

Package: hvtiRutilities (via r-universe)

October 23, 2026

Type Package

Title A Collection of Utilities for Use by HVTI CORR Group at the Cleveland Clinic

Version 1.5.10

Date 2026-10-22

License GPL-3

Encoding UTF-8

URL <https://github.com/ehrlinger/hvtiRutilities>,
<https://ehrlinger.github.io/hvtiRutilities/>

BugReports <https://github.com/ehrlinger/hvtiRutilities/issues>

Description R utilities for the clinical investigations statistics group within The Heart & Vascular Institute at the Cleveland Clinic.

Depends R (>= 4.4.0)

Imports digest, dplyr, haven, jsonlite, labelled, readxl, tools, utils, uuid, withr, yaml

Suggests arrow, callr, e1071, knitr, mice, quarto, randomForestSRC, renv, rmarkdown, testthat (>= 3.0.0), tidyverse, tibble, pkgload, varPro, writexl

VignetteBuilder quarto

SystemRequirements Quarto command line tool
(<<https://github.com/quarto-dev/quarto-cli>>)

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

Config/pak/sysreqs make libicu-dev libx11-dev zlib1g-dev

Repository <https://ehrlinger.r-universe.dev>

Date/Publication 2026-10-23 01:32:08 UTC

RemoteUrl <https://github.com/ehrlinger/hvtiRutilities>

RemoteRef HEAD

RemoteSha 550c42b2760c876ba5b3ac0fdb26c867b9130b4a

Contents

hvtiRutilities-package	4
add_abbreviation	4
add_labels	5
adopt_data_update	6
apply_label_overrides	7
apply_value_labels	8
assert_cohort	10
built_manifest	11
built_path	12
cache_fit	13
capture_provenance	15
check_data_updates	16
clean_labels	17
cohort_counts	18
compare_datasets	19
compare_parity	20
concept_map	21
concept_of	22
covariate_audit	24
covariates_to_numeric	25
data_dictionary	26
dataset_schema	27
followup_check	28
generate_survival_data	30
get_label	32
get_labels	32
guard_snapshot	33
hvti_non_prefixes	34
hvti_prefix_folds	35
hvti_taxonomy	36
hvtiRutilities.news	38
imputed_levels	38
job_census	39
job_files	40
label_map	42
level_map	44
parity_headline	46
parity_tolerance	46
POOL_AFFIXES	47
pool_collinear_pairs	48
preflight_report	49
proc_contents	50
proc_freq	51
proc_means	53
proc_univariate	55
provenance_artifact	57

provenance_data	58
provenance_path	59
prune_to_one_form	60
publish_provenance	61
r_data_types	62
r_dir_impurities	64
read_built	65
read_clinical_data	66
record_provenance	68
register_data	69
review_built	72
review_data_update	73
sample_data	74
sas_macro_defs	75
sas_macro_signature	76
sas_path	76
sas_triage	77
sas_variable_block	79
selection_crowding	80
strip_level_prefix	81
study_abbreviations	82
study_checklist	84
study_checkpoint	84
study_checkpoint_push	86
study_close	87
study_config	89
study_declare	90
study_dir	91
study_need	92
study_root	93
study_setup	94
study_status	96
study_update_identity	98
type_conversion_report	99
update_manifest	100
verify_manifest	102
write_collision_report	104
write_macro_manifest	106

hvtiRutilities-package

hvtiRutilities: Utilities to work with SAS data in R.

Description

hvtirutilities is a utility package for using SAS dataset in R at the Cleveland Clinic Heart, Vascular and Thoracic Institute (HVTI) Clinical Outcome Registries and Research (CORR) department.

add_abbreviation

Add a phrase to a study's abbreviation list

Description

Writes one entry to the abbreviations: mapping in the study's `_study.yml`, replacing any entry for the same phrase, compared ignoring case. `abbreviation = NULL` writes a removal: the study then drops that phrase from the group default list.

Usage

```
add_abbreviation(phrase, abbreviation, start = getwd())
```

Arguments

phrase	The phrase, as it appears in labels.
abbreviation	Its abbreviation, or NULL to remove a default.
start	A directory inside the study; the study is found as study_config finds it.

Details

The entry is checked against the whole merged list, as [study_abbreviations](#) would check it, before anything is written; a refused entry leaves the file as it was. The file is rewritten whole, as [register_data](#) rewrites it, so comments in `_study.yml` are not kept.

Value

The study's abbreviation list after the change, invisibly, as a named list.

See Also

[study_abbreviations](#).

Examples

```
root <- file.path(tempdir(), "add-abbrev-example")
study_setup(root, "Abbreviation example", 1L, renv = FALSE)
add_abbreviation("Surgical procedure", "SP", start = root)
add_abbreviation("Ejection fraction", NULL, start = root)
unlink(root, recursive = TRUE)
```

add_labels

Add or update labels in a label map

Description

Registers new labels in an existing label map, or applies labels directly to a data frame's variable attributes. This is the recommended way to label derived variables (e.g., ratios, binned groups, computed indices) that were not present in the original imported dataset.

When `label_map_df` is a label map (a data frame with key and label columns), the map is updated and returned. When `label_map_df` is a regular data frame (any data frame without the label map structure), labels are applied directly to the data using `labelled::var_label()`, which is the preferred approach because labels travel with the data through `dplyr` operations.

Usage

```
add_labels(label_map_df, new_labels)
```

Arguments

<code>label_map_df</code>	A data frame with key and label columns (as returned by label_map), or any data frame to which labels should be applied directly.
<code>new_labels</code>	A named character vector where names are variable names and values are descriptive labels.

Value

When given a label map: the updated label map data frame. When given a data frame: the data frame with labels applied via `labelled::var_label()`.

See Also

[label_map](#) to extract a label map from data, [apply_label_overrides](#) for bulk overrides from YAML.

Examples

```
# --- Method 1: Update a label map (for reporting) ---
dta <- generate_survival_data(n = 50, seed = 42)
lmap <- label_map(dta)

# Add labels for derived variables
lmap <- add_labels(lmap, c(
  age_group = "Age Group (<40, 40-60, >60)",
  bsa_ratio = "BSA Ratio",
  risk_score = "Composite Risk Score"
))
tail(lmap, 4)

# --- Method 2: Label a data frame directly (preferred) ---
dta$age_group <- cut(dta$age, breaks = c(0, 40, 60, Inf),
  labels = c("<40", "40-60", ">60"))
dta <- add_labels(dta, c(age_group = "Age Group (<40, 40-60, >60)"))
labelled::var_label(dta$age_group)
```

adopt_data_update	<i>Adopt one published dataset release</i>
-------------------	--

Description

Deprecated. Catalog releases are retired; use [update_manifest](#).

Repeats the candidate review and replaces `_study.yml` and `manifest.yaml` as one recoverable pair. The old dated release and its cache files remain on disk. Adoption does not make a Git commit.

The candidate must be named by its exact release ID. The value "latest" is never accepted, because the reviewed release and the adopted release must be the same object.

Usage

```
adopt_data_update(cfg = study_config(), dataset = "study", release_id)
```

Arguments

<code>cfg</code>	List. A study manifest from study_config .
<code>dataset</code>	Character(1). Logical dataset name. Defaults to "study". "built" is a second name for "study".
<code>release_id</code>	Character(1). Exact, newer, published candidate release ID.

Value

The updated [study_status](#) object, returned visibly.

See Also

[check_data_updates](#), [review_data_update](#), [register_data](#)

Examples

```
root <- file.path(tempdir(), "adopt-data-update-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
                 file.path(root, "00_datasets", "built.csv"),
                 row.names = FALSE)
invisible(register_data(root, built = "built.csv"))
cfg <- study_config(root)

# Deprecated: it warns and names update_manifest() instead, then stops,
# because this study is not pinned to a catalog release.
try(adopt_data_update(cfg, release_id = "surgery_cohort-20260921-r1"))
unlink(root, recursive = TRUE)
```

apply_label_overrides *Apply label overrides from a YAML file*

Description

Reads label overrides from a YAML file and applies them to a label map **or** directly to a data frame. This allows study-specific label replacements (e.g., abbreviations, corrections) to be configured externally rather than hard-coded in analysis scripts.

The YAML file should contain a simple mapping of variable names to labels. If the file does not exist, the input is returned unchanged — making it safe to call unconditionally in shared code.

When data is a label map (a data frame with key and label columns), the overrides are applied to the map. When data is any other data frame, labels are applied directly to the data via [add_labels](#), which is the preferred data-first workflow.

Usage

```
apply_label_overrides(data, overrides_file = "labels_overrides.yml")
```

Arguments

data	A data frame: either a label map (with key and label columns, as returned by label_map), or any data frame whose columns should be labeled directly.
overrides_file	Path to a YAML file containing label overrides. Defaults to "labels_overrides.yml" in the current working directory.

Details

The YAML file format is a simple mapping of variable names to labels:

```
age_binned: "Age Group"
bsa_ratio: "BSA Ratio"
cavv_area: "Common AVV Area"
```

This design keeps study-specific label customizations in configuration rather than code. Each study gets its own `labels_overrides.yml` alongside its `config.yml`, and shared helper functions never contain hard-coded replacements.

Value

When given a label map: the updated label map data frame. When given a data frame: the data frame with labels applied via `labelled::var_label()`. In both cases, variables not mentioned in the YAML file are left unchanged.

See Also

[add_labels](#) for programmatic label updates, [label_map](#) for extracting labels from data.

Examples

```
# Create a temporary YAML overrides file
tmp <- tempfile(fileext = ".yaml")
writeLines(c(
  "age: 'Patient Age (years)'",
  "bsa_ratio: 'Body Surface Area Ratio'"
), tmp)

# --- On a label map ---
dta <- generate_survival_data(n = 50, seed = 42)
lmap <- label_map(dta)
lmap <- apply_label_overrides(lmap, overrides_file = tmp)
lmap[lmap$key == "age", ]

# --- Directly on data (preferred) ---
dta <- apply_label_overrides(dta, overrides_file = tmp)
labelled::var_label(dta$age)

unlink(tmp)
```

<code>apply_value_labels</code>	<i>Declare value labels for coded variables from a YAML file</i>
---------------------------------	--

Description

Reads a study's `value_labels.yml` and attaches the declared code-to-text mappings to the matching columns as value labels, via `labelled::val_labels()`. It declares; it does not convert. Pass the result to [r_data_types](#) with `use_value_labels = TRUE` to turn the declarations into factors with text levels.

A `.sas7bdat` file stores a format's *name* (YESNOF.); the code-to-text mapping lives in a `.sas7bcat` catalogue. The study corpus has no catalogues, so this file is not a supplement to one - it is the only place the mapping can come from. Because the declaration is written into the same slot a catalogue would have filled, everything downstream reads it without a second code path.

The function is safe to call unconditionally: a study with no `value_labels.yml` gets its data back unchanged.

Usage

```
apply_value_labels(data, value_labels_file = "value_labels.yml")
```

Arguments

data A data frame whose coded columns should be declared.

value_labels_file Path to a YAML file of value-label declarations. Defaults to "value_labels.yml" in the current working directory. A path that does not exist is not an error.

Details

The file maps each variable to its codes, and each code to the text that code means:

```
disp:
  1: Home
  2: Rehab
  3: SNF
approach:
  1: Ascending aorta only
  2: Ascending aorta plus arch
```

This is the home for enumerated level definitions, and it is deliberately **not** the display label. Level definitions crammed into a variable label are what forced them to be reconstructed by hand in every table, and a label carrying eight mutually exclusive options cannot survive `label_max` (see [label_map](#)). Declaring them here keeps the two jobs apart: the converter reads this file, and a table renderer reads the label.

Two rules worth knowing before relying on the result:

- **Value labels already on the column win.** A column that arrives labelled - from a catalogue, or from an earlier call - is left alone and the variable is named in a warning, so a declaration can never silently overwrite a mapping read from the source.
- **A variable named in the file but absent from the data is reported.** A typo that quietly declares nothing is the failure this file is least able to notice.

The ordered key is reserved for the ordinal declaration and is **refused** rather than partly honoured. Ordinality is blocked on a statistical decision; reserving the key now keeps one home for both declarations without pre-empting it.

Value

data, with value labels attached to the declared columns via `labelled::val_labels()`. Columns not named in the file, and columns that already carry value labels, are returned unchanged.

See Also

[r_data_types](#) for converting declarations to factors, [apply_label_overrides](#) for the same pattern one level up - declaring *variable* labels rather than value labels, [read_clinical_data](#) for the `catalog_file` argument that would supply these mappings if a catalogue existed.

Examples

```
tmp <- tempfile(fileext = ".yaml")
writeLines(c("disp:", " 1: Home", " 2: Rehab", " 3: SNF"), tmp)

dta <- data.frame(dis = c(1, 2, 1, 3))
dta <- apply_value_labels(dta, tmp)
labelled::val_labels(dta$dis)

# Declare, then convert
r_data_types(dta, use_value_labels = TRUE)$dis

unlink(tmp)
```

assert_cohort

Assert the cohort matches a job-level expectation

Description

Compares [cohort_counts](#) against explicitly supplied expected counts and errors on any disagreement. Call it before any analysis that would otherwise run happily on an unreconciled cohort.

Usage

```
assert_cohort(d, expected, event, time)
```

Arguments

d	A data frame.
expected	List containing nonnegative integer n, n_events and n_censored.
event	Character scalar naming the binary event column.
time	Character scalar naming the time column.

Value

invisible(TRUE) on success; otherwise an error.

See Also

[cohort_counts](#)

Examples

```
d <- data.frame(dead = c(1, 1, 0, 0, 0), iv_dead = 1:5)
expected <- list(n = 5L, n_events = 2L, n_censored = 3L)
assert_cohort(d, expected, event = "dead", time = "iv_dead")
```

built_manifest	<i>Record the state of the built dataset</i>
----------------	--

Description

Returns a one-row data frame identifying the built dataset by name, size, modification time and SHA-256. This is the record that lets a later reader tell whether two results were produced from the same data.

Usage

```
built_manifest(cfg = study_config(), dataset = "study")
```

Arguments

cfg	List. A study manifest from study_config .
dataset	Character(1). Logical dataset name. Defaults to "study". "built" is a second name for "study".

Value

A one-row data frame with columns file, size_bytes, mtime and sha256.

See Also

[built_path](#), [record_provenance](#)

Examples

```
root <- file.path(tempdir(), "built-manifest-example")
dir.create(file.path(root, "datasets"), recursive = TRUE,
           showWarnings = FALSE)
yaml::write_yaml(
  list(study = "Example", built = "example.csv"),
  file.path(root, "_study.yml")
)
write.csv(data.frame(dead = c(1, 0, 0), iv_dead = 1:3),
          file.path(root, "datasets", "example.csv"), row.names = FALSE)
built_manifest(study_config(root))
unlink(root, recursive = TRUE)
```

built_path	<i>Path to the study's built dataset</i>
------------	--

Description

Resolves the registered filename beneath the study's logical datasets directory. The path is not checked for existence.

Usage

```
built_path(cfg = study_config(), dataset = "study")
```

Arguments

cfg	List. A study manifest from study_config .
dataset	Character(1). Logical dataset name. Defaults to "study". "built" is a second name for "study".

Value

Character(1). The path to the built dataset.

See Also

[built_manifest](#), [read_built](#)

Examples

```
root <- file.path(tempdir(), "built-path-example")
dir.create(file.path(root, "datasets"), recursive = TRUE,
  showWarnings = FALSE)
yaml::write_yaml(
  list(study = "Example", built = "example.sas7bdat"),
  file.path(root, "_study.yml")
)
built_path(study_config(root))
unlink(root, recursive = TRUE)
```

cache_fit

*Cache an expensive computation, strictly***Description**

Runs an expensive computation (a random forest, a varPro fit, an imputation, a hazard model, a partial dependence grid) once, saves the result, and on later calls loads it instead of recomputing. The saved result is used only if nothing it was built from has changed: the code, every outside value the code reads, the versions of the packages its functions come from, and the seed. If anything changed, `cache_fit()` stops and lists what changed. It never returns a stale result.

Usage

```
cache_fit(
  name,
  code,
  seed = NULL,
  dir = study_dir("estimates"),
  refit = FALSE,
  packages = NULL
)
```

Arguments

name	Character(1). File stem; the result is stored at <code>file.path(dir, paste0(name, ".rds"))</code> .
code	The computation: an inline call or <code>{ }</code> block, or a variable holding a quoted call.
seed	NULL or a single whole number.
dir	Character(1). An existing directory. Defaults to the study's estimates folder.
refit	Logical(1). TRUE recomputes and overwrites the file only when it is stale or unkeyed; a valid cache is still loaded, not recomputed, regardless of <code>refit</code> .
packages	NULL or a character vector of package names whose versions are added to the key. Use it for a package that does the work beneath a wrapper and so never appears in code itself; see Details.

Details

code is not evaluated on a cache hit. It can be written inline, as a single call or a `{ }` block of several steps, or passed as a variable holding a [quoted](#) call built beforehand. Both forms give the same key for the same computation. Passing an object that has already been computed is an error, because the cache could never skip it.

code must be written in the call to `cache_fit()` itself. A wrapper function that forwards its own argument, e.g. `function(nm, expr) cache_fit(nm, expr)`, forces that argument's promise when it resolves code, so the value is already computed and the call is refused as already-computed. A wrapper should instead build and pass a quoted call, for example `cache_fit(nm, bquote(...))` or by passing `quote(...)` through.

The code runs in a new environment whose parent is the caller's, so names assigned inside a block do not appear in the caller.

The key digests each free variable of the code: a name bound by an assignment, a function formal, or a for index is local within that binding's scope, so a name that is also read free elsewhere in the code still counts as an input. Values read indirectly (inside the body of a function it calls, or through `get()`) are not seen; pass such values into the code as arguments. The exception is a function defined in the global environment (as in a shared `_common.R`): its own body is part of the key, so editing it invalidates a cache that calls it, even though values it in turn reads indirectly are still not seen. This reaches only one level deep: a global helper is part of the key only through its own body, not through the bodies of further helpers it in turn calls, so editing a callee of a helper does not invalidate a cache that calls the helper. For example, if `hA <- function(z) hB(z)` is defined globally and `hB`'s body is later edited, the key computed for code that calls `hA` is unchanged.

The result is returned with its key attached as an attribute, and that attached key, not the object's own content, is what a downstream `cache_fit()` digests when this result is passed to it as an input. Mutating the returned object afterwards, for example `fit$coefficients <- ...`, is invisible to that downstream cache, because the attached key does not change to reflect the mutation. If a cached object is modified in place, strip its `hvtiRutilities_cache_key` attribute or re-wrap the result so a downstream cache is not fooled into treating it as unchanged.

The attached-key short cut above applies only to a cached object passed directly as an input. A cached object wrapped inside a list, or a model object built outside `cache_fit()`, carries no such key and is digested whole instead; such objects can digest differently across sessions or platforms even when nothing meaningful about them changed. This shows up as a loud, classed `hvtiRutilities_stale_cache` error rather than a silently wrong cache hit, which is the safer failure mode, but is worth knowing about when a downstream cache goes stale for no apparent reason.

The key records the version of the package that owns each function called in code, and no other. A wrapper is therefore keyed on itself only: `gg_partial_rfsrc(forest, ...)` records `ggRandomForests`, though `randomForestSRC` computes the partial dependence when it runs. An upgrade of `randomForestSRC` reaches that key only through `forest`'s own key, which holds the version at fit time, not the version that runs now. Name such a package in `packages`, e.g. `packages = "randomForestSRC"`, and its version joins the key, so an upgrade makes the cache stale. Adding packages to an existing call makes its cache stale once, with the package reported as (absent) in the stored key. Base-priority packages are ignored, as they are for functions called in code.

The computation must return a value; a result of `NULL` is an error, because there would be nothing to cache.

When `seed` is given, the code runs inside `with_seed`, which leaves the global random number stream as it was. When `seed` is `NULL` and the code consumes random numbers, the result is still saved, with a warning of class `hvtiRutilities_nonreproducible_fit`, and the key records `reproducible = FALSE`.

A stale file stops with an error of class `hvtiRutilities_stale_cache`; a file not written by `cache_fit()` stops with class `hvtiRutilities_unkeyed_cache`. Both are recomputed with `refit = TRUE`.

The result is written to a temporary file and renamed into place, so an interrupted run never leaves a partial file. When `dir` lies inside a study, a provenance sidecar carrying the key and the completed cache file's hash is published with `capture_provenance` and `publish_provenance`. The cache

records an explicit empty data list rather than inferring that the currently registered dataset was an input to the computation.

Value

The result of code, computed or loaded, with its key attached as attribute `hvtiRutilities_cache_key`.

See Also

[capture_provenance](#), [publish_provenance](#), [study_dir](#)

Examples

```
dir <- file.path(tempdir(), "cache-fit-example")
dir.create(dir, showWarnings = FALSE)
x <- c(3, 1, 2)
first <- cache_fit("sorted", sort(x), dir = dir)
again <- cache_fit("sorted", sort(x), dir = dir)
identical(first, again)
unlink(dir, recursive = TRUE)
```

capture_provenance	<i>Capture an explicit job provenance payload</i>
--------------------	---

Description

Freezes the study identity, executing R session, loaded packages, lockfile, and explicit data and artifact records. Each loaded package is recorded with its version and source, and a package installed from a git remote also records the commit it was built from (sha), because one version number can cover more than one build. Capture never resolves a current dataset implicitly. Pass `list()` deliberately when a job has no known direct data input.

Capture does not write a sidecar or inspect a rendered output. Use [publish_provenance](#) only after the completed output exists.

Usage

```
capture_provenance(
  job,
  data,
  artifacts = list(),
  extra = list(),
  cfg = study_config()
)
```

Arguments

job	Character(1). Stable job identity.
data	List of records returned by provenance_data .
artifacts	List of records returned by provenance_artifact .
extra	Named list of job-specific fields. Reserved capture and publication fields cannot be displaced.
cfg	List. A study manifest from study_config .

Value

A captured provenance payload as a plain list.

See Also

[publish_provenance](#), [record_provenance](#)

Examples

```
root <- file.path(tempdir(), "capture-provenance-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
  file.path(root, "00_datasets", "built.csv"),
  row.names = FALSE)
invisible(register_data(root, built = "built.csv"))
cfg <- study_config(root)

payload <- capture_provenance("death-hz-ac",
  data = list(provenance_data(cfg = cfg)),
  cfg = cfg)

names(payload)
unlink(root, recursive = TRUE)
```

check_data_updates	<i>Check a study for published dataset updates</i>
--------------------	--

Description

Deprecated. Catalog releases are retired; use [study_status](#) or [verify_manifest](#).

