Package: EDI (via r-universe)

October 18, 2026

Type Package

Title Experimental Design and Inference

Version 1.0.3

Description Implements a comprehensive suite of experimental designs, both fixed (e.g., block, stratified, matched-pair, cluster, factorial, and mixed-integer-programming-based optimal designs) and sequential (including matching-on-the-fly designs, biased coin designs, and covariate-adaptive urn designs that assign treatment one subject at a time while maintaining covariate balance), for continuous, incidence, count, proportion, survival, and ordinal response types. For each design and response type combination, provides the corresponding inference procedures, including exact, asymptotic, distribution-free, and resampling-based (bootstrap, jackknife, and randomization) methods, so that estimation and testing are always matched to how the data were generated. An 'InferenceSuite' facility runs all applicable inference procedures for a given design and response type at once and reports a single Cauchy-combined p-value summarizing their evidence. A built-in simulation framework supports power analysis and operating-characteristic studies across designs, response types, and inference procedures, with optional parallelization via 'mirai'. Missing covariate data is handled automatically via built-in imputation. Core numerical routines are implemented in C++ via 'Rcpp' for speed on large designs and simulation studies. Machine-specific tuning for optimization is included.

License GPL-3

Encoding UTF-8

Language en-US

Depends R (>= 3.5.0)

LinkingTo Rcpp, RcppEigen, RcppNumerical

Imports R6, checkmate, Rcpp, data.table, survival, missRanger, missForest, numDeriv, digest, methods, MASS, RhpcBLASctl, randomizr

Suggests anticlust, blockTools, clinfun, coin, dplyr, mirai, aftgee, betareg (>= 3.2-6), copula, doParallel, ggplot2, fixest, gamlss.dist, geepack, glmmTMB, glmnet, icenReg, interval, jsonlite, knitr, lme4, multgee, nbpMatching, nlme, nnet, optmatch, ordinal, pscl, qpdf, quantreg, R.utils, RcppXPtrUtils, Rfit, rmarkdown, testthat, VGAM, ompr, ompr.roi, ROI.plugin.glpk, sandwich, withr

VignetteBuilder knitr

Collate 'design_abstract.R' 'design_blocking_abstract.R' 'design_matching_abstract.R' 'design_fixed_abstract.R' 'design_component_registry.R' 'design_class_factory.R' 'design_fixed_greedy.R' 'helper_optimal_shared.R' 'helper_optimal_milp_solvers.R' 'helper_user_compiled_fn.R' 'helper_optimal_annealing.R' 'design_fixed_greedy_d_optimal.R' 'design_fixed_bernoulli.R' 'design_fixed_binary_match.R' 'design_fixed_blocked_cluster.R' 'design_fixed_blocking.R' 'design_fixed_cluster.R' 'design_fixed_factorial.R' 'design_fixed_ibcrd.R' 'design_fixed_matching_greedy_pair_switching.R' 'design_fixed_optimal.R' 'design_fixed_optimal_blocks.R' 'design_fixed_rerandomization.R' 'design_observational.R' 'design_observational_blocks.R' 'design_observational_matching.R' 'design_seq_one_by_one_abstract.R' 'design_custom_extensions.R' 'design_seq_one_by_one_atkinson.R' 'design_seq_one_by_one_bernoulli.R' 'design_seq_one_by_one_efron.R' 'design_seq_one_by_one_ibcrd.R' 'design_seq_one_by_one_KK14.R' 'design_seq_one_by_one_KK21_stepwise.R' 'design_seq_one_by_one_KK21.R' 'design_seq_one_by_one_pocock_simon.R' 'design_seq_one_by_one_random_block_size.R' 'design_seq_one_by_one_spbr.R' 'design_seq_one_by_one_urn.R' 'design_class_registry.R' 'EDI.R' 'helper_glm_fit.R' 'globals.R' 'comprehensive_slow_paths.R' 'helper_additional_asserts.R' 'contracts_mixins.R' 'helper_robust_sandwich.R' 'inference_all_abstract_mle_or_KM_summary_table.R' 'inference_ext_ci_inversion.R' 'inference_ext_information_matrix.R' 'inference_ext_likelihood_test_memoization.R' 'inference_mixin_off_optimum_likelihood_eval.R' 'inference_all_abstract_asymp_lik.R' 'inference_all_abstract_marginal_estimand.R' 'inference_ext_bartlett_approx.R' 'inference_mixin_cordeiro_ferrari_approx.R' 'inference_mixin_lemonte_gradient_approx.R' 'inference_mixin_kk_gee_shared.R'

'inference_mixin_kk_glmm_shared.R'
'helper_reused_worker_private_cache.R'
'inference_mixin_kk_passthrough.R'
'inference_mixin_kk_passthrough_compound.R'
'helper_bootstrap_ci.R' 'contracts_resampling_draws.R'
'inference_ext_bca_bootstrap_ci.R'
'inference_ext_exchangeable_resampling_units.R'
'inference_ext_minimum_volatility_selector.R'
'inference_ext_m_out_of_n_bootstrap.R'
'inference_ext_prw_subsampling.R'
'inference_all_abstract_non_param_boot.R'
'inference_all_abstract_rand_bootstrap.R'
'inference_all_abstract_rand_bootstrap_ci.R'
'inference_all_abstract_bayesian_bootstrap.R'
'inference_all_abstract_jackknife.R'
'inference_all_abstract_exact.R'
'inference_all_abstract_asymp.R'
'inference_ext_param_bootstrap_estimate.R'
'inference_all_abstract_param_boot.R'
'inference_all_abstract_asymp_lik_std_mod_cache.R'
'inference_all_abstract_count_likelihood.R'
'inference_ext_quantile_rand_ci.R'
'inference_all_abstract_rand_ci.R'
'inference_ext_sequential_mc_pval.R'
'inference_all_abstract_rand.R' 'inference_all_abstract.R'
'inference_all_abstract_KK_passthrough_compound.R'
'inference_all_abstract_quantile_rand_ci.R'
'inference_all_KK_mean_diff_IVWC.R'
'inference_all_KK_quantile_regr_ivwc_abstract.R'
'inference_all_KK_quantile_regr_one_lik_abstract.R'
'inference_all_KK_wilcox_ivwc.R' 'inference_all_average_diff.R'
'inference_all_simple_mean_diff_pooled_var.R'
'inference_all_simple_wilcox.R'
'inference_continuous_KK_bai_abstract.R'
'inference_continuous_KK_glmm.R'
'inference_continuous_KK_ols_ivwc.R'
'inference_continuous_KK_ols_one_lik.R'
'inference_continuous_KK_quantile_regr_ivwc.R'
'inference_continuous_KK_quantile_regr_one_lik.R'
'inference_continuous_KK_robust_regr_ivwc.R'
'inference_continuous_KK_robust_regr_one_lik.R'
'inference_continuous_KK14_bai.R'
'inference_continuous_KK21_bai.R' 'inference_continuous_lin.R'
'inference_continuous_ols.R'
'inference_continuous_quantile_regr.R'
'inference_continuous_robust_regr.R'
'inference_count_composite_likelihood.R'
'inference_count_KK_combined.R'

'inference_count_KK_cond_poisson.R' 'inference_count_KK_gee.R'
'inference_count_negbin.R' 'inference_count_poisson.R'
'inference_count_quasipoisson.R'
'inference_count_robust_poisson.R'
'inference_count_zero_augmented_poisson_abstract.R'
'inference_count_hurdle.R' 'inference_count_zero_inflated.R'
'inference_custom_extensions.R' 'inference_rand_custom.R'
'inference_helpers_zhang.R'
'inference_incidence_binomial_identity.R'
'inference_incidence_cmh.R'
'inference_incidence_exact_binomial.R'
'inference_incidence_exact_zhang.R'
'inference_incidence_extended_robins.R' 'helper_gcomp.R'
'helper_marginal_estimand.R'
'inference_incidence_gcomp_abstract.R'
'inference_incidence_gcomp.R'
'inference_incidence_KK_cond_logit_glmm_abstract.R'
'inference_incidence_KK_cond_logit_glmm.R'
'inference_incidence_KK_cond_logit.R'
'inference_incidence_KK_combined.R'
'inference_incidence_KK_gcomp_abstract.R'
'inference_incidence_KK_marginal_abstract.R'
'inference_incidence_KK_marginal.R'
'inference_incidence_KK_newcombe_ivwc_univ.R'
'inference_incidence_log_binomial.R'
'inference_incidence_logit.R'
'inference_incidence_miettinen_nurminen_univ.R'
'inference_incidence_modified_poisson.R'
'inference_incidence_newcombe_univ.R'
'inference_incidence_probit.R'
'inference_incidence_risk_diff.R' 'inference_incid_wald.R'
'inference_indicidence_exact_fisher.R'
'inference_ordinal_adj_cat_logit.R'
'inference_ordinal_cauchit.R' 'inference_ordinal_cloglog.R'
'inference_ordinal_gcomp.R'
'inference_ordinal_jonckheere_terpstra_test.R'
'helper_ordinal_mixed_fit.R'
'inference_ordinal_KK_clmm_abstract.R'
'inference_ordinal_KK_combined.R'
'inference_ordinal_KK_cond_adj_cat_logit.R'
'inference_ordinal_KK_cond_logit_abstract.R'
'inference_ordinal_ordered_probit.R'
'inference_ordinal_paired_sign_test.R'
'inference_ordinal_partial_proportional_odds.R'
'inference_ordinal_proportional_odds.R'
'inference_ordinal_ridit.R'
'inference_ordinal_stereotype_logit.R'
'inference_proportion_beta.R'

'inference_proportion_fractional_logit.R'
'inference_proportion_gcomp.R'
'inference_proportion_KK_combined.R'
'inference_proportion_KK_quantile_regr_ivwc.R'
'inference_proportion_KK_quantile_regr_one_lik.R'
'inference_proportion_quantile_regr.R'
'inference_proportion_zero_one_inflated_beta.R'
'inference_suite.R' 'inference_survival_coxph.R'
'inference_survival_dep_cens_transform.R'
'inference_survival_gehan_wilcox.R'
'inference_survival_GLMM_weibull_frailty_loggamma.R'
'inference_survival_KK_lwa_cox_ivwc_abstract.R'
'inference_survival_KK_lwa_cox_one_lik_abstract.R'
'inference_survival_KK_lwa_cox.R'
'inference_survival_KK_rank_regr_ivwc_abstract.R'
'inference_survival_KK_rank_regr.R'
'inference_survival_KK_strat_cox.R'
'inference_survival_GLMM_weibull_frailty_normal.R'
'inference_survival_KK_weibull_marginal.R'
'inference_survival_km_diff.R' 'inference_survival_log_rank.R'
'inference_survival_rmst.R' 'inference_survival_strat_cox.R'
'inference_survival_weibull.R' 'inference_class_registry.R'
'helper_package_checks.R' 'helper_math.R'
'helper_model_matrix.R' 'helper_response_asserts.R'
'helper_robust_regression.R' 'helper_rcpp_doc_stubs.R'
'helper_matching.R' 'helper_survival_fits.R' 'helper_zoib.R'
'helper_inference_survival_turnbull.R' 'RcppExports.R'
'simulations_framework.R' 'simulation_framework_report.R'
'local_machine_tuning_synthetic_fixtures.R'
'local_machine_tuning_harness.R' 'local_machine_tuning_axes.R'
'local_machine_tuning_persistence.R'
'local_machine_tuning_correctness.R' 'local_machine_tuning.R'
'zzz.R'

URL https://github.com/kapelner/EDI, https://kapelner.github.io/EDI/, https://pypi.org/project/edi-kernels/

BugReports https://github.com/kapelner/EDI/issues

Config/roxygen2/version 8.0.0.9000

Repository https://kapelner.r-universe.dev

Date/Publication 2026-10-18 14:21:17 UTC

RemoteUrl https://github.com/kapelner/EDI

RemoteRef HEAD

RemoteSha c55198fef97b7ed24f1ea41e6556c7b88715b3ca

RemoteSubdir R/EDI

Contents

build_cox_data_cache_cpp . . . 10
build_stratified_cox_data_cache_cpp . . . 11
clear_local_EDI_optimization . . . 13
DesignFixedBernoulli . . . 14
DesignFixedBinaryMatch . . . 16
DesignFixedBlockedCluster . . . 19
DesignFixedBlocking . . . 21
DesignFixedCluster . . . 24
DesignFixedFactorial . . . 26
DesignFixedGreedy . . . 28
DesignFixedGreedyDOptimal . . . 30
DesignFixediBCRD . . . 35
DesignFixedMatchingGreedyPairSwitching . . . 37
DesignFixedOptimal . . . 39
DesignFixedOptimalBlocks . . . 45
DesignFixedRerandomization . . . 49
DesignSeqOneByOneAtkinson . . . 52
DesignSeqOneByOneBernoulli . . . 54
DesignSeqOneByOneEfron . . . 56
DesignSeqOneByOneiBCRD . . . 58
DesignSeqOneByOneKK14 . . . 60
DesignSeqOneByOneKK21 . . . 63
DesignSeqOneByOneKK21stepwise . . . 67
DesignSeqOneByOnePocockSimon . . . 71
DesignSeqOneByOneRandomBlockSize . . . 73
DesignSeqOneByOneSPBR . . . 75
DesignSeqOneByOneUrn . . . 77
EDI . . . 79
edi_build_info_cpp . . . 80
EDI_COMPREHENSIVE_SLOW_PATHS . . . 81
exact_jonckheere_terpstra_pval_cpp . . . 82
fast_beta_regression . . . 84
fast_beta_regression_with_var . . . 85
fast_coxph_regression . . . 87
fast_coxph_regression_cpp . . . 89
fast_coxph_regression_prebuilt_cpp . . . 91
fast_logistic_regression . . . 92
fast_logistic_regression_cpp . . . 94
fast_logistic_regression_weighted_cpp . . . 96
fast_logistic_regression_with_var . . . 97
fast_logistic_regression_with_var_cpp . . . 99
fast_negbin_regression . . . 100
fast_negbin_regression_with_var . . . 102
fast_ols_cpp . . . 103
fast_ols_with_var_cpp . . . 104
fast_poisson_regression_cpp . . . 105

fast_poisson_regression_weighted_cpp . . . 107
fast_poisson_regression_with_var_cpp . . . 109
fast_quasipoisson_regression_with_var_cpp . . . 111
fast_weibull_regression . . . 113
generate_covariate_dataset . . . 115
get_bootstrap_dispatch_policy . . . 116
get_cold_start_dispatch_policy . . . 118
get_local_EDI_optimization . . . 119
get_optimization_dispatch_policy . . . 119
get_parallel_dispatch_policy . . . 120
get_warm_start_dispatch_policy . . . 121
InferenceAllKKMeanDiffIVWC . . . 123
InferenceAllKKWilcoxIVWC . . . 126
InferenceAllSimpleAverageDiff . . . 131
InferenceAllSimpleMeanDiffPooledVar . . . 135
InferenceAllSimpleWilcox . . . 138
InferenceBaiAdjustedTKK14 . . . 143
InferenceBaiAdjustedTKK21 . . . 146
InferenceContinKKGLMM . . . 149
InferenceContinKKOLSIVWC . . . 152
InferenceContinKKOLSOneLik . . . 155
InferenceContinKKQuantileRegrIVWC . . . 158
InferenceContinKKQuantileRegrOneLik . . . 160
InferenceContinKKRobustRegrIVWC . . . 163
InferenceContinKKRobustRegrOneLik . . . 165
InferenceContinLin . . . 168
InferenceContinOLS . . . 171
InferenceContinQuantileRegr . . . 174
InferenceContinRobustRegr . . . 176
InferenceCountHurdleNegBin . . . 179
InferenceCountHurdlePoisson . . . 183
InferenceCountKKCondPoissonOneLik . . . 186
InferenceCountKKGLMM . . . 189
InferenceCountKKHurdlePoissonIVWC . . . 194
InferenceCountKKHurdlePoissonOneLik . . . 196
InferenceCountNegBin . . . 199
InferenceCountPoisson . . . 203
InferenceCountPoissonKKGEE . . . 210
InferenceCountQuasiPoisson . . . 212
InferenceCountRobustPoisson . . . 215
InferenceCountZeroInflatedNegBin . . . 217
InferenceCountZeroInflatedPoisson . . . 219
InferenceIncidBinomialIdentityRiskDiff . . . 222
InferenceIncidCMH . . . 224
InferenceIncidExactBinomial . . . 227
InferenceIncidExactFisher . . . 229
InferenceIncidExactZhang . . . 231
InferenceIncidExtendedRobins . . . 233

InferenceIncidGCompRiskDiff . . . 235
InferenceIncidGCompRiskRatio . . . 248
InferenceIncidKKCondLogitGLMMIVWC . . . 262
InferenceIncidKKCondLogitGLMMOneLik . . . 264
InferenceIncidKKCondLogitIVWC . . . 265
InferenceIncidKKCondLogitOneLik . . . 267
InferenceIncidKKGCompRiskDiff . . . 269
InferenceIncidKKGCompRiskRatio . . . 272
InferenceIncidKKGEE . . . 274
InferenceIncidKKModifiedPoisson . . . 275
InferenceIncidKKNewcombeRiskDiff . . . 277
InferenceIncidLogBinomial . . . 278
InferenceIncidLogRegr . . . 280
InferenceIncidMiettinenNurminenRiskDiff . . . 282
InferenceIncidModifiedPoisson . . . 286
InferenceIncidNewcombeRiskDiff . . . 288
InferenceIncidProbitRegr . . . 291
InferenceIncidRiskDiff . . . 293
InferenceIncidWald . . . 296
InferenceOrdinalAdjCatLogitRegr . . . 297
InferenceOrdinalCauchitRegr . . . 300
InferenceOrdinalCloglogRegr . . . 303
InferenceOrdinalContRatioRegr . . . 305
InferenceOrdinalGCompMeanDiff . . . 308
InferenceOrdinalJonckheereTerpstraTest . . . 311
InferenceOrdinalKKCLMM . . . 314
InferenceOrdinalKKCLMMCauchit . . . 315
InferenceOrdinalKKCLMMCloglog . . . 317
InferenceOrdinalKKCLMMProbit . . . 318
InferenceOrdinalKKCondAdjCatLogitRegr . . . 319
InferenceOrdinalKKGEE . . . 322
InferenceOrdinalKKGLMM . . . 324
InferenceOrdinalOrderedProbitRegr . . . 328
InferenceOrdinalPairedSignTest . . . 331
InferenceOrdinalPartialProportionalOddsRegr . . . 334
InferenceOrdinalPropOddsRegr . . . 338
InferenceOrdinalRidit . . . 341
InferenceOrdinalStereotypeLogitRegr . . . 344
InferencePropBetaRegr . . . 348
InferencePropFractionalLogit . . . 351
InferencePropGCompMeanDiff . . . 353
InferencePropKKGEE . . . 359
InferencePropKKGLMM . . . 361
InferencePropKKQuantileRegrIVWC . . . 362
InferencePropKKQuantileRegrOneLik . . . 364
InferencePropQuantileRegr . . . 366
InferencePropZeroOneInflatedBetaRegr . . . 369
InferenceRandCustom . . . 372

InferenceSuite . . . 373
InferenceSurvivalCoxPHRegr . . . 382
InferenceSurvivalDepCensTransformRegr . . . 385
InferenceSurvivalGehanWilcox . . . 386
InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC . . . 392
InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik . . . 395
InferenceSurvivalGLMMWeibullFrailtyNormalIVWC . . . 398
InferenceSurvivalGLMMWeibullFrailtyNormalOneLik . . . 400
InferenceSurvivalKKLWACoxPHIVWC . . . 403
InferenceSurvivalKKLWACoxPHOneLik . . . 406
InferenceSurvivalKKRankRegrIVWC . . . 409
InferenceSurvivalKKStratCoxPHIVWC . . . 411
InferenceSurvivalKKStratCoxPHOneLik . . . 414
InferenceSurvivalKKWeibullMarginal . . . 417
InferenceSurvivalKMDiff . . . 418
InferenceSurvivalLogRank . . . 423
InferenceSurvivalRestrictedMeanDiff . . . 426
InferenceSurvivalStratCoxPHRegr . . . 429
InferenceSurvivalWeibullRegr . . . 432
inv_logit . . . 434
logit . . . 435
mn_pvalue_cpp . . . 436
print.EDIInferenceSuiteResults . . . 437
robust_negbinreg . . . 438
robust_survreg . . . 438
robust_survreg_with_surv_object . . . 439
sample_mode . . . 441
set_cold_start_dispatch_policy . . . 441
set_num_cores . . . 442
set_optimization_dispatch_policy . . . 443
set_parallel_dispatch_policy . . . 444
set_warm_start_dispatch_policy . . . 445
SimulationFramework . . . 446
SimulationFrameworkReport . . . 454
summary.EDIInferenceSuiteResults . . . 456
summary_glm_lean . . . 457
transform_cont_y_based_on_response_type . . . 458
tune_EDI_for_this_machine . . . 459
unset_num_cores . . . 461

Index **463**

`build_cox_data_cache_cpp`

Build a Reusable Unstratified Cox Data Cache (C++ Backend)

Description

Precomputes and caches the sorted risk-set structure needed to evaluate the Cox partial-likelihood, score, and Hessian, so that repeated Newton-Raphson or L-BFGS fits on the same `(X, y, dead)` data (e.g. across bootstrap/randomization replicates of the treatment column, or successive `estimate_only` vs. full-variance calls) do not repeat the $O(n \log n)$ sort and event-time tabulation on every call. This is the unstratified counterpart of `build_stratified_cox_data_cache_cpp`; the returned cache is consumed by `fast_coxph_regression_prebuilt_cpp`, which implements the same Cox partial-likelihood as `fast_coxph_regression` and `fast_coxph_regression_cpp`.

Usage

```
build_cox_data_cache_cpp(X, y, dead)
```

Arguments

`X`	A numeric matrix of predictor variables (no intercept column); n rows, one per subject.
`y`	A numeric vector of length n giving the observed (event or censoring) time for each subject.
`dead`	A numeric vector of length n with values in `{0, 1}` indicating event status (1 = event/death, 0 = right-censored).

Details

Model. No fitting happens here; this function only prepares the single risk set (stratum) used by the Breslow-tied Cox partial likelihood

$$\ell(\beta) = \sum_k \Big[\big(\textstyle\sum_{i \in D_k} x_i \big)^\top \beta - d_k \log\big(\textstyle\sum_{j \in R_k} e^{x_j^\top \beta} \big) \Big],$$

where D_k is the set of subjects with an event at the k-th unique event time and R_k is the risk set (all subjects with `y >=` that event time). See `fast_coxph_regression` for the full model, scale, and optimizer contract; this page documents only the cache construction.

What is cached. Internally builds a single `CoxData` record that: (1) sorts subjects by ascending `y` (observed/censoring time), breaking ties by placing events (`dead == 1`) before censored observations at the same time; (2) stores the sort-permuted `y`, `dead`, and row-major copy of `X`; and (3) tabulates the vector of unique event times and, for each, the number of tied events (`event_counts`), which drives the Breslow tie-handling correction in the partial-likelihood, score, and Hessian.

Input conventions. `X` is an $n \times p$ design matrix with one row per subject and **no intercept column** (Cox models are fit on the partial likelihood, which has no intercept). `y` is the observed time (event or censoring time), and `dead` is a 0/1 event indicator (1 = event, 0 = right-censored); both

must have length n matching `nrow(X)`. Tied event times are handled via the Breslow approximation (via `event_counts`), not the exact (Cox) or Efron method. There is no support for left- or interval-censored data at this layer; classes with more general censoring (see `supports_interval_or_left_censored_data()` in `InferenceEngine`) bypass this cache and dispatch to **icenReg** instead. `NA`/non-finite values in `X`, `y`, or `dead` and negative values of `y` are not checked or handled at this layer; callers are responsible for filtering or imputing upstream.

Return value and object lifetime. Returns an `externalptr` (`Rcpp::XPtr`) wrapping a heap-allocated `std::vector<CoxData>` of length 1 (a single unstratified stratum), so that `fast_coxph_regression_prebuilt_cp` can share the same stratified interface as the stratified cache. The pointer owns its memory (finalizer registered via `XPtr(..., true)`) and is freed automatically by R's garbage collector; it must not be serialized (e.g. via `saveRDS`) or reused after the R session that created it exits. The cache is immutable once built and safe to reuse across many calls to `fast_coxph_regression_prebuilt_cpp` as long as `X`, `y`, and `dead` have not changed; callers (e.g. `InferenceCoxPH$private$cox_data_cache`) are responsible for invalidating and rebuilding the cache when any of the full `w`, `y`, `dead`, or `X` inputs changes.

Complexity. $O(n \log n)$ for the sort plus $O(n)$ for the tabulation pass, where n is the number of subjects; memory use is $O(np)$ for the row-major copy of `X` plus $O(n)$ for the sorted `y`/`dead` and event-time tables.

Value

An `externalptr` to a cached, sorted Cox risk-set representation of `(X, y, dead)`, for use as the `cox_data_xptr` argument of `fast_coxph_regression_prebuilt_cpp`.

See Also

`build_stratified_cox_data_cache_cpp` for the stratified (multiple risk-set) analog, `fast_coxph_regression_prebuilt` for the fitting routine that consumes this cache, and `fast_coxph_regression` for the full Cox model documentation, including the partial-likelihood derivation, tie-handling, and references. Analogous Python API: lifelines CoxPHFitter.

`build_stratified_cox_data_cache_cpp`

Build a Reusable Stratified Cox Data Cache (C++ Backend)

Description

Precomputes and caches, per stratum, the sorted risk-set structure needed to evaluate the stratified Cox partial-likelihood, score, and Hessian, so that repeated Newton-Raphson or L-BFGS fits on the same `(X, y, dead, strata)` data (e.g. across bootstrap/randomization replicates of the treatment column) do not repeat the per-stratum sort and event-time tabulation on every call. This is the stratified counterpart of `build_cox_data_cache_cpp`; the returned cache is consumed by `fast_coxph_regression_prebuilt_cpp`, which implements the same partial-likelihood family as `fast_coxph_regression` and `fast_coxph_regression_cpp`, extended to sum the log-partial-likelihood, score, and Hessian across independent strata-specific risk sets while sharing a single regression coefficient vector β across all strata (a shared-β, per-stratum-baseline-hazard stratified Cox model).

Usage

```
build_stratified_cox_data_cache_cpp(X, y, dead, strata)
```

Arguments

X	A numeric matrix of predictor variables (no intercept column); n rows, one per subject.
y	A numeric vector of length n giving the observed (event or censoring) time for each subject.
dead	A numeric vector of length n with values in {0, 1} indicating event status (1 = event/death, 0 = right-censored).
strata	An integer vector of length n giving the stratum (block/cluster) label of each subject. Risk sets are formed separately within each distinct label.

Details

Model. No fitting happens here; this function only partitions subjects into per-stratum risk sets and prepares each one for the Breslow-tied stratified Cox partial likelihood

$$\ell(\beta) = \sum_{s} \sum_{k \in s} \Big[\big(\textstyle\sum_{i \in D_{sk}} x_i \big)^\top \beta - d_{sk} \log\big(\textstyle\sum_{j \in R_{sk}} e^{x_j^\top \beta} \big) \Big],$$

where the outer sum runs over strata s (distinct baseline hazards), D_{sk} is the set of subjects with an event at the k-th unique event time within stratum s, and R_{sk} is the corresponding within-stratum risk set (subjects in stratum s with y >= that event time). Risk sets never cross strata, so subjects are only ever compared to other subjects in the same stratum; β is shared across strata while the baseline hazard is allowed to differ arbitrarily by stratum. See fast_coxph_regression for the unstratified model, scale, and optimizer contract; this page documents only the per-stratum cache construction.

What is cached. Subjects are grouped into strata by their integer strata label (via an ordered std::map<int, ...>, so strata are processed and stored in ascending label order). Within each stratum, a CoxData record is built exactly as in build_cox_data_cache_cpp: subjects are sorted by ascending y, ties broken by placing events before censored observations, and the per-stratum unique event times and tied-event counts (event_counts) are tabulated to drive the within-stratum Breslow tie-handling correction.

Input conventions. X is an $n \times p$ design matrix with one row per subject and **no intercept column**. y is the observed time (event or censoring time) and dead is a 0/1 event indicator (1 = event, 0 = right-censored); both have length n matching nrow(X). strata is a length-n integer vector of stratum (block/cluster) labels; labels need not be contiguous or start at 1, and a stratum with a single subject or with no observed events contributes an empty or degenerate risk set that carries no information to the partial likelihood, score, or Hessian (its rows are still stored, but they will not affect the fit). Callers are expected to have already dropped uninformative strata upstream (see get_informative_rows() in the stratified-Cox inference class) when that matters for numerical stability; this function does not filter strata itself. As with build_cox_data_cache_cpp, tied event times use the Breslow approximation, there is no support for left- or interval-censored data at this layer, and NA/non-finite values in X, y, dead, or strata are not checked or handled here.

Return value and object lifetime. Returns an externalptr (Rcpp::XPtr) wrapping a heap-allocated std::vector<CoxData> with one element per distinct stratum label (ordered ascending

by label), consumed by `fast_coxph_regression_prebuilt_cpp` exactly like the length-1 vector returned by `build_cox_data_cache_cpp`. Lifetime, garbage-collection, and non-serializability semantics are identical to `build_cox_data_cache_cpp`: the cache is immutable once built and safe to reuse across repeated fits as long as `X`, `y`, `dead`, and `strata` have not changed; callers (e.g. `InferenceStratifiedCoxPH$private$strat_cox_data_cache`) are responsible for invalidating and rebuilding it when any of the full `w`, `y`, `dead`, or `X` inputs changes (including covariates that determine strata).

Complexity. $O(n \log(n/S))$ for the per-stratum sorts plus $O(n)$ for the tabulation passes, where n is the number of subjects and S is the number of distinct strata; memory use is $O(np)$ for the per-stratum row-major copies of `X` plus $O(n)$ for the sorted `y`/`dead` and event-time tables.

Value

An `externalptr` to a cached, per-stratum sorted Cox risk-set representation of `(X, y, dead, strata)`, for use as the `cox_data_xptr` argument of `fast_coxph_regression_prebuilt_cpp`.

See Also

`build_cox_data_cache_cpp` for the unstratified (single risk-set) analog, `fast_coxph_regression_prebuilt_cpp` for the fitting routine that consumes this cache, and `fast_coxph_regression` for the full Cox model documentation, including the partial-likelihood derivation, tie-handling, and references. Analogous Python API: lifelines CoxPHFitter (stratified fits via the `strata` argument) and statsmodels duration models.

`clear_local_EDI_optimization`

Delete this machine's saved EDI tuning and return to shipped defaults

---

Description

Removes the per-user config file written by `tune_EDI_for_this_machine` (so the next `library(EDI)` starts from the package's built-in performance-policy defaults) and resets the in-session cold-start, warm-start, and optimizer-algorithm dispatch policies to those defaults right away.

Usage

```
clear_local_EDI_optimization()
```

Value

Invisibly, `TRUE` if a saved tuning existed and was removed, `FALSE` if there was none.

See Also

`tune_EDI_for_this_machine`, `get_local_EDI_optimization`.

Examples

```
clear_local_EDI_optimization()
```

DesignFixedBernoulli *A Fixed-Sample-Size Bernoulli (Independent-Coin-Flip) Randomized Design*

Description

A fixed-sample-size DesignFixed in which each subject's treatment assignment w_i is drawn independently as $w_i \overset{iid}{\sim} \text{Bernoulli}(p)$, $i = 1, \ldots, n$, where p is prob_T. This is the classical Bernoulli (independent-coin-flip) randomized design: unlike DesignFixediBCRD (complete randomization), which fixes the number of treated subjects at exactly $\text{round}(np)$, a Bernoulli design leaves the realized number of treated subjects $n_T = \sum_i w_i \sim \text{Binomial}(n, p)$ random; the trade-off is independence across subjects (useful for some asymptotic/martingale arguments) at the cost of not guaranteeing exact balance, which can matter for small n or for inference procedures (e.g. exact permutation tests over a fixed number of treated) that assume a fixed n_T.

Draw mechanism. draw_ws_raw(r) delegates to generate_permutations_bernoulli_cpp(), which fills an $n \times r$ matrix of independent $\text{Bernoulli}(p)$ draws (one column per requested replicate, via a Mersenne Twister RNG seeded once per call from R's RNG state), so r replicate allocation vectors are generated with a single C++ call rather than r separate calls into R's own random-number generation. assign_w_to_all_subjects() draws a single such allocation (r = 1) and applies it to all subjects at once.

No exchange/balance search. Because subjects are treated independently, there is no optimization step analogous to DesignFixedGreedyDOptimal: covariates, if supplied, do not influence the assignment probabilities or realized allocation at all.

Super classes

Design -> DesignFixed -> DesignFixedBernoulli

Methods

Public methods:

- DesignFixedBernoulli$is_a_bernoulli_capable()
- DesignFixedBernoulli$new()
- DesignFixedBernoulli$clone()

DesignFixedBernoulli$is_a_bernoulli_capable(): Characterization: this design draws each subject's treatment assignment as an independent $\text{Bernoulli}(p)$ coin flip (see class documentation), so it is Bernoulli-capable by construction.

Usage:

```
DesignFixedBernoulli$is_a_bernoulli_capable()
```

Returns: Always TRUE for this class.

`DesignFixedBernoulli$new()`: Initialize a fixed Bernoulli (independent-coin-flip) experimental design. Unlike `DesignFixediBCRD`, the realized number of treated subjects is not fixed at `n * prob_T`; it is random ($\text{Binomial}(n, prob_T)$) because each subject's assignment is an independent coin flip (see class documentation).

Usage:

```
DesignFixedBernoulli$new(
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` "continuous", "incidence", "proportion", "count", "survival", or "ordinal".

`prob_T` Per-subject probability p that a given subject is assigned to treatment; need not be `0.5` (unlike `DesignFixedGreedyDOptimal`).

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` A flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new 'DesignFixedBernoulli' object

`DesignFixedBernoulli$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignFixedBernoulli$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Neyman, J. (1923, transl. 1990). "On the Application of Probability Theory to Agricultural Experiments." *Statistical Science*, 5(4), 465-472, for the potential-outcomes framework under which Bernoulli and complete randomization are compared; see also randomized experiment for orientation on Bernoulli vs. complete (restricted) randomization.

Examples

```
des = DesignFixedBernoulli$new(n = 10, response_type = 'continuous')
des$add_all_subjects_to_experiment(data.frame(x1 = rnorm(10)))
des$assign_w_to_all_subjects()
```

DesignFixedBinaryMatch

A Fixed, Non-Bipartite-Matched-Pair Design with Within-Pair Randomization

Description

A fixed-sample-size DesignFixed that (1) partitions the n subjects into $n/2$ disjoint matched pairs by solving a non-bipartite (optimal) pairwise-matching problem on a covariate distance matrix, minimizing the total within-pair distance across all pairs, then (2) randomizes treatment **within** each pair independently: for pair k, one of its two members is assigned to treatment and the other to control with probability $1/2$ each, independently across pairs. This is the classical matched-pair randomized design (a special case of blocking with block size 2), which guarantees exact covariate balance within every pair (up to the matching algorithm's distance metric) while preserving randomization-based inference validity, in contrast to DesignFixedGreedyDOptimal, which balance the covariates in aggregate via an optimality criterion but do not guarantee pairwise closeness of any two subjects.

Matching algorithm. Pairing is computed once (lazily, on first call to draw_ws_raw()/assign_w_to_all_subjects(), via private$ensure_matching_structure_computed()) by compute_binary_match_structure(), which forms an $n \times n$ pairwise distance matrix — Mahalanobis distance if mahal_match = TRUE (the default; using the sample covariance of the covariates, ridge-regularized if singular), or Euclidean distance otherwise — and solves the minimum-weight non-bipartite perfect matching on that distance matrix via nonbimatch (**nbpMatching**, a Suggests-only dependency; loading is deferred until matching is actually needed, so pre-computed w vectors injected via m never require it). For a single covariate ($p = 1$), pairing instead reduces to simply sorting subjects by that covariate and pairing consecutive subjects, since the non-bipartite matching problem is trivial in one dimension. The resulting pairing is cached in private$bms/private$m for the lifetime of the design object (or until explicitly reset via set_m()) and is not recomputed per draw.

Within-pair randomization. Given the fixed pairing, each replicate allocation (see draw_binary_match_assignments_cpp independently flips, for every pair, which of its two members is treated (an independent fair coin flip per pair per replicate, using a splitmix64-seeded Mersenne Twister per replicate column for reproducible parallel draws); this guarantees exactly $n/2$ treated subjects overall (only prob_T = 0.5 is supported; the constructor errors otherwise). Passing a pre-computed m to the constructor supplies the matched-pair structure directly (each pair ID occurring in exactly 2 rows), bypassing the matching computation entirely while keeping the same within-pair randomization.

No-covariate fallback. If no covariates are available at draw time (private$m is NULL, e.g. matching hasn't run and no explicit m was supplied), draw_ws_raw() falls back to an unmatched balanced complete randomization (a uniformly random permutation of $n/2$ ones and $n/2$ zeros), since there is no covariate information to match on.

Batch pregeneration. draw_binary_match_assignments_cpp()'s output is trusted unvalidated – it guarantees exactly n x r valid $\{0, 1\}$ columns with $n/2$ treated subjects per column by construction (see fix_design_hierarchy.md, "AllocationMatrixValidation"). supports_batch_w_pregeneration() returns TRUE so that the calling framework generates all replicate w vectors for a simulation cell in one batch (amortizing the one-time **nbpMatching** matching cost across all replicates of that cell) rather than recomputing the matching structure per replicate.

Super classes

Design -> DesignFixed -> DesignFixedBinaryMatch

Methods

Public methods:

- DesignFixedBinaryMatch$is_a_kk_matching_capable()
- DesignFixedBinaryMatch$supports_batch_w_pregeneration()
- DesignFixedBinaryMatch$new()
- DesignFixedBinaryMatch$assign_w_to_all_subjects()
- DesignFixedBinaryMatch$clone()

DesignFixedBinaryMatch$is_a_kk_matching_capable(): Characterization: this design computes its matched-pair structure on the fly from covariates (see class documentation), so it is KK matching-capable by construction.

Usage:

```
DesignFixedBinaryMatch$is_a_kk_matching_capable()
```

Returns: Always TRUE for this class.

DesignFixedBinaryMatch$supports_batch_w_pregeneration(): Returns TRUE so the calling framework pre-generates all replicate w vectors for a simulation cell in one batch, paying the one-time **nbpMatching** non-bipartite matching cost once per cell and reusing the resulting pairing across all replicates, rather than recomputing it per replicate.

Usage:

```
DesignFixedBinaryMatch$supports_batch_w_pregeneration()
```

Returns: Always TRUE for this class.

DesignFixedBinaryMatch$new(): Initialize a binary (non-bipartite) matched-pair fixed experimental design. Matching itself is deferred until the first draw (see class documentation); this constructor only records configuration and, if m is supplied, installs the explicit pairing immediately.

Usage:

```
DesignFixedBinaryMatch$new(
  response_type,
  prob_T = 0.5,
  mahal_match = TRUE,
  match_objective = c("distance", "squared_distance"),
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  m = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` The data type of response values.

`prob_T` The probability of the treatment assignment. Must be 0.5, since within-pair randomization only supports an even 1-treated/1-control split per pair.

`mahal_match` Match using Mahalanobis distance (accounting for covariate correlation/scale) if `TRUE` (default), else Euclidean distance on the raw covariate matrix.

`match_objective` Matching objective. `"distance"` (default) matches **nbpMatching**; `"squared_distance"` reproduces EDI through version 1.0.4.

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`m` Optional integer vector of explicit matched-pair identifiers, one per subject. If supplied, ‘n‘ must also be supplied, ‘length(m)‘ must equal ‘n‘, all values must be positive, and each pair ID must occur exactly twice. This bypasses the package-computed matching step.

`verbose` Flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new ‘DesignFixedBinaryMatch‘ object

`DesignFixedBinaryMatch$assign_w_to_all_subjects():` Assign treatment to all subjects (see `DesignFixed$assign_w_to_all_subjects()` for the general contract). Before delegating, this override ensures the matched-pair structure is computed (`private$ensure_matching_structure_computed()`) even when `w_precomputed` is supplied and `draw_ws_according_to_design()` is therefore never called — downstream code (e.g. blocked/matched-pair inference) still needs `private$m` to be populated regardless of how `w` was obtained.

Usage:

```
DesignFixedBinaryMatch$assign_w_to_all_subjects(w_precomputed = NULL)
```

Arguments:

`w_precomputed` Optional {0,1} numeric vector of length `n`. If supplied, it is used directly as the treatment allocation instead of drawing a fresh within-pair-randomized allocation.

`DesignFixedBinaryMatch$clone():` The objects of this class are cloneable with this method.

Usage:

```
DesignFixedBinaryMatch$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Greevy, R., Lu, B., Silber, J. H., and Rosenbaum, P. (2004). "Optimal multivariate matching before randomization." *Biostatistics*, 5(2), 263-275, doi:10.1093/biostatistics/5.2.263, for optimal non-bipartite matched-pair designs prior to randomization. See also matched pair and Mahalanobis distance for orientation.

Examples

```
des = DesignFixedBinaryMatch$new(n = 10, response_type = 'continuous')
des$add_all_subjects_to_experiment(data.frame(x1 = rnorm(10)))
des$assign_w_to_all_subjects()
```

DesignFixedBlockedCluster

A Fixed, Blocked-and-Clustered Randomized Design

Description

A fixed-sample-size DesignFixed in which the unit of randomization is the **cluster**, not the individual subject: within each block (stratum, formed from strata_cols), whole clusters (identified by cluster_col) are jointly randomized to treatment or control, so all subjects in the same cluster always receive the same assignment. This is the design used when individual-level randomization is infeasible or invalid (e.g. clusters are classrooms, clinics, or households where within-cluster interference/spillover would violate SUTVA under individual randomization), combined with blocking to improve precision by comparing clusters only to other clusters in the same stratum.

Randomization mechanism. Blocking keys are computed per subject via private$get_strata_keys() (shared with other blocking-structure designs): categorical columns in strata_cols are used as-is, continuous columns are discretized into preferred_num_bins_for_continuous_covariate quantile-based bins, and multiple strata_cols are combined into a single composite block key. Within each resulting block, whole clusters (by cluster_col) are randomized to treatment with probability prob_T via block_and_cluster_ra (**randomizr**), which performs blocked-and-clustered complete random assignment: within each block, clusters (not subjects) are permuted so that, subject to rounding, the target proportion prob_T of clusters in that block is treated, and every subject in a treated cluster receives $w = 1$. r independent replicate allocation columns are generated via replicate() (one **randomizr** call per replicate; there is no batch/vectorized draw path for this design, unlike DesignFixedBinaryMatch).

Cluster-aware bootstrap. draw_bootstrap_indices() overrides the default subject-level bootstrap to resample at the cluster level via resample_group_rows_cpp(): with bootstrap_type = "within_blocks" (the default when bootstrap_type is NULL), clusters are resampled with replacement *within* each block, preserving the block structure; otherwise, whole blocks (strata) are themselves resampled with replacement. This mirrors the standard cluster-robust bootstrap principle that resampling must occur at the level of the randomization unit (clusters), not individual subjects, to yield a valid variance/interval estimate under cluster-correlated outcomes.

Super classes

Design -> DesignFixed -> DesignFixedBlockedCluster

Methods

Public methods:

- DesignFixedBlockedCluster$is_a_cluster_capable()
- DesignFixedBlockedCluster$new()

- DesignFixedBlockedCluster$clone()

`DesignFixedBlockedCluster$is_a_cluster_capable()`: Characterization: this design randomizes whole clusters (see class documentation), so it is cluster-structured by construction.

Usage:

```
DesignFixedBlockedCluster$is_a_cluster_capable()
```

Returns: Always `TRUE` for this class.

`DesignFixedBlockedCluster$new()`: Initialize a blocked and cluster randomized fixed experimental design.

Usage:

```
DesignFixedBlockedCluster$new(
  strata_cols,
  cluster_col,
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  preferred_num_bins_for_continuous_covariate = 2,
  num_bins_for_continuous_covariate = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`strata_cols` A character vector of column names to use for stratification (blocks).

`cluster_col` The column name in the data that identifies the cluster for each subject.

`response_type` The data type of response values.

`prob_T` The target probability that a given cluster within a block is assigned to treatment (subjects inherit their cluster's assignment).

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`preferred_num_bins_for_continuous_covariate` The number of quantile bins to use for continuous strata. Default is 2.

`num_bins_for_continuous_covariate` Deprecated alias for 'preferred_num_bins_for_continuous_covariate'.

`verbose` Flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new 'DesignFixedBlockedCluster' object

`DesignFixedBlockedCluster$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignFixedBlockedCluster$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Middleton, J. A., and Aronow, P. M. (2015). "Unbiased estimation of the average treatment effect in cluster-randomized experiments." *Statistics, Politics and Policy*, 6(1-2), 39-75, doi:10.1515/spp-20130002, for blocked/clustered randomized-assignment inference; see also the **randomizr** package vignette for the assignment-generation conventions this design relies on, and cluster randomized controlled trial for orientation.

Examples

```
des = DesignFixedBlockedCluster$new(n = 20, response_type = 'continuous',
  strata_cols = 'x2', cluster_col = 'cl')
X = data.frame(x1 = rnorm(20), x2 = factor(rep(1:2, each = 10)), cl = factor(rep(1:10, each = 2)))
des$add_all_subjects_to_experiment(X)
des$assign_w_to_all_subjects()
```

DesignFixedBlocking *A Fixed, Stratified-Block Randomized Design*

Description

A fixed-sample-size `DesignFixed` that first partitions subjects into blocks (strata) formed from covariates, then randomizes treatment **independently within each block** at probability `prob_T` (via `block_ra` when **randomizr** is installed, else an internal `generate_permutations_blocking_cpp()` fallback). Blocking on a covariate removes its between-block variation from the treatment-effect comparison (comparisons are always within-block), improving precision relative to unblocked randomization whenever the blocking covariate(s) are prognostic of the outcome, at the cost of requiring the analysis to account for the blocking structure (e.g. via a block/stratum fixed effect or a CMH-type test). This differs from `DesignFixedBlockedCluster`, which randomizes whole *clusters* of subjects together within each block rather than subjects individually.

Block construction. Blocking keys are computed by `private$get_strata_keys()` (shared across blocking-structure designs): each column in `strata_cols` contributes a categorical key (continuous columns are discretized into `preferred_num_bins_for_continuous_covariate` quantile bins), and multiple columns are combined into one composite block key per subject; if `strata_cols` is `NULL`, all available covariate columns are used. `B_target` caps the number of resulting blocks by greedily adding `strata_cols` in order only while the running block count stays at or below the target (earlier columns take priority); `exact_num_blocks = TRUE` instead hard-fails if the greedy construction does not land on exactly `B_target` blocks. `equal_block_sizes = TRUE` (the default) additionally requires every block to have the same subject count, checked once at construction (via `n %% B_target`) if `n` and `B_target` are both already known, and again once covariates arrive; some downstream inference classes (`InferenceIncidCMH`, `InferenceIncidExtendedRobins`) require equal block sizes unconditionally, regardless of this flag. An explicit `m` (one block ID per subject) bypasses covariate-derived block construction entirely.

Within-block randomization and bootstrap. Within each block, treatment is assigned independently via `block_ra`'s complete random assignment (subject to rounding, `prob_T` of each block's subjects are treated); the internal C++ fallback (`generate_permutations_blocking_cpp()`) is used only if **randomizr** is not installed. `draw_bootstrap_indices()` resamples *within* each block

by default (bootstrap_type = "within_blocks" or NULL, via stratified_bootstrap_indices_cpp()), or resamples whole blocks with replacement otherwise (via resample_group_rows_cpp()) — mirroring the block structure in the resampling scheme, analogous to the cluster-level bootstrap in DesignFixedBlockedCluster.

Super classes

Design -> DesignFixed -> DesignFixedBlocking

Methods

Public methods:

- DesignFixedBlocking$new()
- DesignFixedBlocking$clone()

DesignFixedBlocking$new(): Initialize a fixed stratified-block randomized experimental design. Block construction and validation follow the rules described in the class documentation; see the parameter descriptions below for the greedy B_target/exact_num_blocks/equal_block_sizes contract.

Usage:

```
DesignFixedBlocking$new(
  strata_cols = NULL,
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  preferred_num_bins_for_continuous_covariate = 2,
  B_target = NULL,
  exact_num_blocks = FALSE,
  equal_block_sizes = TRUE,
  m = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

strata_cols A character vector of column names to use for stratification. If ‘NULL‘ (the default), all available covariate columns are used.

response_type "continuous", "incidence", "proportion", "count", "survival", or "ordinal".

prob_T Probability of treatment assignment.

include_is_missing_as_a_new_feature Flag for missingness indicators.

n The sample size.

preferred_num_bins_for_continuous_covariate The number of quantile bins to use for continuous strata. Default is 2.

`B_target` The target number of blocks. Columns from ‘strata_cols‘ are added greedily in order, each column being included only if it does not push the total number of unique blocks beyond this target. For categorical covariates their natural levels are used; for continuous covariates ‘preferred_num_bins_for_continuous_covariate‘ quantile bins are used. Earlier columns are always preferred over later ones. When ‘n‘ is known at construction time, the default is the largest divisor of ‘n‘ that is at most ‘floor(sqrt(n))‘ (so the default always satisfies ‘equal_block_sizes = TRUE‘); if ‘n‘ is not yet known, it is resolved to ‘floor(sqrt(n))‘ when subjects are added. Set ‘B_target = NULL‘ to use all columns unconditionally. An explicitly supplied ‘B_target‘ that does not divide ‘n‘ still errors immediately when ‘equal_block_sizes = TRUE‘. Set ‘exact_num_blocks = TRUE‘ to hard fail if the final key construction does not produce exactly ‘B_target‘ blocks.

`exact_num_blocks` Whether to require the greedy key construction to produce exactly ‘B_target‘ blocks. Default ‘FALSE‘.

`equal_block_sizes` Whether to require all blocks to have the same number of subjects. Default ‘TRUE‘. When ‘TRUE‘ and both ‘n‘ and ‘B_target‘ are known at construction time, an error is raised immediately if ‘n‘ is not divisible by ‘B_target‘. A second check fires when subjects are added: if the covariate-based strata produce unequal block counts the design errors at that point. Set to ‘FALSE‘ to allow unequal blocks (note that ‘InferenceIncidCMH‘ and ‘InferenceIncidExtendedRobins‘ still require equal block sizes regardless).

`m` Optional integer vector of explicit block identifiers, one per subject. If supplied, ‘n‘ must also be supplied and ‘length(m)‘ must equal ‘n‘. The constructor then records this blocking structure immediately via ‘set_m()‘, bypassing covariate-derived strata construction.

`verbose` A flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new ‘DesignFixedBlocking‘ object

`DesignFixedBlocking$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignFixedBlocking$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Fisher, R. A. (1935). *The Design of Experiments*. Oliver and Boyd, for the original rationale for blocking in randomized experiments; Cochran, W. G., and Cox, G. M. (1957). *Experimental Designs* (2nd ed.), Wiley, for stratified (randomized block) design theory. See also randomized block design for orientation.

Examples

```
des = DesignFixedBlocking$new(n = 20, response_type = 'continuous',
  strata_cols = 'x2', equal_block_sizes = FALSE)
X = data.frame(x1 = rnorm(20), x2 = factor(rep(1:2, 10)))
des$add_all_subjects_to_experiment(X)
des$assign_w_to_all_subjects()
```

DesignFixedCluster *A Fixed, Unblocked Cluster Randomized Design*

Description

A fixed-sample-size DesignFixed in which whole clusters of subjects (identified by cluster_col), rather than individual subjects, are the unit of randomization: every subject in a given cluster always receives the same treatment assignment. This is the unblocked analog of DesignFixedBlockedCluster — there is no stratification step here, so clusters are randomized to treatment as a single pool rather than within strata. Cluster-level randomization is required whenever individual-level randomization would create within-cluster interference/spillover that violates SUTVA (e.g. clusters are classrooms, clinics, villages, or households), at the cost of an effective sample size driven by the number of *clusters*, not subjects, and a corresponding need for cluster-aware inference.

Randomization mechanism. draw_ws_raw(r) extracts each subject's cluster ID from cluster_col (erroring if any are missing) and calls cluster_ra (**randomizr**) once per replicate, which performs complete random assignment at the cluster level: subject to rounding, prob_T of clusters are assigned to treatment, and all subjects sharing a cluster inherit that cluster's assignment. r independent replicate columns are generated via replicate() (one **randomizr** call per replicate).

Cluster-aware bootstrap. draw_bootstrap_indices() overrides the default subject-level bootstrap to resample whole clusters with replacement (via resample_group_rows_cpp()) rather than individual rows, since outcomes are correlated within a cluster (shared assignment plus, typically, shared context) and the exchangeable resampling unit for a valid bootstrap variance/interval estimate is therefore the cluster, not the subject.

Super classes

Design -> DesignFixed -> DesignFixedCluster

Methods

Public methods:

- DesignFixedCluster$is_a_cluster_capable()
- DesignFixedCluster$new()
- DesignFixedCluster$clone()

DesignFixedCluster$is_a_cluster_capable(): Characterization: this design randomizes whole clusters (see class documentation), so it is cluster-structured by construction.

Usage:

```
DesignFixedCluster$is_a_cluster_capable()
```

Returns: Always TRUE for this class.

DesignFixedCluster$new(): Initialize a cluster randomized fixed experimental design (no blocking/stratification; see DesignFixedBlockedCluster if stratification is also needed).

Usage:

```
DesignFixedCluster$new(
  cluster_col,
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`cluster_col` The column name in the data that identifies the cluster for each subject.

`response_type` The data type of response values.

`prob_T` The target probability that a given cluster is assigned to treatment (subjects inherit their cluster's assignment).

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` Flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new 'DesignFixedCluster' object

`DesignFixedCluster$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignFixedCluster$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Middleton, J. A., and Aronow, P. M. (2015). "Unbiased estimation of the average treatment effect in cluster-randomized experiments." *Statistics, Politics and Policy*, 6(1-2), 39-75, doi:10.1515/spp-20130002. See also cluster randomized controlled trial for orientation, and `DesignFixedBlockedCluster` for the blocked variant of this design.

Examples

```
des = DesignFixedCluster$new(n = 20, response_type = 'continuous', cluster_col = 'cl')
X = data.frame(x = rnorm(20), cl = factor(rep(1:5, each = 4)))
des$add_all_subjects_to_experiment(X)
des$assign_w_to_all_subjects()
```

DesignFixedFactorial *A Fixed, Balanced Two-Arm Factorial Design*

Description

A fixed-sample-size DesignFixed for factorial treatment structures: subjects are assigned to one of the cells of a factorial combination of one or more named factors (e.g. `list(treatment = 2)`, or, once multi-arm support lands, `list(drug = 2, dose = 2)` for a 2×2 design), with assignment counts balanced as evenly as possible across cells within each replicate draw.

Currently restricted to exactly two total factor-level combinations (i.e. two arms), e.g. a single two-level factor — the product of levels across all `factors` entries must equal exactly 2; the constructor errors otherwise. In this two-arm regime, the design reduces to a balanced complete randomization between cell 1 ($w = 0$) and cell 2 ($w = 1$), and `w` follows the same {0,1} internal / {-1,+1} public convention as every other `Design` subclass, so `DesignFixedFactorial` inherits `assign_w_to_all_subjects()`, `draw_ws_according_to_design()`, and `get_w()` unmodified from DesignFixed/Design and works unmodified with every Inference class; only `draw_ws_raw()` (the low-level allocation-vector generator) and `get_w_factorial()` (an additional factor-level accessor, see below) are specific to this class. Support for more than two combinations (true multi-factor, multi-arm designs) is tracked separately — see `package_metadata/new_feature_plans/multi_arm_des`

Allocation generation. `draw_ws_raw(r)` builds a base allocation vector by repeating the sequence of cell indices `0:(num_combinations - 1)` out to length n (so cells are as close to equally represented as possible, off by at most one subject when n is not a multiple of the number of cells), then independently permutes (sample) that base vector once per replicate column to produce `r` balanced-but-randomized allocations.

Super classes

Design -> DesignFixed -> `DesignFixedFactorial`

Methods

Public methods:

- `DesignFixedFactorial$new()`
- `DesignFixedFactorial$get_w_factorial()`
- `DesignFixedFactorial$clone()`

`DesignFixedFactorial$new()`**:** Initialize a factorial fixed experimental design. The product of levels implied by `factors` must currently equal exactly 2 (see class documentation); any other total raises an error.

Usage:

```
DesignFixedFactorial$new(
  factors,
  response_type,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
```

```
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`factors` A list where names are factor names and values are number of levels (e.g. list(treatment = 2)). The product of levels across all factors must currently equal exactly 2 (two-arm only).

`response_type` The data type of response values.

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` Flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new 'DesignFixedFactorial' object

`DesignFixedFactorial$get_w_factorial()`: Decode each subject's scalar cell index (`private$w`, in `0:(num_combinations - 1)`) back into its per-factor level assignments, using the same `expand.grid` enumeration (`private$combinations`) established at construction. This is the inverse of the encoding `draw_ws_raw()` produces, and is the only way to recover individual factor levels once support for more than two total combinations lands, since `get_w()` (inherited, see class documentation) only ever returns the scalar 0/1 (or -1/+1) cell index.

Usage:

```
DesignFixedFactorial$get_w_factorial()
```

Returns: A data frame with `n` rows and one column per entry of `factors`, giving each subject's level (an integer in `1:levels`) for that factor; `NULL` if treatment has not yet been assigned to all subjects (i.e. `private$w` is empty or contains `NA`).

`DesignFixedFactorial$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignFixedFactorial$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
des = DesignFixedFactorial$new(n = 12, response_type = 'continuous', factors = list(treatment = 2))
des$add_all_subjects_to_experiment(data.frame(x=1:12))
des$assign_w_to_all_subjects()
```

DesignFixedGreedy *A Fixed, Covariate-Balanced Design via Greedy Pairwise-Swap Search*

Description

A fixed-sample-size DesignFixed that searches, among balanced ($n/2$-treated) allocations, for one that directly minimizes a covariate imbalance criterion $f(d)$, $d = M(2w - 1)$, via a native C++ (RcppEigen + OpenMP) greedy pairwise-swap search (greedy_design_search_cpp()). Unlike DesignFixedGreedyDOptimal, which optimizes a model-based information-matrix criterion (D-/A-optimality) implied by an assumed linear model, this design optimizes a direct covariate-distance criterion between the treated and control group means/covariance, with no linear-model assumption: the two supported objectives are

- "mahal_dist" (default): $d = L^{-1}X^\top(2w - 1)/n$ where $\Sigma = LL^\top$ is the Cholesky factor of the covariate covariance matrix, and $f(d) = \|d\|_2^2$ — the squared Mahalanobis distance between the treated and control covariate means, accounting for covariate correlation/scale. Falls back to "abs_sum_diff" if the covariate covariance matrix is (numerically) singular, since the Cholesky factorization then fails.
- "abs_sum_diff": $d = X_{\text{std}}^\top(2w - 1)/n$ (covariates standardized to unit variance, no correlation adjustment) and $f(d) = \|d\|_1$ — the sum of absolute standardized mean differences across covariates.

