies records to be pruned based on the count (assumed to be the first statistic displayed when a compound statistic (e.g., ## / ## (XX.X percent) is presented).

Usage

```
count_pruner(
  count = 0,
  cat_include = NULL,
  cat_exclude = NULL,
  cols = c("TRT01A")
)
```

Arguments

`count`	`(numeric)` count threshold. Function will keep all records strictly greater than this threshold.
`cat_include`	`(character)` Category to be considered for pruning
`cat_exclude`	`(character)` Category to be excluded from pruning
`cols`	`(character)` column path (character or integer (column indices))

Value

Function that can be utilized as pruning function in prune_table.

Examples

```
ADSL <- data.frame(
  USUBJID = c(
    "XXXXX01", "XXXXX02", "XXXXX03", "XXXXX04", "XXXXX05",
    "XXXXX06", "XXXXX07", "XXXXX08", "XXXXX09", "XXXXX10"
  ),
  TRT01P = factor(
    c(
      "ARMA", "ARMB", "ARMA", "ARMB", "ARMB",
      "Placebo", "Placebo", "Placebo", "ARMA", "ARMB"
    )
  ),
  FASFL = c("Y", "Y", "Y", "Y", "N", "Y", "Y", "Y", "Y", "Y"),
  SAFFL = c("N", "N", "N", "N", "N", "N", "N", "N", "N", "N"),
  PKFL = c("N", "N", "N", "N", "N", "N", "N", "N", "N", "N")
)

lyt <- basic_table() |>
  split_cols_by("TRT01P") |>
  add_overall_col("Total") |>
  analyze("FASFL",
    var_labels = "Analysis set:",
    afun = a_freq_j,
    extra_args = list(label = "Full", val = "Y"),
    show_labels = "visible"
  ) |>
  analyze("SAFFL",
    var_labels = "Analysis set:",
    afun = a_freq_j,
    extra_args = list(label = "Safety", val = "Y"),
    show_labels = "visible"
  ) |>
  analyze("PKFL",
    var_labels = "Analysis set:",
    afun = a_freq_j,
    extra_args = list(label = "PK", val = "Y"),
    show_labels = "visible"
  )

result <- build_table(lyt, ADSL)

result

result <- prune_table(
  result,
  prune_func = count_pruner(cat_exclude = c("Safety"), cols = "Total")
)

result
```

coxph_hr *Workaround statistics function to add HR with CI*

Description

This is a workaround for `tern::s_coxph_pairwise()`, which adds a statistic containing the hazard ratio estimate together with the confidence interval.

Usage

```
a_coxph_hr(
  df,
    .var,
    ref_path,
    .spl_context,
    ...,
    .stats = NULL,
    .formats = NULL,
    .labels = NULL,
    .indent_mods = NULL
  )

  s_coxph_hr(
    df,
    .ref_group,
    .in_ref_col,
    .var,
    is_event,
    strata = NULL,
    control = control_coxph(),
    alternative = c("two.sided", "less", "greater")
  )
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`ref_path`	(`character`) global reference group specification, see `get_ref_info()`.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`...`	additional arguments for the lower level functions.
`.stats`	(`character`) statistics to select for the table.
`.formats`	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.
`.labels`	(named `character`) labels for the statistics (without indent).
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.
`.ref_group`	(`data.frame` or `vector`) the data corresponding to the reference group.

.in_ref_col	(logical) TRUE when working with the reference level, FALSE otherwise.
is_event	(character) variable name storing Logical values: TRUE if event, FALSE if time to event is censored.
strata	(character or NULL) variable names indicating stratification factors.
control	(list) relevant list of control options.
alternative	(string) whether two.sided, or one-sided less or greater p-value should be displayed.

Value

- s_coxph_hr returns a list containing the same statistics returned by tern::s_coxph_pairwise and the additional lr_stat_df statistic.
- a_coxph_hr returns a VerticalRowsSection object.

Functions

- a_coxph_hr(): Formatted analysis function which is used as afun.
- s_coxph_hr(): Statistics function forked from tern::s_coxph_pairwise(). The difference is that it returns the additional statistic lr_stat_df (log rank statistic with degrees of freedom).

Examples

```
library(dplyr)

adtte_f <- tern::tern_ex_adtte |>
  filter(PARAMCD == "OS") |>
  mutate(is_event = CNSR == 0)

df <- adtte_f |> filter(ARMCD == "ARM A")
df_ref_group <- adtte_f |> filter(ARMCD == "ARM B")

basic_table() |>
  split_cols_by(var = "ARMCD", ref_group = "ARM A") |>
  add_colcounts() |>
  analyze("AVAL",
    afun = s_coxph_hr,
    extra_args = list(is_event = "is_event"),
    var_labels = "Unstratified Analysis",
    show_labels = "visible"
  ) |>
  build_table(df = adtte_f)

basic_table() |>
  split_cols_by(var = "ARMCD", ref_group = "ARM A") |>
```

```
    add_colcounts() |>
    analyze("AVAL",
      afun = s_coxph_hr,
      extra_args = list(
        is_event = "is_event",
        strata = "SEX",
        control = tern::control_coxph(pval_method = "wald")
      ),
      var_labels = "Unstratified Analysis",
      show_labels = "visible"
    ) |>
    build_table(df = adtte_f)
  adtte_f <- tern::tern_ex_adtte |>
    dplyr::filter(PARAMCD == "OS") |>
    dplyr::mutate(is_event = CNSR == 0)
  df <- adtte_f |> dplyr::filter(ARMCD == "ARM A")
  df_ref <- adtte_f |> dplyr::filter(ARMCD == "ARM B")

  s_coxph_hr(
    df = df,
    .ref_group = df_ref,
    .in_ref_col = FALSE,
    .var = "AVAL",
    is_event = "is_event",
    strata = NULL
  )
```

create_colspan_map *Creation of Column Spanning Mapping Dataframe*

Description

A function used for creating a data frame containing the map that is compatible with rtables split function. `trim_levels_to_map`

Usage

```
create_colspan_map(
  df,
  non_active_grp = c("Placebo"),
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Study Agent",
  colspan_var = "colspan_trt",
  trt_var = "TRT01A",
  active_first = TRUE
)
```

Arguments

`df`	(`data.frame`) The name of the data frame in which the spanning variable is to be appended to
`non_active_grp`	(`character`) The value(s) of the treatments that represent the non-active or comparator treatment groups default value = c('Placebo')
`non_active_grp_span_lbl`	(`character`) The assigned value of the spanning variable for the non-active or comparator treatment groups default value = ''
`active_grp_span_lbl`	(`character`) The assigned value of the spanning variable for the active treatment group(s) default value = 'Active Study Agent'
`colspan_var`	(`character`) The desired name of the newly created spanning variable default value = 'colspan_trt'
`trt_var`	(`character`) The name of the treatment variable that is used to determine which spanning treatment group value to apply. default value = 'TRT01A'
`active_first`	(`logical`) whether the active columns come first.

Details

This function creates a data frame containing the map that is compatible with rtables split function `trim_levels_to_map`. The levels of the specified trt_var variable will be stored within the trt_var variable and the colspan_var variable will contain the corresponding spanning header value for each treatment group.

Value

A data frame that contains the map to be used with rtables split function `trim_levels_to_map`.

Examples

```
library(tibble)

df <- tribble(
  ~TRT01A,
  "Placebo",
  "Active 1",
  "Active 2"
)

df$TRT01A <- factor(df$TRT01A, levels = c("Placebo", "Active 1", "Active 2"))

colspan_map <- create_colspan_map(
  df = df,
  non_active_grp = c("Placebo"),
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Study Agent",
  colspan_var = "colspan_trt",
  trt_var = "TRT01A"
)

colspan_map
```

create_colspan_var *Creation of Column Spanning Variables*

Description

A function used for creating a spanning variable for treatment groups.

Usage

```
create_colspan_var(
  df,
  non_active_grp = c("Placebo"),
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Study Agent",
  colspan_var = "colspan_trt",
  trt_var = "TRT01A"
)
```

Arguments

`df`	(`data.frame`) The name of the data frame in which the spanning variable is to be appended to
`non_active_grp`	(`character`) The value(s) of the treatments that represent the non-active or comparator treatment groups default value = c('Placebo')
`non_active_grp_span_lbl`	(`character`) The assigned value of the spanning variable for the non-active or comparator treatment groups default value = ''
`active_grp_span_lbl`	(`character`) The assigned value of the spanning variable for the active treatment group(s) default value = 'Active Study Agent'
`colspan_var`	(`character`) The desired name of the newly created spanning variable default value = 'colspan_trt'
`trt_var`	(`character`) The name of the treatment variable that is used to determine which spanning treatment group value to apply. default value = 'TRT01A'

Details

This function creates a spanning variable for treatment groups that is intended to be used within the column space.

Value

A data frame that contains the new variable as specified in colspan_var.

Examples

```
library(tibble)

df <- tribble(
  ~TRT01A,
  "Placebo",
  "Active 1",
  "Active 2"
)

df$TRT01A <- factor(df$TRT01A, levels = c("Placebo", "Active 1", "Active 2"))

colspan_var <- create_colspan_var(
  df = df,
  non_active_grp = c("Placebo"),
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Treatment",
  colspan_var = "colspan_trt",
  trt_var = "TRT01A"
)

colspan_var
```

`c_proportion_logical` *c_function for proportion of* `TRUE` *in logical vector*

Description

A simple statistics function which prepares the numbers with percentages in the required format, for use in a split content row. The denominator here is from the column N. Note that we don't use here .alt_df because that might not have required row split variables available.

Usage

```
c_proportion_logical(x, labelstr, label_fstr, format, .N_col)
```

Arguments

x	(logical) binary variable we want to analyze.
labelstr	(string) label string.
label_fstr	(string) format string for the label.
format	(character or list) format for the statistics.
.N_col	(numeric) number of columns.

Value

The `rtables::in_rows()` result with the proportion statistics.

See Also

`s_proportion_logical()` for the related statistics function.

`do_exclude_split` *Predicate to Check if Split Should be Excluded*

Description

Predicate to Check if Split Should be Excluded

Usage

```
do_exclude_split(exclude_levels, .spl_context)
```

Arguments

exclude_levels	(list) A named list where names correspond to split variables and values are vectors of levels to exclude.
.spl_context	(data.frame) gives information about ancestor split states that is passed by `rtables`.

Value

`TRUE` if the current split context matches any of the exclude levels, `FALSE` otherwise.

Examples

```
do_exclude_split(
  exclude_levels = list(AVISIT = "Baseline"),
  .spl_context = data.frame(
    split = c("AVISIT", "ARM"),
    value = c("Week 4", "Placebo")
  )
)
do_exclude_split(
  exclude_levels = list(AVISIT = "Baseline"),
  .spl_context = data.frame(
    split = c("AVISIT", "ARM"),
    value = c("Baseline", "Placebo")
  )
)
```

event_free *Workaround statistics function to time point survival estimate with CI*

Description

This is a workaround for `tern::s_surv_timepoint()`, which adds a statistic containing the time point specific survival estimate together with the confidence interval.

Usage

```
a_event_free(
  df,
  .var,
  ...,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL
)

s_event_free(
  df,
  .var,
  time_point,
  time_unit,
  is_event,
  percent = FALSE,
  control = control_surv_timepoint()
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`...`	additional arguments for the lower level functions.
`.stats`	(`character`) statistics to select for the table.
`.formats`	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.
`.labels`	(named `character`) labels for the statistics (without indent).
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.
`time_point`	(`numeric`) time point at which to estimate survival.
`time_unit`	(`string`) unit of time for the time point.
`is_event`	(`character`) variable name storing Logical values: `TRUE` if event, `FALSE` if time to event is censored.
`percent`	(`flag`) whether to return in percent or not.
`control`	(`list`) relevant list of control options.

Value

- `s_event_free` returns a list as returned by the `tern::s_surv_timepoint()` with an additional three-dimensional statistic `event_free_ci` which combines the `event_free_rate` and `rate_ci` statistics.
- `a_event_free` is analogous to tern::a_surv_timepoint but with the additional three-dimensional statistic described above available via `.stats`.

Functions

- `a_event_free()`: Formatted analysis function which is used as `afun`.
- `s_event_free()`: Statistics function which works like `tern::s_surv_timepoint()`, the difference is that it returns the additional statistic `event_free_ci`.

Examples

```
adtte_f <- tern::tern_ex_adtte |>
  dplyr::filter(PARAMCD == "OS") |>
  dplyr::mutate(
    AVAL = tern::day2month(AVAL),
    is_event = CNSR == 0
  )

basic_table() |>
  split_cols_by(var = "ARMCD") |>
  analyze(
    vars = "AVAL",
    afun = a_event_free,
    show_labels = "hidden",
    na_str = tern::default_na_str(),
    extra_args = list(
      time_unit = "week",
      time_point = 3,
      is_event = "is_event"
    )
  ) |>
  build_table(df = adtte_f)
adtte_f <- tern::tern_ex_adtte |>
  dplyr::filter(PARAMCD == "OS") |>
  dplyr::mutate(
    AVAL = tern::day2month(AVAL),
    is_event = CNSR == 0
  )

s_event_free(
  df = adtte_f,
  .var = "AVAL",
  time_point = 6,
  is_event = "is_event",
  time_unit = "month"
)
```

`export_as_docx_j` *Export a VTableTree or a listing_df object into docx*

Description

[Experimental]

This function is based on `rtables.officer::export_as_docx()`. See notes to understand why this is experimental.

Usage

```
export_as_docx_j(
    tt,
    tblid,
    output_dir,
    theme = theme_docx_default_j(font = "Times New Roman", font_size = 9L, bold = NULL),
    add_page_break = FALSE,
    titles_as_header = TRUE,
    integrate_footers = TRUE,
    section_properties = officer::prop_section(page_size = officer::page_size(width = 11,
      height = 8.5, orient = orientation), page_margins = officer::page_mar(bottom = 1, top
      = 1, right = 1, left = 1, gutter = 0, footer = 1, header = 1)),
    doc_metadata = NULL,
    template_file = system.file("template_file.docx", package = "junco"),
    orientation = "portrait",
    paginate = FALSE,
    nosplitin = character(),
    string_map = junco::default_str_map,
    markup_df_docx = dps_markup_df_docx,
    combined_docx = FALSE,
    tlgtype = (utils::getFromNamespace("tlg_type", "junco"))(tt),
    col_gap = ifelse(tlgtype == "Listing", 0.5, 3),
    pagenum = ifelse(tlgtype == "Listing", TRUE, FALSE),
    round_type = formatters::obj_round_type(tt),
    alignments = list(),
    border = flextable::fp_border_default(width = 0.75, color = "black"),
    border_mat = make_header_bordmat(obj = tt),
    watermark = FALSE,
    ...
)
```

Arguments

`tt`	a VTableTree or a listing_df object to export.
`tblid`	Character. Output ID to be displayed in the Title and last line of footer.
`output_dir`	a directory path to save the docx.
`theme`	(optional) a function factory. See theme_docx_default_j() or rtables.officer::theme_docx_default() for more details.
`add_page_break`	(optional) Default = FALSE.
`titles_as_header`	(optional) Default = TRUE.
`integrate_footers`	(optional) Default = TRUE.
`section_properties`	(optional). A "prop_section" object containing information about page size, orientation, margins, etc. See officer::prop_section() for more details. No need to be specified by end user.
`doc_metadata`	(optional). Default = NULL.

