

Package: hvtiRbootstrap (via r-universe)

October 20, 2026

Title Bootstrap Model Building for the HVTI CORR Group

Version 0.9.6

Description Builds models by bootstrap resampling, in the manner of the Cleveland Clinic CORR group's SAS macro library: fit a model on each of many bootstrap replicates, record which variables survive selection, and report how often each appeared. Ports the 'bootreg', 'SUMBOOT' and 'cluster' macros.

License GPL-3

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

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

Encoding UTF-8

Depends R (>= 4.4.0)

Imports stats, withr

Suggests knitr, quarto (>= 1.4), survival, testthat (>= 3.0.0)

VignetteBuilder quarto

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

Config/testthat/edition 3

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.1.0

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

Date/Publication 2026-10-15 20:48:51 UTC

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

RemoteRef HEAD

RemoteSha 2d7047b05a041ef9e53adad07e8285f861a0ef49

Contents

boot_bag	2
boot_chunk_files	4
boot_clusters	5
boot_concepts	6
boot_dropped	7
boot_frequencies	9
boot_health	10
boot_pool_chunks	12
boot_predict_ci	13
boot_provenance	15
boot_seeds	16
boot_select	17
boot_shortfall	19
boot_summary	21
boot_validate	22
fit_cox	24
fit_linear	25
fit_logistic	26
Index	27

boot_bag	<i>A screen the reporting layer can read</i>
----------	--

Description

Convert a `boot_select()` result into the bag that `boot_validate()` accepts and every reporting function reads.

Usage

```
boot_bag(x, base_params, requested, manifest, dropped = NULL, usable = NULL)
```

Arguments

x	A boot_selection from <code>boot_select()</code> , run under 0.9.2 or later. An object from an earlier version carries no <code>\$control</code> and is refused by name rather than failing on an absent list element.
base_params	Character. The terms that are the base model rather than candidates. They are excluded from every frequency, and the first of them is the parameter <code>boot_health()</code> watches for a zero standard deviation. Must name terms the screen carries. A Cox model has no intercept , so "(Intercept)" copied from a logistic runner names nothing and is refused.
requested	Numeric. How many candidates the runner offered , before it dropped any. Not derivable here: a candidate dropped before screening never became a column.

manifest	A named list describing the dataset screened, indexed by name. <code>list(sha256 = ...)</code> at minimum.
dropped	Optional data frame of candidates dropped before screening, as <code>boot_dropped()</code> reports it. Absent means nothing was dropped.
usable	Optional numeric, checked rather than used. Supply it to assert the count the runner believed it screened; a disagreement with the screen's own term count is refused, because one of the two is describing a different run.

Details

`boot_select()` returns a **wide** object: one row per replicate, one column per term, NA where a term was not selected. The reporting layer reads a **long** one, because it was extracted from a hazard runner that writes long. Nothing converted between them, so this package's own screen function could not reach its own report and every study would have hand-written the pivot plus nine provenance fields. This is that conversion, written once.

The pivot drops NA rather than writing it. That is not an optimisation: `boot_frequencies()` counts a term's rows against `n_boot`, so a row written for an unselected term would be counted as a selection and every frequency would rise. The replicate count travels in `n_boot`, never in the row count.

Everything the run knows about itself comes from `x$control` rather than from an argument, so a bag cannot claim an entry level the screen did not use. The arguments here are the facts `boot_select()` cannot know: which terms are the base model, how many candidates existed before the runner dropped any, what dataset was screened, and what was dropped.

Value

A list carrying the fields `boot_validate()` requires: `n_boot`, `seed`, `slentry`, `slstay`, `base_params`, `requested`, `usable`, `n_rows`, `elapsed_mins`, `manifest`, `engine`, `dropped` when supplied, and `boot` holding replicates, `summary`, `n_success` and `n_failed`. `boot$summary` is keyed parameter – as `boot$replicates` and `boot_health()` are, and as `boot_pool_chunks()` returns – rather than `boot_summary()`'s variable, and carries `sel_q025` and `sel_q975` alongside the `%SUMBOOT` statistics. Those two are the spread of the coefficient across the replicates that **selected** it, not a confidence interval: a term chosen half the time has an interval over half the replicates. Validated before it is returned, so this function cannot emit a bag that a report will refuse three chunks into a render.

See Also

`boot_select()` for the screen, `boot_provenance()` and `boot_frequencies()` for what reads the result.

Examples

```
set.seed(1)
n <- 200
x1 <- rnorm(n)
df <- data.frame(y = 2 * x1 + rnorm(n), x1 = x1, x2 = rnorm(n))
fit <- boot_select(df, y ~ x1 + x2, fit_linear, n_rep = 10, seed = 42)
bag <- boot_bag(fit, base_params = "(Intercept)", requested = 2,
               manifest = list(sha256 = "example"))
```

```
boot_frequencies(bag)
```

boot_chunk_files	<i>Find the chunk files of a chunked bootstrap run</i>
------------------	--

Description

Returned in a deterministic order, and named so a single full run (`bagging.rds`) and a chunked run never collide.

Usage

```
boot_chunk_files(out_dir, prefix = "bagging")
```

Arguments

<code>out_dir</code>	Directory to look in.
<code>prefix</code>	File-name prefix the runner used.

Value

A sorted character vector of full paths, possibly empty.

See Also

[boot_pool_chunks\(\)](#)

Examples

```
out_dir <- file.path(tempdir(), "boot-chunk-files-example")
dir.create(out_dir)
invisible(file.create(file.path(out_dir, c(
  "bagging.chunk2.rds", "bagging.chunk1.rds", "bagging.rds", "notes.txt"
))))
basename(boot_chunk_files(out_dir))
unlink(out_dir, recursive = TRUE)
```

boot_clusters	<i>Aggregate selection frequencies over clusters of correlated variables</i>
---------------	--

Description

When several candidate terms measure the same thing - weight and body surface area, say - each one's individual selection frequency understates the cluster's importance, because replicates split between them. This reports how often **at least one** member of a cluster was selected. The R port of %cluster (bootstrap.clusters.sas in the CORR macro library).