Search algorithm. Starting from a random balanced (Fisher-Yates) allocation, the search runs in one of two modes selected by n_iter: Inf (default) runs **exhaustive best-improvement** search — each round scans every (treated, control) pair, applies the single globally best improving swap, and repeats until no swap improves $f(d)$, guaranteeing convergence to a strict local optimum; a positive integer instead runs exactly that many **stochastic** steps, each picking a uniformly random (treated, control) pair and accepting the swap only if it improves $f(d)$ (with patience-based early stopping). r independent design searches (one per requested replicate) run in parallel via OpenMP, each with its own std::mt19937 generator. **This search's randomization is reproducible via the constructor's** seed **argument**: per-thread RNGs are seeded from R's own RNG state (GetRNGstate()/unif_rand()) before the parallel region begins, so private$maybe_set_seed() does govern the resulting allocation, independent of the number of OpenMP threads used.

Same draws as DesignFixedGreedyDOptimal **at the defaults.** With objective = "mahal_dist" and n_iter = Inf (the defaults), this class returns exactly the same allocations as DesignFixedGreedyDOptimal with its defaults (objective = "D", no prior_precision) for the same seed. Despite the "no linear-model assumption" framing above, the default criterion coincides with D-optimality: with the number of treated subjects fixed, the information about the treatment effect in the linear model $y = \beta_T w + \beta_0 + X\gamma + \epsilon$ equals a constant minus a positive multiple of the squared Mahalanobis distance between the treated and control covariate means. Both criteria therefore rank every allocation the same way, and the same swap search from the same starting allocation reaches the same allocation (the derivation is in DesignFixedGreedyDOptimal's documentation). The draws differ when this class uses objective = "abs_sum_diff", a finite n_iter, or the "abs_sum_diff" fallback after a failed Cholesky factorization, and when DesignFixedGreedyDOptimal uses an unequal prob_T (which this class does not support), objective = "A" with interest = "all" or a covariate subset, or a prior_precision.

Constraints and fallbacks. Only exactly balanced allocation (`prob_T = 0.5`, n even) is supported; the constructor errors otherwise. If no covariates are available, the search degenerates to pure balanced Fisher-Yates randomization (no swap search, since there is nothing to balance on). `greedy_design_search_cpp()`'s output is trusted unvalidated – it guarantees exactly `n x r` valid $\{0, 1\}$ columns with $n/2$ treated subjects per column by construction (see fix_design_hierarchy.md, "AllocationMatrixValidation").

Super classes

`Design` -> `DesignFixed` -> `DesignFixedGreedy`

Methods

Public methods:

- `DesignFixedGreedy$new()`
- `DesignFixedGreedy$supports_batch_w_pregeneration()`
- `DesignFixedGreedy$clone()`

`DesignFixedGreedy$new()`: Initialize a greedy pairwise-swap search fixed experimental design. Only `prob_T = 0.5` is supported (see class documentation).

Usage:

```
DesignFixedGreedy$new(
  response_type,
  prob_T = 0.5,
  objective = "mahal_dist",
  n_iter = Inf,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` The data type of response values.

`prob_T` The probability of the treatment assignment. Must be 0.5.

`objective` The covariate-imbalance objective to minimize: either `"mahal_dist"` (default, squared Mahalanobis distance between treated and control covariate means) or `"abs_sum_diff"` (sum of absolute standardized mean differences); see class documentation for the exact criteria.

`n_iter` Number of swap iterations. `Inf` (default) uses exhaustive best-improvement search guaranteed to reach a strict local optimum. A positive integer runs that many stochastic random-pair iterations with patience-based early stopping.

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` Flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility. This design's search *is* reproducible via `seed` (see class documentation).

Returns: A new 'DesignFixedGreedy' object

`DesignFixedGreedy$supports_batch_w_pregeneration()`: Returns `TRUE` so the calling framework pre-generates all replicate `w` vectors for a simulation cell in a single batched call to `greedy_design_search_cpp()` (which parallelizes the `r` independent searches over OpenMP threads internally), rather than issuing `r` separate single-replicate C++ calls.

Usage:

```
DesignFixedGreedy$supports_batch_w_pregeneration()
```

Returns: Always `TRUE` for this class.

`DesignFixedGreedy$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignFixedGreedy$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Krieger, A. M., Azriel, D., and Kapelner, A. (2019). "Nearly random designs with greatly improved balance." *Biometrika*, 106(3), 695-701, doi:10.1093/biomet/asz026, for the greedy-swap balance-optimization approach this class implements. See also Mahalanobis distance for orientation on the default objective.

Examples

```
des = DesignFixedGreedy$new(n = 10, response_type = 'continuous')
```

`DesignFixedGreedyDOptimal`

A Fixed, Model-Based Optimal Design via Greedy Pairwise-Exchange Search

---

Description

A fixed-sample-size `DesignFixed` that searches, among allocations with exactly $n_T = \mathrm{round}(n \cdot \mathrm{prob}_T)$ treated subjects, for allocations optimizing a *model-based information-matrix* criterion implied by the linear model $y = \beta_T w + Z_0\gamma + \epsilon$ with $Z_0 = [1\ X]$, via a native C++ greedy pairwise-exchange (Fedorov/ DETMAX-style) local search. This class is the merger of the former `DesignFixedDOptimal` and `DesignFixedAOptimal` classes; the criterion is selected by the `objective` and `interest` constructor arguments.

The optimality-criterion family and its argument mapping. Write $M(w) = [w\ Z_0]^\top [w\ Z_0]$ for the information (moment) matrix, $P = Z_0(Z_0^\top Z_0)^{-1}Z_0^\top$, and $s(w) = n_T - w^\top P w$ (the treatment-coefficient information given the covariate block). The classical criteria map to constructor arguments as follows:

D_M **– full-matrix determinant optimality (maximize** $\det M(w)$**, i.e.** $|M|$**)** `objective = "D", interest = "all"`. By the Schur-complement identity $\det M(w) = \det(Z_0^\top Z_0)\cdot s(w)$ (with $w^\top w = n_T$ fixed and Z_0 not depending on w), the covariate block factors out, so D_M reduces to maximizing $s(w)$.

D_s **– subset determinant optimality (minimize** $\det(K^\top M(w)^{-1}K)$ **for a coordinate-selection** K**: the treatment coeffi**
the default `objective = "D", interest = "treatment"` is D_s with the interest set = {treatment}; `interest = ~ x1 + x2` (a one-sided formula) or `interest = c("x1", "x2")` (model-matrix column names) selects treatment + those covariates. Because w enters only the treatment row/column of $M(w)$, every such D_s criterion factorizes as $\det(V_{SS})/s(w)$ with $\det(V_{SS})$ constant in w – so *all* determinant-type settings (D_M and every D_s) select identical allocations and share the same search kernel. For the single treatment contrast, D_s-, c-, and per-parameter A-optimality coincide as well, which is why `objective = "A", interest = "treatment"` is silently equivalent to the default (allowed by design; no message is emitted).

D_A **– general contrast optimality (minimize** $\det(A^\top M(w)^{-1}A)$ **for an arbitrary contrast matrix** A**)**
`interest = <contrast matrix>` – arrives with Stage 2 of the merge plan (the generalized-criterion kernel) and currently raises an informative error, as do interest sets excluding the treatment coefficient.

D_B **(and** A_B**) – Bayesian optimality (criteria computed on the posterior information** $M(w) + R$**)**
`prior_precision =` a scalar τ or a matrix R_0, combined with either `objective`; see the Bayesian section below for exactly which coefficients a scalar τ penalizes.

A – trace optimality (minimize $\mathrm{tr}(K^\top M(w)^{-1}K)$**)** `objective = "A"` with `interest = "all"` (all parameters: objective $(w^\top H w + 1)/s(w)$, $H = Z_0(Z_0^\top Z_0)^{-2}Z_0^\top$), or with `interest =` formula/names (A_s: same kernel with the subset-restricted H_S; see below). Unlike the determinant family, trace criteria over different interest sets generally select *different* allocations.

Bayesian variants. Supplying `prior_precision` replaces $(Z_0^\top Z_0)^{-1}$ with the ridge-regularized $(Z_0^\top Z_0 + R_0)^{-1}$ in the construction of P (and H), yielding Bayesian D_B/A_B-optimality. A scalar τ penalizes the *covariate coefficients only* – the treatment coefficient and the intercept are unpenalized ($R_0 = \tau \cdot \mathrm{diag}(0, 1, \ldots, 1)$ over Z_0's columns) – and, when `standardize_covariates = TRUE` (the default), the covariates are centered and scaled to unit variance first so τ is interpretable per standardized coefficient. A full matrix `prior_precision` is used as R_0 verbatim (dimensions $(1+p) \times (1+p)$ over $[\mathrm{intercept}, \mathrm{covariates}]$ of the design's model matrix; `standardize_covariates` is ignored).

Search algorithm. For each of the `r` requested allocations independently: start from a uniformly random balanced-count allocation (a BCRD draw with exactly n_T treated), then repeatedly apply the single best improving treated/control pairwise exchange until no exchange improves the criterion (a strict local optimum). The returned allocations therefore form a restricted-randomization distribution over locally optimal allocations, which is what makes randomization inference possible for this design. **The search is reproducible via the constructor's** `seed` **argument**: the C++ kernels seed a local generator from R's own RNG stream, so a fixed `seed` yields identical draws (this corrects the former classes' documentation, which predated the RNG migration).

Relation to `DesignFixedGreedy`**: the same draws at the defaults.** With `objective = "D"` (any `interest`) or `objective = "A", interest = "treatment"`, no `prior_precision`, and `prob_T`

= 0.5, this class returns exactly the same allocations as DesignFixedGreedy with its defaults (objective = "mahal_dist", n_iter = Inf) for the same seed. This is a mathematical identity, not a coincidence. Write X_c for the centered covariates, S for their sample covariance, and $n_C = n - n_T$. Because the column space of $Z_0 = [1\ X]$ splits orthogonally into the intercept and X_c, $w^\top P w = n_T^2/n + (X_c^\top w)^\top (X_c^\top X_c)^{-1}(X_c^\top w)$, and $X_c^\top w = (n_T n_C/n)(\bar{x}_T - \bar{x}_C)$. Hence

$$s(w) = \frac{n_T n_C}{n} - \frac{n_T^2 n_C^2}{n^2(n-1)}(\bar{x}_T - \bar{x}_C)^\top S^{-1}(\bar{x}_T - \bar{x}_C) :$$

the information about the treatment effect is a constant minus a positive multiple of the squared Mahalanobis distance between the treated and control covariate means, which is DesignFixedGreedy's default criterion up to a positive factor. Maximizing one is minimizing the other, so the two rank every allocation the same way; both draw the same starting allocation from the seed and apply the same best-improving-swap rule, so they take the same steps to the same local optimum. In words: once the number of treated subjects is fixed, balancing the covariate means in Mahalanobis distance and maximizing the precision of the treatment effect in a linear model with an intercept are the same goal.

The two classes give different draws when:

- prob_T is not 0.5 (or n is odd). DesignFixedGreedy refuses these; this class runs them. The identity above holds for any fixed n_T, so this class is then the Mahalanobis-balancing greedy search at an unequal allocation.
- objective = "A" with interest = "all" or a covariate subset. The trace criterion $(w^\top H w + 1)/s(w)$ also weighs the precision of the covariate coefficients, so it is not a function of the Mahalanobis distance alone.
- prior_precision is supplied. The ridge term replaces S with a shrunken covariance, so the ranking moves away from plain Mahalanobis distance; the larger the prior precision, the more draws differ.
- DesignFixedGreedy is run with objective = "abs_sum_diff", with a finite n_iter (stochastic swaps), or falls back to "abs_sum_diff" because the Cholesky factorization of its covariate covariance fails.

The two kernels compute swap gains with different arithmetic, so a floating-point near-tie between two candidate swaps could in principle be broken differently. In practice, use DesignFixedGreedy for balanced Mahalanobis balancing (it also offers "abs_sum_diff" and stochastic search), and this class for unequal allocation, A-optimality, or a Bayesian prior.

Covariate-subset criteria (D_s/A_s) via interest = **a formula or names.** interest also accepts a one-sided formula (e.g. ~ x1 + x2) or a character vector of model-matrix column names, meaning *the treatment coefficient plus the named covariate coefficients* (the treatment is always in the interest set; the intercept never is). Both reduce to the existing kernels with no new machinery: under objective = "D", because w only enters the treatment row/column of $M(w)$, the subset determinant factorizes as $\det(K^\top M(w)^{-1} K) = \det(V_{SS})/s(w)$ with $\det(V_{SS})$ constant in w – so subset-D selects allocations *identical* to the default treatment-focused criterion (allowed silently, like objective = "A", interest = "treatment"); under objective = "A", the subset trace criterion is $(w^\top H_S w + 1)/s(w)$ with $H_S = (Z_0 V S)(Z_0 V S)^\top$ built from the selected columns – the same trace kernel with a subset-restricted H. Formula terms are expanded against the design's model matrix, so factor covariates must be referred to by their expanded model-matrix column names. Note that restricting the design's model matrix itself via design_formula also changes the default covariate set downstream inference adjusts for (Inference$initialize() inherits the design's formula),

whereas `interest` affects the allocation criterion only. General *contrast matrices* (D_A), and interest sets excluding the treatment coefficient, arrive with Stage 2 of the merge plan (the generalized-criterion kernel; see `package_metadata/finished_features/fix_design_hierarchy.md`).

Constraints and fallbacks. `prob_T` may be any value in $(0, 1)$ for which $1 \leq \mathrm{round}(n \cdot \mathrm{prob}_T) \leq n - 1$. If no covariates are available, the search degenerates to pure random allocation with n_T treated (there is no criterion to optimize).

Super classes

`Design` -> `DesignFixed` -> `DesignFixedGreedyDOptimal`

Methods

Public methods:

- `DesignFixedGreedyDOptimal$new()`
- `DesignFixedGreedyDOptimal$get_objective()`
- `DesignFixedGreedyDOptimal$get_interest()`
- `DesignFixedGreedyDOptimal$get_prior_precision()`
- `DesignFixedGreedyDOptimal$clone()`

`DesignFixedGreedyDOptimal$new()`: Initialize a model-based optimal-search fixed experimental design. Covariates, if any, are supplied later via `add_all_subjects_to_experiment()`; the optimality search itself does not run until `assign_w_to_all_subjects()` (or `draw_ws_according_to_design()`) is called.

Usage:

```
DesignFixedGreedyDOptimal$new(
  response_type,
  prob_T = 0.5,
  objective = "D",
  interest = "treatment",
  prior_precision = NULL,
  standardize_covariates = TRUE,
  n_iter = Inf,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` "continuous", "incidence", "proportion", "count", "survival", or "ordinal". Determines only which downstream inference/response machinery this design is paired with; it does not affect the optimality search itself.

`prob_T` Probability of treatment assignment, in $(0, 1)$. The search fixes the treated count at $\mathrm{round}(n \cdot prob_T)$.

`objective` The optimality criterion: `"D"` (default, determinant) or `"A"` (trace). See the class documentation for the exact criteria and for why `objective = "A"` with `interest = "treatment"` is equivalent to the default.

`interest` Which parameters the criterion targets: `"treatment"` (default), `"all"`, a one-sided formula (e.g. `~ x1 + x2`), a single formula *string* (e.g. `"x1 * x2 + x7"`, promoted to `~ x1 * x2 + x7`), or a character vector of model-matrix column names – all but `"all"` meaning the treatment coefficient plus the named covariate coefficients (D_s/A_s; see class documentation, including why subset-D selects the same allocations as the default). Formula terms (including interactions like `x1:x2`) must correspond to columns of the design's model matrix: to target an interaction coefficient, the interaction must be in `design_formula` too – you cannot be "interested in" a coefficient the working model does not contain. Contrast matrices (general D_A) arrive with Stage 2 of the merge plan and currently raise an error.

`prior_precision` `NULL` (default, non-Bayesian), a single positive scalar τ (ridge prior precision on the covariate coefficients only; treatment and intercept unpenalized), or a full $(1+p) \times (1+p)$ symmetric prior-precision matrix R_0 over $[\text{intercept}, \text{covariates}]$.

`standardize_covariates` If `TRUE` (default) and `prior_precision` is a scalar, covariates are centered and scaled to unit variance before the penalized criterion matrices are built. Ignored otherwise.

`n_iter` Number of exchange iterations. `Inf` (default) runs the exhaustive best-improvement search to a strict local optimum. Finite values (the stochastic swap mode shared with `DesignFixedGreedy`) arrive with the Stage-2 shared search engine and currently raise an error.

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` Sample size (if fixed).

`verbose` Flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility. Unlike the former `DesignFixedDOptimal/DesignFixedAOptimal` documentation claimed, the optimality search *is* reproducible via `seed` (see class documentation).

Returns: A new ‘DesignFixedGreedyDOptimal‘ object

`DesignFixedGreedyDOptimal$get_objective()`: The optimality criterion this design was constructed with.

Usage:

```
DesignFixedGreedyDOptimal$get_objective()
```

Returns: `"D"` or `"A"`.

`DesignFixedGreedyDOptimal$get_interest()`: The parameter-interest setting this design was constructed with.

Usage:

```
DesignFixedGreedyDOptimal$get_interest()
```

Returns: `"treatment"` or `"all"`.

`DesignFixedGreedyDOptimal$get_prior_precision()`: The Bayesian prior precision this design was constructed with.

Usage:

```
DesignFixedGreedyDOptimal$get_prior_precision()
```

Returns: NULL, a positive scalar, or a symmetric matrix.

`DesignFixedGreedyDOptimal$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignFixedGreedyDOptimal$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

References

Atkinson, A. C., Donev, A. N., and Tobias, R. D. (2007). *Optimum Experimental Designs, with SAS*. Oxford University Press, for the D-/A-optimality criteria and exchange algorithms for constrained design search. See also optimal design for orientation.

Examples

```
des = DesignFixedGreedyDOptimal$new(n = 10, response_type = 'continuous')
des$add_all_subjects_to_experiment(data.frame(x1 = rnorm(10)))
des$assign_w_to_all_subjects()
```

DesignFixediBCRD — *A Fixed, Individually Balanced Completely Randomized Design (iBCRD)*

Description

A fixed-sample-size DesignFixed implementing the individually balanced complete randomized design (iBCRD): the number of treated subjects is fixed at exactly $n_T = \mathrm{round}(n \cdot prob_T)$, and each allocation w with exactly n_T ones is drawn uniformly at random from the $\binom{n}{n_T}$ possible such allocations (via a Fisher-Yates shuffle of a base vector with n_T ones and $n - n_T$ zeros). This is the classical "complete randomization" reference design of randomization inference: unlike DesignFixedBernoulli (independent per-subject coin flips, random n_T), n_T is fixed here, which is what makes exact permutation/randomization tests over the $\binom{n}{n_T}$ allocations well-defined; unlike DesignFixedGreedyDOptimal/ DesignFixedGreedy, no covariate information is used to select among those allocations — every one of the $\binom{n}{n_T}$ allocations is equally likely.

Draw mechanism. draw_ws_raw(r) delegates to generate_permutations_ibcrd_cpp(), which builds one base allocation vector (n_T ones followed by $n - n_T$ zeros) and independently shuffles (portable Fisher-Yates) a fresh copy of it per replicate column, seeded from R's own RNG stream (so seed does govern reproducibility here, unlike the A-/D-optimal exchange searches). assign_w_to_all_subjects() draws one such allocation (r = 1) and applies it to all subjects.

Single implicit block. The constructor sets private$m to a constant vector of 1s (a single block containing every subject) once n is known, so that shared blocking/matching machinery that expects a block-membership vector treats the whole sample as one block by default.

Super classes

Design -> DesignFixed -> DesignFixediBCRD

Methods

Public methods:

- DesignFixediBCRD$new()
- DesignFixediBCRD$clone()

DesignFixediBCRD$new(): Initialize a fixed individually balanced completely randomized experimental design (see class documentation for the exact randomization law).

Usage:

```
DesignFixediBCRD$new(
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

response_type "continuous", "incidence", "proportion", "count", "survival", or "ordinal".

prob_T Target probability of treatment assignment; the realized number of treated subjects is fixed at round(n * prob_T) for every draw (unlike DesignFixedBernoulli, where it is random).

include_is_missing_as_a_new_feature Flag for missingness indicators.

n The sample size.

verbose A flag for verbosity.

missingness_method How to handle missing values in covariates.

design_formula A formula object.

seed Integer seed for reproducibility.

Returns: A new ‘DesignFixediBCRD‘ object

DesignFixediBCRD$clone(): The objects of this class are cloneable with this method.

Usage:

```
DesignFixediBCRD$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

References

Fisher, R. A. (1935). *The Design of Experiments*. Oliver and Boyd, for complete randomization as the canonical reference design of randomization inference. See also randomized experiment for orientation on complete vs. Bernoulli randomization.

Examples

```
des = DesignFixediBCRD$new(n = 10, response_type = 'continuous')
des$add_all_subjects_to_experiment(data.frame(x1 = rnorm(10)))
des$assign_w_to_all_subjects()
```

DesignFixedMatchingGreedyPairSwitching

A Fixed, Matched-Pair Design with Greedy Which-Member-Treated Optimization

---

Description

A fixed-sample-size DesignFixed that combines DesignFixedBinaryMatch's non-bipartite matched-pair structure with DesignFixedGreedy's greedy imbalance-minimization search, restricted so every move respects the pairing: subjects are first paired by covariate closeness (as in DesignFixedBinaryMatch), guaranteeing exactly one treated and one control subject per pair; then, rather than assigning within-pair treatment status by a coin flip, the greedy search (greedy_design_search_cpp(), pair-constrained mode) chooses *which* member of each pair is treated so as to directly minimize the same aggregate covariate-imbalance objective as DesignFixedGreedy (squared Mahalanobis distance or sum of absolute standardized mean differences between the treated and control group means) across the *whole* sample, not just within each pair. This targets both close within-pair matches (from the matching step) and low aggregate covariate imbalance (from the greedy refinement) simultaneously — a strictly more constrained search than plain DesignFixedGreedy, since only the $2^{n/2}$ which-member-treated assignments consistent with the fixed pairing are considered, rather than all $\binom{n}{n/2}$ balanced allocations.

Search algorithm. Pairing is computed by compute_binary_match_structure() exactly as in DesignFixedBinaryMatch (Mahalanobis or Euclidean distance per objective), lazily on first draw and cached in private$bms. Given the pairing, each replicate search initializes with a random coin flip per pair (which member starts treated), then in exhaustive mode (n_iter = Inf, default) repeatedly finds and applies the single pair-flip that most decreases the imbalance objective, stopping at a strict local optimum (or runs exactly n_iter random-pair stochastic flip-if-improving steps otherwise, with patience-based early stopping) — the same two search modes as DesignFixedGreedy, but with moves restricted to "flip which side of a given pair is treated" rather than "swap any treated/control pair of subjects." Random initialization and swap selection are seeded from R's own RNG (via greedy_design_search_cpp(), which derives one stream per draw index from a master seed), so seed governs reproducibility here and the draws are the same at every set_num_cores() setting.

Pair-preserving bootstrap. draw_bootstrap_indices() resamples whole matched pairs (via draw_matching_bootstrap_sample_cpp()) rather than individual subjects, since the greedy search only ever flips which member of a pair is treated (never crosses pairs), so w always has exactly one treated subject per pair — the pair, not the subject, is the exchangeable resampling unit.

Constraints. Only prob_T = 0.5 is supported (the constructor errors otherwise), and n must be divisible by 4 (draw_ws_raw() errors otherwise); n/2 matched pairs are formed regardless of parity, but the additional divisible-by-4 requirement is enforced by this class specifically (unlike DesignFixedBinaryMatch, which only requires even n).

Super classes

Design -> DesignFixed -> DesignFixedMatchingGreedyPairSwitching

Methods

Public methods:

- DesignFixedMatchingGreedyPairSwitching$new()
- DesignFixedMatchingGreedyPairSwitching$supports_batch_w_pregeneration()
- DesignFixedMatchingGreedyPairSwitching$clone()

DesignFixedMatchingGreedyPairSwitching$new(): Initialize a fixed design that performs binary matching followed by greedy which-member-treated optimization (see class documentation). Only prob_T = 0.5 is supported, and n must be divisible by 4.

Usage:

```
DesignFixedMatchingGreedyPairSwitching$new(
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n,
  verbose = FALSE,
  objective = "mahal_dist",
  match_objective = c("distance", "squared_distance"),
  n_iter = Inf,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

response_type The data type of response values.

prob_T The probability of treatment assignment. Must be 0.5.

include_is_missing_as_a_new_feature Flag for missingness indicators.

n The sample size; must be divisible by 4.

verbose A flag for verbosity.

objective The covariate-imbalance objective to minimize when choosing which pair member is treated: either "mahal_dist" (default, squared Mahalanobis distance between treated/control means, also used as the matching distance) or "abs_sum_diff" (sum of absolute standardized mean differences); see class documentation for the exact criteria.

match_objective Matching objective used to form pairs. "distance" (default) matches **nbpMatching**; "squared_distance" reproduces EDI through version 1.0.4.

n_iter Number of swap iterations. Inf (default) uses exhaustive best-improvement search guaranteed to reach a strict local optimum. A positive integer runs that many stochastic random-pair iterations with patience-based early stopping.

missingness_method How to handle missing values in covariates.

design_formula A formula object.

seed Integer seed for reproducibility.

Returns: A new `DesignFixedMatchingGreedyPairSwitching` object.

`DesignFixedMatchingGreedyPairSwitching$supports_batch_w_pregeneration()`**:** Returns `TRUE` so the calling framework pre-generates all replicate `w` vectors for a simulation cell in one batched call to `greedy_design_search_cpp()`, paying the one-time **nbpMatching** pairing cost once per cell (cached in `private$bms`) and reusing it across all replicates and the OpenMP-parallelized greedy searches, rather than recomputing the pairing per replicate.

Usage:

```
DesignFixedMatchingGreedyPairSwitching$supports_batch_w_pregeneration()
```

Returns: Always `TRUE` for this class.

`DesignFixedMatchingGreedyPairSwitching$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DesignFixedMatchingGreedyPairSwitching$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Krieger, A. M., Azriel, D., and Kapelner, A. (2019). "Nearly random designs with greatly improved balance." *Biometrika*, 106(3), 695-701, doi:10.1093/biomet/asz026; Greevy, R., Lu, B., Silber, J. H., and Rosenbaum, P. (2004). "Optimal multivariate matching before randomization." *Biostatistics*, 5(2), 263-275, doi:10.1093/biostatistics/5.2.263, for the matched-pair design this class refines.

Examples

```
des = DesignFixedMatchingGreedyPairSwitching$new(n = 10, response_type = 'continuous')
```

`DesignFixedOptimal` *A Fixed, Deterministic Single-Allocation Optimal Design*

Description

A fixed-sample-size `DesignFixed` that computes **exactly one** allocation w^* – the minimizer of a chosen covariate-imbalance or information objective over all allocations with $n_T = \mathrm{round}(n \cdot \mathrm{prob}_T)$ treated subjects – by numerical optimization, rather than drawing from a restricted-randomization distribution the way `DesignFixedGreedy`/ `DesignFixedGreedyDOptimal` do.

The objective family and its argument mapping. Write $Z_0 = [1\ X]$, $P = Z_0(Z_0^\top Z_0)^{-1}Z_0^\top$, and $s(w) = n_T - w^\top P w$. The objectives map to constructor arguments and solved forms as follows:

`"D"` **(determinant /** D_M**,** D_s**) and** `"A"` **with** `interest = "treatment"` maximize $s(w)$, solved as the binary quadratic program $\min_w w^\top P w$ – identical criteria and `interest`/`prior_precision` semantics to `DesignFixedGreedyDOptimal` (the same shared construction machinery is used, so the two classes optimize literally the same matrices).

"A" **with** interest = "all" **or a covariate subset** (A, A_s) minimize $(w^\top Hw+1)/s(w)$ with the sibling class's H/H_S, solved exactly via Dinkelbach's algorithm (Dinkelbach 1967) over product-linearized MILP subproblems.

Bayesian D_B/A_B prior_precision = a scalar τ (covariates only; intercept and treatment unpenalized) or a full matrix R_0, replacing $(Z_0^\top Z_0)^{-1}$ with the ridge-regularized inverse – identical to the sibling class.

"mahal_dist" / "abs_sum_diff" DesignFixedGreedy's covariate-imbalance criteria, definitionally identical (column-centered X, the same standardization and singular-covariance fallback), translated to exactly solvable forms: the Mahalanobis criterion is the pure quadratic $w^\top Qw$ with $Q = 4X\Sigma^{-1}X^\top/n^2$, and the absolute-sum criterion is an l1 objective solved by the standard linear MILP.

"custom" a user-compiled black box under the user_compiled_fns.h calling convention (double f(const Eigen::MatrixXd& X, const Eigen::VectorXd& w), minimized), supplied via custom_objective; always solved by the annealing path (no structure to linearize).

Solvers and certificates. solver = "auto" (default) uses the exact "ompr" MILP path (optimum_certificate = "global", a certified global optimum) wherever tractable – always for "abs_sum_diff" (pure linear MILP); up to solver_args$linearization_max_n (default 20, set by a GLPK benchmark: the product linearization adds $n(n-1)/2$ auxiliaries and branch-and-bound cost climbs steeply past $n \approx 20$) for the quadratic and Dinkelbach criteria – and the native simulated-annealing solver beyond it, or always for "custom". The annealing solver is a formal method, not a heuristic: Metropolis acceptance over treated/control swaps with a configurable cooling schedule, for which Hajek (1988) proves convergence in probability to the global optimum under a slow-enough (logarithmic) schedule; the practical geometric schedule used by default is asymptotically motivated only, so its certificate is always "annealing_converged", never "global". solver = "ompr"/"annealing" force a path. Commercial backends extend the exact range via solver_args$roi_solver; see that parameter's wiring guides.

Inference. There is no usable randomization distribution conditional on the observed data (given X there is exactly one w^* up to the mirror coin), so permutation-style randomization tests/CIs are unavailable (supports_randomization_draw() is FALSE); the bootstrap randomization test IS available (the mechanism – "optimize this dataset" – is replayed on each resampled covariate matrix), as is all model-based and plain-resampling inference.

The mirror coin. At prob_T = 0.5, whenever the mirror $1-w^*$ is a verified co-optimum (checked numerically by evaluating the objective, never by a symmetry table), a fair seeded coin picks between w^* and its mirror (mirror_coin = TRUE, the default). This restores exact treated/control label symmetry – and estimator unbiasedness – at zero cost to balance. A mirror that evaluates strictly better than the solver's answer raises an error (it would be a solver bug).

Super classes

Design -> DesignFixed -> DesignFixedOptimal

Methods

Public methods:

- DesignFixedOptimal$new()
- DesignFixedOptimal$get_objective()

- DesignFixedOptimal$get_interest()
- DesignFixedOptimal$get_prior_precision()
- DesignFixedOptimal$get_solver()
- DesignFixedOptimal$get_mirror_coin()
- DesignFixedOptimal$get_optimization_diagnostics()
- DesignFixedOptimal$supports_randomization_draw()
- DesignFixedOptimal$prepare_for_resampling_replay()
- DesignFixedOptimal$clone()

DesignFixedOptimal$new(): Initialize a deterministic single-allocation optimal fixed experimental design. The optimization itself does not run until assign_w_to_all_subjects() (or draw_ws_according_to_design(r = 1)) is called.

Usage:

```
DesignFixedOptimal$new(
  response_type,
  prob_T = 0.5,
  objective = "D",
  interest = "treatment",
  prior_precision = NULL,
  standardize_covariates = TRUE,
  custom_objective = NULL,
  solver = "auto",
  solver_args = list(),
  mirror_coin = TRUE,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

response_type The data type of response values.

prob_T Probability of treatment assignment, in $(0, 1)$; the solve fixes the treated count at $\mathrm{round}(n \cdot prob_T)$.

objective "D" (default), "A", "mahal_dist", "abs_sum_diff", or "custom"; see the class documentation.

interest For objective = "D"/"A" only: "treatment" (default), "all", a one-sided formula, a formula string, or model-matrix column names – identical semantics to DesignFixedGreedyDOptimal.

prior_precision For objective = "D"/"A" only: NULL (default), a positive scalar τ, or a symmetric prior-precision matrix R_0 – identical semantics to DesignFixedGreedyDOptimal.

standardize_covariates If TRUE (default) and prior_precision is a scalar, covariates are standardized before the penalized criterion matrices are built. ("mahal_dist"/"abs_sum_diff" standardize internally per their definitions regardless.)

custom_objective Required iff objective = "custom" (and forbidden otherwise): an RcppXPtrUtils::cppXPtr() external pointer or a C++ source string under the user_compiled_fns.h calling convention

(double f(const Eigen::MatrixXd& X, const Eigen::VectorXd& w); X is the design's model matrix, w a candidate 0/1 allocation; the returned value is *minimized*). A source string is compiled through the same cppXPtr mechanism and retained so parallel workers can recompile locally. **A plain R function is not accepted, and cannot be:** the annealing solver evaluates the objective once per candidate swap – typically thousands of times per chain, times n_chains, and again per BRT replicate – and an R-call round-trip on every one of those evaluations is orders of magnitude too slow to be usable, not merely slower. Since "custom" is always solved by annealing (never the MILP path), there is no lower-frequency code path where an R closure would be merely inconvenient; the restriction is a hard performance requirement of this objective's only execution path. See the class examples for a worked cppXPtr() construction. **Save/ reload note:** a raw cppXPtr() object does not survive saveRDS()/readRDS() (see Design's "Saving and loading" section); pass a C++ source string instead if this design needs to be reloadable.

solver "auto" (default), "ompr", or "annealing".

solver_args A named list of solver tuning arguments. Supported: roi_solver ("glpk"/"gurobi"/"cplex" – a *closed* set; arbitrary ROI plugin names are rejected), linearization_max_n, max_dinkelbach_iter, n_chains, max_iter, initial_temp, cooling_rate, and the explicit BRT-replay override brt_solver with its annealing controls brt_max_iter and brt_n_chains. Without brt_solver, BRT replays inherit the observed assignment's ordinary solver profile.

Wiring up Gurobi (roi_solver = "gurobi"): (1) obtain a Gurobi license (free academic licenses are available) and install the Gurobi Optimizer itself – this sets up GUROBI_HOME and the license file, entirely outside this package's control; (2) install Gurobi's own R package, which is *not on CRAN* – it ships inside the Gurobi installation: R CMD INSTALL "$GUROBI_HOME/R/gurobi_<version>_R_<Rmajor.minor>.tar.gz" (exact filename depends on your Gurobi version and platform); (3) install.packages("ROI.plugin.gurobi") from CRAN; (4) verify "gurobi" %in% ROI::ROI_registered_solvers() after loading the plugin; (5) pass solver_args = list(roi_solver = "gurobi").

Wiring up CPLEX (roi_solver = "cplex"): (1) obtain an IBM CPLEX license (free academic licenses are available) and install IBM ILOG CPLEX Optimization Studio; (2) install Rcplex (CRAN) – unlike the Gurobi bridge, it *compiles from source against your local CPLEX SDK* and must be pointed at your CPLEX version's include/lib directories at install time; follow Rcplex's own INSTALL instructions for your CPLEX version rather than a fixed command, since the flags change across CPLEX releases; (3) install.packages("ROI.plugin.cplex") from CRAN; (4) verify "cplex" %in% ROI::ROI_registered_solvers(); (5) pass solver_args = list(roi_solver = "cplex").

ROI.plugin.gurobi/ROI.plugin.cplex/Rcplex are deliberately *never* listed in this package's Suggests: declaring them would misrepresent the dependency as something install.packages() could satisfy, when the vendor installation/license underneath cannot be. Availability is checked lazily at solve time; if the plugin loads but the solve fails, the likely cause is a missing vendor installation or license.

mirror_coin If TRUE (default), flip a fair seeded coin between w^* and a *verified co-optimal* mirror $1-w^*$ after every solve (only possible at prob_T = 0.5); see the class documentation.

include_is_missing_as_a_new_feature Flag for missingness indicators.

n Sample size (if fixed).

verbose Flag for verbosity.

missingness_method How to handle missing values in covariates.

design_formula A formula object.

`seed` Integer seed for reproducibility (consumed by the annealing solver and the mirror coin; MILP solves are deterministic up to the seeded label flip).

Returns: A new ‘DesignFixedOptimal‘ object

`DesignFixedOptimal$get_objective()`: The objective this design was constructed with.

Usage:

```
DesignFixedOptimal$get_objective()
```

Returns: One of `"D"`, `"A"`, `"mahal_dist"`, `"abs_sum_diff"`, `"custom"`.

`DesignFixedOptimal$get_interest()`: The parameter-interest setting (`"D"`/`"A"` only).

Usage:

```
DesignFixedOptimal$get_interest()
```

Returns: The `interest` construction argument.

`DesignFixedOptimal$get_prior_precision()`: The Bayesian prior precision this design was constructed with.

Usage:

```
DesignFixedOptimal$get_prior_precision()
```

Returns: `NULL`, a positive scalar, or a symmetric matrix.

`DesignFixedOptimal$get_solver()`: The solver setting this design was constructed with.

Usage:

```
DesignFixedOptimal$get_solver()
```

Returns: `"auto"`, `"ompr"`, or `"annealing"`.

`DesignFixedOptimal$get_mirror_coin()`: The mirror-coin setting this design was constructed with.

Usage:

```
DesignFixedOptimal$get_mirror_coin()
```

Returns: `TRUE` or `FALSE`.

`DesignFixedOptimal$get_optimization_diagnostics()`: Diagnostics cached by the most recent solve: the solver used, `optimum_certificate` (`"global"` for exact `"ompr"` solves, `"annealing_converged"` otherwise), the achieved objective value, mirror-coin outcome (`mirror_feasible`/`mirror_tied`/`mirror_flipped`), elapsed time, and the solver's own detail fields.

Usage:

```
DesignFixedOptimal$get_optimization_diagnostics()
```

Returns: A named list, or `NULL` if no solve has run yet.

`DesignFixedOptimal$supports_randomization_draw()`: Characterization: `FALSE` – given the observed data there is exactly one w^* (up to the vacuous 2-atom mirror pair), so there is no randomization distribution to draw from and permutation-style randomization tests/CIs are unavailable. The bootstrap randomization test remains available via `supports_resampling_replay()` (the deterministic mechanism is replayed on each resample).

Usage:

```
DesignFixedOptimal$supports_randomization_draw()
```

Returns: Always `FALSE` for this class.

`DesignFixedOptimal$prepare_for_resampling_replay()`: BRT replicate-mode switch (called by the bootstrap-randomization-test machinery ahead of each replayed draw; see `Design$prepare_for_resampling_repl` Subsequent solves inherit the observed assignment's ordinary solver and budget, so each replay targets the same allocation mechanism. Set `solver_args$brt_solver = "annealing"` to opt into the legacy reduced `brt_max_iter`/`brt_n_chains` replay profile, or `"ompr"` to request the MILP path when the objective permits it. Idempotent; the mirror coin still applies per replicate (the BRT replays the coin-inclusive mechanism).

Usage:

```
DesignFixedOptimal$prepare_for_resampling_replay()
```

Returns: `invisible(NULL)`.

`DesignFixedOptimal$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignFixedOptimal$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Dinkelbach, W. (1967). On nonlinear fractional programming. *Management Science* 13(7):492-498, for the exact A-optimality reduction. Hajek, B. (1988). Cooling schedules for optimal annealing. *Mathematics of Operations Research* 13(2):311-329, for the annealing solver's formal convergence property. Atkinson, A. C., Donev, A. N., and Tobias, R. D. (2007). *Optimum Experimental Designs, with SAS*. Oxford University Press, for the D-/A-optimality criteria.

Examples

```
# objective = "mahal_dist" (the default MILP path) needs a MILP solver --
# ompr + ompr.roi + a ROI plugin (ROI.plugin.glpk by default), all Suggests,
# not installed automatically with EDI. Confirmed 2026-09-25: ungated, this
# example errored outright on R CMD check's --no-suggests leg (and would do
# the same for any real user without the optional MILP-solver stack), not
# just a CI-specific issue.
if (requireNamespace("ompr", quietly = TRUE) &&
    requireNamespace("ompr.roi", quietly = TRUE) &&
    requireNamespace("ROI.plugin.glpk", quietly = TRUE)) {
 des = DesignFixedOptimal$new(n = 14, response_type = 'continuous', objective = "mahal_dist")
  des$add_all_subjects_to_experiment(data.frame(x1 = rnorm(14)))
  des$assign_w_to_all_subjects()
  des$get_optimization_diagnostics()
}


# A custom compiled objective (the user_compiled_fns.h calling convention),
```

```
# built with RcppXPtrUtils::cppXPtr() -- here, squared imbalance of the
# centered covariate sums. Compiling it needs a C++ toolchain and takes
# several seconds:
if (requireNamespace("RcppXPtrUtils", quietly = TRUE)) {
  fobj = RcppXPtrUtils::cppXPtr(
    "double f(const Eigen::MatrixXd& X, const Eigen::VectorXd& w) {
      Eigen::RowVectorXd mu = X.colwise().mean();
      Eigen::MatrixXd Xc = X.rowwise() - mu;
      Eigen::VectorXd s = 2.0 * w - Eigen::VectorXd::Ones(X.rows());
      return (Xc.transpose() * s).squaredNorm();
    }", depends = "RcppEigen")
  des2 = DesignFixedOptimal$new(n = 14, response_type = 'continuous',
    objective = "custom", custom_objective = fobj)
  des2$add_all_subjects_to_experiment(data.frame(x1 = rnorm(14)))
  des2$assign_w_to_all_subjects()
}
```

DesignFixedOptimalBlocks

A Fixed, Covariate-Homogeneous-Block Randomized Design

Description

A fixed-sample-size DesignFixed that first partitions subjects into B approximately equal-sized, covariate-homogeneous blocks by (approximately or exactly) minimizing total within-block pairwise covariate distance $\sum_{k=1}^{B} \sum_{i,j \in \text{block } k, i<j} D(x_i, x_j)$, then randomizes treatment independently within each resulting block at probability prob_T (via block_ra, the same mechanism as DesignFixedBlocking). Unlike DesignFixedBlocking, which forms blocks from user-specified column-wise strata (categorical levels / quantile bins), here blocks are formed directly from a multivariate distance/clustering criterion over all covariates jointly, so this design generalizes matched-pair designs (DesignFixedBinaryMatch) from block size 2 to arbitrary block size B. When B is omitted and n is known at initialization, the default is floor(sqrt(n)), truncated below at 1 (the usual heuristic block count for balancing within-block homogeneity against within-block sample size).

Block-formation algorithms. method selects among three block construction strategies with different exactness/scalability trade-offs (see the new() argument documentation for details): "K-way" (default, balanced k-means-style anticlustering via **anticlust**, fast and empirically close to optimal), "greedy" (nearest-neighbor greedy matching via **blockTools**, fast even for large n), and "ompr" (an exact mixed-integer program solved with GLPK via **ompr/ompr.roi**, globally optimal but scaling as $O(n^2 B)$ in the number of decision variables — practical only for small n). Block membership is computed lazily (on first draw, via get_or_compute_block_ids()) and cached in private$block_ids for reuse across replicates.

Distance specification ("ompr" only). dist selects the pairwise distance $D(x_i, x_j)$ the exact solver minimizes: "euclidean", "sum_abs_diff" (sum of absolute coordinate differences), "mahal" (default, Mahalanobis distance accounting for covariate correlation/scale), or a user-supplied distance function. The "K-way" and "greedy" methods use their own respective packages' built-in distance conventions and do not consult dist.

No-covariate fallback. If the covariate matrix has zero columns, block membership is assigned by simple round-robin (`rep(seq_len(B), length.out = n)`) rather than by any of the three clustering algorithms, since there is no covariate information to cluster on.

Solver backend (`method = "ompr"` only). `roi_solver` selects the MILP backend **ompr.roi** dispatches to, a **closed set** `c("glpk", "gurobi", "cplex")` (default `"glpk"`) – validated against this set, not passed through to `ROI::ROI_registered_solvers()` unchecked, so a typo or an unsupported solver name fails fast with a clear message rather than an opaque **ompr** error three layers down. Gurobi and CPLEX are supported because they extend which block-formation problems stay practical, not just which are expressible: GLPK's branch-and-bound is single-threaded with no commercial-grade presolve/cutting-plane machinery, while Gurobi/CPLEX are typically an order of magnitude faster on the same MILP and solve in parallel, meaningfully extending the n for which the exact `"ompr"` method stays practical. Scope is deliberately closed to these two for now – not because other ROI plugins (CBC, SYMPHONY, ...) wouldn't work mechanically (the dispatch is generic), but because Gurobi and CPLEX are the two most widely used commercial solvers and the only ones worth a maintained step-by-step guide at this point; extending the closed set is a small, low-risk addition later if a real need for a third backend appears, not a reason to leave the set open-ended now.

Wiring up Gurobi:

1. Obtain a Gurobi license (a free academic license is available from Gurobi for non-commercial use) and install the Gurobi Optimizer itself. This sets up `GUROBI_HOME` and the license file (`gurobi.lic`, discoverable via the `GRB_LICENSE_FILE` environment variable or Gurobi's default search path) – entirely outside this package's control or dependency graph.
2. Install Gurobi's own R package. **Not available via CRAN** – it ships inside the Gurobi installation itself: `R CMD INSTALL "$GUROBI_HOME/R/gurobi_<version>_R_<Rmajor.minor>.tar.gz"` (exact filename/path depends on your Gurobi version and platform; see the `R/` subdirectory of your Gurobi install). This is the vendor interface `ROI.plugin.gurobi` wraps – required even though it's not what you call directly.
3. Install the ROI bridge package from CRAN: `install.packages("ROI.plugin.gurobi")`. This package **is** on CRAN (it only depends on **ROI** + the `gurobi` R package from step 2 being present at load time) and is the only new artifact this class's own dependency graph ever touches.
4. Verify: after `library(ROI.plugin.gurobi)`, `"gurobi" %in% ROI::ROI_registered_solvers()` should be `TRUE`.
5. Pass `roi_solver = "gurobi"` to the constructor.

Wiring up CPLEX:

1. Obtain an IBM CPLEX license (a free academic license is available from IBM) and install IBM ILOG CPLEX Optimization Studio.
2. Install **Rcplex** (CRAN), CPLEX's R interface. Unlike `ROI.plugin.gurobi`, **Rcplex** is a **source package that compiles against your local CPLEX installation** – it needs to be pointed at your CPLEX SDK's include/lib directories at install time (typically via `configure.args` to `install.packages()`, naming your CPLEX version's `cplex/include/cplex/lib/<platform>` paths). The exact flag names and paths are CPLEX-version- and platform-specific – **follow Rcplex's own `INSTALL`/README instructions for your installed CPLEX version** rather than a fixed command copied from here, since this changes across CPLEX releases.

3. Install the ROI bridge package from CRAN: `install.packages("ROI.plugin.cplex")` (depends on **Rcplex** from step 2 being present and working).
4. Verify: after `library(ROI.plugin.cplex)`, `"cplex" %in% ROI::ROI_registered_solvers()` should be `TRUE`.
5. Pass `roi_solver = "cplex"` to the constructor.

Dependency-graph consequence: `ROI.plugin.gurobi`/`ROI.plugin.cplex` (and **Rcplex**) are **never added to** `Suggests` – they're free/CRAN- available themselves, but declaring them would misrepresent the dependency as something `install.packages("EDI", dependencies = TRUE)` could satisfy, when the vendor package/license underneath cannot be. The lazy-check pattern already used for **ompr/ompr.roi**/`ROI.plugin.glpk` extends naturally: check `requireNamespace("ROI.plugin.gurobi"/"ROI.plugin` at solve time (not at package load or class-definition time) for whichever `roi_solver` was requested, and error informatively – naming the missing package and, if that's present but the solve still fails, noting the likely cause is a missing vendor license/installation, not something this class can diagnose further.

Bootstrap. `draw_bootstrap_indices()` resamples within blocks by default (`bootstrap_type = "within_blocks"` or `NULL`, via `stratified_bootstrap_indices_cpp()`) or resamples whole blocks otherwise (via `resample_group_rows_cpp()`), mirroring the block structure in the resampling scheme, as in `DesignFixedBlocking`.

Super classes

`Design` -> `DesignFixed` -> `DesignFixedOptimalBlocks`

Methods

Public methods:

- `DesignFixedOptimalBlocks$supports_batch_w_pregeneration()`
- `DesignFixedOptimalBlocks$new()`
- `DesignFixedOptimalBlocks$clone()`

`DesignFixedOptimalBlocks$supports_batch_w_pregeneration()`**:** Returns `TRUE` so the calling framework pre-generates all replicate `w` vectors for a simulation cell in one batch, paying the one-time block-formation cost (K-way anticlustering, greedy matching, or the exact **ompr**/GLPK solve) once per cell and reusing the resulting block assignment across replicates, rather than recomputing it per replicate.

Usage:

```
DesignFixedOptimalBlocks$supports_batch_w_pregeneration()
```

Returns: Always `TRUE` for this class.

`DesignFixedOptimalBlocks$new()`**:** Initialize a fixed optimal-blocks design. Block formation itself is deferred until the first draw (see class documentation); this constructor only validates and records configuration, including checking that `B` (or its `floor(sqrt(n))` default) admits a feasible block-size partition of `n` when `n` is already known.

Usage:

```
DesignFixedOptimalBlocks$new(
  B = NULL,
  method = "K-way",
  dist = "mahal",
  roi_solver = "glpk",
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`B` Number of blocks to form. If omitted and `n` is supplied, defaults to `floor(sqrt(n))`, with a minimum of 1.

`method` Algorithm used to partition subjects into blocks.

- `"K-way"` **(default)** Balanced k-means anticlustering via `anticlust::balanced_clustering`. Requires the **anticlust** package. Produces well-spread blocks and is significantly faster than `"greedy"` (e.g., ~10x faster for $n = 200, p = 10, B = 10$) while achieving a better within-block distance objective (e.g., ~4% lower).
- `"greedy"` Greedy nearest-neighbour matching via `blockTools::block`. Requires the **blockTools** package. Fast even for large n.
- `"ompr"` Exact mixed-integer programme solved with GLPK via **ompr**. Globally optimal but scales as $O(n^2 B)$ in variables and is only practical for small n.

`dist` Distance specification used only when `method = "ompr"`. Either a function or one of `"euclidean"`, `"sum_abs_diff"`, or `"mahal"`. Default is `"mahal"`.

`roi_solver` MILP backend used only when `method = "ompr"`. A closed set `c("glpk", "gurobi", "cplex")`, default `"glpk"`. See the class documentation's "Solver backend" and wiring-guide sections for the Gurobi/CPLEX setup steps.

`response_type` The response type for the design.

`prob_T` Treatment assignment probability within each block.

`include_is_missing_as_a_new_feature` Whether to include missingness indicators.

`n` Planned sample size.

`verbose` Whether to print progress messages.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new `DesignFixedOptimalBlocks` object.

`DesignFixedOptimalBlocks$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DesignFixedOptimalBlocks$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Higgins, M. J., Sävje, F., and Sekhon, J. S. (2016). "Improving massive experiments with threshold blocking." *Proceedings of the National Academy of Sciences*, 113(27), 7369-7376, doi:10.1073/pnas.1510504113, for optimal/near-optimal covariate-based blocking prior to randomization. See also randomized block design for orientation and Mahalanobis distance for the default `"ompr"` distance.

Examples

```
des = DesignFixedOptimalBlocks$new(n = 9, response_type = 'continuous')
des$add_all_subjects_to_experiment(data.frame(x = rnorm(9)))
des$assign_w_to_all_subjects()
```

`DesignFixedRerandomization`

A Fixed Rerandomization Design (Rejection-Sampled on Covariate Balance)

Description

A fixed-sample-size `DesignFixed` implementing rerandomization (Morgan and Rubin, 2012): candidate allocations w are drawn from the design's base randomization law (balanced complete randomization when `prob_T = 0.5`, i.i.d. $\text{Bernoulli}(prob_T)$ draws otherwise — see `generate_one_rerandomized_w()`) and only **accepted** if a covariate imbalance criterion $M(w)$ between the treated and control groups falls below a threshold `a` (`obj_val_cutoff`), i.e. the accepted allocations are drawn from the base randomization distribution truncated to $\{w : M(w) \leq a\}$. Unlike `DesignFixedGreedy`/`DesignFixedGreedyDOptimal`, which *search* for a single well-balanced allocation, rerandomization instead *filters* the design's own randomization distribution, which is what makes it directly compatible with Fisherian/randomization-based inference (the accepted allocations remain a well-defined — if truncated — randomization distribution, valid for randomization tests/CIs restricted to that truncated support).

Two mutually exclusive acceptance modes are supported (specifying both errors): `obj_val_cutoff` accepts/rejects each draw against a fixed threshold `a`; `prop_acceptable` instead draws $r/prop_acceptable$ candidates and keeps the r with the *lowest* $M(w)$ (an empirical top-quantile acceptance region, equivalent in the large-draw limit to an implied cutoff at the `prop_acceptable` quantile of $M(w)$'s distribution under the base randomization law).

Objective $M(w)$. `objective = "mahal_dist"` (default) uses the squared Mahalanobis distance between treated and control covariate means, $M(w) = (\bar{X}_T - \bar{X}_C)^\top S^{-1}(\bar{X}_T - \bar{X}_C)$, where S is the sample covariance of all covariates (ridge-regularized by $10^{-6}I$ if $|\det S| < 10^{-10}$), computed once and cached in `private$S_inv`. `objective = "abs_sum_diff"` instead uses the sum of absolute mean differences, $M(w) = \sum_j |\bar{X}_{T,j} - \bar{X}_{C,j}|$, with no correlation adjustment. Only these two objectives are supported; any other value errors at draw time.

Fast path (native C++). When `prob_T = 0.5` and n is even, candidate generation and filtering run via a parallel C++ rejection sampler (`rerandomization_search_cpp()`), which internally works with a rescaled objective f_{cpp}: $f_{\text{cpp}} = M(w)/4$ for `"mahal_dist"` and $f_{\text{cpp}} = M(w)/2$ (on a GED-standardized scale) for `"abs_sum_diff"`; the user-facing `obj_val_cutoff` is converted to

this internal scale before being passed to C++, so the accepted-allocation semantics are unaffected, but this rescaling is a backend implementation detail worth knowing when comparing C++-path and pure-R-path acceptance rates for the same nominal cutoff. The sampler draws up to `max(r * 1000, 100000)` candidates internally; if fewer than `r` allocations are accepted within that budget (an overly tight cutoff), this errors naming how many were actually found – loosen `obj_val_cutoff` or use `prop_acceptable` instead. (Earlier versions silently recycled the accepted set to pad out to `r`, duplicating some draws; fixed, since that meant some "independent" replicates were literal duplicates of an accepted allocation.)

Seed reproducibility and multi-core parallelism. The C++ fast path derives each candidate's local RNG stream from its logical draw index, rather than from the OpenMP worker evaluating it. It evaluates deterministic, contiguous candidate blocks in fixed-size waves and returns accepted allocations in draw-index order. Consequently, a fixed `seed` returns the same allocations at every configured core count; adding cores changes only which worker evaluates a candidate. The package default is one core, but this reproducibility guarantee also holds after `set_num_cores` or when using a fork cluster/mirai daemons.

`prop_acceptable` **path.** Uses `complete_randomization_forced_balanced_cpp()` (balanced case) or `complete_randomization_imbalanced_cpp()` (`prob_T != 0.5`) to draw $n_{\text{draw}} = \text{round}(r/prop_acceptable)$ candidate allocations in one batched call, computes $M(w)$ for all of them via `compute_objective_vals_cpp()`, and keeps the r with smallest $M(w)$.

Pure-R fallback (unbalanced or odd n, `obj_val_cutoff` mode only). Draws one candidate at a time via `generate_one_rerandomized_w()` in an unbounded `repeat` loop that accepts the first candidate with $M(w) \leq a$. **Unlike the C++ fast path, this fallback has no draw-count safety limit**: if `obj_val_cutoff` is set tight enough that acceptance probability under the base randomization law is extremely small for this n/ covariate structure, this loop can run for a very long time (in principle indefinitely) before finding an acceptable draw.

Super classes

`Design` -> `DesignFixed` -> `DesignFixedRerandomization`

Methods

Public methods:

- `DesignFixedRerandomization$new()`
- `DesignFixedRerandomization$clone()`

`DesignFixedRerandomization$new()`**:** Initialize a rerandomization fixed experimental design. Exactly one of `obj_val_cutoff`/`prop_acceptable` may be specified (or neither, which accepts every candidate, i.e. no filtering); supplying both raises an error. See class documentation for the exact acceptance semantics of each mode and the covariate-imbalance objective.

Usage:

```
DesignFixedRerandomization$new(
  response_type,
  prob_T = 0.5,
  obj_val_cutoff = NULL,
  prop_acceptable = NULL,
  objective = "mahal_dist",
```

```
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` The data type of response values.