`template_file`	(optional). Default = "doc/template_file.docx". Paragraph styles are inherited from this file.
`orientation`	(optional) Default = "portrait". One of: "portrait", "landscape".
`paginate`	(optional) Default = FALSE.
`nosplitin`	(optional) Default = character(). Named list.
`string_map`	(optional) Default = default_str_map.
`markup_df_docx`	(optional) Default = dps_markup_df_docx.
`combined_docx`	(optional). Default = FALSE. Whether to also export an "allparts" docx version.
`tlgtype`	(optional). No need to be specified by end user.
`col_gap`	(optional). Default = 3 (Tables) or 0.5 (Listings).
`pagenum`	(optional). Whether to display page numbers. Only applicable to listings (i.e. for tables and figures this argument is ignored).
`round_type`	(`"iec"` or `"sas"`) the type of rounding to perform. iec, the default, performs rounding compliant with IEC 60559, while sas performs nearest-value rounding consistent with rounding within SAS. See `[formatters::format_value()]` for more details.
`alignments`	(`list`) List of named lists. Vectorized. (Default = `list()`) Used to specify individual column or cell alignments. Each named list contains `row`, `col`, and `value`.
`border`	(optional) an `fp_border` object.
`border_mat`	(`matrix`) A `m x k` matrix where m is the number of columns of `tt` and k is the number of lines the header takes up. See tidytlg::add_bottom_borders for what the matrix should contain. Users should only specify this when the default behavior does not meet their needs.
`watermark`	(`logical`) whether to display the watermark "Confidential". By default, this is set to FALSE. In the future, this argument will be the actual watermark (i.e. a string) to display.
`...`	other parameters.

Note

This function has been tested for common use cases but may not work or have unexpected or undesired behavior in corner cases. As such it is not considered fully production ready and is being made available for further testing and early adoption. Please report any issues you encounter to the developers. This function may be removed from junco in the future if the functionality is merged into `rtables.officer`.

export_graph_as_docx *export_graph_as_docx*

Description

[Experimental]

Export graph in DOCX format. See notes to understand why this is experimental.

Usage

```
export_graph_as_docx(
  g = NULL,
  plotnames = NULL,
  tblid,
  output_dir,
  title = NULL,
  footers = NULL,
  orientation = "portrait",
  plotwidth = 8,
  plotheight = 5.51,
  units = c("in", "cm", "mm", "px")[1],
  border = flextable::fp_border_default(width = 0.75, color = "black")
)
```

Arguments

`g`	(optional) Default = NULL. A `ggplot2` object, or a list of them, to export. At least one of `g` or `plotnames` must be provided. If both are provided, 'g' precedes and 'plotnames' will be ignored.
`plotnames`	(optional) Default = NULL. A file path, or a list of them, to previously saved .png files. These will be opened and exported in the output file. At least one of `g` or `plotnames` must be provided. If both are provided, 'g' precedes and 'plotnames' will be ignored.
`tblid`	Character. Output ID that will appear in the Title and footer.
`output_dir`	Character. File path where to save the output.
`title`	(optional) Default = NULL. Character, or list of them, with the titles to be displayed.
`footers`	(optional) Default = NULL. Character, or list of them, with the footers to be displayed.
`orientation`	(optional) Default = "portrait". One of: "portrait", "landscape".
`plotwidth`	(optional) Default = 8. Plot size in units expressed by the units argument. If not supplied, uses the size of the current graphics device.
`plotheight`	(optional) Default = 5.51. Plot size in units expressed by the units argument. If not supplied, uses the size of the current graphics device.

`units`	(optional) Default = "in". One of the following units in which the plotwidth and plotheight arguments are expressed: "in", "cm", "mm" or "px".
`border`	(optional). An `fp_border` object to use as borders for the Title and Footers.

Note

This function has been tested for common use cases but may not work or have unexpected or undesired behavior in corner cases. As such it is not considered fully production ready and is being made available for further testing and early adoption. Please report any issues you encounter to the developers. This function may be removed from junco in the future if the functionality is merged into `rtables.officer`.

`find_missing_chg_after_avisit` *Helper for Finding AVISIT after which CHG are all Missing*

Description

Helper for Finding AVISIT after which CHG are all Missing.

Usage

```
find_missing_chg_after_avisit(df)
```

Arguments

`df`	(`data.frame`) with `CHG` and `AVISIT` variables.

Value

A string with either the factor level after which `AVISIT` is all missing, or `NA`.

Examples

```
df <- data.frame(
  AVISIT = factor(c(1, 2, 3, 4, 5)),
  CHG = c(5, NA, NA, NA, 3)
)
find_missing_chg_after_avisit(df)

df2 <- data.frame(
  AVISIT = factor(c(1, 2, 3, 4, 5)),
  CHG = c(5, NA, 3, NA, NA)
)
find_missing_chg_after_avisit(df2)

df3 <- data.frame(
  AVISIT = factor(c(1, 2, 3, 4, 5)),
  CHG = c(NA, NA, NA, NA, NA)
)
find_missing_chg_after_avisit(df3)
```

`fit_ancova` ANCOVA *Analysis*

Description

Performs the `ANCOVA` analysis, separately for each visit.

Usage

```
fit_ancova(
  vars = list(response = "AVAL", covariates = c(), arm = "ARM", visit = "AVISIT", id =
    "USUBJID"),
  data,
  conf_level = 0.95,
  weights_emmeans = "proportional"
)
```

Arguments

`vars`	(named `list` of `string` or `character`) specifying the variables in the `ANCOVA` analysis. The following elements need to be included as character vectors and match corresponding columns in `data`: • `response`: the response variable. • `covariates`: the additional covariate terms (might also include interactions). • `id`: the subject ID variable (not really needed for the computations but for internal logistics). • `arm`: the treatment group variable (factor). • `visit`: the visit variable (factor). Note that the `arm` variable is by default included in the model, thus should not be part of `covariates`.
`data`	(`data.frame`) with all the variables specified in `vars`. Records with missing values in any independent variables will be excluded.
`conf_level`	(`proportion`) confidence level of the interval.
`weights_emmeans`	(`string`) argument from `emmeans::emmeans()`, `'counterfactual'` by default.

Value

A `tern_model` object which is a list with model results:

- `fit`: A list with a fitted `stats::lm()` result for each visit.
- `mse`: Mean squared error, i.e. variance estimate, for each visit.
- `df`: Degrees of freedom for the variance estimate for each visit.
- `lsmeans`: This is a list with data frames `estimates` and `contrasts`. The attribute `weights` savse the settings used (`weights_emmeans`).
- `vars`: The variable list.
- `labels`: Corresponding list with variable labels extracted from `data`.
- `ref_level`: The reference level for the arm variable, which is always the first level.
- `treatment_levels`: The treatment levels for the arm variable.
- `conf_level`: The confidence level which was used to construct the `lsmeans` confidence intervals.

Examples

```
library(mmrm)

fit <- fit_ancova(
  vars = list(
    response = "FEV1",
    covariates = c("RACE", "SEX"),
    arm = "ARMCD",
    id = "USUBJID",
    visit = "AVISIT"
  ),
  data = fev_data,
  conf_level = 0.9,
  weights_emmeans = "equal"
)
```

`fit_mmrm_j` MMRM *Analysis*

Description

Does the `MMRM` analysis. Multiple other functions can be called on the result to produce tables and graphs.

Usage

```
fit_mmrm_j(
  vars = list(response = "AVAL", covariates = c(), id = "USUBJID", arm = "ARM", visit =
    "AVISIT"),
  data,
  conf_level = 0.95,
  cor_struct = "unstructured",
  weights_emmeans = "counterfactual",
  averages_emmeans = list(),
  ...
)
```

Arguments

`vars`	(named `list` of `string` or `character`) specifying the variables in the `MMRM`. The following elements need to be included as character vectors and match corresponding columns in `data`: • `response`: the response variable. • `covariates`: the additional covariate terms (might also include interactions). • `id`: the subject ID variable. • `arm`: the treatment group variable (factor). • `visit`: the visit variable (factor). • `weights`: optional weights variable (if `NULL` or omitted then no weights will be used). Note that the main effects and interaction of `arm` and `visit` are by default included in the model.
`data`	(`data.frame`) with all the variables specified in `vars`. Records with missing values in any independent variables will be excluded.
`conf_level`	(`proportion`) confidence level of the interval.
`cor_struct`	(`string`) specifying the covariance structure, defaults to `'unstructured'`. See the details.
`weights_emmeans`	(`string`) argument from `emmeans::emmeans()`, `'counterfactual'` by default.
`averages_emmeans`	(`list`) optional named list of visit levels which should be averaged and reported along side the single visits.
`...`	additional arguments for `mmrm::mmrm()`, in particular `reml` and options listed in `mmrm::mmrm_control()`.

Details

Multiple different degree of freedom adjustments are available via the `method` argument for `mmrm::mmrm()`. In addition, covariance matrix adjustments are available via `vcov`. Please see `mmrm::mmrm_control()` for details and additional useful options.

For the covariance structure (`cor_struct`), the user can choose among the following options.

- `unstructured`: Unstructured covariance matrix. This is the most flexible choice and default. If there are `T` visits, then `T * (T+1) / 2` variance parameters are used.
- `toeplitz`: Homogeneous Toeplitz covariance matrix, which uses `T` variance parameters.
- `heterogeneous toeplitz`: Heterogeneous Toeplitz covariance matrix, which uses `2 * T - 1` variance parameters.
- `ante-dependence`: Homogeneous Ante-Dependence covariance matrix, which uses `T` variance parameters.
- `heterogeneous ante-dependence`: Heterogeneous Ante-Dependence covariance matrix, which uses `2 * T - 1` variance parameters.
- `auto-regressive`: Homogeneous Auto-Regressive (order 1) covariance matrix, which uses 2 variance parameters.
- `heterogeneous auto-regressive`: Heterogeneous Auto-Regressive (order 1) covariance matrix, which uses `T + 1` variance parameters.
- `compound symmetry`: Homogeneous Compound Symmetry covariance matrix, which uses 2 variance parameters.
- `heterogeneous compound symmetry`: Heterogeneous Compound Symmetry covariance matrix, which uses `T + 1` variance parameters.

Value

A `tern_model` object which is a list with model results:

- `fit`: The `mmrm` object which was fitted to the data. Note that via `mmrm::component(fit, 'optimizer')` the finally used optimization algorithm can be obtained, which can be useful for refitting the model later on.
- `cov_estimate`: The matrix with the covariance matrix estimate.
- `diagnostics`: A list with model diagnostic statistics (REML criterion, AIC, corrected AIC, BIC).
- `lsmeans`: This is a list with data frames `estimates` and `contrasts`. The attributes `averages` and `weights` save the settings used (`averages_emmeans` and `weights_emmeans`).
- `vars`: The variable list.
- `labels`: Corresponding list with variable labels extracted from `data`.
- `cor_struct`: input.
- `ref_level`: The reference level for the arm variable, which is always the first level.
- `treatment_levels`: The treatment levels for the arm variable.
- `conf_level`: The confidence level which was used to construct the `lsmeans` confidence intervals.
- `additional`: List with any additional inputs passed via `...`

Note

This function has the _j suffix to distinguish it from `mmrm::fit_mmrm()`. It is a copy from the `tern.mmrm` package and later will be replaced by tern.mmrm::fit_mmrm(). No new features are included in this function here.

Examples

```
mmrm_results <- fit_mmrm_j(
  vars = list(
    response = "FEV1",
    covariates = c("RACE", "SEX"),
    id = "USUBJID",
    arm = "ARMCD",
    visit = "AVISIT"
  ),
  data = mmrm::fev_data,
  cor_struct = "unstructured",
  weights_emmeans = "equal",
  averages_emmeans = list(
    "VIS1+2" = c("VIS1", "VIS2")
  )
)
```

`get_mmrm_lsmeans` *Extract Least Square Means from* `MMRM`

Description

Extracts the least square means from an `MMRM` fit.

Usage

```
get_mmrm_lsmeans(fit, vars, conf_level, weights, averages = list())
```

Arguments

`fit`	(mmrm) result of `mmrm::mmrm()`.
`vars`	(named `list` of `string` or `character`) specifying the variables in the `MMRM`. The following elements need to be included as character vectors and match corresponding columns in `data`: • `response`: the response variable. • `covariates`: the additional covariate terms (might also include interactions). • `id`: the subject ID variable. • `arm`: the treatment group variable (factor). • `visit`: the visit variable (factor).

	• `weights`: optional weights variable (if `NULL` or omitted then no weights will be used). Note that the main effects and interaction of `arm` and `visit` are by default included in the model.
`conf_level`	(`proportion`) confidence level of the interval.
`weights`	(`string`) type of weights to be used for the least square means, see `emmeans::emmeans()` for details.
`averages`	(`list`) named list of visit levels which should be averaged and reported along side the single visits.

Value

A list with data frames `estimates` and `contrasts`. The attributes `averages` and `weights` save the settings used.

`get_ref_info` *Obtain Reference Information for a Global Reference Group*

Description

This helper function can be used in custom analysis functions, by passing an extra argument `ref_path` which defines a global reference group by the corresponding column split hierarchy levels.

Usage

```
get_ref_info(ref_path, .spl_context, .var = NULL)
```

Arguments

`ref_path`	(`character`) reference group specification as an `rtables` `colpath`, see details.
`.spl_context`	(`data.frame`) see rtables::spl_context.
`.var`	(`character`) the variable being analyzed, see rtables::additional_fun_params.

Details

The reference group is specified in `colpath` hierarchical fashion in `ref_path`: the first column split variable is the first element, and the level to use is the second element. It continues until the last column split variable with last level to use. Note that depending on `.var`, either a `data.frame` (if `.var` is `NULL`) or a vector (otherwise) is returned. This allows usage for analysis functions with `df` and `x` arguments, respectively.

Value

A list with `ref_group` and `in_ref_col`, which can be used as `.ref_group` and `.in_ref_col` as if being directly passed to an analysis function by `rtables`, see rtables::additional_fun_params.

Examples

```
dm <- DM
dm$colspan_trt <- factor(
  ifelse(dm$ARM == "B: Placebo", " ", "Active Study Agent"),
  levels = c("Active Study Agent", " ")
)
colspan_trt_map <- create_colspan_map(
  dm,
  non_active_grp = "B: Placebo",
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Study Agent",
  colspan_var = "colspan_trt",
  trt_var = "ARM"
)

standard_afun <- function(x, .ref_group, .in_ref_col) {
  in_rows(
    "Difference of Averages" = non_ref_rcell(
      mean(x) - mean(.ref_group),
      is_ref = .in_ref_col,
      format = "xx.xx"
    )
  )
}

result_afun <- function(x, ref_path, .spl_context, .var) {
  ref <- get_ref_info(ref_path, .spl_context, .var)
  standard_afun(x, .ref_group = ref$ref_group, .in_ref_col = ref$in_ref_col)
}

ref_path <- c("colspan_trt", " ", "ARM", "B: Placebo")

lyt <- basic_table() |>
  split_cols_by(
    "colspan_trt",
    split_fun = trim_levels_to_map(map = colspan_trt_map)
  ) |>
  split_cols_by("ARM") |>
  analyze(
    "AGE",
    extra_args = list(ref_path = ref_path),
    afun = result_afun
  )

build_table(lyt, dm)
```

get_titles_from_file *Get Titles/Footers For Table From Sources*

Description

Retrieves the titles and footnotes for a given table from a CSV/XLSX file or a data.frame.

Usage

```
get_titles_from_file(
  id,
  file = .find_titles_file(input_path),
  input_path = ".",
  title_df = .read_titles_file(file)
)
```

Arguments

`id`	(`character(1)`) The identifier for the table of interest.
`file`	(`character(1)`) A path to CSV or xlsx file containing title and footer information for one or more outputs. See Details. Ignored if `title_df` is specified.
`input_path`	(`character(1)`) A path to look for titles.csv/titles.xlsx. Ignored if `file` or `title_df` is specified.
`title_df`	(`data.frame`) A data.frame containing titles and footers for one or more outputs. See Details.

Details

Retrieves the titles for a given output id (see below) and outputs a list containing the title and footnote objects supported by rtables. Both titles.csv and titles.xlsx (*if* `readxl` *is installed*) files are supported, with titles.csv being checked first.

```
Data is expected to have `TABLE ID`, `IDENTIFIER`, and `TEXT` columns,
where `IDENTIFIER` has the value `TITLE` for a title and `FOOT*` for
footer materials where `*` is a positive integer. `TEXT` contains
the value of the title/footer to be applied.
```

Value

List object containing: title, subtitles, main_footer, prov_footer for the table of interest. Note: the subtitles and prov_footer are currently set to NULL. Suitable for use with `set_titles()`.

get_visit_levels — *Get Visit Levels in Order Defined by Numeric Version*

Description

Get Visit Levels in Order Defined by Numeric Version

Usage

```
get_visit_levels(visit_cat, visit_n)
```

Arguments

`visit_cat`	(`character`) the categorical version.
`visit_n`	(`numeric`) the numeric version.