Compares each release-aware dataset contract with its producer-owned catalog. The check verifies the pinned release and every later published candidate without changing `_study.yml`, `manifest.yml`, or any cache file. Legacy dataset contracts have no update rows.

Usage

```
check_data_updates(cfg = study_config(), dataset = NULL)
```

Arguments

cfg	List. A study manifest from study_config .
dataset	Character(1) or NULL. Logical dataset name. When omitted, check every release-aware dataset in the study.

Value

An object of class "data_update_report": a data frame with columns dataset, scope, pinned_release_id, candidate_release_id, sequence, file, status, is_latest, and detail. Status is one of "CURRENT", "UPDATE AVAILABLE", "UPDATE STATUS UNKNOWN", "WITHDRAWN", or "FAIL".

See Also

[review_data_update](#), [adopt_data_update](#), [read_built](#)

Examples

```
root <- file.path(tempdir(), "check-data-updates-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
  file.path(root, "00_datasets", "built.csv"),
  row.names = FALSE)
invisible(register_data(root, built = "built.csv"))
cfg <- study_config(root)

# Deprecated: it warns and names study_status() instead. A study registered
# without a catalog release has nothing to report.
check_data_updates(cfg)
study_status(root)
unlink(root, recursive = TRUE)
```

clean_labels

Apply label overrides from a YAML file (deprecated)

Description

Deprecated. Use [apply_label_overrides](#)() instead. `clean_labels()` has been renamed for clarity. This function is kept as an alias for backward compatibility.

Usage

```
clean_labels(label_map_df, overrides_file = "labels_overrides.yml")
```

Arguments

label_map_df	A data frame (label map or dataset) passed to apply_label_overrides .
overrides_file	Path to a YAML file containing label overrides. Defaults to "labels_overrides.yml".

Value

When given a label map: the updated label map data frame. When given a data frame: the data frame with labels applied via `labelled::var_label()`. In both cases, variables not mentioned in the YAML file are left unchanged.

Examples

```
# Use apply_label_overrides() instead
tmp <- tempfile(fileext = ".yaml")
writeLines("age: 'Patient Age (years)'", tmp)

library(labelled)
dta <- data.frame(age = c(25, 30, 35))
var_label(dta$age) <- "Patient Age"
lmap <- label_map(dta)

lmap <- clean_labels(lmap, overrides_file = tmp)
unlink(tmp)
```

cohort_counts

Count the analysable cohort

Description

Counts rows for which both explicitly supplied event and time columns are present, and the events among them.

The event column must be logical or numeric binary coding: FALSE/TRUE or 0/1. Rows missing either column are excluded from the analysable cohort.

Usage

```
cohort_counts(d, event, time)
```

Arguments

d	A data frame.
event	Character scalar naming the binary event column.
time	Character scalar naming the time column.

Value

A list with integer elements `n`, `n_events` and `n_censored`.

See Also

[assert_cohort](#)

Examples

```
d <- data.frame(dead = c(1, 1, 0, 0, 0), iv_dead = 1:5)
cohort_counts(d, event = "dead", time = "iv_dead")
```

compare_datasets	<i>Compare two versions of a dataset</i>
------------------	--

Description

Summarises the differences between two data pulls of the same dataset: added or dropped columns, row count changes, type changes, and label changes. This is useful for auditing data drift when a new extract arrives, and pairs naturally with the manifest system.

Usage

```
compare_datasets(old, new)
```

Arguments

old	A data frame representing the previous version of the dataset.
new	A data frame representing the current version of the dataset.

Value

A list with the following elements:

rows_old Number of rows in old

rows_new Number of rows in new

cols_added Character vector of column names present in new but not old

cols_dropped Character vector of column names present in old but not new

type_changes Data frame with columns variable, old_class, new_class for shared columns whose primary class changed

label_changes Data frame with columns variable, old_label, new_label for shared columns whose label changed

See Also

[update_manifest](#), [verify_manifest](#)

Examples

```
# Simulate two data pulls
v1 <- generate_survival_data(n = 100, seed = 1)
v2 <- generate_survival_data(n = 120, seed = 2)

# Add a column to v2 and drop one
v2$new_var <- rnorm(120)
v2$dead <- NULL

diff <- compare_datasets(v1, v2)
diff$rows_old
diff$rows_new
diff$cols_added
diff$cols_dropped
```

compare_parity

Compare one R quantity against its SAS reference

Description

Errors – never warns, never skips – when the quantity is absent on either side. A comparison that cannot fail is worse than no comparison.

Usage

```
compare_parity(quantity, r, sas, class, source = "1st", digits = NA_integer_)
```

Arguments

quantity	Name of the quantity, used in the report.
r	Value computed in R.
sas	Value from the SAS reference.
class	Tolerance class; see parity_tolerance .
source	Where the SAS value came from – "1st" or "outhaz".
digits	For class = "printed", the number of decimal places the reference was printed to, as a single non-negative whole number. The tolerance becomes half of the last place.

Details

The outcome is three-state. R_BETTER fires only for a log-likelihood that exceeds SAS's beyond tolerance: a multi-start optimizer regularly beats a single-start one, and recording that as a failure would train the reader to distrust a real improvement.

Value

A one-row data frame.

Relative discrepancy at a zero reference

When the SAS value is zero there is no relative scale. The reported rel_diff is 0 if the two agree exactly and Inf otherwise – never NA, which [parity_headline](#) would drop, reporting a real failure as no discrepancy at all.

Examples

```
compare_parity("log_likelihood", r = -239.1941, sas = -239.194,
               class = "loglik")
```

concept_map	<i>Map a candidate pool to concepts</i>
-------------	---

Description

One row per candidate: which concept it belongs to, and whether it is that concept’s representative. Returned rather than applied, so a caller can put the map in its report **before** anything is discarded.

Usage

```
concept_map(pool, prefer = character(0), aliases = character(0), ...)
```

Arguments

- pool Character vector of candidate names.
- prefer Named character vector c(concept = variable).
- aliases Named character vector, as for [concept_of](#).
- ... Passed to [concept_of](#).

Details

prefer overrides which form represents a concept, as c(concept = variable). **The default is a modelling assumption, not a neutral choice:** keeping the plain form asserts the effect is linear in it. Measured on one study, ln_crcl selected in 43.8% of replicates against 18.0% for crcl_pr, so defaulting to the plain form would have *lowered* a genuine risk factor’s frequency. Override where the study has a view; the map goes into the report either way, so the choice is visible rather than implied.

Value

A data frame with columns variable, concept and representative, ordered by concept with the representative first.

See Also

[concept_of](#), [prune_to_one_form](#)

Examples

```
concept_map(c("age", "ln_age", "age2", "agee"))
```

concept_of	<i>The concept a candidate variable belongs to</i>
------------	--

Description

A SAS candidate pool offers every transformation of a variable as a separate candidate: age, ln_age, in_age, age2 and agee all compete. A stepwise screen then spends its steps on them, and a per-variable selection frequency reports each form separately. On one real screen five forms of age each cleared 80% while the retained set collapsed to four distinct clinical concepts – and the paper the job comes from reports **concepts**.

Usage

```
concept_of(  
  v,  
  pool,  
  affixes = POOL_AFFIXES,  
  plain_suffix = POOL_PLAIN_SUFFIX,  
  aliases = character(0),  
  min_stem = POOL_MIN_STEM  
)
```

Arguments

v	Character scalar: the variable name.
pool	Character vector: the candidate pool v belongs to.
affixes	Character vector of regular expressions for the study's transformation affixes. See POOL_AFFIXES .
plain_suffix	Suffix carried by untransformed measurements.
aliases	Named character vector, c(variable = concept), declaring groupings no affix rule can reach. An alias on the variable itself wins over every rule: it is the study saying so directly.
min_stem	Minimum stem length before the prefix rule is tried.

Details

The rule is deliberately conservative. A variable joins a concept only when stripping one known affix yields a name that is *itself* in the pool (or that name plus plain_suffix). So zexp2 groups under zexp because zexp is a candidate, and ln_crcl groups under crcl_pr because that is the plain form present. A name whose stem is absent stays its own concept: agee does not join age, because no rule strips a trailing e.

Grouping too little costs some pruning. Grouping too much **silently merges two clinical concepts and discards a real candidate**, which is the error that cannot be seen in the output. The rule errs the safe way, and callers are expected to report the map so a human can check it.

Two mechanisms are needed, and only one can be mechanical. `vars.sas` truncates the stem when it names a derivative, so stripping the affix yields a name absent by exact match:

- *Truncation* – `effi` is a **prefix** of `effic`, so it can be found. Handled automatically, but only when the stem prefixes exactly one pool member that is not itself a derived form, and only when the stem is at least `min_stem` characters. Ambiguity declines to group rather than guessing.
- *Contraction* – `arin` is not a prefix of `area_int` and no rule can find it. Declare it via `aliases`, so the judgement is recorded in the job rather than inferred by the function.

Left ungrouped on a real screen, `effic` at 86.4% and `ln_effi` at 60.8% were both reported as retained risk factors at $r = 0.9988$, and `in_arin` at 70.6% hid a concept whose union across its forms was 86.4%.

Value

A character scalar: the concept, or `v` itself when no rule applies.

A permanent compatibility layer, not a temporary shim

Stem truncation exists because SAS capped variable names at 8 characters. A modern builder has no such cap and a pool built that way needs none of this, because `ln_efficiency` strips cleanly to `efficiency`.

But the two regimes coexist indefinitely: reproducing a legacy SAS job in R is routine work, not a migration with an end date, so truncated pools keep arriving from `built*.sas7bdat` extracts for as long as old jobs are re-run. **Do not delete the prefix rule or the alias table when a new builder ships.** They are how a legacy pool is read. The right expectation is that new datasets simply never need them.

See Also

[concept_map](#), [prune_to_one_form](#), [selection_crowding](#)

Examples

```
pool <- c("age", "ln_age", "age2", "agee")
vapply(pool, concept_of, character(1), pool = pool)
```

covariate_audit	<i>Audit what would happen to each covariate in a model</i>
-----------------	---

Description

Reports, one row per variable, what a column is and what using it as a model covariate would do to it – so a job can **report** the decision rather than make it silently.

Usage

```
covariate_audit(data, vars, max_levels = 12L)
```

Arguments

data	A data frame.
vars	Character vector of covariate names.
max_levels	Integer. Columns with more distinct values than this are treated as continuous and their non-integer levels are reported as NA.

Details

The problem this exists for. `vars.sas` is commonly called as `%vars(missing=1, impute=1)`, which mean-imputes missing values and adds a paired `ms_*` missing indicator. So a 0/1 clinical variable arrives carrying three distinct values: 0, 1, and the cohort mean. One real example: `hx_htn`'s third value is 0.714047, the prevalence of hypertension, not any patient's hypertension status.

In SAS these columns are numeric and enter a model **linearly**: an imputed row contributes the mean, and the `ms_*` indicator carries whatever the missingness itself is worth. That is the design the `.sas` job specifies.

Read into R they arrive as **factors**. Put a factor in a model formula and `model.matrix()` builds dummies from it, which makes "this value was imputed" its own category with its own coefficient. **That is a different model, with a different parameter count, and nothing in the output says so.** Multi-level counts have the same problem for a different reason: a `surg_num` with levels 1, 2, 3, 4 is a number of previous operations, not four unordered categories.

`max_levels` bounds where an imputed value is looked for at all. On a **continuous** covariate every value is its own level and most are non-integers, so an unbounded search returns the whole column.

That bound is also an honest statement of the limit. Mean imputation *is* detectable in a discrete column, because the mean is not a value the column can legitimately take. In a continuous column it is invisible by construction – an imputed mean looks exactly like a measured value – and the paired `ms_*` indicator is the only record that it happened. So NA in `noninteger_levels` means **"not knowable here"**, not "none".

Value

A data frame with columns variable, storage, n_levels, noninteger_levels and action. noninteger_levels reports value (rows) **per level**: per level rather than as a total because the total cannot distinguish a mean-imputed binary (one such level, in a minority of rows) from an inverse transformation whose several non-integer levels are ordinary data. A blank means "no such level"; NA means "not looked for". action is what [covariates_to_numeric](#) would do, and an action beginning ERROR marks a variable that must not be fitted.

See Also

[covariates_to_numeric](#), [imputed_levels](#)

Examples

```
d <- data.frame(hx_htn = factor(c("0", "1", "0.714047")),
               age = c(60, 71, 55))
covariate_audit(d, c("hx_htn", "age"))
```

`covariates_to_numeric` *Apply what covariate_audit() reports*

Description

Factors whose levels are all numeric become numeric, so they enter a model linearly the way the SAS job specifies.

A factor with a genuinely non-numeric level is **left alone** and will be dummy-coded – the right outcome for a real categorical, and the wrong one for a mean-imputed binary. [covariate_audit](#)'s action column is where a reader sees which of the two happened; check it before reading any coefficient.

Usage

```
covariates_to_numeric(data, vars)
```

Arguments

data	A data frame.
vars	Character vector of covariate names.

Value

data, with the convertible columns coerced to numeric.

See Also

[covariate_audit](#)

Examples

```
d <- data.frame(hx_htn = factor(c("0", "1", "0.714047")))
str(covariates_to_numeric(d, "hx_htn"))
```

data_dictionary

*Build a data dictionary from a labeled dataset***Description**

Creates a summary data frame describing every variable in a dataset: name, label, R class, number of unique values, percent missing, and a compact range or level summary.

This is the single most useful artifact for documenting a clinical dataset. It can be written directly to CSV or included in a Quarto document as a table.

Usage

```
data_dictionary(data)
```

Arguments

data A data frame, tibble, or similar tabular object.

Value

A data frame with one row per variable and the following columns:

variable Column name

label Descriptive label (falls back to variable name if unlabeled)

class Primary R class of the column

n_unique Number of unique non-NA values

pct_missing Percentage of rows that are NA (0-100)

summary Compact summary: min / median / max for numeric, level counts for factor/character, TRUE percentage for logical

See Also

[proc_contents](#) for the full SAS-style variable table, [proc_means](#) for numeric summaries, [label_map](#) for extracting labels only, [r_data_types](#) for automatic type conversion before building the dictionary.

Examples

```
# Quick dictionary from synthetic clinical data
dta <- generate_survival_data(n = 100, seed = 42)
dict <- data_dictionary(dta)
head(dict, 10)

# After type conversion
dta_clean <- r_data_types(dta,
  factor_size = 5,
  skip_vars = c("ccfid", "iv_dead", "iv_reop", "iv_opyrs")
)
dict_clean <- data_dictionary(dta_clean)
head(dict_clean, 10)

# Write to CSV for documentation
out <- tempfile(fileext = ".csv")
utils::write.csv(data_dictionary(dta), out, row.names = FALSE)
unlink(out)
```

dataset_schema	<i>Column-level schema of a dataset</i>
----------------	---

Description

One row per column: creation position, name, R class, SAS two-valued type, `format.sas` and label. This is the durable description of a source dataset — what a schema sidecar records, and what a later read is compared against.

Usage

```
dataset_schema(data)
```

Arguments

`data` A data frame, tibble, or similar tabular object.

Details

label and format are read from the column attributes directly and are NA when the source carries none. This is the difference from [proc_contents](#), which fills an absent label with the variable's own name: right for a printed listing, wrong for a record that outlives the source dataset.

Nothing here describes the data, only its shape. Two reads of an unchanged file produce an identical schema, which is what makes the sidecar's hash meaningful.

Value

A data frame with columns `num` (creation position, integer), `variable`, `class` (first R class), `type` ("Num"/"Char"), `format` (SAS format or NA) and `label` (or NA).

See Also

[proc_contents](#) for a printable listing that also summarises completeness.

Examples

```
d <- data.frame(x = 1:3, y = letters[1:3])
attr(d$x, "label") <- "An identifier"
dataset_schema(d)
```

followup_check

Check recorded follow-up before a time-related analysis

Description

Summarises an event indicator and its follow-up intervals the way the dc-gfup template reports them: counts for the whole cohort and its event and censored subsets, the missing, negative and zero intervals, a [proc_means](#) table for each subset, and the rows worth a second look. It checks recorded follow-up; it does not establish completeness against a close date.

Usage

```
followup_check(
  data,
  event,
  followup,
  identifier = NULL,
  max_rows = 25L,
  stats = c("n", "nmiss", "mean", "std", "min", "p15", "median", "p85", "max")
)
```

Arguments

data	A data frame, one row per patient.
event	Name of the event indicator: 1 or TRUE for an event, 0 or FALSE for censored. Missing values are allowed and counted.
followup	Names of one or more follow-up interval columns, in years.
identifier	NULL (the default), or the name of one column to show in review.
max_rows	The most suspicious rows to return in review, in data order. Default 25.
stats	Character vector of SAS statistic keywords for the means tables, passed to proc_means and validated the same way. Default c("n", "nmiss", "mean", "std", "min", "p15", "median", "p85", "max").

Details

Every argument is checked before anything is computed, and every column that is not in data is named in one error, so a job with three misspellings says so once.

A row with a missing event is counted in `missing_event` and belongs to neither the event nor the censored subset. A row is *suspicious* when its event is missing or any of its intervals is missing, negative or zero.

The quartiles in `intervals` use R's default interpolated quantiles (`type = 7`), while `proc_means` uses the SAS `QNTLDEF=5` estimator, so a median in the two tables can disagree on a small subset. The difference is deliberate: `intervals` is a quick screen, and `means` is the table to compare against SAS.

`means` reports the 15th and 85th percentiles rather than the quartiles, because those are what the group's descriptive summaries show. SAS PROC MEANS prints no percentiles unless asked, so a parity comparison names the same statistics on the SAS side, or passes the SAS job's list to `stats`.

Identifiers are never included unless named in `identifier`. Check the result before sharing a report that shows review with one.

Value

An object of class `followup_check`, a list of four components, three data frames and a list of three:

`cohort` One row: full, event, censored and `missing_event` counts.

`intervals` One row per interval: missing, negative and zero counts, then min, q1, median, q3, mean, sd and max of the observed values.

`means` A named list, full, event and censored, of `proc_means` tables over the intervals, one column per statistic in `stats`.

`review` Up to `max_rows` suspicious rows, with the event, the intervals and any identifier.

See Also

`proc_means`, `cohort_counts`, which counts the analysable cohort rather than checking follow-up.

Examples

```
d <- data.frame(dead = c(1, 0, 0, NA, 1), iv_dead = c(2.5, 4, 0, 1.2, NA))
fc <- followup_check(d, event = "dead", followup = "iv_dead")
fc
fc$cohort
fc$review
```

```
generate_survival_data
```

Generate Synthetic Cardiac Surgery Survival Data

Description

Creates a simulated dataset that mimics a cardiac surgery cohort, suitable for testing survival analysis workflows in `'rf.survival.study.qmd'`. Variable labels are attached via `labelled::var_label()` so they are compatible with `haven` and `hvtiRutilities::label_map()`.

Usage

```
generate_survival_data(n = 500, seed = 1024)
```

Arguments

<code>n</code>	Integer. Number of patients to simulate. Default 500.
<code>seed</code>	Integer. Random seed passed to set.seed for reproducibility. Default 1024.

Details

Survival times are drawn from a Weibull distribution (shape = 1.5, ~8-year median follow-up) with a linear predictor built from LVEF, age, baseline hemoglobin, NYHA class, and eGFR. Administrative censoring is applied uniformly between 1 and 15 years. Reoperation probability is modelled via a logistic function of bypass time and NYHA class.

The returned data frame includes the five columns excluded from `model_data` in the template (`ccfid`, `reop`, `iv_reop`, `origin_year`, `iv_opyrs`), plus the survival outcome (`iv_dead`, `dead`) and 17 clinical predictors spanning demographics, pre-operative labs, cardiac function, and surgical variables.

The global RNG state is saved and restored on exit, so calling this function does not affect subsequent random number generation in the session.

Value

A data.frame with `n` rows and 24 columns:

ccfid Character. Patient identifier ("PT00001", ...).

origin_year Integer. Calendar year corresponding to `iv_opyrs = 0` (randomly sampled between 1998 and 2018).

iv_opyrs Numeric. Observation interval length in years measured from `origin_year`; always greater than or equal to `iv_dead`.

iv_dead Numeric. Observed follow-up time in years (min of true survival time and administrative censoring time).

dead Integer. Event indicator: 1 = death, 0 = censored.

reop Integer. Reoperation indicator: 1 = yes, 0 = no.

iv_reop Numeric. Time to reoperation (years); NA if no reoperation.

age Numeric. Age at surgery in years (range 1–85).

sex Factor. "Male" / "Female" (55 / 45 %).

bmi Numeric. Body mass index in kg/m² (range 15–50).

hgb_bs Numeric. Baseline hemoglobin in g/dL (range 6–18).

wbc_bs Numeric. Baseline WBC count in K/ μ L (range 1.5–20).

plate_bs Numeric. Baseline platelet count in K/ μ L (range 50–500).

gfr_bs Numeric. Baseline eGFR in mL/min/1.73 m² (range 10–120).

lvefvs_b Numeric. Baseline LV ejection fraction in % (range 15–75).

lvmass_b Numeric. Baseline LV mass in g (range 60–400).

lvmsi_b Numeric. Baseline LV mass index in g/m² (range 40–220); derived as `lvmass_b / (bmi * 0.5)`.

stvoli_b Numeric. Baseline systolic stroke-volume index in mL/m² (range 20–110).

stvoid_b Numeric. Baseline diastolic stroke-volume index in mL/m² (range 40–180).

bypass_time Numeric. Cardiopulmonary bypass time in minutes (range 20–240).

xclamp_time Numeric. Aortic cross-clamp time in minutes; sampled as 50–75 % of `bypass_time`.

nyha_class Ordered factor. NYHA functional class I–IV.

diabetes Factor. Diabetes mellitus: "No" / "Yes" (75 / 25 %).

hypertension Factor. Hypertension: "No" / "Yes" (55 / 45 %).