`prob_T` The probability of the treatment assignment.

`obj_val_cutoff` The maximum allowable objective value a; a candidate allocation is accepted iff $M(w) \leq a$. Cannot be specified together with prop_acceptable.

`prop_acceptable` The proportion of randomizations to accept (draws r/prop_acceptable total, returns r lowest). Cannot be specified together with obj_val_cutoff.

`objective` The covariate-imbalance objective $M(w)$ to filter on: either `"mahal_dist"` (default, squared Mahalanobis distance) or `"abs_sum_diff"` (sum of absolute mean differences); see class documentation for the exact formulas.

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` Flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new ‘DesignFixedRerandomization‘ object

`DesignFixedRerandomization$clone():` The objects of this class are cloneable with this method.

Usage:

```
DesignFixedRerandomization$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Morgan, K. L., and Rubin, D. B. (2012). "Rerandomization to improve covariate balance in experiments." *The Annals of Statistics*, 40(2), 1263-1282, doi:10.1214/12AOS1008, for the rerandomization framework and its randomization-inference validity. See also Mahalanobis distance for the default objective.

Examples

```
des = DesignFixedRerandomization$new(n = 10, response_type = 'continuous')
```

DesignSeqOneByOneAtkinson

Atkinson's (1982) Covariate-Adjusted Biased Coin Sequential Design

Description

A DesignSeqOneByOne that assigns each newly arriving subject's treatment via Atkinson's (1982) D_A-optimum biased coin: a coin whose treatment probability is skewed away from prob_T toward whichever assignment would most improve the current design's efficiency for estimating the treatment effect D_A-optimally, given the covariates observed so far. Compared to a fixed-probability coin (e.g. DesignSeqOneByOneBernoulli), this improves covariate balance/estimation efficiency online without ever fully determinizing the assignment (the coin is always strictly between 0 and 1, so randomization-based inference remains valid), at the cost of requiring a numerically well-conditioned design matrix to compute the bias.

Assignment rule. For subject t, let $Z_{t-1} = [w_{1:t-1}, 1, X_{1:t-1}]$ be the (treatment, intercept, covariates) design matrix accumulated from the first $t-1$ subjects, and let $M = (t-1)(Z_{t-1}^\top Z_{t-1})^{-1}$. Writing x_t for the new subject's covariate vector (with a leading 1 for the intercept) and $A = M_{[1,2:]} \cdot x_t$ (the treatment row of M, projected onto x_t), the treatment probability is

$$\pi_t = \frac{\left(M_{11}/A + 1\right)^2}{\left(M_{11}/A + 1\right)^2 + 1},$$

clamped to $[0, 1]$, and subject t is assigned to treatment with probability π_t. This is Atkinson's biased-coin formula for D_A-optimal sequential design: the coin biases toward the assignment that would most reduce the variance of the treatment-effect estimate under the linear model implied by Z_t, converging toward more extreme (but never fully deterministic) probabilities as the current covariate imbalance grows in directions that matter for that estimate.

Fallback to a fair(-ish) coin. For the first ncol(private$Xraw) + 3 subjects (too few observations for $Z_{t-1}^\top Z_{t-1}$ to be reliably invertible), and whenever the C++ computation encounters a non-invertible design matrix, a non-finite bias term, or any other numerical failure (caught via tryCatch()), assignment falls back to an unbiased $\mathrm{Bernoulli}(prob_T)$ draw instead of Atkinson's rule.

Reproducibility. The per-subject C++ draw (atkinson_assign_weight_cpp()) seeds its own generator from R's RNG stream per call, so seed governs reproducibility of the resulting assignment sequence in the usual way.

Super classes

Design -> DesignSeqOneByOne -> DesignSeqOneByOneAtkinson

Methods

Public methods:

- DesignSeqOneByOneAtkinson$new()
- DesignSeqOneByOneAtkinson$assign_wt()

- DesignSeqOneByOneAtkinson$clone()

`DesignSeqOneByOneAtkinson$new()`: Initialize an Atkinson (1982) biased-coin sequential experimental design (see class documentation for the assignment rule).

Usage:

```
DesignSeqOneByOneAtkinson$new(
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` The data type of response values.

`prob_T` The nominal probability of treatment assignment; used as the fallback coin probability early in the trial and whenever Atkinson's rule cannot be computed (see class documentation), and as the reference probability the biased coin is skewed away from otherwise.

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` A flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new 'DesignSeqOneByOneAtkinson' object

`DesignSeqOneByOneAtkinson$assign_wt()`: Draw the next subject's treatment assignment via Atkinson's (1982) D_A-optimum biased coin (see class documentation for the exact probability formula), falling back to an unbiased $\mathrm{Bernoulli}(prob_T)$ draw early in the trial or on numerical failure of the biased-coin computation.

Usage:

```
DesignSeqOneByOneAtkinson$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOneAtkinson$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOneAtkinson$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Atkinson, A. C. (1982). "Optimum biased coin designs for sequential clinical trials with prognostic factors." *Biometrika*, 69(1), 61-67, doi:10.1093/biomet/69.1.61. See also randomized experiment for orientation on biased-coin sequential designs.

Examples

```
seq_des = DesignSeqOneByOneAtkinson$new(n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
```

DesignSeqOneByOneBernoulli

A Sequential Bernoulli (Independent-Coin-Flip) Randomized Design

---

Description

A DesignSeqOneByOne in which each arriving subject's treatment assignment is drawn independently as $w_t \overset{iid}{\sim} \text{Bernoulli}(prob_T)$, with no dependence on covariates or on prior assignments — the direct sequential-enrollment analog of DesignFixedBernoulli. As in the fixed-sample version, the realized number of treated subjects after t arrivals is random ($\text{Binomial}(t, prob_T)$), in contrast to sequential designs that actively balance assignment counts or covariates (e.g. DesignSeqOneByOneAtkinson).

Nonparametric bootstrap

The ordinary row bootstrap for this design is supported under a prespecified fixed sample size: choose n before enrollment and stop after exactly n subjects, without using interim outcomes to decide when to stop. EDI does not implement sequential monitoring or check that this stopping condition was followed. In particular, n = NULL does not satisfy the documented fixed-sample justification, even though the bootstrap method is not blocked at runtime. The usual assumptions of iid subjects and potential outcomes, and a regular estimator, also apply.

Super classes

Design -> DesignSeqOneByOne -> DesignSeqOneByOneBernoulli

Methods

Public methods:

- DesignSeqOneByOneBernoulli$is_a_bernoulli_capable()
- DesignSeqOneByOneBernoulli$new()
- DesignSeqOneByOneBernoulli$assign_wt()
- DesignSeqOneByOneBernoulli$clone()

DesignSeqOneByOneBernoulli$is_a_bernoulli_capable(): Characterization: this design draws each subject's treatment assignment as an independent $\text{Bernoulli}(prob_T)$ coin flip (see class documentation), so it is Bernoulli-capable by construction.

Usage:

```
DesignSeqOneByOneBernoulli$is_a_bernoulli_capable()
```

Returns: Always TRUE for this class.

DesignSeqOneByOneBernoulli$new(): Initialize a Bernoulli (independent-coin-flip) sequential experimental design.

Usage:

```
DesignSeqOneByOneBernoulli$new(
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` The data type of response values which must be one of the following: "continuous", "incidence", "proportion", "count", "survival", "ordinal".

`prob_T` The probability of the treatment assignment. This defaults to `0.5`.

`include_is_missing_as_a_new_feature` If missing data is present in a variable, should we include another dummy variable for its missingness? The default is `TRUE`.

`n` The prespecified sample size for fixed-sample inference. Default is `NULL`; the nonparametric bootstrap justification above requires a fixed `n`.

`verbose` A flag indicating whether messages should be displayed.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new ‘DesignSeqOneByOneBernoulli‘ object

`DesignSeqOneByOneBernoulli$assign_wt():` Draw the next subject's treatment assignment as a single independent $\mathrm{Bernoulli}(prob_T)$ coin flip (see class documentation); does not consult covariates or prior assignments.

Usage:

```
DesignSeqOneByOneBernoulli$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOneBernoulli$clone():` The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOneBernoulli$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
```

DesignSeqOneByOneEfron

Efron's (1971) Biased Coin Sequential Design

Description

A DesignSeqOneByOne implementing Efron's (1971) biased coin: no covariates are used, only the running counts of treated (n_T) and control (n_C) subjects assigned so far. If the counts are currently equal, the next subject is assigned by a fair $\text{Bernoulli}(0.5)$ coin; otherwise, the next subject is assigned to the currently **under-represented** group with probability `weighted_coin_prob` (> 0.5, e.g. the classical $2/3$) and to the over-represented group with probability `1 - weighted_coin_prob`. This keeps the running treatment/control counts close to balanced throughout enrollment (unlike DesignSeqOneByOneBernoulli, whose running counts can drift arbitrarily far from balanced) while remaining strictly randomized at every step (the coin is always strictly between `1 - weighted_coin_prob` and `weighted_coin_prob`, never fully deterministic), unlike a purely deterministic alternating allocation. This is a count-balancing design only — it does not use covariates at all, in contrast to DesignSeqOneByOneAtkinson/ DesignSeqOneByOneKK21, which bias the coin toward covariate balance rather than (or in addition to) count balance.

Super classes

Design -> DesignSeqOneByOne -> DesignSeqOneByOneEfron

Methods

Public methods:

- DesignSeqOneByOneEfron$new()
- DesignSeqOneByOneEfron$assign_wt()
- DesignSeqOneByOneEfron$clone()

`DesignSeqOneByOneEfron$new()`: Initialize an Efron (1971) biased coin sequential experimental design (see class documentation for the exact assignment rule).

Usage:

```
DesignSeqOneByOneEfron$new(
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  weighted_coin_prob = 2/3,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` "continuous", "incidence", "proportion", "count", "survival", or "ordinal".

`prob_T` Nominal probability of treatment assignment; used only as the fair-coin probability when the running treated/control counts are exactly equal (see `assign_wt()`).

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` A flag for verbosity.

`weighted_coin_prob` The probability (> 0.5) of assigning the next subject to whichever of treatment/control currently has fewer subjects, when the running counts are unequal. Default $2/3$, the value from Efron (1971).

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new ‘DesignSeqOneByOneEfron‘ object

`DesignSeqOneByOneEfron$assign_wt()`: Draw the next subject's treatment assignment via Efron's (1971) biased coin (see class documentation): a fair coin if the running treated/control counts are equal, otherwise a coin biased toward the currently under-represented group at probability `weighted_coin_prob`.

Usage:

```
DesignSeqOneByOneEfron$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOneEfron$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOneEfron$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Efron, B. (1971). "Forcing a sequential experiment to be balanced." *Biometrika*, 58(3), 403-417, doi:10.1093/biomet/58.3.403. See also randomized experiment for orientation on biased-coin sequential designs.

Examples

```
seq_des = DesignSeqOneByOneEfron$new(n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
```

DesignSeqOneByOneiBCRD

A Sequential Design Guaranteeing Exact Terminal Balance (Random Allocation Rule)

Description

A DesignSeqOneByOne implementing the "random allocation rule" (a sequential realization of complete randomization): treatment is assigned to each arriving subject with probability equal to the fraction of **remaining treatment slots** among all remaining slots, $\Pr(w_t = 1) = n_{T,\text{rem}}/(n_{T,\text{rem}} + n_{C,\text{rem}})$, where $n_{T,\text{rem}} = \text{round}(n \cdot prob_T) - n_T$ and $n_{C,\text{rem}} = (n - \text{round}(n \cdot prob_T)) - n_C$ are the treatment/control slots not yet used, given the running counts n_T, n_C. This guarantees the realized sequence, once all n subjects have arrived, has *exactly* $\text{round}(n \cdot prob_T)$ treated subjects — the same terminal allocation-count guarantee as DesignFixediBCRD's complete randomization, but realized online as subjects arrive one at a time rather than all at once, and with every prefix of the sequence itself drawn from the correct conditional (hypergeometric) distribution given the slots used so far. If a slot type is exhausted ($n_{T,\text{rem}} \leq 0$ or $n_{C,\text{rem}} \leq 0$), the remaining subjects are deterministically assigned to whichever type still has open slots.

No target n: falls back to Bernoulli. If n was not supplied at construction (private$n is NULL), there is no terminal target to balance toward, so assign_wt() falls back to an unbiased $\text{Bernoulli}(prob_T)$ draw for every subject instead (equivalent to DesignSeqOneByOneBernoulli).

Single implicit block. add_one_subject_to_experiment_and_assign() overrides the inherited method only to additionally set private$m to a constant vector of 1s (a single block containing every subject enrolled so far) after each assignment, mirroring the fixed-sample DesignFixediBCRD's single-block convention for shared blocking/matching machinery.

Super classes

Design -> DesignSeqOneByOne -> DesignSeqOneByOneiBCRD

Methods

Public methods:

- DesignSeqOneByOneiBCRD$new()
- DesignSeqOneByOneiBCRD$add_one_subject_to_experiment_and_assign()
- DesignSeqOneByOneiBCRD$assign_wt()
- DesignSeqOneByOneiBCRD$clone()

DesignSeqOneByOneiBCRD$new(): Initialize a sequential design targeting exact terminal treatment/control balance (see class documentation for the assignment rule and the no-fixed-n fallback).

Usage:

```
DesignSeqOneByOneiBCRD$new(
  response_type,
  prob_T = 0.5,
```

```
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` "continuous", "incidence", "proportion", "count", "survival", or "ordinal".

`prob_T` Target probability of treatment assignment; the terminal number of treated subjects is fixed at `round(n * prob_T)` when `n` is known (see class documentation).

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The planned (target) sample size; if `NULL`, there is no terminal balance target and assignment falls back to an unbiased Bernoulli coin (see class documentation).

`verbose` A flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new 'DesignSeqOneByOneiBCRD' object

`DesignSeqOneByOneiBCRD$add_one_subject_to_experiment_and_assign()`: Add one subject to the experiment and assign treatment via `assign_wt()` (delegating to the inherited `DesignSeqOneByOne$add_one_su` then set `private$m` to a single-block vector of 1s covering every subject enrolled so far (see class documentation), overwritten on every call rather than only once all subjects have arrived.

Usage:

```
DesignSeqOneByOneiBCRD$add_one_subject_to_experiment_and_assign(x_new)
```

Arguments:

`x_new` A data frame with one row representing the new subject's covariates.

Returns: The treatment assignment (0 or 1) for the newly added subject.

`DesignSeqOneByOneiBCRD$assign_wt()`: Draw the next subject's treatment assignment via the random allocation rule (see class documentation): with probability equal to the fraction of remaining treatment slots among all remaining slots, or a deterministic assignment if one slot type is exhausted; falls back to an unbiased Bernoulli coin if no fixed `n` was supplied.

Usage:

```
DesignSeqOneByOneiBCRD$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOneiBCRD$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOneiBCRD$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Rosenberger, W. F., and Lachin, J. M. (2016). *Randomization in Clinical Trials: Theory and Practice* (2nd ed.), Wiley, for the random allocation rule as a sequential implementation of complete randomization. See also DesignFixediBCRD for the fixed-sample (all-at-once) version of the same terminal randomization law.

Examples

```
seq_des = DesignSeqOneByOneiBCRD$new(n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
```

DesignSeqOneByOneKK14 *Kapelner and Krieger's (2014) Sequential "Matching-on-the-Fly" Design*

Description

A DesignSeqOneByOne that matches subjects to each other **as they arrive**, rather than requiring all subjects up front (as DesignFixedBinaryMatch does): each new subject is either matched to its nearest (Mahalanobis-distance) unmatched prior subject in the "reservoir" — if that distance is small enough to pass a statistical closeness test — and assigned the **opposite** treatment from its match, or, if no sufficiently close match exists (or matching hasn't started yet), randomized and added to the reservoir for future subjects to potentially match against.

Burn-in. For the first `t_0_pct * n` subjects (default 35%), or whenever no covariates are yet available, subjects are simply randomized ($\text{Bernoulli}(prob_T)$) and placed in the reservoir (`private$m` set to 0 for that subject) — matching does not begin until enough subjects have accumulated to estimate a stable covariate covariance structure.

Matching test. Once past burn-in, for new subject t with covariate vector x_t, the squared Mahalanobis distance (via `compute_proportional_mahal_distances_cpp()`, using the sample covariance of all prior subjects' covariates, ridge-regularized by `.Machine$double.eps`) to every subject currently in the reservoir is computed, and the closest one is a candidate match. The match is **accepted** only if that squared distance falls below a threshold T^2_{cutoff} derived from an F critical value,

$$T^2_{\text{cutoff}} = \frac{p(n-1)}{n-p}\, F_{p,\, t-p}(\lambda),$$

where p is the rank of the covariate matrix so far, n is the design's target (planned) sample size, t is the number of subjects enrolled so far, and λ (`lambda`, default 0.1) is the F-distribution quantile level — i.e. this is a Hotelling's T^2-type test of whether the candidate pair's covariate difference is small enough to plausibly be exchangeable "noise" rather than a meaningful covariate mismatch; `lambda` controls how strict that test is (smaller `lambda` accepts fewer, closer matches). If accepted, both subjects are recorded as a new match (`private$m`), and the new subject receives the **opposite** treatment of its match, guaranteeing exactly one treated and one control per matched pair — the same guarantee DesignFixedBinaryMatch provides, but formed incrementally rather than all at once. If rejected (or the reservoir is empty), the subject is randomized and added to the reservoir instead.

Lifecycle note. The `morrison` and `p` constructor arguments are currently recorded on the object but not consulted anywhere in the matching or assignment logic in this version of the class; treat them as reserved for future use rather than as active configuration.

Super classes

`Design` -> `DesignSeqOneByOne` -> `DesignSeqOneByOneKK14`

Methods

Public methods:

- `DesignSeqOneByOneKK14$is_a_kk_matching_capable()`
- `DesignSeqOneByOneKK14$new()`
- `DesignSeqOneByOneKK14$assign_wt()`
- `DesignSeqOneByOneKK14$clone()`

`DesignSeqOneByOneKK14$is_a_kk_matching_capable()`**:** Characterization: this design matches subjects to each other incrementally as they arrive (see class documentation), so it is KK matching-on-the-fly-capable by construction.

Usage:

```
DesignSeqOneByOneKK14$is_a_kk_matching_capable()
```

Returns: Always `TRUE` for this class.

`DesignSeqOneByOneKK14$new()`**:** Initialize a KK14 sequential matching-on-the-fly experimental design (see class documentation for the burn-in and matching-test rules).

Usage:

```
DesignSeqOneByOneKK14$new(
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  lambda = NULL,
  t_0_pct = NULL,
  morrison = FALSE,
  p = NULL,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`response_type` "continuous", "incidence", "proportion", "count", "survival", or "ordinal".

`prob_T` Probability of treatment assignment used for burn-in/ unmatched (reservoir) subjects; matched subjects instead always receive the opposite assignment of their match (see class documentation).

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` A flag for verbosity.

`lambda` The F-distribution quantile level controlling how strict the matching-acceptance test is (default 0.1; smaller values accept fewer, closer matches). See class documentation for the exact threshold formula.

`t_0_pct` The fraction of `n` subjects to randomize into the reservoir before matching begins (default 0.35).

`morrison` Currently unused by this class's matching/assignment logic; reserved for future use.

`p` Currently unused by this class's matching/assignment logic; reserved for future use.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new 'DesignSeqOneByOneKK14' object

`DesignSeqOneByOneKK14$assign_wt():` Draw the next subject's treatment assignment via KK14 matching-on-the-fly (see class documentation): during burn-in, or if no sufficiently close reservoir match exists, randomize and add the subject to the reservoir; otherwise match to the nearest reservoir subject and assign the opposite treatment.

Usage:

```
DesignSeqOneByOneKK14$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOneKK14$clone():` The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOneKK14$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A., and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388, doi:10.1111/biom.12148. See also Hotelling's T-squared distribution for the matching-test statistic, and Mahalanobis distance.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
```

DesignSeqOneByOneKK21 *Kapelner and Krieger's (2021) Outcome-Weighted Sequential Matching-on-the-Fly Design*

Description

A DesignSeqOneByOneKK14 extension that replaces KK14's unweighted Mahalanobis matching distance with a **response-weighted** squared distance: at each assignment, a per-covariate weight vector is re-estimated from the responses observed so far (weighting each covariate by the estimated strength of its association with the response, e.g. an absolute standardized regression coefficient), and the new subject is matched to its nearest reservoir subject under that weighted distance rather than the raw (unweighted) Mahalanobis distance KK14 uses. Weighting the match by outcome association means matching effort is spent preferentially on *prognostic* covariates (those that actually explain outcome variance) rather than equally on every covariate, which is intended to improve estimator efficiency beyond what outcome-agnostic matching achieves. `morrison = TRUE` additionally switches to Morrison and Owen's (2025) alternative calibration of the matching-acceptance threshold (differing in the fixed- vs. variable-n settings) and removes KK14's burn-in wait before matching begins.

Weight estimation. `compute_weights()` dispatches on `response_type` to a corresponding `kk21_*_weights_cpp()` backend that fits a per-covariate simple regression of the response on that covariate (given all responses observed so far) and returns the absolute t-statistic (coefficient over its standard error) as that covariate's weight: OLS for `"continuous"`, logistic for `"incidence"`, negative-binomial (or, if `count_use_speedup = TRUE`, OLS on `log(y + 1)`) for `"count"`, beta regression (or OLS on the logit scale if `proportion_use_speedup = TRUE`) for `"proportion"`, Weibull/lognormal/ log-logistic AFT (or OLS on `log(y)` if `survival_use_speedup_for_no_censoring = TRUE` and there is no censoring yet) for `"survival"`, and proportional-odds (or OLS on the numeric-coerced level if `ordinal_use_speedup = TRUE`) for `"ordinal"`. Weights are normalized to sum to 1 and cached per-iteration in `private$iteration_weights` (retrievable via `get_iteration_weights()`); the `*_use_speedup` flags trade weight accuracy for speed by substituting a fast continuous-regression proxy for the response-type-appropriate GLM/AFT/ordinal fit on every single assignment call.

How much weight accuracy do the speedups cost? A weight-level simulation (October 2026; eight standard-normal covariates with decaying true coefficients, n from 15 to 250, count, proportion and ordinal responses) found the loss to be negligible. Across all three response types the fast OLS-on-transform weights were as close to the GLM's own large-sample target (its normalized weights at $n = 20000$) as the GLM's finite-sample weights themselves were: the total-variation distance between the normalized fast and GLM weight vectors was 0.02 to 0.16, versus 0.1 to 0.43 for each method's own sampling noise, and the two large-sample weight vectors differed by under 2% of total weight mass. The two weight vectors selected the same nearest reservoir match 75 to 98% of the time, and the disagreements were near-ties between two reservoir candidates. Running the full design end to end ($n = 200$, 150 replications per setting, with `DesignSeqOneByOneKK14` as the unweighted reference) confirmed this downstream: the share of the true linear predictor's variance remaining inside matched pairs was indistinguishable between the fast and GLM weights for count (0.192 vs 0.191), sparse count with about 70% zeros (0.187 vs 0.188), proportion (0.147 vs 0.148) and four-level ordinal (0.181 vs 0.183) responses, with Monte Carlo standard errors near 0.004 and the same number of matched pairs (about 90 of 200) under both. That within-pair imbalance is the design-side term in the matched-pairs estimator's variance, so it is the channel through which

the weights affect power; power itself was not simulated, because a gap this small (under 2% of a term that is itself only part of the total variance) is far below what a direct power simulation of feasible size could resolve. The approximation error is therefore swamped by the estimation noise both methods share, which is why the `*_use_speedup` flags default to `TRUE`.

Weighted matching test. The weighted squared distance from the new subject to every reservoir subject is computed (`compute_weighted_sqd_distances_cpp()`), and the match is accepted only if the minimum weighted distance falls below the `private$compute_lambda()` quantile of a *bootstrapped* reference distribution of weighted pairwise distances among subjects enrolled so far (`compute_bootstrapped_weighted_sqd_distances_cpp()`, `num_boot` resamples) — a nonparametric, simulation-based acceptance threshold, in contrast to KK14's closed-form F-distribution threshold. As in KK14, an accepted match receives the opposite treatment of its match; a rejected (or empty-reservoir) draw is randomized and added to the reservoir.

Fallback to KK14. Before enough responses have accumulated to fit the weight-estimation regressions reliably (fewer than `2 * (ncol(X) + 2)` non-missing responses — two observations per regression parameter, at minimum `ncol(X) + 2`), `assign_wt()` falls back to the inherited (unweighted) KK14 assignment rule entirely, via `super$assign_wt()`.

Super classes

`Design` -> `DesignSeqOneByOne` -> `DesignSeqOneByOneKK14` -> `DesignSeqOneByOneKK21`

Methods

Public methods:

- `DesignSeqOneByOneKK21$new()`
- `DesignSeqOneByOneKK21$get_iteration_weights()`
- `DesignSeqOneByOneKK21$get_covariate_weights()`
- `DesignSeqOneByOneKK21$assign_wt()`
- `DesignSeqOneByOneKK21$clone()`

`DesignSeqOneByOneKK21$new()`**:** Initialize a matching-on-the-fly sequential experimental design which matches based on Kapelner and Krieger (2021) with option to use matching parameters of Morrison and Owen (2025)

Usage:

```
DesignSeqOneByOneKK21$new(
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  lambda = NULL,
  t_0_pct = NULL,
  morrison = FALSE,
  p = NULL,
  num_boot = NULL,
  count_use_speedup = TRUE,
  proportion_use_speedup = TRUE,
```

```
  survival_use_speedup_for_no_censoring = TRUE,
  ordinal_use_speedup = TRUE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL,
  ...
)
```

Arguments:

`response_type` The data type of response values which must be one of the following: "continuous", "incidence", "proportion", "count", "survival". This package will enforce that all added responses via the `add_one_subject_response` method will be of the appropriate type.

`prob_T` The probability of the treatment assignment. This defaults to `0.5`.

`include_is_missing_as_a_new_feature` If missing data is present in a variable, should we include another dummy variable for its missingness in addition to imputing its value? If the feature is type factor, instead of creating a new column, we allow missingness to be its own level. The default is `TRUE`.

`n` The sample size (if fixed). Default is `NULL` for not fixed.

`verbose` A flag indicating whether messages should be displayed to the user. Default is `FALSE`.

`lambda` The quantile cutoff of the subject distance distribution for determining matches. If unspecified and `morrison = FALSE`, default is 10%.

`t_0_pct` The percentage of total sample size n where matching begins. If unspecified and `morrison = FALSE`, default is 35%.

`morrison` Default is `FALSE` which implies matching via the KK14 algorithm using `lambda` and `t_0_pct` matching. If `TRUE`, we use Morrison and Owen (2025)'s formula for `lambda` which differs in the fixed n versus variable n settings and matching begins immediately with no wait for a certain reservoir size like in KK14.

`p` The number of covariate features. Must be specified when `morrison = TRUE` otherwise do not specify this argument.

`num_boot` the number of bootstrap samples taken to approximate the subject-distance distribution. Default is 500.

`count_use_speedup` Should we speed up the estimation of the weights in the response = count case via a continuous regression on log(y + 1). instead of a negative binomial regression each time? This is at the expense of the weights being less accurate. Default is `TRUE`. See the *How much weight accuracy do the speedups cost?* paragraph of the Details section for simulation evidence that the accuracy lost is negligible.

`proportion_use_speedup` Should we speed up the estimation of the weights in the response = proportion case via a continuous regression on log(y / (1 - y)) instead of a beta regression each time? This is at the expense of the weights being less accurate. Default is `TRUE`. See the *How much weight accuracy do the speedups cost?* paragraph of the Details section for simulation evidence that the accuracy lost is negligible.

`survival_use_speedup_for_no_censoring` Should we speed up the estimation of the weights in the response = survival case via a continuous regression on log(y) instead of a Weibull AFT regression each time, but only when there is no censoring in the data collected so far? This is at the expense of the weights being less accurate when censoring is present. Default is `TRUE`.

`ordinal_use_speedup` Should we speed up the estimation of the weights in the response = ordinal case via a continuous regression on the ordinal levels coerced to numeric. instead of a proportional odds model each time? This is at the expense of the weights being less accurate. Default is `TRUE`. See the *How much weight accuracy do the speedups cost?* paragraph of the Details section for simulation evidence that the accuracy lost is negligible.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

`...` Extra arguments passed to the `DesignSeqOneByOneKK14` superclass.

Returns: A new 'DesignSeqOneByOneKK21' object

Examples:

```
seq_des = DesignSeqOneByOneKK21$new(n = 6, response_type = "continuous")
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x = rnorm(1)))
```

`DesignSeqOneByOneKK21$get_iteration_weights()`: Retrieve the full history of normalized covariate weight vectors computed by `compute_weights()` across every assignment call so far (see class documentation), keyed by subject index `t`, for inspecting how the outcome-informed weighting evolved as data accrued.

Usage:

```
DesignSeqOneByOneKK21$get_iteration_weights()
```

Returns: A list of numeric weight vectors (one per assignment call at which weights were computed), each summing to 1 and named by covariate.

`DesignSeqOneByOneKK21$get_covariate_weights()`: Retrieve the normalized covariate weight vector from the most recent assignment call (see class documentation for how weights are estimated).

Usage:

```
DesignSeqOneByOneKK21$get_covariate_weights()
```

Returns: A numeric vector of weights, one per covariate, summing to 1 and named by covariate; `NULL` if weights have not yet been computed (e.g. still in the KK14 fallback regime).

`DesignSeqOneByOneKK21$assign_wt()`: Draw the next subject's treatment assignment via the KK21 outcome-weighted matching-on-the-fly rule (see class documentation): falls back to unweighted KK14 matching if too few responses have accumulated to estimate covariate weights, otherwise re-estimates weights, matches to the nearest reservoir subject under the weighted distance if it clears the bootstrapped acceptance threshold (assigning the opposite treatment), or randomizes into the reservoir otherwise.

Usage:

```
DesignSeqOneByOneKK21$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOneKK21$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOneKK21$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A., and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388, doi:10.1111/biom.12148, for the base matching-on-the-fly algorithm this class extends with outcome-weighted distances (Kapelner and Krieger, 2021); see also Morrison, T., and Owen, A. B. (2025) for the alternative `morrison = TRUE` threshold calibration referenced by the `morrison` argument.

Examples

```
seq_des = DesignSeqOneByOneKK21$new(n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))

## ------------------------------------------------
## Method `DesignSeqOneByOneKK21$new()`
## ------------------------------------------------


seq_des = DesignSeqOneByOneKK21$new(n = 6, response_type = "continuous")
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x = rnorm(1)))
```

`DesignSeqOneByOneKK21stepwise`

Stepwise Variant of the KK21 Outcome-Weighted Sequential Matching Design

---

Description

A `DesignSeqOneByOneKK21` variant that computes its per-covariate matching weights via **forward stepwise selection** (`compute_weights_KK21stepwise()`) instead of KK21's independent marginal-association regressions: covariates are added to a growing "selected" set one at a time, at each step choosing whichever remaining covariate has the largest absolute association statistic *conditional on* (i.e. in a model that also includes) the covariates already selected and the treatment-assignment column, rather than each covariate's association with the response considered in isolation. This targets the case where covariates are mutually correlated: KK21's marginal weights can assign similar high weight to several collinear prognostic covariates (effectively double-counting the same information), whereas the stepwise conditional weights down-weight a covariate once its explanatory content is already captured by previously selected covariates.

Weight computation. For each response type, a family-appropriate model (OLS/logistic/negative-binomial/beta/AFT survival/proportional-odds, matching the same response-type dispatch and `*_use_speedup` fast-path conventions as `DesignSeqOneByOneKK21`) is repeatedly refit, each time regressing the response on one candidate remaining covariate plus all previously selected covariates plus the treatment column `ws`; the candidate with the largest absolute association statistic is selected next and assigned that statistic as its weight, then removed from the candidate pool, and the process repeats until every covariate has been assigned a weight (an $O(p^2)$ number of model fits per assignment call, for p covariates). If a candidate's model fit fails to converge (e.g. perfect separation or rank

deficiency) partway through, the remaining not-yet-selected covariates' weights are left NA internally and then replaced with 0 (excluding them from the weighted matching distance) rather than propagating the failure.

Randomization draws. Because the stepwise models include the treatment column ws, the covariate weights (and therefore which subjects end up paired) depend in principle on the realized assignment path, even under the sharp null where the responses are fixed. Re-randomization for randomization tests and confidence intervals is therefore controlled by condition_on_ws_for_rand_tests:

- TRUE (default): condition on the realized reservoir/pair structure exactly as DesignSeqOneByOneKK14 (and DesignSeqOneByOneKK21) do: each matched pair's assignments are flipped with probability 1/2 and reservoir subjects are redrawn independently with probability prob_T. This is fast but **approximate**, not exact: since w is randomized it should be (nearly) independent of the covariates, so partialling it out barely changes the conditional association statistics and the pair structure is almost always the same under a different assignment path. In simulation the resulting inference is very close to the exact replay.
- FALSE: exact. Every randomization draw replays the entire sequential design from subject 1, refitting the stepwise weights at every step using only the responses available at that point, so pairs may form differently in each draw. This costs on the order of $r \times n$ stepwise weight fits per set of r draws and is dramatically slower.

Everything else is inherited from KK21. Burn-in fallback to KK14, the bootstrapped acceptance-threshold test, matched-pair assignment, and the morrison/lambda/t_0_pct matching-schedule options are all unchanged from DesignSeqOneByOneKK21; only *how* the covariate weight vector is computed differs.

Super classes

Design -> DesignSeqOneByOne -> DesignSeqOneByOneKK14 -> DesignSeqOneByOneKK21 -> DesignSeqOneByOneKK21ste

Methods

Public methods:

- DesignSeqOneByOneKK21stepwise$new()
- DesignSeqOneByOneKK21stepwise$clone()

DesignSeqOneByOneKK21stepwise$new(): Initialize a matching-on-the-fly sequential experimental design whose covariate matching weights are computed via forward stepwise selection (see class documentation), based on the stepwise version of Kapelner and Krieger (2021) with option to use matching parameters of Morrison and Owen (2025)

Usage:

```
DesignSeqOneByOneKK21stepwise$new(
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  lambda = NULL,
  t_0_pct = NULL,
```

```
  morrison = FALSE,
  p = NULL,
  num_boot = NULL,
  count_use_speedup = TRUE,
  proportion_use_speedup = TRUE,
  survival_use_speedup_for_no_censoring = TRUE,
  ordinal_use_speedup = TRUE,
  missingness_method = "impute",
  design_formula = ~.,
  condition_on_ws_for_rand_tests = TRUE,
  ...
)
```

Arguments:

`response_type` The data type of response values which must be one of the following: "continuous", "incidence", "proportion", "count", "survival". This package will enforce that all added responses via the `add_one_subject_response` method will be of the appropriate type.

`prob_T` The probability of the treatment assignment. This defaults to `0.5`.

`include_is_missing_as_a_new_feature` If missing data is present in a variable, should we include another dummy variable for its missingness in addition to imputing its value? If the feature is type factor, instead of creating a new column, we allow missingness to be its own level. The default is `TRUE`.

`n` The sample size (if fixed). Default is `NULL` for not fixed.

`verbose` A flag indicating whether messages should be displayed to the user. Default is `FALSE`.

`lambda` The quantile cutoff of the subject distance distribution for determining matches. If unspecified and `morrison = FALSE`, default is 10%.

`t_0_pct` The percentage of total sample size n where matching begins. If unspecified and `morrison = FALSE`, default is 35%.

`morrison` Default is `FALSE` which implies matching via the KK14 algorithm using `lambda` and `t_0_pct` matching. If `TRUE`, we use Morrison and Owen (2025)'s formula for `lambda` which differs in the fixed n versus variable n settings and matching begins immediately with no wait for a certain reservoir size like in KK14.

`p` The number of covariate features. Must be specified when `morrison = TRUE` otherwise do not specify this argument.

`num_boot` the number of bootstrap samples taken to approximate the subject-distance distribution. Default is 500.

`count_use_speedup` Should we speed up the estimation of the weights in the response = count case via a continuous regression on log(y + 1). instead of a negative binomial regression each time? This is at the expense of the weights being less accurate. Default is `TRUE`.

`proportion_use_speedup` Should we speed up the estimation of the weights in the response = proportion case via a continuous regression on log(y / (1 - y)) instead of a beta regression each time? This is at the expense of the weights being less accurate. Default is `TRUE`.

`survival_use_speedup_for_no_censoring` Should we speed up the estimation of the weights in the response = survival case via a continuous regression on log(y) instead of a Weibull AFT regression each time, but only when there is no censoring in the data collected so far? This is at the expense of the weights being less accurate when censoring is present. Default is `TRUE`.

`ordinal_use_speedup` Should we speed up the estimation of the weights in the response = ordinal case via a continuous regression on the ordinal levels coerced to numeric. instead of a proportional odds model each time? This is at the expense of the weights being less accurate. Default is `TRUE`.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`condition_on_ws_for_rand_tests` Should randomization draws (used by randomization tests and randomization confidence intervals) condition on the realized reservoir/matched-pair structure, as in KK14? If `TRUE` (default), each draw flips each matched pair and redraws the reservoir; this is fast but approximate, because the stepwise weights condition on the treatment column so the pairs could in principle differ under another assignment path (since `w` is randomized it should be independent of the covariates, and in simulation the approximation is very close). If `FALSE`, each draw replays the entire sequential design, refitting the stepwise weights at every step; this is exact but much slower. See the class documentation for details.

`...` Extra arguments passed to the `DesignSeqOneByOneKK21` superclass.

Returns: A new ‘DesignSeqOneByOneKK21stepwise‘ object

Examples:

```
seq_des = DesignSeqOneByOneKK21stepwise$new(n = 6, response_type = "continuous")
```

`DesignSeqOneByOneKK21stepwise$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOneKK21stepwise$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A., and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388, doi:10.1111/biom.12148, for the base matching-on-the-fly algorithm; the outcome-weighted extension follows Kapelner and Krieger (2021), with this class using a forward-stepwise (rather than marginal) weight-estimation scheme. See also Morrison, T., and Owen, A. B. (2025) for the alternative `morrison = TRUE` threshold calibration referenced by the `morrison` argument.

Examples

```
seq_des = DesignSeqOneByOneKK21stepwise$new(n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))

## ------------------------------------------------
## Method `DesignSeqOneByOneKK21stepwise$new()`
## ------------------------------------------------

seq_des = DesignSeqOneByOneKK21stepwise$new(n = 6, response_type = "continuous")
```

DesignSeqOneByOnePocockSimon

Pocock and Simon's (1975) Minimization Sequential Design

Description

A DesignSeqOneByOne implementing Pocock and Simon's minimization method: for each categorical covariate in strata_cols, the design tracks a running treated/control count per covariate *level* (private$counts), and for a new subject computes, for each candidate treatment arm $k \in \{0, 1\}$, a weighted total imbalance

$$G_k = \sum_j weights_j \cdot \mathrm{Var}\big(\text{counts at subject's level of covariate } j, \text{ after hypothetically assigning arm } k\big),$$

where the variance is taken across the two treatment arms' hypothetical counts at that covariate level (so G_k is large when arm k would leave the subject's covariate-level counts unbalanced, summed with weights across covariates). The subject is then assigned to whichever arm minimizes G_k with probability p_best (and to the other arm with probability 1 - p_best), or — if the two arms are exactly tied — via a plain $\mathrm{Bernoulli}(prob_T)$ draw. Unlike DesignSeqOneByOneAtkinson/DesignSeqOneByOneKK14, which use continuous covariate distances, minimization operates on categorical/discretized strata and balances marginal covariate-level counts directly rather than a multivariate distance or matched-pair structure.

Level bookkeeping. private$ensure_factor_metadata() maintains a mapping from each observed level of each strata_cols column to a row index in private$counts (an (total levels across all covariates) x 2 matrix of running treated/control counts), growing both the level map and counts as new levels are encountered; missing values are treated as their own level (distinct from a literal category named "NA").

Non-resampling bootstrap. draw_bootstrap_indices() always performs a plain i.i.d. nonparametric bootstrap over subjects (sample_int_replace_cpp()), since minimization's adaptive assignment process has no simple exchangeable resampling unit to preserve (each subject's assignment probability depends on the full sequence of covariate levels and assignments that preceded it).

Super classes

Design -> DesignSeqOneByOne -> DesignSeqOneByOnePocockSimon

Methods

Public methods:

- DesignSeqOneByOnePocockSimon$new()
- DesignSeqOneByOnePocockSimon$assign_wt()
- DesignSeqOneByOnePocockSimon$clone()

DesignSeqOneByOnePocockSimon$new(): Initialize a Pocock and Simon (1975) minimization sequential experimental design (see class documentation for the exact imbalance criterion and assignment rule).

Usage:

```
DesignSeqOneByOnePocockSimon$new(
  strata_cols,
  weights = NULL,
  p_best = 0.8,
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`strata_cols` The names of the covariates to be used for minimization. These must be factor or categorical variables.

`weights` A numeric vector of per-covariate weights $weights_j$ in the imbalance criterion G_k (see class documentation), one per entry of `strata_cols`, in the same order. Defaults to 1 for all (equal-weighted covariates).

`p_best` The probability of assigning the treatment arm that minimizes G_k (see class documentation); the complementary arm is assigned with probability `1 - p_best`. Defaults to 0.8 (an 80/20 biased coin favoring the balancing arm, rather than a fully deterministic minimization rule).

`response_type` The data type of response values.

`prob_T` The probability of the treatment assignment used only when the two arms' imbalance is exactly tied (see class documentation).

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` Flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new 'DesignSeqOneByOnePocockSimon' object

`DesignSeqOneByOnePocockSimon$assign_wt():` Draw the next subject's treatment assignment via Pocock and Simon minimization (see class documentation for the exact imbalance criterion G_k and the `p_best`/`prob_T` assignment rule), and update the running per-covariate-level treated/control counts in-place to reflect this assignment.

Usage:

```
DesignSeqOneByOnePocockSimon$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOnePocockSimon$clone():` The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOnePocockSimon$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Pocock, S. J., and Simon, R. (1975). "Sequential treatment assignment with balancing for prognostic factors in the controlled clinical trial." *Biometrics*, 31(1), 103-115, doi:10.2307/2529712. See also minimisation (clinical trials) for orientation.

Examples

```
seq_des = DesignSeqOneByOnePocockSimon$new(strata_cols = 'x1', n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = factor(1, levels=1:2)))
```

DesignSeqOneByOneRandomBlockSize

A Sequential Permuted-Block Design with Randomly Varying Block Sizes

---

Description

A `DesignSeqOneByOne` implementing permuted-block randomization with randomly varying block size: subjects are assigned from a queue of pre-shuffled treatment labels (a "block"), refilled with a fresh `sample`d block whenever it empties. Each new block's size is itself drawn uniformly at random from `block_sizes` (rather than being fixed), and each block internally contains exactly `round(block_size * prob_T)` treated and `block_size - round(block_size * prob_T)` control labels in random order. Randomizing the block size (rather than using a single fixed block length, as in classical permuted-block designs) is a standard clinical-trials safeguard against **selection bias**: with a fixed, known block size, unblinded staff could predict the last assignment(s) in a block from the ones already observed, whereas an unpredictable block size makes this much harder while still guaranteeing near-perfect treatment/control balance throughout enrollment (balance is exact at every block boundary and never worse than one full block's imbalance in between). If `strata_cols` is supplied, a separate independent sequence of blocks is maintained per stratum (one queue per distinct combination of `strata_cols` values), so balance holds within each stratum, not just overall.

Block-size / `prob_T` compatibility. Every entry of `block_sizes` must yield an integer number of treated subjects when multiplied by `prob_T` (checked at construction: `abs(bs * prob_T - round(bs * prob_T)) <= 1e-10` for every `bs`); a block size that would require a fractional number of treated subjects is rejected.

Per-stratum queues. `private$strata_states` is a hashed environment mapping each stratum key (or the literal key `"overall"` when `strata_cols` is `NULL`) to the vector of not-yet-used assignments remaining in that stratum's current block; `assign_wt()` pops the next assignment from the relevant queue, refilling it with a freshly drawn block (random size, randomly ordered) whenever it is empty.

Bootstrap. `draw_bootstrap_indices()` resamples within strata (via `stratified_bootstrap_indices_cpp()`) when `strata_cols` is supplied, or performs a plain i.i.d. nonparametric bootstrap over subjects otherwise.

Super classes

Design -> DesignSeqOneByOne -> DesignSeqOneByOneRandomBlockSize

Methods

Public methods:

- DesignSeqOneByOneRandomBlockSize$new()
- DesignSeqOneByOneRandomBlockSize$assign_wt()
- DesignSeqOneByOneRandomBlockSize$clone()

DesignSeqOneByOneRandomBlockSize$new(): Initialize a sequential permuted-block experimental design with randomly varying block size (see class documentation for the exact block-refill rule and its selection-bias rationale).

Usage:

```
DesignSeqOneByOneRandomBlockSize$new(
  strata_cols = NULL,
  block_sizes = c(4, 6, 8),
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`strata_cols` A character vector of column names to use for stratification. If NULL, simple blocking is used.

`block_sizes` A vector of positive integers representing the possible block sizes to choose from. Each must be a multiple of the inverse of `prob_T` to ensure integer treatment/control counts.

`response_type` The data type of response values which must be one of the following: "continuous", "incidence", "proportion", "count", "survival", "ordinal".

`prob_T` The probability of the treatment assignment. This defaults to `0.5`.

`include_is_missing_as_a_new_feature` If missing data is present in a variable, should we include another dummy variable for its missingness? Default is `TRUE`.

`n` The sample size (if fixed). Default is `NULL` for not fixed.

`verbose` A flag indicating whether messages should be displayed. Default is `FALSE`.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new ‘DesignSeqOneByOneRandomBlockSize‘ object

DesignSeqOneByOneRandomBlockSize$assign_wt(): Pop the next treatment assignment from the current subject’s stratum block queue (see class documentation), refilling that queue with a freshly drawn random-size, randomly-ordered block first if it is empty.

Usage:

```
DesignSeqOneByOneRandomBlockSize$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOneRandomBlockSize$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOneRandomBlockSize$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Efron, B. (1971). "Forcing a sequential experiment to be balanced." *Biometrika*, 58(3), 403-417, doi:10.1093/biomet/58.3.403, for sequential balanced-block randomization background. See also block randomisation for orientation on permuted-block designs and the selection-bias rationale for varying block size.

Examples

```
seq_des = DesignSeqOneByOneRandomBlockSize$new(n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
```

DesignSeqOneByOneSPBR *A Stratified Permuted-Block Sequential Design (SPBR) with Fixed Block Size*

Description

A `DesignSeqOneByOne` implementing classical stratified permuted-block randomization: subjects are assigned from a per-stratum queue of pre-shuffled treatment labels (a "block" of `block_size` labels, containing exactly `round(block_size * prob_T)` treated and the remainder control, in random order), refilled with a fresh randomly-ordered block of the **same fixed size** whenever a stratum's queue empties. This is the fixed-block-size, mandatory-stratification counterpart of `DesignSeqOneByOneRandomBlockSize` (which varies block size across draws and makes stratification optional): here, `strata_cols` is required, and a single fixed `block_size` is used for every stratum and every block, guaranteeing exact treatment/control balance within each stratum at every block boundary.

Block-size / `prob_T` compatibility. `block_size` must yield an integer number of treated subjects: the constructor errors unless `abs(block_size * prob_T - round(block_size * prob_T)) <= 1e-10`.

Per-stratum queues. As in `DesignSeqOneByOneRandomBlockSize`, `private$strata_states` is a hashed environment mapping each stratum key (concatenated `strata_cols` values, `"NA"` for missing) to the vector of not-yet-used assignments remaining in that stratum's current block; `assign_wt()` pops the next assignment, refilling with a fresh block when empty. `draw_bootstrap_indices()` resamples within strata by default (`bootstrap_type = "within_blocks"` or `NULL`) or resamples whole strata otherwise.

Super classes

Design -> DesignSeqOneByOne -> DesignSeqOneByOneSPBR

Methods

Public methods:

- DesignSeqOneByOneSPBR$new()
- DesignSeqOneByOneSPBR$assign_wt()
- DesignSeqOneByOneSPBR$clone()

DesignSeqOneByOneSPBR$new(): Initialize a stratified permuted-block sequential experimental design with fixed block size (see class documentation for the exact block-refill rule).

Usage:

```
DesignSeqOneByOneSPBR$new(
  strata_cols,
  block_size = 4,
  response_type,
  prob_T = 0.5,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`strata_cols` A character vector of column names to use for stratification.

`block_size` The size of the permuted blocks (fixed; see class documentation for its compatibility requirement with `prob_T`).

`response_type` "continuous", "incidence", "proportion", "count", "survival", or "ordinal".

`prob_T` Probability of treatment assignment.

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` A flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new ‘DesignSeqOneByOneSPBR‘ object

DesignSeqOneByOneSPBR$assign_wt(): Pop the next treatment assignment from the current subject’s stratum block queue (see class documentation), refilling that queue with a freshly drawn fixed-size, randomly-ordered block first if it is empty.

Usage:

```
DesignSeqOneByOneSPBR$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOneSPBR$clone()`: The objects of this class are cloneable with this method.

Usage:

```
DesignSeqOneByOneSPBR$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Zelen, M. (1974). "The randomization and stratification of patients to clinical trials." *Journal of Chronic Diseases*, 27(7-8), 365-375, doi:10.1016/00219681(74)900150, for stratified permuted-block randomization. See also block randomisation for orientation, and `DesignSeqOneByOneRandomBlockSize` for the randomly-varying-block-size variant.

Examples

```
seq_des = DesignSeqOneByOneSPBR$new(strata_cols = 'x1', n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = factor(1, levels=1:2)))
```

`DesignSeqOneByOneUrn` *Wei's (1977, 1978) Adaptive Urn Sequential Design, UD(α, β)*

Description

A `DesignSeqOneByOne` implementing Wei's adaptive biased-coin urn design $UD(\alpha, \beta)$: conceptually, an urn starts with α balls of each type (treatment and control), and each assignment is drawn proportionally to the current ball counts, then β balls of the **opposite** type to whatever was drawn are added back to the urn (so drawing treatment adds β control balls, and vice versa), pushing subsequent draws toward the under-represented arm. No covariates are used; only the running treated/control counts n_T, n_C matter, via the closed-form assignment probability

$$\Pr(w_t = 1) = \frac{\alpha + \beta\, n_C}{2\alpha + \beta(n_T + n_C)}.$$

Like `DesignSeqOneByOneEfron`, this design balances running assignment counts online while remaining strictly randomized (the probability is always strictly between 0 and 1 for finite $\alpha, \beta > 0$); unlike Efron's design (which only distinguishes "balanced" vs. "imbalanced" and applies a single fixed `weighted_coin_prob` in the imbalanced case), the urn design's bias toward the under-represented arm scales continuously and smoothly with the current *degree* of imbalance, tuned by the ratio β/α: larger β/α yields stronger balancing pressure, and $\beta = 0$ recovers a fixed $\mathrm{Bernoulli}(0.5)$ coin (no adaptation at all).

Super classes

`Design` -> `DesignSeqOneByOne` -> `DesignSeqOneByOneUrn`

Methods

Public methods:

- DesignSeqOneByOneUrn$new()
- DesignSeqOneByOneUrn$assign_wt()
- DesignSeqOneByOneUrn$clone()

`DesignSeqOneByOneUrn$new()`: Initialize Wei's UD(α, β) adaptive urn sequential experimental design (see class documentation for the exact assignment-probability formula).

Usage:

```
DesignSeqOneByOneUrn$new(
  alpha = 1,
  beta = 1,
  response_type,
  include_is_missing_as_a_new_feature = TRUE,
  n = NULL,
  verbose = FALSE,
  missingness_method = "impute",
  design_formula = ~.,
  seed = NULL
)
```

Arguments:

`alpha` The initial number of balls of each type (Treatment/Control) in the conceptual urn; larger `alpha` relative to `beta` weakens the balancing effect (assignment probabilities stay closer to 0.5 for longer).

`beta` The number of balls of the **opposite** type added to the urn after each assignment; `beta = 0` recovers an unbiased $\mathrm{Bernoulli}(0.5)$ coin (no balancing).

`response_type` The data type of response values.

`include_is_missing_as_a_new_feature` Flag for missingness indicators.

`n` The sample size.

`verbose` A flag for verbosity.

`missingness_method` How to handle missing values in covariates.

`design_formula` A formula object.

`seed` Integer seed for reproducibility.

Returns: A new 'DesignSeqOneByOneUrn' object

`DesignSeqOneByOneUrn$assign_wt()`: Draw the next subject's treatment assignment from Wei's UD(α, β) urn probability (see class documentation), computed from the running treated/control counts.

Usage:

```
DesignSeqOneByOneUrn$assign_wt()
```

Returns: The treatment assignment (0 or 1) for the next subject.

`DesignSeqOneByOneUrn$clone()`: The objects of this class are cloneable with this method.

Usage:

`DesignSeqOneByOneUrn$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

References

Wei, L. J. (1977). "A class of designs for sequential clinical trials." *Journal of the American Statistical Association*, 72(358), 382-386, doi:10.1080/01621459.1977.10481006; Wei, L. J. (1978). "The adaptive biased coin design for sequential experiments." *The Annals of Statistics*, 6(1), 92-100, doi:10.1214/aos/1176344068. See also randomized experiment for orientation on adaptive biased-coin sequential designs.

Examples

```
seq_des = DesignSeqOneByOneUrn$new(n = 6, response_type = 'continuous')
seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
```

EDI *Experimental Design and Inference*

Description

EDI (Experimental Design and Inference)

Details

Provides comprehensive support for many fixed and sequential experimental designs and many inferential methods (parametric, nonparametric, exact) for response types continuous, incidence, count, proportion, survival (with censoring) and ordinal. Supports automatic missing data imputation, parallelization, provides robustness fallbacks and is optimized with C++.

Author(s)

Adam Kapelner <adam.kapelner@mail.huji.ac.il>

References

Adam Kapelner and Abba Krieger A Matching Procedure for Sequential Experiments that Iteratively Learns which Covariates Improve Power, Arxiv 2010.05980

See Also

Useful links:

- https://github.com/kapelner/EDI
- https://kapelner.github.io/EDI/
- https://pypi.org/project/edi-kernels/
- Report bugs at https://github.com/kapelner/EDI/issues

edi_build_info_cpp *Return EDI Build Information (C++ Backend)*

Description

Returns the compiler and package build metadata that was baked into the currently loaded EDI shared object **at compile time** (via preprocessor macros defined in `edi_build_flags.h`, generated by the package's build tooling), not anything queried live from the running system. This is intended for benchmark reports and reproducibility audits where the exact compiler, flags, and build environment used to produce the installed binary matter (e.g. explaining performance differences between two installations of the same package version, or confirming a binary was built with a particular optimization/vectorization configuration).

Usage

```
edi_build_info_cpp()
```

Details

Every field is a fixed string or boolean baked in when the C++ source was compiled (via preprocessor macro substitution); calling this function multiple times within the same R session always returns identical values, and the values reflect the build environment, not the environment the function happens to be *called* from. If EDI is reinstalled/recompiled, existing R sessions that already loaded the old shared object continue to report the old build's metadata until they restart and load the new one.

Value

A named list with the following fields, all length-1 character strings unless noted otherwise:

capture_method How the build metadata below was captured by the package's build tooling (`EDI_BUILD_CAPTURE_METHOD`).

build_timestamp Timestamp of compilation (`EDI_BUILD_TIMESTAMP`).

build_host Hostname of the machine that compiled this binary (`EDI_BUILD_HOST`).

r_home `R_HOME` of the R installation used to build the package.

r_version R version string used to build the package.

r_cxx20 The C++20 compiler command R's build configuration reported.

r_cxx20std The C++ standard flag (e.g. `-std=gnu++20`) used.

r_cxx20flags Additional C++20 compiler flags from R's configuration.

r_shlib_openmp_cxxflags OpenMP C++ flags R's configuration supplies for shared-library builds (empty if OpenMP support was not available/enabled).

env_edi_portable, env_edi_disable_vectorization, env_edi_native_speed, env_edi_native_lto The values (or default/unset indicator) of the corresponding `EDI_*` environment variables that were set at build time to control portable-vs-native code generation, vectorization, and link-time optimization; see the package's build documentation for what each controls.

pkg_cppflags, pkg_cxxflags, pkg_libs The package-level PKG_CPPFLAGS/ PKG_CXXFLAGS/PKG_LIBS used when compiling/linking EDI's C++ sources.

compiler The compiler identification string (__VERSION__).

compiler_optimize_macro Logical; TRUE iff the compiler defined __OPTIMIZE__ (i.e. optimizations were enabled) when this translation unit was compiled.

compiler_fast_math_macro Logical; TRUE iff __FAST_MATH__ was defined (i.e. non-IEEE-compliant fast-math optimizations were enabled), which can affect floating-point reproducibility/edge-case behavior (NaN/Inf handling, exact rounding) relative to a standard-compliant build.

eigen_dont_vectorize_macro Logical; TRUE iff EIGEN_DONT_VECTORIZE was defined, disabling Eigen's SIMD vectorization for this build (e.g. for portability to CPUs lacking the vector instructions a native build would target).

Examples

```
info = edi_build_info_cpp()
info$pkg_cxxflags
```

EDI_COMPREHENSIVE_SLOW_PATHS

Comprehensive-test slow-path registry

Description

Performance-based exclusions used by EDI's comprehensive test harness. These rules describe paths that are implemented but intentionally omitted from routine exhaustive execution because their observed runtime is too high. They do not change an inference class's public capabilities and must not be interpreted as "not implemented" declarations.

Usage

```
EDI_COMPREHENSIVE_SLOW_PATHS
```

Format

A named list. ‘exact_operations‘ contains keys of the form ‘response_type||InferenceClass||operation‘, optionally suffixed with ‘||model_formula‘ (e.g. ‘||~1‘) to restrict the entry to that one formula – an operation with no formula suffix is skipped for every formula, matching the pre-2026-09-08 behavior. ‘model_formula‘ is matched as the deparsed formula text (‘"~1"‘, ‘"~."‘, ...). Every other element contains formula-, dataset-, and design-independent concrete inference-class names for the named slow-path family.

Details

Rule-name prefixes use the harness vocabulary: ‘boot‘ is ordinary nonparametric bootstrap, ‘bbt‘ is Bayesian bootstrap, ‘brt‘ is bootstrap randomization, ‘pboot‘/‘param_bootstrap‘ are parametric bootstrap, and ‘rand‘ is randomization inference. Suffixes identify the affected CI, p-value, or typed variant. A class can appear in more than one category.

‘InferenceSuite$run_all_inference()‘ also omits matching class/method/type combinations when its ‘methods‘ argument is left at the default ‘NULL‘. Supplying ‘methods‘ explicitly opts into the requested paths even when they appear in this registry. Because of that coupling, an ‘exact_operations‘ entry must never name the base ‘compute_estimate‘ operation itself (only a CI/p-value sub-method, e.g. ‘compute_rand_two_sided_pval‘) – doing so doesn’t just skip one slow sub-computation, it silently drops the whole class from ‘run_all_inference()‘’s default output (found 2026-08-26: a pre-existing ‘"survival||InferenceSurvivalCoxPHRegr||compute_estimate"‘ entry, harmless while this registry was internal-only, started doing exactly that the moment ‘run_all_inference()‘ began consulting it – ‘compute_estimate()‘ for that class takes ~0.09s, nowhere near "too slow"; removed).

See Also

[InferenceSuite]

---

exact_jonckheere_terpstra_pval_cpp

Exact Two-Group Jonckheere-Terpstra Test via Full Randomization Enumeration (C++ Backend)

Description

Computes the exact randomization-distribution p-value and a probabilistic-index effect size for the two-group Jonckheere-Terpstra statistic — which, with exactly two groups (`w` in `{0, 1}`), coincides with the Wilcoxon-Mann-Whitney U statistic generalized to handle ties (repeated ordinal levels in `y`):

$$U = \sum_k t_k \big(2L_k + (n_k - t_k)\big)/2,$$

summed over the K distinct observed levels of `y` (in increasing order), where n_k is the total count at level k, t_k is the observed count of `w == 1` subjects at level k, and L_k is the number of subjects at strictly lower levels (this is the standard "number of favorable comparisons" Mann-Whitney statistic, adapted for tied/grouped ordinal data — a tie at the same level contributes $1/2$ rather than 0 or 1). The exact (not asymptotic, not Monte Carlo) null/reference distribution of this statistic under the sharp null of no treatment effect is obtained by enumerating, via a dynamic-programming recursion over levels (`recurse_jt_distribution()`), every way to distribute `n_treat` "treated" labels among the n subjects consistent with the fixed per-level totals n_k — i.e. the exact multivariate hypergeometric randomization distribution of the statistic conditional on the observed level counts, with each configuration’s probability computed in log-space from log-binomial-coefficient weights to avoid overflow for larger n.

Usage

```
exact_jonckheere_terpstra_pval_cpp(y, w)
```

Arguments

y	Integer (or integer-coercible) vector of length n giving each subject's ordinal response value; ties (repeated values) are handled via the grouped Mann-Whitney formula above. Must not contain NA.
w	Integer (or integer-coercible) vector of length n with values in {0, 1} giving each subject's group membership; both groups must be non-empty and no NA is permitted.

Details

Randomization test, not a model-based test. This is a randomization (permutation) test in the Fisherian sense: it conditions on the observed marginal level counts n_k and the group sizes n_treat/n_control, and asks how extreme the observed statistic is relative to every other way those same labels could have been randomly assigned, so its validity does not depend on any distributional assumption about y beyond exchangeability under the null. The two-sided p-value is $p_{\text{exact}} = \min(1, 2\min(p_{\text{lower}}, p_{\text{upper}}))$, where p_{lower}/p_{upper} are the exact one-sided tail probabilities of the randomization distribution at or below / at or above the observed statistic.

Effect size (superiority). superiority is the probabilistic index $\Pr(Y_T > Y_C) + \frac{1}{2}\Pr(Y_T = Y_C)$ (equivalently U rescaled to $[0, 1]$ by dividing by $n_{\text{treat}} \cdot n_{\text{control}}$), the probability a randomly chosen treated subject's ordinal outcome exceeds a randomly chosen control subject's, counting ties as half a win; 0.5 indicates no stochastic ordering between groups, and 1/0 indicate the treated group's outcomes are uniformly higher/lower.

Input conventions. y is coerced to integer and treated as an ordinal (or any orderable-by-integer-value) response with an arbitrary number of tied levels; w must be an integer/coercible-to-integer vector of {0, 1} values with both groups non-empty. NA in either y or w is not permitted and raises an error, as does a non-{0,1} value in w or an empty input.

Complexity. The recursion's state space scales with the number of distinct possible statistic values ($O(n_{\text{treat}} \cdot n_{\text{control}})$ many), and thread-local buffers are reused (not reallocated) across repeated calls within the same thread for the same or smaller problem sizes; this exact enumeration is exponential in the number of distinct levels/group sizes in the worst case (unlike an asymptotic normal-approximation JT test), so it is intended for small-to- moderate n where exactness matters more than raw speed.

Value

A list with components stat2 (twice the observed U statistic, an integer, used internally to keep the enumeration in integer arithmetic), n_treat, n_control (the two group sizes), superiority (the probabilistic-index effect size described above), p_lower, p_upper (the exact one-sided randomization-distribution tail probabilities), and p_exact (the exact two-sided p-value).

See Also

Mann-Whitney U test and Jonckheere's trend test for background; analogous Python API: SciPy mannwhitneyu (method="exact" for the same exact-enumeration approach, though SciPy's exact

path does not handle ties the same way).

`fast_beta_regression` *Fast Beta Regression (R Wrapper)*

Description

Fits the beta regression model of Ferrari and Cribari-Neto (2004) for a continuous response strictly in $(0, 1)$, with mean linked to the covariates via the logit link and a single (constant) precision parameter ϕ. See `fast_beta_regression_cpp` for the full model equation, parameter layout, and optimizer contract implemented by the C++ backend this function wraps; this page documents only the R-level fallback chain and response-scale conventions.

Usage

```
fast_beta_regression(
  X,
  y,
  start_phi = 10,
  optimization_alg = "lbfgs",
  warm_start_beta = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

`X` A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.