Usage

```
boot_clusters(x, clusters)
```

Arguments

x	A boot_selection from boot_select() , or a numeric matrix with one row per replicate and one column per term.
clusters	Named list mapping a cluster name to its member terms.

Details

n_any is not the sum of the members' individual counts: a replicate that selected two members counts once.

The macro also tabulates its ncluster counter - the full distribution of *how many* members appeared together in a replicate. That is deliberately not reproduced: the per-variable frequencies are already [boot_summary\(\)](#)'s job, and "at least one" is the number neither function can derive from the other.

Value

A data frame with columns cluster, n_any, pct_any, members, sorted by descending n_any.

See Also

[boot_summary\(\)](#) for the per-variable frequencies, and [boot_select\(\)](#), which produces the replicates.

Examples

```
set.seed(1)
n <- 300
x1 <- rnorm(n)
df <- data.frame(y = 2 * x1 + rnorm(n), x1 = x1,
                 x2 = rnorm(n), noise = rnorm(n))

fit <- boot_select(df, y ~ x1 + x2 + noise, fit_linear,
                  n_rep = 50, seed = 42)
```

```
# x1 and x2 stand in for a correlated pair -- weight and body surface area,
# say. n_any is not the sum of their individual counts: a replicate that
# selected both counts once.
boot_clusters(fit, list(size = c("x1", "x2"), noise = "noise"))
```

boot_concepts	<i>Selection frequencies grouped by concept</i>
---------------	---

Description

A per-form selection frequency answers "how often was *this form* selected". It cannot answer "how often was *this concept* selected", and that is the number a paper quotes.

Usage

```
boot_concepts(bag, concept_map, phase = NULL, threshold = NULL)
```

Arguments

bag	A bootstrap screen. Checked by <code>boot_validate()</code> .
concept_map	A data frame with columns variable and concept, mapping each screened variable to the concept it is a form of. This is the study's own vocabulary and is received, never inferred: which names are forms of one thing is a fact about an institution's naming conventions, not about statistics. A variable the map does not name is treated as a concept of its own rather than dropped, so nothing leaves the concept view silently.
phase	A function mapping a term to its phase, or NULL. As in <code>boot_frequencies()</code> : with NULL there is no phase dimension, and the concept_map is then keyed on whole terms rather than phase-stripped names.
threshold	The retention cutoff, as a percentage, or NULL.

Details

The gap runs both ways. Competing forms split replicates between them, so a concept reads weaker than any single figure suggests; or two forms both clear the cutoff and one finding is reported twice.

pct_any **is not a sum**, for the same reason `boot_clusters()`'s n_any is not: a replicate that selected two forms of one concept counts **once**. Two forms at 30% each are anywhere between 30% and 60% of replicates depending entirely on how often the same replicate took both, and no marginal percentage records that – so this is computed from the replicate table, never from the summary. The computation is `boot_clusters()`'s, called rather than repeated.

spread is pct_any – best_form_pct: how much the per-form view understates the concept. A spread of zero means the concept has one form and the two views agree.

n_retained is the crowding number. A phase that spent several of its slots on forms of **one** concept is budget-limited by redundancy, and that is invisible in a coefficient table; the concepts where n_retained > 1 are the crowded ones.

Nothing here collapses `boot_frequencies()`. Every form keeps its own row and its own frequency there, because two forms of one concept may carry different information: on one real pool, `in_zexp` is `1/zexp` and yet correlates with `zexp` at only -0.195, because `zexp` spans a 4000-fold range. That study’s published model uses both, in the same phase, both significant. This table is an additional view, not a replacement.

Value

A data frame with one row per concept – per concept per phase when phase is supplied – with columns `concept`, `n_forms`, `forms`, `n_any`, `pct_any`, `best_form_pct`, `spread`, `n_retained` and `retained`, plus `phase`. `n_retained` and `retained` are NA when no threshold is given.

See Also

`boot_frequencies()` for the per-form view this groups, and `boot_clusters()` for the same at-least-one count over a group named by declaration rather than by name.

Examples

```
bag <- list(
  n_boot = 4L, seed = 101, slentry = 0.07, slstay = 0.05,
  base_params = "base", requested = 4L, usable = 3L, n_rows = 500L,
  elapsed_mins = 120, manifest = list(sha256 = "abc123"),
  boot = list(
    replicates = data.frame(
      replicate = c(1L, 1L, 1L, 2L, 2L, 3L, 3L, 4L),
      parameter = c("base", "early.age", "early.ln_age",
                    "base", "early.age", "base", "early.ln_age", "base"),
      estimate = c(1, 0.5, 0.4, 1.1, 0.6, 0.9, 0.3, 1.2)),
    summary = data.frame(parameter = "base", n = 4L, pct = 100),
    n_success = 4L, n_failed = 0L))

map <- data.frame(variable = c("age", "ln_age"), concept = c("Age", "Age"))

# Both forms sit at 50%, and replicate 1 took both -- so the concept is at
# 75%, not 100%.
boot_concepts(bag, map, phase = function(term) sub("[.]*$", "", term))
```

boot_dropped	<i>Candidates the screen never saw</i>
--------------	--

Description

The candidates a runner dropped before screening, with the reason each was dropped.

Usage

```
boot_dropped(bag)
```

Arguments

`bag` A bootstrap screen. Checked by `boot_validate()`.

Details

A candidate the screen never saw **cannot appear at any frequency**, so its absence from `boot_frequencies()` is indistinguishable from never having been selected. That is the whole reason this table exists: without it, a constant column and a genuinely unselected variable read the same way.

Whatever columns the runner wrote are kept. Rows are ordered by phase and then reason when those columns are present, so that a reader meets whole classes of dropped candidate together rather than scattered down a table that runs to dozens of rows on a real pool.

Unlike `boot_frequencies()` and `boot_concepts()` this takes **no phase argument**. `bag$dropped` carries its own phase column, written by the runner that did the dropping, so there is nothing for a term-splitting rule to do; a candidate dropped before screening never became a model term.