See Also

[rfsrc](#), [Surv](#), [read_sas](#)

Examples

```
dta <- generate_survival_data(n = 200, seed = 42)
str(dta)
table(dta$dead)          # event rate
summary(dta$iv_dead)     # follow-up distribution

# Drop admin columns to match model_data in the Quarto template
model_data <- dta[, !names(dta) %in% c("ccfid", "reop", "iv_reop",
  "origin_year", "iv_opyrs")]
```

get_label	<i>Look up the label for a single variable</i>
-----------	--

Description

Returns the descriptive label for one variable name from a label map. This is a safer alternative to the manual `match()` pattern, providing clear errors on typos and missing variables.

Usage

```
get_label(label_map_df, variable)
```

Arguments

label_map_df	A data frame with key and label columns, as returned by label_map .
variable	A single character string: the variable name to look up.

Value

A single character string: the label for the requested variable.

Examples

```
dta <- generate_survival_data(n = 50, seed = 42)
lmap <- label_map(dta)

get_label(lmap, "age")
get_label(lmap, "hgb_bs")

# Use in plot titles
var <- "lvefvs_b"
plot(dta[[var]], main = get_label(lmap, var), ylab = get_label(lmap, var))
```

get_labels	<i>Look up labels for multiple variables at once</i>
------------	--

Description

A vectorized companion to [get_label](#). Returns a named character vector of labels for one or more variable names, making it convenient to label axes, table columns, or multi-panel plots in a single call.

Variables not found in the label map cause an error (just like `get_label`), so typos are caught immediately.

Usage

```
get_labels(label_map_df, variables)
```

Arguments

`label_map_df` A data frame with key and label columns, as returned by [label_map](#).
`variables` A character vector of variable names to look up.

Value

A named character vector with names equal to `variables` and values equal to the corresponding labels.

See Also

[get_label](#) for single-variable lookup, [label_map](#) to extract a label map from data.

Examples

```
dta <- generate_survival_data(n = 50, seed = 42)
lmap <- label_map(dta)

# Look up several labels at once
get_labels(lmap, c("age", "bmi", "hgb_bs"))

# Useful for table column headers
vars <- c("age", "bmi", "lvefvs_b")
headers <- get_labels(lmap, vars)
print(headers)
```

guard_snapshot

Refuse a renv snapshot taken from the wrong R

Description

Stops before `renv::snapshot()` when the R running this session has a different major.minor version than the R recorded in the study's `renv.lock`. A snapshot from the wrong R rewrites the lockfile's R version and drops every locked package the session's library lacks. On 2026-10-22, R 4.6.1 snapshotted a study locked to R 4.5.3, and **boostmtree** and **ggBoostedTrees** fell out of its lockfile.

A patch difference (4.5.1 against 4.5.3) passes, as does a study with no `renv.lock`.

Use `r_upgrade = TRUE` only when you are deliberately moving the study to a new R. The guard then still stops if any package in the lockfile is not installed in this session's library, because the snapshot would drop it; restore those first with `renv::restore()`.

[study_need](#) and [study_setup](#) call this before they snapshot, so you need to call it yourself only before a snapshot of your own.

Usage

```
guard_snapshot(
  root,
  r_upgrade = FALSE,
  session = .session_r(),
  installed = .installed_version
)
```

Arguments

<code>root</code>	Character(1). The study root, holding <code>renv.lock</code> .
<code>r_upgrade</code>	Logical(1). The study is being moved to this session's R on purpose.
<code>session</code>	The R version of this session, a numeric_version . Defaults to the running R; set it only to test the guard.
<code>installed</code>	A function taking a package name and returning its installed version, or <code>NA_character_</code> when it is not installed. Defaults to a lookup in this session's library; set it only to test the guard.

Value

TRUE, invisibly, when the snapshot may go ahead. Otherwise an error that names both R versions and the R to start.

Examples

```
root <- tempfile("study")
dir.create(root)
# A study with no renv.lock has nothing to protect:
guard_snapshot(root)

# A lock written by R 4.5.3, checked from a session running R 4.6.1:
writeLines('{ "R": { "Version": "4.5.3"}, "Packages": {} }',
  file.path(root, "renv.lock"))
try(guard_snapshot(root, session = numeric_version("4.6.1")))
unlink(root, recursive = TRUE)
```

<code>hvti_non_prefixes</code>	<i>Second fields that are not analysis prefixes</i>
--------------------------------	---

Description

Some file names lead with a utility name rather than an analysis prefix — plots, PPTs. They are listed here so the test suite can tell "not a prefix" apart from "a prefix nobody documented". Without this distinction the taxonomy either fills with non-prefixes or stops catching real omissions.

Usage

```
hvti_non_prefixes()
```

Value

A character vector.

Examples

```
hvti_non_prefixes()
```

hvti_prefix_folds	<i>Legacy prefixes folded into a taxonomy prefix</i>
-------------------	--

Description

A legacy prefix that names the same job as a prefix in [hvti_taxonomy](#), so a census counts its files under that prefix. pm folds into lm: a review of the job catalog on 2026-09-11 found that the propensity work filed as pm belongs with the logistic models in lm.

Usage

```
hvti_prefix_folds()
```

Details

The map is stored, never derived. A fold is a judgement about what a legacy name means, and a rule that inferred it from spelling would also fold names that merely look alike. Every name is absent from `hvti_taxonomy()` and every value is present in it; the tests pin both.

Value

A named character vector, `c(legacy = prefix)`.

See Also

[job_files](#), which applies it.

Examples

```
hvti_prefix_folds()
```

hvti_taxonomy	<i>Analysis prefix taxonomy</i>
---------------	---------------------------------

Description

The prefix system inherited from the original CORR analysis binder. The prefix encodes both the type of analysis and the folder the job belongs in.

Usage

```
hvti_taxonomy()
```

Details

This is data rather than documentation on purpose. The same table lived in a README and drifted from the files it described; as a function it is checked by the test suite against the templates actually present.

Folder names are load-bearing. Row order is not, folder order included, as of 2026-09-03.

A template directory and a numbered study directory are both spelled <NN>_<folder>, as in 20_distributions; a legacy study keeps the bare name, as in distributions, and `study_dir` resolves either. The name after the digits, or the bare name, must be a folder in this table. `hvtiRtemplates` tests that every template directory names one, so renaming or removing a folder that holds a template turns its CI red. The digits are *assigned*, not derived from this table. They are hardcoded in `study_dir`'s layout map here and in the template directory names there, and estimates is 90 though it is the table's fifth folder, because it holds saved output rather than jobs. Reordering rows, or folders, changes nothing downstream.

Adding or renaming a folder is not free on the study side. `study_dir` knows only the folders in its own layout map, and a test compares that map's names with `unique(hvti_taxonomy()$folder)`, so a folder added here and not there turns this package's CI red rather than becoming one `study_dir()` refuses as unknown. The test compares names only; the new folder's digits still have to be assigned in the map.

Templates carry no ordinal. They were once named <NN>.<MM>-<prefix>, with NN taken from this table's folder order through a hardcoded map in `hvtiRtemplates`; the ordinal and that map were both retired on 2026-09-03 (`hvtiRtemplates:dev/specs/2026-09-03-template-identity-design.md`). A template is now <prefix>[-<qualifier>].qmd, and the catalog that lists them is `hvtiRtemplates'` own `inst/extdata/templates.json`.

History, kept because it is why the digits are assigned: moving hs out of analyses in 1.1.6 shifted bh from sixth to fifth while its shipped filename stayed 04.06, and nothing caught it. bh was renumbered 04.05. Identity derived from a row position breaks when an upstream row moves. A job named 04.06-bh, from `hvtiRtemplates` before its 1.1.0, is the same template as 04.05-bh.

hs is filed under graphs, which reads oddly for a job named "setup". It was moved there from analyses on 2026-08-29 on the corpus rather than on the name: all ten `tp.hs.*` templates in the SAS library and ten of the eleven R hs jobs in `/studies` sit in `graphs/`. It computes patient-level predictions that the plotting jobs beside it consume, in the `setup / uses_setup` pairing the corpus uses throughout. See `hvtiRtemplates:dev/specs/2026-08-29-hs-template-design.md`.

si and mi are two prefixes, not one, and that is the point. 223 studies call single mean imputation, 326 call multiple imputation, and 18 call both. They are different methods with different inferential properties, and 18 studies running both means one prefix could not label them unambiguously. *mi* is safe here only because it is *paired*: standing alone it reads as multiple imputation to a statistician, so using it for the single-imputation job would misname exactly the thing the split exists to distinguish.

Neither replaces vars, whose description still says "imputations" on purpose. *vars* is the job that enhances a dataset with temporary variables, imputations and propensity terms together – porting *vars.sas* found mean imputation across 394 variables inside it. *si* and *mi* name jobs whose *whole purpose* is imputation. A job that imputes as one step among several is *vars*; a job that exists to impute is *si* or *mi*.

folder names two different things. For most rows it is the analysis type's home folder, matched to a job prefix. One row, *estimates*, is an artifact kind rather than a job type: it holds serialized fits and cached results written by one job and read by a later one in the same set, and no analysis produces it directly. That row's prefix is NA, not a string, because there is no prefix to assign it.

The random forest family splits on the OUTCOME axis, not the package axis. *rfs*, *rfc* and *rfr* name the survival, classification and regression outcomes, which is the split *bh/bl/bc/bn/bq/br* and *pm/rm/cm* already use. It is a rename of a distinction the corpus draws already, not a new one imposed: the jobs carry the outcome in field two today, as *tp.rfsrc.survival.R* and its siblings.

rf and rfsrc are retained as legacy umbrella rows. They named the same set as each other, on the package axis, and neither is templated. They stay in the table so a census can still resolve the corpus that uses them, which is not small: *rfsrc* measures 131 studies and 2,295 jobs, more than *rf* (47), *rfs* (25) and *rfc* (19) combined, twice over. Deleting the rows was rejected on two grounds. [hvti_prefix_folds](#) cannot express the demotion, because a fold is one-to-one and *rfsrc* spans all three outcomes, so there is no single prefix to fold it into. And an absent prefix reads to the census as one nobody documented, which is the opposite of what a superseded name means. Demoted here means *never templated*, and since 2026-09-18 it is machine-readable as well as worded: the umbrella column is TRUE on exactly these two rows, so a consumer can exempt them without hard-coding a list. The two rows keep distinct descriptions so a census hit on a legacy job can still say which spelling it used.

The demotion is recorded here, and only here. Until 2026-09-19 *hvtiR*'s job catalog also marked *rf* and *rfsrc* with a retire disposition, which coincided with this demotion by decision. That catalog has moved to *hvtiRtemplates* as its template catalog, read by *hvtiRtemplates::template_catalog()*, where every row is a template owed in that package. It has no retire disposition, and *rf* and *rfsrc* have no row at all; its guard that every taxonomy prefix has a catalog row exempts them by reading the umbrella column above. *hvtiR::jobs()* was removed in *hvtiR* 1.2.0.

sid and vt are filed under analyses with the rest of the family, and compose on the *rf** jobs rather than replacing them. The *rfc/rfs* folder contradiction is deliberately NOT settled here: the taxonomy says analyses while the SAS library files them under graphs and documents, and the 2026-08-27 census found the same contradiction corpus-wide for *ac/hz*. It is a systemic open question with its own spec owed, and the ML batch is not blocked waiting on it. See [hvtiRtemplates:dev/specs/2026-09-17-ml-family-roadmap-design.md](#).

Value

A data frame with columns *prefix*, *name*, *folder*, *description* and *umbrella*. *prefix* is NA for the one row that names an artifact kind rather than an analysis type. *umbrella* is logical: TRUE for

the legacy umbrella prefixes `rf` and `rfsrc`, for which no template is owed; `FALSE` for every other prefix; `NA` for the artifact row.

Examples

```
head(hvti_taxonomy())
```

<code>hvtiRutilities.news</code>	<i>Display the NEWS file</i>
----------------------------------	------------------------------

Description

Display the NEWS.md file for the `hvtiRutilities` package showing release notes and change history.

Usage

```
hvtiRutilities.news()
```

Value

Invisibly returns `NULL`. Called for side effect of displaying NEWS.

Examples

```
# Opens a pager, so only in an interactive session.
if (interactive()) hvtiRutilities.news()
```

<code>imputed_levels</code>	<i>Levels of a column that are not clean integers</i>
-----------------------------	---

Description

On a mean-imputed column these are the imputed value, which is the signature that imputation happened at all: a genuine 0/1 indicator has no such level.

Usage

```
imputed_levels(x)
```

Arguments

<code>x</code>	A vector or factor.
----------------	---------------------

Value

A character vector of the offending levels, possibly empty.

See Also[covariate_audit](#)**Examples**

```
imputed_levels(c(0, 1, 0.714047, 1))
```

job_census	<i>Roll a job-file inventory up to studies and prefixes</i>
------------	---

Description

Answers the question the template roadmap keeps needing: for each job prefix, which studies have run it and how many jobs each.

Usage

```
job_census(x)

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

Arguments

x	Character roots to sweep – see job_files for what a root must be – or a data frame returned by job_files . A data frame missing any of that function’s 16 columns is an error, not a silently empty census.
...	Ignored; present for S3 consistency with print.

Details

Two count columns, deliberately. `n_jobs` counts distinct stems and is the honest unit – the `.lst`, `.sas` and `.log` of one job are one job, and an editor backup does not create a second. `n_files` counts rows, and exists because the hand-count this replaces counted files; keeping both means the new output can be reconciled against the table that already drove a decision.

A template is not a job, by either marker. `is_template` is strictly the legacy `tp. prefix` (spec section 4); a file written in the template naming convention (section 4.2, e.g. `03.01-ac.qmd`) is *also* not a job, but with `is_template = FALSE`. `is_template_naming` carries that second signal, and the roll-up keys on both, so a template-convention file never shares a row with a genuine job at the same (study, prefix, folder) – if it did, that row could outlive the genuine job and still read as an attestation the distinct-studies gate (a prefix needs a second study running the real job before it is templatable) would wrongly count.

The `job_files()` rows are kept on the result as the “files” attribute, so the accounting – unplaced files, unknown prefixes, misfiled jobs – stays reachable from the summary rather than being computed and thrown away.

The roll-up cannot key every row, and says so rather than shrinking quietly: a file with no study or no prefix has nothing to group by. Those rows are attached as the "not_rolled_up" attribute and counted on the first line of the print, because what was dropped is the *union* of the unplaced and the unparsed and a reader cannot compute a union from two separate totals.

Subsetting. A row subset (x[i,]) keeps both attributes. A *column* subset (x[, j]) drops them while leaving the class in place, so the result still dispatches to this print method with nothing to print from; print() detects that and errors rather than reporting zeros. The printed coverage line ("Rolled up N of M files") always describes the full original sweep, even from a row-subset x: its totals are fixed as an attribute when job_census() builds x, not recomputed from x\$n_files at print time.

Value

A data frame of class hvti_job_census with one row per (study, prefix, folder, is_template, is_template_naming) and columns n_jobs and n_files. is_template_naming is TRUE when the row's files use the template naming convention (section 4.2) – distinct from is_template, which is strictly the legacy tp. marker. The originating job_files() rows are attached as the "files" attribute, and the rows that could not be rolled up – those with no study or no prefix – as the "not_rolled_up" attribute.

See Also

[job_files](#)

Examples

```
d <- file.path(tempdir(), "job-census-example")
dir.create(file.path(d, "alpha", "distributions"), recursive = TRUE)
file.create(file.path(d, "alpha", "distributions", "hz.dead.lst"))
file.create(file.path(d, "alpha", "distributions", "hz.dead.sas"))
job_census(d)
unlink(d, recursive = TRUE)
```

job_files	<i>Inventory the job files under one or more corpus roots</i>
-----------	---

Description

Walks each root and returns one row per file. This is a filename-only sweep: no file is opened, nothing is parsed, and there is no TemporalHazard dependency.

Usage

```
job_files(roots)
```

Arguments

roots Character. One or more directories to sweep. A root is a directory that *contains* studies, not a study itself: the study is read as the path from the root down to the taxonomy folder's parent, so a root that is itself a study leaves nothing to name it and every prefix collapses to one study called "." (warned about). Each root must exist and be a directory, or this errors. A duplicate root is dropped; a root nested inside another warns and the inner one is dropped, since its files would otherwise be counted twice under two study labels.

Details

Nothing is filtered out. Placement and classification are columns, not reasons to drop a row, so a file this sweep cannot classify stays findable. A sweep that reports only what it kept makes a missing job indistinguishable from a job that does not exist. The one exception is tooling: a path with a .git or .checkpoint component holds version-control internals or a study checkpoint's mirror, not study files, and is not listed.

A scaffolded name, <subject>-<type>-<prefix>[-<qualifier>].qmd as written by `hvtiRtemplates::add_job()`, is parsed with its qualifier. So is the template-first form, <prefix>[.<qualifier>].<subject>.<type>.qmd. The `-runner.R` (or `.runner.R`) that `add_job()` writes beside some reports takes the report's stem, so it counts as a file of that job rather than as a second job.

A legacy prefix listed in [hvti_prefix_folds](#) is reported as the prefix it folds into, so a `pm.*` file counts as `lm`. stem keeps the name as written, so a folded file stays findable.

There is deliberately no extension allowlist. See vignette-adjacent design note `dev/specs/2026-08-26-job-type-invent` section 4.4: a plausible default tuned on the hazard prefixes would have dropped every R-side job in the corpus.

`folder_ok` is `folder == folder_expected`, and is therefore NA – not FALSE – whenever either side is unknown. An unplaced file has no folder and an unparsed name has no expected folder; neither is a misfiled job, and `!folder_ok` must not select them.

Run this server-side. It stats every file beneath roots, which is metadata-latency-bound over an SMB mount – a 40-file scan has timed out at two minutes over the share.

Value

A data frame with one row per file and the columns `path`, `study`, `folder`, `status`, `depth`, `naming`, `prefix`, `is_template`, `qualifier1`, `qualifiers`, `n_qualifiers`, `stem`, `ext`, `prefix_class`, `folder_expected` and `folder_ok`. The three qualifier columns carry the dot-fields of a legacy name that lie between the prefix and the extension, or the trailing qualifier field of a scaffolded name, and are NA (count zero) for every other convention. They are named by position because the second field means a different thing in each taxonomy folder; see [hvti_taxonomy](#) and the design note referenced in `'R/job_names.R'`. Zero rows if the roots hold no files.

See Also

[job_census](#), [hvti_taxonomy](#)

Examples

```
d <- file.path(tempdir(), "job-files-example")
dir.create(file.path(d, "alpha", "distributions"), recursive = TRUE)
file.create(file.path(d, "alpha", "distributions", "hz.dead.lst"))
job_files(d)[, c("study", "folder", "prefix", "status")]
unlink(d, recursive = TRUE)
```

label_map

Build a lookup map of data labels

Description

Extracts variable labels from a labeled dataset and returns them as a data frame with variable names (keys) and their corresponding labels. This is particularly useful when working with SAS datasets that include variable labels, or any dataset labeled with the `labelled` package.

A warning is issued when more than 50% of columns carry no label at all. This typically indicates the data was imported from a source without labels (e.g., plain CSV) and labels should be supplied via [add_labels](#) or a `labels_overrides.yml` file (see [apply_label_overrides](#)). A variable whose real label happens to equal its own name does **not** count towards the threshold: the absent label is read as NA rather than filled, so the two cases are distinguishable here even though [proc_contents](#) cannot tell them apart.

Labels longer than `label_max` are cut for display. The cut breaks on a word boundary and is marked with `...`, and the source text is kept in `label_full` - truncation is a view, not a change to the data. [dataset_schema](#) deliberately has no such parameter: it records the label as the source carried it, because a schema sidecar outlives the source dataset.

The variable-name fallback is **exempt** from the cap. A name standing in for a missing label passes through whole, however long, and truncated is FALSE. A truncated name would match nothing in the data and would read as a deliberately short label rather than as a missing one, destroying the signal the fallback exists to give.

Labels that differ stay different. Cutting each label on its own can make two labels identical once their distinguishing ends are cut off. Within one map, labels that differ in `label_full` always differ in `label`. Each step below runs only on labels still over the cap or still colliding:

1. A heading, the text before the first ":", " - " or ";", shared by two or more labels, one of them over the cap, is abbreviated in every label that carries it: to its entry in abbreviations, or else to the initials of its words, skipping small words such as "of" and "and". "Surgical procedure" becomes "SP". A one-word heading is left alone, and two headings with the same initials are neither abbreviated.
2. Any phrase in abbreviations is applied, whole words and ignoring case, to a label still over the cap. A label that fits is never abbreviated this way.
3. The label is cut on a word boundary and marked.
4. Cut labels that still collide keep both ends, "Ascending aorta ... plus arch".
5. A label that still collides is shown whole, over the cap, and `over_cap` marks it: a long label is a layout problem a reader can see, two identical labels a wrong figure nobody can.

The abbreviations actually shown come back as the `abbreviations` attribute, to print as a key beneath a figure or table, and per variable as the `abbreviations_by_key` attribute (key, abbreviation, expansion), so a key under one section can list only what that section's labels use.

Usage

```
label_map(data, label_max = 40, abbreviations = NULL)
```

Arguments

<code>data</code>	A data frame, tibble, or similar object with variable labels (typically created using the <code>labelled</code> package or imported from SAS).
<code>label_max</code>	Maximum length of a displayed label, in characters, including the <code>...</code> marker. Defaults to 40, the historical convention. Must be at least 4, so that a cut always has room to be marked; use <code>Inf</code> or <code>NA</code> to disable truncation. Does not apply to a variable name filled in for a missing label.
<code>abbreviations</code>	<code>NULL</code> , or a named character vector in which each name is a phrase and each value is that phrase's abbreviation: <code>c("Left ventricular" = "LV")</code> . Used only on labels over the cap, and for a shared heading in place of its initials. An expansion attribute, as study_abbreviations returns, names the term each abbreviation stands for in the key. A word-like abbreviation such as "Preop" takes the case of the text it replaces.

Value

A data frame with five columns, and an `abbreviations` attribute: a data frame of abbreviation and expansion for every abbreviation the labels show, with no rows when there are none.

key Character vector of variable names from the input dataset

label Character vector of labels fit to print: the variable label cut to `label_max`, or the variable's own name where the source carries no label

label_full The label as the source carried it, never truncated, or the variable name where there is none

truncated Logical: `TRUE` where label was cut from `label_full`. Always `FALSE` for a filled variable name. `subset(x, truncated)` is the report of what was cut

over_cap Logical: `TRUE` where label is longer than `label_max` because every shorter form collided with another label. `subset(x, over_cap)` is the report

See Also

[get_label](#) for looking up a single label, [add_labels](#) for registering labels for derived variables, [apply_label_overrides](#) for applying study-specific overrides from a YAML file.

Examples