`y` A numeric vector of the response variable, with values strictly between 0 and 1. See `sanitize_beta_response()` (internal) for how boundary values (exact 0s/1s) are handled before fitting.

`start_phi` A numeric value, the starting value for the precision parameter phi. Defaults to 10.

`optimization_alg` Optimization algorithm: `"lbfgs"` (default) or `"newton_raphson"`; see `.normalize_optimizer_algori`

`warm_start_beta` Optional starting values for coefficients β.

`warm_start_fisher_info` Optional initial Fisher Information matrix, used to warm-start curvature information for the optimizer.

Details

Fallback chain. The primary implementation uses the C++ backend (`fast_beta_regression_cpp`). If that fails to converge or errors, the function falls back to **betareg**, which is listed in Suggests and is not installed automatically with **EDI**. If **betareg** is also unavailable (or itself fails), a final fallback of OLS on `logit(y)` is used — this last resort is always available (no external dependency) but does not respect the beta distribution's mean-variance relationship or estimate ϕ at all, so its coefficients should be treated as an approximate, non-model-based summary rather than a true beta-regression fit. Install **betareg** manually to enable the intermediate fallback. A `warning()` is issued whenever a fallback stage is used, naming which stage and the triggering error, so callers can detect when the primary fit failed even though a result was still returned.

Value

A list containing the following components:

`b` A numeric vector of the estimated beta regression coefficients $\hat{\beta}$ (on the logit-of-mean scale: `plogis(X %*% b)` gives the fitted mean $\hat{\mu}$), from whichever stage of the fallback chain (see Details) ultimately succeeded.

`phi` The estimated precision parameter $\hat{\phi}$ (only present when the C++ backend or the **betareg** fallback succeeds; absent from the final OLS-on-`logit(y)` fallback, which has no precision parameter).

`fisher_information` The working-weights Fisher information matrix from the C++ backend (see `fast_beta_regression_cpp`); only present when that backend succeeds.

Examples

```
X = matrix(rnorm(500), 100, 5)
y = runif(100)
fast_beta_regression(X, y)
```

`fast_beta_regression_with_var` *Fast Beta Regression with Variance Calculation (R Wrapper)*

Description

Fits the same beta regression model as `fast_beta_regression` (see `fast_beta_regression_cpp` for the full model equation and parameterization) and additionally reports the estimated variance of a caller-selected coefficient, extracted from the fitted parameter variance-covariance matrix (see `fast_beta_regression_with_var_cpp` for how that matrix is computed and its plain-inverse numerical caveat on rank-deficient designs).

Usage

```
fast_beta_regression_with_var(
  X,
  y,
  start_phi = 10,
  j = 2,
  optimization_alg = "lbfgs",
  warm_start_beta = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

`X` A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.

`y` A numeric vector of the response variable, with values strictly between 0 and 1. See `sanitize_beta_response()` (internal) for boundary-value handling.

`start_phi` A numeric value, the starting value for the precision parameter phi. Defaults to 10.

`j` The 1-based index (into `X`'s columns, i.e. into β) of the coefficient to report the variance of as `ssq_b_j`. Defaults to 2 (the package's usual convention for the treatment-effect column when an intercept occupies column 1).

`optimization_alg` Optimization algorithm: `"lbfgs"` (default) or `"newton_raphson"`; see `.normalize_optimizer_algori`

`warm_start_beta` Optional starting values for coefficients β.

`warm_start_fisher_info` Optional initial Fisher Information matrix, used to warm-start curvature information for the optimizer.

Details

The primary implementation uses a C++ backend. If that fails, the function falls back to **betareg**, which is listed in Suggests and is not installed automatically with **EDI**. If **betareg** is also unavailable, a final fallback of OLS on `logit(y)` is used. Install **betareg** manually to enable the intermediate fallback.

Value

A list containing the following components:

`b` A numeric vector of the estimated beta regression coefficients $\hat{\beta}$ (logit-of-mean scale).

`ssq_b_j` The estimated variance (squared standard error) of the `j`-th coefficient, $\widehat{\mathrm{Var}}(\hat{\beta}_j)$, i.e. the `j`-th diagonal entry of `vcov`. `NA` if the primary C++ fit failed and a fallback stage without a variance estimate was used (see Details).

ssq_b_2 — The estimated variance of the second coefficient specifically ($\widehat{\mathrm{Var}}(\hat{\beta}_2)$), regardless of the j argument — provided as a convenience since column 2 is the package's usual treatment-effect position. Identical to ssq_b_j when j = 2.

Examples

```
X = matrix(rnorm(100), 10, 10)
y = runif(10)
fast_beta_regression_with_var(X, y)
```

---

fast_coxph_regression *Fast Cox Proportional Hazards Regression (R Wrapper)*

Description

Fits the Cox proportional-hazards partial-likelihood model documented in full at build_cox_data_cache_cpp (model equation, Breslow tie-handling, and input conventions). This R-level wrapper dispatches to either the package's own native C++ implementation (fast_coxph_regression_cpp, the default and recommended path) or, for cross-checking or when the Rcpp path is unavailable, an elastic-net-with-zero-penalty Cox fit via **glmnet** (use_rcpp = FALSE) — **not survival::coxph**, despite that being the more commonly used reference implementation for Cox models in R.

Usage

```
fast_coxph_regression(
  X,
  y,
  dead,
  use_rcpp = TRUE,
  estimate_only = FALSE,
  optimization_alg = "lbfgs",
  warm_start_beta = NULL,
  warm_start_fisher_info = NULL,
  smart_cold_start = TRUE
)
```

Arguments

X — A numeric matrix of predictor variables. It is assumed that an intercept term is handled implicitly by the Cox model and should not be included in X.

y — A numeric vector representing the observed time (event time or censoring time).

dead — A numeric vector (0 or 1) indicating event status (1 for event, 0 for censored).

use_rcpp — Logical. If TRUE (default), use the optimized Rcpp implementation (fast_coxph_regression_cpp). If FALSE, use **glmnet**'s Cox path at zero penalty (glmnet(..., family = "cox",lambda = 0)) instead.

`estimate_only` Logical. If `TRUE`, skip variance-covariance matrix calculation for speed. Only affects the `use_rcpp = TRUE` path; the **glmnet** fallback path does not compute a variance-covariance matrix at all (`vcov` is never populated when `use_rcpp = FALSE`, regardless of `estimate_only`).

`optimization_alg` Optimization algorithm: `"newton_raphson"` (default) or `"lbfgs"`. Only affects the `use_rcpp = TRUE` path; unused when `use_rcpp = FALSE`.

`warm_start_beta` Optional starting values for coefficients. If provided, `smart_cold_start` is ignored. Only affects the `use_rcpp = TRUE` path.

`warm_start_fisher_info` Optional initial Fisher Information matrix. Only affects the `use_rcpp = TRUE` path.

`smart_cold_start` Logical. If `TRUE` (default), use an initial OLS-based guess when starting from scratch (a "cold start") with no prior knowledge. This is ignored if `warm_start_beta` is provided. Only affects the `use_rcpp = TRUE` path.

Details

Failure semantics. If the C++ fit (`use_rcpp = TRUE`) errors or fails to report `converged`, this function stops with an error rather than silently falling back to **glmnet** — the two code paths are alternative *caller choices*, not an automatic fallback chain (contrast with, e.g., `fast_beta_regression`'s automatic **betareg** fallback).

glmnet dependency. When `use_rcpp = FALSE`, this function requires the **glmnet** package, which is listed in Suggests and is not installed automatically with **EDI**; it errors immediately if **glmnet** is not installed.

Value

A list. When `use_rcpp = TRUE` (default), a list with components

b, coefficients A numeric vector of the estimated log-hazard-ratio coefficients $\hat{\beta}$ (`b` and `coefficients` are identical; both are populated for interface consistency with the package's other `fast_*` wrappers).

vcov The variance-covariance matrix of $\hat{\beta}$, or `NULL` when `estimate_only = TRUE`.

neg_log_lik The negative Cox partial log-likelihood at the fitted coefficients.

fisher_information The Hessian of the negative partial log-likelihood at the fitted coefficients.

When `use_rcpp = FALSE`, only a single component, `b` (the **glmnet**-fitted coefficient vector via `coef()`, in **glmnet**'s own sparse-matrix representation rather than a plain numeric vector) — none of `coefficients`/`vcov`/`neg_log_lik`/ `fisher_information` are present on this path.

See Also

`build_cox_data_cache_cpp` for the full Cox partial-likelihood model, Breslow tie-handling, and input conventions; `fast_coxph_regression_cpp` for the native C++ backend this wrapper calls by default.

Examples

```
X = matrix(rnorm(500), 100, 5)
y = runif(100)
dead = rbinom(100, 1, 0.5)
fast_coxph_regression(X, y, dead)
```

`fast_coxph_regression_cpp`
Fast Cox Proportional Hazards Regression, One-Shot Fit (C++ Backend)

Description

Fits the unstratified Cox proportional-hazards partial-likelihood model documented in full at `build_cox_data_cache_cpp` — the same model, Breslow tie-handling, and input conventions — in a single call that internally builds the sorted risk-set cache, runs the optimizer, and discards the cache afterward. Use this entry point for a one-off fit; use `build_cox_data_cache_cpp` plus `fast_coxph_regression_prebuilt_cpp` instead when fitting the same `(X, y, dead)` repeatedly (e.g. across bootstrap/randomization replicates), to avoid rebuilding the risk-set cache on every call. `fast_coxph_regression` is the R-level wrapper around this backend (with an **survival**-free-of-Rcpp fallback path via **glmnet**).

Usage

```
fast_coxph_regression_cpp(
  X,
  y,
  dead,
  warm_start_beta = NULL,
  smart_cold_start = TRUE,
  estimate_only = FALSE,
  maxit = 20L,
  tol = 1e-9,
  cluster = NULL,
  fixed_idx = NULL,
  fixed_values = NULL,
  optimization_alg = "newton_raphson",
  warm_start_fisher_info = NULL
)
```

Arguments

`X`	A numeric matrix of predictor variables (no intercept column; see `build_cox_data_cache_cpp`).
`y`	Numeric vector of observed (event or censoring) times.
`dead`	Numeric vector with values in `{0, 1}`: event indicator (1 = event, 0 = right-censored).

`warm_start_beta`
: Optional starting values for the coefficients β.

`smart_cold_start`
: Logical. If `TRUE` (default) and no `warm_start_beta` is supplied, use an OLS-based initial guess rather than a zero cold start.

`estimate_only`
: Logical. If `TRUE`, skip variance-covariance matrix calculation for speed.

`maxit`
: Maximum number of Newton-Raphson/L-BFGS iterations.

`tol`
: Convergence tolerance.

`cluster`
: Optional clustering variable; when supplied, the returned variance-covariance matrix uses a cluster-robust (grouped) sandwich correction instead of the naive model-based inverse-information variance, i.e. one that remains asymptotically valid under within-cluster correlation of the martingale residuals.

`fixed_idx`
: Optional integer indices of coefficients to hold fixed rather than estimate.

`fixed_values`
: Optional values to fix the parameters named by `fixed_idx` at; must be the same length as `fixed_idx`.

`optimization_alg`
: Optimization algorithm: `"newton_raphson"` (default) or `"lbfgs"`.

`warm_start_fisher_info`
: Optional initial Fisher Information matrix to warm-start curvature information for the optimizer.

Value

A list containing the following components:

coefficients A numeric vector of the estimated log-hazard-ratio coefficients $\hat{\beta}$.

vcov The variance-covariance matrix of $\hat{\beta}$ (naive inverse-information, or cluster-robust sandwich if `cluster` is supplied); omitted/not computed when `estimate_only = TRUE`.

neg_ll The negative Cox partial log-likelihood at the final iteration.

converged A logical value indicating whether the algorithm converged.

iterations The number of optimizer iterations performed.

fisher_information The Hessian of the negative partial log-likelihood at the fitted coefficients (the observed information matrix).

gradient_norm The norm of the score (gradient) vector at convergence, a diagnostic of how tightly the convergence criterion was met.

See Also

`build_cox_data_cache_cpp` for the full Cox partial-likelihood model, Breslow tie-handling, and input conventions this function implements; `fast_coxph_regression_prebuilt_cpp` for the cache-reusing variant; `fast_coxph_regression` for the R-level wrapper.

fast_coxph_regression_prebuilt_cpp

Fast Cox Proportional Hazards Regression, Cache-Reusing Fit (C++ Backend)

Description

Fits the same unstratified-*or*-stratified Cox partial-likelihood model documented at `build_cox_data_cache_cpp` / `build_stratified_cox_data_cache_cpp`, but takes a **pre-built** risk-set cache (`cox_data_xptr`, an `externalptr` produced by one of those two functions) instead of raw `(X, y, dead)` data, skipping the sort/tabulation step on every call. This is the entry point the package's Cox inference classes (e.g. `InferenceCoxPH`, `InferenceStratifiedCoxPH`) use for repeated fits on the same data (successive `estimate_only` vs. full-variance calls, or bootstrap/ randomization replicates that only change the treatment column of `X`, rebuilding the cache only when the covariates or assignment actually change). `fast_coxph_regression_cpp` is the equivalent one-shot entry point that builds and discards the cache internally, for callers that only need a single fit.

Usage

```
fast_coxph_regression_prebuilt_cpp(
  cox_data_xptr,
  warm_start_beta = NULL,
  smart_cold_start = TRUE,
  estimate_only = FALSE,
  maxit = 20L,
  tol = 1e-9,
  fixed_idx = NULL,
  fixed_values = NULL,
  optimization_alg = "newton_raphson",
  warm_start_fisher_info = NULL
)
```

Arguments

`cox_data_xptr`	An `externalptr` to a cached Cox risk-set representation, as returned by `build_cox_data_cache_cpp` (unstratified) or `build_stratified_cox_data_cache_cpp` (stratified).
`warm_start_beta`	Optional starting values for the coefficients β.
`smart_cold_start`	Logical. If `TRUE` (default) and no `warm_start_beta` is supplied, use an OLS-based initial guess rather than a zero cold start.
`estimate_only`	Logical. If `TRUE`, skip variance-covariance matrix calculation for speed.
`maxit`	Maximum number of Newton-Raphson/L-BFGS iterations.
`tol`	Convergence tolerance.
`fixed_idx`	Optional integer indices of coefficients to hold fixed rather than estimate.

`fixed_values` Optional values to fix the parameters named by `fixed_idx` at; must be the same length as `fixed_idx`.

`optimization_alg` Optimization algorithm: `"newton_raphson"` (default) or `"lbfgs"`.

`warm_start_fisher_info` Optional initial Fisher Information matrix to warm-start curvature information for the optimizer.

Details

Because `cox_data_xptr` carries a fixed, already-sorted risk-set structure (whether unstratified — one risk set — or stratified — one risk set per stratum, depending on which cache-building function produced it), the number and identity of subjects/strata are entirely determined by the cache; only the optimization behavior (warm starts, convergence, algorithm) is configurable through this function's own arguments. Passing a stale cache (built from data that has since changed) silently fits the model to the *cached* data, not the caller's current `X`/`y`/`dead` — callers are responsible for invalidating and rebuilding the cache when the underlying data changes; see `build_cox_data_cache_cpp` for the exact caching/mutation contract.

Value

A list containing the following components:

coefficients A numeric vector of the estimated log-hazard-ratio coefficients $\hat{\beta}$.

vcov The variance-covariance matrix of $\hat{\beta}$; omitted when `estimate_only = TRUE`.

neg_ll The negative Cox partial log-likelihood at the final iteration.

converged A logical value indicating whether the algorithm converged.

iterations The number of optimizer iterations performed.

fisher_information The Hessian of the negative partial log-likelihood at the fitted coefficients.

gradient_norm The norm of the score (gradient) vector at convergence.

See Also

`build_cox_data_cache_cpp`/`build_stratified_cox_data_cache_cpp` for building the required cache and the full Cox partial-likelihood model documentation; `fast_coxph_regression_cpp` for the one-shot (build-and-discard) variant.

`fast_logistic_regression`

Fast Logistic Regression, Estimate Only (R Wrapper)

Description

Fits the logistic regression model documented in full at fast_logistic_regression_cpp (log-odds-ratio interpretation, IRLS/L-BFGS/Newton-Raphson optimization) via that C++ backend, returning only the point estimate $\hat{\beta}$ — no variance-covariance matrix or per-coefficient standard errors are computed. Unlike fast_logistic_regression_with_var, this function does **not** attempt to detect or retry on (quasi-)complete separation; if the underlying C++ fit errors for any reason, this function silently returns b as a vector of NAs (of length ncol(X)) rather than raising an error or retrying with fewer covariates.

Usage

```
fast_logistic_regression(
  X,
  y,
  optimization_alg = "lbfgs",
  warm_start_beta = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

X — A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in X if desired.

y — A numeric vector of the response variable, expected to be binary (0 or 1).

optimization_alg — Optimization algorithm: "lbfgs" (default), "newton_raphson", or "irls".

warm_start_beta — Optional starting values for the coefficients.

warm_start_fisher_info — Optional initial Fisher Information matrix.

Value

A list containing the following component:

b A numeric vector of the estimated logistic regression coefficients $\hat{\beta}$, or a vector of NA_real_ (length ncol(X)) if the underlying fit errored.

See Also

fast_logistic_regression_cpp for the underlying backend and full model documentation; fast_logistic_regression_ for the variance- augmented, separation-retrying variant.

Examples

```
X = matrix(rnorm(500), 100, 5)
y = rbinom(100, 1, 0.5)
fast_logistic_regression(X, y)
```

fast_logistic_regression_cpp

Fast Logistic Regression, Estimate Only (C++ Backend)

Description

Fits the standard binary logistic regression model, $\mathrm{logit}(\mu_i) = \Pr(Y_i = 1)$'s log-odds $= x_i^\top \beta$, $\mu_i = \mathrm{logit}^{-1}(x_i^\top \beta)$, via maximum likelihood. Coefficients are directly interpretable as **log odds ratios**: e^{β_j} is the multiplicative change in the odds $\mu_i/(1-\mu_i)$ per unit change in covariate j. This is the package's baseline binary-response fitting backend, used wherever an incidence/binary outcome needs a logit-link fit (as opposed to the log-link or identity-link constrained binomial models in `fast_log_binomial_regression_cpp`/ `fast_identity_binomial_regression_cpp`, which target relative risk / risk difference scales instead of odds ratios).

Usage

```
fast_logistic_regression_cpp(
  X,
  y,
  warm_start_beta = NULL,
  smart_cold_start = FALSE,
  maxit = 100L,
  tol = 1e-8,
  fixed_idx = NULL,
  fixed_values = NULL,
  optimization_alg = "irls",
  warm_start_weights = NULL,
  warm_start_fisher_info = NULL,
  estimate_only = FALSE
)
```

Arguments

`X` A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.

`y` A numeric vector of the response variable, expected to be binary (0 or 1).

`warm_start_beta` Optional starting values for coefficients β. If provided, `smart_cold_start` is ignored.

`smart_cold_start` Logical. If `TRUE` and no `warm_start_beta` is supplied, use an OLS-based initial guess rather than a zero cold start. Default `FALSE` for this function (unlike most of the package's other `fast_*` fitters, which default this to `TRUE`), since IRLS (the default optimizer here) is typically robust enough from a zero start for well-behaved logistic regression problems.

`maxit` Maximum number of iterations for the algorithm. Defaults to 100.

`tol` Convergence tolerance. Defaults to 1e-8.

`fixed_idx` Optional integer indices of coefficients to hold fixed rather than estimate.

`fixed_values` Optional values to fix the parameters named by `fixed_idx` at.

`optimization_alg`
Optimization algorithm: `"irls"` (default, classical iteratively-reweighted-least-squares Fisher scoring), `"lbfgs"`, or `"newton_raphson"`; see `.normalize_optimizer_algorithm`.

`warm_start_weights`
Optional initial IRLS working weights for the first iteration.

`warm_start_fisher_info`
Optional initial Fisher Information matrix to warm-start curvature information.

`estimate_only` Logical. If `TRUE`, skip the working-weights/ score/Fisher-information computation after convergence, returning only `b`, `converged`, `num_iter`, `hit_iteration_cap`, and `gradient_norm`.

Value

A list containing the following components:

b A numeric vector of the estimated logistic regression coefficients $\hat{\beta}$.

w The IRLS working weights $\hat{\mu}_i(1 - \hat{\mu}_i)$ at the final iteration (the Bernoulli variance function evaluated at the fitted probabilities); omitted when `estimate_only = TRUE`.

num_iter The number of optimizer iterations performed.

fisher_information The working-weights curvature matrix $X^\top W X$; omitted when `estimate_only = TRUE`.

score The score (gradient of the log-likelihood) vector at the fitted coefficients; omitted when `estimate_only = TRUE`.

neg_ll The negative log-likelihood at the fitted coefficients; omitted when `estimate_only = TRUE`.

converged A logical value indicating whether the final gradient norm was below `tol` (`gradient_norm < tol`); uniform across the `"irls"`/`"lbfgs"` optimizers.

hit_iteration_cap A logical value, mutually exclusive with `converged`: `TRUE` iff the optimizer exhausted `maxit` iterations without meeting the gradient-norm convergence criterion.

gradient_norm The norm of the score vector at the returned coefficients, a diagnostic of how tightly the convergence criterion was met.

See Also

`fast_logistic_regression_with_var_cpp` for the variance-augmented variant; `fast_logistic_regression` for the R-level wrapper; `fast_log_binomial_regression_cpp`/ `fast_identity_binomial_regression_cpp` for the log-link/ identity-link analogs targeting relative-risk/risk-difference scales.

fast_logistic_regression_weighted_cpp

Fast Weighted Logistic Regression, Estimate Only (C++ Backend)

Description

Fits the same logistic regression model as `fast_logistic_regression_cpp` (see that page for the full model, log-odds-ratio interpretation, and optimizer contract), with each observation's contribution to the log-likelihood and IRLS working weights multiplied by a row weight `weights[i]`. Setting all weights to 1 recovers `fast_logistic_regression_cpp` exactly; this is the backend used when the logistic model must be fit on bootstrap-reweighted or otherwise weighted data.

Usage

```
fast_logistic_regression_weighted_cpp(
  X,
  y,
  weights,
  warm_start_beta = NULL,
  smart_cold_start = FALSE,
  maxit = 100L,
  tol = 1e-8,
  fixed_idx = NULL,
  fixed_values = NULL,
  optimization_alg = "irls",
  warm_start_weights = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

`X`	A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.
`y`	A numeric vector of the response variable, expected to be binary (0 or 1).
`weights`	A numeric vector of weights for each observation.
`warm_start_beta`	Optional starting values for coefficients β. If provided, `smart_cold_start` is ignored.
`smart_cold_start`	Logical. If `TRUE` and no `warm_start_beta` is supplied, use an OLS-based initial guess rather than a zero cold start.
`maxit`	Maximum number of iterations for the IRLS algorithm. Defaults to 100.
`tol`	Convergence tolerance. Defaults to 1e-8.
`fixed_idx`	Optional integer indices of coefficients to hold fixed rather than estimate.
`fixed_values`	Optional values to fix the parameters named by `fixed_idx` at.

`optimization_alg`
: Optimization algorithm: `"irls"` (default), `"lbfgs"`, or `"newton_raphson"`; see `.normalize_optimizer_algorithm`.

`warm_start_weights`
: Optional initial IRLS working weights for the first iteration.

`warm_start_fisher_info`
: Optional initial Fisher Information matrix to warm-start curvature information.

Value

A list containing the following components:

b A numeric vector of the estimated logistic regression coefficients $\hat{\beta}$.

mu The fitted probabilities $\hat{\mu}_i$.

XtWX, fisher_information Two aliases for the same working-weights curvature matrix $X^\top W X$ at the final iteration.

score The (weighted) score vector at the fitted coefficients.

neg_ll The weighted negative log-likelihood at the fitted coefficients.

converged A logical value indicating whether the final gradient norm was below `tol` (`gradient_norm < tol`); uniform across the `"irls"`/`"lbfgs"` optimizers.

num_iter The number of optimizer iterations performed.

hit_iteration_cap A logical value, mutually exclusive with `converged`: `TRUE` iff the optimizer exhausted `maxit` iterations without meeting the gradient-norm convergence criterion.

gradient_norm The norm of the score vector at the returned coefficients.

See Also

`fast_logistic_regression_cpp` for the unweighted model and full documentation.

`fast_logistic_regression_with_var`
: *Fast Logistic Regression with Variance, Auto-Retrying on Separation (R Wrapper)*

Description

Fits the logistic regression model documented in full at `fast_logistic_regression_cpp` (log-odds-ratio interpretation, IRLS/L-BFGS/Newton-Raphson optimization) via `fast_logistic_regression_with_var_cpp`, and additionally **detects and automatically retries on (quasi-)complete separation** — the well-known logistic-regression failure mode where the MLE does not exist because some linear combination of covariates perfectly (or near-perfectly) predicts the outcome, causing the optimizer's coefficient estimates to diverge to a large-but-finite value that would otherwise silently pass ordinary `is.finite()` convergence checks and corrupt downstream confidence intervals.

Usage

```
fast_logistic_regression_with_var(
  X,
  y,
  j = 2,
  optimization_alg = "lbfgs",
  warm_start_beta = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

`X` A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.

`y` A numeric vector of the response variable, expected to be binary (0 or 1).

`j` The index of the coefficient to compute the variance for. Defaults to 2.

`optimization_alg` Optimization algorithm: `"lbfgs"` (default), `"irls"`, or `"newton_raphson"`.

`warm_start_beta` Optional starting values for the coefficients.

`warm_start_fisher_info` Optional initial Fisher Information matrix.

Details

Separation detection and retry. After each fit attempt, `is_separated_coefficient_magnitude()` checks whether any fitted coefficient exceeds a fixed separation-detection threshold (`EDI_SEPARATION_THRESHOLD`); if so, the fit is treated as `converged = FALSE` regardless of what the underlying C++ optimizer itself reported. On non-convergence (including detected separation), the **covariate (column index $\geq$ 3, i.e. never the intercept in column 1 or the treatment column in column 2) with the largest absolute fitted coefficient** is dropped, and the model is refit on the reduced design; this repeats until either the fit converges or only the intercept and treatment columns remain. If separation persists even with just those two columns, this function `stop()`s with an explicit "complete separation detected" error rather than returning a corrupted variance estimate.

Interpretation caveat. Because covariates can be silently dropped by this retry loop, the returned `b` may have fewer coefficients than `ncol(X)` implies, and the fitted model's covariate adjustment set can differ from what was requested; callers relying on a specific covariate being present in the final fit should check for this rather than assume it always is.

Value

A list containing the following components:

b A numeric vector of the obtained logistic regression coefficients $\hat{\beta}$, from whichever (possibly covariate-reduced) fit in the retry sequence ultimately converged.

ssq_b_j The squared standard error (variance) of the j-th estimated coefficient.

ssq_b_2 The squared standard error (variance) of the second estimated coefficient, which typically corresponds to the treatment effect.

See Also

fast_logistic_regression_with_var_cpp for the underlying single-fit (no retry) backend and its variance-computation details; fast_logistic_regression_cpp for the full model documentation.

Examples

```
X = matrix(rnorm(100), 10, 10)
y = rbinom(10, 1, 0.5)
fast_logistic_regression_with_var(X, y)
```

---

fast_logistic_regression_with_var_cpp

Fast Logistic Regression with Targeted Variance (C++ Backend)

Description

Fits the same logistic regression model as fast_logistic_regression_cpp (see that page for the full model and log-odds-ratio interpretation) and additionally computes the variance of two coefficients — the caller-selected j-th coefficient and, separately, the 2nd coefficient (the package's usual treatment-effect position) — via a targeted diagonal-entry inversion of the working-weights Fisher information, rather than a full matrix inverse. Unlike fast_logistic_regression_cpp, maxit and tol are not exposed here: they are fixed internally at 100 iterations and 1e-8 tolerance.

Usage

```
fast_logistic_regression_with_var_cpp(
  X,
  y,
  j = 2L,
  warm_start_beta = NULL,
  smart_cold_start = FALSE,
  fixed_idx = NULL,
  fixed_values = NULL,
  optimization_alg = "irls",
  warm_start_weights = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

X	A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in X if desired.
y	A numeric vector of the response variable, expected to be binary (0 or 1).
j	1-based index (into X's columns) of the coefficient to compute ssq_b_j for. Defaults to 2.

`warm_start_beta`
: Optional starting values for coefficients β. If provided, `smart_cold_start` is ignored.

`smart_cold_start`
: Logical. If `TRUE` and no `warm_start_beta` is supplied, use an OLS-based initial guess rather than a zero cold start.

`fixed_idx`
: Optional integer indices of coefficients to hold fixed rather than estimate.

`fixed_values`
: Optional values to fix the parameters named by `fixed_idx` at.

`optimization_alg`
: Optimization algorithm: `"irls"` (default), `"lbfgs"`, or `"newton_raphson"`; see `.normalize_optimizer_algorithm`.

`warm_start_weights`
: Optional initial IRLS working weights for the first iteration.

`warm_start_fisher_info`
: Optional initial Fisher Information matrix to warm-start curvature information.

Details

Variance computation. The working-weights Fisher information $X^\top W X$ is restricted to the free (non-`fixed_idx`) parameters; `ssq_b_j` and `ssq_b_2` are each obtained via a single targeted diagonal-entry inversion (`compute_diagonal_inverse_entry()`) at the free-parameter position corresponding to `j` and to column 2, respectively — `NA` if the relevant coefficient is out of range or was itself fixed via `fixed_idx`. `ssq_b_2` is always computed (when `ncol(X) >= 2`) regardless of what `j` is, so a caller interested in the treatment effect's variance does not need to pass `j = 2` explicitly.

Value

A list with components `b`/`params` (estimated coefficients $\hat{\beta}$, identical to each other), `ssq_b_j`/`ssq_b_2` (the two targeted coefficient variances described in Details), `score` (the score vector at $\hat{\beta}$), `observed_information`/`fisher_information` (three aliases for the same working-weights curvature matrix, tagged `information_type = "fisher"`), `hessian` (the negative of that matrix), `neg_loglik`/`neg_ll` (aliases for the negative log-likelihood), `loglik` (its negation), `converged` (logical, `gradient_norm < tol`, uniform across optimizers), `num_iter`, `hit_iteration_cap` (logical, mutually exclusive with `converged`), and `gradient_norm`.

See Also

`fast_logistic_regression_cpp` for the estimate-only variant and the full model documentation.

`fast_negbin_regression`
Fast Negative Binomial Regression, Estimate-Only (R Wrapper)

Description

This function provides a fast implementation of mean/dispersion-parameterized negative binomial regression, wrapping a C++ backend (fast_neg_bin_cpp; see that page, and fast_dnbinom_mu_vec_cpp, for the full model). It returns point estimates only (no standard errors) — see fast_negbin_regression_with_var for the variance-computing counterpart. Columns 1 and 2 of X (conventionally the intercept and treatment indicator) are always kept; this wrapper adds automatic, silent handling of **rank-deficient or numerically unstable covariate sets** beyond those first two columns: (1) if warm_start_params is supplied, a single fit is attempted on the full matrix using the warm start, and its result is returned if successful; (2) otherwise, upfront, any covariate columns (3 onward) found rank-deficient by qr are dropped before the first fit attempt; (3) if the C++ fit still fails (e.g. an L-BFGS line-search failure), covariates are dropped one at a time, in reverse QR-pivot order (most redundant first), retrying after each drop, until the fit succeeds or only the intercept and treatment columns remain — at which point, if it still fails, this function stop()s with an explicit error rather than returning a corrupted fit.

Usage

```
fast_negbin_regression(
  X,
  y,
  optimization_alg = "lbfgs",
  warm_start_params = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

X
: A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in X if desired.

y
: A numeric vector of the response variable, representing count data.

optimization_alg
: Optimization algorithm: "lbfgs" (default) or "newton_raphson".

warm_start_params
: Optional starting values for coefficients and log_theta, passed straight through to the C++ backend for a single warm-started fit attempt (bypassing the QR-dropping retry sequence).

warm_start_fisher_info
: Optional initial Fisher information matrix, used only together with warm_start_params.

Value

A list containing the following components:

b
: A numeric vector of the estimated negative binomial regression coefficients $\hat{\beta}$ (and log_theta), from whichever (possibly covariate-reduced) fit in the retry sequence ultimately succeeded.

fisher_information
: The C++ backend's returned Fisher information matrix for the successful fit.

See Also

fast_negbin_regression_with_var for the variance-computing, similarly retry-hardened wrapper; fast_neg_bin_cpp for the underlying backend and full model documentation.

Examples

```
X = matrix(rnorm(100), 10, 10)
y = rpois(10, 2)
fast_negbin_regression(X, y)
```

---

fast_negbin_regression_with_var

Fast Negative Binomial Regression with Variance Calculation (R Wrapper)

Description

This function provides a fast implementation of negative binomial regression, wrapping a C++ backend (fast_neg_bin_with_var_cpp; see that page, and fast_dnbinom_mu_vec_cpp, for the full mean/dispersion negative-binomial model). Columns 1 and 2 of X (conventionally the intercept and treatment indicator) are always kept; this wrapper adds automatic, silent handling of **rank-deficient or numerically unstable covariate sets** beyond those first two columns: (1) upfront, any covariate columns (3 onward) found rank-deficient by qr are dropped before the first fit attempt; (2) if the C++ fit still fails (e.g. an L-BFGS line-search failure), covariates are dropped one at a time, in reverse QR-pivot order (most redundant first), retrying after each drop, until the fit succeeds or only the intercept and treatment columns remain — at which point, if it still fails, this function stop()s with an explicit error rather than returning a corrupted fit.

Usage

```
fast_negbin_regression_with_var(X, y, j = 2, optimization_alg = "lbfgs")
```

Arguments

X	A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in X if desired.
y	A numeric vector of the response variable, representing count data.
j	The index of the coefficient to compute the variance for. Defaults to 2.
optimization_alg	Optimization algorithm: "lbfgs" (default) or "newton_raphson".

Value

A list containing the following components:

b A numeric vector of the estimated negative binomial regression coefficients $\hat{\beta}$, from whichever (possibly covariate-reduced) fit in the retry sequence ultimately succeeded.

ssq_b_j The variance of the j-th estimated coefficient, computed by inverting the C++ backend's returned Hessian/Fisher-information matrix directly (via `solve`, not the backend's own `vcov`); `NA` if that inversion fails or `j` exceeds the number of columns remaining after covariate-dropping.

ssq_b_2 The variance of the second estimated coefficient specifically (typically the treatment effect), computed the same way, regardless of what `j` is.

See Also

`fast_neg_bin_with_var_cpp` for the underlying backend (no automatic rank-deficiency retry) and full model documentation; `fast_negbin_regression` for the estimate-only, similarly retry-hardened wrapper.

Examples

```
X = matrix(rnorm(100), 10, 10)
y = rpois(10, 2)
fast_negbin_regression_with_var(X, y)
```

---

fast_ols_cpp — *Fast Ordinary Least Squares (OLS) Regression, Estimate-Only (C++ Backend)*

Description

Solves the ordinary least squares normal equations $\hat{\beta} = (X^\top X)^{-1} X^\top y$ via Eigen's `LDLT` Cholesky decomposition of $X^\top X$; if that decomposition fails (e.g. X is rank-deficient), it falls back to a column-pivoted QR decomposition of X directly (`Eigen::ColPivHouseholderQR`), which returns a minimum-norm least-squares solution even when X is not full rank. Estimate-only: no standard errors or covariance matrix are computed, only the coefficient vector $\hat{\beta}$.

Usage

```
fast_ols_cpp(X, y, fixed_idx = NULL, fixed_values = NULL)
```

Arguments

`X`	A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.
`y`	A numeric vector of the (continuous) response variable.
`fixed_idx`	Optional integer vector of 1-indexed columns of `X` whose coefficients should be held fixed at `fixed_values` rather than estimated.
`fixed_values`	Optional numeric vector, parallel to `fixed_idx`, of the fixed coefficient values.

Value

A list containing the following component:

b A numeric vector of the estimated regression coefficients $\hat{\beta}$ (with any `fixed_idx` entries set to `fixed_values`); if the solve produces non-finite values, this is instead a vector of `NaN`.

Fixed (offset) coefficients

`fixed_idx` (1-indexed columns of `X`) and `fixed_values` optionally hold a subset of coefficients at caller-supplied constant values rather than estimating them: those columns' contribution $X_{\text{fixed}}\beta_{\text{fixed}}$ is subtracted out of `y` first, and only the remaining ("free") columns are fit by least squares; the fixed coefficients are then copied back into $\hat{\beta}$ unchanged. This is used, e.g., to fit a model with a known/offset intercept without re-estimating it.

See Also

`fast_ols_with_var_cpp` for the variance-computing counterpart.

`fast_ols_with_var_cpp` *Fast Ordinary Least Squares (OLS) Regression with Variance (C++ Backend)*

Description

As `fast_ols_cpp`, but additionally computes the classical OLS variance estimate $\hat{\sigma}^2 = \text{SSE}/(n-p)$ (with $\text{SSE} = y^\top y - \hat{\beta}^\top X^\top y$ on the fixed-parameter-adjusted response, and p the number of **free** — non-fixed — columns) and the sampling variance of two coefficients, $\widehat{\text{Var}}(\hat{\beta}_k) = \hat{\sigma}^2\,[(X^\top X)^{-1}]_{kk}$, obtained from the same Cholesky (`LDLT`) factorization used to solve for $\hat{\beta}$, without forming the full inverse matrix. If the `LDLT` decomposition fails (rank-deficient X), this function falls back to a QR solve exactly as `fast_ols_cpp` does, but in that case **no variance quantities are computed**: `ssq_b_j`, `ssq_b_2`, and `XtX` are omitted from the result and `converged` is `FALSE`.

Usage

```
fast_ols_with_var_cpp(X, y, j = 2L, fixed_idx = NULL, fixed_values = NULL)
```

Arguments

`X`	A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.
`y`	A numeric vector of the (continuous) response variable.
`j`	This function will compute the variance of the jth (1-indexed) coefficient estimator. Default is 2 (conventionally the treatment effect).
`fixed_idx`	Optional integer vector of 1-indexed columns of `X` whose coefficients should be held fixed at `fixed_values` rather than estimated.
`fixed_values`	Optional numeric vector, parallel to `fixed_idx`, of the fixed coefficient values.

Value

A list containing the following components (the last four only present when the `LDLT` solve succeeds):

b A numeric vector of the estimated regression coefficients $\hat{\beta}$.

converged `TRUE` if the `LDLT` solve succeeded, `FALSE` if the QR fallback was used.

sigma2_hat The estimated residual variance $\hat{\sigma}^2$.

XtX The (free-coefficient) $X^\top X$ matrix, expanded back to full $p \times p$ shape with zeros in the fixed-coefficient rows/columns.

ssq_b_j The variance of the j-th coefficient estimator, `NA` if `j` indexes a fixed coefficient.

ssq_b_2 The variance of the second coefficient estimator specifically (typically the treatment effect), regardless of what `j` is; equal to `ssq_b_j` when `j = 2`. `NA` if the second column is a fixed coefficient.

Fixed (offset) coefficients

`fixed_idx` (1-indexed columns of `X`) and `fixed_values` optionally hold a subset of coefficients at caller-supplied constant values rather than estimating them: those columns' contribution $X_{\text{fixed}}\beta_{\text{fixed}}$ is subtracted out of `y` first, and only the remaining ("free") columns are fit by least squares; the fixed coefficients are then copied back into $\hat{\beta}$ unchanged. This is used, e.g., to fit a model with a known/offset intercept without re-estimating it.

See Also

`fast_ols_cpp` for the estimate-only counterpart.

---

fast_poisson_regression_cpp

Fast Poisson Regression, Estimate-Only (C++ Backend)

Description

Fits a Poisson regression with the canonical log link, $Y_i \sim \text{Poisson}(\mu_i)$, $\mu_i = \exp(x_i^\top \beta)$ (with $\eta_i = x_i^\top \beta$ clamped above at 700 before exponentiating, to avoid overflow), by maximum likelihood. By default (`optimization_alg = "irls"`), fitting uses iteratively reweighted least squares: at each iteration the Fisher-scoring (Poisson canonical-link, so Fisher = observed) system $X^\top W X \, \delta = X^\top (y - \mu)$ is solved via `Eigen::LDLT`, with a backtracking step-halving line search (up to 10 halvings) accepting the step only if it does not increase the negative log-likelihood; convergence is declared when the score norm falls below `tol`. Passing `optimization_alg = "lbfgs"` or `"newton_raphson"` instead routes through the generic likelihood optimizer (`.normalize_optimizer_algorithm`) on the raw (non-IRLS) negative log-likelihood/gradient/Hessian.

Usage

```
fast_poisson_regression_cpp(
  X,
  y,
  warm_start_beta = NULL,
  smart_cold_start = FALSE,
  maxit = 100L,
  tol = 1e-8,
  fixed_idx = NULL,
  fixed_values = NULL,
  optimization_alg = "irls",
  warm_start_weights = NULL,
  warm_start_fisher_info = NULL,
  estimate_only = FALSE
)
```

Arguments

- `X` — A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.
- `y` — A numeric vector of the response variable, expected to be nonnegative-integer counts.
- `warm_start_beta` — Optional starting values for coefficients β. If provided, `smart_cold_start` is ignored.
- `smart_cold_start` — Logical. If `TRUE` and no `warm_start_beta` is supplied, use a Poisson-specific heuristic initial guess rather than a zero cold start.
- `maxit` — Maximum number of iterations. Defaults to 100.
- `tol` — Convergence tolerance. Defaults to 1e-8.
- `fixed_idx` — Optional integer indices of coefficients to hold fixed rather than estimate.
- `fixed_values` — Optional values to fix the parameters named by `fixed_idx` at.
- `optimization_alg` — Optimization algorithm: `"irls"` (default), `"lbfgs"`, or `"newton_raphson"`.
- `warm_start_weights` — Accepted but unused; see Details.
- `warm_start_fisher_info` — Optional initial curvature (information) matrix to warm-start the first iteration.
- `estimate_only` — If `TRUE`, skip computing `mu`, `XtWX`/`fisher_information`, and `score`, returning only `b`, `converged`, `num_iter`, `hit_iteration_cap`, and `gradient_norm`.

Value

A list containing the following components:

b A numeric vector of the estimated Poisson regression coefficients $\hat{\beta}$.

mu (omitted if `estimate_only = TRUE`) The fitted means $\hat{\mu}_i$.

XtWX, fisher_information (omitted if `estimate_only = TRUE`) Two aliases for the same curvature matrix $X^\top W X$ ($W = \mathrm{diag}(\hat{\mu}_i)$) at the fitted coefficients — the Fisher information, which for the canonical log link coincides with the observed information.

score (omitted if `estimate_only = TRUE`) The score vector $X^\top(y - \hat{\mu})$ at the fitted coefficients.

neg_ll (omitted if `estimate_only = TRUE`) The negative log-likelihood at the fitted coefficients.

converged A logical value indicating whether the final gradient norm was below `tol` (`gradient_norm < tol`); uniform across the `"irls"`/`"lbfgs"`/`"newton_raphson"` optimizers.

num_iter The number of iterations performed.

hit_iteration_cap A logical value, mutually exclusive with `converged`: `TRUE` iff the optimizer exhausted `maxit` iterations without meeting the gradient-norm convergence criterion.

gradient_norm The norm of the score vector at the returned coefficients.

Fixed parameters, warm starts

`fixed_idx` and `fixed_values` optionally hold a subset of coefficients fixed at caller-supplied constant values (their contribution is folded into the linear predictor as an offset) rather than estimated. `warm_start_beta` supplies starting coefficients directly; otherwise, if `smart_cold_start = TRUE`, a Poisson-specific heuristic start is used, and if `warm_start_fisher_info` is also supplied (or, absent that, when `smart_cold_start = TRUE`), it seeds the curvature estimate used for the very first IRLS step (or first quasi-Newton step, for the non-IRLS algorithms). `warm_start_weights` is accepted for interface parity with sibling functions but is **not consulted anywhere** in this function's fitting logic.

See Also

`fast_poisson_regression_weighted_cpp` for the observation-weighted variant; `fast_poisson_regression_with_var_` for the variance-computing variant; `fast_quasipoisson_regression_with_var_cpp` for the overdispersion-corrected variant.

`fast_poisson_regression_weighted_cpp` *Fast Weighted Poisson Regression (C++ Backend)*

Description

Fits the same Poisson log-link model as `fast_poisson_regression_cpp` (see that page for the full model and optimizer contract), with each observation's contribution to the log-likelihood, score, and IRLS working weights multiplied by a row weight `weights[i]`. Setting all weights to 1 recovers `fast_poisson_regression_cpp` exactly. Always fits with `estimate_only = FALSE` (there is no flag to skip the post-fit `mu`/ `XtWX`/`score` computation for this variant).

Usage

```
fast_poisson_regression_weighted_cpp(
  X,
  y,
  weights,
  warm_start_beta = NULL,
  smart_cold_start = FALSE,
  maxit = 100L,
  tol = 1e-8,
  fixed_idx = NULL,
  fixed_values = NULL,
  optimization_alg = "irls",
  warm_start_weights = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

`X` A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.

`y` A numeric vector of the response variable, expected to be nonnegative-integer counts.

`weights` A numeric vector of nonnegative weights, one per observation.

`warm_start_beta` Optional starting values for coefficients β. If provided, `smart_cold_start` is ignored.

`smart_cold_start` Logical. If `TRUE` and no `warm_start_beta` is supplied, use a Poisson-specific heuristic initial guess rather than a zero cold start.

`maxit` Maximum number of iterations. Defaults to 100.

`tol` Convergence tolerance. Defaults to 1e-8.

`fixed_idx` Optional integer indices of coefficients to hold fixed rather than estimate.

`fixed_values` Optional values to fix the parameters named by `fixed_idx` at.

`optimization_alg` Optimization algorithm: `"irls"` (default), `"lbfgs"`, or `"newton_raphson"`.

`warm_start_weights` Accepted but unused; see `fast_poisson_regression_cpp` Details.

`warm_start_fisher_info` Optional initial curvature (information) matrix to warm-start the first iteration.

Value

A list containing the following components:

b A numeric vector of the estimated Poisson regression coefficients $\hat{\beta}$.

mu The fitted means $\hat{\mu}_i$.

XtWX, fisher_information Two aliases for the same (weighted) curvature matrix $X^\top W X$, $W = \mathrm{diag}(\texttt{weights}_i \hat{\mu}_i)$, at the fitted coefficients.

score The weighted score vector at the fitted coefficients.

neg_ll The weighted negative log-likelihood at the fitted coefficients.

converged A logical value indicating whether the algorithm converged.

iterations The number of iterations performed.

gradient_norm The norm of the score vector at convergence.

See Also

fast_poisson_regression_cpp for the unweighted variant and full model documentation.

---

fast_poisson_regression_with_var_cpp
Fast Poisson Regression with Variance Calculation (C++ Backend)

Description

Fits the same Poisson log-link model as fast_poisson_regression_cpp (see that page for the full model and optimizer contract; always with `estimate_only = FALSE`), and additionally inverts the fitted Fisher information matrix (`Eigen::LDLT` on the free-coefficient submatrix, via `compute_diagonal_inverse_entry`) to report the variance of two coefficients.

Usage

```
fast_poisson_regression_with_var_cpp(
  X,
  y,
  j = 2L,
  warm_start_beta = NULL,
  smart_cold_start = FALSE,
  maxit = 100L,
  tol = 1e-8,
  fixed_idx = NULL,
  fixed_values = NULL,
  optimization_alg = "irls",
  warm_start_weights = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

`X`	A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.
`y`	A numeric vector of the response variable, expected to be nonnegative-integer counts.

`j` The 1-indexed coefficient whose variance to compute in `ssq_b_j`. Defaults to 2.

`warm_start_beta` Optional starting values for coefficients β. If provided, `smart_cold_start` is ignored.

`smart_cold_start` Logical. If `TRUE` and no `warm_start_beta` is supplied, use a Poisson-specific heuristic initial guess rather than a zero cold start.

`maxit` Maximum number of iterations. Defaults to 100.

`tol` Convergence tolerance. Defaults to 1e-8.

`fixed_idx` Optional integer indices of coefficients to hold fixed rather than estimate.

`fixed_values` Optional values to fix the parameters named by `fixed_idx` at.

`optimization_alg` Optimization algorithm: `"irls"` (default), `"lbfgs"`, or `"newton_raphson"`.

`warm_start_weights` Accepted but unused; see `fast_poisson_regression_cpp` Details.

`warm_start_fisher_info` Optional initial curvature (information) matrix to warm-start the first iteration.

Value

A list containing the following components:

b, params A numeric vector of the estimated Poisson regression coefficients $\hat{\beta}$ (both aliases of the same vector).

ssq_b_j The variance of the j-th coefficient estimator; `NA` if `j` indexes a fixed coefficient.

ssq_b_2 The variance of the second coefficient estimator specifically (typically the treatment effect), regardless of what `j` is; equal to `ssq_b_j` when `j = 2`. `NA` if the second column is a fixed coefficient.

mu The fitted means $\hat{\mu}_i$.

converged A logical value indicating whether the final gradient norm was below `tol` (`gradient_norm < tol`); uniform across the `"irls"`/`"lbfgs"`/`"newton_raphson"` optimizers.

num_iter The number of iterations performed.

score The score vector at the fitted coefficients.

observed_information, fisher_information, information Three aliases for the same $X^\top W X$ curvature matrix (`information_type` is always `"fisher"`).

hessian The negative of that same matrix (the actual Hessian of the log-likelihood).

neg_loglik, neg_ll The negative log-likelihood at the fitted coefficients (two aliases).

loglik The log-likelihood (`-neg_ll`), or `NA` if `neg_ll` is non-finite.

hit_iteration_cap A logical value, mutually exclusive with `converged`: `TRUE` iff the optimizer exhausted `maxit` iterations without meeting the gradient-norm convergence criterion.

gradient_norm The norm of the score vector at the returned coefficients.

See Also

fast_poisson_regression_cpp for the estimate-only variant and full model documentation; fast_quasipoisson_regress for the overdispersion-corrected variant.

---

fast_quasipoisson_regression_with_var_cpp

Fast Quasi-Poisson Regression with Variance Calculation (C++ Back-end)

Description

Fits the same Poisson log-link mean model as fast_poisson_regression_cpp (see that page for the full model and optimizer contract; point estimates $\hat{\beta}$ are identical to what that function would return), but instead of the plain Poisson Fisher-information-based variance, scales it by an estimated **overdispersion** parameter to obtain quasi-likelihood standard errors that are robust to variance-mean deviations from the strict Poisson assumption $\mathrm{Var}(Y_i) = \mu_i$. The dispersion is the Pearson-statistic-based moment estimator,

$$\hat{\phi} = \frac{1}{n-p} \sum_{i=1}^{n} \frac{(y_i - \hat{\mu}_i)^2}{\hat{\mu}_i},$$

computed only when the residual degrees of freedom $n - p > 0$; the reported coefficient variances are then $\widehat{\mathrm{Var}}(\hat{\beta}_k) = \hat{\phi}\,[(X^\top \hat{W} X)^{-1}]_{kk}$ (the ordinary Poisson Fisher information scaled by $\hat{\phi}$), matching the standard quasi-Poisson GLM correction (as in `stats::glm(family = quasipoisson())`). If $n \leq p$, or $\hat{\phi}$ comes out non-finite or non-positive, `ssq_b_j`/`ssq_b_2`/`dispersion` are left at their `NA` defaults (point estimates `b` and `mu` are still returned).