`dropped` is not a required field. A runner that drops nothing may not write it at all, and that is reported as no rows rather than as a fault.

Value

A data frame of the dropped candidates, with no rows when nothing was dropped.

See Also

`boot_frequencies()` for the candidates that were screened.

Examples

```
bag <- list(
  n_boot = 4L, seed = 101, slentry = 0.07, slstay = 0.05,
  base_params = "base", requested = 4L, usable = 3L, n_rows = 500L,
  elapsed_mins = 120, manifest = list(sha256 = "abc123"),
  dropped = data.frame(variable = c("zexp", "agee"),
    phase = c("late", "early"),
    reason = c("constant", "all missing")),
  boot = list(
    replicates = data.frame(replicate = 1L, parameter = "base",
      estimate = 1),
    summary = data.frame(parameter = "base", n = 1L, pct = 25),
    n_success = 4L, n_failed = 0L))
boot_dropped(bag)
```

boot_frequencies	<i>Selection frequencies, with the error they carry</i>
------------------	---

Description

How often each candidate term survived selection, and how much of that figure is noise.

Usage

```
boot_frequencies(bag, phase = NULL, threshold = NULL)
```

Arguments

bag	A bootstrap screen. Checked by <code>boot_validate()</code> .
phase	A function mapping a term to its phase, or NULL. With NULL there is no phase dimension and no phase column. A multiphase hazard screen passes <code>function(term) sub("[.].*\$", "", term)</code> , which reads <code>early.age</code> as the early phase's decision about age; a logistic, linear or Cox screen has no phases and passes nothing. The rule is applied one term at a time, so it need not be vectorised. <code>variable</code> is the term with the returned phase label stripped from its head, and that happens only when a separator follows it: a label that runs straight into an alphanumeric character is an accidental prefix, and the term stands unstripped rather than being silently mangled.
threshold	The retention cutoff, as a percentage, or NULL. This is a reporting decision and not something the run recorded: <code>slentry</code> and <code>slstay</code> governed each replicate's stepwise fit, and neither says how often a variable must survive to be worth carrying forward.

Details

A selection frequency is an **estimate**, not a count of something fixed. It carries Monte-Carlo error of roughly $\sqrt{p(1-p) / n_{\text{boot}}}$, which at $p = 0.5$ over 500 replicates is about 2.2 percentage points – so a variable sitting within a few points of the retention cutoff can fall on either side of it on resampling noise alone.

`mc_error` is computed **per variable**, not once for the table. The error is largest at $p = 0.5$ and shrinks towards either end, so a single quoted value overstates it for the variables at the top and understates it in the middle – and the middle is where the retention decision is being made.

`n` and `pct` come from `boot_summary()`, which is the function held to exact parity with `%SUMBOOT`. They are not recomputed here: a second implementation of the selection count is the one thing this package cannot afford. What this function adds is the error, the cutoff and the optional phase grouping.

The base parameters are dropped. They are in every replicate by construction, and reporting them at 100% invites them to be read as the screen's most reliable findings.

Value

A data frame with columns `variable`, `term`, `n`, `pct`, `mc_error`, `near_threshold` and `retained`, plus `phase` when `phase` is supplied. `term` is the name as the screen recorded it and `variable` the same name without its phase; they are identical when there is no phase rule. `near_threshold` marks a term within two Monte-Carlo errors of the cutoff – retention that would not survive a rerun with different seeds – and both it and `retained` are NA when no threshold is given, because with no cutoff there is no retention decision to make a claim about.

See Also

`boot_summary()`, which computes the counts; `boot_concepts()` for the same frequencies grouped by concept; `boot_dropped()` for the candidates that never reached a replicate at all.

Examples

```
bag <- list(
  n_boot = 4L, seed = 101, slentry = 0.07, slstay = 0.05,
  base_params = "base", requested = 4L, usable = 3L, n_rows = 500L,
  elapsed_mins = 120, manifest = list(sha256 = "abc123"),
  boot = list(
    replicates = data.frame(
      replicate = c(1L, 1L, 2L, 3L, 4L),
      parameter = c("base", "early.age", "early.age", "base", "late.bmi"),
      estimate = c(1, 0.5, 0.6, 0.9, 0.2)),
    summary = data.frame(parameter = "base", n = 2L, pct = 50),
    n_success = 4L, n_failed = 0L))

boot_frequencies(bag)

# A multiphase screen passes its own term-splitting rule.
boot_frequencies(bag, phase = function(term) sub("[.]*$", "", term),
  threshold = 50)
```

boot_health

Did the bootstrap screen actually run?

Description

Four facts about a screen, two of which are checks that catch a run which produced a perfectly ordinary-looking table of numbers that mean nothing.

Usage

```
boot_health(bag)
```

Arguments

`bag` A bootstrap screen. Checked by `boot_validate()`.

Details

A screen that selected nothing is a failure, not a finding. It is the signature of a formula that did not survive the per-replicate rewrite: the refit errors, the error is caught, the step reports nothing accepted, and the screen halts having selected nothing – with no warning and `n_failed = 0`. The summary then reads as a table of perfectly reliable variables. Absence from a frequency table cannot say this; a row that says `ok = FALSE` can.

A free base parameter must vary across resamples. A bootstrap built on a vector interface returns the original fit every replicate, with `n_success = 500`, `n_failed = 0` and no warning at all. The standard deviation of the first free base parameter is the only tell, and a value of exactly zero is that failure. It is NA, never 0, when there is nothing to take a standard deviation of: zero is the *claim* that the parameter did not vary, and one replicate cannot support it.

`free_sd` is recomputed when the screen does not carry it. Only `boot_pool_chunks()` writes that field, so a single unchunked run has none – and an unchunked run is the shape most likely to have been built the way this check exists to catch. Skipping it there would turn the check off in exactly the case that needs it.

This function **returns** its findings rather than stopping. A report decides how to present them: a callout, a red cell, or a `stop()` of its own.

Value

A data frame with one row per check and columns `check`, `value`, `ok` and `note`. `ok` is FALSE for a failed check, TRUE for a passed one, and NA for a row that is a fact rather than a check. `note` says what a failure means and is NA otherwise.