```
# Generate labeled survival data
dta <- generate_survival_data(n = 50, seed = 42)
lmap <- label_map(dta)
```

```

head(lmap)

# Use for publication-ready tables
summary_vars <- c("age", "bmi", "hgb_bs")
tbl <- data.frame(
  variable = summary_vars,
  description = lmap$label[match(summary_vars, lmap$key)],
  mean = sapply(dta[summary_vars], mean)
)
print(tbl)

# With sample data (has labels)
dta <- sample_data(n = 20)
label_map(dta)

# Which labels were cut, and what they were
subset(label_map(dta, label_max = 20), truncated)

# Keep the source text
label_map(dta, label_max = Inf)

# Labels sharing a heading keep what tells them apart, and the key says why
procs <- data.frame(avr = 1, mvr = 1)
attr(procs$avr, "label") <- "Surgical procedure: aortic valve replacement with root enlargement"
attr(procs$mvr, "label") <- "Surgical procedure: mitral valve repair"
lmap <- label_map(procs)
lmap$label
attr(lmap, "abbreviations")

```

level_map

Report what stripping a code prefix would do to a dataset's levels

Description

One row per level of each discrete variable, giving the text as stored and the text that would be displayed once a leading code prefix is removed. Like [strip_level_prefix](#) it reports; it changes nothing.

Usage

```
level_map(data, vars = NULL, max_levels = 20L)
```

Arguments

data	A data frame.
vars	Optional character vector of column names to report. Defaults to every discrete column - factor, value-labelled, or character. Naming a column that is not in data is an error.

max_levels Maximum distinct values a column may have before it is skipped. A single positive whole number, finite; defaults to 20. To report every column, pass a number at least as large as the widest one - Inf is refused, because a threshold that never fires reads as a cap that is working.

Details

The shape deliberately mirrors [label_map](#) one level down: level, level_full and stripped read the same way as label, label_full and truncated. So `subset(level_map(data), stripped)` answers "which levels will print differently", exactly as `subset(label_map(data), truncated)` answers the question for labels.

Where the codes come from depends on what the column carries:

- a column with value labels - from `value_labels.yml` via [apply_value_labels](#), or from a SAS format catalog - gives the declared codes;
- a factor gives its integer level positions, which is what the column actually stores;
- a character column has no codes, and code is NA.

Plain numeric and logical columns contribute no rows: they have no level text for a prefix to sit in front of.

High-cardinality columns are skipped, not expanded. A character column of identifiers would otherwise contribute thousands of rows to a report meant to be read. Columns with more than `max_levels` distinct values are omitted and named in a warning, so a skip is never silent.

Value

A data frame with one row per reported level and the columns

key The variable name

code The code the level maps to, as character, or NA

level The text to display, with a prefix stripped where the rule in [strip_level_prefix](#) allows it

level_full The level text as stored, never modified

stripped Logical: TRUE where level differs from level_full

Zero rows, with those columns, when nothing is reportable.

See Also

[strip_level_prefix](#) for the rule itself, [label_map](#) for the same idea applied to variable labels, [apply_value_labels](#) for declaring level text outright.

Examples

```
dta <- data.frame(
  disp = factor(c("1. Home", "2. Rehab", "1. Home")),
  vess = factor(c("1 vessel", "2 vessel", "1 vessel"))
)
level_map(dta)
```

```
# Which levels will print differently
subset(level_map(dta), stripped)
```

parity_headline	<i>Summarise a parity table as one reviewer-facing claim</i>
-----------------	--

Description

The headline, not the badge, is the claim to report. It is falsifiable and independent of whatever thresholds were chosen. A maximum relative discrepancy of exactly zero across many quantities is not a triumph – it means nothing was really compared – so it is flagged rather than celebrated.

Usage

```
parity_headline(df)
```

Arguments

df A data frame of `compare_parity` rows.

Value

A single string.

Examples

```
parity_headline(compare_parity("a", 1.0001, 1, class = "mle_printed"))
```

parity_tolerance	<i>Tolerance classes for SAS parity comparison</i>
------------------	--

Description

Tolerances are derived from what limits agreement for each quantity, not tuned until things pass. Three regimes:

Usage

```
parity_tolerance(class)
```

Arguments

class One of "count", "printed", "loglik", "mle_stored", "mle_printed", "vcov_stored", "curvature".

Details

- **Printed references are intervals, not numbers.** When SAS prints -239.194 , the value that produced it lies in $[-239.1945, -239.1935)$. Half a unit in the last printed place is a floor that is derived.
- **Stored references carry machine precision**, so that floor does not apply. What remains is that two implementations run different optimizers on the same likelihood and converge to different points.
- **Counts are exact, or there is a bug.**

Value

A list with `rtol` and `atol`.

Examples

```
parity_tolerance("loglik")
```

POOL_AFFIXES

Affix conventions for grouping a SAS candidate pool into concepts

Description

Defaults for `concept_of`, `concept_map` and friends, describing the naming conventions a legacy `vars.sas` uses: `ln_` (log), `in_` (inverse), `in2` (inverse squared) and a trailing 2 (squared).

`POOL_PLAIN_SUFFIX` is the suffix untransformed measurements carry (`crc1_pr`, `bun_pr`, `hct_pr`). `POOL_MIN_STEM` guards the prefix rule: two characters would match half a pool by accident.

These are **defaults, not constants** – pass your study’s own conventions through the affixes, `plain_suffix` and `min_stem` arguments where they differ, rather than editing a copy of this package.

Usage

`POOL_AFFIXES`

`POOL_PLAIN_SUFFIX`

`POOL_MIN_STEM`

Format

Character vectors, and an integer for `POOL_MIN_STEM`.

Examples

```
POOL_AFFIXES
pool <- c("age", "age2", "crcl_pr", "ln_crcl_pr")
concept_map(pool)
# A study whose trailing 2 does not mean "squared" drops that affix.
concept_map(pool, affixes = setdiff(POOL_AFFIXES, "2$"))
```

pool_collinear_pairs *Candidate pairs carrying the same information under unrelated names*

Description

What this catches that pruning cannot. `concept_map` groups *transformations*, by stripping a known affix and finding the parent. Two candidates that are numerically the same information but share no affix relationship are invisible to it.

male and female are the case that got through a real screen: exact complements, both offered, and 101 of 500 replicates selected **both**. With a free `log_mu` a design holding both is singular, so those replicates were fitting a rank-deficient model and nothing in the output said so.

Usage

```
pool_collinear_pairs(data, pool, threshold = 0.99)
```

Arguments

<code>data</code>	A data frame holding the candidate columns.
<code>pool</code>	Character vector of candidate names.
<code>threshold</code>	Absolute correlation at or above which a pair is reported.

Details

Run it on the pool actually screened, after pruning. On an unpruned pool every `ln_x` correlates with its own `x` at ~ 1 and the report is a wall of expected pairs. After pruning, anything flagged is something pruning could not catch, which is exactly the set a human needs to look at.

`complement` separates the two cases a reader treats differently: an exact complement ($x + y == 1$) is a coding duplicate and one of the pair should simply not be a candidate, while a high correlation between genuinely different measurements is a judgement call about what the model can identify.

Value

A data frame with columns `var1`, `var2`, `r` and `complement`, ordered by decreasing `abs(r)`. Empty when the pool is clean, so the caller decides whether to warn, fail, or print it.

See Also

[concept_map](#), [prune_to_one_form](#)

Examples

```
d <- data.frame(male = c(1, 0, 1, 0), female = c(0, 1, 0, 1),
               age = c(60, 71, 55, 68))
pool_collinear_pairs(d, c("male", "female", "age"))
```

preflight_report	<i>Audit the analysis environment before running an analysis</i>
------------------	--

Description

Reports the version of every package a hazard-family analysis depends on. Run it on the machine that will do the work, before anything else.

Usage

```
preflight_report(extra = character(0))
```

Arguments

extra Character vector of additional package names to report.

Details

numDeriv is only a *Suggests* of TemporalHazard, so install_github() does not pull it. Its absence silently costs standard errors on any interval- or left-censored multiphase fit: the analytic multiphase Hessian declines for those statuses by design and the optimizer falls back to numDeriv; with it missing there is no third option, and vcov() returns a bare logical while rcond and pd come back NA with nothing naming the cause.

R itself is reported from R.version, not from packageVersion(), so it is dropped from the package list: a caller who names it in extra would otherwise get a second R row reporting it as not found.

Value

A data frame with columns component, found, version and notes.

Examples

```
preflight_report()
```

proc_contents	<i>Describe a dataset's variables, in the style of SAS PROC CONTENTS</i>
---------------	--

Description

Reports the variable-level metadata that SAS PROC CONTENTS prints: creation position, name, SAS type, format, and label, alongside the R class and simple completeness counts.

Usage

```
proc_contents(data, order = c("alpha", "varnum"))
```

Arguments

data	A data frame, tibble, or similar tabular object.
order	Variable ordering. "alpha" (default) sorts case-insensitively by name, matching SAS's default <i>Alphabetic List of Variables and Attributes</i> ; "varnum" keeps creation order. The num column always reports creation position regardless of sort.

Details

Reading a .sas7bdat through **haven** is lossy. Variable name, label, format .sas, and creation order survive; SAS storage LENGTH, POS (position within the observation), informat, and the dataset's created/modified timestamps do not. Those fields are omitted rather than inferred, because inferred values would look authoritative and would disagree with the source dataset whenever its LENGTH statement differed from the default.

"Survive" means preserved *when the source carries them*, not present on every variable. A SAS variable need not have a label or a format, and many do not: in one 879-variable clinical build, 865 variables carried a label and 395 carried a format .sas. An NA in format therefore reports that the source variable had no format, not that one was lost in reading. label does not behave this way — see *Value*.

The SAS Type column is two-valued, so the mapping from R is deliberately lossy: character and factor become "Char"; everything else, including logical, Date, and POSIXct, becomes "Num". A SAS date is a number carrying a date format, so type reports "Num" while class reports "Date" — keeping the disagreement visible rather than hiding it.

For a data frame with columns but zero rows — a fully filtered cohort, say — pct_missing is NaN rather than a number in 0-100, because `mean(is.na(x))` on an empty vector is $0/0$. The proportion of missing values among no values is genuinely undefined, and reporting 0 would assert that nothing is missing. Callers formatting this column for display should handle NaN explicitly.

Value

An object of class proc_contents: a list with two elements.

header One-row data frame with observations (row count), variables (column count), and label (dataset label, or NA)

variables Data frame with one row per column and the columns num (creation position), variable, type ("Num"/"Char"), format (SAS format, or NA), label, class (R class), n_unique (distinct non-NA values), and pct_missing (0-100, 1 decimal place, or NaN when data has zero rows)

label is never NA: it falls back to the variable's own name when the source carries no label, via `labelled::var_label(null_action = "fill")`. An unlabelled variable is therefore indistinguishable here from one whose label equals its name, and both are common. Callers recording this output as a durable description of a source dataset — a schema sidecar, a data dictionary kept after the source is retired — should read labels from the source attributes directly rather than from this column, or they will record a fallback as though it were the dataset's own metadata.

See Also

[proc_means](#) for numeric summaries, [data_dictionary](#) for a flattened documentation table.

Examples

```
dta <- sample_data(n = 50)
pc <- proc_contents(dta)
pc$header
pc$variables

# Creation order rather than alphabetical
proc_contents(dta, order = "varnum")$variables
```

proc_freq	<i>Frequency tables, in the style of SAS PROC FREQ</i>
-----------	--

Description

Produces the table SAS PROC FREQ prints for a TABLES statement: one row per observed combination of levels, with frequencies and percentages. One call is one table; a SAS step with several TABLES statements is several calls.

Usage

```
proc_freq(data, tables, missing = FALSE, list = FALSE, weights = NULL)
```

Arguments

data	A data frame.
tables	Character vector of one or more column names defining the table.
missing	Logical. FALSE (the SAS default) excludes rows with a missing value in any tables variable from the table and from every denominator, and reports their count in the frequency_missing attribute. TRUE, SAS / MISSING, treats missing as a level, with each SAS special missing value a level of its own.

<code>list</code>	Logical. TRUE, SAS / LIST, gives cumulative columns and percentages of the grand total for an n-way table. It has no effect on a one-way table.
<code>weights</code>	Character or NULL. Name of a single numeric column of weights, SAS WEIGHT. Frequency becomes the sum of weights. Rows with a missing weight are excluded entirely; non-positive weights are an error, as in proc_means .

Details

The last name in tables is the column variable, the one before it the row variable, and any earlier names are strata, as in SAS TABLES a*b*c.

Percentages depend on list, not only the columns shown. For a crosstab of three or more variables, SAS prints one row-by-column table per stratum and computes Percent within that stratum. With `list = TRUE`, Percent is of the grand total. The two agree for one- and two-way tables and disagree from three variables on.

Missing values sort first. SAS treats missing as the smallest value, so under `missing = TRUE` the missing level heads the table and the cumulative columns count it first. `dplyr::count()` places NA last, which moves every cumulative value.

Rows follow SAS ORDER=INTERNAL: factors in declared level order, numbers by value, character values by byte value (the C locale, so "B" sorts before "a"). Unused factor levels are omitted, as SAS omits zero-count levels without SPARSE.

A `haven_labelled` variable is grouped on its value label, as SAS forms levels from formatted values: stored codes sharing a label form one row, shown as the smallest of those codes, and a `<var>_label` column holding the label follows it. A value with no label is its own row. Variable labels on the tables columns are kept.

What counts as missing. As in SAS, a blank or whitespace-only character value is missing, and so is NaN. SAS special missing values, read by `haven` as [tagged_na](#), are separate missing levels in SAS order: `._`, then `.`, then `.A` to `.Z`.

Where this differs from SAS. Unlabelled doubles are grouped on their exact values, whereas SAS groups on the printed (BEST12.) value, so values that differ past about 12 significant digits stay separate rows here. Non-positive weights are an error, whereas SAS drops zero weights and ignores negative ones; [proc_means](#) makes the same deliberate choice.

Percentages are not rounded. SAS rounds only for display, and a rounded result would fail [compare_parity](#) against SAS output.

Tests of association (CHISQ, FISHER, MEASURES and the rest) are deliberately absent, for the same reason [proc_means](#) has no inference statistics.

Value

A data frame with one column per tables variable (each followed by a `<var>_label` column when the variable carries value labels), then:

- for a one-way table, or `list = TRUE`: Frequency, Percent, Cum_Frequency, Cum_Percent;
- for an n-way crosstab: Frequency, Percent (within stratum), Row_Percent, Col_Percent.

Frequency is integer when unweighted and double when weighted. The attribute `frequency_missing` holds the count (or weight) of excluded missing rows, and is 0 when none were excluded. A table with no remaining rows is returned with zero rows, not an error.

See Also

[proc_means](#), [proc_contents](#)

Examples

```
dta <- generate_survival_data(n = 200, seed = 42)

# proc freq; table dead / missing;
proc_freq(dta, "dead", missing = TRUE)

# proc freq; table sex * dead;
proc_freq(dta, c("sex", "dead"))

# proc freq; table sex * dead / list;
proc_freq(dta, c("sex", "dead"), list = TRUE)
```

proc_means

Summarise numeric variables, in the style of SAS PROC MEANS

Description

Produces the table SAS PROC MEANS prints: one row per analysis variable, one column per requested statistic, in the order requested.

The default statistics are **not** SAS's. PROC MEANS with no statistic list prints N, mean, standard deviation, minimum and maximum, and no percentiles. The HVTI group reports the 15th and 85th percentiles alongside the median in its descriptive summaries, rather than the quartiles, so the default here adds p15, median and p85. A SAS parity comparison must therefore name the same statistics on both sides: pass the SAS job's list to stats, or add P15 MEDIAN P85 to the SAS PROC MEANS statement.

Usage

```
proc_means(
  data,
  vars = NULL,
  class = NULL,
  stats = c("n", "mean", "std", "min", "p15", "median", "p85", "max"),
  weights = NULL
)
```

Arguments

data	A data frame, tibble, or similar tabular object.
vars	Character vector of columns to analyse. NULL (default) selects every numeric column not named in class.

class	Character vector of grouping columns, prepended to the result as leading columns. Rows with a missing value in any class variable are dropped, matching SAS's default.
stats	Character vector of SAS statistic keywords. The default, <code>c("n", "mean", "std", "min", "p15", "median", "p85", "max")</code> , is the group's descriptive summary, not SAS's own five; see the description. Counts: "n", "nmiss", "nobs", "sumwgt". Location: "mean", "median", "mode". Spread: "std", "var", "stderr", "cv", "min", "max", "range", "qrange", "q1", "q3", or any "pNN" for NN from 1 to 99. Sums: "sum", "uss", "css". Shape: "skewness", "kurtosis".
weights	Character or NULL. Name of a single numeric column of data to use as an observation weight, mirroring the SAS WEIGHT statement. Observations whose weight is missing are excluded from every statistic except nobs, which counts them, as SAS does. A zero or negative weight is an error naming the offending rows: SAS's own handling of non-positive weights varies across procedures and versions, so this fails loudly rather than encode a guess.

Details

Quantiles use `stats::quantile(type = 2)`, which is the R equivalent of SAS's default `QNTLDEF=5`. This matters: R's own default is `type = 7`, a different estimator that disagrees with SAS on the small and even-numbered samples clinical subgroups produce. For `c(1, 2, 3, 4)`, the first quartile is 1.5 in SAS and 1.75 under R's default. The median agrees, which is why the discrepancy hides.

With `vars = NULL` all numeric columns are analysed, matching SAS's behaviour when the VAR statement is omitted. Logical columns are not `is.numeric()` in R and so are excluded from that default set; naming one in `vars` coerces it to 0/1, making mean a proportion as PROC MEANS would give.

Rows are ordered by analysis variable first, then by class level. A factor class variable orders by its declared level order rather than alphabetically, matching SAS's default `ORDER=INTERNAL`; this keeps ordered clinical scales such as NYHA class in their clinical sequence.

Weights do not apply uniformly, and this follows PROC MEANS rather than being a simplification. Weighted: mean, std, var, cv, stderr, sum, uss, css, skewness, kurtosis, sumwgt. Unweighted: n, nmiss, nobs, min, max, range, mode, and every quantile – median, q1, q3, pNN and qrange. PROC MEANS does not compute weighted quantiles at all; that is PROC UNIVARIATE. So `proc_means(d, stats = "median", weights = "wt")` returns the *unweighted* median.

mode returns the smallest value among tied modes, and NA when no value repeats, except that a single observation is its own mode; all three match SAS. Weighted stderr divides the weighted standard deviation by the square root of the sum of the weights, as SAS does, not by the square root of the count. skewness and kurtosis are the adjusted Fisher-Pearson forms SAS uses, not R's naive moment ratios, and are NA for a constant column rather than NaN.

The PROC UNIVARIATE inference statistics (NORMAL, PROBN, T, PROBT, MSIGN, PROBM, SIGNRANK, PROBS) are deliberately absent: they would make this a hypothesis-testing function rather than a summary one. CLM, the confidence limits of the mean, is absent for the same reason.

cv is NA when the mean is zero, matching SAS; R's arithmetic would give Inf.

Value

A data frame with one row per analysis variable per class combination. Columns are the class variables (when supplied), then variable, label, then one column per requested statistic in the order given by stats. Count statistics (n, nmiss) are integer; all others are numeric.

See Also

[proc_contents](#) for variable metadata.

Examples

```
dta <- generate_survival_data(n = 200, seed = 42)

# The group's default statistics over every numeric variable
head(proc_means(dta))

# Named variables and an explicit statistic list
proc_means(dta, vars = c("age", "bmi"),
           stats = c("n", "mean", "median", "p15"))

# Weighted statistics
dta <- data.frame(age = c(51, 63, 47, 72), wt = c(1, 2, 1, 4))
proc_means(dta, vars = "age", stats = c("n", "sumwgt", "mean"),
           weights = "wt")
```

proc_univariate	<i>Summarise and test numeric variables, in the style of SAS PROC UNIVARIATE</i>
-----------------	--

Description

Produces the statistics data set SAS PROC UNIVARIATE writes with OUTPUT OUT=: one row per analysis variable (per class level), one column per requested statistic in the order requested, then one column per PCTLPTS= percentile. Beyond [proc_means](#) it adds weighted and decimal percentiles, VARDEF=, and the tests for location and normality.

Usage