Usage

```
fast_quasipoisson_regression_with_var_cpp(
  X,
  y,
  j = 2L,
  warm_start_beta = NULL,
  smart_cold_start = FALSE,
  maxit = 100L,
  tol = 1e-8,
  fixed_idx = NULL,
  fixed_values = NULL,
  optimization_alg = "irls",
  warm_start_weights = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

- `X` A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in `X` if desired.
- `y` A numeric vector of the response variable, expected to be nonnegative-integer counts.
- `j` The 1-indexed coefficient whose variance to compute in `ssq_b_j`. Defaults to 2.
- `warm_start_beta` Optional starting values for coefficients β. If provided, `smart_cold_start` is ignored.
- `smart_cold_start` Logical. If `TRUE` and no `warm_start_beta` is supplied, use a Poisson-specific heuristic initial guess rather than a zero cold start.
- `maxit` Maximum number of iterations. Defaults to 100.
- `tol` Convergence tolerance. Defaults to 1e-8.
- `fixed_idx` Optional integer indices of coefficients to hold fixed rather than estimate.
- `fixed_values` Optional values to fix the parameters named by `fixed_idx` at.
- `optimization_alg` Optimization algorithm: `"irls"` (default), `"lbfgs"`, or `"newton_raphson"`.
- `warm_start_weights` Accepted but unused; see `fast_poisson_regression_cpp` Details.
- `warm_start_fisher_info` Optional initial curvature (information) matrix to warm-start the first iteration.

Value

A list containing the following components:

b A numeric vector of the estimated Poisson regression coefficients $\hat{\beta}$ (point estimates, unaffected by the dispersion correction).

ssq_b_j The dispersion-scaled variance of the j-th coefficient estimator; `NA` if `j` indexes a fixed coefficient or the dispersion estimate is unusable.

ssq_b_2 The dispersion-scaled variance of the second coefficient estimator specifically (typically the treatment effect), regardless of what `j` is; equal to `ssq_b_j` when `j = 2`.

dispersion The estimated Pearson-based overdispersion parameter $\hat{\phi}$, or `NA` if the residual degrees of freedom are not positive.

mu The fitted means $\hat{\mu}_i$.

converged A logical value indicating whether the algorithm converged.

iterations The number of iterations performed.

gradient_norm The norm of the score vector at convergence.

See Also

`fast_poisson_regression_with_var_cpp` for the plain (non-overdispersion-corrected) variance variant; `fast_poisson_regression_cpp` for the estimate-only variant and full mean-model documentation.

fast_weibull_regression

Fast Weibull AFT Regression (R Wrapper: Rcpp Backend or **survival***)*

Description

Fits the Weibull accelerated failure time model documented in full at fast_weibull_regression_general_cpp ($\log T_i = \eta_i + \sigma W_i$, $\eta_i = x_i^\top \beta$, right-censoring only), via either that C++ backend (use_rcpp = TRUE, the default) or survreg with dist = "weibull" (use_rcpp = FALSE) as a fallback/cross-check implementation.

Usage

```
fast_weibull_regression(
  y,
  dead,
  X,
  use_rcpp = TRUE,
  estimate_only = FALSE,
  optimization_alg = "lbfgs",
  warm_start_params = NULL,
  warm_start_fisher_info = NULL
)
```

Arguments

y	Observed survival/censoring times (must be positive).
dead	Event indicator: 1 for an exactly observed event, 0 for right-censored (survival known only to exceed y[i]).
X	A numeric matrix of predictor variables. It is assumed that an intercept column (e.g., a column of ones) is already included in X if desired.
use_rcpp	Logical. If TRUE (default), use the optimized Rcpp implementation. If FALSE, use survreg.
estimate_only	Logical. If TRUE, skip variance-covariance matrix calculation for speed. Only has an effect when use_rcpp = TRUE.
optimization_alg	Optimization algorithm: "lbfgs" (default) or "newton_raphson". Only has an effect when use_rcpp = TRUE.
warm_start_params	Optional starting values for $[\beta, \log \sigma]$. Only has an effect when use_rcpp = TRUE.
warm_start_fisher_info	Optional initial curvature (Fisher/observed information) matrix. Only has an effect when use_rcpp = TRUE.

Details

When use_rcpp = TRUE, an intercept column is prepended to X automatically if not already present (detected as a first column of all 1s), fitting always starts from a zero cold start (smart_cold_start = FALSE is hardcoded, regardless of whether warm_start_params is supplied), and a non-converged C++ fit is escalated to an R-level stop() rather than returned silently.

When use_rcpp = FALSE, any existing intercept column is stripped and survreg is left to add its own; remaining covariate columns are first passed through drop_linearly_dependent_cols to remove collinear columns before fitting (silently — no error or warning is raised for dropped columns). estimate_only, optimization_alg, warm_start_params, and warm_start_fisher_info have **no effect** on this path — survreg always computes the full variance-covariance matrix, and std_errs (from sqrt(diag(vcov))) is included only in this path's return value, not the Rcpp path's. Both non-finite coefficients and (unless estimate_only = TRUE, which is ignored on this path regardless) non-finite variance-covariance entries from survreg are escalated to an R-level stop().

Value

A list containing the following components:

coefficients A numeric vector of the estimated Weibull regression coefficients $\hat{\beta}$, including the intercept.

log_sigma The logarithm of the fitted scale parameter $\hat{\sigma}$ of the Weibull AFT distribution.

vcov The variance-covariance matrix of the estimated coefficients, or NULL if estimate_only = TRUE (Rcpp path only).

neg_log_lik (Rcpp path only) The negative log-likelihood at the fitted parameters.

fisher_information (Rcpp path only) The observed information matrix, or NULL if estimate_only = TRUE.

std_errs (**survival** path only) The coefficient standard errors, sqrt(diag(vcov)).

See Also

fast_weibull_regression_general_cpp for the full model documentation and Rcpp backend contract.

Examples

```
X = matrix(rnorm(500), 100, 5)
y = runif(100)
dead = rbinom(100, 1, 0.5)
fast_weibull_regression(y, dead, X)
```

generate_covariate_dataset

Generate Synthetic Simulation Covariates and Continuous Response

Description

A helper function to generate synthetic covariates and a latent continuous response identical to the logic used within `SimulationFramework`. Covariates may be supplied directly via `X_mat` or drawn randomly via `cov_draw_method`; exactly one of the two must be non-`NULL`.

Usage

```
generate_covariate_dataset(
  n,
  p,
  cond_exp_func_model = c("linear", "nonlinear"),
  norm_sq_beta_vec = 1,
  X_mat = NULL,
  cov_draw_method = stats::rnorm,
  cov_draw_method_args = list(mean = 0, sd = 1)
)
```

Arguments

`n` Integer. Sample size (number of rows).

`p` Integer. Number of covariates (number of columns).

`cond_exp_func_model` Character scalar. Either `"linear"` (latent response is a weighted linear combination of covariates) or `"nonlinear"` (Friedman 1991 function applied to the first five covariates; requires `p >= 5`).

`norm_sq_beta_vec` Positive numeric scalar. The desired squared Euclidean norm of the coefficient vector, i.e. `sum(beta^2)`. The coefficient vector (or the overall Friedman scale) is rescaled so that this quantity equals `norm_sq_beta_vec`. Default `1`.

`X_mat` Numeric matrix of dimensions `n x p`, or `NULL` (default). When supplied, this matrix is used directly as the covariate matrix and `cov_draw_method` must be `NULL`.

`cov_draw_method` A function used to draw `n * p` i.i.d. covariate values, or `NULL`. The function must accept the number of draws as its first positional argument followed by any named arguments in `cov_draw_method_args`. Default `stats::rnorm`. Must be `NULL` when `X_mat` is supplied.

`cov_draw_method_args` Named list of additional arguments forwarded to `cov_draw_method` beyond the sample-size first argument. Default is `list(mean = 0, sd = 1)`.

Details

Two conditional-expectation models are supported, both rescaled so that the latent response `y_cont` has a controlled signal magnitude:

`"linear"` $y_i = x_i^\top \beta$, with β a fixed evenly spaced sequence from 1 to -1 across the p covariates (`seq(1, -1, length.out = p)`) — i.e. the first covariate gets the strongest positive effect, the last the strongest negative effect, and (for even p) covariates near the middle get effects near zero — rescaled so that $\|\beta\|_2^2 =$ `norm_sq_beta_vec`.

`"nonlinear"` The Friedman (1991) MARS benchmark function applied to the first five covariates (requires $p \geq 5$; covariates beyond the fifth do not enter the response at all),

$$y_i = c\left(10\sin(\pi x_{i1}x_{i2}) + 20(x_{i3} - 0.5)^2 + 10x_{i4} + 5x_{i5}\right),$$

with the overall scale c chosen so that $c^2 \sum_k \beta_{\text{friedman},k}^2 =$ `norm_sq_beta_vec` for the nominal term coefficients $\beta_{\text{friedman}} = (10, 20, 10, 5)$ (the $\sin$ term's coefficient is not part of this nominal vector, so the realized $\|c \cdot (10, 20, 10, 5)\|_2^2$ may not exactly equal `norm_sq_beta_vec` once the $\sin$ term's own variance is accounted for — `norm_sq_beta_vec` calibrates the linear-in-covariate part of the scale, not the full response variance). This function assumes each covariate lies roughly in $[-1, 1]$, matching Friedman's original specification; covariates drawn or supplied on a very different scale will not reproduce the intended signal shape.

Value

A list with two elements: `X` (a data frame of covariates) and `y_cont` (a numeric vector of the latent continuous response).

References

Friedman, J. H. (1991). "Multivariate Adaptive Regression Splines." *The Annals of Statistics*, 19(1), 1-67, doi:10.1214/aos/1176347963, for the nonlinear benchmark function used when `cond_exp_func_model = "nonlinear"`.

Examples

```
generate_covariate_dataset(n = 10, p = 5)
```

`get_bootstrap_dispatch_policy`

Get the default bootstrap dispatch policy

Description

Returns EDI's built-in policy table, consulted by the internal (non-exported) dispatcher `edi_bootstrap_dispatch_policy()` for choosing which bootstrap confidence-interval type — `"bca"` (bias-corrected and accelerated) or `"percentile"` — an inference class uses by default.

Usage

```
get_bootstrap_dispatch_policy()
```

Details

The dispatcher resolves a type for a given inference-class name (and, if available, the fitted inference object) in this precedence order, returning the first match:

1. If the object's experimental design class matches a key of `design_class_overrides` (via `is`), and the inference class name matches one of that design's named regular-expression patterns, use the associated type.
2. Otherwise, if the inference class name matches one of `inference_class_overrides`'s named regular-expression patterns (checked in list order, first match wins), use the associated type.
3. Otherwise, fall back to `default_type` (`"bca"`).

Whichever type is resolved by that process, a final safety check applies: if the resolved type is `"bca"` and the fitted object reports (via its private `jackknife_block_size_gt_one_unsupported()` method) that BCa's required jackknife computation is unsupported for its current data (e.g. a block size greater than 1), the type is silently downgraded to `"percentile"` instead. This override table exists because BCa is the generally preferred default (it corrects for both bias and skewness in the bootstrap distribution), but is empirically unreliable or computationally unsupported for specific inference/ design class combinations — the `"percentile"` overrides listed here were added as those cases were identified, not derived from a general rule.

Value

A named list describing the default bootstrap type configuration, with components `default_type` (the fallback type, `"bca"`), `inference_class_overrides` (a named character vector: regular-expression pattern names to bootstrap-type values, matched against the inference class name), and `design_class_overrides` (a named list keyed by experimental design class name, each value itself a named character vector of pattern-to-type overrides scoped to that design).

See Also

`get_parallel_dispatch_policy` for the analogous policy controlling forced-serial dispatch; `get_optimization_dispatc` for the analogous policy controlling default optimizer algorithm choice.

Examples

```
get_bootstrap_dispatch_policy()
```

get_cold_start_dispatch_policy

Get the default cold-start dispatch policy

Description

Returns EDI's built-in policy table, consulted by the internal (non-exported) dispatcher `edi_cold_start_dispatch_policy()` for the `smart_cold_start` default used by each inference class's C++ model-fitting backend. A `TRUE` entry means the solver initializes via an OLS (or otherwise model-appropriate heuristic) warm-up before iterating; `FALSE` means a plain zero-vector cold start. Benchmarks show the OLS warm-up is net-negative for logistic and Poisson IRLS at typical trial sizes (the one extra OLS solve costs more than the IRLS iterations it saves), so those families — along with several G-computation-based incidence/proportion inference classes — default to `FALSE` here.

Usage

```
get_cold_start_dispatch_policy()
```

Details

The dispatcher checks the inference class name against `inference_class_overrides`'s named regular-expression patterns in list order, returning the associated logical value at the first match; if none match, it falls back to `default` (`TRUE`). Unlike `get_bootstrap_dispatch_policy`, there is no separate design-class-scoped override table here — only a single flat pattern list.

Value

A named list with `default` (logical, the fallback when no override pattern matches; `TRUE` in the built-in policy) and `inference_class_overrides` (a named logical vector: regular-expression pattern names to `TRUE`/`FALSE` values, matched against the inference class name). These `TRUE`/`FALSE` defaults are empirical performance judgments computed on the maintainer's machine, not correctness facts — the same heuristic can be net-positive or net-negative depending on your hardware's core count, cache sizes, and BLAS backend. Run `tune_EDI_for_this_machine` to re-measure this axis on your own machine and persist any better setting it finds.

See Also

`get_bootstrap_dispatch_policy` and `get_optimization_dispatch_policy` for the analogous policies controlling bootstrap CI type and default optimizer algorithm; `set_cold_start_dispatch_policy` to override this policy at runtime; `tune_EDI_for_this_machine` to re-benchmark it on your own hardware.

Examples

```
get_cold_start_dispatch_policy()
```

get_local_EDI_optimization

Show this machine's saved EDI tuning, if any

Description

Reads the per-user config file written by `tune_EDI_for_this_machine` and returns it as an `EDILocalMachineTuning` object (whose print method shows when and how it was produced, the hardware fingerprint it was measured on, and every policy deviation it stores). Does **not** apply anything – application happens inside `tune_EDI_for_this_machine()` itself and at package load.

Usage

```
get_local_EDI_optimization()
```

Value

Invisibly, the saved `EDILocalMachineTuning` object, or `NULL` (with a message) if no valid saved tuning exists.

See Also

`tune_EDI_for_this_machine`, `clear_local_EDI_optimization`.

Examples

```
get_local_EDI_optimization()
```

get_optimization_dispatch_policy

Get the default optimization dispatch policy

Description

Returns EDI's built-in policy table, consulted by the internal (non-exported) dispatcher `edi_optimization_dispatch_polic` for choosing which optimization algorithm (`"newton_raphson"`, `"lbfgs"`, or `"irls"`) an inference class's C++ model-fitting backend uses by default.

Usage

```
get_optimization_dispatch_policy()
```

Details

The dispatcher checks the inference class name against `inference_class_overrides`'s named regular-expression patterns in list order, returning the associated algorithm string at the first match; if none match, it falls back to `default_alg` (`"newton_raphson"`). Unlike `get_bootstrap_dispatch_policy`, there is no separate design-class-scoped override table here — only a single flat pattern list. The built-in overrides are empirical, chosen per model family based on which algorithm converges fastest/most reliably for that likelihood surface in practice — e.g. plain-vanilla generalized linear models with a canonical or near-canonical link (Poisson, quasi-Poisson, robust Poisson, various incidence models) default to `"irls"`, most non-canonical-link and ordinal/survival models default to `"lbfgs"`, and stratified Cox PH and most matched (`KK*GLMM`-adjacent) models default to `"newton_raphson"`.

Value

A named list with components `default_alg` (the fallback algorithm, `"newton_raphson"` in the built-in policy) and `inference_class_overrides` (a named character vector: regular-expression pattern names to algorithm-name values, matched against the inference class name). Which algorithm converges fastest/most reliably per family is partly a hardware fact (relative cost of Hessian solves vs. L-BFGS iterations depends on BLAS and cache), so these defaults, computed on the maintainer's machine, are not necessarily optimal on yours. Run `tune_EDI_for_this_machine` to re-measure this axis on your own machine — it will only switch a family's algorithm when the candidate converges on every benchmark replicate, never trading speed for a convergence failure.

See Also

`get_bootstrap_dispatch_policy` and `get_cold_start_dispatch_policy` for the analogous policies controlling bootstrap CI type and cold-start behavior; `.normalize_optimizer_algorithm` for how a resolved algorithm string is validated/normalized before being passed to a C++ backend; `tune_EDI_for_this_machine` to re-benchmark this policy on your own hardware.

Examples

```
get_optimization_dispatch_policy()
```

`get_parallel_dispatch_policy`

Get the default parallel dispatch policy

Description

Returns EDI's built-in **blocklist-first** policy table, consulted by an internal (non-exported) dispatcher, for deciding whether a given `"bootstrap"` or `"rand_ci"` (randomization confidence interval) operation is forced to run **serially** rather than in parallel, for a given inference class and response type. "Blocklist-first" means every operation is parallel-eligible by default; only combinations explicitly matched below are forced serial.

Usage

```
get_parallel_dispatch_policy()
```

Details

For a given `operation` (`"bootstrap"` or `"rand_ci"`), the dispatcher looks up that operation's sublist (e.g. `bootstrap`) and forces serial execution if either the inference class name matches any of `serial_inference_class_patterns` (regular expressions, PCRE-flavored — `serial_inference_class_patterns` supports lookahead, e.g. the built-in `"^InferenceSurvival(?!.*KK)"` pattern matches non-KK survival inference classes but excludes matched-design (KK) survival variants) or the response type matches any of `serial_response_types` exactly. In the built-in policy, incidence-response inference generally runs serially for both operations (its bootstrap/randomization resampling is not currently parallel-safe or not worth parallelizing at typical trial sizes), and non-KK survival inference and `InferenceAllKKWilcoxIVWC` are additionally forced serial for bootstrapping specifically.

Value

A named list with two components, `bootstrap` and `rand_ci`, each itself a list with `serial_inference_class_patterns` (a character vector of PCRE-flavored regular expressions matched against the inference class name) and `serial_response_types` (a character vector of exact response-type strings) — any match in either forces that operation to run serially rather than in parallel. Unlike `get_cold_start_dispatch_policy`, `get_warm_start_dispatch_policy`, and `get_optimization_dispatch_policy`, this table is a **correctness** fact, not a performance one — every entry exists because that operation is not currently parallel-safe for that class or response type, not because it happens to be slower in parallel. `tune_EDI_for_this_machine` benchmarks a related but separate question — at what sample size parallel execution starts to beat serial, and which core count wins — and will never propose unserializing anything named here.

See Also

`get_bootstrap_dispatch_policy`, `get_cold_start_dispatch_policy`, and `get_optimization_dispatch_policy` for the analogous policies controlling bootstrap CI type, cold-start behavior, and default optimizer algorithm (none of which control parallel-vs-serial dispatch); `tune_EDI_for_this_machine` for the machine-dependent parallel-crossover/core-count benchmark this blocklist constrains but is never itself a target of.

Examples

```
get_parallel_dispatch_policy()
```

`get_warm_start_dispatch_policy`

Get the default warm-start dispatch policy

Description

Returns EDI's built-in policy table for choosing whether **warm starts** (reusing a previous fit's parameters/curvature to seed the next fit, e.g. across bootstrap or randomization replicates) are enabled for a given inference class during a given resampling or simulation `operation` (one of `"jackknife"`, `"non_param_boot"`, `"bayesian_boot"`, `"param_boot"`, or `"rand"`).

Usage

```
get_warm_start_dispatch_policy()
```

Details

The internal (non-exported) dispatcher, `edi_warm_start_dispatch_policy(inference_class, operation, n)`, consults this table's `default` plus, per operation, two override layers: `inference_class_overrides` (a sample-size-independent pattern table, as before) and `n_conditioned_overrides` (a list of `list(pattern, value, n_min, n_max)` rules — each pattern only applies when the current sample size `n` falls in `[n_min, n_max)` — encoding empirical findings like "the extra bookkeeping only pays off once resampling is expensive enough per replicate, so disable below n=200/500/1000 for these families"). Both layers are returned by this function and are both reachable via `set_warm_start_dispatch_policy`.

Value

A named list with `default` (logical, `TRUE` in the built-in policy) and one component per operation (`jackknife`, `non_param_boot`, `bayesian_boot`, `param_boot`, `rand`), each itself a list containing `inference_class_overrides` (a named logical vector: regular-expression pattern names to `TRUE`/`FALSE` values, matched against the inference class name, applied regardless of sample size) and `n_conditioned_overrides` (a list of `list(pattern, value, n_min, n_max)` rules, applied only when `n` is supplied and falls in `[n_min, n_max)`); see Details. Like the cold-start table, every `TRUE`/`FALSE` entry here (including the n-thresholds) is an empirical performance judgment computed on the maintainer's machine, not a correctness fact. Run `tune_EDI_for_this_machine` to re-measure this axis, per resampling operation and sample size, on your own machine.

See Also

`get_cold_start_dispatch_policy` for the analogous (simpler, single-layer) policy governing the initial cold-start heuristic rather than cross-replicate warm-starting; `set_warm_start_dispatch_policy` to override this table at runtime; `tune_EDI_for_this_machine` to re-benchmark it on your own hardware.

Examples

```
get_warm_start_dispatch_policy()
```

`InferenceAllKKMeanDiffIVWC`

Mean-Difference IVWC Inference for KK Matching-on-the-Fly Designs

Description

Fits a compound (inverse-variance-weighted combination, "IVWC") mean-difference estimator of the treatment effect for continuous responses under a `DesignSeqOneByOne`-family KK matching-on-the-fly design (see `DesignSeqOneByOneKK14` and `DesignSeqOneByOneKK21`). Such a design produces two structurally different kinds of subjects: subjects successfully matched into pairs during the sequential design, and unmatched "reservoir" subjects randomized independently. This estimator combines both:

$$\hat{\beta}_T = w^* \bar{d} + (1 - w^*)\, \bar{r}, \qquad w^* = \frac{\widehat{\mathrm{Var}}(\bar{r})}{\widehat{\mathrm{Var}}(\bar{r}) + \widehat{\mathrm{Var}}(\bar{d})},$$

where $\bar{d}$ is the mean within-pair (treated minus control) difference among matched subjects and $\bar{r}$ is the treated-minus-control difference in means among reservoir subjects, weighted inversely by their estimated variances (see `$compute_asymp_confidence_interval()` for the full variance formula and the fallback behavior when only one of the two sub-estimates is usable). Inference is Wald-only: this class has no likelihood tier (`likelihood_tier = "none"`) and provides asymptotic Wald, randomization, and bootstrap (including Bayesian bootstrap) confidence intervals and p-values, but no score/likelihood-ratio/gradient tests.

Initialize KK IVWC mean-difference inference.

Computes the compound IVWC (inverse-variance-weighted compound) mean-difference point estimate $\hat{\beta}_T$: the inverse-variance-weighted combination $w^* \bar{d} + (1 - w^*)\, \bar{r}$ of the matched-pair mean within-pair difference $\bar{d}$ and the reservoir treated-minus-control mean difference $\bar{r}$, falling back to whichever of the two is usable if the other is not (see `$compute_asymp_confidence_interval()` for the full weighting formula and usability conditions).

Computes a $1 - \alpha$ level frequentist confidence interval for the compound IVWC (inverse-variance-weighted compound) mean-difference estimator $\hat{\beta}_T$.

Computes a two-sided **Wald** p-value for the compound IVWC mean-difference estimator $\hat{\beta}_T$ testing $H_0 : \beta_T = \texttt{delta}$, using the same asymptotically-normal point estimate and standard error ($z = (\hat{\beta}_T - \texttt{delta}) / \widehat{\mathrm{SE}}(\hat{\beta}_T)$) that `$compute_asymp_confidence_interval()` inverts to form its interval — see that method's documentation for the full inverse-variance-weighted combination formula. This class has no likelihood tier (`likelihood_tier = "none"`), so no score, likelihood-ratio, or gradient test is available here; this is a plain Wald test, not a likelihood-backed one.

Details

The point estimate combines two sub-estimates depending on which are usable: the mean within-pair difference among matched subjects, $\bar{d}$, with estimated variance $\widehat{\mathrm{Var}}(\bar{d})$, and the treated-minus-control difference in means among reservoir (unmatched) subjects, $\bar{r}$, with estimated variance $\widehat{\mathrm{Var}}(\bar{r})$. When both are usable (at least 2 matched pairs and at least 2 treated/2 control reservoir

subjects, with finite positive variance estimates), they are combined by classical inverse-variance weighting,

$$\hat{\beta}_T = w^* \bar{d} + (1 - w^*)\, \bar{r}, \qquad w^* = \frac{\widehat{\mathrm{Var}}(\bar{r})}{\widehat{\mathrm{Var}}(\bar{r}) + \widehat{\mathrm{Var}}(\bar{d})},$$

with combined variance the standard inverse-variance-pooled form $\widehat{\mathrm{Var}}(\hat{\beta}_T) = \left(\widehat{\mathrm{Var}}(\bar{r})^{-1} + \widehat{\mathrm{Var}}(\bar{d})^{-1}\right)^{-1} = \widehat{\mathrm{Var}}(\bar{r})\, \widehat{\mathrm{Var}}(\bar{d}) / \left(\widehat{\mathrm{Var}}(\bar{r}) + \widehat{\mathrm{Var}}(\bar{d})\right)$. If only one of the two sub-estimates is usable (e.g. the reservoir is empty or degenerate, or no pairs matched), $\hat{\beta}_T$ and its variance fall back to that sub-estimate alone. The compound estimator is treated as asymptotically normal, so the interval is $\hat{\beta}_T \pm z_{1-\alpha/2}\sqrt{\widehat{\mathrm{Var}}(\hat{\beta}_T)}$ (or a t-based critical value, depending on `private$compute_z_or_t_ci_from_s_and_df`'s degrees-of-freedom resolution).

Value

The setting-appropriate (see description) numeric estimate of the treatment effect

A (1 - alpha)-sized frequentist confidence interval for the treatment effect

The approximate frequentist p-value

Legacy status

Legacy class. Not fully tested in `comprehensive_tests.R`; prefer a more actively maintained KK continuous-response inference class (e.g. `InferenceContinKKOLSIVWC`) for new analyses unless this specific unadjusted mean-difference estimator is required.

Super class

`Inference` -> `InferenceAllKKMeanDiffIVWC`

Methods

Public methods:

- `InferenceAllKKMeanDiffIVWC$new()`
- `InferenceAllKKMeanDiffIVWC$compute_estimate()`
- `InferenceAllKKMeanDiffIVWC$compute_asymp_confidence_interval()`
- `InferenceAllKKMeanDiffIVWC$compute_asymp_two_sided_pval()`
- `InferenceAllKKMeanDiffIVWC$clone()`

`InferenceAllKKMeanDiffIVWC$new()`:

Usage:

```
InferenceAllKKMeanDiffIVWC$new(
  des_obj,
  verbose = FALSE,
  harden = TRUE,
  model_formula = NULL,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A KK matching-on-the-fly design object.

`verbose` Whether to print progress messages.

`harden` Whether to use hardened model-matrix fitting.

`model_formula` Optional formula for covariate adjustment.

`smart_cold_start_default` Whether to use smart cold start values.

`InferenceAllKKMeanDiffIVWC$compute_estimate()`:

Usage:

```
InferenceAllKKMeanDiffIVWC$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If `TRUE`, compute only the point estimate $\hat{\beta}_T$ and skip the variance-component computations needed for confidence intervals or p-values (faster when only the point estimate is needed).

`InferenceAllKKMeanDiffIVWC$compute_asymp_confidence_interval()`:

Usage:

```
InferenceAllKKMeanDiffIVWC$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`InferenceAllKKMeanDiffIVWC$compute_asymp_two_sided_pval()`:

Usage:

```
InferenceAllKKMeanDiffIVWC$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null difference to test against. For any treatment effect at all this is set to zero (the default).

`InferenceAllKKMeanDiffIVWC$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceAllKKMeanDiffIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A., and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388, doi:10.1111/biom.12148, for the KK matching-on-the-fly design this estimator targets, and for the inverse-variance combination of matched-pair and reservoir estimates.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 6, response_type = "continuous")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2 : 10])
seq_des$add_all_subject_responses(c(4.71, 1.23, 4.78, 6.11, 5.95, 8.43))

seq_des_inf = InferenceAllKKMeanDiffIVWC$
  new(seq_des)
seq_des_inf$compute_estimate()
seq_des_inf$compute_asymp_confidence_interval()
seq_des_inf$compute_asymp_two_sided_pval()
```

InferenceAllKKWilcoxIVWC

Non-parametric Wilcoxon-based Compound Inference for KK Matching-on-the-Fly Designs

---

Description

Fits a non-parametric, rank-based compound (inverse-variance-weighted, IVWC) estimator of the treatment effect under a DesignSeqOneByOne-family KK matching-on-the-fly design (see DesignSeqOneByOneKK14). For matched pairs, the sub-estimate $\hat{\beta}_m$ is the **Hodges-Lehmann** estimate from a Wilcoxon signed-rank test on the within-pair differences (the median of the Walsh averages $(d_i + d_j)/2$); for reservoir (unmatched) subjects, $\hat{\beta}_r$ is the Hodges-Lehmann estimate from a Wilcoxon rank-sum (Mann-Whitney U) test on treated-vs-control reservoir responses (the median of all pairwise differences). The two are combined by classical inverse-variance weighting,

$$\hat{\beta}_T = w^* \hat{\beta}_m + (1 - w^*)\, \hat{\beta}_r, \qquad w^* = \frac{\widehat{\mathrm{Var}}(\hat{\beta}_r)}{\widehat{\mathrm{Var}}(\hat{\beta}_r) + \widehat{\mathrm{Var}}(\hat{\beta}_m)},$$

with variance the standard inverse-variance-pooled form (see `$compute_estimate()`'s method-level documentation for the full formula and fallback behavior when only one sub-estimate is usable). Because it is built from Hodges-Lehmann/Wilcoxon estimators rather than sample means, this method is robust to outliers and does not assume a specific parametric distribution for the response — but it does not currently support censored survival data or incidence (binary) responses (see `$initialize()`), and its jackknife methods all report explicit non-estimability rather than computing a (statistically unreliable) delete-1 jackknife of the Hodges-Lehmann functional.

Initialize KK inverse-variance combined Wilcoxon inference and prepare matched/reservoir rank-based components used by InferenceAllKKWilcoxIVWC. Requires `des_obj` to be a KK matching-on-the-fly-capable design (`des_obj$is_a_kk_matching_capable()`); errors otherwise. Also rejects `response_type = "incidence"` (a message-and-error recommends a compound mean-difference

or conditional-logistic estimator instead — rank-based methods are not well suited to binary outcomes) and rejects censored survival data (recommends a restricted-mean or Cox-based method instead, since this estimator has no censoring handling). Legal `response_type` values are `"continuous"`, `"count"`, `"proportion"`, `"survival"` (uncensored only), and `"ordinal"`.

Returns the estimated treatment effect: an inverse-variance-weighted compound (IVWC) of two Hodges-Lehmann median-shift estimates.

Computes a $1 - \alpha$ level confidence interval for the compound Hodges-Lehmann treatment effect estimator. Although each sub-estimate is itself derived from a non-parametric rank test, the inverse-variance-weighted **combination** $\hat{\beta}_T$ (see `$compute_estimate()` for the full formula) is treated as asymptotically normal, so the interval is $\hat{\beta}_T \pm z_{1-\alpha/2}\sqrt{\widehat{\mathrm{Var}}(\hat{\beta}_T)}$ (or a t-based critical value, depending on `private$compute_z_or_t_ci_from_s_and_df`'s degrees-of-freedom resolution).

Compute the KK Wilcoxon compound two-sided p-value testing $H_0 : \beta_T =$ `delta`, from the same asymptotically-normal compound estimate/variance ($z = (\hat{\beta}_T - \texttt{delta})/\widehat{\mathrm{SE}}(\hat{\beta}_T)$) that `$compute_asymp_confidence_inter` inverts to form its interval — see that method's documentation, and `$compute_estimate()`, for the compound Hodges-Lehmann estimator's full formula. Only `delta = 0` is currently supported: a non-zero null shift raises an error (when assertions are enabled) rather than testing it, because the underlying Wilcoxon tests' null-shift handling has not been extended to the compound combined estimator. See related simple Wilcoxon behavior in `InferenceAllSimpleWilcox`.

Reports the jackknife point-estimate as explicitly non-estimable for this compound Hodges-Lehmann estimator, rather than computing a leave-one-out jackknife. Deletion-based (jackknife) resampling of a Hodges-Lehmann/Wilcoxon-derived statistic is known to behave poorly — the median-of-Walsh-averages functional is not smooth enough for the delete-1 jackknife's linear-approximation machinery to be reliable at the small matched-pair/reservoir sample sizes typical of KK designs, and combining two already-jackknife-unstable sub-estimates compounds the problem. This method exists purely to record that unavailability (via `private$cache_nonestimable_estimate()`) rather than silently returning a misleading number; see `InferenceJackknife` for the shared jackknife contract this method participates in.

Reports the jackknife bias-correction estimate as non-estimable for this Wilcoxon compound estimator, for the same reason as `$compute_jackknife_estimate()` (the Hodges-Lehmann functional is not smooth enough for the delete-1 jackknife); see `InferenceJackknife` for the shared jackknife contract.

Reports the jackknife standard error as non-estimable for this Wilcoxon compound estimator, for the same reason as `$compute_jackknife_estimate()`; see `InferenceJackknife` for the shared jackknife contract.

Reports the jackknife-Wald p-value as non-estimable here, for the same reason as `$compute_jackknife_estimate()`; see `InferenceJackknife`.

Reports the jackknife-Wald confidence interval as non-estimable here, for the same reason as `$compute_jackknife_estimat` see `InferenceJackknife`.

Details

For matched pairs, $\hat{\beta}_m$ is the Hodges-Lehmann estimate from a Wilcoxon **signed-rank** test on the within-pair differences (`stats::wilcox.test(diffs, conf.int = TRUE)`'s `estimate`, the median of the Walsh averages $(d_i + d_j)/2$), with normal-model calibrated variance $(2\pi/3)\,\mathrm{Var}(\mathrm{Walsh})/m$. For reservoir (unmatched) subjects, $\hat{\beta}_r$ is the Hodges-Lehmann estimate from a Wilcoxon **rank-sum** test between treated and control reservoir responses (median of all pairwise differences $y_{T,i} -$

$y_{C,j}$), with normal-model calibrated variance $\pi \operatorname{Var}(Y_T - Y_C)(n_T + n_C)/(6n_T n_C)$. When both sub-estimates are usable, the compound estimate is the inverse-variance-weighted combination

$$\hat{\beta}_T = w^* \hat{\beta}_m + (1 - w^*)\, \hat{\beta}_r, \qquad w^* = \frac{\widehat{\operatorname{Var}}(\hat{\beta}_r)}{\widehat{\operatorname{Var}}(\hat{\beta}_r) + \widehat{\operatorname{Var}}(\hat{\beta}_m)},$$

with combined variance $\widehat{\operatorname{Var}}(\hat{\beta}_r)\, \widehat{\operatorname{Var}}(\hat{\beta}_m)/(\widehat{\operatorname{Var}}(\hat{\beta}_r) + \widehat{\operatorname{Var}}(\hat{\beta}_m))$ — the same combination scheme as `InferenceAllKKMeanDiffIVWC`, but applied to rank-based rather than mean-based sub-estimates. If only one sub-estimate is usable (e.g. no matched pairs, or a degenerate reservoir), $\hat{\beta}_T$ falls back to that sub-estimate alone.

Super class

`Inference` -> `InferenceAllKKWilcoxIVWC`

Methods

Public methods:

- `InferenceAllKKWilcoxIVWC$new()`
- `InferenceAllKKWilcoxIVWC$compute_estimate()`
- `InferenceAllKKWilcoxIVWC$compute_asymp_confidence_interval()`
- `InferenceAllKKWilcoxIVWC$compute_asymp_two_sided_pval()`
- `InferenceAllKKWilcoxIVWC$compute_jackknife_estimate()`
- `InferenceAllKKWilcoxIVWC$compute_jackknife_bias_estimate()`
- `InferenceAllKKWilcoxIVWC$compute_jackknife_std_error()`
- `InferenceAllKKWilcoxIVWC$compute_jackknife_wald_two_sided_pval()`
- `InferenceAllKKWilcoxIVWC$compute_jackknife_wald_confidence_interval()`
- `InferenceAllKKWilcoxIVWC$clone()`

`InferenceAllKKWilcoxIVWC$new()`:

Usage:

```
InferenceAllKKWilcoxIVWC$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A DesignSeqOneByOne object (must be a KK design).

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`InferenceAllKKWilcoxIVWC$compute_estimate()`:

Usage:

```
InferenceAllKKWilcoxIVWC$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, omit only the final combined standard error; component variances are still fitted because they determine IVWC weights.

`InferenceAllKKWilcoxIVWC$compute_asymp_confidence_interval()`:

Usage:

```
InferenceAllKKWilcoxIVWC$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`alpha` Significance level. Default 0.05.

`InferenceAllKKWilcoxIVWC$compute_asymp_two_sided_pval()`:

Usage:

```
InferenceAllKKWilcoxIVWC$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null difference to test against. For any treatment effect at all this is set to zero (the default).

`delta` Null treatment-effect value. Default 0.

`InferenceAllKKWilcoxIVWC$compute_jackknife_estimate()`:

Usage:

```
InferenceAllKKWilcoxIVWC$compute_jackknife_estimate(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`InferenceAllKKWilcoxIVWC$compute_jackknife_bias_estimate()`:

Usage:

```
InferenceAllKKWilcoxIVWC$compute_jackknife_bias_estimate(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`InferenceAllKKWilcoxIVWC$compute_jackknife_std_error()`:

Usage:

```
InferenceAllKKWilcoxIVWC$compute_jackknife_std_error(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`InferenceAllKKWilcoxIVWC$compute_jackknife_wald_two_sided_pval():`

Usage:

```
InferenceAllKKWilcoxIVWC$compute_jackknife_wald_two_sided_pval(
  delta = 0,
  unit = "auto"
)
```

Arguments:

`delta` The null difference to test against. For any treatment effect at all this is set to zero (the default).

`delta` Null treatment-effect value. Default 0.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`InferenceAllKKWilcoxIVWC$compute_jackknife_wald_confidence_interval():`

Usage:

```
InferenceAllKKWilcoxIVWC$compute_jackknife_wald_confidence_interval(
  alpha = 0.05,
  unit = "auto"
)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`alpha` Significance level. Default 0.05.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`unit` Deletion unit. Default `"auto"`.

`InferenceAllKKWilcoxIVWC$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceAllKKWilcoxIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Hodges, J. L., and Lehmann, E. L. (1963). "Estimates of Location Based on Rank Tests." *The Annals of Mathematical Statistics*, 34(2), 598-611, doi:10.1214/aoms/1177704172, for the Hodges-Lehmann estimator underlying both sub-estimates; Kapelner, A., and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388, doi:10.1111/biom.12148, for the KK matching-on-the-fly design and the inverse-variance combination of matched-pair and reservoir estimates.

Legacy class. Not fully tested in `comprehensive_tests.R`.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceAllKKWilcoxIVWC$new(seq_des)
inf$compute_estimate()
```

`InferenceAllSimpleAverageDiff`
Simple Mean-Difference Inference for Continuous Responses

Description

Fits the simplest possible treatment-effect estimator for a continuous response: the unadjusted difference in sample means between the treated and control arms, $\hat{\beta}_T = \bar{y}_T - \bar{y}_C$, with no covariate adjustment. Inference is by **Welch's unequal-variance t-test**: standard error $\sqrt{s_T^2/n_T + s_C^2/n_C}$ (per-arm sample variances, not pooled) with Satterthwaite-Welch degrees of freedom — see `$compute_asymp_confidence_i` for the exact formula. This class has no likelihood tier (`likelihood_tier = "none"`) and provides asymptotic Wald, randomization, and bootstrap (including Bayesian bootstrap) confidence intervals and p-values. Warm starts are disabled for this class, since the simple mean difference is a closed-form estimator (no iterative fit to warm-start).

Initialize a simple mean-difference inference object.

Computes a $1 - \alpha$ level confidence interval for the simple (unadjusted) mean-difference treatment effect $\hat{\beta}_T = \bar{y}_T - \bar{y}_C$, using **Welch's unequal-variance** formula: standard error $\widehat{\mathrm{SE}}(\hat{\beta}_T) = \sqrt{s_T^2/n_T + s_C^2/n_C}$ (sample variances s_T^2, s_C^2 computed separately per arm, not pooled) with Satterthwaite-Welch degrees of freedom $\mathrm{df} = (s_T^2/n_T + s_C^2/n_C)^2 / \left(\frac{(s_T^2/n_T)^2}{n_T-1} + \frac{(s_C^2/n_C)^2}{n_C-1}\right)$; the interval is $\hat{\beta}_T \pm t_{\mathrm{df},1-\alpha/2}\,\widehat{\mathrm{SE}}(\hat{\beta}_T)$. Requires at least 2 observations per arm; otherwise the standard error and interval are `NA`. See `InferenceAsymp` for the shared asymptotic confidence-interval contract this delegates to.

Computes a two-sided Welch's t-test p-value testing $H_0 : \beta_T =$ `delta`, from the same Welch unequal-variance standard error and Satterthwaite-Welch degrees of freedom used by `$compute_asymp_confidence_interv`

— see that method's documentation for the full formula. See `InferenceAsymp` for the shared asymptotic two-sided p-value contract this delegates to.

Computes the simple (unadjusted) mean-difference point estimate $\hat{\beta}_T = \bar{y}_T - \bar{y}_C$, the difference in sample means between the treated and control arms. `NA` if either arm has zero observations. See `InferenceMLEorKMSummaryTable` for the shared estimate-contract this participates in.

Recomputes the simple mean-difference estimate under subject/block bootstrap weights (used by the Bayesian bootstrap and related weighted-resampling machinery — see `InferenceNonParamBootstrap`). The weighted point estimate is $\hat{\beta}_T = \bar{y}_T^w - \bar{y}_C^w$, weighted arm means $\bar{y}_T^w = \sum_i r_i y_i \mathbb{1}[w_i = 1] / \sum_i r_i \mathbb{1}[w_i = 1]$ (and analogously for control), where r_i are the expanded row weights. Unless `estimate_only = TRUE`, the standard error treats the bootstrap weights as frequency weights. Within each arm, with total weight $W = \sum r_i$ and weighted residual sum of squares $Q = \sum r_i (y_i - \bar{y}_w)^2$, the variance contribution is $Q/[W(W-1)]$; the Satterthwaite-Welch degrees of freedom likewise use $W - 1$. This reduces exactly to the ordinary Welch formula for unit weights. Rows with non-finite or non-positive weight, or a non-finite response, are dropped before computing; if no rows survive, returns `NA` with all cached variance components set to `NA`.

Value

A two-sided p-value.

The setting-appropriate (see description) numeric estimate of the treatment effect

Super class

`Inference` -> `InferenceAllSimpleAverageDiff`

Methods

Public methods:

- `InferenceAllSimpleAverageDiff$compute_rand_two_sided_pval()`
- `InferenceAllSimpleAverageDiff$new()`
- `InferenceAllSimpleAverageDiff$compute_asymp_confidence_interval()`
- `InferenceAllSimpleAverageDiff$compute_asymp_two_sided_pval()`
- `InferenceAllSimpleAverageDiff$compute_estimate()`
- `InferenceAllSimpleAverageDiff$compute_estimate_with_bootstrap_weights()`
- `InferenceAllSimpleAverageDiff$clone()`

`InferenceAllSimpleAverageDiff$compute_rand_two_sided_pval()`**:** Uses the randomization-CI layer's two-sided p-value contract (`InferenceRandCI`'s version, not `InferenceRand`'s): for incidence responses this dispatches to the Zhang exact randomization test where applicable rather than refusing outright, matching this class's pre-migration old-ladder behavior (see `InferenceIncidRiskDiff`'s identical rationale). Previously bound to `InferenceRand`'s version instead, which silently regressed Zhang dispatch after migration – see `inference_all_abstract_rand_ci.R`'s `compute_rand_two_sided_pval` for why it's now safe to splice this in outside the old inheritance chain.

Usage:

```
InferenceAllSimpleAverageDiff$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
```

```
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  type = NULL,
  args_for_type = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors.

`delta` Null treatment effect value.

`delta` Null treatment effect value.

`transform_responses` Response transformation to apply during the test. For survival responses the default `"log"` multiplies the recorded times of the units treated under each reference allocation by e^{δ}, event and censoring times alike, with censoring indicators unchanged – the rank-based AFT residual construction (Tsiatis 1990; Wei, Ying and Lin 1990; Jin, Lin, Wei and Ying 2003); see `compute_rand_confidence_interval()` for the assumptions.

`na.rm` Whether to remove non-finite simulated statistics.

`show_progress` Whether to show progress.

`permutations` Optional pre-generated assignment draws.

`type` Optional incidence-specific exact randomization type.

`args_for_type` Optional arguments keyed by `type`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceAllSimpleAverageDiff$new()`:

Usage:

```
InferenceAllSimpleAverageDiff$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  max_resample_attempts = 50L,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A DesignSeqOneByOne object whose entire n subjects are assigned and response y is recorded within.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages. Default `FALSE`.

`max_resample_attempts` Maximum number of times a single bootstrap replicate may be redrawn when the drawn sample fails validity screening. If all attempts fail the replicate is recorded as `NA`, silently reducing the effective `B`. Must be a positive integer. Default `50L`.

`smart_cold_start_default` Whether to use smart cold start values.

`InferenceAllSimpleAverageDiff$compute_asymp_confidence_interval()`:

Usage:

```
InferenceAllSimpleAverageDiff$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Confidence level.

`InferenceAllSimpleAverageDiff$compute_asymp_two_sided_pval():`

Usage:

```
InferenceAllSimpleAverageDiff$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect value.

`delta` Null treatment effect value.

`InferenceAllSimpleAverageDiff$compute_estimate():`

Usage:

```
InferenceAllSimpleAverageDiff$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`estimate_only` If TRUE, skip variance calculations.

`InferenceAllSimpleAverageDiff$compute_estimate_with_bootstrap_weights():`

Usage:

```
InferenceAllSimpleAverageDiff$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Row weights for the bootstrap sample.

`estimate_only` If TRUE, skip variance component calculations.

`estimate_only` If TRUE, skip variance calculations.

`InferenceAllSimpleAverageDiff$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceAllSimpleAverageDiff$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Welch, B. L. (1947). "The Generalization of 'Student's' Problem when Several Different Population Variances are Involved." *Biometrika*, 34(1-2), 28-35, doi:10.1093/biomet/34.12.28, for the unequal-variance t-test and its Satterthwaite-Welch degrees-of-freedom approximation used here.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "continuous")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2 : 10])
seq_des$add_all_subject_responses(c(4.71, 1.23, 4.78, 6.11, 5.95, 8.43))

seq_des_inf = InferenceAllSimpleAverageDiff$new(seq_des)
seq_des_inf$compute_estimate()
seq_des_inf$compute_asymp_confidence_interval()
seq_des_inf$compute_asymp_two_sided_pval()
```

InferenceAllSimpleMeanDiffPooledVar

Simple Mean-Difference Inference with Pooled Variance

Description

Fits the same unadjusted mean-difference point estimate as `InferenceAllSimpleAverageDiff`, $\hat{\beta}_T = \bar{y}_T - \bar{y}_C$, but performs inference via the classical **pooled equal-variance** Student's t-test instead of Welch's unequal-variance version: pooled variance $s_p^2 = \left((n_T - 1)s_T^2 + (n_C - 1)s_C^2\right)/(n_T + n_C - 2)$, standard error $s_p\sqrt{1/n_T + 1/n_C}$, and exact degrees of freedom $n_T + n_C - 2$ — see `$compute_asymp_confidence_interval()` for the full formula. This assumes the two arms have equal population variance; prefer `InferenceAllSimpleAverageDiff` when that assumption is doubtful, since the pooled estimator's nominal coverage degrades under heteroskedasticity with unequal arm sizes. This class does not support censored survival data (enforced at construction). This class has no likelihood tier (`likelihood_tier = "none"`) and provides asymptotic Wald, randomization, and bootstrap (including Bayesian bootstrap) confidence intervals and p-values. Warm starts are disabled for this class, since the simple mean difference is a closed-form estimator (no iterative fit to warm-start).

Initialize simple pooled-variance mean-difference inference for continuous responses and prepare the pooled standard-error calculation used by `InferenceAllSimpleMeanDiffPooledVar`. Disables warm starts (closed-form estimator) and asserts `des_obj` has no censored observations (unsupported by this class).

Computes a $1 - \alpha$ level confidence interval for the simple (unadjusted) mean-difference treatment effect $\hat{\beta}_T = \bar{y}_T - \bar{y}_C$, using the classical **pooled equal-variance** Student's t-test formula (unlike `InferenceAllSimpleAverageDiff`'s Welch unequal-variance version): the pooled variance estimate $s_p^2 = \left((n_T - 1)s_T^2 + (n_C - 1)s_C^2\right)/(n_T + n_C - 2)$ gives standard error $\widehat{\mathrm{SE}}(\hat{\beta}_T) = s_p\sqrt{1/n_T + 1/n_C}$ with exact degrees of freedom $n_T + n_C - 2$; the interval is $\hat{\beta}_T \pm t_{\mathrm{df},1-\alpha/2}\,\widehat{\mathrm{SE}}(\hat{\beta}_T)$. Assumes equal population variances in the two arms — use `InferenceAllSimpleAverageDiff` instead when that assumption is doubtful. Requires at least 2 observations per arm; otherwise returns `c(NA, NA)`. See `InferenceAsymp` for the shared asymptotic confidence-interval contract this participates in.

Computes a two-sided pooled-variance Student's t-test p-value testing $H_0 : \beta_T =$ `delta`, from the same pooled standard error and exact $n_T + n_C - 2$ degrees of freedom used by `$compute_asymp_confidence_interval()` — see that method's documentation for the full formula. See `InferenceAsymp` for the shared asymptotic two-sided p-value contract this participates in.

Value

A new `InferenceAllSimpleMeanDiffPooledVar` object.

Super class

`Inference` -> `InferenceAllSimpleMeanDiffPooledVar`

Methods

Public methods:

- `InferenceAllSimpleMeanDiffPooledVar$new()`
- `InferenceAllSimpleMeanDiffPooledVar$compute_asymp_confidence_interval()`
- `InferenceAllSimpleMeanDiffPooledVar$compute_asymp_two_sided_pval()`
- `InferenceAllSimpleMeanDiffPooledVar$compute_estimate_with_bootstrap_weights()`
- `InferenceAllSimpleMeanDiffPooledVar$clone()`

`InferenceAllSimpleMeanDiffPooledVar$new()`:

Usage:

```
InferenceAllSimpleMeanDiffPooledVar$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed design object.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`InferenceAllSimpleMeanDiffPooledVar$compute_asymp_confidence_interval()`:

Usage:

```
InferenceAllSimpleMeanDiffPooledVar$compute_asymp_confidence_interval(
  alpha = 0.05
)
```

Arguments:

`alpha` Confidence level.

`InferenceAllSimpleMeanDiffPooledVar$compute_asymp_two_sided_pval():`

Usage:

```
InferenceAllSimpleMeanDiffPooledVar$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect value.

`InferenceAllSimpleMeanDiffPooledVar$compute_estimate_with_bootstrap_weights():` Recomputes the pooled-variance difference in means and its pooled (equal-variance) standard error under bootstrap weights, so bootstrap and studentized intervals use the same variance convention as the asymptotic methods; see `InferenceBayesianBootstrap`.

Usage:

```
InferenceAllSimpleMeanDiffPooledVar$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferenceAllSimpleMeanDiffPooledVar$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceAllSimpleMeanDiffPooledVar$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Student [Gosset, W. S.] (1908). "The Probable Error of a Mean." *Biometrika*, 6(1), 1-25, doi:10.1093/biomet/6.1.1, for the pooled-variance two-sample t-test used here.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceAllSimpleMeanDiffPooledVar$new(seq_des)
inf$compute_estimate()
```

InferenceAllSimpleWilcox

Simple Wilcoxon Rank-Sum (Hodges-Lehmann) Inference

Description

Fits a non-parametric treatment-effect estimator based on the two-sample Wilcoxon rank-sum test: the point estimate is the **Hodges-Lehmann** location-shift estimate (the median of all pairwise treatment-minus-control differences $y_{T,i} - y_{C,j}$), and both the confidence interval and two-sided p-value are the standard rank-based Wilcoxon quantities from `stats::wilcox.test()` (normal approximation with continuity correction), not Wald intervals/tests built around the point estimate and a separately estimated standard error. Robust to outliers and does not assume normality or equal arm variances. Not supported for incidence (binary) responses (the Hodges-Lehmann estimator degenerates on 0/1 data — use `InferenceAllSimpleAverageDiff` or a conditional-logistic estimator instead) or censored survival data (use `InferenceSurvivalGehanWilcox` instead). This class has no likelihood tier (`likelihood_tier = "none"`) and does not support the Bayesian bootstrap; its jackknife methods all report explicit non-estimability rather than computing a (statistically unreliable) delete-1 jackknife of the Hodges-Lehmann functional.

Initialize simple Wilcoxon inference and prepare the rank-based treatment statistic used by `InferenceAllSimpleWilcox`. Rejects `response_type = "incidence"` (Hodges-Lehmann degenerates on binary data) and rejects censored survival data at construction; see the class-level documentation for recommended alternatives in both cases. Legal `response_type` values are `"continuous"`, `"count"`, `"proportion"`, `"survival"` (uncensored only), and `"ordinal"`.

Returns the **Hodges-Lehmann** estimate of location shift: the median of all pairwise treatment-minus-control differences $y_{T,i} - y_{C,j}$ (via `wilcox_hl_point_estimate_cpp()`), the standard point estimate associated with the Wilcoxon rank-sum test. Robust to outliers and does not assume normality or equal variances.

Wilcoxon rank-sum test two-sided p-value testing $H_0 : \beta_T =$ `delta` (via `stats::wilcox.test(yT, yC - delta, exact = FALSE)$p.value`, the normal approximation with continuity correction) — a genuine rank-based test, **not** a Wald test built from the Hodges-Lehmann estimate and its standard error, despite living alongside `$compute_asymp_confidence_interval()` in this class's "asymptotic" method family. For `delta != 0`, the control arm's values are shifted by `delta` before testing, so the test checks whether y_T and $y_C +$ `delta` come from the same distribution.

Returns the Hodges-Lehmann confidence interval directly from `stats::wilcox.test(yT, yC, conf.int = TRUE, exact = FALSE, conf.level = 1 - alpha)` — the standard nonparametric interval associated with the Wilcoxon rank-sum test, based on inverting the rank-sum test statistic rather than a Wald normal-approximation interval around `$compute_estimate()`'s point estimate (though the two coincide asymptotically).

Delegates to the genuine rank-based `$compute_asymp_two_sided_pval()` rather than the generic Wald-component z/t formula.

Fixed 2026-09-06: this class did not override `compute_wald_two_sided_pval`, so it fell through to the composed `Wald` component's generic `(estimate - delta) / se` formula built from `compute_estimate()` (the Hodges-Lehmann median-of-pairwise- differences) and `get_standard_error()`. On heavily tied, small-integer count/ordinal data the Hodges-Lehmann estimate lands on exactly 0 far more often than a continuous estimator would, so the Wald statistic came out exactly `0/se = 0` regardless of

se, forcing p = 1 deterministically (observed: pinned at 1 in ~75-98 compute_asymp_two_sided_pval() does not have this failure mode.

Delegates to the genuine rank-based $compute_asymp_confidence_interval() rather than the generic Wald normal-approximation interval, for the same reason as compute_wald_two_sided_pval above.

Fixed 2026-09-06: the generic Wald component's normal-approximation interval is built from get_standard_error(), which this class derives by back-solving se = (ci[2]-ci[1]) / (2*1.96) from stats::wilcox.test()'s own asymptotic CI width. Under heavy ties, that root search can converge to a numerically near-zero-width interval as a search artifact, not a real sampling-uncertainty statement; that spurious near-zero SE then produced a near-[0,0] Wald interval (observed in over 1,200 rows of comprehensive-results data). The rank-based compute_asymp_confidence_interval() inverts the rank-sum test directly and does not go through this derived SE at all.

Reports the jackknife point-estimate as explicitly non-estimable for this Hodges-Lehmann estimator, rather than computing a leave-one-out jackknife: the median-of-pairwise-differences functional is not smooth enough for the delete-1 jackknife's linear-approximation machinery to be reliable. This method exists purely to record that unavailability (via private$cache_nonestimable_estimate()) rather than silently returning a misleading number; see InferenceJackknife for the shared jackknife contract this method participates in.

Reports the jackknife bias-correction estimate as non-estimable for this simple Wilcoxon estimator, for the same reason as $compute_jackknife_estimate() (the Hodges-Lehmann functional is not smooth enough for the delete-1 jackknife); see InferenceJackknife for the shared jackknife contract.

Reports the jackknife standard error as non-estimable for this simple Wilcoxon estimator, for the same reason as $compute_jackknife_estimate(); see InferenceJackknife for the shared jackknife contract.

Reports the jackknife-Wald p-value as non-estimable here, for the same reason as $compute_jackknife_estimate(); see InferenceJackknife.

Reports the jackknife-Wald confidence interval as non-estimable here, for the same reason as $compute_jackknife_estimat see InferenceJackknife.