Value

The unique visit levels in the order defined by the numeric version.

Examples

```
get_visit_levels(
  visit_cat = c("Week 1", "Week 11", "Week 2"),
  visit_n = c(1, 5, 2)
)
```

h_get_trtvar_refpath — *Get Treatment Variable Reference Path*

Description

Retrieves the treatment variable reference path from the provided context.

Usage

```
h_get_trtvar_refpath(ref_path, .spl_context, df)
```

Arguments

`ref_path`	(`character`) Reference path for treatment variable.
`.spl_context`	(`data.frame`) Current split context.
`df`	(`data.frame`) Data frame.

Value

List containing treatment variable details.

---

h_odds_ratio *Helper functions for odds ratio estimation*

Description

[Stable]

Functions to calculate odds ratios in `s_odds_ratio_j()`.

Usage

```
or_glm_j(data, conf_level)

or_clogit_j(data, conf_level, method = "exact")

or_cmh(data, conf_level)
```

Arguments

`data`	(`data.frame`) data frame containing at least the variables `rsp` and `grp`, and optionally `strata` for `or_clogit_j()`.
`conf_level`	(`numeric`) confidence level for the confidence interval.
`method`	(`string`) whether to use the correct (`'exact'`) calculation in the conditional likelihood or one of the approximations, or the CMH method. See `survival::clogit()` for details.

Value

A named `list` of elements `or_ci`, `n_tot` and `pval`.

Functions

- `or_glm_j()`: Estimates the odds ratio based on `stats::glm()`. Note that there must be exactly 2 groups in `data` as specified by the `grp` variable.
- `or_clogit_j()`: Estimates the odds ratio based on `survival::clogit()`. This is done for the whole data set including all groups, since the results are not the same as when doing pairwise comparisons between the groups.
- `or_cmh()`: Estimates the odds ratio based on CMH. Note that there must be exactly 2 groups in `data` as specified by the `grp` variable.

See Also

odds_ratio

Examples

```
data <- data.frame(
  rsp = as.logical(c(1, 1, 0, 1, 0, 0, 1, 1)),
  grp = letters[c(1, 1, 1, 2, 2, 2, 1, 2)],
  strata = letters[c(1, 2, 1, 2, 2, 2, 1, 2)],
  stringsAsFactors = TRUE
)

or_glm_j(data, conf_level = 0.95)

data <- data.frame(
  rsp = as.logical(c(1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0)),
  grp = letters[c(1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3)],
  strata = LETTERS[c(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2)],
  stringsAsFactors = TRUE
)

or_clogit_j(data, conf_level = 0.95)

set.seed(123)
data <- data.frame(
  rsp = as.logical(rbinom(n = 40, size = 1, prob = 0.5)),
  grp = letters[sample(1:2, size = 40, replace = TRUE)],
  strata = LETTERS[sample(1:2, size = 40, replace = TRUE)],
  stringsAsFactors = TRUE
)

or_cmh(data, conf_level = 0.95)
```

`inches_to_spaces` *Conversion of inches to spaces.*

Description

Conversion of inches to spaces.

Usage

```
inches_to_spaces(ins, fontspec, raw = FALSE, tol = sqrt(.Machine$double.eps))
```

Arguments

`ins`	(numeric) Vector of widths in inches.

`fontspec`	(`font_spec`) The font specification to use.
`raw`	(`logical(1)`) Should the answer be returned unrounded (`TRUE`), or rounded to the nearest reasonable value (`FALSE`, the default).
`tol`	(`numeric(1)`) The numeric tolerance. Values between an integer `n`, and `n+tol` will be returned as `n`, rather than `n+1`, if `raw == FALSE`. Ignored when `raw` is `TRUE`.

Value

The number of either fractional (`raw = TRUE`) or whole (`raw = FALSE`) spaces that will fit within `ins` inches in the specified font.

`insert_blank_line` *Insertion of Blank Lines in a Layout*

Description

This is a hack for `rtables` in order to be able to add row gaps, i.e. blank lines. In particular, by default this function needs to maintain a global state for avoiding duplicate table names. The global state variable is hidden by using a dot in front of its name. However, this likely won't work with parallelisation across multiple threads and also causes non-reproducibility of the resulting `rtables` object. Therefore also a custom table name can be used.

Usage

```
insert_blank_line(lyt, table_names = NULL)
```

Arguments

`lyt`	(`layout`) input layout where analyses will be added to.
`table_names`	(`character`) this can be customized in case that the same `vars` are analyzed multiple times, to avoid warnings from `rtables`.

Value

The modified layout now including a blank line after the current row content.

Examples

```
ADSL <- ex_adsl

lyt <- basic_table() |>
  split_cols_by("ARM") |>
  split_rows_by("STRATA1") |>
  analyze(vars = "AGE", afun = function(x) {
    in_rows(
      "Mean (sd)" = rcell(c(mean(x), sd(x)), format = "xx.xx (xx.xx)")
    )
  }) |>
  insert_blank_line() |>
  analyze(vars = "AGE", table_names = "AGE_Range", afun = function(x) {
    in_rows(
      "Range" = rcell(range(x), format = "xx.xx - xx.xx")
    )
  })
build_table(lyt, ADSL)
```

`jjcsformat_xx` *Utility for specifying custom formats*

Description

Utility for specifying custom formats that can be used as a format in `formatters::format_value`

A function factory to generate formatting functions for p-value formatting that support rounding close to the significance level specified.

A function factory to generate formatting functions for range formatting that includes information about the censoring of survival times.

Usage

```
jjcsformat_xx(
  str,
  na_str = na_str_dflt,
  na_str_dflt = "NE",
  replace_na_dflt = TRUE
)

jjcsformat_pval_fct(alpha = 0.05)

jjcsformat_range_fct(str, censor_char = "+")
```

Arguments

`str`	(`string`) the format specifying the number of digits to be used, for the range values, e.g. `"xx.xx"`.

`na_str`	String for NA values.
`na_str_dflt`	Character to represent NA value
`replace_na_dflt`	logical(1). Should an `na_string` of "NA" within the formatters framework be overridden by `na_str_default`? Defaults to `TRUE`, as a way to have a different default na string behavior from the base `formatters` framework.
`alpha`	(`numeric`) the significance level to account for during rounding.
`censor_char`	(`string`) the character (of length 1) to be appended to `min` or `max`

Value

Either a supported format string, or a formatting function that can be used as format in `formatters::format_value`

The p-value in the standard format. If `count` is 0, the format is `0`. If it is smaller than 0.001, then `<0.001`, if it is larger than 0.999, then `>0.999` is returned. Otherwise, 3 digits are used. In the special case that rounding from below would make the string equal to the specified `alpha`, then a higher number of digits is used to be able to still see the difference. For example, 0.0048 is not rounded to 0.005 but stays at 0.0048 if `alpha = 0.005` is set.

A function that formats a numeric vector with 4 elements:

- minimum
- maximum
- censored minimum? (1 if censored, 0 if event)
- censored maximum? (1 if censored, 0 if event) The range along with the censoring information is returned as a string with the specified numeric format as `(min, max)`, and the `censor_char` is appended to `min` or `max` if these have been censored.

See Also

Other JJCS formatting functions: `count and fraction related formatting functions`

Examples

```
value <- c(1.65, 8.645)
fmt <- jjcsformat_xx("xx.x")
is.function(fmt)
fmt
format_value(value[1], fmt, round_type = "sas")
format_value(value[1], fmt, round_type = "iec")
if (is.function(fmt)) fmt(value[1])

fmt2 <- jjcsformat_xx("xx.x (xx.xxx)")
is.function(fmt2)
value <- c(1.65, 8.645)
format_value(value, fmt2, round_type = "sas")
format_value(value, fmt2, round_type = "iec")
# only possible when resulting format is a function
```

```
if (is.function(fmt2)) fmt2(value, round_type = "sas")

value <- c(1.65, NA)
format_value(value, fmt2, round_type = "iec", na_str = c("ne1", "ne2"))
if (is.function(fmt2)) fmt2(value, round_type = "iec", na_str = c("ne1", "ne2"))
my_pval_format <- jjcsformat_pval_fct(0.005)
my_pval_format(0.2802359)
my_pval_format(0.0048)
my_pval_format(0.00499)
my_pval_format(0.004999999)
my_pval_format(0.0051)
my_pval_format(0.0009)
my_pval_format(0.9991)

my_range_format <- jjcsformat_range_fct("xx.xx")
my_range_format(c(0.35235, 99.2342, 1, 0))
my_range_format(c(0.35235, 99.2342, 0, 1))
my_range_format(c(0.35235, 99.2342, 0, 0))
my_range_format(c(0.35235, 99.2342, 1, 1))
my_range_format <- jjcsformat_range_fct("xx.xx", censor_char = "*")
my_range_format(c(0.35235, 99.2342, 1, 1))
```

`jjcs_num_formats` *Numeric Formatting Function*

Description

Formatting setter for selected numerical statistics.

Usage

```
jjcs_num_formats(d, cap = 4)
```

Arguments

d	(`numeric`) precision of individual values
cap	(`numeric`) cap to numerical precision (d > cap – will use precision as if cap was specified as precision)

Value

list:

- fmt : named vector with formatting function (jjcsformat_xx) for numerical stats: range, median, mean_sd, sd
- spec : named vector with formatting specifications for numerical stats: range, median, mean_sd, sd

Examples

```
P1_precision <- jjcs_num_formats(d = 0)$fmt
jjcs_num_formats(2)$fmt
jjcs_num_formats(2)$spec
```

`jj_complex_scorefun` *Complex Scoring Function*

Description

A function used for sorting AE tables (and others) as required.

Usage

```
jj_complex_scorefun(
  spanningheadercolvar = "colspan_trt",
  usefirstcol = FALSE,
  colpath = NULL,
  firstcat = NULL,
  lastcat = NULL
)
```

Arguments

`spanningheadercolvar`	(character) Name of spanning header variable that defines the active treatment columns. If you do not have an active treatment spanning header column then user can define this as NA.
`usefirstcol`	(logical) This allows you to just use the first column of the table to sort on.
`colpath`	(character) Name of column path that is needed to sort by (default=NULL). This overrides other arguments if specified (except firstcat and lastcat which will be applied if requested on this colpath).
`firstcat`	(logical) If you wish to put any category at the top of the list despite any n's, user can specify it here.
`lastcat`	(logical) If you wish to put any category at the bottom of the list despite any n's, user can specify it here.

Details

This sort function sorts as follows:

- Takes all the columns from a specified spanning column header (default= colspan_trt) and sorts by the last treatment column within this.
- If no spanning column header variable exists (e.g you have only one active treatment arm and have decided to remove the spanning header from your layout), it will sort by the first treatment column in your table.

This function is not really designed for tables that have sub-columns. However, if users wish to override any default sorting behavior, they can simply specify their own colpath to use for sorting on (default = NULL)

Value

A function which can be used as a score function (scorefun in `sort_at_path`).

Examples

```
library(dplyr)
ADAE <- data.frame(
  USUBJID = c(
    "XXXXX01", "XXXXX02", "XXXXX03", "XXXXX04", "XXXXX05",
    "XXXXX06", "XXXXX07", "XXXXX08", "XXXXX09", "XXXXX10"
  ),
  AEBODSYS = c(
    "SOC 1", "SOC 2", "SOC 1", "SOC 2", "SOC 2",
    "SOC 2", "SOC 2", "SOC 1", "SOC 2", "SOC 1"
  ),
  AEDECOD = c(
    "Coded Term 2", "Coded Term 1", "Coded Term 3", "Coded Term 4",
    "Coded Term 4", "Coded Term 4", "Coded Term 5", "Coded Term 3",
    "Coded Term 1", "Coded Term 2"
  ),
  TRT01A = c(
    "ARMA", "ARMB", "ARMA", "ARMB", "ARMB",
    "Placebo", "Placebo", "Placebo", "ARMA", "ARMB"
  ),
  TRTEMFL = c("Y", "Y", "N", "Y", "Y", "Y", "Y", "N", "Y", "Y")
)

ADAE <- ADAE |>
  dplyr::mutate(TRT01A = as.factor(TRT01A))

ADAE$colspan_trt <- factor(ifelse(ADAE$TRT01A == "Placebo", " ", "Active Study Agent"),
  levels = c("Active Study Agent", " ")
)

ADAE$rrisk_header <- "Risk Difference (%) (95% CI)"
ADAE$rrisk_label <- paste(ADAE$TRT01A, paste("vs", "Placebo"))

colspan_trt_map <- create_colspan_map(ADAE,
  non_active_grp = "Placebo",
  non_active_grp_span_lbl = " ",
  active_grp_span_lbl = "Active Study Agent",
  colspan_var = "colspan_trt",
  trt_var = "TRT01A"
)

ref_path <- c("colspan_trt", " ", "TRT01A", "Placebo")

ADSL <- unique(ADAE |> select(USUBJID, "colspan_trt", "rrisk_header", "rrisk_label", "TRT01A"))

lyt <- basic_table() |>
  split_cols_by(
    "colspan_trt",
    split_fun = trim_levels_to_map(map = colspan_trt_map)
  ) |>
  split_cols_by("TRT01A") |>
  split_cols_by("rrisk_header", nested = FALSE) |>
  split_cols_by(
    "TRT01A",
    labels_var = "rrisk_label",
    split_fun = remove_split_levels("Placebo")
  ) |>
  analyze(
    "TRTEMFL",
    a_freq_j,
    show_labels = "hidden",
    extra_args = list(
      method = "wald",
      label = "Subjects with >=1 AE",
      ref_path = ref_path,
      .stats = "count_unique_fraction"
    )
  ) |>
  split_rows_by("AEBODSYS",
    split_label = "System Organ Class",
    split_fun = trim_levels_in_group("AEDECOD"),
    label_pos = "topleft",
    section_div = c(" "),
    nested = FALSE
  ) |>
  summarize_row_groups(
    "AEBODSYS",
    cfun = a_freq_j,
    extra_args = list(
      method = "wald",
      ref_path = ref_path,
      .stats = "count_unique_fraction"
    )
  ) |>
  analyze(
    "AEDECOD",
    afun = a_freq_j,
    extra_args = list(
      method = "wald",
      ref_path = ref_path,
      .stats = "count_unique_fraction"
    )
  )

result <- build_table(lyt, ADAE, alt_counts_df = ADSL)

result

result <- sort_at_path(
  result,
  c("root", "AEBODSYS"),
  scorefun = jj_complex_scorefun()
)

result <- sort_at_path(
  result,
  c("root", "AEBODSYS", "*", "AEDECOD"),
  scorefun = jj_complex_scorefun()
)

result
```

`keep_non_null_rows` *Pruning Function to accommodate removal of completely NULL rows within a table*

Description

Condition function on individual analysis rows. Flag as FALSE when all columns are NULL, as then the row should not be kept. To be utilized as a row_condition in function tern::keep_rows

Usage

```
keep_non_null_rows(tr)
```

Arguments

`tr`	(`TableTree`) The TableTree object to prune.

Value

A function that can be utilized as a row_condition in the tern::keep_rows function.

Examples

```
library(dplyr)

ADSL <- data.frame(
  USUBJID = c(
    "XXXXX01", "XXXXX02", "XXXXX03", "XXXXX04", "XXXXX05",
    "XXXXX06", "XXXXX07", "XXXXX08", "XXXXX09", "XXXXX10"
  ),
  TRT01P = c(
    "ARMA", "ARMB", "ARMA", "ARMB", "ARMB", "Placebo",
    "Placebo", "Placebo", "ARMA", "ARMB"
  ),
  AGE = c(34, 56, 75, 81, 45, 75, 48, 19, 32, 31),
  SAFFL = c("N", "N", "N", "N", "N", "N", "N", "N", "N", "N"),
  PKFL = c("N", "N", "N", "N", "N", "N", "N", "N", "N", "N")
)

ADSL <- ADSL |>
  mutate(TRT01P = as.factor(TRT01P))

create_blank_line <- function(x) {
  list(
    "Mean" = rcell(mean(x), format = "xx.x"),
    " " = rcell(NULL),
    "Max" = rcell(max(x))
  )
}

lyt <- basic_table() |>
  split_cols_by("TRT01P") |>
  analyze("AGE", afun = create_blank_line)

result <- build_table(lyt, ADSL)

result
result <- prune_table(result, prune_func = tern::keep_rows(keep_non_null_rows))

result
```

`listing_column_widths` *Define Column Widths*

Description

`def_colwidths` uses heuristics to determine suitable column widths given a table or listing, and a font.