```
proc_univariate(
  data,
  vars = NULL,
  class = NULL,
  stats = c("n", "median", "mean", "std", "cv", "min", "max"),
  weights = NULL,
  pctlpts = NULL,
  pctlpre = "p",
  mu0 = 0,
  vardef = c("df", "n", "wdf", "weight")
)
```

Arguments

data	A data frame, tibble, or similar tabular object.
vars	Character vector of columns to analyse. NULL (default) selects every numeric column not named in class or weights.
class	Character vector of grouping columns, prepended to the result as leading columns. Rows with a missing value in any class variable are dropped, and levels follow ORDER=INTERNAL, as in proc_means . Unlike <code>proc_means()</code> , a level whose every weight is missing has no row, as in SAS.
stats	Character vector of SAS statistic keywords: every keyword proc_means accepts, plus "stdmean", "t", "probt", "msign", "probm", "signrank", "probs", "normal" and "probn".
weights	Character or NULL. Name of a single numeric column of data to use as an observation weight, SAS WEIGHT. Observations whose weight is missing are excluded from every statistic except nobs, which counts them.
pctlpts	Numeric vector of percentile points in $[0, 100]$, decimals allowed, or NULL. SAS PCTLPTS=.
pctlpre	Single string prefixed to the pctlpts column names. SAS PCTLPRE=. The point follows with . replaced by _, so 2.5 gives p2_5.
mu0	Single finite number, the null value for t, msign and signrank. SAS MU0=.
vardef	The variance divisor: "df" (default), "n", "wdf" or "weight". SAS VARDEF=.

Details

Weighting. Weights apply as in [proc_means](#), except that the quantiles are weighted: median, q1, q3, qrange, pNN and the pctlpts columns. The weighted percentile sorts the values with their cumulative weights S_i ; the 0th percentile is the minimum and the 100th the maximum; otherwise, when $S_i = pW$ the result is the mean of x_i and x_{i+1} , else the first x_i with $S_i > pW$. Unweighted quantiles use `stats::quantile(type = 2)`, SAS's default PCTLDEF=5. mode stays unweighted, as in SAS. stderr and stdmean divide the standard deviation by the square root of the sum of the weights. A zero or negative weight is an error; SAS instead treats a negative weight as zero and still counts the observation.

VARDEF. The variance divisor is $n - 1$ ("df"), n ("n"), $W - 1$ ("wdf") or W ("weight"), where W is the sum of the weights (n without weights). It changes var, std and cv. Under "n", skewness and kurtosis are the moment forms rather than the adjusted ones.

Statistics SAS does not compute are NA, as SAS writes missing:

- msign, probm, signrank, probs, normal, probn when weights is given;
- t, probt, stderr and stdmean when vardef is not "df";
- skewness and kurtosis when vardef is "wdf" or "weight";
- normal and probn above 2000 observations, with one warning per call: SAS switches to a Kolmogorov D test there, which is not ported;
- std, var, cv, stderr, stdmean, t, probt at one observation, skewness below three and kurtosis below four;

- `t`, `probt`, `skewness`, `kurtosis`, `normal`, `probn` when every value is equal (`cv` is then 0 unless the mean is zero, where it is NA);
- `msign`, `probm`, `signrank`, `probs` when no value differs from μ_0 ;
- `mode` when no value repeats among two or more values.

Tests. `t` is Student's t for the mean against μ_0 ; `msign` the sign statistic; `signrank` the Wilcoxon signed rank statistic, with an exact p -value for 20 or fewer non-zero differences and SAS's t approximation above that. `normal` is the Shapiro-Wilk W , from `stats::shapiro.test()`, which implements Royston's 1995 algorithm; SAS implements Royston's 1992 one, so `normal` and `probn` agree with SAS to about 10^{-8} , not to machine precision. At two observations both are 1, as SAS reports.

Value

A data frame with one row per analysis variable per class combination. Columns are the class variables (when supplied), then `variable`, `label`, then one column per `stats` keyword in the order given, then one column per `pctlpts` point in the order given. `n`, `nmiss` and `nobs` are integer; all others are numeric.

See Also

[proc_means](#) for the descriptive statistics alone.

Examples

```
dta <- generate_survival_data(n = 200, seed = 42)

# SAS unistats default statistics over every numeric variable
head(proc_univariate(dta))

# Decimal percentiles, as for a bootstrap interval
proc_univariate(dta, vars = "age", stats = "n",
  pctlpts = c(2.5, 50, 97.5))

# Tests for location and normality
proc_univariate(dta, vars = "bmi",
  stats = c("mean", "t", "probt", "normal", "probn"),
  mu0 = 25)
```

provenance_artifact	<i>Snapshot an input artifact for provenance</i>
---------------------	--

Description

Records an existing artifact beneath the study root by canonical study-relative path, role, byte count, modification time, and SHA-256 hash.

Usage

```
provenance_artifact(path, role = "input", cfg = study_config())
```

Arguments

path	Character(1). Existing artifact path, absolute or relative to the study root.
role	Character(1). The artifact's role in this job.
cfg	List. A study manifest from study_config .

Value

A plain list containing one immutable-by-convention artifact record.

See Also

[provenance_data](#), [capture_provenance](#)

Examples

```
root <- file.path(tempdir(), "provenance-artifact-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
  file.path(root, "00_datasets", "built.csv"),
  row.names = FALSE)
invisible(register_data(root, built = "built.csv"))
cfg <- study_config(root)

writeLines("x <- 1", file.path(root, "30_analyses", "fit.R"))
str(provenance_artifact("30_analyses/fit.R", role = "code", cfg = cfg))
unlink(root, recursive = TRUE)
```

provenance_data

Snapshot a registered data file for provenance

Description

Records the authoritative physical file selected by dataset. For a manifest entry with role: "primary", this is the promoted Parquet file rather than the retired source, and for a dataset registered with [register_data](#) it is the registered version, the dated Parquet file, rather than the source it was converted from. The returned plain list contains the logical dataset name, a canonical study-relative path, its role, byte count, modification time, and SHA-256 hash.

Usage

```
provenance_data(
  dataset = "study",
  cfg = study_config(),
  role = "analysis",
  working = .hvti_working_default()
)
```

Arguments

dataset	Character(1). Logical registered dataset name. "built" is a second name for "study".
cfg	List. A study manifest from study_config .
role	Character(1). The file's role in this job.
working	Logical. If TRUE, record the working source file, as read_built (working = TRUE) reads it.

Value

A plain list containing one immutable-by-convention data record.

See Also

[provenance_artifact](#), [capture_provenance](#)

Examples

```
root <- file.path(tempdir(), "provenance-data-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
  file.path(root, "00_datasets", "built.csv"),
  row.names = FALSE)
invisible(register_data(root, built = "built.csv"))
cfg <- study_config(root)

# The registered Parquet file, not the CSV it was converted from.
str(provenance_data("study", cfg = cfg))
unlink(root, recursive = TRUE)
```

provenance_path	<i>Name the provenance sidecar for an output</i>
-----------------	--

Description

Returns the path of the sidecar belonging to a rendered output: the output path with its extension replaced by `.provenance.json`.

Usage

```
provenance_path(path)
```

Arguments

path Character(1). Path to a rendered output.

Value

Character(1). The sidecar path.

See Also

[publish_provenance](#), [record_provenance](#)

Examples

```
provenance_path("_output/death-hz-ac.html")
```

prune_to_one_form	<i>Prune a candidate pool to one form per concept</i>
-------------------	---

Description

Prune a candidate pool to one form per concept

Usage

```
prune_to_one_form(pool, ...)
```

Arguments

pool Character vector of candidate names.
 ... Passed to [concept_map](#).

Value

The kept names, in the pool's original order, so the scope handed to a screen stays stable across runs.

See Also

[concept_map](#), [pool_collinear_pairs](#)

Examples

```
prune_to_one_form(c("age", "ln_age", "age2", "bsa"))
```

publish_provenance	<i>Publish captured provenance beside a completed output</i>
--------------------	--

Description

Requires an existing regular output, binds its byte count and SHA-256 hash to a previously captured payload, and writes the JSON sidecar through a same-directory temporary file and rename. Capture-session facts are not recomputed during publication.

Usage

```
publish_provenance(path, payload)
```

Arguments

path	Character(1). Existing completed output path.
payload	A payload returned by capture_provenance .

Value

Invisibly, the published record as a list.

See Also

[capture_provenance](#), [record_provenance](#), [provenance_path](#)

Examples

```
root <- file.path(tempdir(), "publish-provenance-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
  file.path(root, "00_datasets", "built.csv"),
  row.names = FALSE)
invisible(register_data(root, built = "built.csv"))
cfg <- study_config(root)

payload <- capture_provenance("death-hz-ac",
  data = list(provenance_data(cfg = cfg)),
  cfg = cfg)

# Publish only once the output exists.
out <- file.path(root, "40_graphs", "death-hz-ac.html")
writeLines("<p>report</p>", out)
invisible(publish_provenance(out, payload))
file.exists(provenance_path(out))
unlink(root, recursive = TRUE)
```

r_data_types

*Automatically infer and convert data types***Description**

Intelligently converts column types in a dataset based on their content. Handles character-to-factor conversion, binary numeric variables, and various NA representations. Preserves variable labels from SAS/labelled data.

Usage

```
r_data_types(
  dataset,
  factor_size = 10,
  skip_vars = NULL,
  binary_factor = FALSE,
  use_value_labels = FALSE
)
```

Arguments

dataset	A data frame, tibble, data.table, or similar tabular object
factor_size	Integer threshold for factor conversion. Numeric variables with fewer than this many unique values (but more than 2) will be converted to factors. Must be between 2 and 50. Default is 10.
skip_vars	Character vector of column names to exclude from conversion. These columns will remain unchanged. Default is NULL (convert all columns).
binary_factor	Logical. If TRUE, binary variables are converted to factors instead of logical. Default is FALSE (convert to logical).
use_value_labels	Logical. If TRUE, a column carrying value labels – what haven reads from a SAS numeric-plus-format variable – is converted through to_factor , so the level text is kept. If FALSE the value labels are dropped and the numeric codes are converted instead. Default is FALSE, which warns once per session; the default becomes TRUE in a later release.

Details

Each column is converted by the first rule that matches, in this order:

1. When `use_value_labels = TRUE` and the column carries value labels, they become the factor levels. This runs ahead of every rule below: a declared type is not subject to a threshold on distinct values.
2. Character strings "NA", "na", "Na" and "nA" become NA.

3. Numeric or integer columns with exactly 2 distinct values become logical, or factors when `binary_factor = TRUE` – but only when the two values are 0 and 1. Any other 2-distinct-value coding (a SAS-style 1/2, say) becomes a factor outright, regardless of `binary_factor`: `as.logical()` maps every nonzero code to `TRUE`, so a 1/2 column would otherwise become `TRUE` for every row and silently lose one of its two categories.
4. Remaining character columns become factors.
5. Numeric columns with 3 to `factor_size` distinct values become factors.
6. Logical columns become factors when `binary_factor = TRUE`.

Date, POSIXct, and POSIXlt columns are never altered by type conversion.

Which rule fired on which column is recorded on the result and read back with [type_conversion_report](#).

Value

An object of the same class as `dataset` with columns converted according to the function's rules.

Variable labels are preserved. The result carries a per-column conversion report, read with [type_conversion_report](#).

See Also

[type_conversion_report](#) for the record of which rule fired on which column.

Examples

```
# Basic usage with sample data
dta <- sample_data(n = 100)
str(dta) # Original types
dta_converted <- r_data_types(dta, use_value_labels = FALSE)
str(dta_converted) # Converted types

# Real data example with mtcars
str(datasets::mtcars$vs) # numeric (0/1)
mtcars_converted <- r_data_types(datasets::mtcars,
                                use_value_labels = FALSE)
str(mtcars_converted$vs) # logical (FALSE/TRUE)

# Skip specific columns
mtcars_partial <- r_data_types(datasets::mtcars,
                              skip_vars = c("vs", "am"),
                              use_value_labels = FALSE)
str(mtcars_partial$vs) # Still numeric (unchanged)

# Control factor creation threshold
mtcars_strict <- r_data_types(datasets::mtcars, factor_size = 5,
                             use_value_labels = FALSE)

# Keep binary variables as factors
mtcars_factors <- r_data_types(datasets::mtcars, binary_factor = TRUE,
                              use_value_labels = FALSE)
str(mtcars_factors$vs) # Factor instead of logical

# Keep the level text of a SAS formatted variable
```

```

disp <- haven::labelled(c(1, 2, 1, 3),
                        labels = c(Home = 1, Rehab = 2, SNF = 3),
                        label = "Discharge disposition")
converted <- r_data_types(data.frame(disp = disp), use_value_labels = TRUE)
levels(converted$disp)
type_conversion_report(converted)

```

r_dir_impurities	<i>Report top-level executable code in an R directory</i>
------------------	---

Description

Parses every .R file in `dir` and reports any top-level expression that is not an assignment. Function definitions and constants pass; calls do not.

Use this on any directory that is sourced wholesale, where a stray call would execute on every render.

Usage

```
r_dir_impurities(dir)
```

Arguments

`dir` Character(1). Directory to check.

Value

A character vector of complaints, one per offending expression, empty when the directory is clean. The caller decides whether to warn or to fail.

Examples

```

d <- file.path(tempdir(), "purity-example")
dir.create(d, showWarnings = FALSE)
writeLines(c("f <- function(x) x + 1", "print(f(1))"),
           file.path(d, "example.R"))
r_dir_impurities(d)
unlink(d, recursive = TRUE)

```

read_built*Read the study's built dataset*

Description

Reads the dataset named in `_study.yml` and normalises its types so that both available read paths deliver the same frame.

For a dataset registered with [register_data](#), the registered version (a dated parquet) is read, after its checksum is checked. If the source file has been rebuilt since, the registered version is still read and a message of class `hvtiRutilities_source_changed` says so and names [update_manifest\(\)](#).

A combined dataset (see [register_data](#)) is still read when a dataset it was built from has been updated since. A message of class `hvtiRutilities_parent_changed`, which inherits `hvtiRutilities_out_of_date`, names the parents' recorded and current versions and the commands that update it. A combined dataset with no recorded parent versions is reported the same way, as unrecorded, until it is rebuilt and registered again.

Only a dated version is registered data. Any other read is a working read: it is allowed only in the interactive console and in jobs in the study's datasets or descriptive folder, and it signals a message of class `hvtiRutilities_working_read`. Review a rebuilt source with [review_built\(\)](#) and register it with [update_manifest\(\)](#). The release branch below is deprecated.

For a release-aware contract, the pinned release is verified before cache access. A later valid release emits a message of class `hvtiRutilities_update_available`; an unavailable catalog or invalid candidate emits `hvtiRutilities_update_status_unknown` and the pinned data are still read. A changed pinned release errors with class `hvtiRutilities_release_integrity`. A withdrawn pin errors with class `hvtiRutilities_withdrawn_release` unless `allow_withdrawn` is `TRUE`.

The normalisation is not cosmetic. [read_clinical_data](#) converts SAS 0/1 numerics to logical while `haven::read_sas()` leaves them numeric, and downstream modelling code rejects a logical status vector outright — so without this the same document would run under one read path and fail under the other. Labelled vectors are likewise reduced to plain vectors, keeping the SAS variable label as an attribute because listings print labels rather than names.

Usage

```
read_built(  
  cfg = study_config(),  
  refresh = FALSE,  
  dataset = "study",  
  allow_withdrawn = FALSE,  
  working = .hvti_working_default()  
)
```

Arguments

`cfg` List. A study manifest from [study_config](#).

refresh	Logical. If TRUE, force a re-read from the source and a reconversion regardless of the manifest's role or cached stamp. "The source changed" is not always something a timestamp can express – a rebuild that preserves mtime, a restored backup, a correction applied out of band. Errors if the manifest entry has role: "primary": that role means the source has been retired and the parquet is authoritative, so there is nothing to refresh from. It also errors for a dataset registered as a dated parquet, and names <code>update_manifest()</code> , which registers a rebuilt source.
dataset	Character(1). Logical dataset name. Defaults to "study". "built" is a second name for "study".
allow_withdrawn	Deprecated. A dated version is never withdrawn, so there is nothing to allow. Logical. If TRUE, allow a deliberately pinned withdrawn release to be read for revision work. The default is FALSE.
working	Logical. If TRUE, read the working dataset as it stands rather than its registered version, writing nothing. Defaults to TRUE when the environment variable HVTI_DATA is "working". Any read that is not of a dated version is a working read: it is allowed only in the interactive console and in jobs in the study's datasets or descriptive folder, and signals a message of class <code>hvtiRutilities_working_read</code> .

Value

A data frame with lower-cased names, no logical columns and no `haven_labelled` columns.

See Also

`built_manifest`, `assert_cohort`

Examples

```
if (requireNamespace("arrow", quietly = TRUE)) {
  root <- file.path(tempdir(), "read-built-example")
  suppressMessages(study_setup(root, "Example", 1L, renv = FALSE))
  write.csv(data.frame(DEAD = c(1, 0, 0), IV_DEAD = 1:3),
            file.path(study_dir("datasets", root), "example.csv"), row.names = FALSE)
  suppressMessages(register_data(root, "example.csv"))
  names(read_built(study_config(root)))
  unlink(root, recursive = TRUE)
}
```

Description

A convenience wrapper that detects the file type, reads the data with the appropriate reader, and optionally runs `r_data_types` to convert column types. This saves novice users from having to remember which package reads which format and ensures labels are preserved.

Supported formats:

- `.sas7bdat` SAS datasets via `haven::read_sas()`
- `.csv` Comma-separated files via `utils::read.csv()` with `check.names = FALSE`, so column names are preserved exactly as written in the file (spaces, hyphens, and special characters are not silently converted to `.`).
- `.xlsx`, `.xls` Excel workbooks via `readxl::read_excel()`
- `.rds` R serialized objects via `readRDS()`
- `.parquet` Parquet files via `arrow::read_parquet()`; needs the **arrow** package

Usage

```
read_clinical_data(file, convert_types = FALSE, ..., catalog_file = NULL)
```

Arguments

- | | |
|----------------------------|--|
| <code>file</code> | Character. Path to the dataset file. |
| <code>convert_types</code> | Logical. Apply <code>r_data_types</code> to the data after reading. Defaults to <code>FALSE</code> : the file is returned as read. <code>TRUE</code> converts any two-valued numeric column to logical, which is wrong for 0/1 event and censoring flags, so type conversion belongs to a declared variable-derivation step rather than to reading. |
| <code>...</code> | Additional arguments passed to <code>r_data_types</code> (e.g., <code>factor_size</code> , <code>skip_vars</code> , <code>binary_factor</code> , <code>use_value_labels</code>). Ignored when <code>convert_types = FALSE</code> . |
| <code>catalog_file</code> | Character. Path to a SAS format catalog (<code>.sas7bcat</code>), passed to <code>haven::read_sas()</code> . Defaults to <code>NULL</code> , which is what was read before this argument existed, so no existing call changes. Only valid for a <code>.sas7bdat</code> data file; supplying one with any other format is an error rather than a silent no-op. Must be named, because it follows <code>...</code> . |

Details

SAS format catalogs. A `.sas7bdat` file stores a format's *name* – `YESNOF.` – and not its values. The code-to-text mapping lives in a separate `.sas7bcat` catalog. Without one, reading a SAS dataset yields numeric codes and no value labels, however the read is written, and no amount of downstream work can recover text that never arrived. Pass `catalog_file` to read the two together.

A catalog on its own is only half the journey. With `convert_types = TRUE`, `r_data_types` discards value labels unless `use_value_labels = TRUE` is passed through `...`, so supplying a catalog without it warns.

Value

A data frame with labels preserved and (optionally) types converted.

See Also

[r_data_types](#) for details on type conversion, [label_map](#) to extract labels after reading.

Examples

```
# Read a CSV
tmp <- tempfile(fileext = ".csv")
write.csv(mtcars, tmp, row.names = FALSE)
dta <- read_clinical_data(tmp, convert_types = FALSE)
str(dta[, 1:5])
unlink(tmp)

# Read without type conversion
tmp <- tempfile(fileext = ".csv")
write.csv(mtcars, tmp, row.names = FALSE)
dta_raw <- read_clinical_data(tmp, convert_types = FALSE)
str(dta_raw[, 1:5])
unlink(tmp)

# Read an RDS file
tmp <- tempfile(fileext = ".rds")
saveRDS(iris, tmp)
dta <- read_clinical_data(tmp, convert_types = TRUE, factor_size = 5)
str(dta)
unlink(tmp)
```

record_provenance

Capture and publish provenance for an existing output

Description

Convenience wrapper for an output that already exists. It captures an explicit provenance payload and immediately publishes it beside the output.

Usage

```
record_provenance(
  path,
  data,
  artifacts = list(),
  extra = list(),
  cfg = study_config()
)
```

Arguments

path Character(1). Existing completed output path.

data List of records returned by [provenance_data](#).

artifacts	List of records returned by provenance_artifact .
extra	Named list of job-specific fields. Reserved fields cannot be displaced.
cfg	List. A study manifest from study_config .

Value

Invisibly, the published record as a list.

See Also

[capture_provenance](#), [publish_provenance](#), [provenance_path](#)

Examples

```
root <- file.path(tempdir(), "record-provenance-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
  file.path(root, "00_datasets", "built.csv"),
  row.names = FALSE)
invisible(register_data(root, built = "built.csv"))
cfg <- study_config(root)

out <- file.path(root, "40_graphs", "death-hz-ac.html")
writeLines("<p>report</p>", out)
rec <- record_provenance(out, data = list(provenance_data(cfg = cfg)),
  cfg = cfg)

rec$output$file
file.exists(provenance_path(out))
unlink(root, recursive = TRUE)
```

register_data	<i>Register a study dataset</i>
---------------	---------------------------------

Description

Completes the default study data contract or adds one distinctly named dataset. The function derives row and column counts from the file and replaces `_study.yml` and `manifest.yaml` only after both updated files have been prepared successfully.

Registration converts the file once to a dated parquet in the same folder, `<name>_YYYYMMDD.parquet`, with its column record beside it as `<name>_YYYYMMDD.schema.csv`. That parquet is what [read_built](#) and the job templates read, so the source file may be rebuilt freely; run [update_manifest\(\)](#) to register a rebuilt file as a new version. The date is the file's modification date unless `extract_date` is given. Conversion needs the **arrow** package. A release-aware registration (`catalog_dataset` and `release_id`) records the catalog's file as it is and converts nothing.

A dataset is registered once: registering it again stops with "already registered". So a study whose datasets are already registered adds `kind`, `key` or `parents` by editing that dataset's entry in `_study.yml`,

then running `update_manifest()`. The key is checked when the next version is registered. A combined dataset records its parents' versions only once its source has been rebuilt, and reads as out of date until then. A release-aware dataset is skipped by `update_manifest()`: its parents' versions are recorded when a release is adopted with `adopt_data_update()`, and its key is checked only at registration.

Usage