Super class

Inference -> InferenceAllSimpleWilcox

Methods

Public methods:

- InferenceAllSimpleWilcox$new()
- InferenceAllSimpleWilcox$compute_estimate()
- InferenceAllSimpleWilcox$compute_asymp_two_sided_pval()
- InferenceAllSimpleWilcox$compute_asymp_confidence_interval()
- InferenceAllSimpleWilcox$compute_wald_two_sided_pval()
- InferenceAllSimpleWilcox$compute_wald_confidence_interval()
- InferenceAllSimpleWilcox$compute_jackknife_estimate()
- InferenceAllSimpleWilcox$compute_jackknife_bias_estimate()

- InferenceAllSimpleWilcox$compute_jackknife_std_error()
- InferenceAllSimpleWilcox$compute_jackknife_wald_two_sided_pval()
- InferenceAllSimpleWilcox$compute_jackknife_wald_confidence_interval()
- InferenceAllSimpleWilcox$clone()

InferenceAllSimpleWilcox$new():

Usage:

```
InferenceAllSimpleWilcox$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  max_resample_attempts = 50L,
  smart_cold_start_default = NULL
)
```

Arguments:

des_obj A completed DesignSeqOneByOne object.

model_formula Optional formula for covariate adjustment. If NULL (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

verbose Whether to print progress messages. Default FALSE.

max_resample_attempts Maximum number of times a single bootstrap replicate may be re-drawn when the drawn sample fails validity screening. If all attempts fail the replicate is recorded as NA, silently reducing the effective B. Must be a positive integer. Default 50L.

smart_cold_start_default Flag for consistent API.

InferenceAllSimpleWilcox$compute_estimate():

Usage:

```
InferenceAllSimpleWilcox$compute_estimate(estimate_only = FALSE)
```

Arguments:

estimate_only If TRUE, skip variance component calculations.

InferenceAllSimpleWilcox$compute_asymp_two_sided_pval():

Usage:

```
InferenceAllSimpleWilcox$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

delta Null treatment effect. Default 0.

delta Null treatment effect. Default 0.

delta Null treatment-effect value. Default 0.

InferenceAllSimpleWilcox$compute_asymp_confidence_interval():

Usage:

```
InferenceAllSimpleWilcox$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.

`InferenceAllSimpleWilcox$compute_wald_two_sided_pval()`:

Usage:

```
InferenceAllSimpleWilcox$compute_wald_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect. Default 0.
`delta` Null treatment effect. Default 0.
`delta` Null treatment-effect value. Default 0.

`InferenceAllSimpleWilcox$compute_wald_confidence_interval()`:

Usage:

```
InferenceAllSimpleWilcox$compute_wald_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.

`InferenceAllSimpleWilcox$compute_jackknife_estimate()`:

Usage:

```
InferenceAllSimpleWilcox$compute_jackknife_estimate(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.

`InferenceAllSimpleWilcox$compute_jackknife_bias_estimate()`:

Usage:

```
InferenceAllSimpleWilcox$compute_jackknife_bias_estimate(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.

`InferenceAllSimpleWilcox$compute_jackknife_std_error()`:

Usage:

```
InferenceAllSimpleWilcox$compute_jackknife_std_error(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.

`InferenceAllSimpleWilcox$compute_jackknife_wald_two_sided_pval()`:

Usage:

```
InferenceAllSimpleWilcox$compute_jackknife_wald_two_sided_pval(
  delta = 0,
  unit = "auto"
)
```

Arguments:

`delta` Null treatment effect. Default 0.
`delta` Null treatment effect. Default 0.
`delta` Null treatment-effect value. Default 0.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.

`InferenceAllSimpleWilcox$compute_jackknife_wald_confidence_interval()`:

Usage:

```
InferenceAllSimpleWilcox$compute_jackknife_wald_confidence_interval(
  alpha = 0.05,
  unit = "auto"
)
```

Arguments:

`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.

`InferenceAllSimpleWilcox$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceAllSimpleWilcox$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Hodges, J. L., and Lehmann, E. L. (1963). "Estimates of Location Based on Rank Tests." *The Annals of Mathematical Statistics*, 34(2), 598-611, doi:10.1214/aoms/1177704172, for the Hodges-Lehmann estimator; Wilcoxon, F. (1945). "Individual Comparisons by Ranking Methods." *Biometrics Bulletin*, 1(6), 80-83, doi:10.2307/3001968, for the underlying rank-sum test.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "continuous")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2 : 10])
seq_des$add_all_subject_responses(c(4.71, 1.23, 4.78, 6.11, 5.95, 8.43))

seq_des_inf = InferenceAllSimpleWilcox$new(seq_des)
seq_des_inf$compute_estimate()
```

InferenceBaiAdjustedTKK14

Bai Adjusted-t Mean-Difference Inference for KK14 Designs

Description

Continuous-response mean-difference inference for designs assigned by DesignSeqOneByOneKK14 (the Kapelner-Krieger 2014 sequential matching-on-the-fly design). The point estimate and its variance are the closed-form Bai-adjusted-t combination of the matched-pairs mean difference and the unmatched-reservoir mean difference, inverse-variance-weighted when both are usable; the full formula, pair-distance definition, and confidence-interval/p-value construction are shared with InferenceBaiAdjustedTKK21. The two leaves differ only in how pair distance is defined during matching: this class (KK14) uses the plain squared Euclidean distance $\sum_j (x_{1j} - x_{2j})^2$ between candidate subjects' covariate vectors, unlike KK21's covariate-weighted distance. Because the estimator is closed-form, initialization does not use warm starts (there is no iterative fit to warm-start).

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether compute_rand_two_sided_pval() is actually usable on this instance right now – FALSE exactly when it would stop(): an incidence-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see private$should_use_design_ra TRUE for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so InferenceSuite can check this before attempting the sentinel instead of relying on the stop() being silently swallowed into a pval = NA "ok" row – the single source of truth for both this check and compute_rand_two_sided_pval()'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Legacy class. Not fully tested in `comprehensive_tests.R`.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceBaiAdjustedTKK14`

Methods

Public methods:

- `InferenceBaiAdjustedTKK14$approximate_randomization_distribution_beta_hat_T()`
- `InferenceBaiAdjustedTKK14$supports_rand_pval_for_incidence()`
- `InferenceBaiAdjustedTKK14$compute_rand_two_sided_pval()`
- `InferenceBaiAdjustedTKK14$clone()`

`InferenceBaiAdjustedTKK14$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceBaiAdjustedTKK14$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceBaiAdjustedTKK14$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceBaiAdjustedTKK14$supports_rand_pval_for_incidence()
```

`InferenceBaiAdjustedTKK14$compute_rand_two_sided_pval()`:

Usage:

```
InferenceBaiAdjustedTKK14$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceBaiAdjustedTKK14$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceBaiAdjustedTKK14$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceBaiAdjustedTKK14$new(seq_des)
inf$compute_estimate()
```

InferenceBaiAdjustedTKK21

Bai Adjusted-t Mean-Difference Inference for KK21 Designs

Description

Continuous-response mean-difference inference for designs assigned by DesignSeqOneByOneKK21 (the Kapelner-Krieger 2021 sequential matching-on-the-fly design with covariate-weighted matching). The point estimate and its variance are the closed-form Bai-adjusted-t combination of the matched-pairs mean difference and the unmatched-reservoir mean difference, inverse-variance-weighted when both are usable; see InferenceBaiAdjustedTKK14 for the full formula, the pair-distance definition, and the confidence-interval/p-value construction shared with this class. The two leaves differ only in how pair distance is defined during matching: this class (KK21) uses the design's covariate-weighted squared distance $\sum_j w_j(x_{1j} - x_{2j})^2$, where w_j are the design's covariate_weights (see DesignSeqOneByOneKK21), unlike KK14's unweighted distance. Because the estimator is closed-form, initialization does not use warm starts (there is no iterative fit to warm-start).

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether compute_rand_two_sided_pval() is actually usable on this instance right now – FALSE exactly when it would stop(): an incidence-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see private$should_use_design_ra TRUE for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so InferenceSuite can check this before attempting the sentinel instead of relying on the stop() being silently swallowed into a pval = NA "ok" row – the single source of truth for both this check and compute_rand_two_sided_pval()'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Legacy class. Not fully tested in comprehensive_tests.R.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceBaiAdjustedTKK21

Methods

Public methods:

- InferenceBaiAdjustedTKK21$approximate_randomization_distribution_beta_hat_T()
- InferenceBaiAdjustedTKK21$supports_rand_pval_for_incidence()
- InferenceBaiAdjustedTKK21$compute_rand_two_sided_pval()
- InferenceBaiAdjustedTKK21$clone()

InferenceBaiAdjustedTKK21$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceBaiAdjustedTKK21$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

r Number of randomization vectors. Default 501.

r Number of randomization vectors.

delta The null difference. Default 0.

delta Null difference.

transform_responses Type of transformation. Default "none".

transform_responses Transformation.

show_progress Show progress bar. Default TRUE.

show_progress Show progress.

permutations Pre-computed permutations. Default NULL.

permutations Pre-computed permutations.

debug If TRUE, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default FALSE.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceBaiAdjustedTKK21$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceBaiAdjustedTKK21$supports_rand_pval_for_incidence()
```

`InferenceBaiAdjustedTKK21$compute_rand_two_sided_pval()`:

Usage:

```
InferenceBaiAdjustedTKK21$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceBaiAdjustedTKK21$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceBaiAdjustedTKK21$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
```

```
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceBaiAdjustedTKK21$new(seq_des)
inf$compute_estimate()
```

`InferenceContinKKGLMM` *Linear Mixed Model Inference for KK Designs with Continuous Response*

Description

Fits a linear mixed model for continuous responses under a KK matching-on-the-fly design. The matched-pair strata enter as a subject-level random intercept `(1 | group_id)`, accounting for within-pair correlation.

When `use_rcpp = TRUE` (default) the likelihood is maximised by an internal Rcpp/L-BFGS routine that requires no external packages. Set `use_rcpp = FALSE` to fall back to **glmmTMB**.

The treatment coefficient β_T is on the response's natural (untransformed) scale — a mean difference, not a ratio or log-scale effect. `likelihood_tier = "full"`: likelihood-ratio, score, and Wald tests are all available when the model converges (see `$get_likelihood_test_spec()` inherited from the shared count/GLMM likelihood plumbing). Validity requires the random-intercept-per-pair structure to correctly capture the design's matching dependence and the usual linear mixed model assumptions (conditional normality of responses and pair effects, correctly specified fixed-effects formula).

Super class

`Inference` -> `InferenceContinKKGLMM`

Methods

Public methods:

- `InferenceContinKKGLMM$new()`
- `InferenceContinKKGLMM$compute_estimate()`
- `InferenceContinKKGLMM$compute_estimate_with_bootstrap_weights()`
- `InferenceContinKKGLMM$compute_asymp_confidence_interval()`
- `InferenceContinKKGLMM$compute_asymp_two_sided_pval()`
- `InferenceContinKKGLMM$clone()`

`InferenceContinKKGLMM$new()`: Initialize inference for the linear mixed model $Y_i = \beta_0 + \beta_T W_i + X_i^\top \gamma + b_{g(i)} + \epsilon_i$, $b_g \sim N(0, \sigma_b^2)$, $\epsilon_i \sim N(0, \sigma_e^2)$, where $g(i)$ is subject i's matched-pair group id, W_i is the treatment indicator, X_i are covariates, and β_T is the treatment effect (mean difference on the response's natural scale). The random intercept b_g absorbs the within-pair correlation induced by matching, so β_T's standard error correctly reflects the design.

Usage:

```
InferenceContinKKGLMM$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  use_gls_fast_path = TRUE,
  use_gls_fast_path_bootstrap = FALSE,
  verbose = FALSE,
  smart_cold_start_default = NULL,
  optimization_alg = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a continuous response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`use_rcpp` Logical. If `TRUE` (default), use the optimised Rcpp Gaussian LMM implementation (no external package required). If `FALSE`, use **glmmTMB**.

`use_gls_fast_path` Logical. If `TRUE` (default), use a fast GLS estimator (no optimisation) for `estimate_only` calls during randomisation inference once variance components are cached from a prior full fit. Statistically exact: fixing VC at the null-fit MLE and permuting only the treatment assignment gives a valid permutation test by exchangeability. Set `FALSE` to always run full L-BFGS.

`use_gls_fast_path_bootstrap` Logical. If `TRUE`, also use the fast GLS estimator for non-studentised bootstrap draws (`estimate_only = TRUE` weighted calls). Asymptotically valid by the plug-in principle (VC orthogonal to beta_T in the Fisher information), but not exact in finite samples. Default `FALSE`.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`optimization_alg` The optimization algorithm to use. Default is dispatched via policy.

`InferenceContinKKGLMM$compute_estimate()`**:** Fits the linear mixed model by maximum likelihood and returns $\hat{\beta}_T$. When `use_rcpp = TRUE` (the default), the log-likelihood $\ell(\beta, \sigma_b^2, \sigma_e^2)$ is maximized directly by an internal Rcpp/L-BFGS routine over β and the log-variance components; otherwise **glmmTMB** performs the fit. If `use_gls_fast_path = TRUE` and variance components are already cached from a prior full fit (used during randomization inference, where only the treatment column changes across permutations), `estimate_only = TRUE` calls instead solve the generalized least-squares problem at the cached variance components — exact under exchangeability of the permuted treatment assignment, since fixing the variance components at their null-fit MLE and permuting only W preserves the permutation test's validity.

Usage:

```
InferenceContinKKGLMM$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations (standard error, degrees of freedom) needed for confidence intervals or p-values; only $\hat{\beta}_T$ is returned.

`InferenceContinKKGLMM$compute_estimate_with_bootstrap_weights()`**:** Refits the linear mixed model with subject/block-level weights applied to each row's contribution to the likelihood

(Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded from subject/block level to individual rows via `private$expand_subject_or_block_weights_to_row_weights()`), and returns the reweighted estimate $\hat{\beta}_T^{(w)}$. When weights are effectively constant, this collapses to the unweighted `compute_estimate()` call (returns `df = Inf` to signal a degenerate/skipped bootstrap replicate rather than refitting). When `use_rcpp = TRUE`, a weighted Rcpp fast path is tried first via `private$weighted_rcpp_estimate()`; otherwise `private$compute_weighted_glmm_bootstrap_estimate()` refits via **glmmTMB**-based machinery.

Usage:

```
InferenceContinKKGLMM$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Numeric vector of nonnegative weights, one per matched-pair group or reservoir subject (bootstrap draw weights), expanded to per-row weights before fitting.

`estimate_only` Logical. If TRUE, skip standard-error computation.

Returns: The reweighted treatment estimate $\hat{\beta}_T^{(w)}$.

`InferenceContinKKGLMM$compute_asymp_confidence_interval()`**:** Computes a $1-\alpha$ Wald confidence interval for β_T from the fitted-model standard error, using a normal or t critical value depending on the resolved degrees of freedom (see `private$compute_z_or_t_ci_from_s_and_df`).

Usage:

```
InferenceContinKKGLMM$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level of the interval is $1 -$ `alpha`. Default `0.05`.

`InferenceContinKKGLMM$compute_asymp_two_sided_pval()`**:** Computes a two-sided Wald p-value for $H_0 : \beta_T =$ `delta` using the fitted-model estimate and standard error (same statistic that `$compute_asymp_confidence_interval()` inverts).

Usage:

```
InferenceContinKKGLMM$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment-effect value. Default `0`.

`InferenceContinKKGLMM$clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
InferenceContinKKGLMM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Comparable Python API: statsmodels MixedLM. See also: Mixed model (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinKKGLMM$new(seq_des)
inf$compute_estimate()
```

InferenceContinKKOLSIVWC

OLS IVWC Compound Inference for KK Designs

Description

Fits a variance-weighted compound estimator for KK matching-on-the-fly designs with continuous responses using OLS regression for matched-pair differences and reservoir outcomes, with the treatment indicator and, optionally, all recorded covariates as predictors. Note that warm starts are disabled for this class as OLS is a closed-form estimator and does not benefit from initialization.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

The point estimate $\hat{\beta}_T$ is the inverse-variance-weighted combination of an OLS fit on matched-pair within-pair differences and an OLS fit on reservoir (unmatched) subjects' outcomes, falling back to whichever sub-fit is usable if the other is not — the same compound combination rule used by `InferenceAllKKMeanDiffIVWC`, generalized here to allow covariate adjustment via `model_formula`. `likelihood_tier = "none"`: this is an estimating-equation (least squares) estimator, not a fitted likelihood, so only Wald-type asymptotic inference is available (no likelihood-ratio or score test).

Legacy class. Not fully tested in `comprehensive_tests.R`.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one

per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceContinKKOLSIVWC

Methods

Public methods:

- InferenceContinKKOLSIVWC$approximate_randomization_distribution_beta_hat_T()
- InferenceContinKKOLSIVWC$supports_rand_pval_for_incidence()
- InferenceContinKKOLSIVWC$compute_rand_two_sided_pval()
- InferenceContinKKOLSIVWC$clone()

InferenceContinKKOLSIVWC$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceContinKKOLSIVWC$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

r Number of randomization vectors. Default 501.

r Number of randomization vectors.

delta The null difference. Default 0.

delta Null difference.

transform_responses Type of transformation. Default "none".

transform_responses Transformation.

show_progress Show progress bar. Default TRUE.

show_progress Show progress.

permutations Pre-computed permutations. Default NULL.

permutations Pre-computed permutations.

debug If TRUE, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default FALSE.

zero_one_logit_clamp The clamping amount for exact 0 and 1 values when logging

zero_one_logit_clamp The clamping amount for exact 0 and 1 values when logging

`InferenceContinKKOLSIVWC$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceContinKKOLSIVWC$supports_rand_pval_for_incidence()
```

`InferenceContinKKOLSIVWC$compute_rand_two_sided_pval()`:

Usage:

```
InferenceContinKKOLSIVWC$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceContinKKOLSIVWC$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceContinKKOLSIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinKKOLSIVWC$new(seq_des)
inf$compute_estimate()
```

InferenceContinKKOLSOneLik

OLS Combined-Likelihood Inference for KK Designs

Description

Fits a single stacked OLS regression over matched-pair differences and reservoir observations for KK matching-on-the-fly designs with continuous responses, using the treatment indicator and, optionally, all recorded covariates as predictors. Note that warm starts are disabled for this class as OLS is a closed-form estimator and does not benefit from initialization.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Model. Let m be the number of matched pairs and $n_R = n_{RT} + n_{RC}$ the number of unmatched reservoir subjects. The design matrix stacks two blocks: m matched-pair difference rows (each row's response is the within-pair outcome difference, coded with an implicit unit treatment column and covariate differences X_d), and n_R reservoir rows (raw covariates plus a treatment/matching-status indicator column). The stacked regression is fit by ordinary least squares (`lm.fit`), and $\hat{\beta}_T$ is the coefficient on the treatment/matched-difference column, i.e. an additive mean-difference estimand on the outcome's natural scale. If only matched pairs exist, `j_treat = 1`; if only reservoir data exist, `j_treat = 2`; if both exist, the combined design uses `j_treat = 2`. If neither matched pairs nor a treatment-and-control-populated reservoir exist, the estimate is marked nonestimable (`"no_usable_matched_or_reservoir_data"`).

Variance. Standard errors use the HC2 heteroskedasticity-consistent sandwich estimator (`ols_hc2_post_fit_cpp`), not the classical OLS variance, so the Wald confidence interval/p-value are robust to heteroskedasticity across the matched/reservoir blocks.

Likelihood tier. `likelihood_tier = "full"`: this is a genuine Gaussian likelihood (not a quasi-likelihood or partial likelihood), so score, gradient, and likelihood-ratio testing types are available in addition to Wald, and an exact (not higher-order-accurate) Bartlett correction reproduces base R's `lm()` classical partial F-test exactly under the classical homoskedastic-Gaussian-errors assumption (a stronger assumption than the HC2-robust Wald path uses, so the two paths need not agree numerically).

Assumptions. Continuous response; independent matched pairs and/or independent reservoir subjects; no censoring (`assertNoCensoring()` is enforced); a KK matching-on-the-fly design (`DesignSeqOneByOneKK14` or subclass).

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceContinKKOLSOneLik

Methods

Public methods:

- InferenceContinKKOLSOneLik$approximate_randomization_distribution_beta_hat_T()
- InferenceContinKKOLSOneLik$supports_rand_pval_for_incidence()
- InferenceContinKKOLSOneLik$compute_rand_two_sided_pval()
- InferenceContinKKOLSOneLik$clone()

InferenceContinKKOLSOneLik$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceContinKKOLSOneLik$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

r Number of randomization vectors. Default 501.

r Number of randomization vectors.

delta The null difference. Default 0.

delta Null difference.

transform_responses Type of transformation. Default "none".

transform_responses Transformation.

show_progress Show progress bar. Default TRUE.

show_progress Show progress.

permutations Pre-computed permutations. Default NULL.

permutations Pre-computed permutations.

debug If TRUE, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default FALSE.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceContinKKOLSOneLik$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceContinKKOLSOneLik$supports_rand_pval_for_incidence()
```

`InferenceContinKKOLSOneLik$compute_rand_two_sided_pval()`:

Usage:

```
InferenceContinKKOLSOneLik$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceContinKKOLSOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceContinKKOLSOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A. and Krieger, A. M. (2014). Matching on-the-fly: Sequential allocation with higher power and efficiency. *Biometrics*, 70(2), 378-388. doi:10.1111/biom.12148. (KK14 in `REFERENCES.md`.)

See Also

InferenceContinKKOLSIVWC for the inverse-variance-weighted-combination alternative to this one-likelihood combined-fit approach; analogous Python API: statsmodels GLM/OLS.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinKKOLSOneLik$new(seq_des)
inf$compute_estimate()
```

InferenceContinKKQuantileRegrIVWC

Quantile Regression Compound Estimator for KK Matching-on-the-Fly Designs

---

Description

A variance-weighted compound quantile regression estimator for KK matching-on-the-fly designs with continuous responses. The estimator combines:

1. Quantile regression on within-pair differences (matched pairs)
2. Quantile regression on reservoir subjects (treatment vs control)

using the same variance-weighted combination logic as the OLS compound estimator.

Default quantile: `tau = 0.5` **(median regression).** At `tau = 0.5` this estimates the median treatment effect, which is the canonical nonparametric location estimator and is more robust to outliers and heavy-tailed response distributions than the OLS mean-based estimator. To target a different quantile of the treatment effect distribution — for example the 25th or 75th percentile — pass `tau = 0.25` or `tau = 0.75` to the constructor:

```
inf = InferenceContinKKQuantileRegrIVWC$
new(seq_des, tau = 0.75)
```

Any value strictly between 0 and 1 is accepted.

Standard errors use Powell's "nid" sandwich estimator (non-iid), which is more robust than the "iid" (constant-density) assumption; the implementation falls back to "iid" on failure. Asymptotic z-based inference is used throughout.

The randomization-based confidence interval is inherited from the base class and is valid for location-shift models at all quantiles: shifting y by delta maps the tau-th quantile treatment effect to delta under the null.

This class requires the **quantreg** package, which is listed in Suggests and is not installed automatically with **EDI**. Install **quantreg** before using this class.

Legacy class. Not fully tested in `comprehensive_tests.R`.

Super class

Inference -> InferenceContinKKQuantileRegrIVWC

Methods

Public methods:

- InferenceContinKKQuantileRegrIVWC$new()
- InferenceContinKKQuantileRegrIVWC$clone()

InferenceContinKKQuantileRegrIVWC$new(): Initialize continuous-response KK IVWC quantile-regression inference; see InferenceContinKKQuantileRegrIVWC.

Usage:

```
InferenceContinKKQuantileRegrIVWC$new(
  des_obj,
  model_formula = NULL,
  tau = 0.5,
  verbose = FALSE
)
```

Arguments:

des_obj A DesignSeqOneByOne object whose entire n subjects are assigned and response y is recorded within.

model_formula Optional formula for covariate adjustment. If NULL (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

tau The quantile level for regression, strictly between 0 and 1. The default tau = 0.5 estimates the median treatment effect. Pass a different value (e.g. tau = 0.25 or tau = 0.75) to target the corresponding percentile of the treatment effect distribution.

verbose A flag indicating whether messages should be displayed to the user. Default is FALSE.

Examples:

```
set.seed(1)
x_dat <- data.frame(
  x1 = c(-1.2, -0.7, -0.2, 0.3, 0.8, 1.3, 1.8, 2.3),
  x2 = c(0, 1, 0, 1, 0, 1, 0, 1)
)
seq_des <- DesignSeqOneByOneKK14$new(n = nrow(x_dat), response_type = "continuous", verbose =
FALSE)
for (i in seq_len(nrow(x_dat))) {
  seq_des$add_one_subject_to_experiment_and_assign(x_dat[i, , drop = FALSE])
}
seq_des$add_all_subject_responses(c(1.2, 0.9, 1.5, 1.8, 2.1, 1.7, 2.6, 2.2))
infer <- InferenceContinKKQuantileRegrIVWC$new(seq_des, verbose = FALSE)
infer
```

InferenceContinKKQuantileRegrIVWC$clone(): The objects of this class are cloneable with this method.

Usage:

`InferenceContinKKQuantileRegrIVWC$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinKKQuantileRegrIVWC$new(seq_des)
inf$compute_estimate()


## ----------------------------------------------
## Method `InferenceContinKKQuantileRegrIVWC$new()`
## ----------------------------------------------

set.seed(1)
x_dat <- data.frame(
  x1 = c(-1.2, -0.7, -0.2, 0.3, 0.8, 1.3, 1.8, 2.3),
  x2 = c(0, 1, 0, 1, 0, 1, 0, 1)
)
seq_des <- DesignSeqOneByOneKK14$new(n = nrow(x_dat), response_type = "continuous", verbose =
FALSE)
for (i in seq_len(nrow(x_dat))) {
  seq_des$add_one_subject_to_experiment_and_assign(x_dat[i, , drop = FALSE])
}
seq_des$add_all_subject_responses(c(1.2, 0.9, 1.5, 1.8, 2.1, 1.7, 2.6, 2.2))
infer <- InferenceContinKKQuantileRegrIVWC$new(seq_des, verbose = FALSE)
infer
```

InferenceContinKKQuantileRegrOneLik

Quantile Regression Combined-Likelihood Compound Estimator for KK Designs (Continuous)

Description

Fits the combined stacked quantile regression (matched-pair differences + reservoir) using the treatment indicator and all recorded covariates for continuous responses. Minimises the joint check-function loss over both data sources simultaneously. Inference is based on the stacked combined-likelihood quantile-regression fit.

Model. Analogous to `InferenceContinKKOLSOneLik`'s single stacked design (matched-pair difference rows plus reservoir rows fit jointly), but the objective is the `quantreg` check-function loss $\rho_\tau(u) = u(\tau - \mathbf{1}_{u<0})$ rather than squared error, so the estimand β_T is the treatment effect on the

τ-th quantile of the response, not the mean. `tau = 0.5` (default) targets the median treatment effect, which is more robust to outliers and heavy-tailed responses than the OLS mean-based estimator; any value strictly between 0 and 1 is accepted. `likelihood_tier = "none"`: no likelihood-based (score/gradient/lik_ratio) testing types, only Wald.

Assumptions. Continuous response; independent matched pairs and/or independent reservoir subjects; no censoring; a KK matching-on-the-fly design. Requires the **quantreg** package (listed in Suggests, not installed automatically).

Super class

`Inference` -> `InferenceContinKKQuantileRegrOneLik`

Methods

Public methods:

- `InferenceContinKKQuantileRegrOneLik$new()`
- `InferenceContinKKQuantileRegrOneLik$clone()`

`InferenceContinKKQuantileRegrOneLik$new()`**:** Initialize continuous-response KK combined-likelihood quantile-regression inference. The shared stacked matched-pair/reservoir quantile-regression fit and its `tau` semantics are documented on `InferenceContinKKQuantileRegrOneLik` and on the shared `compute_estimate()` method it inherits from the `KKQuantileRegrOneLik` component.

Usage:

```
InferenceContinKKQuantileRegrOneLik$new(
  des_obj,
  model_formula = NULL,
  tau = 0.5,
  verbose = FALSE
)
```

Arguments:

`des_obj` A DesignSeqOneByOne object whose entire n subjects are assigned and response y is recorded within.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`tau` The quantile level for regression, strictly between 0 and 1. Default is 0.5.

`verbose` Whether to print progress messages.

Examples:

```
set.seed(1)
x_dat <- data.frame(
  x1 = c(-1.2, -0.7, -0.2, 0.3, 0.8, 1.3, 1.8, 2.3),
  x2 = c(0, 1, 0, 1, 0, 1, 0, 1)
)
seq_des <- DesignSeqOneByOneKK14$new(n = nrow(x_dat), response_type = "continuous", verbose =
FALSE)
```

```
for (i in seq_len(nrow(x_dat))) {
  seq_des$add_one_subject_to_experiment_and_assign(x_dat[i, , drop = FALSE])
}
seq_des$add_all_subject_responses(c(1.2, 0.9, 1.5, 1.8, 2.1, 1.7, 2.6, 2.2))
infer <- InferenceContinKKQuantileRegrOneLik$new(seq_des, verbose
= FALSE)
infer
```

`InferenceContinKKQuantileRegrOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceContinKKQuantileRegrOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A. and Krieger, A. M. (2014). Matching on-the-fly: Sequential allocation with higher power and efficiency. *Biometrics*, 70(2), 378-388. doi:10.1111/biom.12148. (KK14 in `REFERENCES.md`.)
Koenker, R. (2005). *Quantile Regression*. Cambridge University Press.

See Also

`InferenceContinKKQuantileRegrIVWC` for the inverse-variance-weighted-combination alternative to this one-likelihood combined-fit approach. Quantile regression (orientation).

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinKKQuantileRegrOneLik$new(seq_des)
inf$compute_estimate()


## ------------------------------------------------
## Method `InferenceContinKKQuantileRegrOneLik$new()`
## ------------------------------------------------

set.seed(1)
x_dat <- data.frame(
  x1 = c(-1.2, -0.7, -0.2, 0.3, 0.8, 1.3, 1.8, 2.3),
  x2 = c(0, 1, 0, 1, 0, 1, 0, 1)
)
seq_des <- DesignSeqOneByOneKK14$new(n = nrow(x_dat), response_type = "continuous", verbose =
FALSE)
for (i in seq_len(nrow(x_dat))) {
  seq_des$add_one_subject_to_experiment_and_assign(x_dat[i, , drop = FALSE])
```

```
}
seq_des$add_all_subject_responses(c(1.2, 0.9, 1.5, 1.8, 2.1, 1.7, 2.6, 2.2))
infer <- InferenceContinKKQuantileRegrOneLik$new(seq_des, verbose
= FALSE)
infer
```

`InferenceContinKKRobustRegrIVWC`
Robust-Regression IVWC Compound Inference for KK Designs

Description

Fits a variance-weighted compound estimator for KK matching-on-the-fly designs with continuous responses using robust regression for matched-pair differences and reservoir outcomes, with treatment and, optionally, all recorded covariates as predictors.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceContinKKRobustRegrIVWC`

Methods

Public methods:

- `InferenceContinKKRobustRegrIVWC$approximate_randomization_distribution_beta_hat_T()`
- `InferenceContinKKRobustRegrIVWC$supports_rand_pval_for_incidence()`

- InferenceContinKKRobustRegrIVWC$compute_rand_two_sided_pval()
- InferenceContinKKRobustRegrIVWC$clone()

InferenceContinKKRobustRegrIVWC$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceContinKKRobustRegrIVWC$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

InferenceContinKKRobustRegrIVWC$supports_rand_pval_for_incidence():

Usage:

```
InferenceContinKKRobustRegrIVWC$supports_rand_pval_for_incidence()
```

InferenceContinKKRobustRegrIVWC$compute_rand_two_sided_pval():

Usage:

```
InferenceContinKKRobustRegrIVWC$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceContinKKRobustRegrIVWC$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceContinKKRobustRegrIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinKKRobustRegrIVWC$new(seq_des)
inf$compute_estimate()
```

InferenceContinKKRobustRegrOneLik

Robust-Regression Combined-Likelihood Inference for KK Designs

---

Description

Fits a single stacked robust regression over matched-pair differences and reservoir observations for KK matching-on-the-fly designs with continuous responses.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization

statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Model. Analogous to `InferenceContinKKOLSOneLik`'s single stacked design (matched-pair difference rows plus reservoir rows fit in one regression, treatment coefficient β_T), but fit with a robust M/MM-estimator (`MASS::rlm`, or an internal Rcpp IRLS kernel when `use_rcpp = TRUE`) instead of ordinary least squares. `"MM"` (the default `method`) starts from an LQS-based high-breakdown fit; `"M"` can warm-start from OLS (`start_with_ols = TRUE`).

Likelihood tier. `likelihood_tier = "quasi"`: the robust objective is not a normalized likelihood, so only Wald-type asymptotic inference is available (`compute_wald_confidence_interval()`/ `compute_wald_two_sided_pval()`, aliased by the standard `compute_asymp_*` names) — no score/gradient/likelihood-ratio testing types, unlike the OLS one-likelihood sibling.

Assumptions. Continuous response; independent matched pairs and/or independent reservoir subjects; no censoring; a KK matching-on-the-fly design. Robust regression trades some efficiency under exactly-Gaussian errors for resistance to outliers and heavy tails.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceContinKKRobustRegrOneLik`

Methods

Public methods:

- `InferenceContinKKRobustRegrOneLik$approximate_randomization_distribution_beta_hat_T()`
- `InferenceContinKKRobustRegrOneLik$supports_rand_pval_for_incidence()`
- `InferenceContinKKRobustRegrOneLik$compute_rand_two_sided_pval()`
- `InferenceContinKKRobustRegrOneLik$clone()`

`InferenceContinKKRobustRegrOneLik$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceContinKKRobustRegrOneLik$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceContinKKRobustRegrOneLik$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceContinKKRobustRegrOneLik$supports_rand_pval_for_incidence()
```

`InferenceContinKKRobustRegrOneLik$compute_rand_two_sided_pval()`:

Usage:

```
InferenceContinKKRobustRegrOneLik$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceContinKKRobustRegrOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceContinKKRobustRegrOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A. and Krieger, A. M. (2014). Matching on-the-fly: Sequential allocation with higher power and efficiency. *Biometrics*, 70(2), 378-388. doi:10.1111/biom.12148. (KK14 in `REFERENCES.md`.)

See Also

`InferenceContinKKRobustRegrIVWC` for the inverse-variance-weighted-combination alternative to this one-likelihood combined-fit approach. Analogous Python API: statsmodels RLM.

`InferenceContinLin` *Lin (2013) Covariate-Adjusted OLS Inference for Continuous Responses*

Description

Fits Lin's (2013) covariate-adjusted linear estimator for continuous responses: OLS of Y_i on $[1, W_i, X_i^c, W_i X_i^c]$, where $X_i^c = X_i - \bar{X}$ are covariates centered at their sample means and $W_i X_i^c$ are treatment-by-centered-covariate interactions (omitted when there are no covariates, reducing to plain OLS with $\hat{\beta}_T$ the simple mean difference). Centering makes $\hat{\beta}_T$ interpretable as the average treatment effect regardless of whether interactions are included, following the design-based reinterpretation of Freedman's critique of ANCOVA in randomized experiments. Standard errors use the HC2 heteroskedasticity-consistent (Huber-White-type) covariance estimator (`ols_hc2_post_fit_cpp`), not classical OLS SEs assuming homoskedasticity — this is the estimator Lin (2013) recommends since it remains conservative under treatment-effect heterogeneity, unlike the classical or HC0 sandwich variants. `likelihood_tier = "full"`: Wald, score, gradient, and likelihood-ratio tests are all available, with the parametric-likelihood bootstrap using the OLS Gaussian-errors model

($Y_i \mid x_i \sim N(x_i^\top \beta, \sigma^2)$) as the generative null even though the design-based HC2 standard error does not itself assume homoskedastic Gaussian errors — the likelihood-ratio/score/gradient machinery is a secondary, model-based inference path alongside the primary HC2-Wald and randomization paths. Validity of $\hat{\beta}_T$ as an average-treatment-effect estimator relies on randomization (of W), not on any particular outcome model; the working linear model need not be correctly specified.

Super class

`Inference` -> `InferenceContinLin`

Methods

Public methods:

- `InferenceContinLin$new()`
- `InferenceContinLin$compute_estimate()`
- `InferenceContinLin$compute_estimate_with_bootstrap_weights()`
- `InferenceContinLin$compute_asymp_confidence_interval()`
- `InferenceContinLin$compute_asymp_two_sided_pval()`
- `InferenceContinLin$clone()`

`InferenceContinLin$new()`: Initialize inference for Lin's (2013) covariate-adjusted OLS estimator (intercept, treatment, centered covariates, and treatment-by-centered-covariate interactions); see `InferenceContinLin` for the model form. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferenceContinLin$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  harden = TRUE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a continuous response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`harden` Flag for consistent API.

`smart_cold_start_default` Flag for consistent API.

`InferenceContinLin$compute_estimate()`: Fits Lin's covariate-adjusted OLS model (`stats`'s `lm.fit` on the centered-covariate design matrix) and returns $\hat{\beta}_T$, the estimated average treatment effect. A design matrix that is rank-deficient or has fewer usable rows than columns, or a fit with non-finite coefficients, is cached as nonestimable rather than returned.

Usage:

```
InferenceContinLin$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If `TRUE`, skip HC2 variance computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferenceContinLin$compute_estimate_with_bootstrap_weights()`: Refits Lin's model with subject/block-level weights applied to a weighted least-squares fit (`stats::lm.wfit`) — Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded to row level via `private$expand_subject_or_bl` — and returns the reweighted estimate $\hat{\beta}_T^{(w)}$. When `estimate_only = FALSE`, also computes a weighted-residual variance estimate (not the HC2 estimator used by `compute_asymp_confidence_interval()`) for internal bootstrap diagnostics. Rows with non-finite or non-positive weight, or non-finite response, are dropped from the weighted fit; if no rows remain, or the fit fails, the estimate is `NA`.

Usage:

```
InferenceContinLin$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferenceContinLin$compute_asymp_confidence_interval()`: Wald confidence interval for β_T using the HC2 heteroskedasticity-robust standard error (`ols_hc2_post_fit_cpp`); see `InferenceAsymp` for the shared t/z interval contract. Fits the model first if not already cached.

Usage:

```
InferenceContinLin$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level. The default is 0.05.

`InferenceContinLin$compute_asymp_two_sided_pval()`: Two-sided Wald test of $H_0 : \beta_T =$ `delta` using the HC2 heteroskedasticity-robust standard error; see `InferenceAsymp` for the shared t/z test contract. Fits the model first if not already cached.

Usage:

```
InferenceContinLin$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment effect. Defaults to 0.

`InferenceContinLin$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceContinLin$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Lin, W. (2013). "Agnostic notes on regression adjustments to experimental data: Reexamining Freedman's critique." *The Annals of Applied Statistics*, 7(1), 295-318, doi:10.1214/12AOAS583.

See Also

`InferenceContinOLS` for the uncentered, non-interacted OLS estimator this class generalizes.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinLin$new(seq_des)
inf$compute_estimate()
```

InferenceContinOLS *OLS Inference for Continuous Responses*

Description

Fits an ordinary least squares regression for continuous responses: $Y_i = \beta_0 + \beta_T W_i + X_i^\top \gamma + \epsilon_i$, using the treatment indicator and, optionally, all recorded covariates as predictors (uncentered, no treatment-covariate interactions — see `InferenceContinLin` for the centered-covariate, interacted variant). $\hat{\beta}_T$ is a mean difference on the response's natural scale. `likelihood_tier = "full"`: Wald, score, gradient, and likelihood-ratio tests are all available (`fast_ols_cpp`/`fast_ols_with_var_cpp`), with the parametric-likelihood bootstrap using the OLS Gaussian-errors model as the generative null. Standard errors are the closed-form OLS variance under homoskedastic errors (unlike `InferenceContinLin`'s HC2 heteroskedasticity-robust SE). Warm starts are disabled (`fit_warm_start_enabled = FALSE` set at construction) because OLS is a closed-form estimator and gains nothing from an iterative optimizer's warm-started initial values. Validity requires the usual OLS assumptions: correctly specified linear predictor, and (for the asymptotic/likelihood inference path specifically) homoskedastic, approximately normal errors; the randomization-inference path relies only on randomization of W.

Super class

`Inference` -> `InferenceContinOLS`

Methods

Public methods:

- `InferenceContinOLS$new()`
- `InferenceContinOLS$compute_estimate()`
- `InferenceContinOLS$compute_estimate_with_bootstrap_weights()`

- InferenceContinOLS$compute_asymp_confidence_interval()
- InferenceContinOLS$compute_asymp_two_sided_pval()
- InferenceContinOLS$clone()

InferenceContinOLS$new(): Initialize inference for the OLS model $Y_i = \beta_0 + \beta_T W_i + X_i^\top \gamma + \epsilon_i$; see InferenceContinOLS for the model form. Disables warm-started optimizer initial values (not applicable to this closed-form estimator). Does not fit the model; the fit is deferred to the first call to compute_estimate() or a method that requires it.

Usage:

```
InferenceContinOLS$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  max_resample_attempts = 50L,
  harden = TRUE,
  smart_cold_start_default = NULL
)
```

Arguments:

des_obj A completed Design object with a continuous response.

model_formula Optional formula for covariate adjustment. If NULL (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

verbose Whether to print progress messages.

max_resample_attempts Maximum number of times a single bootstrap replicate may be re-drawn when the drawn sample fails validity screening. Default 50L.

harden Whether to apply robustness measures.

smart_cold_start_default Flag for consistent API.

InferenceContinOLS$compute_estimate(): Fits the OLS model (fast_ols_cpp/fast_ols_with_var_cpp) and returns $\hat{\beta}_T$. If not hardened (private$harden == FALSE), fits directly on the full design matrix; otherwise uses QR column-dropping hardening to handle rank-deficient designs. A fit with a non-finite treatment coefficient is cached as nonestimable rather than returned.

Usage:

```
InferenceContinOLS$compute_estimate(estimate_only = FALSE)
```

Arguments:

estimate_only If TRUE, skip variance component calculations.

InferenceContinOLS$compute_estimate_with_bootstrap_weights(): Refits the OLS model with subject/block-level weights applied to a weighted least-squares fit (stats::lm.wfit) — Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded to row level via private$expand_subject_or_bl — and returns the reweighted estimate $\hat{\beta}_T^{(w)}$. When estimate_only = FALSE, also computes a weighted-residual variance estimate for internal bootstrap diagnostics. Rows with non-finite or non-positive weight, or non-finite response, are dropped from the weighted fit; if no rows remain, or the fit fails, the estimate is NA.

Usage:

```
InferenceContinOLS$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferenceContinOLS$compute_asymp_confidence_interval()`: Uses the shared asymptotic confidence-interval contract; see `InferenceAsymp`.

Usage:

```
InferenceContinOLS$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`InferenceContinOLS$compute_asymp_two_sided_pval()`: Uses the shared asymptotic two-sided p-value contract; see `InferenceAsymp`.

Usage:

```
InferenceContinOLS$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null difference to test against. Default is zero.

`InferenceContinOLS$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceContinOLS$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Rosenbaum, P. R. (2002). *Observational Studies* (2nd ed.). Springer, for the OLS mean-difference estimator's design-based justification under randomization.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinOLS$new(seq_des)
inf$compute_estimate()
```

`InferenceContinQuantileRegr`

Quantile Regression Inference for Continuous Responses

Description

Fits a linear quantile regression, $Q_\tau(Y_i \mid x_i) = x_i^\top \beta_\tau$, for a continuous response, estimating β_τ by minimizing the asymmetric ("pinball" / "check") loss

$$\hat{\beta}_\tau = \arg\min_\beta \sum_{i=1}^{n} \rho_\tau(y_i - x_i^\top \beta), \qquad \rho_\tau(u) = u\,(\tau - \mathbb{1}[u < 0])\,,$$

via **quantreg**'s simplex method (`quantreg::rq/rq.fit(...,method = "br")`). The treatment coefficient is the estimated shift in the τ-th conditional quantile of the response attributable to treatment, holding any other covariates in `model_formula` fixed; by default `tau = 0.5`, so this is median regression (robust to outliers and distributional skew relative to mean-based estimators, at the cost of losing the mean-shift interpretation away from $\tau = 0.5$).

Standard errors use **quantreg**'s Powell-style `"nid"` (non-i.i.d., kernel-based sparsity/local-density estimator) sandwich covariance when available, with fallback to the simpler `"iid"` estimator if needed. Inference (confidence intervals, p-values) is based on the resulting asymptotic normal approximation, not an exact finite-sample distribution.

This class requires the **quantreg** package, which is listed under `Suggests` and is not installed automatically with **EDI**. Install **quantreg** manually before use.

Super class

`Inference` -> `InferenceContinQuantileRegr`

Methods

Public methods:

- `InferenceContinQuantileRegr$new()`
- `InferenceContinQuantileRegr$compute_estimate()`
- `InferenceContinQuantileRegr$compute_estimate_with_bootstrap_weights()`
- `InferenceContinQuantileRegr$compute_asymp_confidence_interval()`
- `InferenceContinQuantileRegr$compute_asymp_two_sided_pval()`
- `InferenceContinQuantileRegr$clone()`

`InferenceContinQuantileRegr$new():` Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize a quantile-regression inference object for a completed design with a continuous, uncensored response. Requires the **quantreg** package to be installed.

Usage:

```
InferenceContinQuantileRegr$new(
  des_obj,
  model_formula = NULL,
  tau = 0.5,
  verbose = FALSE
)
```

Arguments:

`des_obj` A completed `Design` object with a continuous response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`tau` The quantile $\tau \in (0, 1)$ to estimate (default 0.5, i.e. median regression).

`verbose` Whether to print progress messages. Default `FALSE`.

`InferenceContinQuantileRegr$compute_estimate()`: Computes the treatment coefficient $\hat{\beta}_{T,\tau}$ from a check-loss quantile regression fit at quantile `tau` (see class documentation for the full model). Rank-deficient covariate columns are dropped before fitting (see `private$reduce_design_matrix_for_quantil` returns `NA` if the reduced design has no usable treatment column or too few residual degrees of freedom.

Usage:

```
InferenceContinQuantileRegr$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`InferenceContinQuantileRegr$compute_estimate_with_bootstrap_weights()`: Recomputes the quantile-regression treatment estimate under subject/block bootstrap weights (`quantreg::rq(..., weights = row_weights)`), used by the Bayesian bootstrap and related weighted-resampling machinery; see `InferenceBayesianBootstrap`. Unlike `$compute_estimate()`, this always reduces the design matrix from scratch (`reuse_factorizations = FALSE`) rather than reusing a cached rank-reduction from a prior warm-started fit.

Usage:

```
InferenceContinQuantileRegr$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferenceContinQuantileRegr$compute_asymp_confidence_interval()`: Computes a $1 - \alpha$ level confidence interval for the quantile-regression treatment coefficient $\hat{\beta}_{T,\tau}$, using **quantreg**'s Powell-style `"nid"` asymptotic standard error (falling back to `"iid"` if unavailable — see class documentation) and residual degrees of freedom $n - p$. See `InferenceAsymp` for the shared asymptotic confidence-interval contract this delegates to.

Usage:

`InferenceContinQuantileRegr$compute_asymp_confidence_interval(alpha = 0.05)`

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`InferenceContinQuantileRegr$compute_asymp_two_sided_pval()`**:** Computes a two-sided Wald p-value testing $H_0 : \beta_{T,\tau} =$ `delta`, from the same **quantreg** sandwich standard error and degrees of freedom used by `$compute_asymp_confidence_interval()`. See `InferenceAsymp` for the shared asymptotic two-sided p-value contract this delegates to.

Usage:

`InferenceContinQuantileRegr$compute_asymp_two_sided_pval(delta = 0)`

Arguments:

`delta` The null difference to test against. Default is zero.

`InferenceContinQuantileRegr$clone()`**:** The objects of this class are cloneable with this method.

Usage:

`InferenceContinQuantileRegr$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

References

Koenker, R., and Bassett, G. (1978). "Regression Quantiles." *Econometrica*, 46(1), 33-50, doi:10.2307/1913643, for the check-loss quantile regression estimator; Koenker, R. (2005). *Quantile Regression*, Cambridge University Press, for the Powell-style sandwich standard error estimators used here.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinQuantileRegr$new(seq_des)
inf$compute_estimate()
```

`InferenceContinRobustRegr`

Robust (M/MM-Estimator) Regression Inference for Continuous Responses

---

Description

Fits a robust linear regression — by default via this package's fast_robust_regression_cpp C++ backend (use_rcpp = TRUE; see that page for the full M/MM-estimator model, weight functions, and asymptotic variance formula), or via MASS::rlm when use_rcpp = FALSE — for a continuous response using the treatment indicator and, optionally, all recorded covariates as predictors. This provides a Huber/MM-style robustness upgrade over ordinary least squares when outcomes are heavy-tailed or outlier-prone, down-weighting large residuals rather than letting them dominate the fit the way squared-error loss does.

The method argument is passed through to either backend and may be either "M" (Huber's psi function) or "MM" (Tukey's bisquare weight, the default — higher breakdown point than "M" at some cost in asymptotic efficiency under normality). When use_rcpp = TRUE (the default), coefficient standard errors come from the C++ backend's own M-estimator asymptotic variance (ssq_b_j); when FALSE, they come from the coefficient table returned by summary.rlm(). Either way, confidence intervals and p-values use that standard error with residual degrees of freedom $n - p$ in a normal-theory Wald approximation (not an exact finite-sample distribution).

Super class

Inference -> InferenceContinRobustRegr

Methods

Public methods:

- InferenceContinRobustRegr$new()
- InferenceContinRobustRegr$compute_estimate()
- InferenceContinRobustRegr$compute_estimate_with_bootstrap_weights()
- InferenceContinRobustRegr$compute_asymp_confidence_interval()
- InferenceContinRobustRegr$compute_asymp_two_sided_pval()
- InferenceContinRobustRegr$clone()

InferenceContinRobustRegr$new(): Uses the shared randomization two-sided p-value contract; see InferenceRand.

Initialize a robust-regression inference object for a completed design with a continuous, uncensored response.

Usage:

```
InferenceContinRobustRegr$new(
  des_obj,
  model_formula = NULL,
  method = "MM",
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

des_obj A completed Design object with a continuous response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`method` Robust estimation method, `"M"` (Huber) or `"MM"` (Tukey bisquare, higher breakdown point). Default `"MM"`.

`use_rcpp` Whether to use the `fast_robust_regression_cpp` C++ backend (`TRUE`, default) instead of `MASS::rlm` (`FALSE`).

`verbose` Whether to print progress messages. Default `FALSE`.

`smart_cold_start_default` Whether to use smart starting values for the optimizer.

`InferenceContinRobustRegr$compute_estimate():` Computes the robust-regression treatment coefficient $\hat{\beta}_T$ from an M/MM-estimator fit (see class documentation for the full model and `use_rcpp` backend choice). Rank-deficient covariate columns are dropped before fitting via `private$fit_with_hardened_qr_column_dropping()`.

Usage:

```
InferenceContinRobustRegr$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`InferenceContinRobustRegr$compute_estimate_with_bootstrap_weights():` Recomputes the robust-regression treatment estimate under subject/block bootstrap weights, used by the Bayesian bootstrap and related weighted-resampling machinery; see `InferenceBayesianBootstrap`. When `use_rcpp = TRUE`, reproduces `MASS::rlm`'s default `wt.method = "inv.var"` weighting exactly by pre-multiplying `X` and `y` by $\sqrt{\text{weight}}$ and running unweighted M-estimation on the transformed data via the C++ backend (rather than adding native weight support to that backend); when `use_rcpp = FALSE`, passes `weights` directly to `MASS::rlm`. This variant never populates a standard error or degrees of freedom (both left `NA`) — it is estimate-only by construction, matching the `estimate_only` default of the fast path it always uses internally.

Usage:

```
InferenceContinRobustRegr$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferenceContinRobustRegr$compute_asymp_confidence_interval():` Computes a $1-\alpha$ level confidence interval for the robust-regression treatment coefficient $\hat{\beta}_T$, using the M/MM-estimator's asymptotic standard error (see class documentation for its source depending on `use_rcpp`) and residual degrees of freedom $n-p$ in a normal-theory Wald approximation. See `InferenceAsymp` for the shared asymptotic confidence-interval contract this delegates to.

Usage:

```
InferenceContinRobustRegr$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`InferenceContinRobustRegr$compute_asymp_two_sided_pval()`: Computes a two-sided Wald p-value testing $H_0 : \beta_T =$ `delta`, from the same M/MM-estimator standard error and residual degrees of freedom used by `$compute_asymp_confidence_interval()`. See `InferenceAsymp` for the shared asymptotic two-sided p-value contract this delegates to.

Usage:

```
InferenceContinRobustRegr$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null difference to test against. Default is zero.

`InferenceContinRobustRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceContinRobustRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'continuous')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(10))
inf = InferenceContinRobustRegr$new(seq_des)
inf$compute_estimate()
```

`InferenceCountHurdleNegBin`

Hurdle Negative Binomial Regression Inference for Count Responses

---

Description

Fits a hurdle negative binomial regression for count responses: a binary hurdle submodel $P(Y_i > 0) = \text{logit}^{-1}(X_i^{h\top}\gamma^h)$ (fit jointly with the count submodel) crossed with a zero-truncated negative-binomial count submodel for $Y_i \mid Y_i > 0$: $\log E[Y_i \mid Y_i > 0, w_i, x_i] = \beta_0 + \beta_T w_i + x_i^\top \gamma$, $\text{Var}(Y_i \mid Y_i > 0) = \mu_i + \mu_i^2/\theta$ (`fast_hurdle_negbin_cpp`/ `fast_hurdle_negbin_with_var_cpp`). The hurdle and count submodels may use different covariate formulas (`model_formula`/`model_formula_hurdle`). The reported treatment effect is the coefficient from the conditional (truncated, $Y > 0$) count component, on the log-rate scale, **conditional on clearing the hurdle**: it is not the effect on the unconditional mean $E[Y]$, which also depends on how treatment shifts the hurdle-crossing probability. A marginal (unconditional-mean) estimand is not yet implemented for this class (see `marginal_estimand_report.md`). `likelihood_tier = "full"`: Wald, gradient, and (bootstrap-calibrated) likelihood-ratio tests are available for the count submodel's treatment coefficient; a plain score test is not exposed. **Jackknife inference is not supported**: delete-one refits of this two-part model with a jointly-estimated dispersion parameter are numerically unstable, so `compute_jackknife_estimate()` and related methods report explicit non-estimability rather than attempting delete-one refits.

Super class

Inference -> InferenceCountHurdleNegBin

Methods

Public methods:

- InferenceCountHurdleNegBin$new()
- InferenceCountHurdleNegBin$compute_asymp_confidence_interval()
- InferenceCountHurdleNegBin$compute_asymp_two_sided_pval()
- InferenceCountHurdleNegBin$compute_gradient_two_sided_pval()
- InferenceCountHurdleNegBin$compute_gradient_confidence_interval()
- InferenceCountHurdleNegBin$compute_estimate_with_bootstrap_weights()
- InferenceCountHurdleNegBin$compute_jackknife_estimate()
- InferenceCountHurdleNegBin$compute_jackknife_bias_estimate()
- InferenceCountHurdleNegBin$compute_jackknife_std_error()
- InferenceCountHurdleNegBin$compute_jackknife_wald_two_sided_pval()
- InferenceCountHurdleNegBin$compute_jackknife_wald_confidence_interval()
- InferenceCountHurdleNegBin$clone()

InferenceCountHurdleNegBin$new(): Initialize inference for the two-part hurdle negative binomial model (binary hurdle submodel plus zero-truncated negative-binomial count submodel); see InferenceCountHurdleNegBin for the model form. Does not fit the model; the fit is deferred to the first call to compute_estimate() or a method that requires it.

Usage:

```
InferenceCountHurdleNegBin$new(
  des_obj,
  model_formula = NULL,
  model_formula_hurdle = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

des_obj A completed Design object.

model_formula Optional formula for covariate adjustment.

model_formula_hurdle Formula for the hurdle submodel. If NULL (default), it uses the same formula as model_formula.

verbose A flag indicating whether messages should be displayed.

smart_cold_start_default Whether to use smart cold start values.

InferenceCountHurdleNegBin$compute_asymp_confidence_interval(): Compute the hurdle negative-binomial asymptotic confidence interval for the treatment coefficient, using the shared count-likelihood semantics documented in InferenceCountLikelihood.

Usage:

```
InferenceCountHurdleNegBin$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The significance level (default 0.05).

`InferenceCountHurdleNegBin$compute_asymp_two_sided_pval()`: Compute the hurdle negative-binomial asymptotic two-sided p-value for the treatment coefficient, falling back through the shared count-likelihood machinery when needed; see `InferenceCountLikelihood`.

Usage:

```
InferenceCountHurdleNegBin$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment effect (default 0).

`InferenceCountHurdleNegBin$compute_gradient_two_sided_pval()`: Gradient test of H_0 : $\beta_T =$ `delta` on the truncated count submodel's treatment coefficient (a score-test variant using the observed rather than expected information); see `InferenceCountLikelihood` for the shared likelihood-test dispatch.

Usage:

```
InferenceCountHurdleNegBin$compute_gradient_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment effect (default 0).

`InferenceCountHurdleNegBin$compute_gradient_confidence_interval()`: Compute a hurdle negative-binomial likelihood-based confidence interval by inverting the configured likelihood test. See `InferenceCountLikelihood` for related score, likelihood-ratio, and gradient methods.

Usage:

```
InferenceCountHurdleNegBin$compute_gradient_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The significance level (default 0.05).

`InferenceCountHurdleNegBin$compute_estimate_with_bootstrap_weights()`: Refits the hurdle negative-binomial model with subject/ block-level weights (Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded to row level via `private$expand_subject_or_block_weights_to_row_weights()`) via **glmmTMB**'s `glmmTMB(family = truncated_nbinom2())` (not the package's internal C++ solver, which has no weighted variant for this model), and returns the reweighted conditional-count log-rate-ratio estimate $\hat{\beta}_T^{(w)}$. Requires the **glmmTMB** package; errors if unavailable. No standard error is computed (`s_beta_hat_T` is always `NA`). A fit that fails, or whose fitted treatment coefficient is missing or non-finite, is cached as nonestimable and returns `NA`.

Usage:

```
InferenceCountHurdleNegBin$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferenceCountHurdleNegBin$compute_jackknife_estimate()`: Hurdle negative-binomial delete-one refits are unstable for jackknife inference; report explicit non-estimability.