See Also

`boot_provenance()` for where the screen came from, and `boot_shortfall()` for whether the pool is the run that was launched – a different question, and one nothing inside a chunk can answer.

Examples

```
bag <- list(
  n_boot = 4L, seed = 101, slentry = 0.07, slstay = 0.05,
  base_params = "base", requested = 4L, usable = 3L, n_rows = 500L,
  elapsed_mins = 120, manifest = list(sha256 = "abc123"),
  boot = list(
    replicates = data.frame(
      replicate = c(1L, 1L, 2L, 3L, 4L),
      parameter = c("base", "early.age", "base", "base", "base"),
      estimate = c(1, 0.5, 1.1, 0.9, 1.2)),
    summary = data.frame(parameter = "base", n = 4L, pct = 100),
    n_success = 4L, n_failed = 0L))
boot_health(bag)

# A screen that selected nothing outside the base model.
bag$boot$replicates <- bag$boot$replicates[-2, ]
boot_health(bag)
```

boot_pool_chunks	<i>Pool chunked bootstrap runs into one screen</i>
------------------	--

Description

Folds a list of chunk objects into a single object of the same shape, so a report reads a pooled screen and a single-run screen the same way.

Usage

```
boot_pool_chunks(chunks)
```

Arguments

chunks	A list of chunk objects, each carrying at least seed, n_boot, slentry, slstay, base_params, usable, max_steps, elapsed_mins, a manifest with an md5 or sha256, an engine provenance field (th_sha or th_version), and a boot list with replicates, n_success and n_failed.
--------	--

Details

Six things must agree for chunks to be draws from one screen, and each is checked rather than assumed. **None of them fails loudly on its own:**

- Two chunks sharing a **seed** contain literally the same replicates. Pooling them counts each twice, halves nothing, and reports a Monte-Carlo error smaller than the run actually has.
- A **dataset rewritten mid-run** gives chunks that each look fine and describe different cohorts.
- A different **step cap** changes the screen, not just its cost: under a tighter cap a variable that would have entered late never gets the chance.

Neither shows up as an error in a frequency table. Both show up as a slightly different number.

Absence is checked before agreement, and that order is the whole point. A field no chunk records formats identically in every chunk, so it passes unanimously and the accessor hands back NULL as the agreed value. A gate that cannot tell "everyone agrees" from "nobody recorded it" is not a gate, and it is worse than having none, because it reads as a check that happened. This was not hypothetical: max_steps was absent from every chunk in one test fixture and the suite was green.

Replicate ids run 1 . . n_boot **inside** each chunk, so they collide across chunks and are offset before stacking. Without that, replicate 1 of chunk 2 merges into replicate 1 of chunk 1 and a variable selected in both counts once instead of twice — understating every frequency, in a table that looks entirely ordinary.

The pooled summary is **recomputed from the pooled replicates**, never averaged across chunks: a percentage of a percentage is not a percentage of the whole, and the chunks need not be the same size.

Value

One object shaped like a single chunk, with boot\$replicates stacked and re-indexed, boot\$summary recomputed, and n_chunks, seeds and a joined seed string added.

See Also

[boot_chunk_files\(\)](#) to find them, [boot_shortfall\(\)](#) to check the pool is the run that was launched, and [boot_summary\(\)](#) for the per-variable frequencies.

Examples

```
# Two chunks as the hazard runner writes them: same screen, different seeds.
chunk <- function(seed) list(
  seed = seed, n_boot = 2L, slentry = 0.1, slstay = 0.05,
  base_params = "early.log_mu", usable = c(early = 10L), max_steps = 50L,
  elapsed_mins = 1, manifest = list(sha256 = "example"),
  th_version = "1.2.16",
  boot = list(n_success = 2L, n_failed = 0L, replicates = data.frame(
    replicate = c(1L, 1L, 2L), parameter = c("early.log_mu", "early.age",
      "early.log_mu"), estimate = c(0.1, 0.02, 0.2)))
)
pooled <- boot_pool_chunks(list(chunk(1), chunk(2)))
pooled$n_boot
pooled$boot$summary

# The same seed twice is the same replicates twice, so it is refused.
try(boot_pool_chunks(list(chunk(1), chunk(1))))
```

boot_predict_ci

Band an estimate by bootstrap resampling

Description

The R port of %BNMNR and %BNPREV (bn.* in the CORR macro library). Resamples data, computes statistic on each replicate, and reports the percentile intervals of each estimated quantity.

Usage

```
boot_predict_ci(
  data,
  statistic,
  n_rep = 1000,
  fraction = 1,
  id = NULL,
  max_attempts = 10 * n_rep,
  seed = NULL,
  ...
)
```

Arguments

<code>data</code>	A data frame.
<code>statistic</code>	Function of <code>(data, ...)</code> returning a named numeric vector of one replicate's estimates, or NULL if the replicate failed. %BNMNR equivalent: the PROC NLMIXED block.
<code>n_rep</code>	Number of valid replicates (%BNMNR RESAMPL=). Divergence: the macro defaults to 100; this defaults to 1000, matching <code>boot_select()</code> in this package. A hundred replicates puts the 2.5th percentile on the third order statistic, where it is visibly unstable. Pass <code>n_rep = 100</code> to reproduce the macro.
<code>fraction</code>	Fraction of units drawn per replicate (%BNMNR FRACTION=). Applied, as the macro applies it – unlike %bootreg, which computes it and then draws the full size anyway.
<code>id</code>	Column naming the resampling unit, or NULL to draw rows. When given, units are drawn with replacement and renumbered , so a unit drawn twice becomes two distinct units; the new id is in a <code>.boot_unit</code> column. %BNMNR equivalent: the patient-level INDAT= joined to the repeated INMULT=.
<code>max_attempts</code>	Budget of draws before giving up. Divergence: %BNMNR has no cap and never terminates when every replicate fails. Pass Inf to restore that.
<code>seed</code>	Optional integer for reproducibility (%BNMNR SEED=).
<code>...</code>	Passed to <code>statistic</code> .