```
register_data(
  root = getwd(),
  built,
  dataset = "study",
  role = c("study", "named"),
  population = NULL,
  source = NULL,
  extract_date = NULL,
  catalog_dataset = NULL,
  release_id = NULL,
  kind = NULL,
  key = NULL,
  parents = NULL
)
```

Arguments

<code>root</code>	Character. Study root or a directory beneath it.
<code>built</code>	Character(1). Dataset filename within the logical datasets directory, including its extension.
<code>dataset</code>	Character(1). The dataset's name. Choose it only with <code>role = "named"</code> ; <code>role = "study"</code> requires the default, <code>dataset = "study"</code> . A name starts with a lowercase letter and uses only lowercase letters, digits and underscores. "study" and "built" are reserved, "built" being a second name for the study dataset. Read the dataset back with <code>read_built(dataset = "<name>")</code> .
<code>role</code>	Character(1). Takes one value, which picks the slot the dataset fills. "study", the default when the argument is left out, registers the study's single main dataset. It is recorded under <code>built:</code> in <code>_study.yml</code> and read by <code>read_built()</code> with no arguments. A study has exactly one, so registering it a second time stops with "already registered". "named" adds another dataset under the name given in <code>dataset</code> . It is recorded under <code>additional_datasets:</code> and read with <code>read_built(dataset = "<name>")</code> .
<code>population</code>	Character(1) or NULL. Population description.
<code>source</code>	Character(1) or NULL. Data-source description.
<code>extract_date</code>	Character, Date, or NULL. Extraction date. The file modification date is used when omitted.
<code>catalog_dataset, release_id</code>	Deprecated. Catalog releases are retired; use a build record sidecar and plain <code>register_data()</code> . Character(1) or NULL. Producer catalog dataset ID and exact

	published release ID. Supply both to make the study contract release-aware, or neither for a legacy registration. A dataset already registered as dated versions cannot be moved to a release this way: that would drop every registered version.
kind	Character(1) or NULL. What the dataset is: "built" (the study dataset), "subset", "ancillary" (many rows per patient, such as echoes or labs, joined to the cohort by a job) or "combined" (built by joining others).
key	Character or NULL. The columns that make each row unique, such as c("ccfid", "echo_date"). Checked at registration, which stops if any row repeats on it. Jobs use it unless they set their own.
parents	Character or NULL. For kind = "combined" only: the registered datasets it was built from. Their current versions are recorded, so a later update to a parent marks this dataset out of date. "built" is a second name for "study".

Value

An object of class "study_status", returned visibly.

Where a subset goes

A subset of the study dataset has three homes, and which to pick depends on who needs it.

- A WHERE condition in one job writes no file. The job's `hvtiRtemplates::read_job_data()` applies the condition and records it. Use this for a subset that only one job analyses.
- An analysis set from `hvtiRdatabuild::write_analysis_set()` is saved as `00_datasets/<name>.parquet` with `<name>.set.yml` beside it, and is derived from the study dataset only. A job reads it with `DATASET <- "built"` and `ANALYSIS_SET <- "<name>"`. Use this for a subset of the study dataset that several jobs share.
- A separately registered dataset, role = "named" with kind = "subset", is for a subset built some other way, such as from a second source or by a script outside the study's build.

Exclusions that define the cohort itself belong in the build job (`hvtiRtemplates::build_cohort()`), not in a subset.

See Also

[study_setup](#), [study_config](#), [update_manifest](#)

Examples

```
root <- file.path(tempdir(), "register-data-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
  file.path(root, "00_datasets", "built.csv"),
  row.names = FALSE)
register_data(root, built = "built.csv", key = "ccfid")
# The CSV is kept; the registered version is a dated Parquet file.
list.files(file.path(root, "00_datasets"))
unlink(root, recursive = TRUE)
```

```
# An additional dataset is registered with role = "named", under a name of
# its own, and read back by that name.
root <- file.path(tempdir(), "register-data-named-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
  file.path(root, "00_datasets", "built.csv"),
  row.names = FALSE)
invisible(register_data(root, built = "built.csv", key = "ccfid"))
utils::write.csv(
  data.frame(ccfid = c(1, 1, 2), echo_date = c("2020-01-05", "2020-06-01", "2020-02-10"),
    lvef = c(55, 50, 60)),
  file.path(root, "00_datasets", "echo.csv"),
  row.names = FALSE
)
invisible(register_data(root, built = "echo.csv", dataset = "echo", role = "named",
  kind = "ancillary", key = c("ccfid", "echo_date")))
read_built(study_config(root), dataset = "echo")
unlink(root, recursive = TRUE)
```

review_built

Review a rebuilt study dataset before registering it

Description

Compares the working dataset, as it stands in the study's datasets folder, with its registered version, the dated parquet jobs read. It reads both and writes nothing. Run it after a rebuild and before [update_manifest\(\)](#), which registers the rebuilt file as the next version. It describes how the data changed; it does not certify that the new file is analytically or clinically correct.

Usage

```
review_built(cfg = study_config(), dataset = "study")
```

Arguments

cfg	List. A study manifest from study_config .
dataset	Character(1). Logical dataset name. Defaults to "study". "built" is a second name for "study".

Value

An object of class "built_review" with

dataset The logical dataset name.

registered The current version record from `manifest.yaml`.

working A list with the source file's sha256, n_rows and n_cols.

changed TRUE when the source differs from the registered version.

comparison A [compare_datasets](#) result, or NULL when unchanged.

See Also[update_manifest](#), [read_built](#)**Examples**

```

if (requireNamespace("arrow", quietly = TRUE)) {
  root <- file.path(tempdir(), "review-built-example")
  suppressMessages(study_setup(root, "Example", 1L, renv = FALSE))
  path <- file.path(study_dir("datasets", root), "example.csv")
  write.csv(data.frame(DEAD = c(1, 0, 0), IV_DEAD = 1:3), path, row.names = FALSE)
  suppressMessages(register_data(root, "example.csv"))
  write.csv(data.frame(DEAD = c(1, 0, 0, 1), IV_DEAD = 1:4), path, row.names = FALSE)
  review_built(study_config(root))
  unlink(root, recursive = TRUE)
}

```

review_data_update	<i>Review one published dataset release</i>
--------------------	---

Description

Deprecated. Catalog releases are retired; use [review_built](#).

Verifies the pinned release and one exact, newer candidate, then compares their structure. Review reads the source files directly; it does not write caches or change either study manifest. It describes data drift, but it does not certify that a candidate is analytically or clinically correct.

Usage

```
review_data_update(cfg = study_config(), dataset = "study", release_id)
```

Arguments

<code>cfg</code>	List. A study manifest from study_config .
<code>dataset</code>	Character(1). Logical dataset name. Defaults to "study". "built" is a second name for "study".
<code>release_id</code>	Character(1). Exact candidate release ID. The value "latest" is not accepted as an alias.

Value

An object of class "data_update_review" with

dataset The logical study dataset name.

pinned,candidate The catalog records for both releases.

comparison A [compare_datasets](#) result.

See Also

[check_data_updates](#), [adopt_data_update](#)

Examples

```
root <- file.path(tempdir(), "review-data-update-example")
invisible(study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE))
utils::write.csv(data.frame(ccfid = 1:3, age = c(61, 70, 55)),
  file.path(root, "00_datasets", "built.csv"),
  row.names = FALSE)
invisible(register_data(root, built = "built.csv"))
cfg <- study_config(root)

# Deprecated: it warns and names review_built() instead, then stops,
# because this study is not pinned to a catalog release.
try(review_data_update(cfg, release_id = "surgery_cohort-20260921-r1"))
unlink(root, recursive = TRUE)
```

sample_data

Generate a sample dataset for testing

Description

Creates a data frame with labeled columns of various types, suitable for demonstrating [r_data_types](#) and [label_map](#).

Usage

```
sample_data(n = 100)
```

Arguments

n Number of records to generate. Default 100.

Value

A data frame with n rows and 7 labeled columns:

id Integer sequence (Patient Identifier)

boolean Integer 1/2 (Binary Indicator)

logical Character "F"/"T" (Logical Status)

f_real Uniform random values (Random Uniform Value)

float Normal random values (Random Normal Value)

char Character "male"/"female" (Gender)

factor Factor C1-C5 (Category Group)

Examples

```
# Create and inspect labeled data
dta <- sample_data(n = 20)
str(dta)
label_map(dta)

# Full workflow: generate, convert types, extract labels
dta <- sample_data(n = 100)
dta_clean <- r_data_types(dta, skip_vars = "id")
lmap <- label_map(dta_clean)
print(lmap)
```

sas_macro_defs

*Extract every macro definition from a SAS file***Description**

Scans a ‘.sas’ file and returns one row for each ‘ contains. SAS files in the legacy CORR library are macro *collections*, not single macros, so a file routinely defines several.

Usage

```
sas_macro_defs(file)
```

Arguments

file Character. Path to a ‘.sas’ file.

Details

Matching is case-insensitive: SAS macro names are case-insensitive and the legacy corpus mixes ‘ definitions are handled by depth counting, so an inner ‘ terminate an outer macro.

The body hash is a SHA-256 digest of the definition with case folded and whitespace runs collapsed. Two definitions with the same hash are semantically identical up to formatting.

An unmatched ‘ containing no ‘

Value

A data.frame with one row per macro definition and columns file, macro, params, body_hash, line_start, and line_end. Macro and parameter names are lowercased.

Examples

```
f <- system.file("extdata", "example_macro.sas", package = "hvtiRutilities")
if (nzchar(f)) sas_macro_defs(f)
```

sas_macro_signature	<i>Parse the documentation header of a legacy SAS macro file</i>
---------------------	--

Description

Legacy CORR macros carry a structured comment block naming the macro, its purpose, its documented call signature, and a modification log. This function extracts those fields. They are used as evidence during human review of divergent macro redefinitions, and the documented call is a direct input to the Phase 1 SAS harness.

Usage

```
sas_macro_signature(file)
```

Arguments

file	Character. Path to a '.sas' file.
------	-----------------------------------

Details

All fields are optional. A file with no header block yields a row of NAs rather than an error, because an undocumented macro is a normal occurrence in this corpus and not a defect.

When several MODIFIED BY lines are present, the **last** date is returned. Note that this date is advisory evidence only; `sas_triage()` never uses it to choose between competing macro definitions.

Value

A one-row `data.frame` with columns `file`, `macro_name`, `short_desc`, `created_on`, `modified_on`, and `documented_call`. Absent fields are `NA_character_`. `macro_name` is lowercased.

Examples

```
f <- system.file("extdata", "example_macro.sas", package = "hvtiRutilities")
if (nzchar(f)) sas_macro_signature(f)
```

sas_path	<i>Build a path under the study root</i>
----------	--

Description

Joins its arguments onto the study root. A first component naming a logical study directory is resolved through the study's numbered or legacy layout. Other components are passed through unchanged. Use this instead of any literal path: the same study is mounted at different absolute paths on the analysis server and on a laptop.

Usage

```
sas_path(..., start = getwd())
```

Arguments

- ... Character. Path components, passed to file.path().
- start Character. Directory to start the upward walk from. Defaults to getwd().

Value

Character(1). The joined path. It is not checked for existence.

See Also

[study_root](#)

Examples

```
root <- file.path(tempdir(), "sas-path-example")
dir.create(file.path(root, "datasets"), recursive = TRUE,
           showWarnings = FALSE)
yaml::write_yaml(
  list(study = "Example", built = "example.sas7bdat"),
  file.path(root, "_study.yml")
)
sas_path("datasets", start = root)
unlink(root, recursive = TRUE)
```

sas_triage	<i>Triage a directory of legacy SAS macro files</i>
------------	---

Description

Applies an ordered rule ladder to every SAS source file in dir and returns one row per macro definition, each carrying a decision and the evidence that justifies it.

Usage

```
sas_triage(dir, overrides = NULL)
```

Arguments

- dir Character. Path to a directory of SAS source files. Use a local clone; the canonical library lives on a network volume where file operations are slow.
- overrides Character or NULL. Path to a 'macro_overrides.yaml' recording human decisions. Each entry requires macro, canonical_file, rationale, decided_by, and decided_on.

Details

Only the top level of `dir` is scanned; subdirectories are ignored.

Candidate files are selected by excluding known non-source suffixes (logs, listings, documents, binary data sets, numbered RCS backups) rather than by matching `.sas`. The library names files with dots as word separators – `deciles.hazard` and `lm.cprobs` are names, not stems with extensions – and many macro files carry no extension at all, so a `.sas-only` pattern silently omits them.

File-level rules, applied first, in order:

1. Copy of `*` – drop, as a filename-prefix duplicate.
2. `*.sas~` – drop, as an editor backup superseded by construction.
3. Fails `.sas_lint()` – drop, citing the specific lint failure.

Definition-level rules, applied per macro name across surviving files:

1. [4.] Defined in exactly one file – canonical.
2. [5.] Defined in several files, all bodies hashing identically – canonical, recording every defining file.
3. [6.] Defined in several files with two or more distinct bodies – **ambiguous**. No canonical file is chosen.

Rule 6 never auto-resolves. Modification dates and visibility are attached as evidence for a human, never consumed as a tiebreaker, because a tool that silently picks a winner launders a coin-flip into a committed artifact. Filesystem `mtime` is deliberately never consulted: the corpus froze in 2019 and lives on a network volume where timestamps do not survive copies.

Ambiguity is resolved by a human writing an entry into overrides and re-running. This makes the result reproducible and auditable.

Value

A `data.frame` with one row per macro definition and columns `file`, `macro`, `params`, `body_hash`, `line_start`, `line_end`, `visibility`, `decision`, `rule`, and `evidence`. Rows are sorted by macro then file, so the result is deterministic.

Examples

```
d <- system.file("extdata", "macros", package = "hvtiRutilities")
if (nzchar(d)) sas_triage(d)
```

sas_variable_block	<i>Read a variable list out of a SAS job</i>
--------------------	--

Description

The SAS jobs this package's callers reproduce specify their variables as blocks in the .sas: a phase name on its own line, then comma-separated names, terminated by a bare ;.

Reading them rather than transcribing them is the point. A transcribed list drifts from the job it claims to reproduce, and nothing catches it.

Usage

```
sas_variable_block(lines, marker, after = NULL, what = "block")
```

Arguments

lines	Character vector: the .sas file, as from readLines().
marker	Regular expression matching the line that opens the block, e.g. "^\\s*early\\s*\$".
after	Optional regular expression. When given, the search starts at the first line matching it, so a block can be located inside one particular macro rather than the first one in the file.
what	Character label for the block, used in error messages.

Details

Comment handling is the whole difficulty, and getting it wrong is silent in both directions. Measured on a real screen (bh.dead_s3_JR.sas):

- Stripping /*...*/ line by line leaves banner comments intact, because /***** Patient Variables *****/ contains * and so does not match a [*]* body. The banner text then glues onto the **first** name on the next line and that name vanishes: female, afib_pr, plvidd and size were all lost this way.
- A /* ... */ spanning two lines is never stripped at all, so a commented-**out** variable is read as live: avet_con entered a screen that way.

Collapsing the block first and stripping lazily handles both, and that is what this function does.

Three name forms are normalised. name=value appears where a job carries starting or converged estimates; only the name is taken, because a job's converged values are its answer for its own study and must not be inherited by a copy of it. name/I marks a variable forced into the model, and the suffix is an option on the name rather than part of it. Names are lowercased, because SAS variable names are case-insensitive and [read_clinical_data](#) lowercases them.

Value

A character vector of unique variable names, lowercased, in the order they appear in the block.

See Also

[sas_path](#), [covariate_audit](#)

Examples

```
sas <- c("%macro final;", "  early", "    age, ln_age, /* dropped */",
        "    bsa=0.31", "  ;", "%mend;")
sas_variable_block(sas, "^\\s*early\\s*$", after = "^\\s*%macro\\s+final")
```

selection_crowding	<i>Concepts represented more than once among selected covariates</i>
--------------------	--

Description

[concept_map](#) and [prune_to_one_form](#) act on a candidate pool **before** a screen. This is the after-the-fact view, and it is the only one available for a job run without pruning – which is the case for any faithful reproduction of a SAS %macro model, since SAS had no notion of a concept.

Usage

```
selection_crowding(selected, aliases = character(0), ...)
```

Arguments

selected	Character vector of phase-qualified selected parameter names.
aliases	Named character vector, as for concept_of . Only those whose target was selected in a given phase are forwarded, because concept_map refuses an alias pointing outside its pool – right for a <i>candidate</i> pool, wrong here, where a parent form may legitimately be absent (<i>in_arin</i> can be selected while <i>area_int</i> is not).
...	Passed to concept_map .

Details

Why it is worth reporting. A step budget is shared: *max_steps* caps total steps across both phases. So a slot spent on a second form of a concept already in the model is a slot no other variable could have. When the selection also stopped *on* the cap, the two facts together say the model is not merely budget-limited but budget-limited **by redundancy** – and neither fact is visible in a coefficient table.

Grouping is [concept_map](#)'s, so it is conservative by construction. Under-grouping only understates crowding; over-grouping would assert two clinical concepts are one. This errs the safe way, and the result is a **floor**: a concept nobody listed is still counted as separate variables.

selected is phase-qualified, as `setdiff(names(coef(sel)),names(coef(base)))` returns it: "late.age", "early.zexp2". Phases are counted separately – one form of age in each phase is two slots but not a crowded concept.

Value

A data frame with columns phase, concept, n_forms and forms, ordered by phase then decreasing n_forms. Empty when nothing is crowded.

See Also

[concept_map](#), [prune_to_one_form](#)

Examples

```
selection_crowding(c("early.age", "early.ln_age", "late.bsa"))
```

strip_level_prefix	<i>Strip a leading code prefix from level text</i>
--------------------	--

Description

Removes a leading integer code and its separator from the text of a variable's levels, so "1. Yes" prints as "Yes". It is a display helper: it returns new text and **never** rewrites the levels of the object it was given.

Usage

```
strip_level_prefix(x)
```

Arguments

x	A character vector of level text, or a factor. Given a factor, the function reads levels(x) and returns one element per level - it does not return one element per observation, and it does not modify x.
---	---

Details

A separator is required. A leading integer is removed only when one of . : =) - follows it, with optional whitespace either side. A bare digit followed by a space is left alone.

That restraint is the point. A rule general enough to turn "0 No" into "No" also turns "1 vessel disease", "2 vessel" and "3 vessel" into "vessel disease", "vessel" and "vessel" - three distinct levels becoming two identical strings. Nothing errors; a downstream table() or model merges them and the level count silently stops matching the data dictionary. The failure presents as data rather than as a bug.

So the rule is biased: **a missed strip is visible in the output and fixable by declaring the level text in value_labels.yml (see [apply_value_labels](#)); a wrong strip is silent and corrupts the level set.** Two consequences follow.

- Unpunctuated "0 No" and "1 Yes" are **not** stripped and still print with their prefix.

- Text whose remainder begins with a digit is not stripped either, because "1-2 vessels" matches leading-integer-then-hyphen and would otherwise become "2 vessels", colliding with a real "2 vessels" level. Ranges are ordinary in this domain.

Collisions revert rather than merge. If stripping would bring two *different* texts together - two codes carrying the same words, or a stripped entry landing on one that was already bare - every entry involved keeps its original text and a warning names the text they collided on. The level set is preserved whatever the input, which is the guarantee this function is worth having for.

Repeated copies of the same text are not a collision. Two occurrences of "1. Yes" both become "Yes" without complaint, because the level set had one member before and has one after. That is what makes the function usable on a raw character column, where repeats are ordinary.

Stripping never produces an empty string: "1." is returned unchanged.

Value

A character vector of display text: one element per input element, or one per level when `x` is a factor. NA is returned as NA.

See Also

[level_map](#) to see what stripping would do across a dataset, [apply_value_labels](#) for declaring level text outright rather than tidying it afterwards.

Examples

```
strip_level_prefix(c("1. Yes", "0 = No", "01 - home"))

# A separator is required, so these survive intact
strip_level_prefix(c("1 vessel disease", "2 vessel", "3 vessel"))

# And so does a range, whose remainder begins with a digit
strip_level_prefix("1-2 vessels")

# A factor is read through its levels, and is not modified
f <- factor(c("1. Yes", "0. No"))
strip_level_prefix(f)
levels(f)
```

study_abbreviations *The abbreviation list a study's labels use*

Description

Merges the three abbreviation lists a job's labels can draw on, and checks the result, so that [label_map](#) shortens labels the same way in every job of a study:

1. extra, the job's own entries;
2. the study's abbreviations: mapping in `_study.yml`;

3. the group default list shipped with this package.

A higher level replaces a lower level's entry for the same phrase, compared ignoring case, and an entry of null (NA or NULL in extra) removes it.

Usage

```
study_abbreviations(cfg = study_config(), extra = NULL, defaults = TRUE)
```

Arguments

<code>cfg</code>	A study configuration, as returned by study_config .
<code>extra</code>	The job's own entries: a named character vector or list, phrase to abbreviation, with NA or NULL to remove a phrase a lower level supplies. NULL for none.
<code>defaults</code>	FALSE leaves the group default list out.

Details

Every list is checked before anything is merged, and every problem is reported in one error: an abbreviation must be one non-empty string no longer than its phrase, and a phrase may appear only once in a list. After merging, two phrases may not share an abbreviation, because the shortened label could then mean either; the error names both phrases and the level each came from. Abbreviations are compared ignoring case, with one exception for the house style: R for replacement and r for repair (AVR, AVr) may differ only in that last letter's case.

One entry may list several spellings of a term, separated by " | ": "Red blood cell | Red blood cells" = "RBC". The spellings share the entry's abbreviation, which is not a clash, and the first spelling is the expansion a key prints.

The list is a display input. It is never written into the stored labels.

Value

A named character vector, one element per spelling, phrase to abbreviation, ready for `label_map(abbreviations = )`. Removed phrases do not appear. Two attributes of the same length:

source each entry's level: "job", "study" or "default";

expansion the term each abbreviation stands for, the first spelling of its entry, which [label_map](#) prints in its key.

See Also

[add_abbreviation](#) to add to a study's list, [label_map](#), which applies it.

Examples

```
root <- file.path(tempdir(), "abbrev-example")
study_setup(root, "Abbreviation example", 1L, renv = FALSE)
add_abbreviation("Surgical procedure", "SP", start = root)
study_abbreviations(study_config(root, require_data = FALSE),
  extra = c("Left ventricular outflow tract" = "LVOT"))
unlink(root, recursive = TRUE)
```

study_checklist	<i>Render a study audit as a markdown checklist</i>
-----------------	---

Description

Turns a [study_status](#) result into markdown: one checkbox per check, ticked where the check passed ("OK") or the study is closed ("CLOSED") and left open otherwise, with the detail alongside. File counts follow.

Usage

```
study_checklist(status, path = NULL)
```

Arguments

status	An object of class "study_status", from study_status or study_setup .
path	Character(1) or NULL. Where to write the markdown. When NULL (default) the lines are returned instead of written.

Value

When path is NULL, a character vector of markdown lines. Otherwise path, invisibly, after writing.

See Also

[study_status](#), [study_setup](#)

Examples