Usage:

```
InferenceCountHurdleNegBin$compute_jackknife_estimate(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.

`InferenceCountHurdleNegBin$compute_jackknife_bias_estimate()`: Report that the jackknife bias estimate is unavailable for hurdle negative-binomial fits because delete-one refits are unstable; see `InferenceJackknife` for the shared jackknife contract.

Usage:

```
InferenceCountHurdleNegBin$compute_jackknife_bias_estimate(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.

`InferenceCountHurdleNegBin$compute_jackknife_std_error()`: Report that the jackknife standard error is unavailable for hurdle negative-binomial fits because delete-one refits are unstable; see `InferenceJackknife` for the shared jackknife contract.

Usage:

```
InferenceCountHurdleNegBin$compute_jackknife_std_error(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.

`InferenceCountHurdleNegBin$compute_jackknife_wald_two_sided_pval()`: Reports that jackknife-Wald p-values are unavailable here; see `InferenceJackknife`.

Usage:

```
InferenceCountHurdleNegBin$compute_jackknife_wald_two_sided_pval(
  delta = 0,
  unit = "auto"
)
```

Arguments:

`delta` Null treatment-effect value. Default 0.

`unit` Deletion unit. Default `"auto"`.

`InferenceCountHurdleNegBin$compute_jackknife_wald_confidence_interval()`: Reports that jackknife-Wald intervals are unavailable here; see `InferenceJackknife`.

Usage:

```
InferenceCountHurdleNegBin$compute_jackknife_wald_confidence_interval(
  alpha = 0.05,
  unit = "auto"
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`unit` Deletion unit. Default `"auto"`.

`InferenceCountHurdleNegBin$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountHurdleNegBin$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Mullahy, J. (1986). "Specification and Testing of Some Modified Count Data Models." *Journal of Econometrics*, 33(3), 341-365, doi:10.1016/03044076(86)900023, for the hurdle count-model framework.

See Also

`InferenceCountNegBin` for the single-part negative binomial model this class's count submodel generalizes to two parts.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountHurdleNegBin$new(seq_des, model_formula = ~ x1)
inf$compute_estimate()
```

`InferenceCountHurdlePoisson`

Hurdle Poisson Regression Inference for Count Responses

Description

Fits a hurdle Poisson regression for count responses: a binary hurdle submodel $P(Y_i > 0) = \text{logit}^{-1}(X_i^{h\top}\gamma^h)$ (fit jointly with the count submodel) crossed with a zero-truncated Poisson count submodel for $Y_i \mid Y_i > 0$: $\log E[Y_i \mid Y_i > 0, w_i, x_i] = \beta_0 + \beta_T w_i + x_i^\top \gamma$. The hurdle and count submodels may use different covariate formulas (`model_formula`/`model_formula_hurdle`). The reported treatment effect is the coefficient from the conditional (truncated, $Y > 0$) count component, on the log-rate scale, **conditional on clearing the hurdle**: it is not the effect on the unconditional mean $E[Y]$, which also depends on how treatment shifts the hurdle-crossing probability, under the default `estimand = "conditional"`. `likelihood_tier = "full"`: Wald, gradient, and (bootstrap-calibrated) likelihood-ratio tests are available for the count submodel's treatment coefficient under

that estimand; a plain score test is not exposed. **Jackknife inference is not supported**: delete-one refits of this two-part model are numerically unstable, so `compute_jackknife_estimate()` and related methods report explicit non-estimability rather than attempting delete-one refits. Unlike `InferenceCountHurdleNegBin`, the count submodel here assumes Poisson (equidispersion) conditional on clearing the hurdle, with no separate dispersion parameter.

Marginal (unconditional-mean) estimand. Via `set_estimand()`, this class also supports `estimand = "marginal_mean_diff"` and `"marginal_ratio"`: the g-computation average, over the empirical covariate distribution, of the model-implied unconditional mean $E[Y_i \mid w_i, x_i] = (1 - \pi(x_i)) \cdot \lambda(x_i)/(1 - e^{-\lambda(x_i)})$ — the hurdle-crossing probability times the **zero-truncated** Poisson mean, $E[Y \mid Y > 0] = \lambda/(1 - e^{-\lambda})$ (exact for Poisson: truncating at 0 changes the normalizing constant but not the rate parameter λ; see Cameron and Trivedi, *Regression Analysis of Count Data*, ch. 4.2) — at $w_i = 1$ vs. $w_i = 0$. A pure post-fit transform of the same maximum-likelihood fit (no refit), with a delta-method standard error against the sandwich-robust covariance matrix already used for this class's conditional Wald inference. Only `"wald"`-type inference is available under a marginal estimand.

Super classes

`Inference` -> `InferenceCountZeroAugmentedPoissonAbstract` -> `InferenceCountHurdlePoisson`

Methods

Public methods:

- `InferenceCountHurdlePoisson$new()`
- `InferenceCountHurdlePoisson$compute_estimate()`
- `InferenceCountHurdlePoisson$compute_asymp_confidence_interval()`
- `InferenceCountHurdlePoisson$compute_asymp_two_sided_pval()`
- `InferenceCountHurdlePoisson$clone()`

`InferenceCountHurdlePoisson$new()`: Initialize inference for the hurdle Poisson model (binary hurdle submodel plus zero-truncated Poisson count submodel); see `InferenceCountHurdlePoisson` for the model form. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferenceCountHurdlePoisson$new(
  des_obj,
  model_formula = NULL,
  model_formula_hurdle = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL,
  optimization_alg = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a count response.

`model_formula` Optional formula for the count submodel.

`model_formula_hurdle` Formula for the hurdle submodel. If `NULL` (default), it uses the same formula as `model_formula`.

`use_rcpp` Logical. If `TRUE` (default), use the internal Rcpp implementation. If `FALSE`, use **glmmTMB**.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`optimization_alg` Optimization algorithm. Default is dispatched via policy.

`InferenceCountHurdlePoisson$compute_estimate()`: Fits the hurdle Poisson model. Under the default `estimand = "conditional"`, returns $\hat{\beta}_T$, the treatment log-rate coefficient from the zero-truncated count submodel (conditional on clearing the hurdle). Under `estimand = "marginal_mean_diff"` or `"marginal_ratio"` (set via `set_estimand()`), returns the g-computation marginal mean difference or log-scale marginal ratio of the unconditional mean instead — see the class-level `@details` for the formula. A pure post-fit transform of the same cached fit, no refit.

Usage:

```
InferenceCountHurdlePoisson$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip standard-error computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferenceCountHurdlePoisson$compute_asymp_confidence_interval()`: Asymptotic confidence interval. Under the conditional estimand, delegates to the shared zero-augmented count-model Wald/ bootstrap-fallback contract; under a marginal estimand, the delta-method interval computed by `compute_estimate()`. Calls `self$compute_estimate()` first (not `private$shared()` directly) so the estimand-aware cache is always current regardless of call order.

Usage:

```
InferenceCountHurdlePoisson$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The significance level (default 0.05).

`InferenceCountHurdlePoisson$compute_asymp_two_sided_pval()`: Asymptotic two-sided p-value, dispatched exactly as `compute_asymp_confidence_interval()`; see that method's description for the marginal-estimand path.

Usage:

```
InferenceCountHurdlePoisson$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment effect under the current estimand (default 0).

`InferenceCountHurdlePoisson$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountHurdlePoisson$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Mullahy, J. (1986). "Specification and Testing of Some Modified Count Data Models." *Journal of Econometrics*, 33(3), 341-365, doi:10.1016/03044076(86)900023, for the hurdle count-model framework.

See Also

`InferenceCountPoisson` for the single-part Poisson model this class's count submodel generalizes to two parts; `InferenceCountHurdleNegBin` for the overdispersion-robust negative-binomial variant (does not support a marginal estimand — the mean-function derivation here is Poisson-specific).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountHurdlePoisson$new(seq_des)
inf$compute_estimate()
```

`InferenceCountKKCondPoissonOneLik`

One-Likelihood Conditional-Poisson Inference for KK Count Designs

---

Description

Estimates a treatment log-rate-ratio β_T for count outcomes collected under a KK matching-on-the-fly design (`DesignSeqOneByOneKK14` or subclass) by maximizing a single combined likelihood that couples a conditional (within-matched-pair, intercept-free) Poisson likelihood for matched subjects with an ordinary Poisson likelihood for reservoir subjects, sharing one treatment coefficient across both pieces. This is the "one-likelihood" alternative to the inverse-variance-weighted combination (`...IVWC` pattern used elsewhere in the KK family): rather than fitting matched and reservoir models separately and pooling by inverse-variance weights, the treatment coefficient here is estimated jointly from the full combined log-likelihood, and its standard error, score, likelihood-ratio, and gradient statistics are all "design-conservative" – each is the pointwise-wider of the model-based asymptotic quantity and a design-based quantity computed by treating the estimate as a plug-in statistic under `InferenceAsymp`'s z/t machinery, so inference never overstates precision relative to the design alone.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok"

row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Estimand. β_T, the treatment coefficient in a log-linear (Poisson) mean model $E[Y \mid w, x] = \exp(\beta_0 + \beta_T w + x\beta)$, interpreted as a log rate ratio (equivalently, $\exp(\hat{\beta}_T)$ is the treatment-vs-control incidence rate ratio).

Model. Matched subjects contribute a conditional-Poisson term that eliminates the pair-specific nuisance intercept by conditioning on the pair total count (removing the need to estimate one intercept per pair); reservoir subjects contribute an ordinary Poisson log-likelihood with a single shared intercept. Both pieces are summed into one combined negative log-likelihood and maximized jointly in $(\beta_0, \beta_T, \beta)$ (see `get_cpoisson_combined_hessian_cpp` and `fast_cpoisson_combined_with_var_cpp` for the backend fitting contract). `likelihood_tier = "full"`, so likelihood-ratio, score, and gradient tests and a parametric likelihood bootstrap are all available in addition to the design-conservative Wald path.

Assumptions. Independence of counts across matched pairs and reservoir subjects given covariates; correct log-linear mean specification; a KK matching-on-the-fly design supplying the matched/reservoir partition. No response censoring is supported (checked at construction via `assertNoCensoring()`).

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceCountKKCondPoissonOneLik`

Methods

Public methods:

- `InferenceCountKKCondPoissonOneLik$approximate_randomization_distribution_beta_hat_T()`
- `InferenceCountKKCondPoissonOneLik$supports_rand_pval_for_incidence()`
- `InferenceCountKKCondPoissonOneLik$compute_rand_two_sided_pval()`
- `InferenceCountKKCondPoissonOneLik$clone()`

`InferenceCountKKCondPoissonOneLik$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceCountKKCondPoissonOneLik$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceCountKKCondPoissonOneLik$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceCountKKCondPoissonOneLik$supports_rand_pval_for_incidence()
```

`InferenceCountKKCondPoissonOneLik$compute_rand_two_sided_pval()`:

Usage:

```
InferenceCountKKCondPoissonOneLik$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceCountKKCondPoissonOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountKKCondPoissonOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A. and Krieger, A. (2014). "Matching on-the-fly: A group sequential covariate balanced randomization procedure." *arXiv preprint* arXiv:1305.6259. (KK14 in `REFERENCES.md`.)

See Also

Analogous Python API for count models: statsmodels discrete models (`ConditionalPoisson`, `Poisson`). Poisson regression (orientation).

`InferenceCountKKGLMM` *GLMM Inference for KK Designs with Count Response*

Description

Fits a Poisson GLMM for count responses under a KK matching-on-the-fly design. The random intercept per matched pair is integrated out via Gauss-Hermite quadrature.

When `use_rcpp = TRUE` (default) the likelihood is maximised by an internal Rcpp routine. Set `use_rcpp = FALSE` to fall back to **glmmTMB**.

Model. $Y_{ij} \mid b_i \sim \mathrm{Poisson}(\mu_{ij})$ with $\log \mu_{ij} = X_{ij}'\beta + \beta_T \cdot W_{ij} + b_i$, where i indexes matched pairs, $j \in \{1, 2\}$ the two subjects within a pair, W_{ij} is the treatment indicator, and $b_i \sim \mathcal{N}(0, \sigma_b^2)$ is a pair-level random intercept absorbing within-pair correlation induced by matching. β_T is a log-rate (log relative risk) treatment effect: $\exp(\hat{\beta}_T)$ is the estimated rate ratio. The random effect is integrated out of the marginal likelihood by adaptive Gauss-Hermite quadrature rather than a Laplace approximation.

Likelihood tier. `likelihood_tier = "full"`: both Wald (model-based standard error) and likelihood-ratio testing types are available. Because the GLMM likelihood alone does not encode the KK design's matched-pair randomization structure, the likelihood-ratio CI/p-value are conservatively widened/calibrated against the design-aware Wald result (see `compute_lik_ratio_confidence_interval()`/ `compute_lik_ratio_two_sided_pval()`) so the model-based test is never anti-conservative relative to the design.

Assumptions. Count response modeled as conditionally Poisson given the random intercept (equidispersion conditional on b_i); pair-level random effects independent across pairs; a KK matching-on-the-fly design.

Super class

`Inference` -> `InferenceCountKKGLMM`

Methods

Public methods:

- `InferenceCountKKGLMM$new()`
- `InferenceCountKKGLMM$compute_estimate()`
- `InferenceCountKKGLMM$compute_estimate_with_bootstrap_weights()`
- `InferenceCountKKGLMM$compute_wald_confidence_interval()`
- `InferenceCountKKGLMM$compute_wald_two_sided_pval()`
- `InferenceCountKKGLMM$compute_lik_ratio_confidence_interval()`
- `InferenceCountKKGLMM$compute_lik_ratio_two_sided_pval()`
- `InferenceCountKKGLMM$compute_asymp_confidence_interval()`
- `InferenceCountKKGLMM$compute_asymp_two_sided_pval()`
- `InferenceCountKKGLMM$clone()`

`InferenceCountKKGLMM$new()`: Initialize a KK Poisson-GLMM inference object for a matched-pair KK design with a count response and prepare the matched-pair random-intercept likelihood machinery; see the class topic for the model.

Usage:

```
InferenceCountKKGLMM$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  optimization_alg = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed KK matching-on-the-fly `Design` object (`DesignSeqOneByOneKK14` or subclass) with a count response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used.

`use_rcpp` Logical. If `TRUE` (default), maximize the Gauss-Hermite-quadrature marginal likelihood with the internal Rcpp Poisson-GLMM routine; if `FALSE`, fall back to **glmmTMB**.

`optimization_alg` Optimization algorithm passed to the likelihood maximizer. If `NULL` (default), an algorithm is dispatched via the package's optimizer policy.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart starting values for the optimizer.

`InferenceCountKKGLMM$compute_estimate()`: Point estimate of the treatment log-rate coefficient β_T from a Poisson GLMM with a matched-pair random intercept, fit by maximizing the Gauss-Hermite-quadrature-integrated marginal likelihood (internal Rcpp routine when `use_rcpp = TRUE`, else **glmmTMB**). See the class topic for the model form.

Usage:

```
InferenceCountKKGLMM$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If `TRUE`, skip variance-component calculations.

Returns: Numeric scalar: the treatment coefficient on the log-rate (link) scale, i.e. $\exp(\hat{\beta}_T)$ is a rate ratio.

`InferenceCountKKGLMM$compute_estimate_with_bootstrap_weights()`: Recomputes the KK Poisson-GLMM treatment estimate under nonparametric/ Bayesian-bootstrap subject-or-block weights, refitting the weighted GLMM (`compute_weighted_glmm_bootstrap_estimate()`). Standard error, degrees of freedom, and the cached summary table are cleared/set to `NA`/`Inf`/ `NULL` since only the point estimate is meaningful under resampling weights. Falls back to the unweighted point estimate when the weights are effectively constant.

Usage:

```
InferenceCountKKGLMM$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Numeric vector of nonnegative bootstrap replicate weights, one per subject or per matched block (KK match structure).

`estimate_only` If `TRUE`, compute only the weighted point estimate (this method never computes a weighted standard error regardless of this argument).

Returns: Numeric scalar treatment-effect estimate (log-rate scale) under the given weights.

`InferenceCountKKGLMM$compute_wald_confidence_interval()`: Wald confidence interval for the treatment log-rate coefficient: $\hat{\beta}_T \pm t_{1-\alpha/2,\,df} \cdot \hat{se}(\hat{\beta}_T)$, using the model-based GLMM standard error. See `InferenceAsymp` for the shared contract.

Usage:

```
InferenceCountKKGLMM$compute_wald_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

Returns: A length-2 numeric vector `c(lower, upper)` on the log-rate scale.

`InferenceCountKKGLMM$compute_wald_two_sided_pval()`: Two-sided Wald p-value for $H_0 : \beta_T =$ `delta` vs. $H_1 : \beta_T \neq$ `delta`, using the model-based GLMM standard error.

Usage:

```
InferenceCountKKGLMM$compute_wald_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null value of β_T to test against; 0 (the default) tests for any treatment effect at all.

Returns: Numeric scalar p-value in $[0, 1]$.

`InferenceCountKKGLMM$compute_lik_ratio_confidence_interval()`: Likelihood-ratio confidence interval for the treatment log-rate coefficient, inverting the GLMM's profile likelihood-ratio test against χ^2_1. Because the GLMM likelihood does not itself account for the KK matched-pair design's randomization structure, this interval is conservatively widened to be at least as wide as the design-aware Wald interval (`compute_wald_confidence_interval()`) via `.conservative_kk_onelik_ci()` – guarding against the model-based interval being anti-conservative relative to the design.

Usage:

```
InferenceCountKKGLMM$compute_lik_ratio_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

Returns: A length-2 numeric vector `c(lower, upper)` on the log-rate scale.

`InferenceCountKKGLMM$compute_lik_ratio_two_sided_pval()`: Two-sided likelihood-ratio p-value for $H_0 : \beta_T =$ `delta`, from the GLMM's profile likelihood-ratio test referred to χ^2_1. As with `compute_lik_ratio_confidence_interval()`, this is conservatively calibrated (via `.conservative_kk_onelik_pval()`) against the design-aware Wald p-value so the model-based test cannot be anti-conservative relative to the KK matched-pair design.

Usage:

```
InferenceCountKKGLMM$compute_lik_ratio_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null value of β_T to test against; 0 (the default) tests for any treatment effect at all.

Returns: Numeric scalar p-value in $[0, 1]$.

`InferenceCountKKGLMM$compute_asymp_confidence_interval()`: Asymptotic confidence interval, dispatching to `compute_wald_confidence_interval()` or `compute_lik_ratio_confidence_interval()` depending on `self$get_testing_type()` (defaults to Wald if the testing type is neither). See `InferenceAsymp` for the shared testing-type dispatch contract.

Usage:

```
InferenceCountKKGLMM$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

Returns: A length-2 numeric vector `c(lower, upper)` on the log-rate scale.

`InferenceCountKKGLMM$compute_asymp_two_sided_pval()`: Asymptotic two-sided p-value, dispatching to `compute_wald_two_sided_pval()` or `compute_lik_ratio_two_sided_pval()` depending on `self$get_testing_type()` (defaults to Wald if the testing type is neither). See `InferenceAsymp` for the shared testing-type dispatch contract.

Usage:

```
InferenceCountKKGLMM$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null value of β_T to test against; 0 (the default) tests for any treatment effect at all.

Returns: Numeric scalar p-value in $[0, 1]$.

`InferenceCountKKGLMM$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountKKGLMM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A. and Krieger, A. M. (2014). Matching on-the-fly: Sequential allocation with higher power and efficiency. *Biometrics*, 70(2), 378-388. doi:10.1111/biom.12148. (KK14 in `REFERENCES.md`.)

See Also

Analogous Python API for Poisson/count GLMs: statsmodels discrete models. Generalized linear model and Gauss-Hermite quadrature (orientation).

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountKKGLMM$new(seq_des)
inf$compute_estimate()
```

InferenceCountKKHurdlePoissonIVWC

KK Hurdle Poisson IVWC Inference for Count Responses

Description

Inverse-variance weighted combined inference for count responses under a KK matching-on-the-fly design. The matched-pair component is fit with a hurdle-Poisson mixed model using pair random intercepts, and the reservoir component is fit with an ordinary Poisson log-link regression. The reported treatment effect is on the log-rate scale.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceCountKKHurdlePoissonIVWC`

Methods

Public methods:

- `InferenceCountKKHurdlePoissonIVWC$approximate_randomization_distribution_beta_hat_T()`
- `InferenceCountKKHurdlePoissonIVWC$supports_rand_pval_for_incidence()`
- `InferenceCountKKHurdlePoissonIVWC$compute_rand_two_sided_pval()`
- `InferenceCountKKHurdlePoissonIVWC$clone()`

`InferenceCountKKHurdlePoissonIVWC$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceCountKKHurdlePoissonIVWC$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceCountKKHurdlePoissonIVWC$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceCountKKHurdlePoissonIVWC$supports_rand_pval_for_incidence()
```

`InferenceCountKKHurdlePoissonIVWC$compute_rand_two_sided_pval()`:

Usage:

```
InferenceCountKKHurdlePoissonIVWC$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceCountKKHurdlePoissonIVWC$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountKKHurdlePoissonIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`InferenceCountKKHurdlePoissonOneLik`

KK Hurdle-Poisson Combined-Likelihood Inference for Count Responses

---

Description

Fits a two-part hurdle-Poisson model to a KK matching-on-the-fly count design by maximizing a single combined likelihood over the matched pairs and reservoir subjects jointly, rather than fitting matched and reservoir submodels separately and combining them afterward (contrast with the inverse-variance-weighted-combination sibling, `InferenceCountKKHurdlePoissonIVWC`).

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Model. For subject i with count response $Y_i \geq 0$, a hurdle model factors the likelihood into (1) a binary zero-vs-positive part $P(Y_i = 0) = 1 - \pi_i$, $\text{logit}(\pi_i) = X_i'\gamma$, and (2) a zero-truncated Poisson part for the positive counts, $Y_i \mid Y_i > 0 \sim \text{Poisson}_+(\lambda_i)$, $\log \lambda_i = X_i'\beta + \beta_T W_i$, where W_i is the treatment indicator and β_T is the treatment log-rate coefficient for the positive-count submodel (the estimand returned by `compute_estimate()`). Unlike a standard hurdle model fit by maximum likelihood on i.i.d. rows, this class's negative log-likelihood combines the matched-pair rows and reservoir rows of a KK design into one objective (see `private$fit_combined_hurdle()`), so the fitted β_T and its curvature already reflect the design's matched/reservoir structure rather than treating all subjects as exchangeable.

Likelihood tier. `likelihood_tier = "full"`: Wald, score, likelihood-ratio, and gradient testing types are all available (see `get_testing_type()`). Because a hurdle-Poisson combined likelihood does not by itself encode the KK design's finite-sample matched-pair randomization distribution, the score/likelihood-ratio/gradient confidence intervals and p-values are computed twice — once from this class's design-aware asymptotic variance (the same Wald-type calculation used by `compute_wald_confidence_interval()`) and once from the generic likelihood-based calculation inherited from `InferenceAsympLik` — and the wider interval / larger p-value of the two is returned (see `.conservative_kk_onelik_ci()`/`.conservative_kk_onelik_pval()`), so the model-based test is never anti-conservative relative to the design. The Wald confidence interval and p-value fall back to the `BayesianBootstrap` component's bootstrap distribution when the model-based standard error is unavailable or non-finite (e.g. a boundary/separation fit).

Assumptions. Independence across matched pairs and reservoir subjects conditional on covariates; correct specification of the logistic hurdle and log-linear positive-count submodels; a KK matching-on-the-fly design (`DesignSeqOneByOneKK14` or subclass) supplying the matched/reservoir partition. No response censoring is supported (checked at construction via `assertNoCensoring()`).

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceCountKKHurdlePoissonOneLik`

Methods

Public methods:

- `InferenceCountKKHurdlePoissonOneLik$approximate_randomization_distribution_beta_hat_T()`
- `InferenceCountKKHurdlePoissonOneLik$supports_rand_pval_for_incidence()`
- `InferenceCountKKHurdlePoissonOneLik$compute_rand_two_sided_pval()`
- `InferenceCountKKHurdlePoissonOneLik$clone()`

`InferenceCountKKHurdlePoissonOneLik$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceCountKKHurdlePoissonOneLik$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceCountKKHurdlePoissonOneLik$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceCountKKHurdlePoissonOneLik$supports_rand_pval_for_incidence()
```

`InferenceCountKKHurdlePoissonOneLik$compute_rand_two_sided_pval()`:

Usage:

```
InferenceCountKKHurdlePoissonOneLik$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceCountKKHurdlePoissonOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountKKHurdlePoissonOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Mullahy, J. (1986). "Specification and Testing of Some Modified Count Data Models." *Journal of Econometrics*, 33(3), 341-365. doi:10.1016/03044076(86)900023. (Mullahy1986 in `REFERENCES.md`.)

See Also

Analogous Python API for hurdle/zero-truncated count models: statsmodels discrete models (`HurdleCountModel`). Poisson regression (orientation).

`InferenceCountNegBin` *Negative Binomial Regression Inference for Count Responses*

Description

Fits a negative binomial regression for count responses: $Y_i \mid w_i, x_i \sim \mathrm{NegBin}(\mu_i, \theta)$, $\log \mu_i = \beta_0 + \beta_T w_i + x_i^\top \gamma$, $\mathrm{Var}(Y_i) = \mu_i + \mu_i^2/\theta$, jointly maximizing over the regression coefficients and the dispersion parameter θ (`fast_neg_bin_cpp`/`fast_neg_bin_with_var_cpp`). $\hat{\beta}_T$ is a log-rate-ratio: $\exp(\hat{\beta}_T)$ is the estimated rate ratio. Unlike `InferenceCountPoisson`, the negative-binomial model allows overdispersion ($\mathrm{Var}(Y_i) > E[Y_i]$) via θ; smaller θ indicates more overdispersion, and the model converges to Poisson as $\theta \to \infty$. `likelihood_tier = "full"`: Wald, score, gradient, and likelihood-ratio tests are all available, plus parametric-likelihood bootstrap calibration of the likelihood-ratio test (simulating new responses from $\mathrm{NegBin}(\hat{\mu}_i, \hat{\theta})$ under the null). **Jackknife inference is not supported**: delete-one refits of a jointly-estimated dispersion parameter

are numerically unstable, so `compute_jackknife_estimate()` and related methods report explicit non-estimability rather than attempting delete-one refits. Validity requires the negative-binomial mean-variance relationship to hold and the usual correctly-specified-linear-predictor-on-the-log-scale assumption.

Super class

`Inference` -> `InferenceCountNegBin`

Methods

Public methods:

- `InferenceCountNegBin$new()`
- `InferenceCountNegBin$compute_estimate_with_bootstrap_weights()`
- `InferenceCountNegBin$compute_jackknife_estimate()`
- `InferenceCountNegBin$compute_jackknife_bias_estimate()`
- `InferenceCountNegBin$compute_jackknife_std_error()`
- `InferenceCountNegBin$compute_jackknife_wald_two_sided_pval()`
- `InferenceCountNegBin$compute_jackknife_wald_confidence_interval()`
- `InferenceCountNegBin$clone()`

`InferenceCountNegBin$new()`: Initialize inference for the negative binomial regression model $Y_i \mid w_i, x_i \sim \mathrm{NegBin}(\mu_i, \theta)$, $\log \mu_i = \beta_0 + \beta_T w_i + x_i^\top \gamma$; see `InferenceCountNegBin` for the model form. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferenceCountNegBin$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL,
  optimization_alg = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a count response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart optimizer start values by default.

`optimization_alg` Optimization algorithm to use. Default is dispatched via policy.

`InferenceCountNegBin$compute_estimate_with_bootstrap_weights()`: Refits the negative binomial model with subject/block-level weights applied to the fitting log-likelihood (Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded to row level via `private$expand_subject_or_block_weig` via `fast_neg_bin_weighted_cpp`, and returns the reweighted log-rate-ratio estimate $\hat{\beta}_T^{(w)}$. If

the weighted negative-binomial fit fails to converge, falls back to a weighted Poisson GLM (`stats::glm(family = poisson())`) as an estimating-equation-consistent point estimate of the same mean structure (this fallback does not itself estimate θ, so no standard error is computed in that path); no standard error is computed in either path (`s_beta_hat_T` is always `NA`).

Usage:

```
InferenceCountNegBin$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferenceCountNegBin$compute_jackknife_estimate()`: Negative-binomial delete-one refits are unstable for jackknife inference; report explicit non-estimability.

Usage:

```
InferenceCountNegBin$compute_jackknife_estimate(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.

`InferenceCountNegBin$compute_jackknife_bias_estimate()`: Report that the jackknife bias estimate is unavailable for negative-binomial fits when delete-one refits are not stable; see `InferenceJackknife` for the shared jackknife contract.

Usage:

```
InferenceCountNegBin$compute_jackknife_bias_estimate(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.

`InferenceCountNegBin$compute_jackknife_std_error()`: Report that the jackknife standard error is unavailable for negative-binomial fits when delete-one refits are not stable; see `InferenceJackknife` for the shared jackknife contract.

Usage:

```
InferenceCountNegBin$compute_jackknife_std_error(unit = "auto")
```

Arguments:

`unit` Deletion unit. Default `"auto"`.

`InferenceCountNegBin$compute_jackknife_wald_two_sided_pval()`: Reports that jackknife-Wald p-values are unavailable here; see `InferenceJackknife`.

Usage:

```
InferenceCountNegBin$compute_jackknife_wald_two_sided_pval(
  delta = 0,
  unit = "auto"
)
```

Arguments:

`delta` Null treatment-effect value. Default 0.

`unit` Deletion unit. Default `"auto"`.

`InferenceCountNegBin$compute_jackknife_wald_confidence_interval()`: Reports that jackknife-Wald intervals are unavailable here; see `InferenceJackknife`.

Usage:

```
InferenceCountNegBin$compute_jackknife_wald_confidence_interval(
  alpha = 0.05,
  unit = "auto"
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`unit` Deletion unit. Default `"auto"`.

`InferenceCountNegBin$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountNegBin$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Cameron, A. C., and Trivedi, P. K. (2013). *Regression Analysis of Count Data* (2nd ed.). Cambridge University Press, for the negative binomial regression model and its maximum-likelihood theory.

See Also

Comparable Python API: statsmodels discrete models (`NegativeBinomial`). See also: Negative binomial distribution (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountNegBin$new(seq_des)
inf$compute_estimate()
```

`InferenceCountPoisson` *Poisson Regression Inference for Count Responses*

Description

Fits a Poisson log-link regression for count responses: $Y_i \mid w_i, x_i \sim \text{Poisson}(\mu_i)$, $\log \mu_i = \beta_0 + \beta_T w_i + x_i^\top \gamma$ (`fast_poisson_regression_cpp`/ `fast_poisson_regression_with_var_cpp`). $\hat{\beta}_T$ is a log-rate-ratio: $\exp(\hat{\beta}_T)$ is the estimated rate ratio. `likelihood_tier = "full"`: Wald, score, gradient, and likelihood-ratio tests are all available, plus parametric-likelihood bootstrap calibration of the likelihood-ratio test (simulating new Poisson responses under the null).

Design-conservative testing. Every asymptotic/likelihood test method on this class does not report the raw model-based result directly. Instead, it also computes a design-based jackknife-Wald test (`compute_jackknife_wald_two_sided_pval()`/ `compute_jackknife_wald_confidence_interval()`, which do not assume the Poisson mean-variance relationship) and combines the two conservatively: p-values report $\max$ of the model-based and design-based p-values, and confidence intervals report the *union* of the model-based and design-based intervals. This guards against the model-based test being anti-conservative when the Poisson equidispersion assumption ($\text{Var}(Y_i) = E[Y_i]$) fails — a real risk for count data, which is frequently overdispersed (see `InferenceCountNegBin` for a model that estimates dispersion directly instead). If either component is unavailable, the available one is used alone; if neither is available, the result is `NA`. Validity requires the usual correctly-specified linear predictor on the log scale; unlike the raw Poisson likelihood alone, this class's actual reported inference degrades gracefully (rather than becoming anti-conservative) under mean-variance misspecification. This combination applies to the asymptotic likelihood methods, their Bartlett-corrected variants, and likelihood-ratio bootstrap calibration. Generic estimate-resampling intervals (Bayesian bootstrap, ordinary bootstrap, and the direct parametric-bootstrap estimate interval) retain their standard component definitions and are not unioned with a second jackknife interval; use one of the design-conservative likelihood methods when robustness to Poisson equidispersion failure is required.

Estimand. Composes `MarginalEstimand` (`set_estimand()`/`get_estimand()`/`get_supported_estimands()`). Under the default `estimand = "conditional"`, $\hat{\beta}_T$ is the log-rate-ratio above. Under `estimand = "marginal_mean_diff"`, the reported quantity is instead the g-computation marginal rate difference $\frac{1}{n}\sum_i\{\exp(\hat{\beta}_0 + \hat{\beta}_T + X_i^\top \hat{\gamma}) - \exp(\hat{\beta}_0 + X_i^\top \hat{\gamma})\}$. Under `estimand = "marginal_ratio"`, the log of the corresponding marginal rate ratio — **which is numerically identical to the conditional $\hat{\beta}_T$ for this family**: because the log link is linear in w_i with no treatment-by-covariate interaction term, every subject's treated-vs-control mean ratio is $\exp(\hat{\beta}_0 + \hat{\beta}_T + X_i^\top \hat{\gamma}) / \exp(\hat{\beta}_0 + X_i^\top \hat{\gamma}) = \exp(\hat{\beta}_T)$ exactly, so averaging over subjects before or after taking the ratio makes no difference. `"marginal_ratio"` is provided for estimand-API consistency with the other model families, not because it differs numerically from `"conditional"` here; `"marginal_mean_diff"` is the estimand where g-computation actually changes the reported number for a Poisson GLM, since a difference (unlike a ratio) does not collapse under a nonlinear (log) mean function. Because there is no latent submodel for this family (unlike e.g. `InferenceCountZeroInflatedPoisson`'s excess-zero mixture), the marginal mean function is exactly the model's own fitted mean; no separate standardization step beyond the g-computation average is needed. Standard errors under a marginal estimand use the delta method against the model's coefficient covariance (degrees of freedom `Inf`), including in the design-conservative union/max combination above (the design-based jackknife-Wald component also refits under the active estimand); `testing_type` is restricted to `"wald"` whenever the

estimand is non-conditional. The underlying model fit is identical regardless of estimand — switching `estimand` is a pure post-fit transform, never a refit.

Super class

`Inference` -> `InferenceCountPoisson`

Methods

Public methods:

- `InferenceCountPoisson$new()`
- `InferenceCountPoisson$compute_estimate()`
- `InferenceCountPoisson$compute_asymp_confidence_interval()`
- `InferenceCountPoisson$compute_asymp_two_sided_pval()`
- `InferenceCountPoisson$compute_wald_confidence_interval()`
- `InferenceCountPoisson$compute_wald_two_sided_pval()`
- `InferenceCountPoisson$compute_score_confidence_interval()`
- `InferenceCountPoisson$compute_score_two_sided_pval()`
- `InferenceCountPoisson$compute_lik_ratio_confidence_interval()`
- `InferenceCountPoisson$compute_lik_ratio_two_sided_pval()`
- `InferenceCountPoisson$compute_lik_ratio_bartlett_approx_confidence_interval()`
- `InferenceCountPoisson$compute_lik_ratio_bartlett_approx_two_sided_pval()`
- `InferenceCountPoisson$compute_lik_ratio_bartlett_confidence_interval()`
- `InferenceCountPoisson$compute_lik_ratio_bartlett_two_sided_pval()`
- `InferenceCountPoisson$compute_gradient_confidence_interval()`
- `InferenceCountPoisson$compute_gradient_two_sided_pval()`
- `InferenceCountPoisson$compute_lik_ratio_bootstrap_two_sided_pval()`
- `InferenceCountPoisson$compute_lik_ratio_bootstrap_confidence_interval()`
- `InferenceCountPoisson$compute_estimate_with_bootstrap_weights()`
- `InferenceCountPoisson$clone()`

`InferenceCountPoisson$new()`: Initialize inference for the Poisson regression model $Y_i \mid w_i, x_i \sim \mathrm{Poisson}(\mu_i)$, $\log \mu_i = \beta_0 + \beta_T w_i + x_i^\top \gamma$; see `InferenceCountPoisson` for the model form and the design-conservative testing mechanism. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferenceCountPoisson$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL,
  harden = TRUE
)
```

Arguments:

`des_obj` A completed `Design` object with a count response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values by default.

`harden` Whether to apply robustness measures.

`InferenceCountPoisson$compute_estimate()`: Fits the Poisson regression model by maximum likelihood. Under the default `estimand = "conditional"`, returns $\hat{\beta}_T$, the treatment log-rate-ratio. Under `estimand = "marginal_mean_diff"/"marginal_ratio"` (set via `set_estimand()`), returns the g-computation marginal rate difference/log-rate-ratio instead — see the class-level `@details` for the formula. The underlying model fit is identical either way (a pure post-fit transform of the same cached fit, no refit).

Usage:

```
InferenceCountPoisson$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip standard-error computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferenceCountPoisson$compute_asymp_confidence_interval()`: Design-conservative confidence interval for β_T using whichever test type is configured (`private$testing_type`: `"wald"`, `"score"`, `"gradient"`, or `"lik_ratio"`); see `InferenceCountPoisson` for the union-with-jackknife-Wald combination rule.

Usage:

```
InferenceCountPoisson$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level. Default 0.05.

`InferenceCountPoisson$compute_asymp_two_sided_pval()`: Design-conservative two-sided p-value for $H_0 : \beta_T =$ `delta` using whichever test type is configured (`private$testing_type`); see `InferenceCountPoisson` for the max-with-jackknife-Wald combination rule.

Usage:

```
InferenceCountPoisson$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect. Default 0.

`InferenceCountPoisson$compute_wald_confidence_interval()`: Wald confidence interval for β_T using the fitted Poisson model's Fisher-information-based standard error, unioned with the design-based jackknife-Wald interval; see `InferenceCountPoisson` for the combination rule and `InferenceAsymp` for the underlying Wald contract.

Usage:

```
InferenceCountPoisson$compute_wald_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level. Default 0.05.

`InferenceCountPoisson$compute_wald_two_sided_pval()`: Wald test of $H_0 : \beta_T =$ `delta` using the fitted Poisson model's Fisher-information-based standard error, taking the max with the design-based jackknife-Wald p-value; see `InferenceCountPoisson` for the combination rule.

Usage:

```
InferenceCountPoisson$compute_wald_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect. Default 0.

`InferenceCountPoisson$compute_score_confidence_interval()`: Score-test confidence interval for β_T (inverting the Poisson score test at each candidate null, no full re-fit needed at the observed information), unioned with the design-based jackknife-Wald interval; see `InferenceCountPoisson` for the combination rule.

Usage:

```
InferenceCountPoisson$compute_score_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level. Default 0.05.

`InferenceCountPoisson$compute_score_two_sided_pval()`: Score test of $H_0 : \beta_T =$ `delta`, taking the max with the design-based jackknife-Wald p-value; see `InferenceCountPoisson` for the combination rule.

Usage:

```
InferenceCountPoisson$compute_score_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect. Default 0.

`InferenceCountPoisson$compute_lik_ratio_confidence_interval()`: Likelihood-ratio-test confidence interval for β_T (test inversion, requiring a null refit at each candidate value), unioned with the design-based jackknife-Wald interval; see `InferenceCountPoisson` for the combination rule.

Usage:

```
InferenceCountPoisson$compute_lik_ratio_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level. Default 0.05.

`InferenceCountPoisson$compute_lik_ratio_two_sided_pval()`: Likelihood-ratio test of $H_0 : \beta_T =$ `delta`, taking the max with the design-based jackknife-Wald p-value; see `InferenceCountPoisson` for the combination rule.

Usage:

```
InferenceCountPoisson$compute_lik_ratio_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect. Default 0.

`InferenceCountPoisson$compute_lik_ratio_bartlett_approx_confidence_interval()`: Approximate Bartlett-corrected likelihood-ratio confidence interval, unioned with the design-based jackknife-Wald interval.

Usage:

```
InferenceCountPoisson$compute_lik_ratio_bartlett_approx_confidence_interval(
  alpha = 0.05,
  B = 99
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`B` Number of Monte-Carlo replicates used for the Bartlett factor.

`InferenceCountPoisson$compute_lik_ratio_bartlett_approx_two_sided_pval()`: Approximate Bartlett-corrected likelihood-ratio p-value, conservatively combined with the design-based jackknife-Wald p-value.

Usage:

```
InferenceCountPoisson$compute_lik_ratio_bartlett_approx_two_sided_pval(
  delta = 0,
  B = 99
)
```

Arguments:

`delta` Null treatment effect. Default 0.

`B` Number of Monte-Carlo replicates used for the Bartlett factor.

`InferenceCountPoisson$compute_lik_ratio_bartlett_confidence_interval()`: Best-available Bartlett-corrected likelihood-ratio confidence interval, unioned with the design-based jackknife-Wald interval.

Usage:

```
InferenceCountPoisson$compute_lik_ratio_bartlett_confidence_interval(
  alpha = 0.05,
  B = 99
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`B` Number of Monte-Carlo replicates when the approximate correction is used.

`InferenceCountPoisson$compute_lik_ratio_bartlett_two_sided_pval()`: Best-available Bartlett-corrected likelihood-ratio p-value, conservatively combined with the design-based jackknife-Wald p-value.

Usage:

```
InferenceCountPoisson$compute_lik_ratio_bartlett_two_sided_pval(
  delta = 0,
  B = 99
)
```

Arguments:

`delta` Null treatment effect. Default 0.

`B` Number of Monte-Carlo replicates when the approximate correction is used.

`InferenceCountPoisson$compute_gradient_confidence_interval()`: Gradient-test confidence interval for β_T (a score-test variant using the observed rather than expected information), unioned with the design-based jackknife-Wald interval; see `InferenceCountPoisson` for the combination rule.

Usage:

```
InferenceCountPoisson$compute_gradient_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level. Default 0.05.

`InferenceCountPoisson$compute_gradient_two_sided_pval()`: Gradient test of $H_0 : \beta_T =$ `delta`, taking the max with the design-based jackknife-Wald p-value; see `InferenceCountPoisson` for the combination rule.

Usage:

```
InferenceCountPoisson$compute_gradient_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect. Default 0.

`InferenceCountPoisson$compute_lik_ratio_bootstrap_two_sided_pval()`: Parametric-likelihood-bootstrap-calibrated likelihood-ratio test of $H_0 : \beta_T =$ `delta` (simulating new Poisson responses from the null-constrained fit to calibrate the LR statistic's null distribution), taking the max with the design-based jackknife-Wald p-value; see `InferenceCountPoisson` for the combination rule.

Usage:

```
InferenceCountPoisson$compute_lik_ratio_bootstrap_two_sided_pval(
  delta = 0,
  B = 199,
  show_progress = FALSE,
  min_number_usable_samples = 5L,
  max_attempts_per_replicate = 2L
)
```

Arguments:

`delta` Null treatment effect. Default 0.

`B` Number of bootstrap replicates.

`show_progress` Whether to show progress.

`min_number_usable_samples` Minimum usable bootstrap samples.

`max_attempts_per_replicate` Maximum attempts per replicate.

`InferenceCountPoisson$compute_lik_ratio_bootstrap_confidence_interval()`: Parametric-likelihood-bootstrap-calibrated likelihood-ratio confidence interval for β_T (test inversion using the bootstrap-calibrated null distribution), unioned with the design-based jackknife-Wald interval; see `InferenceCountPoisson` for the combination rule.

Usage:

```
InferenceCountPoisson$compute_lik_ratio_bootstrap_confidence_interval(
  alpha = 0.05,
```

```
  B = 199,
  show_progress = FALSE,
  min_number_usable_samples = 5L,
  max_attempts_per_replicate = 2L,
  root_tolerance = NULL,
  max_root_iterations = 8L
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`B` Number of bootstrap replicates.

`show_progress` Whether to show progress.

`min_number_usable_samples` Minimum usable bootstrap samples.

`max_attempts_per_replicate` Maximum attempts per replicate.

`root_tolerance` Root tolerance.

`max_root_iterations` Maximum root iterations.

`InferenceCountPoisson$compute_estimate_with_bootstrap_weights()`: Refits the Poisson model with subject/block-level weights applied to the fitting log-likelihood (Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded to row level via `private$expand_subject_or_block_weights_to_r` via `fast_poisson_regression_weighted_cpp`, and returns the reweighted log-rate-ratio estimate $\hat{\beta}_T^{(w)}$. Uses the same QR column-dropping hardening as the unweighted fit; a hardened fit with a non-finite treatment coefficient is cached as nonestimable and returns `NA`. The weighted refit always targets the conditional treatment coefficient, whatever the active estimand (it is not estimand-aware).

Usage:

```
InferenceCountPoisson$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Row weights for the bootstrap sample.

`estimate_only` If TRUE, skip variance calculations.

`InferenceCountPoisson$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountPoisson$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Cameron, A. C., and Trivedi, P. K. (2013). *Regression Analysis of Count Data* (2nd ed.). Cambridge University Press, for the Poisson regression model and its maximum-likelihood theory.

See Also

Comparable Python API: statsmodels discrete models (`Poisson`). See also: Poisson regression (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountPoisson$new(seq_des)
inf$compute_estimate()


inf$set_seed(1)
inf$compute_lik_ratio_bootstrap_two_sided_pval(delta = 0, B = 9, show_progress = FALSE)
```

InferenceCountPoissonKKGEE

GEE Inference for KK Designs with Count Response

Description

Fits a Generalized Estimating Equations (GEE) model with a Poisson family and log link, $\log E[Y_i \mid x_i] = x_i^\top \beta$, for count responses under a KK matching-on-the-fly design, using an **exchangeable** working correlation structure where each cluster is either a matched pair (2 members) or a reservoir singleton (1 member) — see `$compute_estimate()`'s method-level documentation for the full fitting contract (internal Rcpp solver vs. **geepack** fallback, hardening/retry behavior). GEE is used here purely to fit one marginal model jointly across matched-pair and reservoir subjects while accounting for the within-pair correlation the matching induces, not as a longitudinal/repeated-measures tool. Inference is quasi-likelihood/estimating-equation based (`likelihood_tier = "quasi"`): standard errors are GEE sandwich (robust) standard errors, not model-likelihood-based.

Super class

`Inference` -> `InferenceCountPoissonKKGEE`

Methods

Public methods:

- `InferenceCountPoissonKKGEE$new()`
- `InferenceCountPoissonKKGEE$clone()`

`InferenceCountPoissonKKGEE$new()`: Initialize KK count-response GEE inference, validate the matched/reservoir design, and prepare the exchangeable-working-correlation Poisson (log-link) GEE fitting machinery used by `InferenceCountPoissonKKGEE`.

Usage:

```
InferenceCountPoissonKKGEE$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a count response.

`model_formula` Optional formula for covariate adjustment.

`use_rcpp` Whether to use the internal Rcpp GEE solver (`TRUE`, default) with automatic fallback to `geepack::geeglm` on failure, or always use `geepack::geeglm` directly (`FALSE`).

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`InferenceCountPoissonKKGEE$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountPoissonKKGEE$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Liang, K.-Y., and Zeger, S. L. (1986). "Longitudinal Data Analysis Using Generalized Linear Models." *Biometrika*, 73(1), 13-22, doi:10.1093/biomet/73.1.13, for the GEE estimating-equation framework and sandwich variance estimator used here.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountPoissonKKGEE$new(seq_des)
inf$compute_estimate()
```

InferenceCountQuasiPoisson

Quasi-Poisson Regression Inference for Count Responses

Description

Fits a Poisson log-link mean model, $\log E[Y_i \mid x_i] = x_i^\top \beta$, for count responses using the treatment indicator and, optionally, all recorded covariates as predictors, via `fast_quasipoisson_regression_with_var_cpp` — see that page for the full model and the Pearson-dispersion-scaled ("quasi-Poisson") variance formula, $\widehat{\mathrm{Var}}(\hat{\beta}_k) = \hat{\phi}\,[(X^\top \hat{W} X)^{-1}]_{kk}$, which corrects standard errors for overdispersion ($\mathrm{Var}(Y_i) > E[Y_i]$) relative to the strict Poisson assumption without changing the point estimate $\hat{\beta}$. This class has no likelihood-ratio/score/gradient testing capability (`likelihood_tier = "quasi"`): the dispersion-scaled quasi-likelihood is not a normalized model likelihood, so only Wald inference is available. Rank-deficient covariate columns are dropped automatically before fitting (via `private$fit_with_hardened_qr_column_d`

Estimand. Composes `MarginalEstimand` (`set_estimand()`/`get_estimand()`/`get_supported_estimands()`). Under the default `estimand = "conditional"`, $\hat{\beta}_T$ is the treatment log-rate-ratio. Under `"marginal_mean_diff"` it is the g-computed difference in the average fitted count under treatment vs. control, $\frac{1}{n}\sum_i \{\exp(x_{i1}^\top \hat{\beta}) - \exp(x_{i0}^\top \hat{\beta})\}$, with every subject plugged in at treatment 1 and 0. `"marginal_ratio"` is the log of the corresponding ratio, which for this log-link family equals the conditional $\hat{\beta}_T$ exactly (there is no treatment-by-covariate term), so it is offered for estimand-API consistency; its delta-method SE equals the conditional SE. Under a marginal estimand the standard error is the delta-method SE against the dispersion-scaled coefficient covariance $\hat{\phi}(X^\top \hat{W} X)^{-1}$, i.e. the plain-Poisson marginal SE inflated by $\sqrt{\hat{\phi}}$, with a normal reference (degrees of freedom `Inf`). Switching the estimand is a pure post-fit transform of the cached fit, never a refit. The Bayesian-bootstrap weighted refit is not estimand-aware: it always targets the conditional coefficient.

Super class

`Inference` -> `InferenceCountQuasiPoisson`

Methods

Public methods:

- `InferenceCountQuasiPoisson$new()`
- `InferenceCountQuasiPoisson$compute_estimate()`
- `InferenceCountQuasiPoisson$compute_estimate_with_bootstrap_weights()`
- `InferenceCountQuasiPoisson$compute_asymp_confidence_interval()`
- `InferenceCountQuasiPoisson$compute_asymp_two_sided_pval()`
- `InferenceCountQuasiPoisson$clone()`

`InferenceCountQuasiPoisson$new()`: Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize a quasi-Poisson regression inference object for a completed design with a count, uncensored response.

Usage:

```
InferenceCountQuasiPoisson$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL,
  harden = TRUE
)
```

Arguments:

`des_obj` A completed `Design` object with a count response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`harden` Whether to apply robustness measures.

`InferenceCountQuasiPoisson$compute_estimate()`: Computes the quasi-Poisson point estimate via `fast_quasipoisson_regression_with_var_cpp` (see class documentation for the full model). Under the default `estimand = "conditional"` this is the treatment coefficient $\hat{\beta}_T$; under `"marginal_mean_diff"` or `"marginal_ratio"` (set via `set_estimand()`) it is the g-computed marginal mean difference / log ratio, a pure post-fit transform of the same cached fit (no refit). Rank-deficient covariate columns are dropped before fitting.

Usage:

```
InferenceCountQuasiPoisson$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance calculations.

`InferenceCountQuasiPoisson$compute_estimate_with_bootstrap_weights()`: Recomputes the Poisson-mean-model treatment estimate under subject/block bootstrap weights (via `fast_poisson_regression_weighted_cpp`), used by the Bayesian bootstrap and related weighted-resampling machinery; see `InferenceBayesianBootstrap`. When `estimate_only = FALSE`, also computes a weighted Pearson-dispersion-scaled standard error (fixed 2026-09-07 – previously always `NA` regardless of `estimate_only`, which starved the Bayesian-bootstrap studentized/BCa variants of a per-replicate SE and left them NA on the large majority of calls). The weighted refit always targets the conditional treatment coefficient, whatever the active estimand (it is not estimand-aware).

Usage:

```
InferenceCountQuasiPoisson$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip the dispersion-correction computation.

`InferenceCountQuasiPoisson$compute_asymp_confidence_interval()`: Computes a $1-\alpha$ level Wald confidence interval for the active estimand. Under the default conditional estimand this is the quasi-Poisson treatment coefficient $\hat{\beta}_T$ with the Pearson-dispersion-scaled standard error from `fast_quasipoisson_regression_with_var_cpp` (see class documentation); under a marginal estimand it is the g-computed functional with its delta-method standard error. Both use a normal reference. See `InferenceAsymp` for the shared asymptotic confidence-interval contract this delegates to.

Usage:

```
InferenceCountQuasiPoisson$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Confidence level.

`InferenceCountQuasiPoisson$compute_asymp_two_sided_pval()`: Computes a two-sided Wald p-value testing $H_0 : \beta_T =$ `delta` under the conditional estimand, or that the active marginal functional equals `delta` under a marginal estimand, from the same standard error used by `$compute_asymp_confidence_interval()` (normal reference). See `InferenceAsymp` for the shared asymptotic two-sided p-value contract this delegates to.

Usage:

```
InferenceCountQuasiPoisson$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect value.

`InferenceCountQuasiPoisson$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountQuasiPoisson$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountQuasiPoisson$new(seq_des)
inf$compute_estimate()
```

InferenceCountRobustPoisson

Robust (Sandwich-Variance) Poisson Regression Inference for Count Responses

Description

Fits the same Poisson log-link mean model as `InferenceCountPoisson` (point estimate via `fast_poisson_regression_cpp` maximum likelihood), but computes standard errors via a **Huber-White (Eicker-Huber-White) sandwich** estimator instead of the model-based Poisson Fisher information or the quasi-Poisson dispersion scaling used by `InferenceCountQuasiPoisson`: $\widehat{\mathrm{Var}}(\hat{\beta}) = B\,M\,B$, with "bread" $B = (X^\top \hat{W} X)^{-1}$ (the Poisson Fisher information at $\hat{\beta}$) and "meat" $M = X^\top \mathrm{diag}((y_i - \hat{\mu}_i)^2) X$ (the empirical score outer product), via `robust_sandwich_variance_from_xtwx()`. This is robust to arbitrary mean-variance misspecification (not just proportional overdispersion), at the cost of somewhat higher variance in the SE estimate itself for small samples. This class has no likelihood-ratio/score/gradient testing capability (`likelihood_tier = "quasi"`): only Wald inference is available. Rank-deficient covariate columns are dropped automatically before fitting.

Super class

`Inference` -> `InferenceCountRobustPoisson`

Methods

Public methods:

- `InferenceCountRobustPoisson$new()`
- `InferenceCountRobustPoisson$compute_estimate()`
- `InferenceCountRobustPoisson$compute_estimate_with_bootstrap_weights()`
- `InferenceCountRobustPoisson$compute_asymp_confidence_interval()`
- `InferenceCountRobustPoisson$compute_asymp_two_sided_pval()`
- `InferenceCountRobustPoisson$clone()`

`InferenceCountRobustPoisson$new()`: Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize a robust (sandwich-variance) Poisson regression inference object for a completed design with a count, uncensored response.

Usage:

```
InferenceCountRobustPoisson$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a count response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart starting values for the optimizer.

`InferenceCountRobustPoisson$compute_estimate()`: Computes the Poisson treatment coefficient $\hat{\beta}_T$ via `fast_poisson_regression_cpp` (see class documentation for the sandwich-variance model). Rank-deficient covariate columns are dropped before fitting.

Usage:

```
InferenceCountRobustPoisson$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance calculations.

`InferenceCountRobustPoisson$compute_estimate_with_bootstrap_weights()`: Recomputes the Poisson-mean-model treatment estimate under subject/block bootstrap weights (via `fast_poisson_regression_weighted_cpp`), used by the Bayesian bootstrap and related weighted-resampling machinery; see `InferenceBayesianBootstrap`. When `estimate_only = FALSE`, also computes a weighted Huber-White sandwich standard error (fixed 2026-09-07 – previously always `NA` regardless of `estimate_only`, which starved the Bayesian-bootstrap studentized/BCa variants of a per-replicate SE and left them NA on the large majority of calls).

Usage:

```
InferenceCountRobustPoisson$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip the sandwich-variance computation.

`InferenceCountRobustPoisson$compute_asymp_confidence_interval()`: Computes a $1-\alpha$ level confidence interval for the robust Poisson treatment coefficient $\hat{\beta}_T$, using the Huber-White sandwich standard error (see class documentation). See `InferenceAsymp` for the shared asymptotic confidence-interval contract this delegates to.

Usage:

```
InferenceCountRobustPoisson$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Confidence level.

`InferenceCountRobustPoisson$compute_asymp_two_sided_pval()`: Computes a two-sided Wald p-value testing $H_0 : \beta_T = \texttt{delta}$, from the same Huber-White sandwich standard error used by `$compute_asymp_confidence_interval()`. See `InferenceAsymp` for the shared asymptotic two-sided p-value contract this delegates to.