Details

This is the **interval** branch. It is not variable selection: nothing is chosen, the replicates are a distribution rather than a vote, and there is no NA semantics. For selection, see `boot_select()`.

`statistic` is the fitter contract with the selection semantics removed. It takes one resampled frame and returns a **named numeric vector** of that replicate's estimates, or NULL when the replicate did not fit. The names are the quantities banded, so a caller wanting a curve evaluates it on their own grid inside `statistic` and names the elements – which is what the macro does too, in a PROC NLMIXED block the analyst edits.

The bands are pointwise. Each quantity is summarised independently, as %BNMNR does. A curve drawn through `c11_p95` is the 2.5th percentile at each point, not a 95% region for the curve.

No argument sets a coverage level, here or anywhere in this package. The macros do not take one either: they hardcode `PCTLPTS=2.5 16 50 84 97.5` and return both bands in columns named for their coverage. Percentiles use `stats::quantile()` type = 4, which is SAS's `PCTLDEF=1`. A coverage-shaped name passed through `...` (`conf`, `level`, `alpha`, `probs`, `conf.level`, `coverage`, `ci`, `confidence`, `conf_level`) is refused with an error rather than silently forwarded to `statistic` and ignored.

Value

An object of class `boot_intervals`. `$estimates` is a matrix with one row per valid replicate and one column per quantity; `$intervals` is the per-quantity table with columns `parameter`, `c11_p95`, `c11_p68`, `median`, `clu_p68`, `clu_p95`. `$control` records the run's settings.

See Also

[boot_select\(\)](#) for the selection branch.

Examples

```
# The names of the returned vector are the quantities banded. Here, two
# summaries of the same column; in a real screen they are typically a
# fitted curve evaluated on a grid.
df <- data.frame(x = rnorm(200))
est <- function(d, ...) c(mean = mean(d$x), sd = sd(d$x))

r <- boot_predict_ci(df, est, n_rep = 100, seed = 42)
r
summary(r)
```

boot_provenance

Where a bootstrap screen came from

Description

The facts a reader needs to judge, reproduce or reject a screen: how many replicates, under which entry and stay criteria, over how many candidates and rows, at what cost, against which dataset and which build of the fitting engine.

Usage

```
boot_provenance(bag)
```

Arguments

bag A bootstrap screen: the object [boot_pool_chunks\(\)](#) returns, or a single unchunked run of the same shape. Checked by [boot_validate\(\)](#).

Details

requested and usable are **per phase** on a multiphase screen, and rendering them is the reason this function exists rather than a bare `data.frame()` call in each report. They are collapsed to one labelled string per row – “early 230, late 230” – and never summed: the candidate pool is *offered* to each phase, so 230 and 230 is one pool seen twice rather than 460 candidates.

Two facts travel with a tag that says what kind of evidence they are:

- The **dataset checksum** is “<algo>:<digest>”. An md5 and a sha256 of the same file are different strings, and of different files may not be, so a bare digest recorded without its algorithm is not evidence of anything.
- The **fitting engine** is “sha:<commit>” in preference to “version:<string>”. One real package version existed as two codebases, one with a selection criterion and one without, and the selection criterion is precisely the thing that decides what a screen selects.

Elapsed time is reported as **summed CPU hours**, not wall clock: `boot_pool_chunks()` sums `elapsed_mins` across chunks, so on a chunked run this is total compute and chunks run in parallel finish in a fraction of it.

Seeds are summarised here and listed by `boot_seeds()`. At 25 chunks the joined seed string is a single 270-character cell.

Value

A data frame with columns `item` and `value`, both character, one row per fact. **The row count does not depend on the shape of any field**

- a per-phase requested yields the same rows as a scalar one.

See Also

`boot_seeds()` for the seeds themselves, `boot_health()` for whether the screen ran, and `boot_shortfall()` for whether the pool is the run that was launched.

Examples

```
bag <- list(
  n_boot = 500L, n_chunks = 2L, seed = "101, 202", seeds = c(101, 202),
  slentry = 0.07, slstay = 0.05, base_params = "base",
  requested = c(early = 230L, late = 230L),
  usable = c(early = 226L, late = 226L),
  n_rows = 4000L, elapsed_mins = 150, th_sha = "deadbeef",
  manifest = list(sha256 = "abc123"),
  boot = list(replicates = data.frame(replicate = 1L, parameter = "base",
                                     estimate = 1),
             summary = data.frame(parameter = "base", n = 1L, pct = 100),
             n_success = 500L, n_failed = 0L))
boot_provenance(bag)
boot_seeds(bag)
```

boot_seeds

Every seed a bootstrap screen used

Description

One row per chunk, so that a rerun is reproducible and a duplicate is visible.

Usage

```
boot_seeds(bag)
```

Arguments

`bag` A bootstrap screen. Checked by `boot_validate()`.

Details

Two chunks sharing a seed contain literally the **same** replicates: pooling them counts each twice and reports a Monte-Carlo error smaller than the run actually has. `boot_pool_chunks()` refuses that outright, so a pooled screen cannot reach here with a duplicate. This table is what lets you check a *single* run, and what lets a reader reproduce either.

A single unchunked run never went through `boot_pool_chunks()` and carries only the scalar seed its runner wrote, so that is used when seeds is absent. Seeds are formatted without scientific notation: a seed printed as 1.23e+08 cannot be typed back in.

Value

A data frame with columns chunk (integer) and seed (character).

See Also

`boot_provenance()`, which summarises the count.

Examples

```
bag <- list(
  n_boot = 500L, seed = 4242, slentry = 0.07, slstay = 0.05,
  base_params = "base", requested = 230L, usable = 226L,
  n_rows = 4000L, elapsed_mins = 150, manifest = list(sha256 = "abc123"),
  boot = list(replicates = data.frame(replicate = 1L, parameter = "base",
                                     estimate = 1),
              summary = data.frame(parameter = "base", n = 1L, pct = 100),
              n_success = 500L, n_failed = 0L))
boot_seeds(bag)
```

boot_select

Build a model by bootstrap resampling

Description

The R port of %bootreg (bootstrap.models.sas in the CORR macro library). Fits fitter on each of n_rep bootstrap replicates and records the coefficients each model kept.