```
root <- file.path(tempdir(), "study-checklist-example")
dir.create(root, showWarnings = FALSE)
cat(study_checklist(study_status(root)), sep = "\n")
unlink(root, recursive = TRUE)
```

study_checkpoint	<i>Record a study checkpoint</i>
------------------	----------------------------------

Description

Commits an allow-listed snapshot of the study (code, identity and reproducibility files) to a private git repository in `.checkpoint/repo/`, tags it with the checkpoint kind and a sequence number, records it in the outbox `.checkpoint/log.yml`, and pushes it when `_study.yml` names a remote. Known data formats, credentials and symbolic links are never committed, even through `include: patterns`, and neither is anything under `00_datasets/` or `90_estimates/` or an output file type. Files in `50_documents/` other than `.qmd` and `.bib` sources are not committed; `CHECKPOINT.yml` records their size and checksum.

Usage

```
study_checkpoint(
  kind,
  note = NULL,
  attributes = NULL,
  occurred_at = Sys.Date(),
  root = study_root()
)
```

Arguments

kind	Character(1). A checkpoint kind.
note	Optional character(1), stored with the checkpoint.
attributes	Optional named list of kind-specific details, each a single value, for example <code>list(journal = "JTCVS")</code> .
occurred_at	Date the event happened. Defaults to today.
root	Character. Study root. Defaults to <code>study_root()</code> .

Details

The kind comes from the StudyTracker checkpoint vocabulary, for example "abstract_submitted" or "manuscript_submitted". Automatic kinds such as "data_received" are logged without a snapshot. A checkpoint is committed locally before anything is pushed, so an unreachable remote never loses one; [study_checkpoint_push](#) retries later.

One session at a time: this function, [study_checkpoint_push](#), [study_close](#) and `study_reopen()` hold the lock `.checkpoint/lock` while they run. A call that finds it held by another session stops, naming the holder; retry once that session has finished. The holder refreshes the lock between phases (selection, the snapshot's CHECKPOINT.yml, the commit and tag, delivery), so a lock not refreshed for 6 hours was left by a crashed session and is taken over with a warning.

note and attributes are written to the snapshot, the tag and the outbox, so they leave the study folder. A message says so whenever either is given: they must not contain patient information.

Value

An object of class "study_checkpoint", returned invisibly, with the tag, commit, file count, skipped files and delivery states.

See Also

[study_checkpoint_push](#), [study_close](#), [study_status](#)

Examples

```
if (nzchar(Sys.which("git"))) {
  root <- file.path(tempdir(), "checkpoint-example")
  study_setup(root, "Checkpoint example", 1267L, renv = FALSE)
  writeLines("x <- 1", file.path(root, "30_analyses", "fit.R"))
  # A throwaway identity, so the example commits on a machine with no
```

```

# git user configured.
withr::with_envvar(c(GIT_AUTHOR_NAME = "Example",
                     GIT_AUTHOR_EMAIL = "example@example.org",
                     GIT_COMMITTER_NAME = "Example",
                     GIT_COMMITTER_EMAIL = "example@example.org"), {
  cp <- study_checkpoint("abstract_submitted", root = root)
  print(cp$tag)
})
unlink(root, recursive = TRUE)
}

```

study_checkpoint_push *Push pending study checkpoints*

Description

Retries delivery of every checkpoint, closure and reopening in `.checkpoint/log.yml` that has not reached the remote yet. [study_checkpoint](#) does this on every call; use this function after a network outage, or once `study-setup --verify` has verified a manually entered identity.

Usage

```
study_checkpoint_push(root = study_root())
```

Arguments

`root` Character. Study root. Defaults to `study_root()`.

Details

An entry that a crash left half-written is settled first: completed when its tag exists, otherwise marked abandoned. Abandoned entries are never pushed.

Value

A data frame, returned invisibly, with one row per logged event and columns `type`, `tag`, `state`, `git`, `st` and `reason`.

Repairing a stuck delivery

Two failures leave an entry pending in a way that retrying `study_checkpoint_push()` cannot fix by itself. Each is reported as a warning that names the tag and points back here.

- **Not on main** (a log write was lost after a replay). This one usually repairs itself. Delivery finds the entry's replayed commit in one of three ways, in order: another entry in the log already records the same old commit as its `replayed_from`; exactly one commit

on main carries the entry's own id (checkpoint_id or closure_id); or exactly one commit on main carries the id in the old commit's own message. The last way covers a reopening, whose reopening_id is only in its tag: it tags an existing commit, a closure's or that of a checkpoint taken while the study was closed. Delivery then points the entry at that commit, keeps the old value as replayed_from, and delivers as normal. Only when none of these finds exactly one commit does the entry stay pending with a warning. Then repair it by hand: look the id up with `git -C .checkpoint/repo log --fixed-strings --grep=<id> --format=%H main`, choose the right commit, set the entry's `git_commit` in `.checkpoint/log.yml` to it, and set its `replayed_from` to the old `git_commit`. Then run `study_checkpoint_push()` again. Do not skip `replayed_from`: a later entry that names the same old commit, such as a reopening, heals from it. If no such commit exists, set the entry's state to "abandoned" instead; an abandoned entry is never delivered.

- **Unnumbered tag clash** (two copies of the study each recorded the same unnumbered tag, for example `workspace_created`). The two copies have diverged. Keep one copy's `.checkpoint/` directory, normally the one whose history is already on the remote, move the other copy's `.checkpoint/` aside, and run `study_checkpoint_push()` again from the kept copy.

See Also

[study_checkpoint](#)

Examples

```
if (nzchar(Sys.which("git"))) {
  root <- file.path(tempdir(), "checkpoint-push-example")
  invisible(study_setup(root, "Checkpoint example", 1267L, renv = FALSE))
  # A throwaway identity, so the example commits on a machine with no
  # git user configured.
  withr::with_envvar(c(GIT_AUTHOR_NAME = "Example",
                      GIT_AUTHOR_EMAIL = "example@example.org",
                      GIT_COMMITTER_NAME = "Example",
                      GIT_COMMITTER_EMAIL = "example@example.org"), {
    invisible(study_checkpoint("abstract_submitted", root = root))
    # No remote is configured, so nothing leaves the machine: the
    # checkpoint stays pending, and the reason says why.
    print(study_checkpoint_push(root))
  })
  unlink(root, recursive = TRUE)
}
```

Description

`study_close()` records how a study ended: it takes a final checkpoint snapshot, tags it `closed-<outcome>-<n>` and records a closure in the outbox. It does not make the study read-only: the folder stays writable, and a checkpoint taken on a closed study is still recorded, with a warning. The closure tag is what fixes the code as it stood at closure. `study_reopen()` records a reopening and tags the current snapshot `reopened-<n>`; the next checkpoint snapshots as usual. A study may be closed and reopened any number of times, and every cycle is kept.

Usage

```
study_close(
  outcome,
  reason = NULL,
  publication = NULL,
  superseded_by = NULL,
  closed_at = Sys.Date(),
  root = study_root()
)

study_reopen(
  reason,
  new_lead = NULL,
  reopened_at = Sys.Date(),
  root = study_root()
)
```

Arguments

<code>outcome</code>	Character(1). One of "published", "not_published", "superseded" or "abandoned".
<code>reason</code>	Optional character(1). For <code>study_reopen()</code> , required.
<code>publication</code>	Named list of publication details; required for "published".
<code>superseded_by</code>	One whole positive number, the ST number, given as an integer, a double or a string; required for "superseded".
<code>closed_at, reopened_at</code>	Date of the event. Defaults to today.
<code>root</code>	Character. Study root. Defaults to <code>study_root()</code> .
<code>new_lead</code>	Optional character(1), the username of a new study lead.

Details

The outcomes follow the StudyTracker closure rules:

- "published" needs publication (title, journal, accepted_on, published_on, and at least one of doi and pmid) and an earlier "manuscript_published" checkpoint.
- "superseded" needs superseded_by, the ST number of the study that replaced it.
- "not_published" and "abandoned" need nothing further.

These rules are checked before anything is written, so a close made offline fails at once rather than when the outbox is delivered.

Whether the study is closed is read from the latest closed-* or reopened-* tag on the checkpoint history, counting tags pushed from other copies of the study. Before deciding, both functions sync with the remote, delivering any pending checkpoints first. When the remote cannot be reached, a message says that the closure state is from this copy's last fetch and may be out of date. When a closure or reopening tag in this copy is not on its history, which an interrupted delivery can leave, the state cannot be determined and both functions stop; run [study_checkpoint_push](#) while the remote is reachable.

reason is written to the snapshot, the tag and the outbox, so it leaves the study folder. A message says so whenever it is given: it must not contain patient information.

Value

An object of class "study_checkpoint", returned invisibly.

See Also

[study_checkpoint](#), [study_status](#)

Examples

```
if (nzchar(Sys.which("git"))) {
  root <- file.path(tempdir(), "close-example")
  study_setup(root, "Close example", 1267L, renv = FALSE)
  # A throwaway identity, so the example commits on a machine with no
  # git user configured.
  withr::with_envvar(c(GIT_AUTHOR_NAME = "Example",
                      GIT_AUTHOR_EMAIL = "example@example.org",
                      GIT_COMMITTER_NAME = "Example",
                      GIT_COMMITTER_EMAIL = "example@example.org"), {
    study_close("abandoned", reason = "PI left", root = root)
    study_reopen("new PI", root = root)
  })
  unlink(root, recursive = TRUE)
}
```

study_config

Read the study manifest

Description

Walks up from start until a `_study.yml` is found, parses it, validates the requested identity or data contract, and returns the result with the study root attached.

A study without a manifest must not render, so an absent `_study.yml` is an error rather than a set of defaults. The directories walked are named in the error, because the usual cause is starting from outside the study tree.

Study identity always requires `study`. With `require_data = TRUE`, `built` is also required. It must carry its file extension, because the reader dispatches on the extension.

Usage

```
study_config(start = getwd(), require_data = TRUE)
```

Arguments

<code>start</code>	Character. Directory to start the upward walk from. Defaults to <code>getwd()</code> .
<code>require_data</code>	Logical. If <code>TRUE</code> , require the default dataset and registered file. Use <code>FALSE</code> when only study identity is needed.

Value

The manifest as a list, with `root` and `file` attached. Additive identity and named-dataset fields are retained.

See Also

[study_root](#), [record_provenance](#)

Examples

```
root <- file.path(tempdir(), "study-example")
dir.create(root, showWarnings = FALSE)
yaml::write_yaml(
  list(study = "Example", built = "example.sas7bdat"),
  file.path(root, "_study.yml")
)
cfg <- study_config(root)
cfg$study
unlink(root, recursive = TRUE)
```

study_declare

Declare the packages a study needs

Description

Records package names in the study's DESCRIPTION, the file `renv` reads to build the study's `renv.lock` in explicit mode. Declaring rather than letting `renv` scan the code means a stale template copy can never put a package into the lockfile.

Names are only ever added, and an entry already declared stays as written, version constraint included. Every other field of the file is kept. The file is rewritten only when the set changes, and a DESCRIPTION that is not a study project's (no `Type: Project`) is never edited.

Usage

```
study_declare(packages = character(0), root = NULL)
```

Arguments

packages	Character. Package names to declare.
root	Character(1). The study root; by default the study enclosing the working directory, as study_root finds it.

Value

The sorted declared package names, invisibly.

Examples

```
root <- file.path(tempdir(), "declare-example")
dir.create(root, showWarnings = FALSE)
study_declare("arrow", root)
```

study_dir	<i>Resolve a directory in the study layout</i>
-----------	--

Description

Maps a logical study directory such as "datasets" to the numbered spelling used by new studies or the bare spelling retained by legacy studies. A root containing both layouts is an error.

Usage

```
study_dir(folder, root = study_root())
```

Arguments

folder	Character. One logical study directory name.
root	Character. Study root. Defaults to <code>study_root()</code> .

Value

Character(1). The resolved directory path. Its existence is not required.

See Also

[study_root](#), [sas_path](#)

Examples

```
root <- file.path(tempdir(), "numbered-study-layout")
dir.create(file.path(root, "00_datasets"), recursive = TRUE,
           showWarnings = FALSE)
study_dir("datasets", root)
unlink(root, recursive = TRUE)
```

study_need

*Install a package a study needs, or say how***Description**

Called where a function needs a package the study may not have, such as **arrow** for registered data. In the study's active renv project the packages are declared (see [study_declare](#)); a missing one is installed into the study's library after one question, when the session is interactive, and recorded in `renv.lock`. Otherwise the call stops and names `hvtiR::setup()`, which installs everything a study needs.

Nothing is declared until every package is installed: if the user declines the install, or it fails, the study's DESCRIPTION is left as it was.

An installed package is never upgraded, whatever its version: moving a study's package versions is `hvtiR::update()`'s job.

Usage

```
study_need(
  packages,
  root = NULL,
  caller = "study_need",
  note = NULL,
  only_in_renv = FALSE,
  r_upgrade = FALSE
)
```

Arguments

<code>packages</code>	Character. Package names.
<code>root</code>	Character(1). The study root; by default the study enclosing the working directory, if any.
<code>caller</code>	Character(1). The function that needs the packages, named in the question and the error.
<code>note</code>	Character(1) or NULL. Appended to the error.
<code>only_in_renv</code>	Logical(1). Do nothing outside an active renv study.
<code>r_upgrade</code>	Logical(1). Leave FALSE. The <code>renv.lock</code> snapshot that follows an install is refused from an R whose major.minor differs from the R the lockfile records (see guard_snapshot). Set TRUE only when you are deliberately moving the study to this session's R.

Value

The names of the packages installed, invisibly.

Examples

```
# Nothing to install, and nothing written, outside a study:
study_need("stats")
```

study_root	<i>Locate the study root</i>
------------	------------------------------

Description

Returns the absolute path of the directory holding `_study.yml`, found by walking up from `start`. Errors if there is none, naming the directories walked.

Usage

```
study_root(start = getwd())
```

Arguments

`start` Character. Directory to start the upward walk from. Defaults to `getwd()`.

Value

Character(1). The absolute path of the study root.

See Also

[study_config](#), [sas_path](#)

Examples

```
root <- file.path(tempdir(), "study-root-example")
dir.create(root, showWarnings = FALSE)
yaml::write_yaml(
  list(study = "Example", built = "example.sas7bdat"),
  file.path(root, "_study.yml")
)
study_root(root)
unlink(root, recursive = TRUE)
```

study_setup

Set up study identity and directories

Description

The usual entry point is `hvtiR::setup()`. Creates the numbered directory structure for a new study and writes an identity-only `_study.yml`. With `adopt = TRUE`, an existing study keeps its legacy or numbered directory layout and every existing initialization file.

Data are deliberately not required. Call [register_data](#) after the default or a named dataset exists.

When the root holds no `.Rproj` file, one named for the root directory is written, so that opening the project and [study_root](#) agree on the study root. An existing project is left unchanged.

With `cleanup = TRUE`, the template copies a legacy study carries are deleted before anything else reads the study, anywhere in it: every file whose name starts with `tp.`, every `binder.summaries.doc`, and every other file whose name and exact content match a file in the template corpus at `template_dir`. A study's own file that shares a name with a corpus file but differs from it (an edited `references.bib`, say) is kept, as are `.renvignore` and `.gitignore`. These files are copied in when a study is created, months before any analysis, and the corpus they came from is kept centrally, so no backup is made. A folder the deletions leave holding nothing (desktop litter such as `Thumbs.db` aside) is removed, whatever its name, as is an empty `templates` or `old_RF_templates*` folder; a folder still holding other files stays, as does one holding any folder the cleanup did not empty, and a top-level study folder is never removed. Symbolic links are neither followed nor deleted, and folders named `.git`, `.checkpoint` or `renv` are never entered, at any depth. The root and `template_dir` must not overlap. A file that cannot be deleted stops the setup with an error.

With `renv = TRUE`, the study's `renv` environment is set up after any cleanup, in a separate R process. A complete environment is kept: a `renv/` folder holding `activate.R`, a `.Rprofile` that sources it, and a `renv.lock` that parses as a lockfile. One interrupted after initialization (no lockfile yet) is resumed at the snapshot; any other partial or invalid environment stops with an error. Otherwise the study DESCRIPTION is written (see [study_declare](#)), `renv::init(bare = TRUE)` creates the environment, and the project is loaded and snapshotted explicitly in explicit mode, so `renv.lock` comes from the DESCRIPTION and not from a scan of the study's code. The study's `renv` does not use a per-user package cache (`use.cache` is `FALSE` in `renv/settings.json`), because the library is shared by the team and links into one person's cache that others cannot read break their installs. The `renv` package must be installed.

Usage

```
study_setup(
  root = NULL,
  study,
  study_tracker_id,
  umbrella = NULL,
  owner = NULL,
  irb_number = NULL,
  cvir_no = NULL,
  study_creation_date = NULL,
```

```

adopt = FALSE,
identity_source = c("tracker", "manual"),
cleanup = adopt,
renv = TRUE,
template_dir = "/programs/development/template",
r_upgrade = FALSE
)

```

Arguments

root	Character. Study root to create or adopt. When omitted, the study enclosing the working directory (found as study_root finds it) is used, or the working directory itself when no <code>_study.yml</code> lies above it. An empty working directory is set up as a new study.
study	Character(1). Study title.
study_tracker_id	Integer(1). Study Tracker topic ID.
umbrella, owner, irb_number, cvir_no	Optional identity values.
study_creation_date	Optional Study Tracker creation date.
adopt	Logical. Permit additive setup in an existing root. When the root already holds a <code>_study.yml</code> for the same Study Tracker ID and the session is interactive, <code>study_setup()</code> asks whether to adopt it instead of stopping.
identity_source	Character(1). Where the identity values came from: "tracker" (the default), for a Study Tracker record, or "manual", for values typed by hand while the Tracker was unavailable. A new <code>_study.yml</code> records it as <code>identity_source</code> , with <code>identity_verified</code> set to TRUE for "tracker" and FALSE for "manual". An existing <code>_study.yml</code> is not rewritten, so adoption keeps the recorded source.
cleanup	Logical. Delete the study's <code>tp.*</code> template copies and <code>binder.summaries.doc</code> , and remove the template folders they leave empty. See Details. Defaults to adopt: adopting a study cleans it, including an adoption accepted at the prompt. Pass FALSE to adopt without cleaning.
renv	Logical. Initialize and snapshot the study's <code>renv</code> environment. See Details. Defaults to TRUE; test suites that build temporary studies should pass FALSE.
template_dir	Character(1). The template corpus studies were copied from, read when <code>cleanup</code> = TRUE. Setup stops with an error when it does not exist.
r_upgrade	Logical(1). Leave FALSE. The <code>renv</code> snapshot is refused from an R whose <code>major.minor</code> differs from the R recorded in the study's <code>renv.lock</code> (see guard_snapshot), because it would rewrite the lock's R and drop packages. Set TRUE only when you are deliberately moving the study to this session's R.

Value

An object of class "study_status", returned visibly. With `cleanup` = TRUE it carries a "cleanup" attribute, a list of the deleted files and removed_dirs folders, relative to the root. With `renv` = TRUE it carries a "renv" attribute, "initialized" or "retained".

See Also

[study_status](#), [study_config](#), [study_dir](#)

Examples

```
root <- file.path(tempdir(), "study-setup-example")
study_setup(root, study = "Example", study_tracker_id = 1L, renv = FALSE)
list.files(root)

# Running it again on an existing study needs adopt = TRUE; nothing that
# exists is overwritten.
invisible(study_setup(root, study = "Example", study_tracker_id = 1L,
                      adopt = TRUE, renv = FALSE, cleanup = FALSE))

# cleanup = TRUE deletes the template copies a legacy study carries: tp.*
# files, and files identical to one in the template corpus.
corpus <- file.path(tempdir(), "template-corpus")
dir.create(corpus)
writeLines("# template", file.path(corpus, "readme.R"))
invisible(file.copy(file.path(corpus, "readme.R"),
                    file.path(root, "30_analyses")))
writeLines("# template", file.path(root, "30_analyses", "tp.hazard.R"))
done <- study_setup(root, study = "Example", study_tracker_id = 1L,
                   adopt = TRUE, cleanup = TRUE, renv = FALSE,
                   template_dir = corpus)
attr(done, "cleanup")

unlink(c(root, corpus), recursive = TRUE)
```

study_status

Audit a study's reproducibility readiness

Description

Reports, without writing anything, whether a study has the four things a later result needs in order to be re-derivable: a valid `_study.yml`, an `renv.lock`, a `manifest.yaml` whose checksums still match the data, and a provenance sidecar for every `.qmd` or `.Rmd` source.

Every finding is reported rather than raised. A study with no `_study.yml` is the thing this function exists to describe, so it must not error on one. Checks that cannot run because an earlier one failed are reported "MISSING", never "FAIL" – a check that could not run is not a check that failed, and conflating the two makes the audit unreadable on exactly the legacy studies it is most needed for.

A `_study.yml` whose identity was entered by hand and not yet confirmed against Study Tracker (`identity_verified: false`) is reported "UNVERIFIED". A manifest without the field is a Tracker identity and is reported "OK".

A study that has been checkpointed (it has a `.checkpoint/` directory) adds a checkpoints row: "OK" when every checkpoint, closure and reopening has reached the remote, or "PENDING" when

some are saved locally and not yet pushed. Its detail counts the recorded events, names the last tag, and counts those not yet pushed or not yet in ST. A closed study then adds a closure row with status "CLOSED" and the outcome and date of the latest closure; see [study_close](#). This function never contacts the remote, so when one is configured that row adds *as of the last fetch*. When a closure or reopening tag in this copy is not on its main, the state cannot be determined and the row has status "UNKNOWN".

The manifest.yaml row is "PENDING" when a dataset registered as a dated parquet has had its source file rebuilt since registration. Jobs keep reading the registered version; the detail names the datasets and [update_manifest\(\)](#), which registers the rebuild.

A dataset that _study.yml names and manifest.yaml does not register as a dated version adds a dated_version:<dataset> row with status "PENDING": analysis, distribution, graph and document jobs stop when they read it. The detail names [update_manifest\(\)](#). A dataset pinned to a catalog release adds no such row.

Release-aware datasets add an update:<dataset> row with status "CURRENT", "UPDATE AVAILABLE", "UPDATE STATUS UNKNOWN", or "FAIL". Legacy studies retain the five base rows and their existing dataset rows.

A combined dataset built from parents that have since been updated adds an out_of_date:<dataset> row with status "OUT OF DATE". Its detail names the parents' recorded and current versions and the commands that bring the combined data up to date. A combined dataset with no recorded parent versions is reported the same way, with the parent versions shown as unrecorded. If that check itself cannot run, for example because manifest.yaml cannot be parsed, the row has status "FAIL" and the error as its detail.

Unlike [study_config](#), this function does **not** walk up the directory tree. It asks whether root itself is a study root, so that a subdirectory of a study is never mistaken for one.

The provenance check matches sidecars to sources **by file name**, because [record_provenance](#) writes the sidecar beside the rendered output and Quarto's output directory is usually not the source's directory. Two sources sharing a name therefore cannot be distinguished; the repeated names are reported in the check's detail so the gap is visible rather than silent.

Usage