Usage

```
listing_column_widths(
  mpf,
  incl_header = TRUE,
  col_gap = 0.5,
  pg_width_ins = 8.88,
  fontspec = font_spec("Times", 8, 1.2),
  verbose = FALSE
)

def_colwidths(
  tt,
  fontspec,
  label_width_ins = 2,
  col_gap = ifelse(type == "Listing", 0.5, 3),
  type = tlg_type(tt)
)
```

Arguments

`mpf`	(`listing_df` or `MatrixPrintForm` derived thereof) The listing calculate column widths for.
`incl_header`	(`logical(1)`) Should the constraint to not break up individual words be extended to words in the column labels? Defaults to `TRUE`
`col_gap`	Column gap in spaces. Defaults to `.5` for listings and `3` for tables.
`pg_width_ins`	(`numeric(1)`) Number of inches in width for *the portion of the page the listing will be printed to*. Defaults to `8.88` which corresponds to landscape orientation on a standard page after margins.
`fontspec`	Font specification
`verbose`	(`logical(1)`) Should additional information messages be displayed during the calculation of the column widths? Defaults to `FALSE`.
`tt`	input TableTree
`label_width_ins`	Label Width in Inches.
`type`	Type of the TableTree, used to determine column width calculation method.

Details

Listings are assumed to be rendered landscape on standard A1 paper, such that all columns are rendered on one page. Tables are allowed to be horizontally paginated, and column widths are determined based only on required word wrapping. See the `Automatic Column Widths` vignette for a detailed discussion of the algorithms used.

Value

- `listing_column_widths`: a vector of column widths suitable to use in `tt_to_tlgrtf` and other exporters.
- `def_colwidths`: a vector of column widths (including the label row pseudo-column in the table case) suitable for use rendering `tt` in the specified font.

`make_combo_splitfun` *Split Function Helper*

Description

A function which aids the construction for users to create their own split function for combined columns.

Usage

```
make_combo_splitfun(nm, label = nm, levels = NULL, rm_other_facets = TRUE)
```

Arguments

`nm`	(`character`) Name/virtual 'value' for the new facet.
`label`	(`character`) Label for the new facet.
`levels`	(`character` or NULL) The levels to combine into the new facet, or NULL, indicating the facet should include all incoming data.
`rm_other_facets`	(`logical`) Should facets other than the newly created one be removed. Defaults to `TRUE`.

Value

Function usable directly as a split function.

Examples

```
aesevall_spf <- make_combo_splitfun(nm = "AESEV_ALL", label = "Any AE", levels = NULL)
```

make_rbmi_cluster — *Create a* rbmi *ready cluster*

Description

This function is a wrapper around `parallel::makePSOCKcluster()` but takes care of configuring `rbmi` to be used in the sub-processes as well as loading user defined objects and libraries and setting the seed for reproducibility.

Usage

```
make_rbmi_cluster(cluster_or_cores = 1, objects = NULL, packages = NULL)
```

Arguments

`cluster_or_cores`	(`integer` or `cluster object`) Number of parallel processes to use or an existing cluster to make use of
`objects`	(`list`) A named list of objects to export into the sub-processes
`packages`	(`character vector`) A character vector of libraries to load in the sub-processes

Value

- If `cluster_or_cores` is `1`, this function will return `NULL`.
- If `cluster_or_cores` is a number greater than `1`, a cluster with `cluster_or_cores` cores is returned.
- If `cluster_or_cores` is a cluster created via `parallel::makeCluster()`, then this function returns it after inserting the relevant `rbmi` objects into the existing cluster.

Examples

```
## Not run:
make_rbmi_cluster(5)
closeAllConnections()

VALUE <- 5
myfun <- function(x) {
  x + day(VALUE)
}
make_rbmi_cluster(5, list(VALUE = VALUE, myfun = myfun), c("lubridate"))
closeAllConnections()

cl <- parallel::makeCluster(5)
make_rbmi_cluster(cl)
closeAllConnections()

## End(Not run)
```

odds_ratio *Odds ratio estimation*

Description

[Stable] A set of functions for Odds-Ratio (OR) calculation.

Usage

```
a_odds_ratio_j(
  df,
  .var,
  .df_row,
  ref_path,
  .spl_context,
  ...,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL
)

s_odds_ratio_j(
  df,
  .var,
  .ref_group,
  .in_ref_col,
  .df_row,
  variables = list(arm = NULL, strata = NULL),
  conf_level = 0.95,
  groups_list = NULL,
  na_if_no_events = TRUE,
  method = c("exact", "approximate", "efron", "breslow", "cmh")
)
```

Arguments

`df`	(`data.frame`) input data frame.
`.var`	(`string`) name of the response variable.
`.df_row`	(`data.frame`) data frame containing all rows.
`ref_path`	(`character`) path to the reference group.

`.spl_context`	(`environment`) split context environment.
`...`	Additional arguments passed to the statistics function.
`.stats`	(`character`) statistics to calculate.
`.formats`	(`list`) formats for the statistics.
`.labels`	(`list`) labels for the statistics.
`.indent_mods`	(`list`) indentation modifications for the statistics.
`.ref_group`	(`data.frame`) reference group data frame.
`.in_ref_col`	(`logical`) whether the current column is the reference column.
`variables`	(`list`) list with arm and strata variable names.
`conf_level`	(`numeric`) confidence level for the confidence interval.
`groups_list`	(`list`) list of groups for combination.
`na_if_no_events`	(`flag`) whether the point estimate should be `NA` if there are no events in one arm. The p-value and confidence interval will still be computed.
`method`	(`string`) whether to use the correct (`'exact'`) calculation in the conditional likelihood or one of the approximations, or the CMH method. See `survival::clogit()` for details.

Value

- `a_odds_ratio_j()` returns the corresponding list with formatted `rtables::CellValue()`.
- `s_odds_ratio_j()` returns a named list with the statistics `or_ci` (containing `est`, `lcl`, and `ucl`), `pval` and `n_tot`.

Functions

- `a_odds_ratio_j()`: Formatted analysis function which is used as `afun`. Note that the junco specific `ref_path` and `.spl_context` arguments are used for reference column information.
- `s_odds_ratio_j()`: Statistics function which estimates the odds ratio between a treatment and a control. A `variables` list with `arm` and `strata` variable names must be passed if a stratified analysis is required.

Note

The `a_odds_ratio_j()` and `s_odds_ratio_j()` functions have the `_j` suffix to distinguish them from `tern::a_odds_ratio()` and `tern::s_odds_ratio()`, respectively. These functions differ as follows:

- Additional `method = 'cmh'` option is provided to calculate the Cochran-Mantel-Haenszel estimate.
- The p-value is returned as an additional statistic.

Once these updates are contributed back to `tern`, they can later be replaced by the `tern` versions.

Examples

```
set.seed(12)
dta <- data.frame(
  rsp = sample(c(TRUE, FALSE), 100, TRUE),
  grp = factor(rep(c("A", "B"), each = 50), levels = c("A", "B")),
  strata = factor(sample(c("C", "D"), 100, TRUE))
)

a_odds_ratio_j(
  df = subset(dta, grp == "A"),
  .var = "rsp",
  ref_path = c("grp", "B"),
  .spl_context = data.frame(
    cur_col_split = I(list("grp")),
    cur_col_split_val = I(list(c(grp = "A"))),
    full_parent_df = I(list(dta))
  ),
  .df_row = dta
)


l <- basic_table() |>
  split_cols_by(var = "grp") |>
  analyze(
    "rsp",
    afun = a_odds_ratio_j,
    show_labels = "hidden",
    extra_args = list(
      ref_path = c("grp", "B"),
      .stats = c("or_ci", "pval")
    )
  )

build_table(l, df = dta)

l2 <- basic_table() |>
  split_cols_by(var = "grp") |>
  analyze(
    "rsp",
    afun = a_odds_ratio_j,
    show_labels = "hidden",
    extra_args = list(
      variables = list(arm = "grp", strata = "strata"),
      method = "cmh",
      ref_path = c("grp", "A"),
      .stats = c("or_ci", "pval")
    )
  )

build_table(l2, df = dta)
s_odds_ratio_j(
  df = subset(dta, grp == "A"),
  .var = "rsp",
  .ref_group = subset(dta, grp == "B"),
  .in_ref_col = FALSE,
  .df_row = dta
)

s_odds_ratio_j(
  df = subset(dta, grp == "A"),
  .var = "rsp",
  .ref_group = subset(dta, grp == "B"),
  .in_ref_col = FALSE,
  .df_row = dta,
  variables = list(arm = "grp", strata = "strata")
)

s_odds_ratio_j(
  df = subset(dta, grp == "A"),
  method = "cmh",
  .var = "rsp",
  .ref_group = subset(dta, grp == "B"),
  .in_ref_col = FALSE,
  .df_row = dta,
  variables = list(arm = "grp", strata = c("strata"))
)
```

`par_lapply` *Parallelise Lapply*

Description

Simple wrapper around `lapply` and `parallel::clusterApplyLB` to abstract away the logic of deciding which one to use.

Usage

```
par_lapply(cl, fun, x, ...)
```

Arguments

cl	(cluster object) Cluster created by `parallel::makeCluster()` or NULL
fun	(functions) Function to be run
x	(object) Object to be looped over
...	Extra arguments passed to fun

Value

list of results of calling fun on elements of x.

---

prop_diff *Proportion difference estimation*

Description

The analysis function `a_proportion_diff_j()` can be used to create a layout element to estimate the difference in proportion of responders within a studied population. The primary analysis variable, vars, is a logical variable indicating whether a response has occurred for each record. See the method parameter for options of methods to use when constructing the confidence interval of the proportion difference. A stratification variable can be supplied via the strata element of the variables argument.

Usage

```
a_proportion_diff_j(
  df,
  .var,
  ref_path,
  .spl_context,
  ...,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL
)

s_proportion_diff_j(
  df,
  .var,
  .ref_group,
  .in_ref_col,
  variables = list(strata = NULL),
  conf_level = 0.95,
  method = c("waldcc", "wald", "cmh", "cmh_sato", "cmh_mn", "ha", "newcombe",
    "newcombecc", "strat_newcombe", "strat_newcombecc"),
  weights_method = "cmh"
)
```

Arguments

`df`	(`data.frame`) input data frame.
`.var`	(`string`) name of the response variable.
`ref_path`	(`character`) path to the reference group.
`.spl_context`	(`environment`) split context environment.
`...`	Additional arguments passed to the statistics function.
`.stats`	(`character`) statistics to calculate.
`.formats`	(`list`) formats for the statistics.
`.labels`	(`list`) labels for the statistics.
`.indent_mods`	(`list`) indentation modifications for the statistics.
`.ref_group`	(`data.frame`) reference group data frame.
`.in_ref_col`	(`logical`) whether the current column is the reference column.
`variables`	(`list`) list with strata variable names.
`conf_level`	(`numeric`) confidence level for the confidence interval.
`method`	(`string`) method to use for confidence interval calculation.
`weights_method`	(`string`) method to use for weights calculation in stratified analysis.

Value

- `a_proportion_diff_j()` returns the corresponding list with formatted `rtables::CellValue()`.
- `s_proportion_diff_j()` returns a named list of elements `diff`, `diff_ci`, `diff_est_ci` and `diff_ci_3d`.

Functions

- `a_proportion_diff_j()`: Formatted analysis function which is used as `afun` in `estimate_proportion_diff()`.
- `s_proportion_diff_j()`: Statistics function estimating the difference in terms of responder proportion.

Note

The `a_proportion_diff_j()` function has the `_j` suffix to distinguish it from `tern::a_proportion_diff()`. The functions here are a copy from the `tern` package with additional features:

- Additional statistic `diff_est_ci` is returned.
- `ref_path` needs to be provided as extra argument to specify the control group column.

When performing an unstratified analysis, methods `'cmh'`, `'cmh_sato'`, `'cmh_mn'`, `'strat_newcombe'`, and `'strat_newcombecc'` are not permitted.

Examples

```
nex <- 100
dta <- data.frame(
  "rsp" = sample(c(TRUE, FALSE), nex, TRUE),
  "grp" = sample(c("A", "B"), nex, TRUE),
  "f1" = sample(c("a1", "a2"), nex, TRUE),
  "f2" = sample(c("x", "y", "z"), nex, TRUE),
  stringsAsFactors = TRUE
)

l <- basic_table() |>
  split_cols_by(var = "grp") |>
  analyze(
    vars = "rsp",
    afun = a_proportion_diff_j,
    show_labels = "hidden",
    na_str = tern::default_na_str(),
    extra_args = list(
      conf_level = 0.9,
      method = "ha",
      ref_path = c("grp", "B")
    )
  )

build_table(l, df = dta)

s_proportion_diff_j(
  df = subset(dta, grp == "A"),
  .var = "rsp",
  .ref_group = subset(dta, grp == "B"),
  .in_ref_col = FALSE,
  conf_level = 0.90,
  method = "ha"
)
```

```
s_proportion_diff_j(
  df = subset(dta, grp == "A"),
  .var = "rsp",
  .ref_group = subset(dta, grp == "B"),
  .in_ref_col = FALSE,
  variables = list(strata = c("f1", "f2")),
  conf_level = 0.90,
  method = "cmh"
)
```

`prop_diff_test` *Difference test for two proportions*

Description

[Stable]

The analysis function `a_test_proportion_diff()` can be used to create a layout element to test the difference between two proportions. The primary analysis variable, `vars`, indicates whether a response has occurred for each record. See the `method` parameter for options of methods to use to calculate the p-value. Additionally, a stratification variable can be supplied via the `strata` element of the `variables` argument. The argument `alternative` specifies the direction of the alternative hypothesis.