Usage

```
boot_select(
  data,
  formula,
  fitter,
  n_rep = 1000,
  fraction = 1,
  select = c("stepwise", "none"),
  sle = 0.1,
```

```

    sls = 0.05,
    max_steps = 0,
    max_attempts = 10 * n_rep,
    seed = NULL
)

```

Arguments

data	A data frame.
formula	Model formula offering the candidate terms.
fitter	A fitter such as <code>fit_logistic()</code> , <code>fit_linear()</code> or <code>fit_cox()</code> . <code>%bootreg</code> equivalent: <code>PROC=</code> .
n_rep	Number of valid models to generate (<code>%bootreg RESAMPL=</code>). Replicates whose fit fails are redrawn and do not count, matching the macro's <code>&regrc</code> check.
fraction	Fraction of <code>nrow(data)</code> to draw per replicate (<code>%bootreg FRACTION=</code>). Divergence: the macro documents this parameter but never applies it - it computes <code>ds_size * fraction</code> , prints it, and then always draws <code>ds_size</code> rows. This implementation applies it. Pass <code>fraction = 1</code> (the default) to match SAS behaviour exactly.
select	"stepwise" or "none" (<code>%bootreg SELECT=/FIXED=</code>).
sle, sls	Entry and retention p-value thresholds (<code>%bootreg SLE=, SLS=</code>). Each fitter pins its own entry and removal test to match its <code>PROC=</code> - see <code>fit_linear()</code> , <code>fit_logistic()</code> and <code>fit_cox()</code> - so <code>sle</code> and <code>sls</code> are exactly what they mean in the job being ported. Model fitting is still not parity-tested - see the package's design spec - but these two arguments now select rather than only being recorded. Divergence: candidates are flat term labels, not a hierarchy, and <code>fit_linear()</code> 's removal test respects marginality while <code>fit_logistic()</code> 's and <code>fit_cox()</code> 's Wald removal test does not - so a screen can keep an interaction without either main effect on the latter two, which <code>PROC LOGISTIC</code> 's default <code>HIERARCHY=SINGLE</code> would not do. Not implemented; see D5 in the README's divergence register.
max_steps	Maximum selection steps (<code>%bootreg MAXSTEP=</code>). 0 means no restriction, implemented as a budget scaled to the model - ten passes over the candidate terms, floored at 1000 - which no realistic model reaches. It bounds the p-value step-wise driver's forward/backward loop (<code>.pv_stepwise()</code> in <code>R/stepwise.R</code>) rather than leaving it genuinely unbounded.
max_attempts	Budget of resampling attempts before giving up. Divergence: <code>%bootreg</code> has no such cap - its loop advances only on a successful fit, so a model that fails on every replicate never terminates. That is survivable in a batch job with an operator watching the log; under <code>R CMD</code> check it is an unbounded hang with no diagnostic. Exhausting the budget raises an error reporting how many valid models were obtained. Pass <code>Inf</code> to restore the macro's uncapped behaviour.
seed	Optional integer for reproducibility.

Details

A term the model did not select is NA in that replicate's row. That missingness is the whole design: `boot_summary()` counts non-missing values down a column, so *n* is the number of replicates that chose the term. Do not fill those gaps with zero.

fitter is what PROC= was. Adding a model family means writing a fitter, not another pipeline.

Value

An object of class `boot_selection`. `$coefficients` is a matrix with one row per valid replicate and one column per candidate term, NA where the term was not selected. `$control` records the run's own settings – method, sle, sls, max_steps, fraction, seed, n_rows, n_terms, elapsed_mins and the package version – so that a bag builder `boot_bag()` can take a provenance record from the run rather than from what a caller retypes. `$call` cannot serve that purpose: `match.call()` omits every argument left at its default.

See Also

`boot_summary()` for the per-variable selection frequencies, and `boot_clusters()` for the same question asked of a correlated group. `fit_linear()`, `fit_logistic()` and `fit_cox()` are the supplied fitters.

Examples

```
set.seed(1)
n <- 300
x1 <- rnorm(n)
df <- data.frame(y = 2 * x1 + rnorm(n), x1 = x1,
                 x2 = rnorm(n), noise = rnorm(n))

# n_rep is RESAMPL=. It counts VALID models, so a replicate whose fit fails
# is redrawn and does not consume one -- the macro's &regrc check.
fit <- boot_select(df, y ~ x1 + x2 + noise, fit_linear,
                  n_rep = 50, seed = 42)

fit

# The replicate table itself, one row per model, NA where a term was not
# selected. This is what %bootreg writes to OUTEST=.
head(fit$coefficients, 3)

boot_summary(fit)
```

Description

Chunks land over hours, and pooling whatever is on disk mid-run is a reasonable thing to want. The hazard is that **a partial pool produces a report that is wrong in no visible way**: every health check passes, every frequency is honestly computed, and only the denominator is not the intended one.

Usage

```
boot_shortfall(bag, expect_chunks, expect_boot)
```

Arguments

bag	A pooled object from <code>boot_pool_chunks()</code> , or a single run.
expect_chunks	How many chunks were launched.
expect_boot	How many replicates were wanted in total.

Details

The expected totals are the only thing that can catch it, and they must come from **outside** the chunks — nothing in a chunk knows how many siblings it was launched alongside.

Replicates are checked as well as chunks, because a chunk that ran short is not a missing chunk and counting chunks alone would call that complete. An **over**-count is reported too: it means either the expectation is stale or a stray chunk from another run is being pooled, and both are worth saying.

Value

NULL when the pool is the run that was launched; otherwise a sentence saying how it falls short, suitable for a callout.