```
study_status(root = getwd())
```

Arguments

root Character. The study root to audit. Defaults to getwd().

Value

An object of class "study_status": a list with root, checks (a data frame of item, status – "OK", "MISSING", "FAIL", "UNVERIFIED", "CURRENT", "UPDATE AVAILABLE", "UPDATE STATUS UNKNOWN", "OUT OF DATE", "PENDING", "CLOSED", or "UNKNOWN" – and detail). The five base rows are followed by release-aware update rows, by dataset and update rows for each named dataset, and by the checkpoints and closure rows when they apply. counts lists r_files, qmd, sas_jobs and sidecars.

See Also

[study_setup](#), [study_checklist](#)

Examples

```
root <- file.path(tempdir(), "study-status-example")
dir.create(root, showWarnings = FALSE)
study_status(root)
unlink(root, recursive = TRUE)
```

study_update_identity *Record a verified study identity*

Description

Writes a Study Tracker identity into an existing `_study.yml` and marks it verified: `identity_source: tracker` and `identity_verified: true`. Every other key, including the data contract (`built`, `cohort`, `additional_datasets`, `population`, `citation`), keeps its value.

Usage

```
study_update_identity(identity, start = getwd())
```

Arguments

<code>identity</code>	A named list from the Study Tracker record. It must hold <code>study_tracker_id</code> , equal to the manifest's, and <code>study</code> ; it may hold <code>umbrella</code> , <code>owner</code> , <code>irb_number</code> , <code>cvir_no</code> and <code>study_creation_date</code> .
<code>start</code>	A directory inside the study; the study is found as study_config finds it.

Details

This is the write step of verifying a manual identity. It does not read Study Tracker or compare the record with the typed identity; the caller does both, and passes the record only when they match. A field the record leaves NULL keeps its current value.

The file is rewritten whole and atomically, as [register_data](#) rewrites it, so comments in `_study.yml` are not kept.

Value

The names of the keys whose values changed, invisibly.

See Also

[study_setup](#), [study_status](#).

Examples

```
root <- file.path(tempdir(), "update-identity-example")
study_setup(root, "Typed title", 42L, identity_source = "manual",
             renv = FALSE)
study_update_identity(
  list(study_tracker_id = 42L, study = "Typed title", owner = "Analyst"),
  start = root
)
study_config(root, require_data = FALSE)$identity_verified
unlink(root, recursive = TRUE)
```

type_conversion_report

Read the per-column type-conversion report

Description

Returns the record `r_data_types` attaches to its result: one row per column, naming the rule that fired and where the levels came from.

Usage

```
type_conversion_report(x)
```

Arguments

`x` An object returned by `r_data_types`.

Details

A column that became a factor because it was *declared* one, and a column that became a factor because it happened to have seven distinct values, are otherwise the same object. This is how a caller tells them apart.

The columns are:

- `variable` – the column name.
- `storage_in` – its class as received, before conversion.
- `rule` – one of "value_labels", "binary_logical", "binary_factor", "character_factor", "n_distinct_factor", "unchanged" or "skipped". "unchanged" means every rule was tested and none fired; "skipped" means the column was named in `skip_vars` and no rule was tested.
- `level_source` – "value labels" where the levels came from the column's own value labels, "inference" where they came from counting distinct values, and NA where no levels were made.
- `n_levels` – levels produced, or NA for a non-factor.
- `storage_out` – the class of the returned column.

The report is an attribute of the returned object. Operations that rebuild a data frame – most **dplyr** verbs among them – drop it. Read it directly from the `r_data_types()` result.

Value

A data frame with columns `variable`, `storage_in`, `rule`, `level_source`, `n_levels` and `storage_out`, one row per column of the converted dataset, in column order.

See Also

[r_data_types](#)

Examples

```
converted <- r_data_types(datasets::mtcars, use_value_labels = FALSE)
type_conversion_report(converted)
```

update_manifest

Create or update a dataset manifest file

Description

In a study, run with no arguments after rebuilding a registered dataset: `update_manifest()`. It finds the study from the working directory, converts every registered dataset whose source file has changed to a new dated parquet (`<name>_YYYYMMDD.parquet`, or `_r2`, `_r3` for another version on the same date), keeps every earlier version, records the change in `manifest.yaml`, and prints one line per dataset. Jobs read the new version from then on. Unchanged datasets are left alone. Datasets that are not combined are updated before the combined datasets built from them, and a combined dataset records its parents' versions when its own rebuilt source is registered. One whose parents have since been updated, or that records no parent versions, is listed as out of date in a message after the table until its source is rebuilt and `update_manifest()` is run again. Name one dataset with `dataset`. Release-aware datasets are skipped; use [review_data_update](#) and [adopt_data_update](#). A key violation on one dataset stops the whole run and leaves every dataset as it was. Needs the **arrow** package.

A single file: `update_manifest(file, ...)` records a SHA-256 checksum, row count, extract date and optional provenance fields for one file in a `manifest.yaml`. If the manifest already contains an entry for the named file it is updated in place; otherwise a new entry is appended. The manifest is intended to be committed to version control while the data files themselves are not. It refuses a file a study registers as dated versions, whose entry it would flatten; use the no-argument form for those. It also refuses one of those versions or its schema file, which is another dataset's registered data.

Row counts are detected automatically for **CSV** (`.csv`) files. For **SAS** (`.sas7bdat`) and **Excel** (`.xlsx`, `.xls`) files, automatic row counting is considered "heavy" because it loads the entire dataset/workbook into memory; it is therefore disabled by default and only performed when `options(manifest.allow_heavy_rowcount = TRUE)` is set. For any other format, or when heavy counting is disabled, supply `n_rows` explicitly.

Usage

```
update_manifest(
  file,
  manifest_path = "manifest.yaml",
  extract_date = Sys.Date(),
  n_rows = NULL,
  n_cols = NULL,
  source = NULL,
  sort_key = NULL,
  schema_sha256 = NULL,
  role = c("source", "primary"),
  reader = NULL,
  verbose = FALSE,
  dataset = NULL
)
```

Arguments

file	Character. Path to the dataset file. Omit it inside a study to register every changed dataset; see the description.
manifest_path	Character. Path to the manifest YAML file. Created if it does not exist. Defaults to "manifest.yaml" in the current working directory.
extract_date	Character or Date. The date the data were pulled from the source system. Stored as "YYYY-MM-DD". Defaults to today's date. With no file, the date of the new version; defaults to the source file's modification date.
n_rows	Integer. Number of data rows. When NULL (default) the row count is detected automatically from CSV files, and from SAS/Excel files only when options(manifest.allow_heavy_rowcount = TRUE) is set. all other file types supply this value explicitly.
n_cols	Integer. Column count. Pass it from a frame already read; a row count alone cannot detect a dropped column.
source	Character. Free-text description of the data source (e.g. "Epic EMR, query v4.2, ICD mapping v3.2").
sort_key	Character. Column name(s) that define the canonical sort order of the dataset.
schema_sha256	Character. SHA-256 of this dataset's schema sidecar, making the manifest-to-sidecar link tamper-evident.
role	Character, always "source". This function only ever writes role = "source" entries; a role = "primary" (promoted) entry's sha256 must describe its parquet , not the source this function hashes, so calling it on a promoted entry would write a manifest that can never verify. No exported function performs promotion yet — the internal .update_promoted_entry() is what maintains a promoted entry once one exists. See the <i>Promotion</i> section of the read-layer design spec.
reader	Character. The package and version that produced this derived file (e.g. "haven 2.5.5"). Under role = "primary" the parquet is the data, so the reader that produced it is part of its provenance: a reader defect found later is otherwise unfindable once the source is retired. NULL (default) omits the field.

verbose	Logical. If TRUE, report which manifest entry was added or updated via message . Defaults to FALSE so that scripted or looped calls stay silent.
dataset	Character(1) or NULL. With no file: the one registered dataset to update; NULL updates every one that changed. "built" is a second name for "study". Ignored when file is given.

Value

With file, invisibly returns the updated manifest as a named list. With no file, invisibly returns a data frame with one row per dataset and columns dataset, action ("registered", "migrated", "unchanged", "skipped" or "missing", for a source that is not on disk and has no registered version) and detail.

See Also

[verify_manifest](#)

Examples

```
dir <- file.path(tempdir(), "update-manifest-example")
dir.create(dir)
csv <- file.path(dir, "cohort_20240115.csv")
utils::write.csv(data.frame(patient_id = 1:3, age = c(61, 70, 55)), csv,
  row.names = FALSE)
manifest <- file.path(dir, "manifest.yaml")
update_manifest(csv, manifest_path = manifest, extract_date = "2024-01-15",
  source = "Synthetic example", sort_key = "patient_id")
cat(readLines(manifest), sep = "\n")
verify_manifest(manifest, verbose = TRUE)
unlink(dir, recursive = TRUE)
```

verify_manifest	<i>Verify all datasets listed in a manifest</i>
-----------------	---

Description

Reads a manifest.yaml produced by [update_manifest](#) and, for every entry, confirms that (a) the file exists, (b) its SHA-256 checksum matches the recorded value, and (c) its row count matches. Supported formats for automatic row-count verification: CSV (.csv), SAS (.sas7bdat), and Excel (.xlsx, .xls). For other file types the row-count check is skipped and only the SHA-256 is verified.

Re-deriving the row count of a SAS or Excel file means reading the whole file, so it only happens under options(manifest.allow_heavy_rowcount = TRUE). Without that option the row count is skipped rather than failed, and the entry passes on its checksum, which is the stronger of the two checks. A check that could not run is not a check that failed. A file that cannot be read at all is a different matter and still fails.

An entry recording no SHA-256 fails, because the checksum is the whole of what this function verifies and there is nothing to compare it against. Manifests written by an md5-based writer are the case that turns up in practice; the failure names the algorithm that was recorded instead.

Three outcomes are therefore possible for an entry that passes, and the report distinguishes them rather than reporting all three as simply verified. The count was compared; the count is recorded but was not re-derived; or the manifest records no count at all. The `row_count_checked` column is TRUE only in the first case, and the message names which of the three it was. A build-record row (see below) sets `row_count_checked` to TRUE once its checksum is verified, because a record has no row count.

For a dataset registered with `register_data`, every registered version (the current dated parquet and each earlier one) is checked, and its row count is re-derived from the parquet footer when the arrow package is available. A source file rebuilt since registration is reported with status "PENDING" and never stops: jobs keep reading the registered version until `update_manifest()` registers the new one.

Call this function at the top of every analysis script or Quarto document to ensure data integrity before any results are generated.

Usage

```
verify_manifest(
  manifest_path = .default_manifest_path(),
  data_dir = NULL,
  stop_on_error = TRUE,
  verbose = FALSE,
  strict = FALSE
)
```

Arguments

<code>manifest_path</code>	Character. Path to the manifest YAML file. Defaults to the study's <code>manifest.yaml</code> when run inside a study (a <code>_study.yml</code> here or above), and to " <code>manifest.yaml</code> " in the working directory otherwise.
<code>data_dir</code>	Character. Directory holding the dataset files. When supplied, it is used exactly as given. When NULL (default), the manifest directory is inspected once. <code>00_datasets/</code> supports the numbered layout created by <code>study_setup</code> , <code>datasets/</code> supports an adopted legacy layout, and a manifest with neither uses its own directory. Outside a study, an empty nested directory does not displace files beside the manifest. Both dataset directories together are a mixed layout and produce an error; entries never fall back between layouts.
<code>stop_on_error</code>	Logical. If TRUE (default) the function calls <code>stop()</code> on the first failed check, preventing the analysis from proceeding. Set to FALSE to collect all errors and report them together as a warning.
<code>verbose</code>	Logical. If TRUE, report each passing entry via <code>message</code> . Defaults to FALSE so that scripted or looped calls stay silent; the same information is always available in the returned data frame. Failures are reported through <code>stop()</code> or <code>warning()</code> regardless of this setting.

strict Logical. If TRUE, an entry whose row count could not be re-derived is reported as "FAIL" rather than passing on its checksum alone, and the message names which of the three causes applies: no count recorded in the manifest, a file type whose rows cannot be counted, or heavy row counting left disabled. Defaults to FALSE, which preserves the permissive behaviour described above.

Use it where the question is "did every check actually run", not "is the data intact": a release gate, an archival gate, or a hand-off where a downstream reader will treat "OK" as meaning fully verified. Under the default, "OK" can mean "checksum verified, row count not examined", and the row_count_checked column is the only thing that distinguishes the two.

Value

Invisibly returns a data frame with columns file, status ("OK" or "FAIL"), message, and row_count_checked (logical; TRUE when the row count was re-derived from the file and compared with the manifest; also TRUE for a build-record row whose checksum was verified, since a record has no row count).

See Also

[update_manifest](#)

Examples

```
dir <- file.path(tempdir(), "verify-manifest-example")
dir.create(dir)
csv <- file.path(dir, "cohort_20240115.csv")
utils::write.csv(data.frame(patient_id = 1:3, age = c(61, 70, 55)), csv,
                 row.names = FALSE)
manifest <- file.path(dir, "manifest.yaml")
update_manifest(csv, manifest_path = manifest, extract_date = "2024-01-15")

# Silent unless a check fails; verbose = TRUE reports each passing entry.
verify_manifest(manifest, verbose = TRUE)

# The file changes after registration. Collect failures instead of
# stopping on the first.
utils::write.csv(data.frame(patient_id = 1:4, age = c(61, 70, 55, 48)), csv,
                 row.names = FALSE)
report <- suppressWarnings(verify_manifest(manifest, stop_on_error = FALSE))
report[, c("file", "status")]
unlink(dir, recursive = TRUE)
```

write_collision_report

Write the macro name-collision report

Description

Reports every macro name defined in more than one file, with its distinct body count and defining files. In SAS, %include-ing two files that define the same macro means the second silently shadows the first, so this report is a prerequisite for building a trustworthy SAS harness.

Usage

```
write_collision_report(x, path)
```

Arguments

x	A data.frame returned by sas_triage .
path	Character. Destination .md path.

Details

The report ends with a **Provenance** section derived from the scanned corpus rather than from the clock: a fingerprint over every (macro, file, body hash) triple and the excluded-directory list, the file, definition and name counts, and the package version.

A generated-on field would defeat the byte-for-byte reproducibility that makes the artifact reviewable, which is why neither this report nor [write_macro_manifest](#) carries one. An artifact with no identity at all is the opposite failure: a committed report and a design's prose can disagree with nothing to say which is stale. An input-derived stamp avoids both, because it is a function of the inputs alone.

Determinism is therefore over the corpus *and* the package version, not the corpus alone: re-running under a different version changes the version row, by design, because that is what attributes the artifact to a build. The fingerprint itself depends only on the corpus, so it stays comparable across versions.

The fingerprint follows content, not location. [sas_triage](#) records file basenames, so re-scanning the same corpus from a different mount point yields the same value and a share remount does not invalidate every committed report.

Value

Invisibly, path.

Examples

```
d <- system.file("extdata", "macros", package = "hvtiRutilities")
if (nzchar(d)) {
  write_collision_report(sas_triage(d), tempfile(fileext = ".md"))
}
```

write_macro_manifest	<i>Write the canonical macro manifest</i>
----------------------	---

Description

Serialises a `sas_triage()` decision table to YAML, one entry per macro name, recording its status and every file that defines it.

Usage

```
write_macro_manifest(x, path)
```

Arguments

<code>x</code>	A <code>data.frame</code> returned by sas_triage .
<code>path</code>	Character. Destination ‘.yaml’ path.

Details

The manifest deliberately contains **no timestamp**. It describes a frozen 2019 corpus, and a generated-on field would defeat the byte-for-byte reproducibility that makes the manifest trustworthy as a reviewed artifact. Run metadata belongs in the report, not here. This diverges from [update_manifest](#), whose `extract_date` is meaningful because the datasets it tracks genuinely change over time.

Value

Invisibly, `path`.

Examples

```
d <- system.file("extdata", "macros", package = "hvtiRutilities")
if (nzchar(d)) {
  write_macro_manifest(sas_triage(d), tempfile(fileext = ".yaml"))
}
```

Index

add_abbreviation, [4](#), [83](#)
add_labels, [5](#), [7](#), [8](#), [42](#), [43](#)
adopt_data_update, [6](#), [17](#), [70](#), [74](#), [100](#)
apply_label_overrides, [5](#), [7](#), [9](#), [17](#), [42](#), [43](#)
apply_value_labels, [8](#), [45](#), [81](#), [82](#)
assert_cohort, [10](#), [18](#), [66](#)

built_manifest, [11](#), [12](#), [66](#)
built_path, [11](#), [12](#)

cache_fit, [13](#)
capture_provenance, [14](#), [15](#), [15](#), [58](#), [59](#), [61](#),
[69](#)
check_data_updates, [6](#), [16](#), [74](#)
clean_labels, [17](#)
cohort_counts, [10](#), [18](#), [29](#)
compare_datasets, [19](#), [72](#), [73](#)
compare_parity, [20](#), [46](#), [52](#)
concept_map, [21](#), [23](#), [47](#), [48](#), [60](#), [80](#), [81](#)
concept_of, [21](#), [22](#), [47](#), [80](#)
covariate_audit, [24](#), [25](#), [39](#), [80](#)
covariates_to_numeric, [25](#), [25](#)

data_dictionary, [26](#), [51](#)
dataset_schema, [27](#), [42](#)

followup_check, [28](#)

generate_survival_data, [30](#)
get_label, [32](#), [32](#), [33](#), [43](#)
get_labels, [32](#)
guard_snapshot, [33](#), [92](#), [95](#)

hvti_non_prefixes, [34](#)
hvti_prefix_folds, [35](#), [37](#), [41](#)
hvti_taxonomy, [35](#), [36](#), [41](#)
hvtiRutilities-package, [4](#)
hvtiRutilities.news, [38](#)

imputed_levels, [25](#), [38](#)

job_census, [39](#), [41](#)
job_files, [35](#), [39](#), [40](#), [40](#)

label_map, [5](#), [7–9](#), [26](#), [32](#), [33](#), [42](#), [45](#), [68](#), [74](#),
[82](#), [83](#)
level_map, [44](#), [82](#)

message, [102](#), [103](#)

numeric_version, [34](#)

parity_headline, [21](#), [46](#)
parity_tolerance, [20](#), [46](#)
POOL_AFFIXES, [22](#), [47](#)
pool_collinear_pairs, [48](#), [60](#)
POOL_MIN_STEM (POOL_AFFIXES), [47](#)
POOL_PLAIN_SUFFIX (POOL_AFFIXES), [47](#)
preflight_report, [49](#)
print.hvti_job_census (job_census), [39](#)
proc_contents, [26–28](#), [42](#), [50](#), [53](#), [55](#)
proc_freq, [51](#)
proc_means, [26](#), [28](#), [29](#), [51](#), [52](#), [53](#), [53](#), [55–57](#)
proc_univariate, [55](#)
provenance_artifact, [16](#), [57](#), [59](#), [69](#)
provenance_data, [16](#), [58](#), [58](#), [68](#)
provenance_path, [59](#), [61](#), [69](#)
prune_to_one_form, [21](#), [23](#), [48](#), [60](#), [80](#), [81](#)
publish_provenance, [14–16](#), [60](#), [61](#), [69](#)

quote, [13](#)

r_data_types, [8](#), [9](#), [26](#), [62](#), [67](#), [68](#), [74](#), [99](#), [100](#)
r_dir_impurities, [64](#)
read_built, [12](#), [17](#), [59](#), [65](#), [69](#), [70](#), [73](#)
read_clinical_data, [9](#), [65](#), [66](#), [79](#)
read_sas, [31](#)
record_provenance, [11](#), [16](#), [60](#), [61](#), [68](#), [90](#), [97](#)
register_data, [4](#), [6](#), [58](#), [65](#), [69](#), [94](#), [98](#), [103](#)
review_built, [65](#), [72](#), [73](#)
review_data_update, [6](#), [17](#), [73](#), [100](#)
rfsrc, [31](#)

sample_data, [74](#)
sas_macro_defs, [75](#)
sas_macro_signature, [76](#)
sas_path, [76](#), [80](#), [91](#), [93](#)
sas_triage, [77](#), [105](#), [106](#)
sas_variable_block, [79](#)
selection_crowding, [23](#), [80](#)
set.seed, [30](#)
strip_level_prefix, [44](#), [45](#), [81](#)
study_abbreviations, [4](#), [43](#), [82](#)
study_checklist, [84](#), [98](#)
study_checkpoint, [84](#), [86](#), [87](#), [89](#)
study_checkpoint_push, [85](#), [86](#), [89](#)
study_close, [85](#), [87](#), [97](#)
study_config, [4](#), [6](#), [11](#), [12](#), [16](#), [17](#), [58](#), [59](#), [65](#),
[69](#), [71–73](#), [83](#), [89](#), [93](#), [96–98](#)
study_declare, [90](#), [92](#), [94](#)
study_dir, [15](#), [36](#), [91](#), [96](#)
study_need, [33](#), [92](#)
study_reopen(study_close), [87](#)
study_root, [77](#), [90](#), [91](#), [93](#), [94](#), [95](#)
study_setup, [33](#), [71](#), [84](#), [94](#), [98](#), [103](#)
study_status, [6](#), [16](#), [84](#), [85](#), [89](#), [96](#), [96](#), [98](#)
study_update_identity, [98](#)
Surv, [31](#)

tagged_na, [52](#)
to_factor, [62](#)
type_conversion_report, [63](#), [99](#)

update_manifest, [6](#), [19](#), [65](#), [66](#), [69–73](#), [97](#),
[100](#), [102–104](#), [106](#)

verify_manifest, [16](#), [19](#), [102](#), [102](#)

with_seed, [14](#)
write_collision_report, [104](#)
write_macro_manifest, [105](#), [106](#)