Usage

```
a_test_proportion_diff(
  df,
  .var,
  ref_path,
  .spl_context,
  ...,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.

`ref_path`	(`character`) global reference group specification, see `get_ref_info()`.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`...`	Additional arguments passed to `tern::s_test_proportion_diff()`, including: • `method` (`string`) one of `chisq`, `cmh`, `cmh_wh`, `fisher` or `schouten`; specifies the test used to calculate the p-value.
`.stats`	(`character`) statistics to select for the table.
`.formats`	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.
`.labels`	(named `character`) labels for the statistics (without indent).
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.

Value

- `a_test_proportion_diff()` returns the corresponding list with formatted `rtables::CellValue()`.

Functions

- `a_test_proportion_diff()`: Formatted analysis function which is used as `afun`

Note

This function has been forked from the `tern` package. Additional features are:

- Additional `ref_path` argument for flexible reference column path specification.

Examples

```
dta <- data.frame(
  rsp = sample(c(TRUE, FALSE), 100, TRUE),
  grp = factor(rep(c("A", "B"), each = 50)),
  strata = factor(rep(c("V", "W", "X", "Y", "Z"), each = 20))
)

l <- basic_table() |>
  split_cols_by(var = "grp") |>
  analyze(
    vars = "rsp",
    afun = a_test_proportion_diff,
    show_labels = "hidden",
    extra_args = list(
      method = "cmh",
      variables = list(strata = "strata"),
      ref_path = c("grp", "B")
    )
  )

build_table(l, df = dta)
```

`prop_post_fun` *Split Function for Proportion Analysis Columns*

Description

Here we just split into 3 columns `n`, `%` and `Cum %`.

Usage

```
prop_post_fun(ret, spl, fulldf, .spl_context)

prop_split_fun(df, spl, vals = NULL, labels = NULL, trim = FALSE, .spl_context)
```

Arguments

`ret`	(`list`) return value from the previous split function.
`spl`	(`list`) split information.
`fulldf`	(`data.frame`) full data frame.
`.spl_context`	(`environment`) split context environment.
`df`	A data frame that contains all analysis variables.
`vals`	A character vector that contains values to use for the split.
`labels`	A character vector that contains labels for the statistics (without indent).
`trim`	A single logical that indicates whether to trim the values.

Value

a split function for use in rtables::split_rows_by.

See Also

`rtables::make_split_fun()` describing the requirements for this kind of post-processing function.

prop_ratio_cmh *Relative Risk CMH Statistic*

Description

Calculates the relative risk which is defined as the ratio between the response rates between the experimental treatment group and the control treatment group, adjusted for stratification factors by applying Cochran-Mantel-Haenszel (CMH) weights.

Usage

```
prop_ratio_cmh(rsp, grp, strata, conf_level = 0.95)
```

Arguments

`rsp`	(`logical`) whether each subject is a responder or not.
`grp`	(`factor`) defining the groups.
`strata`	(`factor`) variable with one level per stratum and same length as `rsp`.
`conf_level`	(`proportion`) confidence level of the interval.

Value

A list with elements `rel_risk_ci` and `pval`.

Examples

```
set.seed(2)
rsp <- sample(c(TRUE, FALSE), 100, TRUE)
grp <- sample(c("Placebo", "Treatment"), 100, TRUE)
grp <- factor(grp, levels = c("Placebo", "Treatment"))
strata_data <- data.frame(
  "f1" = sample(c("a", "b"), 100, TRUE),
  "f2" = sample(c("x", "y", "z"), 100, TRUE),
  stringsAsFactors = TRUE
)

prop_ratio_cmh(
  rsp = rsp, grp = grp, strata = interaction(strata_data),
  conf_level = 0.90
)
```

`prop_table_afun` *Formatted Analysis Function for Proportion Analysis*

Description

This function applies to a factor `x` when a column split was prepared with `prop_split_fun()` before.

Usage

```
prop_table_afun(x, .spl_context, formats, add_total_level = FALSE)
```

Arguments

`x`	(`factor`) factor variable to analyze.
`.spl_context`	(`environment`) split context environment.
`formats`	(`list`) formats for the statistics.
`add_total_level`	(`flag`) whether to add a total level.

Details

In the column named `n`, the counts of the categories as well as an optional `Total` count will be shown. In the column named `percent`, the percentages of the categories will be shown, with an optional blank entry for `Total`. In the column named `cum_percent`, the cumulative percentages will be shown instead.

Value

A `VerticalRowsSection` as returned by rtables::in_rows.

`rbmi_analyse` *Analyse Multiple Imputed Datasets*

Description

This function takes multiple imputed datasets (as generated by the impute() function from the rbmi package) and runs an analysis function on each of them.

Usage

```
rbmi_analyse(
  imputations,
  fun = rbmi_ancova,
  delta = NULL,
  ...,
  cluster_or_cores = 1,
  .validate = TRUE
)
```

Arguments

`imputations`	An `imputations` object as created by the impute() function from the rbmi package.
`fun`	An analysis function to be applied to each imputed dataset. See details.
`delta`	A `data.frame` containing the delta transformation to be applied to the imputed datasets prior to running `fun`. See details.
`...`	Additional arguments passed onto `fun`.
`cluster_or_cores`	(`numeric` or `cluster object`) The number of parallel processes to use when running this function. Can also be a cluster object created by `make_rbmi_cluster()`. See the parallelisation section below.
`.validate`	(`logical`) Should `imputations` be checked to ensure it conforms to the required format (default = `TRUE`) ? Can gain a small performance increase if this is set to `FALSE` when analysing a large number of samples.

Details

This function works by performing the following steps:

1. Extract a dataset from the `imputations` object.
2. Apply any delta adjustments as specified by the `delta` argument.
3. Run the analysis function `fun` on the dataset.
4. Repeat steps 1-3 across all of the datasets inside the `imputations` object.
5. Collect and return all of the analysis results.

The analysis function `fun` must take a `data.frame` as its first argument. All other options to `rbmi_analyse()` are passed onto `fun` via `...`. `fun` must return a named list with each element itself being a list containing a single numeric element called `est` (or additionally `se` and `df` if you had originally specified the method_bayes() or method_approxbayes() functions from the rbmi package) i.e.:

```
myfun <- function(dat, ...) {
    mod_1 <- lm(data = dat, outcome ~ group)
    mod_2 <- lm(data = dat, outcome ~ group + covar)
    x <- list(
        trt_1 = list(
            est = coef(mod_1)[['group']],  # Use [[ ]] for safety
            se = sqrt(vcov(mod_1)['group', 'group']), # Use ['','']
            df = df.residual(mod_1)
        ),
        trt_2 = list(
            est = coef(mod_2)[['group']],  # Use [[ ]] for safety
            se = sqrt(vcov(mod_2)['group', 'group']), # Use ['','']
            df = df.residual(mod_2)
        )
     )
     return(x)
 }
```

Please note that the `vars$subjid` column (as defined in the original call to the draws() function from the rbmi package) will be scrambled in the data.frames that are provided to `fun`. This is to say they will not contain the original subject values and as such any hard coding of subject ids is strictly to be avoided.

By default `fun` is the `rbmi_ancova()` function. Please note that this function requires that a `vars` object, as created by the set_vars() function from the rbmi package, is provided via the `vars` argument e.g. `rbmi_analyse(imputeObj, vars = set_vars(...))`. Please see the documentation for `rbmi_ancova()` for full details. Please also note that the theoretical justification for the conditional mean imputation method (`method = method_condmean()` in the draws() function from the rbmi package) relies on the fact that ANCOVA is a linear transformation of the outcomes. Thus care is required when applying alternative analysis functions in this setting.

The `delta` argument can be used to specify offsets to be applied to the outcome variable in the imputed datasets prior to the analysis. This is typically used for sensitivity or tipping point analyses. The delta dataset must contain columns `vars$subjid`, `vars$visit` (as specified in the original call to the draws() function from the rbmi package) and `delta`. Essentially this `data.frame` is merged onto the imputed dataset by `vars$subjid` and `vars$visit` and then the outcome variable is modified by:

```
imputed_data[[vars$outcome]] <- imputed_data[[vars$outcome]] + imputed_data[['delta']]
```

Please note that in order to provide maximum flexibility, the `delta` argument can be used to modify any/all outcome values including those that were not imputed. Care must be taken when defining offsets. It is recommend that you use the helper function delta_template() from the rbmi package to define the delta datasets as this provides utility variables such as `is_missing` which can be used to identify exactly which visits have been imputed.

Value

An `analysis` object, as defined by `rbmi`, representing the desired analysis applied to each of the imputed datasets in `imputations`.

Parallelisation

To speed up the evaluation of `rbmi_analyse()` you can use the `cluster_or_cores` argument to enable parallelisation. Simply providing an integer will get `rbmi` to automatically spawn that many background processes to parallelise across. If you are using a custom analysis function then you need to ensure that any libraries or global objects required by your function are available in the sub-processes. To do this you need to use the `make_rbmi_cluster()` function for example:

```
my_custom_fun <- function(...) <some analysis code>
cl <- make_rbmi_cluster(
    4,
    objects = list('my_custom_fun' = my_custom_fun),
    packages = c('dplyr', 'nlme')
)
rbmi_analyse(
    imputations = imputeObj,
    fun = my_custom_fun,
    cluster_or_cores = cl
)
parallel::stopCluster(cl)
```

Note that there is significant overhead both with setting up the sub-processes and with transferring data back-and-forth between the main process and the sub-processes. As such parallelisation of the `rbmi_analyse()` function tends to only be worth it when you have `> 2000` samples generated by the draws() function from the rbmi package. Conversely using parallelisation if your samples are smaller than this may lead to longer run times than just running it sequentially.

It is important to note that the implementation of parallel processing within the analyse() function from the rbmi package has been optimised around the assumption that the parallel processes will be spawned on the same machine and not a remote cluster. One such optimisation is that the required data is saved to a temporary file on the local disk from which it is then read into each sub-process. This is done to avoid the overhead of transferring the data over the network. Our assumption is that if you are at the stage where you need to be parallelising your analysis over a remote cluster then you would likely be better off parallelising across multiple `rbmi` runs rather than within a single `rbmi` run.

Finally, if you are doing a tipping point analysis you can get a reasonable performance improvement by re-using the cluster between each call to `rbmi_analyse()` e.g.

```
cl <- make_rbmi_cluster(4)
ana_1 <- rbmi_analyse(
    imputations = imputeObj,
    delta = delta_plan_1,
    cluster_or_cores = cl
)
ana_2 <- rbmi_analyse(
    imputations = imputeObj,
    delta = delta_plan_2,
    cluster_or_cores = cl
)
```

```
ana_3 <- rbmi_analyse(
    imputations = imputeObj,
    delta = delta_plan_3,
    cluster_or_cores = cl
)
parallel::clusterStop(cl)
```

See Also

The extract_imputed_dfs() function from the rbmi package for manually extracting imputed datasets.

The delta_template() function from the rbmi package for creating delta data.frames.

`rbmi_ancova()` for the default analysis function.

Examples

```
library(rbmi)
library(dplyr)

dat <- antidepressant_data
dat$GENDER <- as.factor(dat$GENDER)
dat$POOLINV <- as.factor(dat$POOLINV)
set.seed(123)
pat_ids <- sample(levels(dat$PATIENT), nlevels(dat$PATIENT) / 4)
dat <- dat |>
  filter(PATIENT %in% pat_ids) |>
  droplevels()
dat <- expand_locf(
  dat,
  PATIENT = levels(dat$PATIENT),
  VISIT = levels(dat$VISIT),
  vars = c("BASVAL", "THERAPY"),
  group = c("PATIENT"),
  order = c("PATIENT", "VISIT")
)
dat_ice <- dat |>
  arrange(PATIENT, VISIT) |>
  filter(is.na(CHANGE)) |>
  group_by(PATIENT) |>
  slice(1) |>
  ungroup() |>
  select(PATIENT, VISIT) |>
  mutate(strategy = "JR")
dat_ice <- dat_ice[-which(dat_ice$PATIENT == 3618), ]
vars <- set_vars(
  outcome = "CHANGE",
  visit = "VISIT",
  subjid = "PATIENT",
  group = "THERAPY",
  covariates = c("THERAPY")
)
drawObj <- draws(
  data = dat,
  data_ice = dat_ice,
  vars = vars,
  method = method_condmean(type = "jackknife", covariance = "csh"),
  quiet = TRUE
)
references <- c("DRUG" = "PLACEBO", "PLACEBO" = "PLACEBO")
imputeObj <- impute(drawObj, references)

rbmi_analyse(imputations = imputeObj, vars = vars)
```

rbmi_ancova *Analysis of Covariance*

Description

Performs an analysis of covariance between two groups returning the estimated "treatment effect" (i.e. the contrast between the two treatment groups) and the least square means estimates in each group.

Usage

```
rbmi_ancova(
  data,
  vars,
  visits = NULL,
  weights = c("counterfactual", "equal", "proportional_em", "proportional")
)
```

Arguments

data	A `data.frame` containing the data to be used in the model.
vars	A `vars` object as generated by the set_vars() function from the rbmi package. Only the `group`, `visit`, `outcome` and `covariates` elements are required. See details.
visits	An optional character vector specifying which visits to fit the ancova model at. If `NULL`, a separate ancova model will be fit to the outcomes for each visit (as determined by `unique(data[[vars$visit]])`). See details.
weights	Character, either `"counterfactual"` (default), `"equal"`, `"proportional_em"` or `"proportional"`. Specifies the weighting strategy to be used when calculating the lsmeans. See the weighting section for more details.

Details

The function works as follows:

1. Select the first value from `visits`.
2. Subset the data to only the observations that occurred on this visit.
3. Fit a linear model as `vars$outcome` ~ `vars$group` + `vars$covariates`.
4. Extract the "treatment effect" & least square means for each treatment group.
5. Repeat points 2-3 for all other values in `visits`.

If no value for `visits` is provided then it will be set to `unique(data[[vars$visit]])`.

In order to meet the formatting standards set by `rbmi_analyse()` the results will be collapsed into a single list suffixed by the visit name, e.g.:

```
list(
   var_visit_1 = list(est = ...),
   trt_B_visit_1 = list(est = ...),
   lsm_A_visit_1 = list(est = ...),
   lsm_B_visit_1 = list(est = ...),
   var_visit_2 = list(est = ...),
   trt_B_visit_2 = list(est = ...),
   lsm_A_visit_2 = list(est = ...),
   lsm_B_visit_2 = list(est = ...),
   ...
)
```

Please note that "trt" refers to the treatment effects, and "lsm" refers to the least square mean results. In the above example `vars$group` has two factor levels A and B. The new "var" refers to the model estimated variance of the residuals.

If you want to include interaction terms in your model this can be done by providing them to the `covariates` argument of the set_vars() function from the rbmi package e.g. `set_vars(covariates = c("sex*age"))`.

Value

a list of variance (`var_*`), treatment effect (`trt_*`), and least square mean (`lsm_*`) estimates for each visit, organized as described in Details above.

Note

These functions have the `rbmi_` prefix to distinguish them from the corresponding `rbmi` package functions, from which they were copied from. Additional features here include:

- Support for more than two treatment groups.
- Variance estimates are returned.

See Also

`rbmi_analyse()`

`stats::lm()`

The set_vars() function from the rbmi package

`rbmi_ancova_single` *Implements an Analysis of Covariance (ANCOVA)*

Description

Performance analysis of covariance. See `rbmi_ancova()` for full details.

Usage

```
rbmi_ancova_single(
  data,
  outcome,
  group,
  covariates,
  weights = c("counterfactual", "equal", "proportional_em", "proportional")
)
```

Arguments

`data`	A `data.frame` containing the data to be used in the model.
`outcome`	string, the name of the outcome variable in `data`.
`group`	string, the name of the group variable in `data`.
`covariates`	character vector containing the name of any additional covariates to be included in the model as well as any interaction terms.
`weights`	Character, either `"counterfactual"` (default), `"equal"`, `"proportional_em"` or `"proportional"`. Specifies the weighting strategy to be used when calculating the lsmeans. See the weighting section for more details.

Details

- `group` must be a factor variable with only 2 levels.
- `outcome` must be a continuous numeric variable.

Value

a list containing `var` with variance estimates as well as `trt_*` and `lsm_*` entries. See `rbmi_ancova()` for full details.

See Also

`rbmi_ancova()`

Examples

```
iris2 <- iris[iris$Species %in% c("versicolor", "virginica"), ]
iris2$Species <- factor(iris2$Species)
rbmi_ancova_single(iris2, "Sepal.Length", "Species", c("Petal.Length * Petal.Width"))
```

---

rbmi_mmrm *MMRM Analysis for Imputed Datasets*

Description

Performs an MMRM for two or more groups returning the estimated 'treatment effect' (i.e. the contrast between treatment groups and the control group) and the least square means estimates in each group.

Usage

```
rbmi_mmrm(
  data,
  vars,
  cov_struct = c("us", "toep", "cs", "ar1"),
  visits = NULL,
  weights = c("counterfactual", "equal"),
  ...
)
```

Arguments

`data`	(`data.frame`) containing the data to be used in the model.
`vars`	(`vars`) list as generated by the set_vars() function from the rbmi package. Only the `subjid`, `group`, `visit`, `outcome` and `covariates` elements are required. See details.
`cov_struct`	(`string`) the covariance structure to use. Note that the same covariance structure is assumed for all treatment groups.
`visits`	(`NULL` or `character`) An optional character vector specifying which visits to fit the MMRM at. If `NULL`, the MMRM model will be fit to the whole dataset.
`weights`	(`string`) the weighting strategy to be used when calculating the least square means, either `'counterfactual'` or `'equal'`.
`...`	additional arguments passed to `mmrm::mmrm()`, in particular `method` and `vcov` to control the degrees of freedom and variance-covariance adjustment methods as well as `reml` decide between REML and ML estimation.

Details

The function works as follows:

1. Optionally select the subset of the `data` corresponding to ‘visits.
2. Fit an MMRM as `vars$outcome ~ vars$group + vars$visit + vars$covariates` with the specified covariance structure for visits within subjects.
3. Extract the ’treatment effect’ & least square means for each treatment group vs the control group.

In order to meet the formatting standards set by the analyse() function from the rbmi package, the results will be collapsed into a single list suffixed by the visit name, e.g.:

```
list(
   var_B_visit_1 = list(est = ...),
   trt_B_visit_1 = list(est = ...),
   lsm_A_visit_1 = list(est = ...),
   lsm_B_visit_1 = list(est = ...),
   var_B_visit_2 = list(est = ...),
   trt_B_visit_2 = list(est = ...),
   lsm_A_visit_2 = list(est = ...),
   lsm_B_visit_2 = list(est = ...),
   ...
)
```

Please note that ’trt’ refers to the treatment effects, and ’lsm’ refers to the least square mean results. In the above example `vars$group` has two factor levels A and B. The new ’var’ refers to the model estimated variance of the residuals at the given visit, together with the degrees of freedom (which is treatment group specific).

If you want to include additional interaction terms in your model this can be done by providing them to the `covariates` argument of the set_vars() function from the rbmi package e.g. `set_vars(covariates = c('sex*age'))`.

Value

a list of variance (`var_*`), treatment effect (`trt_*`), and least square mean (`lsm_*`) estimates for each visit, organized as described in Details above.

Note

The `group` and `visit` interaction `group:visit` is not included by default in the model, therefore please add that to `covariates` manually if you want to include it. This will make sense in most cases.

See Also

`rbmi_analyse()`

`mmrm::mmrm()`

The set_vars() function from the rbmi package

`rbmi_mmrm_single_info` *Extract Single Visit Information from a Fitted MMRM for Multiple Imputation Analysis*

Description

Extracts relevant estimates from a given fitted MMRM. See `rbmi_mmrm()` for full details.

Usage

```
rbmi_mmrm_single_info(fit, visit_level, visit, group, weights)
```

Arguments

`fit`	(`mmrm`) the fitted MMRM.
`visit_level`	(`string`) the visit level to extract information for.
`visit`	(`string`) the name of the visit variable.
`group`	(`string`) the name of the group variable.
`weights`	(`string`) the weighting strategy to be used when calculating the least square means, either `'counterfactual'` or `'equal'`.

Value

a list with `trt_*`, `var_*` and `lsm_*` elements. See rbmi_mmrm for full details.

See Also

`rbmi_mmrm()`

`rbmi_pool` *Pool analysis results obtained from the imputed datasets*

Description

Pool analysis results obtained from the imputed datasets

Usage

```
rbmi_pool(
  results,
  conf.level = 0.95,
  alternative = c("two.sided", "less", "greater"),
  type = c("percentile", "normal")
)
```

Arguments

`results`	an analysis object created by analyse().
`conf.level`	confidence level of the returned confidence interval. Must be a single number between 0 and 1. Default is 0.95.
`alternative`	a character string specifying the alternative hypothesis, must be one of `"two.sided"` (default), `"greater"` or `"less"`.
`type`	a character string of either `"percentile"` (default) or `"normal"`. Determines what method should be used to calculate the bootstrap confidence intervals. See details. Only used if `method_condmean(type = "bootstrap")` was specified in the original call to draws().

Details

This has been forked from the `rbmi` package, mainly to support in addition the pooling of variance estimates. See pool() for more details.

Value

A list of class `pool`.

`real_add_overall_facet` *Add Overall Facet*

Description

A function to help add an overall facet to your tables.

Usage

```
real_add_overall_facet(name, label)
```

Arguments

`name`	`(character)` Name/virtual 'value' for the new facet.
`label`	`(character)` Label for the new facet.

Value

Function usable directly as a split function.

Note

Current add_overall_facet is bugged. Can be used directly after it's fixed https://github.com/insightsengineering/rtables/issues

Examples

```
splfun <- make_split_fun(post = list(real_add_overall_facet("Total", "Total")))
```

`remove_col_count` *Removal of Unwanted Column Counts*

Description

Remove the N=xx column headers for specified span_label_var columns - default is 'rrisk_header'.

Usage

```
remove_col_count(obj, span_label_var = "rrisk_header")
```

Arguments

`obj`	`(TableTree)` TableTree object.
`span_label_var`	`(character)` The spanning header text variable value for which column headers will be removed from.

Details

This works for only the lowest level of column splitting (since colcounts is used).

Value

TableTree object with column counts in specified columns removed.

`remove_rows` *Pruning function to remove specific rows of a table regardless of counts*

Description

This function will remove all rows of a table based on the row text provided by the user.

Usage

```
remove_rows(removerowtext = NULL, reg_expr = FALSE)
```

Arguments

removerowtext	(character) Define a text string for which any row with row text will be removed.
reg_expr	(logical) Apply removerowtext as a regular expression (grepl with fixed = TRUE)

Value

Function that can be utilized as pruning function in prune_table.

Examples

```
ADSL <- data.frame(
  USUBJID = c(
    "XXXXX01", "XXXXX02", "XXXXX03", "XXXXX04", "XXXXX05",
    "XXXXX06", "XXXXX07", "XXXXX08", "XXXXX09", "XXXXX10"
  ),
  TRT01P = c(
    "ARMA", "ARMB", "ARMA", "ARMB", "ARMB", "Placebo",
    "Placebo", "Placebo", "ARMA", "ARMB"
  ),
  Category = c(
    "Cat 1", "Cat 2", "Cat 1", "Unknown", "Cat 2",
    "Cat 1", "Unknown", "Cat 1", "Cat 2", "Cat 1"
  ),
  SAFFL = c("N", "N", "N", "N", "N", "N", "N", "N", "N", "N"),
  PKFL = c("N", "N", "N", "N", "N", "N", "N", "N", "N", "N")
)

ADSL <- ADSL |>
  dplyr::mutate(TRT01P = as.factor(TRT01P))

lyt <- basic_table() |>
  split_cols_by("TRT01P") |>
  analyze(
    "Category",
    afun = a_freq_j,
    extra_args = list(.stats = "count_unique_fraction")
  )

result <- build_table(lyt, ADSL)

result

result <- prune_table(result, prune_func = remove_rows(removerowtext = "Unknown"))

result
```

---

resp01_acfun   *Formatted Analysis and Content Summary Function for Response Tables (RESP01)*

Description

This function applies to both `factor` and `logical` columns called `.var` from `df`. Depending on the position in the split, it returns the right formatted results for the RESP01 and related layouts.

Usage

```
resp01_acfun(
  df,
  labelstr = NULL,
  label = NULL,
  .var,
  .spl_context,
  include_comp,
  .alt_df,
  conf_level,
  arm,
  strata,
  formats,
  methods
)
```

Arguments

df	(`data.frame`) data set containing all analysis variables.
labelstr	(`character`) label of the level of the parent split currently being summarized (must be present as second argument in Content Row Functions). See `rtables::summarize_row_groups()` for more information.
label	(`string`) only for logicals, which label to use. (For factors, the labels are the factor levels.)

`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`include_comp`	(`character` or `flag`) whether to include comparative statistic results, either `character` for factors or `flag` for logicals.
`.alt_df`	(`data.frame`) alternative data frame used for denominator calculation.
`conf_level`	(`proportion`) confidence level of the interval.
`arm`	(`string`) column name in the data frame that identifies the treatment arms.
`strata`	(`character` or `NULL`) variable names indicating stratification factors.
`formats`	(`list`) containing formats for `prop_ci`, `comp_stat_ci` and `pval`.
`methods`	(`list`) containing methods for comparative statistics. The element `comp_stat_ci` can be 'rr' (relative risk), 'or_cmh' (odds ratio with CMH estimation and p-value) or 'or_logistic' (odds ratio estimated by conditional or standard logistic regression). The element `pval` can be 'fisher' (Fisher's exact test) or 'chisq' (chi-square test), only used when using unstratified analyses with 'or_logistic'. The element `prop_ci` specifies the method for proportion confidence interval calculation.

Value

The formatted result as `rtables::in_rows()` result.

Examples

```
fake_spl_context <- data.frame(
  cur_col_split_val = I(list(c(ARM = "A: Drug X", count_prop = "count_prop")))
)
dm <- droplevels(subset(DM, SEX %in% c("F", "M")))
resp01_acfun(
  dm,
  .alt_df = dm,
  .var = "COUNTRY",
  .spl_context = fake_spl_context,
  conf_level = 0.9,
  include_comp = c("USA", "CHN"),
  arm = "SEX",
  strata = "RACE",
  methods = list(
    comp_stat_ci = "or_cmh",
    pval = "",
    prop_ci = "wald"
  ),
  formats = list(
    prop_ci = jjcsformat_xx("xx.% - xx.%"),
    comp_stat_ci = jjcsformat_xx("xx.xx (xx.xx - xx.xx)"),
    pval = jjcsformat_pval_fct(0.05)
  )
)
fake_spl_context2 <- data.frame(
  cur_col_split_val = I(list(c(ARM = "Overall", comp_stat_ci = "comp_stat_ci")))
)
resp01_acfun(
  dm,
  .alt_df = dm,
  .var = "COUNTRY",
  .spl_context = fake_spl_context2,
  conf_level = 0.9,
  include_comp = c("USA", "CHN"),
  arm = "SEX",
  strata = "RACE",
  methods = list(
    comp_stat_ci = "or_cmh",
    pval = "",
    prop_ci = "wald"
  ),
  formats = list(
    prop_ci = jjcsformat_xx("xx.% - xx.%"),
    comp_stat_ci = jjcsformat_xx("xx.xx (xx.xx - xx.xx)"),
    pval = jjcsformat_pval_fct(0.05)
  )
)
```

resp01_a_comp_stat_factor

Formatted Analysis Function for Comparative Statistic in Response Tables (RESP01)

Description

This function applies to a `factor` column called `.var` from `df`.

Usage

```
resp01_a_comp_stat_factor(df, .var, include, ...)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.

`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`include`	(`character`) for which factor levels to include the comparison statistic results.
`...`	see `resp01_a_comp_stat_logical()` for additional required arguments.

Value

The formatted result as `rtables::rcell()`.

Examples

```
dm <- droplevels(subset(formatters::DM, SEX %in% c("F", "M")))

resp01_a_comp_stat_factor(
  dm,
  .var = "COUNTRY",
  conf_level = 0.9,
  include = c("USA", "CHN"),
  arm = "SEX",
  strata = "RACE",
  stat = "comp_stat_ci",
  methods = list(comp_stat_ci = "or_cmh"),
  formats = list(
    comp_stat_ci = jjcsformat_xx("xx.xx (xx.xx - xx.xx)"),
    pval = jjcsformat_pval_fct(0.05)
  )
)
```

resp01_a_comp_stat_logical *Formatted Analysis Function for Comparative Statistic in Response Tables (RESP01)*

Description

This function applies to a `logical` column called `.var` from `df`. The response proportion is compared between the treatment arms identified by column `arm`.

Usage

```
resp01_a_comp_stat_logical(
  df,
  .var,
  conf_level,
  include,
  arm,
  strata,
  formats,
  methods,
  stat = c("comp_stat_ci", "pval")
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`conf_level`	(`proportion`) confidence level of the interval.
`include`	(`flag`) whether to include the results for this variable.
`arm`	(`string`) column name in the data frame that identifies the treatment arms.
`strata`	(`character` or `NULL`) variable names indicating stratification factors.
`formats`	(`list`) containing formats for `comp_stat_ci` and `pval`.
`methods`	(`list`) containing methods for comparative statistics. The element `comp_stat_ci` can be 'rr' (relative risk), 'or_cmh' (odds ratio with CMH estimation and p-value) or 'or_logistic' (odds ratio estimated by conditional or standard logistic regression). The element `pval` can be 'fisher' (Fisher's exact test) or 'chisq' (chi-square test), only used when using unstratified analyses with 'or_logistic'.
`stat`	(`string`) the statistic to return, either `comp_stat_ci` or `pval`.

Value

The formatted result as `rtables::rcell()`.

See Also

`resp01_a_comp_stat_factor()` for the `factor` equivalent.

Examples

```
dm <- droplevels(subset(formatters::DM, SEX %in% c("F", "M")))
dm$RESP <- as.logical(sample(c(TRUE, FALSE), size = nrow(DM), replace = TRUE))

resp01_a_comp_stat_logical(
  dm,
  .var = "RESP",
  conf_level = 0.9,
  include = TRUE,
  arm = "SEX",
  strata = "RACE",
  stat = "comp_stat_ci",
  methods = list(comp_stat_ci = "or_cmh"),
  formats = list(
    comp_stat_ci = jjcsformat_xx("xx.xx (xx.xx - xx.xx)"),
    pval = jjcsformat_pval_fct(0.05)
  )
)
```

`resp01_counts_cfun` *Content Row Function for Counts of Subgroups in Response Tables (RESP01)*

Description

Content Row Function for Counts of Subgroups in Response Tables (RESP01)

Usage

```
resp01_counts_cfun(df, labelstr, .spl_context, .alt_df, label_fstr)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`labelstr`	(`character`) label of the level of the parent split currently being summarized (must be present as second argument in Content Row Functions). See `rtables::summarize_row_groups()` for more information.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`.alt_df`	(`data.frame`) alternative data frame used for denominator calculation.
`label_fstr`	(`string`) format string for the label.

Value

The correct `rtables::in_rows()` result.

Examples

```
fake_spl_context <- data.frame(
  cur_col_split_val = I(list(c(ARM = "A: Drug X", count_prop = "count_prop")))
)
resp01_counts_cfun(
  df = DM,
  labelstr = "Blue",
  .spl_context = fake_spl_context,
  .alt_df = DM,
  label_fstr = "Color: %s"
)
```

---

resp01_split_fun_fct *Split Function Factory for the Response Tables (RESP01)*

Description

The main purpose here is to have a column dependent split into either comparative statistic (relative risk or odds ratio with p-value) in the 'Overall' column, and count proportions and corresponding confidence intervals in the other treatment arm columns.

Usage

```
resp01_split_fun_fct(method = c("rr", "or_logistic", "or_cmh"), conf_level)
```

Arguments

method	(string) which method to use for the comparative statistics.
conf_level	(proportion) confidence level of the interval.

Value

A split function for use in the response table RESP01 and similar ones.

See Also

rtables::make_split_fun() describing the requirements for this kind of post-processing function.

Examples

```
split_fun <- resp01_split_fun_fct(
  method = "or_cmh",
  conf_level = 0.95
)
```

response_by_var *Count denom fraction statistic*

Description

Derives the count_denom_fraction statistic (i.e., 'xx /xx (xx.x percent)')
Summarizes the number of unique subjects with a response = 'Y' for a given variable (e.g. TRTEMFL) within each category of another variable (e.g., SEX). Note that the denominator is derived using input df, in order to have these aligned with alt_source_df, it is expected that df includes all subjects.

Usage

```
response_by_var(
  df,
  labelstr = NULL,
  .var,
  .N_col,
  resp_var = NULL,
  id = "USUBJID",
  .format = jjcsformat_count_denom_fraction,
  ...
)
```

Arguments

`df`	(`data.frame`) Name of dataframe being analyzed.
`labelstr`	(`character vector`) Custom label for the variable being analyzed.
`.var`	(`character`) Name of the variable being analyzed. Records with non-missing values will be counted in the denominator.
`.N_col`	(`numeric`) The total for the current column.
`resp_var`	(`character`) Name of variable, for which, records with a value of 'Y' will be counted in the numerator.
`id`	(`character`) Name of column in df which will have patient identifiers
`.format`	(`character`) Format for the count/denominator/fraction output.
`...`	Additional arguments passed to the function.

Details

This is an analysis function for use within `analyze`. Arguments `df`, `.var` will be populated automatically by rtables during the tabulation process.