See Also

`boot_pool_chunks()`

Examples

```
# Two chunks as the hazard runner writes them: same screen, different seeds.
chunk <- function(seed) list(
  seed = seed, n_boot = 2L, slentry = 0.1, slstay = 0.05,
  base_params = "early.log_mu", usable = c(early = 10L), max_steps = 50L,
  elapsed_mins = 1, manifest = list(sha256 = "example"),
  th_version = "1.2.16",
  boot = list(n_success = 2L, n_failed = 0L, replicates = data.frame(
    replicate = c(1L, 1L, 2L), parameter = c("early.log_mu", "early.age",
      "early.log_mu"), estimate = c(0.1, 0.02, 0.2)))
)
pooled <- boot_pool_chunks(list(chunk(1), chunk(2)))

# The run that was launched: NULL.
boot_shortfall(pooled, expect_chunks = 2, expect_boot = 4)
```

```
# A third chunk has not landed yet.
boot_shortfall(pooled, expect_chunks = 3, expect_boot = 6)
```

boot_summary

Summarise bootstrap replicates into selection frequencies

Description

The R port of %SUMBOOT (bootstrap.summary.sas in the CORR macro library). For each candidate term: how often it was selected across replicates, and the distribution of its coefficient when it was.

Usage

```
boot_summary(x)
```

Arguments

x A boot_selection from [boot_select\(\)](#), or a numeric matrix with one row per replicate and one column per term.

Details

n counts replicates in which the term was selected, because [boot_select\(\)](#) leaves an unselected term NA. pct is $100 * n / n_rep$ - the denominator is the replicate count, so pct reads as a selection frequency.

This function is held to **exact parity** with %SUMBOOT: given the same replicate table it must produce the same n, pct, mean, sd, min and max. Resampling and model fitting are not parity-tested; see the package's design spec.

Value

A data frame with columns variable, n, pct, mean, sd, min, max, sorted by descending n.

See Also

[boot_select\(\)](#), which produces the replicates; [boot_clusters\(\)](#) for the at-least-one count across a group of correlated terms.

Examples

```
set.seed(1)
n <- 300
x1 <- rnorm(n)
df <- data.frame(y = 2 * x1 + rnorm(n), x1 = x1,
                 x2 = rnorm(n), noise = rnorm(n))
```

```
fit <- boot_select(df, y ~ x1 + x2 + noise, fit_linear,
  n_rep = 50, seed = 42)
boot_summary(fit)

# It also takes a bare replicate matrix, which is how the parity fixtures
# are checked against %SUMBOOT without needing any cohort data.
m <- matrix(c(1, 2, 3, 4, 2, NA, 4, NA), nrow = 4,
  dimnames = list(NULL, c("x1", "x2")))
boot_summary(m)
```

boot_validate

Check that a bootstrap bag has the shape a report reads

Description

The runner that produces a bootstrap screen is a study file, not a templated one, so the fields a report reads are a contract between two files that no shared function enforces. This is that function.

Usage

```
boot_validate(bag)
```

Arguments

bag	A bootstrap screen: the object <code>boot_pool_chunks()</code> returns, or a single unchunked run of the same shape.
-----	--

Details

It replaces the contract chunk that each bootstrap report carried, and it differs from that chunk in the way that matters: **it checks the shape of every field it names, not merely that the name exists**. The chunk passed a real bag happily while requested was a length-2 vector the report could not render, and the resulting defect shipped in three releases.

Three shapes are checked, because three things can be true of a field:

- **Scalar** - `n_boot`, `seed`, `slentry`, `slstay`, `n_rows` and `elapsed_mins` are each one value. A length-2 `n_boot` means the pool was written per phase into a field that is not per phase, and every count downstream is then ambiguous.
- **Per phase** - `requested` and `usable` may be a vector, and on a multiphase screen they are: the candidate pool is *offered* to each phase, so `c(early = 230, late = 230)` is one pool seen twice. They must be numeric and carry at least one value. `integer(0)` satisfies "is present" and then contributes no element rather than one, which collapses any table built from it.
- **Any shape** - `base_params` is read whole, and `boot` must carry replicates, summary, `n_success` and `n_failed`. `boot` being present says nothing about what is inside it. `manifest` is the one exception: it must be a **list**, because it is indexed by name, and `manifest[["sha256"]]` on a named atomic vector lacking that name is not `NULL` but an error – subscript out of bounds, naming neither the field nor the file.

`boot$summary` is additionally checked for the columns parameter, `n` and `pct` – the ones a report reads. It is not required to carry the full shape `boot_bag()` writes, because a runner may report a frequency table and nothing more. A summary keyed variable is `boot_summary()`'s shape rather than a bag's, and is refused by name.

Every failure is reported at once. An author fixing a runner wants the whole list; one at a time turns a single fix into five renders.

`dropped`, `free_sd`, `n_chunks`, `seeds` and the engine provenance fields are **not** required. A single unchunked run legitimately carries none of them, and `boot_provenance()`, `boot_health()` and `boot_dropped()` each say what they do in their absence.

`free_sd` is the one of those with a shape to it: absent is fine, present means a single value. It is the standard deviation of the *first* free base parameter, so one number is all it can be. A per-phase value validated cleanly and then met `boot_health()`'s scalar if, which R makes an error naming neither the field nor the function.

Value

`invisible(TRUE)`. Called for the error it raises otherwise.

See Also

`boot_provenance()`, `boot_frequencies()`, `boot_concepts()` and `boot_health()`, each of which calls this first; `boot_shortfall()` for the different question of whether the pool is the run that was launched.

Examples

```
bag <- list(
  n_boot = 4L, seed = 101, slentry = 0.07, slstay = 0.05,
  base_params = "base", requested = 4L, usable = 3L,
  n_rows = 500L, elapsed_mins = 120, manifest = list(sha256 = "abc123"),
  boot = list(
    replicates = data.frame(replicate = c(1L, 2L),
                           parameter = c("early.age", "early.age"),
                           estimate = c(0.5, 0.6)),
    summary = data.frame(parameter = "early.age", n = 2L, pct = 50),
    n_success = 4L, n_failed = 0L))
boot_validate(bag)

# A per-phase `requested` is valid, not a defect: the pool is offered to
# each phase, so this is one pool seen twice.
bag$requested <- c(early = 230L, late = 230L)
boot_validate(bag)

# A per-phase `n_boot` is a defect, and this is the shape that shipped.
bag$n_boot <- c(500L, 500L)
try(boot_validate(bag))
```

fit_cox

*Fit a Cox proportional-hazards model for one bootstrap replicate***Description**

Fit a Cox proportional-hazards model for one bootstrap replicate

Usage

```
fit_cox(data, formula, select)
```

Arguments

data	A data frame - one bootstrap replicate.
formula	Model formula offering the candidate terms.
select	List with method ("stepwise" or "none"), sle, sls, max_steps. %bootreg equivalents: SELECT=, SLE=, SLS=, MAXSTEP=.

Details

Divergence: PROC PHREG SELECTION=STEPWISE enters a term on the score chi-square. R has no score test for a Cox model - anova.coxph() accepts test = "Rao" but silently ignores it and always returns the likelihood-ratio test - so entry here is by likelihood ratio. The two agree asymptotically and differ only for a term sitting on the entry threshold, so a screen will usually select the same set and may occasionally differ on a borderline candidate. Removal is Wald, matching the macro.

Value

Named numeric vector of kept coefficients, or NULL if the fit errored. Cox models carry no intercept, so none appears in the result - which means a replicate where selection kept nothing returns a **zero-length** vector, not NULL, and counts as a valid replicate.

See Also

[fit_linear\(\)](#) and [fit_logistic\(\)](#); [boot_select\(\)](#).

Examples

```
if (requireNamespace("survival", quietly = TRUE)) {
  set.seed(2)
  n <- 200
  df <- data.frame(time = rexp(n), status = rbinom(n, 1, 0.7),
                   x1 = rnorm(n), noise = rnorm(n))

  fit_cox(df, survival::Surv(time, status) ~ x1 + noise,
          list(method = "stepwise", sle = 0.10, sls = 0.05, max_steps = 0))
}
```

fit_linear	<i>Fit a linear model for one bootstrap replicate</i>
------------	---

Description

Fit a linear model for one bootstrap replicate

Usage

```
fit_linear(data, formula, select)
```

Arguments

data	A data frame - one bootstrap replicate.
formula	Model formula offering the candidate terms.
select	List with method ("stepwise" or "none"), sle, sls, max_steps. %bootreg equivalents: SELECT=, SLE=, SLS=, MAXSTEP=.

Details

Under select = "stepwise", entry and removal both test the partial F, matching PROC REG SELECTION=STEPWISE.

Value

Named numeric vector of kept coefficients, or NULL if the fit failed. NULL tells `boot_select()` to discard the replicate and draw another, reproducing %bootreg's ®rc check. A **warning** does not count as failure - the fit is kept, matching the macro, whose ®rc is a return code that warnings do not set. A zero-length result is likewise not a failure: it means selection kept no terms, which is a valid replicate.

See Also

`fit_logistic()` and `fit_cox()`, the other two PROC= values %bootreg supports; `boot_select()`, which calls a fitter once per replicate.

Examples

```
set.seed(1)
n <- 200
x1 <- rnorm(n)
df <- data.frame(y = 2 * x1 + rnorm(n), x1 = x1, noise = rnorm(n))

# `method = "stepwise"` is the macro's default; `"none"` is FIXED=1, which
# fits the model as written and bootstraps its coefficients instead.
fit_linear(df, y ~ x1 + noise,
  list(method = "stepwise", sle = 0.10, sls = 0.05, max_steps = 0))

fit_linear(df, y ~ x1 + noise,
  list(method = "none", sle = 0.10, sls = 0.05, max_steps = 0))
```

fit_logistic

Fit a logistic model for one bootstrap replicate

Description

Fit a logistic model for one bootstrap replicate

Usage

```
fit_logistic(data, formula, select)
```

Arguments

data	A data frame - one bootstrap replicate.
formula	Model formula offering the candidate terms.
select	List with method ("stepwise" or "none"), sle, sls, max_steps. %bootreg equivalents: SELECT=, SLE=, SLS=, MAXSTEP=.

Details

Under select = "stepwise", entry tests the score chi-square and removal tests Wald, matching PROC LOGISTIC SELECTION=STEPWISE.

Value

Named numeric vector of kept coefficients, or NULL if the fit errored or did not converge. Warnings - notably "fitted probabilities numerically 0 or 1" on a quasi-separated replicate - do not discard a converged fit.

See Also

[fit_linear\(\)](#) and [fit_cox\(\)](#); [boot_select\(\)](#).

Examples

```
set.seed(1)
n <- 200
x1 <- rnorm(n)
df <- data.frame(y = as.integer(x1 + rnorm(n) > 0), x1 = x1,
                 noise = rnorm(n))

fit_logistic(df, y ~ x1 + noise,
             list(method = "stepwise", sle = 0.10, sls = 0.05,
                  max_steps = 0))
```

Index

boot_bag, 2
boot_bag(), 19, 23
boot_chunk_files, 4
boot_chunk_files(), 13
boot_clusters, 5
boot_clusters(), 6, 7, 19, 21
boot_concepts, 6
boot_concepts(), 8, 10, 23
boot_dropped, 7
boot_dropped(), 3, 10, 23
boot_frequencies, 9
boot_frequencies(), 3, 6–8, 23
boot_health, 10
boot_health(), 2, 3, 16, 23
boot_pool_chunks, 12
boot_pool_chunks(), 3, 4, 11, 15–17, 20, 22
boot_predict_ci, 13
boot_provenance, 15
boot_provenance(), 3, 11, 17, 23
boot_seeds, 16
boot_seeds(), 16
boot_select, 17
boot_select(), 2, 3, 5, 14, 15, 21, 24–26
boot_shortfall, 19
boot_shortfall(), 11, 13, 16, 23
boot_summary, 21
boot_summary(), 3, 5, 9, 10, 13, 19, 23
boot_validate, 22
boot_validate(), 2, 3, 6, 8–10, 15, 16

fit_cox, 24
fit_cox(), 18, 19, 25, 26
fit_linear, 25
fit_linear(), 18, 19, 24, 26
fit_logistic, 26
fit_logistic(), 18, 19, 24, 25

stats::quantile(), 14