Value

a `RowsVerticalSection` for use by the internal tabulation machinery of `rtables`

Examples

```
library(dplyr)

ADAE <- data.frame(
  USUBJID = c(
    "XXXXX01", "XXXXX02", "XXXXX03", "XXXXX04", "XXXXX05",
    "XXXXX06", "XXXXX07", "XXXXX08", "XXXXX09", "XXXXX10"
  ),
  SEX_DECODE = c(
    "Female", "Female", "Male", "Female", "Male",
    "Female", "Male", "Female", "Male", "Female"
  ),
  TRT01A = c(
    "ARMA", "ARMB", "ARMA", "ARMB", "ARMB",
    "Placebo", "Placebo", "Placebo", "ARMA", "ARMB"
  ),
  TRTEMFL = c("Y", "Y", "N", "Y", "Y", "Y", "Y", "N", "Y", "Y")
)

ADAE <- ADAE |>
  mutate(
    TRT01A = as.factor(TRT01A),
    SEX_DECODE = as.factor(SEX_DECODE)
  )

lyt <- basic_table() |>
  split_cols_by("TRT01A") |>
  analyze(
    vars = "SEX_DECODE",
    var_labels = "Sex, n/Ns (%)",
    show_labels = "visible",
    afun = response_by_var,
    extra_args = list(resp_var = "TRTEMFL"),
    nested = FALSE
  )

result <- build_table(lyt, ADAE)

result
```

`rm_levels` *Removal of Levels*

Description

Custom function for removing level inside pre step in make_split_fun.

Usage

```
rm_levels(excl)
```

Arguments

`excl`	`(character)` Choose which level(s) to remove

Value

A function implementing pre-processing split behavior (for use in `make_split_fun(pre = )` which removes the levels in `excl` from the data before facets are generated.

`rm_other_facets_fact` *rm_other_facets_fact*

Description

rm_other_facets_fact

Usage

```
rm_other_facets_fact(nm)
```

Arguments

`nm`	character. names of facets to keep. all other facets will be removed

Value

a function suitable for use within the `post` portion make_split_fun

`safe_prune_table` *Safely Prune Table With Empty Table Message If Needed*

Description

Safely Prune Table With Empty Table Message If Needed

Usage

```
safe_prune_table(
  tt,
  prune_func = prune_empty_level,
  stop_depth = NA,
  empty_msg = " - No Data To Display - ",
  spancols = FALSE
)
```

Arguments

`tt`	(`TableTree` or related class) a `TableTree` object representing a populated table.
`prune_func`	(`function`) a function to be called on each subtree which returns `TRUE` if the entire subtree should be removed.
`stop_depth`	(`numeric(1)`) the depth after which subtrees should not be checked for pruning. Defaults to `NA` which indicates pruning should happen at all levels.
`empty_msg`	(`character(1)`) The message to place in the table if no rows were left after pruning
`spancols`	(`logical(1)`) Should `empty_msg` be spanned across the table's columns (`TRUE`) or placed in the rows row label (`FALSE`). Defaults to `FALSE` currently.

Value

`tt` pruned based on the arguments, or, if pruning would remove all rows, a TableTree with the same column structure, and one row containing the empty message spanning all columns.

Examples

```
prfun <- function(tt) TRUE

lyt <- basic_table() |>
  split_cols_by("ARM") |>
  split_cols_by("STRATA1") |>
  split_rows_by("SEX") |>
  analyze("AGE")
```

```
tbl <- build_table(lyt, ex_adsl)

safe_prune_table(tbl, prfun)
```

set_titles *Set Output Titles*

Description

Retrieves titles and footnotes from the list specified in the titles argument and appends them to the TableTree specified in the obj argument.

Usage

```
set_titles(obj, titles)
```

Arguments

obj	(TableTree) The TableTree to which the titles and footnotes will be appended.
titles	(list) The list object containing the titles and footnotes to be appended.

Value

The TableTree object specified in the obj argument, with titles and footnotes appended.

summarize_coxreg_multivar *Layout Generating Function for TEFOS03 and Related Cox Regression Layouts*

Description

Layout Generating Function for TEFOS03 and Related Cox Regression Layouts

Usage

```
summarize_coxreg_multivar(
  lyt,
  var,
  variables,
  control = control_coxreg(),
  formats = list(coef_se = jjcsformat_xx("xx.xx (xx.xx)"), hr_est =
    jjcsformat_xx("xx.xx"), hr_ci = jjcsformat_xx("(xx.xx, xx.xx)"), pval =
    jjcsformat_pval_fct(0))
)
```

Arguments

`lyt`	(`layout`) input layout where analyses will be added to.
`var`	(`string`) any variable from the data, because this is not used.
`variables`	(named `list` of `string`) list of additional analysis variables.
`control`	(`list`) relevant list of control options.
`formats`	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.

Value

`lyt` modified to add the desired cox regression table section.

Examples

```
anl <- tern::tern_ex_adtte |>
  dplyr::mutate(EVENT = 1 - CNSR)

variables <- list(
  time = "AVAL",
  event = "EVENT",
  arm = "ARM",
  covariates = c("SEX", "AGE")
)

basic_table() |>
  summarize_coxreg_multivar(
    var = "STUDYID",
    variables = variables
  ) |>
  build_table(df = anl)
```

`summarize_lsmeans_wide` *Layout Generating Function for LS Means Wide Table Layouts*

Description

Layout Generating Function for LS Means Wide Table Layouts

Usage

```
summarize_lsmeans_wide(
  lyt,
  variables,
  ref_level,
  treatment_levels,
  conf_level,
  pval_sided = "2",
  include_variance = TRUE,
  include_pval = TRUE,
 formats = list(lsmean = jjcsformat_xx("xx.x"), mse = jjcsformat_xx("xx.x"), df =
    jjcsformat_xx("xx."), lsmean_diff = jjcsformat_xx("xx.x"), se =
    jjcsformat_xx("xx.xx"), ci = jjcsformat_xx("(xx.xx, xx.xx)"), pval =
    jjcsformat_pval_fct(0))
)
```

Arguments

`lyt`	(`layout`) empty layout, i.e. result of `rtables::basic_table()`
`variables`	(named `list` of `string`) list of additional analysis variables.
`ref_level`	(`string`) the reference level of the treatment arm variable.
`treatment_levels`	(`character`) the non-reference levels of the treatment arm variable.
`conf_level`	(`proportion`) confidence level of the interval.
`pval_sided`	(`string`) either '2' for two-sided or '1' for 1-sided with greater than control or '-1' for 1-sided with smaller than control alternative hypothesis.
`include_variance`	(`flag`) whether to include the variance statistics (M.S. error and d.f.).
`include_pval`	(`flag`) whether to include the p-value column.
`formats`	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.

Value

Modified layout.

Examples

```
variables <- list(
  response = "FEV1",
  covariates = c("RACE", "SEX"),
  arm = "ARMCD",
  id = "USUBJID",
  visit = "AVISIT"
)
fit <- fit_ancova(
  vars = variables,
  data = mmrm::fev_data,
  conf_level = 0.9,
  weights_emmeans = "equal"
)
anl <- broom::tidy(fit)
basic_table() |>
  summarize_lsmeans_wide(
    variables = variables,
    ref_level = fit$ref_level,
    treatment_levels = fit$treatment_levels,
    pval_sided = "2",
    conf_level = 0.8
  ) |>
  build_table(df = anl)
```

`summarize_mmrm` *Dynamic tabulation of MMRM results with tables*

Description

[Stable]

These functions can be used to produce tables for MMRM results, within tables which are split by arms and visits. This is helpful when higher-level row splits are needed (e.g. splits by parameter or subgroup).

Usage

```
s_summarize_mmrm(
  df,
  .var,
  variables,
  ref_levels,
  .spl_context,
  alternative = c("two.sided", "less", "greater"),
  show_relative = c("reduction", "increase"),
  ...
)
```

```
a_summarize_mmrm(
  df,
  .var,
  .spl_context,
  ...,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`variables`	(named `list` of `string`) list of additional analysis variables.
`ref_levels`	(`list`) with `visit` and `arm` reference levels.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`alternative`	(`string`) whether `two.sided`, or one-sided `less` or `greater` p-value should be displayed.
`show_relative`	(`string`) should the 'reduction' (`control - treatment`, default) or the 'increase' (`treatment - control`) be shown for the relative change from baseline?
`...`	eventually passed to `fit_mmrm_j()` via `h_summarize_mmrm()`.
`.stats`	(`character`) statistics to select for the table.
`.formats`	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.
`.labels`	(named `character`) labels for the statistics (without indent).
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.

Value

- `a_summarize_mmrm()` returns the corresponding list with formatted `rtables::CellValue()`.

Functions

- `s_summarize_mmrm()`: Statistics function which is extracting estimates, not including any results when in the reference visit, and only showing LS mean estimates when in the reference arm and not in reference visit. It uses `s_lsmeans()` for the final processing.
- `a_summarize_mmrm()`: Formatted analysis function which is used as `afun`.

Examples

```
set.seed(123)
longdat <- data.frame(
  ID = rep(DM$ID, 5),
  AVAL = c(
    rep(0, nrow(DM)),
    rnorm(n = nrow(DM) * 4)
  ),
  VISIT = factor(rep(paste0("V", 0:4), each = nrow(DM)))
) |>
  dplyr::inner_join(DM, by = "ID")

basic_table() |>
  split_rows_by("VISIT") |>
  split_cols_by("ARM") |>
  analyze(
    vars = "AVAL",
    afun = a_summarize_mmrm,
    na_str = tern::default_na_str(),
    show_labels = "hidden",
    extra_args = list(
      variables = list(
        covariates = c("AGE"),
        id = "ID",
        arm = "ARM",
        visit = "VISIT"
      ),
      conf_level = 0.9,
      cor_struct = "toeplitz",
      ref_levels = list(VISIT = "V0", ARM = "B: Placebo")
    )
  ) |>
  build_table(longdat) |>
  prune_table(all_zero)
```

`summarize_row_counts` *Layout Creating Function Adding Row Counts*

Description

This is a simple wrapper of `rtables::summarize_row_groups()` and the main additional value is that we can choose whether we want to use the alternative (usually ADSL) data set for the counts (default) or use the original data set.

Usage

```
summarize_row_counts(lyt, label_fstr = "%s", alt_counts = TRUE)
```

Arguments

`lyt`	(`layout`) input layout where analyses will be added to.
`label_fstr`	(`string`) a `sprintf` style format string. It can contain up to one `%s` which takes the current split value and generates the row label.
`alt_counts`	(`flag`) whether row counts should be taken from `alt_counts_df` (`TRUE`) or from `df` (`FALSE`).

Value

A modified layout where the latest row split now has a row group summaries (as created by rtables::summarize_row_groups for the counts).

Examples

```
basic_table() |>
  split_cols_by("ARM") |>
  add_colcounts() |>
  split_rows_by("RACE", split_fun = drop_split_levels) |>
  summarize_row_counts(label_fstr = "RACE value - %s") |>
  analyze("AGE", afun = list_wrap_x(summary), format = "xx.xx") |>
  build_table(DM, alt_counts_df = rbind(DM, DM))
```

`s_ancova_j` *Junco Extended ANCOVA Function*

Description

Extension to tern:::s_ancova, 3 extra statistics are returned:

- `lsmean_se`: Marginal mean and estimated SE in the group.
- `lsmean_ci`: Marginal mean and associated confidence interval in the group.
- `lsmean_diffci`: Difference in mean and associated confidence level in one combined statistic. In addition, the LS mean weights can be specified. In addition, also a NULL .ref_group can be specified, the lsmean_diff related estimates will be returned as NA.

Usage

```
s_ancova_j(
  df,
  .var,
  .df_row,
  variables,
  .ref_group,
  .in_ref_col,
  conf_level,
  interaction_y = FALSE,
  interaction_item = NULL,
  weights_emmeans = "counterfactual"
)
```

Arguments

`df`	(`data.frame`) data set containing all analysis variables.
`.var`	(`string`) single variable name that is passed by `rtables` when requested by a statistics function.
`.df_row`	(`data.frame`) data set that includes all the variables that are called in `.var` and `variables`.
`variables`	(named `list` of `string`) list of additional analysis variables, with expected elements: • `arm` (`string`) group variable, for which the covariate adjusted means of multiple groups will be summarized. Specifically, the first level of `arm` variable is taken as the reference group. • `covariates` (`character`) a vector that can contain single variable names (such as `"X1"`), and/or interaction terms indicated by `"X1 * X2"`.
`.ref_group`	(`data.frame` or `vector`) the data corresponding to the reference group.
`.in_ref_col`	(`flag`) `TRUE` when working with the reference level, `FALSE` otherwise.
`conf_level`	(`proportion`) confidence level of the interval.
`interaction_y`	(`string` or `flag`) a selected item inside of the `interaction_item` variable which will be used to select the specific ANCOVA results. if the interaction is not needed, the default option is `FALSE`.
`interaction_item`	(`string` or `NULL`) name of the variable that should have interactions with arm. if the interaction is not needed, the default option is `NULL`.

`weights_emmeans`
: (string)
argument from `emmeans::emmeans()`, `"counterfactual"` by default.

Value

Returns a named list of 8 statistics (3 extra compared to `tern:::s_ancova()`).

See Also

Other Inclusion of ANCOVA Functions: `a_summarize_ancova_j()`, `a_summarize_aval_chg_diff_j()`

Examples

```
library(dplyr)
library(tern)

df <- iris |> filter(Species == "virginica")
.df_row <- iris
.var <- "Petal.Length"
variables <- list(arm = "Species", covariates = "Sepal.Length * Sepal.Width")
.ref_group <- iris |> filter(Species == "setosa")
conf_level <- 0.95
s_ancova_j(df, .var, .df_row, variables, .ref_group, .in_ref_col = FALSE, conf_level)
```

`s_proportion_factor` *s_function for proportion of factor levels*

Description

A simple statistics function which prepares the numbers with percentages in the required format. The denominator here is from the alternative counts data set in the given row and column split.

If a total row is shown, then here just the total number is shown (without 100%).

Usage

```
s_proportion_factor(
  x,
  .alt_df,
  use_alt_counts = TRUE,
  show_total = c("none", "top", "bottom"),
  total_label = "Total"
)
```

Arguments

`x`	(`factor`) categorical variable we want to analyze.
`.alt_df`	(`data.frame`) alternative data frame used for denominator calculation.
`use_alt_counts`	(`flag`) whether the `.alt_df` should be used for the total, i.e. the denominator. If not, then the number of non-missing values in `x` is used.
`show_total`	(`string`) show the total level optionally on the top or in the bottom of the factor levels.
`total_label`	(`string`) which label to use for the optional total level.

Value

The `rtables::in_rows()` result with the proportion statistics.

See Also

`s_proportion_logical()` for tabulating logical `x`.

`s_proportion_logical` *s_function for proportion of* `TRUE` *in logical vector*

Description

A simple statistics function which prepares the numbers with percentages in the required format. The denominator here is from the alternative counts data set in the given row and column split.

Usage

```
s_proportion_logical(x, label = "Responders", .alt_df)
```

Arguments

`x`	(`logical`) binary variable we want to analyze.
`label`	(`string`) label to use.
`.alt_df`	(`data.frame`) alternative data frame used for denominator calculation.

Value

The `rtables::in_rows()` result with the proportion statistics.

See Also

s_proportion_factor() for tabulating factor x.

---

tabulate_lsmeans *Tabulation of Least Square Means Results*

Description

[Stable]

These functions can be used to produce tables from LS means, e.g. from fit_mmrm_j() or fit_ancova().

Usage

```
## S3 method for class 'tern_model'
tidy(x, ...)

s_lsmeans(
  df,
  .in_ref_col,
  alternative = c("two.sided", "less", "greater"),
  show_relative = c("reduction", "increase")
)

a_lsmeans(
  df,
  ref_path,
  .spl_context,
  ...,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL
)
```

Arguments

x	(numeric) vector of numbers we want to analyze.
...	additional arguments for the lower level functions.
df	(data.frame) data set containing all analysis variables.
.in_ref_col	(logical) TRUE when working with the reference level, FALSE otherwise.
alternative	(string) whether two.sided, or one-sided less or greater p-value should be displayed.

`show_relative`	(`string`) should the 'reduction' (`control - treatment`, default) or the 'increase' (`treatment - control`) be shown for the relative change from baseline?
`ref_path`	(`character`) global reference group specification, see `get_ref_info()`.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`.stats`	(`character`) statistics to select for the table.
`.formats`	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.
`.labels`	(named `character`) labels for the statistics (without indent).
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.

Value

- For `s_lsmeans`, a list containing the same statistics returned by tern.mmrm::s_mmrm_lsmeans, with the additional `diff_mean_est_ci` three-dimensional statistic.
- For `a_lsmeans`, a `VertalRowsSection` as returned by rtables::in_rows.

Functions

- `tidy(tern_model)`: Helper method (for `broom::tidy()`) to prepare a `data.frame` from an `tern_model` object containing the least-squares means and contrasts.
- `s_lsmeans()`: Statistics function which is extracting estimates from a tidied least-squares means data frame.
- `a_lsmeans()`: Formatted Analysis function to be used as `afun`

Note

These functions have been forked from the `tern.mmrm` package. Additional features are:

- Additional `ref_path` argument for tern.mmrm::summarize_lsmeans().
- The function is more general in that it also works for LS means results from ANCOVA
- Additional statistic `diff_mean_est_ci` is returned
- P-value sidedness can be chosen

Examples

```
result <- fit_mmrm_j(
  vars = list(
    response = "FEV1",
    covariates = c("RACE", "SEX"),
    id = "USUBJID",
    arm = "ARMCD",
    visit = "AVISIT"
  ),
  data = mmrm::fev_data,
  cor_struct = "unstructured",
  weights_emmeans = "equal"
)

df <- broom::tidy(result)

s_lsmeans(df[8, ], .in_ref_col = FALSE)
s_lsmeans(df[8, ], .in_ref_col = FALSE, alternative = "greater", show_relative = "increase")

dat_adsl <- mmrm::fev_data |>
  dplyr::select(USUBJID, ARMCD) |>
  unique()

basic_table() |>
  split_cols_by("ARMCD") |>
  add_colcounts() |>
  split_rows_by("AVISIT") |>
  analyze(
    "AVISIT",
    afun = a_lsmeans,
    show_labels = "hidden",
    na_str = tern::default_na_str(),
    extra_args = list(
      .stats = c(
        "n",
        "adj_mean_se",
        "adj_mean_ci",
        "diff_mean_se",
        "diff_mean_ci"
      ),
      .labels = c(
        adj_mean_se = "Adj. LS Mean (Std. Error)",
        adj_mean_ci = "95% CI",
        diff_mean_ci = "95% CI"
      ),
      .formats = c(adj_mean_se = jjcsformat_xx("xx.x (xx.xx)")),
      alternative = "greater",
      ref_path = c("ARMCD", result$ref_level)
    )
  ) |>
  build_table(
    df = broom::tidy(result),
    alt_counts_df = dat_adsl
  )
```

`tabulate_rbmi` *Tabulation of RBMI Results*

Description

[Stable]

These functions can be used to produce tables from RBMI.

Usage

```
h_tidy_pool(x, visit_name, group_names)

s_rbmi_lsmeans(df, .in_ref_col, show_relative = c("reduction", "increase"))

a_rbmi_lsmeans(
  df,
  ref_path,
  .spl_context,
  ...,
  .stats = NULL,
  .formats = NULL,
  .labels = NULL,
  .indent_mods = NULL
)
```

Arguments

`x`	(`list`) is a list of pooled object from `rbmi` analysis results. This list includes analysis results, confidence level, hypothesis testing type.
`visit_name`	(`string`) single visit level.
`group_names`	(`character`) group levels.
`df`	(`data.frame`) input with LS means results.
`.in_ref_col`	(`flag`) whether reference column is specified.
`show_relative`	(`string`) 'reduction' if (`control - treatment`, default) or 'increase' (`treatment - control`) of relative change from baseline?

`ref_path`	(`character`) global reference group specification, see `get_ref_info()`.
`.spl_context`	(`data.frame`) gives information about ancestor split states that is passed by `rtables`.
`...`	additional arguments for the lower level functions.
`.stats`	(`character`) statistics to select for the table.
`.formats`	(named `character` or `list`) formats for the statistics. See Details in `analyze_vars` for more information on the `'auto'` setting.
`.labels`	(named `character`) labels for the statistics (without indent).
`.indent_mods`	(named `integer`) indent modifiers for the labels. Defaults to 0, which corresponds to the unmodified default behavior. Can be negative.

Value

- `h_tidy_pool()` returns a `data.frame` with results of pooled analysis for a single visit.
- `s_rbmi_lsmeans()` returns a list of statistics extracted from a tidied LS means data frame.

Functions

- `h_tidy_pool()`: Helper function to produce data frame with results of pool for a single visit.
- `s_rbmi_lsmeans()`: Statistics function which is extracting estimates from a tidied RBMI results data frame.
- `a_rbmi_lsmeans()`: Formatted Analysis function which is used as `afun`.

Note

These functions have been forked from `tern.rbmi`. Additional features are:

- Additional `ref_path` argument.
- Extraction of variance statistics in the `tidy()` method.
- Adapted to `rbmi` forked functions update with more than two treatment groups.

theme_docx_default_j *Obtain the default theme for the docx*

Description

[Experimental]

This function is based on `rtables.officer::theme_docx_default()`. See notes to understand why this is experimental.

Usage

```
theme_docx_default_j(
  font = "Times New Roman",
  font_size = 9L,
  cell_margins = c(0, 0, 0, 0),
  bold = c("header", "content_rows", "label_rows", "top_left"),
  bold_manual = NULL,
  border = flextable::fp_border_default(width = 0.75, color = "black")
)
```

Arguments

`font`	(`string`) font. Defaults to "Times New Roman".
`font_size`	(`integer(1)`) font size. Defaults to 9.
`cell_margins`	(`numeric(1) or numeric(4)`) a numeric or a vector of four numbers indicating c("left", "right", "top", "bottom"). It defaults to 0mm in Word pt to all 4 margins.
`bold`	(`character`) parts of the table text that should be in bold. Can be any combination of c("header", "content_rows", "label_rows", "top_left"). The first one renders all column names bold (not topleft content). The second and third option use formatters::make_row_df() to render content or/and label rows as bold.
`bold_manual`	(`named list or NULL`) list of index lists. See example for needed structure. Accepted groupings/names are c("header", "body").
`border`	(`fp_border`) border to use. Defaults to width = 0.75 and color = "black"

Value

a function that applies the given theme to a flextable.

Note

This function has been tested for common use cases but may not work or have unexpected or undesired behavior in corner cases. As such it is not considered fully production ready and is being made available for further testing and early adoption. Please report any issues you encounter to the developers. This function may be removed from junco in the future if the functionality is merged into `rtables.officer`.

`tt_to_flextable_j` *Convert a VTableTree or a listing_df object to a flextable*

Description

[Experimental]

This function is based on `rtables.officer::tt_to_flextable()`. See notes to understand why this is experimental.

Usage

```
tt_to_flextable_j(
  tt,
  tblid,
 theme = theme_docx_default_j(font = "Times New Roman", font_size = 9L, bold = NULL),
  border = flextable::fp_border_default(width = 0.75, color = "black"),
  indent_size = NULL,
  titles_as_header = TRUE,
  bold_titles = TRUE,
  integrate_footers = TRUE,
  counts_in_newline = FALSE,
  paginate = FALSE,
  fontspec = formatters::font_spec("Times", 9L, 1.2),
  lpp = NULL,
  cpp = NULL,
  ...,
  colwidths = NULL,
  tf_wrap = !is.null(cpp),
  max_width = cpp,
  total_page_height = 10,
  total_page_width = my_pg_width_by_orient(orientation),
  autofit_to_page = TRUE,
  orientation = "portrait",
  nosplitin = character(),
  string_map = junco::default_str_map,
  markup_df_docx = dps_markup_df_docx,
  reduce_first_col_indentation = FALSE,
  tlgtype = (utils::getFromNamespace("tlg_type", "junco"))(tt),
  col_gap = ifelse(tlgtype == "Listing", 0.5, 3),
  pagenum = ifelse(tlgtype == "Listing", TRUE, FALSE),
  round_type = formatters::obj_round_type(tt),
  alignments = list(),
  border_mat = make_header_bordmat(obj = tt)
)
```

Arguments

`tt`	a VTableTree or a listing_df object
`tblid`	Character. Output ID to be displayed in the Title and last line of footer.
`theme`	(optional) a function factory. See theme_docx_default_j() or rtables.officer::theme_docx_default() for more details.
`border`	(optional) an `fp_border` object.
`indent_size`	(optional) Numeric. Not used and set to 9 points internally.
`titles_as_header`	(optional) Default = TRUE.
`bold_titles`	(optional) Default = TRUE.
`integrate_footers`	(optional) Default = TRUE.
`counts_in_newline`	(optional) Default = FALSE.
`paginate`	(optional) Default = FALSE.
`fontspec`	(optional) a font_spec object.
`lpp`	(optional) Default = NULL. Not used.
`cpp`	(optional) Default = NULL. Not used.
`...`	other arguments.
`colwidths`	(optional) Default = NULL.
`tf_wrap`	(optional) Default = FALSE. Not used.
`max_width`	(optional) Default = NULL. Not used.
`total_page_height`	(optional) Default = 10. Not used.
`total_page_width`	(optional). No need to be specified by end user. Set to 6.38 ("portrait") or 8.88 ("landscape").
`autofit_to_page`	(optional) Default = TRUE. Not used and set to FALSE internally.
`orientation`	(optional) Default = "portrait". One of: "portrait", "landscape".
`nosplitin`	(optional) Default = character(). Named list.
`string_map`	(optional) Default = default_str_map.
`markup_df_docx`	(optional) Default = dps_markup_df_docx.
`reduce_first_col_indentation`	(optional) Default = FALSE.

`tlgtype`	(optional). No need to be specified by end user.
`col_gap`	(optional). Default = 3 (Tables) or 0.5 (Listings).
`pagenum`	(optional). Default = FALSE (Tables) or TRUE (Listings).
`round_type`	(`"iec"` or `"sas"`) the type of rounding to perform. iec, the default, performs rounding compliant with IEC 60559, while sas performs nearest-value rounding consistent with rounding within SAS. See `[formatters::format_value()]` for more details.
`alignments`	(`list`) List of named lists. Vectorized. (Default = `list()`) Used to specify individual column or cell alignments. Each named list contains `row`, `col`, and `value`.
`border_mat`	(`matrix`) A `m x k` matrix where m is the number of columns of `tt` and k is the number of lines the header takes up. See tidytlg::add_bottom_borders for what the matrix should contain. Users should only specify this when the default behavior does not meet their needs.

Value

a flextable object.

Note

This function has been tested for common use cases but may not work or have unexpected or undesired behavior in corner cases. As such it is not considered fully production ready and is being made available for further testing and early adoption. Please report any issues you encounter to the developers. This function may be removed from junco in the future if the functionality is merged into `rtables.officer`.

`tt_to_tbldf` *Create TableTree as DataFrame via gentlg*

Description

Create TableTree as DataFrame via gentlg

Usage

```
tt_to_tbldf(
  tt,
  fontspec = font_spec("Times", 9L, 1),
  string_map = default_str_map,
  markup_df = dps_markup_df,
  round_type = obj_round_type(tt),
  validate = TRUE
)
```

Arguments

`tt`	(`TableTree`) TableTree object to convert to a data frame
`fontspec`	(`font_spec`) Font specification object
`string_map`	(`list`) Unicode mapping for special characters
`markup_df`	(`data.frame`) Data frame containing markup information
`round_type`	(`character(1)`) the type of rounding to perform. See `formatters::format_value()` for more details.
`validate`	logical(1). Whether to validate the table structure using `rtables::validate_table_struct()`. Defaults to `TRUE`. If `FALSE`, a message will be displayed instead of stopping with an error when validation fails.

Value

`tt` represented as a `tbl` data.frame suitable for passing to tidytlg::gentlg via the huxme argument.

`tt_to_tlgrtf` *TableTree to .rtf Conversion*

Description

A function to convert TableTree to .rtf

Usage

```
tt_to_tlgrtf(
  tt,
  file = NULL,
  orientation = c("portrait", "landscape"),
  colwidths = def_colwidths(tt, fontspec, col_gap = col_gap, label_width_ins =
    label_width_ins, type = tlgtype),
  label_width_ins = 2,
  watermark = NULL,
  pagenum = ifelse(tlgtype == "Listing", TRUE, FALSE),
  fontspec = font_spec("Times", 9L, 1.2),
  pg_width = pg_width_by_orient(orientation == "landscape"),
  margins = c(0, 0, 0, 0),
  paginate = tlg_type(tt) == "Table",
  col_gap = ifelse(tlgtype == "Listing", 0.5, 3),
  nosplitin = list(row = character(), col = character()),
  verbose = FALSE,
  tlgtype = tlg_type(tt),
  string_map = default_str_map,
  markup_df = dps_markup_df,
  combined_rtf = FALSE,
  one_table = TRUE,
  border_mat = make_header_bordmat(obj = tt),
  round_type = obj_round_type(tt),
  alignments = list(),
  validate = TRUE,
  ...
)
```

Arguments

`tt`	(`TableTree`) TableTree object to convert to RTF
`file`	(`character(1)`) File to create, including path, but excluding .rtf extension.
`orientation`	(`character`) Orientation of the output ("portrait" or "landscape")
`colwidths`	(`numeric` vector) Column widths for the table
`label_width_ins`	(`numeric`) Label width in inches
`watermark`	(optional) String containing the desired watermark for RTF outputs. Vectorized.
`pagenum`	(`logical`) Whether to add page numbers to the output. Only applicable to listings (i.e. it is ignored for tables and figures).
`fontspec`	(`font_spec`) Font specification object
`pg_width`	(`numeric`) Page width in inches
`margins`	(`numeric` vector) Margins in inches (top, right, bottom, left)
`paginate`	(`logical`) Whether to paginate the output
`col_gap`	(`numeric`) Column gap in spaces
`nosplitin`	(`list`) list(row=, col=). Path elements whose children should not be paginated within if it can be avoided. e.g., list(col="TRT01A") means don't split within treatment arms unless all the associated columns don't fit on a single page.
`verbose`	(`logical`) Whether to print verbose output

`tlgtype`	(`character`) Type of the output (Table, Listing, or Figure)
`string_map`	(`data.frame`) Unicode mapping for special characters
`markup_df`	(`data.frame`) Data frame containing markup information
`combined_rtf`	(`logical(1)`) In the case where the result is broken up into multiple parts due to width, should a combined rtf file also be created. Defaults to `FALSE`.
`one_table`	(`logical(1)`) If `tt` is a (non-`MatrixPrintForm`) list, should the parts be added to the rtf within a single table (`TRUE`, the default) or as separate tables. End users will not generally need to set this.
`border_mat`	(`matrix`) A `m x k` matrix where m is the number of columns of `tt` and k is the number of lines the header takes up. See tidytlg::add_bottom_borders for what the matrix should contain. Users should only specify this when the default behavior does not meet their needs.
`round_type`	(`character(1)`) the type of rounding to perform. See `formatters::format_value()` for more details.
`alignments`	(`list`) List of named lists. Vectorized. (Default = `list()`) Used to specify individual column or cell alignments. Each named list contains `row`, `col`, and `value`, which are passed to `huxtable::set_align()` to set the alignments.
`validate`	logical(1). Whether to validate the table structure using `rtables::validate_table_struct()`. Defaults to `TRUE`. If `FALSE`, a message will be displayed when validation fails.
`...`	Additional arguments passed to gentlg

Details

This function aids in converting the rtables TableTree into the desired .rtf file.

Value

If `file` is non-NULL, this is called for the side-effect of writing one or more RTF files. Otherwise, returns a list of `huxtable` objects.

Note

`file` should always include path. Path will be extracted and passed separately to `gentlg`.

When `one_table` is `FALSE`, only the width of the row label pseudocolumn can be directly controlled due to a limitation in `tidytlg::gentlg`. The proportion of the full page that the first value in colwidths would take up is preserved and all other columns equally split the remaining available width. This will cause, e.g., the elements within the allparts rtf generated when `combined_rtf` is `TRUE` to differ visually from the content of the individual part rtfs.

Index