Usage:

```
InferenceCountRobustPoisson$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect value.

`InferenceCountRobustPoisson$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountRobustPoisson$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountRobustPoisson$new(seq_des)
inf$compute_estimate()
```

InferenceCountZeroInflatedNegBin

Zero-Inflated Negative Binomial Regression Inference for Count Responses

---

Description

Fits a zero-inflated negative binomial regression for count responses: a binary excess-zero submodel $P(\text{structural zero}_i) = \text{logit}^{-1}(X_i^{h\top}\gamma^h)$ mixed with a (non-truncated) negative-binomial count submodel $\log E[Y_i \mid \text{not structural zero}, w_i, x_i] = \beta_0 + \beta_T w_i + x_i^\top \gamma$, $\text{Var}(Y_i \mid \text{not structural zero}) = \mu_i + \mu_i^2/\theta$. Unlike a hurdle model, zero counts can arise from either the structural-zero mechanism or from an ordinary negative-binomial draw of 0. The hurdle and count submodels may use different covariate formulas (`model_formula`/`model_formula_zero`). The reported treatment effect is the coefficient from the conditional count component, on the log-rate scale, **conditional on the response coming from the count process, not the excess-zero-inflation mechanism**: it is not the effect on the unconditional mean $E[Y]$, which also depends on how treatment shifts the excess-zero probability. A marginal (unconditional-mean) estimand is not yet implemented for this class (see `marginal_estimand_report.md`). `likelihood_tier = "full"`: Wald, gradient, score, and (bootstrap-calibrated) likelihood-ratio tests are all available for the count submodel's treatment coefficient (unlike the Poisson variant, this class's private `get_supported_testing_types_impl()` includes `"score"`). **Jackknife inference is not supported**: delete-one refits of this two-part mixture model with a jointly-estimated dispersion parameter are numerically unstable, so `compute_jackknife_estimate()` and related methods report explicit non-estimability rather than attempting delete-one refits.

Super classes

`Inference` -> `InferenceCountZeroAugmentedPoissonAbstract` -> `InferenceCountZeroInflatedNegBin`

Methods

Public methods:

- InferenceCountZeroInflatedNegBin$new()
- InferenceCountZeroInflatedNegBin$compute_estimate()
- InferenceCountZeroInflatedNegBin$clone()

`InferenceCountZeroInflatedNegBin$new()`: Initialize inference for the zero-inflated negative binomial model (binary excess-zero submodel mixed with a negative-binomial count submodel); see `InferenceCountZeroInflatedNegBin` for the model form. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferenceCountZeroInflatedNegBin$new(
  des_obj,
  model_formula = NULL,
  model_formula_zero = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  optimization_alg = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a count response.

`model_formula` Optional formula for covariate adjustment.

`model_formula_zero` Formula for the zero-inflation submodel. If `NULL` (default), it uses the same formula as `model_formula`.

`use_rcpp` Logical. If `TRUE` (default), use our internal Rcpp implementation. If `FALSE`, use **glmmTMB**.

`verbose` Whether to print progress messages.

`optimization_alg` Optimization algorithm. Default is dispatched via policy.

`InferenceCountZeroInflatedNegBin$compute_estimate()`: Fits the zero-inflated negative binomial model. Under the default `estimand = "conditional"`, returns $\hat{\beta}_T$, the treatment log-rate coefficient from the conditional count submodel. Under `estimand = "marginal_mean_diff"` or `"marginal_ratio"` (set via `set_estimand()`), returns the g-computation marginal mean difference or log-scale marginal ratio of the unconditional mean instead – a post-fit transform of the same cached fit, no refit.

Usage:

```
InferenceCountZeroInflatedNegBin$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip standard-error computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferenceCountZeroInflatedNegBin$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountZeroInflatedNegBin$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Lambert, D. (1992). "Zero-Inflated Poisson Regression, with an Application to Defects in Manufacturing." *Technometrics*, 34(1), 1-14, doi:10.2307/1269547, for the zero-inflated count-model framework.

See Also

`InferenceCountNegBin` for the single-part negative binomial model this class's count submodel generalizes; `InferenceCountZeroInflatedPoisson` for the Poisson (equidispersed) variant.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountZeroInflatedNegBin$new(seq_des, model_formula = ~ x1)
inf$compute_estimate()
```

InferenceCountZeroInflatedPoisson

Zero-Inflated Poisson Regression Inference for Count Responses

---

Description

Fits a zero-inflated Poisson regression for count responses: a binary excess-zero submodel $P(\text{structural zero}_i) = \text{logit}^{-1}(X_i^{h\top}\gamma^h)$ mixed with a (non-truncated) Poisson count submodel $\log E[Y_i \mid \text{not structural zero}, w_i, x_i] = \beta_0 + \beta_T w_i + x_i^\top \gamma$. Unlike a hurdle model, zero counts can arise from either the structural-zero mechanism or from an ordinary Poisson draw of 0, so the two mixture components are not identified by disjoint support. The hurdle and count submodels may use different covariate formulas (`model_formula`/`model_formula_zero`). The reported treatment effect is the coefficient from the conditional count component, on the log-rate scale, **conditional on the response coming from the count process, not the excess-zero-inflation mechanism**: it is not the effect on the unconditional mean $E[Y]$, which also depends on how treatment shifts the excess-zero probability, under the default `estimand = "conditional"`. `likelihood_tier = "full"`: Wald, gradient, and (bootstrap-calibrated) likelihood-ratio tests are available for the count submodel's treatment coefficient under that estimand; a plain score test is not exposed. **Jackknife inference is not supported**: delete-one refits of this two-part mixture model are numerically unstable, so `compute_jackknife_estimate()` and related methods report explicit non-estimability rather than attempting delete-one refits.

Marginal (unconditional-mean) estimand. Via `set_estimand()`, this class also supports `estimand = "marginal_mean_diff"` and `"marginal_ratio"`: the g-computation average, over the empirical covariate distribution, of the model-implied unconditional mean $E[Y_i \mid w_i, x_i] = (1 - \pi(x_i))\lambda(x_i)$ (the untruncated Poisson mean weighted by the non-structural-zero probability) at $w_i = 1$ vs. $w_i = 0$ — a mean difference or, on the log scale, a mean ratio. This is a pure post-fit transform of the

same maximum- likelihood fit (no refit), with a delta-method standard error computed against the sandwich-robust covariance matrix already used for this class's conditional Wald inference. Only "wald"-type inference is available under a marginal estimand (no likelihood-ratio/score/gradient test, since the marginal quantity is a functional of the fitted parameters, not itself a likelihood).

Super classes

Inference -> InferenceCountZeroAugmentedPoissonAbstract -> InferenceCountZeroInflatedPoisson

Methods

Public methods:

- InferenceCountZeroInflatedPoisson$new()
- InferenceCountZeroInflatedPoisson$compute_estimate()
- InferenceCountZeroInflatedPoisson$compute_asymp_confidence_interval()
- InferenceCountZeroInflatedPoisson$compute_asymp_two_sided_pval()
- InferenceCountZeroInflatedPoisson$clone()

InferenceCountZeroInflatedPoisson$new(): Initialize inference for the zero-inflated Poisson model (binary excess-zero submodel mixed with a Poisson count submodel); see InferenceCountZeroInflatedPoiss for the model form. Does not fit the model; the fit is deferred to the first call to compute_estimate() or a method that requires it.

Usage:

```
InferenceCountZeroInflatedPoisson$new(
  des_obj,
  model_formula = NULL,
  model_formula_zero = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  optimization_alg = NULL
)
```

Arguments:

des_obj A completed Design object with a count response.

model_formula Optional formula for the count submodel.

model_formula_zero Formula for the zero-inflation submodel. If NULL (default), it uses the same formula as model_formula.

use_rcpp Logical. If TRUE (default), use our internal Rcpp implementation. If FALSE, use **glmmTMB**.

verbose Whether to print progress messages.

optimization_alg Optimization algorithm. Default is dispatched via policy.

InferenceCountZeroInflatedPoisson$compute_estimate(): Fits the zero-inflated Poisson model. Under the default estimand = "conditional", returns $\hat{\beta}_T$, the treatment log-rate coefficient from the conditional count submodel (see the class-level caveat that this is conditional on the response coming from the count process, not an unconditional-mean effect). Under estimand = "marginal_mean_diff" or "marginal_ratio" (set via set_estimand()), returns the g-computation marginal mean difference or log-scale marginal ratio of the unconditional mean $E[Y \mid w, x] = (1 - \pi(x))\lambda(x)$ instead — a pure post-fit transform of the same cached fit, no refit.

Usage:

```
InferenceCountZeroInflatedPoisson$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip standard-error computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferenceCountZeroInflatedPoisson$compute_asymp_confidence_interval()`: Asymptotic confidence interval. Under the conditional estimand, delegates to the shared zero-augmented count-model Wald/ bootstrap-fallback contract; under a marginal estimand, the delta-method interval computed by `compute_estimate()`. Calls `self$compute_estimate()` first (not `private$shared()` directly) so the estimand-aware cache is always current regardless of call order.

Usage:

```
InferenceCountZeroInflatedPoisson$compute_asymp_confidence_interval(
  alpha = 0.05
)
```

Arguments:

`alpha` The significance level (default 0.05).

`InferenceCountZeroInflatedPoisson$compute_asymp_two_sided_pval()`: Asymptotic two-sided p-value, dispatched exactly as `compute_asymp_confidence_interval()`; see that method's description for the marginal-estimand path.

Usage:

```
InferenceCountZeroInflatedPoisson$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment effect under the current estimand (default 0).

`InferenceCountZeroInflatedPoisson$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceCountZeroInflatedPoisson$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Lambert, D. (1992). "Zero-Inflated Poisson Regression, with an Application to Defects in Manufacturing." *Technometrics*, 34(1), 1-14, doi:10.2307/1269547, for the zero-inflated count-model framework.

See Also

`InferenceCountPoisson` for the single-part Poisson model this class's count submodel generalizes; `InferenceCountHurdlePoisson` for the related hurdle (disjoint-support) variant; `InferenceCountZeroInflatedNegB` for the overdispersion-robust negative-binomial variant (does not support a marginal estimand — the mean-function derivation here is Poisson-specific).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'count')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rpois(10, 2))
inf = InferenceCountZeroInflatedPoisson$new(seq_des)
inf$compute_estimate()
```

InferenceIncidBinomialIdentityRiskDiff

Binomial Identity Risk Difference Inference for Incidence Responses

---

Description

Fits a binomial regression with the **identity** link for binary (incidence) responses: $P(Y_i = 1) = \beta_0 + \beta_T W_i + X_i^\top \gamma$, where W_i is the treatment indicator and X_i are optional recorded covariates, by maximum likelihood (`fast_identity_binomial_regression_cpp`/ `fast_identity_binomial_regression_weighted_c` Because the link is the identity rather than the logit, $\hat{\beta}_T$ is directly a **risk difference** on the probability scale, not a log-odds-ratio — the class name and estimand differ from `InferenceIncidLogRegr` for exactly this reason. `likelihood_tier = "full"`: likelihood-ratio, score, gradient, and Wald tests are all available when the model converges, plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test. Because the identity link does not constrain fitted probabilities to $[0, 1]$, fits are hardened by QR column-dropping and rejected as nonestimable when the fitted linear predictor produces implausible coefficients (see `private$is_identity_binomial_fit_reasonable()`); this is a real practical limitation of the identity link relative to logit/probit, not a bug. Validity requires the additive risk-difference model to be correctly specified over the covariate range actually observed (an identity-link fit can be well-behaved in-sample yet imply out-of-range probabilities for other covariate values).

Estimand. Composes `MarginalEstimand` (`set_estimand()`/`get_estimand()`/`get_supported_estimands()`). `estimand = "marginal_mean_diff"` is supported for API consistency with the other GLM families, but **it is algebraically identical to the default** `estimand = "conditional"` **for this class**: the g-computation marginal risk difference is $\frac{1}{n}\sum_i\{(\hat{\beta}_0 + \hat{\beta}_T + X_i^\top \hat{\gamma}) - (\hat{\beta}_0 + X_i^\top \hat{\gamma})\}$, which simplifies to exactly $\hat{\beta}_T$ for every subject (not merely on average) because the identity link is linear in W_i with no treatment-by-covariate interaction term — the per-subject treated-minus-control difference $\hat{\beta}_T$ does not depend on X_i at all, so standardizing over the covariate distribution changes nothing. Contrast with `InferenceCountPoisson`'s `"marginal_ratio"` (also a collapsing case, for the same no-interaction reason) and `"marginal_mean_diff"` (which does not collapse, since a difference does not distribute through the nonlinear log-link mean). This collapsing is a genuine property of the identity-link model, not a wiring bug — it is documented here so a user comparing `estimands` for this class is not surprised the two never differ. Standard errors are computed independently for each estimand (the marginal path uses the delta method against the model's coefficient covariance; the conditional path uses the model information matrix), so while the two point estimates coincide exactly, their standard errors may differ slightly by construction even though both are asymptotically valid.

Super class

`Inference` -> `InferenceIncidBinomialIdentityRiskDiff`

Methods

Public methods:

- `InferenceIncidBinomialIdentityRiskDiff$compute_estimate()`
- `InferenceIncidBinomialIdentityRiskDiff$compute_asymp_confidence_interval()`
- `InferenceIncidBinomialIdentityRiskDiff$compute_asymp_two_sided_pval()`
- `InferenceIncidBinomialIdentityRiskDiff$clone()`

`InferenceIncidBinomialIdentityRiskDiff$compute_estimate():` Fits the identity-link binomial regression model by maximum likelihood. Under the default `estimand = "conditional"`, returns $\hat{\beta}_T$, the risk difference coefficient. Under `estimand = "marginal_mean_diff"` (set via `set_estimand()`), returns the g-computation marginal risk difference — see the class-level `@details` for why this is algebraically identical to the conditional estimate for this family. The underlying model fit is identical either way (a pure post-fit transform of the same cached fit, no refit).

Usage:

```
InferenceIncidBinomialIdentityRiskDiff$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip standard-error computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferenceIncidBinomialIdentityRiskDiff$compute_asymp_confidence_interval():` Wald confidence interval, dispatched by `testing_type` for the conditional estimand; under a marginal estimand `testing_type` is always `"wald"` (the only value `set_estimand()` permits there). Calls `self$compute_estimate()` first (not `private$shared()` directly) so the estimand-aware cache is always current regardless of call order.

Usage:

```
InferenceIncidBinomialIdentityRiskDiff$compute_asymp_confidence_interval(
  alpha = 0.05
)
```

Arguments:

`alpha` Two-sided miscoverage rate; the returned interval targets `1 - alpha` coverage.

`InferenceIncidBinomialIdentityRiskDiff$compute_asymp_two_sided_pval():` Wald two-sided p-value, dispatched by `testing_type` exactly as `compute_asymp_confidence_interval()`; see that method's description for the marginal-estimand always-Wald note.

Usage:

```
InferenceIncidBinomialIdentityRiskDiff$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment-effect value under the current estimand (both scales coincide for this family — see the class-level `@details`).

`InferenceIncidBinomialIdentityRiskDiff$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidBinomialIdentityRiskDiff$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

McCullagh, P., and Nelder, J. A. (1989). *Generalized Linear Models* (2nd ed.). Chapman and Hall/CRC, for the binomial GLM family and identity-link risk-difference parameterization.

See Also

`InferenceIncidLogRegr` (logit link, log-odds-ratio estimand), `InferenceIncidLogBinomial` (log link, log-risk-ratio estimand) for alternative link/estimand choices on the same response type. Comparable Python API: statsmodels GLM (`family=Binomial(link=identity())`). See also: Generalized linear model (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidBinomialIdentityRiskDiff$new(seq_des)
inf$compute_estimate()
```

`InferenceIncidCMH` *CMH Blocked Incidence Inference*

Description

Unadjusted blocked-design incidence inference using the simple mean-difference point estimate with a randomization-based standard error.

Legacy inference class. This class is retained for backwards compatibility and is not comprehensively tested by the package comprehensive-test harness.

Internally, this class recodes treatment assignments to $w_i \in \{-1, +1\}$ (the package-wide convention is $\{0, 1\}$; see `Design`). For a balanced design the treatment-effect estimator is $\hat{\tau} = (2/n)\,\mathbf{y}'\mathbf{w}$, and since $E_w[\mathbf{y}'\mathbf{w}] = 0$ for any balanced randomization the standard error is

$$SE(\hat{\tau}) = \frac{2}{n}\sqrt{\frac{\sum_k (\mathbf{y}'\mathbf{w}_k)^2}{K}}$$

where K draws $\mathbf{w}_1, \ldots, \mathbf{w}_K$ come from the design's reference distribution. Centering at the known zero mean (rather than the sample mean) makes the denominator K rather than $K - 1$.

For blocking designs the expectation is evaluated exactly:

$$SE(\hat{\tau}) = \frac{2}{n}\sqrt{\sum_b \frac{n_{1b}\, n_{0b}}{n_B - 1}}$$

where n_{1b}, n_{0b} are the numbers of positive and negative responses in block b and n_B is the (common) block size. This equals $2\sqrt{V_{\mathrm{CMH}}}$ where V_{CMH} is the CMH variance from Azriel et al. (2026), Equation 3.

For non-blocking designs, the "balanced design" precondition above requires the observed treatment allocation to be *exactly* balanced ($n_T = n_C$), not merely drawn from a $prob_T = 0.5$ mechanism – e.g. plain Bernoulli randomization has $prob_T = 0.5$ but does not guarantee an exactly balanced realized allocation. A warning (not an error) is issued once, the first time the standard error is actually computed (i.e. on the first confidence-interval / p-value / standard-error request, not at construction or for estimate-only use), when this is violated – erroring would make this class unusable with Bernoulli-style non-blocking designs entirely; the warning tells the caller the reported standard error may be miscalibrated.

Super class

`Inference` -> `InferenceIncidCMH`

Methods

Public methods:

- `InferenceIncidCMH$compute_asymp_confidence_interval()`
- `InferenceIncidCMH$compute_asymp_two_sided_pval()`
- `InferenceIncidCMH$new()`
- `InferenceIncidCMH$clone()`

`InferenceIncidCMH$compute_asymp_confidence_interval():` Uses the randomization-CI layer's two-sided p-value contract (`InferenceRandCI`'s version, not `InferenceRand`'s): for incidence responses this dispatches to the Zhang exact randomization test where applicable rather than refusing outright, matching this class's pre-migration old-ladder behavior (it inherited from `InferenceAllSimpleAverageDiff`, whose own pin was already corrected to `InferenceRandCI` – see that file's identical rationale). This class independently composes the same components rather than truly inheriting `InferenceAllSimpleAverageDiff`, so it had its own stale copy of the old `InferenceRand` pin, which silently regressed Zhang dispatch for the non-blocking balanced-design path – found via `test-incid-cmh-extended-robins-migration-golden.R`'s `randomization_pval` case going from ‘"ok"‘ to ‘"unsupported"‘.

Wald confidence interval for the balanced-design/CMH risk-difference estimate $\hat{\tau}$, using the randomization-based (blocking-design: exact CMH variance formula; non-blocking design: Monte Carlo over `se_est_num_vectors` design draws) standard error documented in the class `@details`. See `InferenceAsymp` for the shared Wald contract.

Usage:

```
InferenceIncidCMH$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

Returns: A length-2 numeric vector `c(lower, upper)` on the risk-difference scale.

`InferenceIncidCMH$compute_asymp_two_sided_pval()`: Two-sided Wald p-value for $H_0 : \tau =$ `delta` vs. $H_1 : \tau \neq$ `delta`, using the same randomization-based standard error as `compute_asymp_confidence_interval()`.

Usage:

```
InferenceIncidCMH$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null value of τ to test against; 0 (the default) tests for any treatment effect at all.

Returns: Numeric scalar p-value in $[0, 1]$.

`InferenceIncidCMH$new()`: Initialize Cochran-Mantel-Haenszel incidence inference, validate the stratified binary-response design, and prepare the stratum-adjusted test used by `InferenceIncidCMH`.

Usage:

```
InferenceIncidCMH$new(
  des_obj,
  model_formula = NULL,
  se_est_num_vectors = 5000L,
  verbose = FALSE
)
```

Arguments:

`des_obj` A completed design object.

`model_formula` Optional formula for covariate adjustment.

`se_est_num_vectors` For non-block designs, the number of randomization vectors drawn from the design to estimate the standard error. Default `1000L`.

`verbose` Logical. Whether to print progress messages.

Returns: A new `InferenceIncidCMH` object.

`InferenceIncidCMH$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidCMH$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneRandomBlockSize$new(n = 20, response_type = 'incidence',
  strata_cols = 'x1')
for (i in 1:20) {
  seq_des$add_one_subject_to_experiment_and_assign(
    data.frame(x1 = factor(rep(1:2, 10)[i], levels=1:2)))
}
seq_des$add_all_subject_responses(rbinom(20, 1, 0.5))
inf = InferenceIncidCMH$new(seq_des)
inf$compute_estimate()
```

`InferenceIncidExactBinomial`

Exact Binomial (McNemar-Type) Incidence Inference for Matched-Pair Designs

Description

Performs exact matched-pair inference for binary (incidence) outcomes using only **discordant** matched pairs — pairs where the treated and control member's outcomes differ — the same reduction classical McNemar's test makes. Writing d_+ for the count of discordant pairs where the treated subject had the event and the control did not, and d_- for the reverse, the point estimate is the Haldane-Anscombe continuity-corrected log odds ratio $\log\left((d_+ + 0.5)/(d_- + 0.5)\right)$; the confidence interval inverts the exact (Clopper-Pearson) binomial confidence interval for $d_+/(d_+ + d_-)$ against $1/2$ (via `stats::binom.test`) onto the log-odds scale. Its two-sided p-value uses the corresponding equal-tail (central) exact binomial rule against a null log odds ratio, so the interval and test invert one another for every null value. This class is available for `DesignFixedBinaryMatch` and KK matching-on-the-fly designs. For KK designs, only the matched-pair data are used and the reservoir is ignored. If there are no matched pairs, or no discordant pairs, the relevant quantities are reported as non-estimable rather than as `NaN`/`Inf`.

Initialize exact matched-pair binomial inference for incidence outcomes. Requires `des_obj` to be `DesignFixedBinaryMatch` or a KK matching-on-the-fly-capable design; errors otherwise. Requires an uncensored incidence response.

Computes the Haldane-Anscombe continuity-corrected matched-pair log odds ratio $\log\left((d_+ + 0.5)/(d_- + 0.5)\right)$ from the discordant matched-pair counts (see class documentation for the full model). `NA` if there are no matched pairs.

Value

A new `InferenceIncidExactBinomial` object.

The treatment estimate.

Super class

`Inference` -> `InferenceIncidExactBinomial`

Methods

Public methods:

- `InferenceIncidExactBinomial$new()`
- `InferenceIncidExactBinomial$compute_estimate()`
- `InferenceIncidExactBinomial$clone()`

`InferenceIncidExactBinomial$new()`:

Usage:

```
InferenceIncidExactBinomial$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed design object.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values by default.

`InferenceIncidExactBinomial$compute_estimate()`:

Usage:

```
InferenceIncidExactBinomial$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` Ignored for this estimator (the exact statistic is always cheap to compute; there is no separate variance step to skip).

`InferenceIncidExactBinomial$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidExactBinomial$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidExactBinomial$new(seq_des)
inf$compute_estimate()
```

InferenceIncidExactFisher

Exact Fisher (Conditional Hypergeometric) Incidence Inference

Description

Performs exact conditional inference for binary (incidence) outcomes via Fisher's exact test on one or more 2x2 (treated/control by case/noncase) tables. When the design provides no stratification structure (e.g. an unstructured or iBCRD design), a single overall 2x2 table is built and `fisher.test` is used directly, giving the conditional MLE odds ratio and its exact confidence interval/p-value. When the design has **blocking structure** (`DesignFixedBlocking`, `DesignSeqOneByOneSPBR`, `DesignSeqOneByOneRandomBlockSize`), a separate 2x2 table is built per block-defining covariate stratum. When the design has **matched-pair structure** (KK matching-on-the-fly designs), each matched pair becomes its own 2x2 table, with any reservoir (unmatched) subjects pooled into one additional stratum table. In either stratified case, `mantelhaen.test` (exact conditional test) is used instead, giving the common odds ratio across strata; stratified inference only supports testing/estimating against a null odds ratio of 1 (log odds ratio 0) — a non-zero null shift is rejected with an error. Strata with no cases or no noncases in either arm are dropped before analysis; if no informative strata remain, this errors rather than returning a degenerate result.

Initialize exact Fisher inference for incidence outcomes. Requires an uncensored incidence response; the design's structure (unstructured, blocked, or matched) determines the stratification used at estimation time (see class documentation).

Computes the log of the (conditional MLE, or common-odds-ratio if stratified) odds ratio from `fisher.test` or `mantelhaen.test` (see class documentation for which applies and why).

Value

A new `InferenceIncidExactFisher` object.

The treatment estimate.

Super class

`Inference` -> `InferenceIncidExactFisher`

Methods

Public methods:

- `InferenceIncidExactFisher$new()`
- `InferenceIncidExactFisher$compute_estimate()`
- `InferenceIncidExactFisher$clone()`

`InferenceIncidExactFisher$new()`:

Usage:

```
InferenceIncidExactFisher$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed design object.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values by default.

`InferenceIncidExactFisher$compute_estimate()`:

Usage:

```
InferenceIncidExactFisher$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` Ignored for this estimator (the exact statistic is always cheap to compute; there is no separate variance step to skip).

`InferenceIncidExactFisher$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidExactFisher$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Fisher, R. A. (1935). "The Logic of Inductive Inference." *Journal of the Royal Statistical Society*, 98(1), 39-82, doi:10.2307/2342435, for the exact conditional test underlying `fisher.test`; Mantel, N., and Haenszel, W. (1959). "Statistical Aspects of the Analysis of Data from Retrospective Studies of Disease." *Journal of the National Cancer Institute*, 22(4), 719-748, for the stratified common-odds-ratio test used when the design provides multiple strata.

Examples

```
des = DesignFixediBCRD$new(n = 20, response_type = 'incidence')
des$add_all_subjects_to_experiment(data.frame(x1 = rnorm(20)))
des$assign_w_to_all_subjects()
des$add_all_subject_responses(rbinom(20, 1, 0.5))
inf = InferenceIncidExactFisher$new(des)
inf$compute_estimate()
inf$compute_exact_two_sided_pval_for_treatment_effect()
```

`InferenceIncidExactZhang`

Exact Zhang Combined-Test Incidence Inference

Description

Performs exact inference for a binary (incidence) outcome that **combines** two exact component tests when the design has both matched-pair and reservoir (unmatched) subjects — an internal-to-this-package method (not drawn from external literature) analogous in spirit to `InferenceIncidExactBinomial` (matched pairs) and `InferenceIncidExactFisher` (unmatched 2x2 table), fused into one combined exact test rather than a Wald-style variance combination. The point estimate is always the Haldane-Anscombe continuity-corrected log odds ratio $\log\left((n_{11}+0.5)(n_{00}+0.5)/\left((n_{10}+0.5)(n_{01}+0.5)\right)\right)$ from the pooled 2×2 table across all subjects (matched and reservoir together). For **p-values** and **confidence intervals**, the two subsets are tested separately (an exact matched-pairs binomial test, as in `InferenceIncidExactBinomial`, on discordant pairs; an exact Fisher test on the reservoir 2×2 table, as in `InferenceIncidExactFisher`), and their p-values are combined via `combination_method`: `"Fisher"` (default; $-2(\log p_M + \log p_R) \sim \chi^2_4$ under independence), `"Stouffer"` (averaged z-scores), or `"min_p"` (Šidák-style $1-(1-\min(p_M, p_R))^2$). If only one of the two subsets is informative (e.g. a pure-Bernoulli design with no matching, or no discordant pairs), the combined p-value degenerates to that one component's p-value. Confidence intervals are obtained by numerically inverting (bisection) the combined p-value as a function of the hypothesized log odds ratio, starting from a normal-approximation (Haldane-Anscombe MLE) interval as the search bracket. Requires a Bernoulli-capable or matching-capable design.

Initialize exact Zhang combined-test incidence inference. Requires `des_obj` to be Bernoulli-capable or matching-capable, an uncensored incidence response.

Computes the Haldane-Anscombe continuity-corrected log odds ratio $\log\left((n_{11}+0.5)(n_{00}+0.5)/\left((n_{10}+0.5)(n_{01}+0.5)\right.\right.$ from the pooled 2×2 table across all subjects (matched and reservoir combined) — see class documentation for the full combined-test model.

Computes an exact confidence interval for the log odds ratio by bisection-inverting the combined matched-pairs + Fisher-exact p-value (see class documentation for the full combination methodology).

Value

A new `InferenceIncidExactZhang` object.

The treatment estimate.

A confidence interval.

Super class

`Inference` -> `InferenceIncidExactZhang`

Methods

Public methods:

- InferenceIncidExactZhang$new()
- InferenceIncidExactZhang$compute_estimate()
- InferenceIncidExactZhang$compute_exact_confidence_interval()
- InferenceIncidExactZhang$clone()

InferenceIncidExactZhang$new():

Usage:

```
InferenceIncidExactZhang$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed design object.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values by default.

InferenceIncidExactZhang$compute_estimate():

Usage:

```
InferenceIncidExactZhang$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` Ignored for this estimator (the exact statistic is always cheap to compute; there is no separate variance step to skip).

InferenceIncidExactZhang$compute_exact_confidence_interval():

Usage:

```
InferenceIncidExactZhang$compute_exact_confidence_interval(
  alpha = 0.05,
  pval_epsilon = 1e-06,
  type = NULL,
  args_for_type = NULL
)
```

Arguments:

`alpha` Significance level.

`pval_epsilon` Bisection tolerance for the inversion routine.

`type` Exact inference type; only `"Zhang"` (the default) is supported.

`args_for_type` Optional arguments keyed by exact type; recognizes `combination_method` (`"Fisher"` (default), `"Stouffer"`, or `"min_p"`) inside the `"Zhang"` entry.

InferenceIncidExactZhang$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidExactZhang$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 20, response_type = 'incidence')
for (i in 1:20) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(20, 1, 0.5))
inf = InferenceIncidExactZhang$new(seq_des)
inf$compute_estimate()
inf$compute_exact_two_sided_pval_for_treatment_effect()
```

InferenceIncidExtendedRobins

Extended Robins Blocked Incidence Inference

Description

Unadjusted blocked-design incidence inference using the simple mean-difference point estimate with a block-stratified standard error.

Legacy inference class. This class is retained for backwards compatibility and is not comprehensively tested by the package comprehensive-test harness.

Super class

Inference -> InferenceIncidExtendedRobins

Methods

Public methods:

- InferenceIncidExtendedRobins$compute_asymp_confidence_interval()
- InferenceIncidExtendedRobins$compute_asymp_two_sided_pval()
- InferenceIncidExtendedRobins$new()
- InferenceIncidExtendedRobins$clone()

InferenceIncidExtendedRobins$compute_asymp_confidence_interval(): Uses the randomization-CI layer's two-sided p-value contract (InferenceRandCI's version, not InferenceRand's): for incidence responses this dispatches to the Zhang exact randomization test where applicable rather than refusing outright, matching this class's pre-migration old-ladder behavior (it inherited from InferenceAllSimpleAverageDiff, whose own pin was already corrected to InferenceRandCI – see that file's identical rationale). This class independently composes the same components rather than truly inheriting InferenceAllSimpleAverageDiff, so it had its own stale copy of the

old InferenceRand pin (same bug as InferenceIncidWald/InferenceIncidCMH, fixed alongside them even though this class's own golden test's design doesn't happen to trigger the Zhang-eligible path that would have caught it).
Uses the shared asymptotic confidence-interval contract; see InferenceAsymp.

Usage:

```
InferenceIncidExtendedRobins$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

alpha Numeric. Significance level (default 0.05).

InferenceIncidExtendedRobins$compute_asymp_two_sided_pval(): Uses the shared asymptotic two-sided p-value contract; see InferenceAsymp.

Usage:

```
InferenceIncidExtendedRobins$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

delta Numeric. Null treatment effect value (default 0).

InferenceIncidExtendedRobins$new(): Initialize Extended Robins blocked-design incidence inference.

Usage:

```
InferenceIncidExtendedRobins$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE
)
```

Arguments:

des_obj A completed design object.

model_formula Optional formula for covariate adjustment.

verbose Logical. Whether to print progress messages.

Returns: A new InferenceIncidExtendedRobins object.

InferenceIncidExtendedRobins$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidExtendedRobins$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneRandomBlockSize$new(n = 20, response_type = 'incidence',
  strata_cols = 'x1')
for (i in 1:20) {
  seq_des$add_one_subject_to_experiment_and_assign(
    data.frame(x1 = factor(rep(1:2, 10)[i], levels=1:2)))
}
seq_des$add_all_subject_responses(rbinom(20, 1, 0.5))
inf = InferenceIncidExtendedRobins$new(seq_des)
inf$compute_estimate()
```

`InferenceIncidGCompRiskDiff`

G-Computation Risk-Difference Inference for Binary Responses

Description

Fits a logistic working model, $\text{logit}\ \Pr(Y_i = 1 \mid x_i) = x_i^\top \hat{\beta}$, for an incidence outcome using treatment and, optionally, all recorded covariates, then estimates the marginal (standardized) risk difference $\text{RD} = \overline{\text{risk}}_1 - \overline{\text{risk}}_0$ by G-computation: setting every subject's treatment indicator to 1 (respectively 0) while holding their other observed covariates fixed, averaging the model-implied risk over the empirical covariate distribution under each counterfactual, and differencing — see `gcomp_logistic_point_estimate_cpp` for the exact standardization formula. Wald inference uses the fitted logistic model's sandwich covariance and the delta method. This standard error conditions on the observed covariates used for standardization: it quantifies uncertainty in the fitted coefficients for the sample-conditional standardized risk difference, and does not add sampling variability from a target population's covariate distribution. Bootstrap, randomization, and jackknife methods remain available. `likelihood_tier = "none"` means that this class does not provide likelihood-ratio or score tests; it does not preclude its Wald inference.

Uses the shared nonparametric bootstrap distribution contract; see `InferenceNonParamBootstrap`.

Computes a bootstrap confidence interval for the treatment effect.

Computes a bootstrap two-sided p-value for the treatment effect.

Computes a Bayesian-bootstrap two-sided p-value for the treatment effect.

Computes a Bayesian-bootstrap confidence interval for the treatment effect.

Computes a jackknife-Wald two-sided p-value for the treatment effect.

Computes a jackknife-Wald confidence interval for the treatment effect.

Computes a PRW subsampling two-sided p-value for the treatment effect.

Computes a PRW subsampling confidence interval for the treatment effect.

Computes an m-out-of-n bootstrap two-sided p-value for the treatment effect.

Computes an m-out-of-n bootstrap confidence interval for the treatment effect.

Value

A numeric vector of bootstrap estimates.

Super class

`Inference` -> `InferenceIncidGCompRiskDiff`

Methods

Public methods:

- `InferenceIncidGCompRiskDiff$approximate_bootstrap_distribution_beta_hat_T()`
- `InferenceIncidGCompRiskDiff$compute_bootstrap_confidence_interval()`

- InferenceIncidGCompRiskDiff$compute_bootstrap_two_sided_pval()
- InferenceIncidGCompRiskDiff$compute_bayesian_bootstrap_two_sided_pval()
- InferenceIncidGCompRiskDiff$compute_bayesian_bootstrap_confidence_interval()
- InferenceIncidGCompRiskDiff$compute_jackknife_wald_two_sided_pval()
- InferenceIncidGCompRiskDiff$compute_jackknife_wald_confidence_interval()
- InferenceIncidGCompRiskDiff$compute_subsampling_two_sided_pval()
- InferenceIncidGCompRiskDiff$compute_subsampling_confidence_interval()
- InferenceIncidGCompRiskDiff$compute_m_out_of_n_bootstrap_two_sided_pval()
- InferenceIncidGCompRiskDiff$compute_m_out_of_n_bootstrap_confidence_interval()
- InferenceIncidGCompRiskDiff$clone()

InferenceIncidGCompRiskDiff$approximate_bootstrap_distribution_beta_hat_T():

Usage:

```
InferenceIncidGCompRiskDiff$approximate_bootstrap_distribution_beta_hat_T(
  B = 501,
  show_progress = TRUE,
  debug = FALSE,
  bootstrap_type = NULL
)
```

Arguments:

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`debug` Whether to return diagnostics.

`bootstrap_type` Optional resampling scheme.

`bootstrap_type` Optional empirical-resampling scheme.

`bootstrap_type` Optional empirical-resampling scheme.

`InferenceIncidGCompRiskDiff$compute_bootstrap_confidence_interval()`:

Usage:

```
InferenceIncidGCompRiskDiff$compute_bootstrap_confidence_interval(
  alpha = 0.05,
  B = 501,
  type = NULL,
  na.rm = TRUE,
  show_progress = TRUE,
  min_number_usable_samples = 5L
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default `0.05`.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`InferenceIncidGCompRiskDiff$compute_bootstrap_two_sided_pval()`:

Usage:

```
InferenceIncidGCompRiskDiff$compute_bootstrap_two_sided_pval(
  delta = NULL,
  B = 501,
  type = "symmetric",
  na.rm = FALSE,
  show_progress = TRUE,
  min_number_usable_samples = 5L
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`InferenceIncidGCompRiskDiff$compute_bayesian_bootstrap_two_sided_pval():`

Usage:

```
InferenceIncidGCompRiskDiff$compute_bayesian_bootstrap_two_sided_pval(
  delta = NULL,
  B = 501,
  type = NULL,
  na.rm = FALSE,
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  weighting_unit_type = NULL
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.
`B` Number of Bayesian-bootstrap samples.
`B` Number of Bayesian-bootstrap samples.
`B` Number of subsamples.
`B` Number of subsamples.
`B` Number of resamples.
`B` Number of resamples.
`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.
`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.
`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`
`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`
`type` P-value type.
`type` Confidence-interval type.
`type` P-value type.
`type` Confidence-interval type.
`na.rm` Whether to remove non-finite bootstrap replicates.
`na.rm` Whether to remove non-finite bootstrap replicates.
`na.rm` Whether to remove non-finite bootstrap replicates.
`na.rm` Whether to remove non-finite bootstrap replicates.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`weighting_unit_type` Optional resampling unit override.
`weighting_unit_type` Optional resampling unit override.

`InferenceIncidGCompRiskDiff$compute_bayesian_bootstrap_confidence_interval()`:

Usage:

```
InferenceIncidGCompRiskDiff$compute_bayesian_bootstrap_confidence_interval(
  alpha = 0.05,
  B = 501,
  type = NULL,
  na.rm = TRUE,
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  weighting_unit_type = NULL
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default `0.05`.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`weighting_unit_type` Optional resampling unit override.
`weighting_unit_type` Optional resampling unit override.

`InferenceIncidGCompRiskDiff$compute_jackknife_wald_two_sided_pval()`:

Usage:

```
InferenceIncidGCompRiskDiff$compute_jackknife_wald_two_sided_pval(
  delta = NULL,
  unit = "auto"
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.

`InferenceIncidGCompRiskDiff$compute_jackknife_wald_confidence_interval()`:

Usage:

```
InferenceIncidGCompRiskDiff$compute_jackknife_wald_confidence_interval(
  alpha = 0.05,
  unit = "auto"
)
```

Arguments:

`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default `0.05`.
`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.

`InferenceIncidGCompRiskDiff$compute_subsampling_two_sided_pval()`:

Usage:

```
InferenceIncidGCompRiskDiff$compute_subsampling_two_sided_pval(
  delta = NULL,
  B = 501,
  b = NULL,
  type = "centered",
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  subsampling_type = NULL
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`b` Subsample size. See `InferenceNonParamBootstrap$compute_subsampling_two_sided_pval`.

`b` Subsample size. See `InferenceNonParamBootstrap$compute_subsampling_confidence_interval`.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

show_progress Whether to show a progress bar.

min_number_usable_samples Minimum number of finite bootstrap samples required.

min_number_usable_samples Minimum number of finite bootstrap samples required.

min_number_usable_samples Minimum number of finite bootstrap samples required.

min_number_usable_samples Minimum number of finite bootstrap samples required.

min_number_usable_samples Minimum number of finite subsampled estimates required.

min_number_usable_samples Minimum number of finite subsampled estimates required.

min_number_usable_samples Minimum number of finite resampled estimates required.

min_number_usable_samples Minimum number of finite resampled estimates required.

subsampling_type Optional empirical-resampling scheme.

subsampling_type Optional empirical-resampling scheme.

`InferenceIncidGCompRiskDiff$compute_subsampling_confidence_interval()`:

Usage:

```
InferenceIncidGCompRiskDiff$compute_subsampling_confidence_interval(
  alpha = 0.05,
  B = 501,
  b = NULL,
  type = "basic",
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  subsampling_type = NULL
)
```

Arguments:

alpha Significance level. Default 0.05.

alpha Significance level. Default 0.05.

alpha Significance level. Default `0.05`.

alpha Significance level. Default 0.05.

alpha Significance level. Default 0.05.

B Number of bootstrap samples.

B Number of bootstrap samples.

B Number of bootstrap samples.

B Number of Bayesian-bootstrap samples.

B Number of Bayesian-bootstrap samples.

B Number of subsamples.

B Number of subsamples.

B Number of resamples.

B Number of resamples.

b Subsample size. See `InferenceNonParamBootstrap$compute_subsampling_two_sided_pval`.

b Subsample size. See `InferenceNonParamBootstrap$compute_subsampling_confidence_interval`.

type Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

type Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

type Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`
`type` P-value type.
`type` Confidence-interval type.
`type` P-value type.
`type` Confidence-interval type.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`subsampling_type` Optional empirical-resampling scheme.
`subsampling_type` Optional empirical-resampling scheme.

`InferenceIncidGCompRiskDiff$compute_m_out_of_n_bootstrap_two_sided_pval()`:

Usage:

```
InferenceIncidGCompRiskDiff$compute_m_out_of_n_bootstrap_two_sided_pval(
  delta = NULL,
  B = 501,
  m = NULL,
  type = "centered",
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  bootstrap_type = NULL
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.
`B` Number of bootstrap samples.

`B` Number of bootstrap samples.
`B` Number of bootstrap samples.
`B` Number of Bayesian-bootstrap samples.
`B` Number of Bayesian-bootstrap samples.
`B` Number of subsamples.
`B` Number of subsamples.
`B` Number of resamples.
`B` Number of resamples.
`m` Resample size. See `InferenceNonParamBootstrap$compute_m_out_of_n_bootstrap_two_sided_pval`.
`m` Resample size. See `InferenceNonParamBootstrap$compute_m_out_of_n_bootstrap_confidence_interval`.
`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.
`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.
`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_sid`
`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`
`type` P-value type.
`type` Confidence-interval type.
`type` P-value type.
`type` Confidence-interval type.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`bootstrap_type` Optional resampling scheme.
`bootstrap_type` Optional empirical-resampling scheme.
`bootstrap_type` Optional empirical-resampling scheme.

`InferenceIncidGCompRiskDiff$compute_m_out_of_n_bootstrap_confidence_interval()`:

Usage:

```
InferenceIncidGCompRiskDiff$compute_m_out_of_n_bootstrap_confidence_interval(
  alpha = 0.05,
  B = 501,
  m = NULL,
  type = "basic",
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  bootstrap_type = NULL
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default `0.05`.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`m` Resample size. See `InferenceNonParamBootstrap$compute_m_out_of_n_bootstrap_two_sided_pval`.

`m` Resample size. See `InferenceNonParamBootstrap$compute_m_out_of_n_bootstrap_confidence_interval`.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`bootstrap_type` Optional resampling scheme.

`bootstrap_type` Optional empirical-resampling scheme.

`bootstrap_type` Optional empirical-resampling scheme.

`InferenceIncidGCompRiskDiff$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidGCompRiskDiff$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`InferenceIncidGCompRiskRatio` for the risk-ratio version of this same standardized logistic working model.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidGCompRiskDiff$new(seq_des)
inf$compute_estimate()
```

`InferenceIncidGCompRiskRatio`

G-Computation Risk-Ratio Inference for Binary Responses

Description

Fits a logistic working model, $\text{logit}\ \Pr(Y_i = 1 \mid x_i) = x_i^\top \hat{\beta}$, for an incidence outcome using treatment and, optionally, all recorded covariates, then estimates the marginal (standardized) risk ratio $\text{RR} = \overline{\text{risk}}_1 / \overline{\text{risk}}_0$ by G-computation: setting every subject's treatment indicator to 1 (respectively 0) while holding their other observed covariates fixed, averaging the model-implied risk over the empirical covariate distribution under each counterfactual, and taking the ratio — see `gcomp_logistic_point_estimate_cpp` for the exact standardization formula (`mean1/mean0`). Bootstrap/jackknife inference on this estimand is generally done on the **log** risk-ratio scale internally (see `$compute_bootstrap_confidence_interval()`, `$compute_bayesian_bootstrap_confidence_interval()`, and the jackknife-Wald methods, whose `"basic"/"wald"` interval types route through log-scale-specific helpers for this estimand), then back-transformed, since ratio estimators are typically closer to normally distributed on the log scale. Wald inference uses the fitted logistic model's sandwich covariance and the delta method on the log risk-ratio scale. This standard error conditions on the observed covariates used for standardization: it quantifies uncertainty in the fitted coefficients for the sample-conditional standardized risk ratio, and does not add sampling variability from a target population's covariate distribution. Bootstrap, randomization, and jackknife methods remain available. `likelihood_tier = "none"` means that this class does not provide likelihood-ratio or score tests; it does not preclude its Wald inference.

Uses the shared nonparametric bootstrap distribution contract; see `InferenceNonParamBootstrap`.

Computes a bootstrap confidence interval for the treatment effect.

Computes a bootstrap two-sided p-value for the treatment effect.

Computes a Bayesian-bootstrap two-sided p-value for the treatment effect.

Computes a Bayesian-bootstrap confidence interval for the treatment effect.

Computes a jackknife-Wald two-sided p-value for the treatment effect.

Computes a jackknife-Wald confidence interval for the treatment effect.

Computes a PRW subsampling two-sided p-value for the treatment effect.

Computes a PRW subsampling confidence interval for the treatment effect.

Computes an m-out-of-n bootstrap two-sided p-value for the treatment effect.

Computes an m-out-of-n bootstrap confidence interval for the treatment effect.

Value

A numeric vector of bootstrap estimates.

Super class

`Inference` -> `InferenceIncidGCompRiskRatio`

Methods

Public methods:

- `InferenceIncidGCompRiskRatio$approximate_bootstrap_distribution_beta_hat_T()`
- `InferenceIncidGCompRiskRatio$compute_bootstrap_confidence_interval()`
- `InferenceIncidGCompRiskRatio$compute_bootstrap_two_sided_pval()`

- InferenceIncidGCompRiskRatio$compute_bayesian_bootstrap_two_sided_pval()
- InferenceIncidGCompRiskRatio$compute_bayesian_bootstrap_confidence_interval()
- InferenceIncidGCompRiskRatio$compute_jackknife_wald_two_sided_pval()
- InferenceIncidGCompRiskRatio$compute_jackknife_wald_confidence_interval()
- InferenceIncidGCompRiskRatio$compute_subsampling_two_sided_pval()
- InferenceIncidGCompRiskRatio$compute_subsampling_confidence_interval()
- InferenceIncidGCompRiskRatio$compute_m_out_of_n_bootstrap_two_sided_pval()
- InferenceIncidGCompRiskRatio$compute_m_out_of_n_bootstrap_confidence_interval()
- InferenceIncidGCompRiskRatio$clone()

InferenceIncidGCompRiskRatio$approximate_bootstrap_distribution_beta_hat_T():

Usage:

```
InferenceIncidGCompRiskRatio$approximate_bootstrap_distribution_beta_hat_T(
  B = 501,
  show_progress = TRUE,
  debug = FALSE,
  bootstrap_type = NULL
)
```

Arguments:

B Number of bootstrap samples.
B Number of bootstrap samples.
B Number of bootstrap samples.
B Number of Bayesian-bootstrap samples.
B Number of Bayesian-bootstrap samples.
B Number of subsamples.
B Number of subsamples.
B Number of resamples.
B Number of resamples.
show_progress Whether to show a progress bar.
show_progress Whether to show a progress bar.
show_progress Whether to show a progress bar.
show_progress Whether to show a progress bar.
show_progress Whether to show a progress bar.
show_progress Whether to show a progress bar.
show_progress Whether to show a progress bar.
show_progress Whether to show a progress bar.
show_progress Whether to show a progress bar.
debug Whether to return diagnostics.
bootstrap_type Optional resampling scheme.
bootstrap_type Optional empirical-resampling scheme.
bootstrap_type Optional empirical-resampling scheme.

InferenceIncidGCompRiskRatio$compute_bootstrap_confidence_interval():

Usage:

```
InferenceIncidGCompRiskRatio$compute_bootstrap_confidence_interval(
  alpha = 0.05,
  B = 501,
  type = NULL,
  na.rm = TRUE,
  show_progress = TRUE,
  min_number_usable_samples = 5L
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default `0.05`.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_sid`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`InferenceIncidGCompRiskRatio$compute_bootstrap_two_sided_pval()`:

Usage:

```
InferenceIncidGCompRiskRatio$compute_bootstrap_two_sided_pval(
  delta = NULL,
  B = 501,
  type = "symmetric",
  na.rm = FALSE,
  show_progress = TRUE,
  min_number_usable_samples = 5L
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`InferenceIncidGCompRiskRatio$compute_bayesian_bootstrap_two_sided_pval()`:

Usage:

```
InferenceIncidGCompRiskRatio$compute_bayesian_bootstrap_two_sided_pval(
  delta = NULL,
  B = 501,
  type = NULL,
  na.rm = FALSE,
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  weighting_unit_type = NULL
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`weighting_unit_type` Optional resampling unit override.

`weighting_unit_type` Optional resampling unit override.

`InferenceIncidGCompRiskRatio$compute_bayesian_bootstrap_confidence_interval()`:

Usage:

```
InferenceIncidGCompRiskRatio$compute_bayesian_bootstrap_confidence_interval(
  alpha = 0.05,
  B = 501,
  type = NULL,
  na.rm = TRUE,
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  weighting_unit_type = NULL
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default `0.05`.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`na.rm` Whether to remove non-finite bootstrap replicates.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`weighting_unit_type` Optional resampling unit override.
`weighting_unit_type` Optional resampling unit override.

`InferenceIncidGCompRiskRatio$compute_jackknife_wald_two_sided_pval()`:

Usage:

```
InferenceIncidGCompRiskRatio$compute_jackknife_wald_two_sided_pval(
  delta = NULL,
  unit = "auto"
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.

`InferenceIncidGCompRiskRatio$compute_jackknife_wald_confidence_interval()`:

Usage:

```
InferenceIncidGCompRiskRatio$compute_jackknife_wald_confidence_interval(
  alpha = 0.05,
  unit = "auto"
)
```

Arguments:

`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default `0.05`.
`alpha` Significance level. Default 0.05.
`alpha` Significance level. Default 0.05.
`unit` Deletion unit. Default `"auto"`.
`unit` Deletion unit. Default `"auto"`.

`InferenceIncidGCompRiskRatio$compute_subsampling_two_sided_pval()`:

Usage:

```
InferenceIncidGCompRiskRatio$compute_subsampling_two_sided_pval(
  delta = NULL,
  B = 501,
  b = NULL,
  type = "centered",
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  subsampling_type = NULL
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`b` Subsample size. See `InferenceNonParamBootstrap$compute_subsampling_two_sided_pval`.

`b` Subsample size. See `InferenceNonParamBootstrap$compute_subsampling_confidence_interval`.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`subsampling_type` Optional empirical-resampling scheme.

`subsampling_type` Optional empirical-resampling scheme.

`InferenceIncidGCompRiskRatio$compute_subsampling_confidence_interval()`:

Usage:

```
InferenceIncidGCompRiskRatio$compute_subsampling_confidence_interval(
  alpha = 0.05,
  B = 501,
  b = NULL,
  type = "basic",
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  subsampling_type = NULL
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default `0.05`.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`b` Subsample size. See `InferenceNonParamBootstrap$compute_subsampling_two_sided_pval`.

`b` Subsample size. See `InferenceNonParamBootstrap$compute_subsampling_confidence_interval`.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_side`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`
`type` P-value type.
`type` Confidence-interval type.
`type` P-value type.
`type` Confidence-interval type.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`subsampling_type` Optional empirical-resampling scheme.
`subsampling_type` Optional empirical-resampling scheme.

`InferenceIncidGCompRiskRatio$compute_m_out_of_n_bootstrap_two_sided_pval()`:

Usage:

```
InferenceIncidGCompRiskRatio$compute_m_out_of_n_bootstrap_two_sided_pval(
  delta = NULL,
  B = 501,
  m = NULL,
  type = "centered",
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  bootstrap_type = NULL
)
```

Arguments:

`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` The null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.
`delta` Null treatment effect. Defaults to 0 for RD and 1 for RR.
`B` Number of bootstrap samples.

`B` Number of bootstrap samples.
`B` Number of bootstrap samples.
`B` Number of Bayesian-bootstrap samples.
`B` Number of Bayesian-bootstrap samples.
`B` Number of subsamples.
`B` Number of subsamples.
`B` Number of resamples.
`B` Number of resamples.
`m` Resample size. See `InferenceNonParamBootstrap$compute_m_out_of_n_bootstrap_two_sided_pval`.
`m` Resample size. See `InferenceNonParamBootstrap$compute_m_out_of_n_bootstrap_confidence_interval`.
`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.
`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.
`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_sid`
`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`
`type` P-value type.
`type` Confidence-interval type.
`type` P-value type.
`type` Confidence-interval type.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`show_progress` Whether to show a progress bar.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite bootstrap samples required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite subsampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`min_number_usable_samples` Minimum number of finite resampled estimates required.
`bootstrap_type` Optional resampling scheme.
`bootstrap_type` Optional empirical-resampling scheme.
`bootstrap_type` Optional empirical-resampling scheme.

`InferenceIncidGCompRiskRatio$compute_m_out_of_n_bootstrap_confidence_interval()`:

Usage:

```
InferenceIncidGCompRiskRatio$compute_m_out_of_n_bootstrap_confidence_interval(
  alpha = 0.05,
  B = 501,
  m = NULL,
  type = "basic",
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  bootstrap_type = NULL
)
```

Arguments:

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default `0.05`.

`alpha` Significance level. Default 0.05.

`alpha` Significance level. Default 0.05.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of Bayesian-bootstrap samples.

`B` Number of subsamples.

`B` Number of subsamples.

`B` Number of resamples.

`B` Number of resamples.

`m` Resample size. See `InferenceNonParamBootstrap$compute_m_out_of_n_bootstrap_two_sided_pval`.

`m` Resample size. See `InferenceNonParamBootstrap$compute_m_out_of_n_bootstrap_confidence_interval`.

`type` Bootstrap CI type. See `InferenceNonParamBootstrap$compute_bootstrap_confidence_interval`.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`type` Bayesian-bootstrap p-value type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_two_sid`

`type` Bayesian-bootstrap CI type. See `InferenceBayesianBootstrap$compute_bayesian_bootstrap_confidence_`

`type` P-value type.

`type` Confidence-interval type.

`type` P-value type.

`type` Confidence-interval type.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`show_progress` Whether to show a progress bar.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite subsampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`min_number_usable_samples` Minimum number of finite resampled estimates required.

`bootstrap_type` Optional resampling scheme.

`bootstrap_type` Optional empirical-resampling scheme.

`bootstrap_type` Optional empirical-resampling scheme.

`InferenceIncidGCompRiskRatio$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidGCompRiskRatio$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`InferenceIncidGCompRiskDiff` for the risk-difference version of this same standardized logistic working model.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidGCompRiskRatio$new(seq_des)
inf$compute_estimate()
```

`InferenceIncidKKCondLogitGLMMIVWC`

Conditional Logistic Plus GLMM IVWC Inference for KK Designs

Description

Fits a combined conditional-logit-plus-random-intercept-GLMM likelihood for incidence responses under a KK matching-on-the-fly design, where **reservoir (unmatched) subjects are excluded from the GLMM component** (`private$combine_reservoir_into_glmm() == FALSE`): only concordant matched pairs contribute their random-intercept GLMM likelihood alongside the discordant-pair conditional-logit term, both sharing a single treatment coefficient β_T. See `InferencePropKKGLMM` for the full model form (conditional-logit-on-discordant plus random-intercept-GLMM, jointly maximized) and `InferenceAbstractKKCondLogitGLMM` for the shared fitting/caching contract. Contrast with the sibling `InferenceIncidKKCondLogitGLMMOneLik`, which instead includes reservoir subjects in the GLMM component (`combine_reservoir_into_glmm() == TRUE`) — this class's naming ("IVWC") reflects that reservoir information, when used, is intended to be combined with this fit's estimate via inverse-variance weighting rather than folded into the same likelihood.

Details

Legacy class. Not fully tested in `comprehensive_tests.R`. See `InferenceAbstractKKCondLogitGLMM`'s known estimability limitation at high covariate counts, which applies here too.

Super classes

`Inference` -> `InferenceAbstractKKCondLogitGLMM` -> `InferenceIncidKKCondLogitGLMMIVWC`

Methods

Public methods:

- `InferenceIncidKKCondLogitGLMMIVWC$clone()`

`InferenceIncidKKCondLogitGLMMIVWC$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidKKCondLogitGLMMIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidKKCondLogitGLMMIVWC$new(seq_des)
inf$compute_estimate()
```

InferenceIncidKKCondLogitGLMMOneLik

Conditional Logistic Plus GLMM Combined-Likelihood Inference for KK Designs

Description

Fits a combined conditional-logit-plus-random-intercept-GLMM likelihood for incidence responses under a KK matching-on-the-fly design, where **reservoir (unmatched) subjects are included in the GLMM component** (`private$combine_reservoir_into_glmm() == TRUE`), so all subjects (discordant matched pairs, concordant matched pairs, and reservoir) enter one joint likelihood with a single treatment coefficient β_T. See `InferencePropKKGLMM` for the full model form (conditional-logit-on-discordant plus random-intercept-GLMM, jointly maximized) and `InferenceAbstractKKCondLogitGLMM` for the shared fitting/caching contract. Contrast with the sibling `InferenceIncidKKCondLogitGLMMIVWC`, which excludes reservoir subjects from the GLMM component. See `InferenceAbstractKKCondLogitGLMM`'s known estimability limitation at high covariate counts, which applies here too.

Super classes

`Inference` -> `InferenceAbstractKKCondLogitGLMM` -> `InferenceIncidKKCondLogitGLMMOneLik`

Methods

Public methods:

- `InferenceIncidKKCondLogitGLMMOneLik$new()`
- `InferenceIncidKKCondLogitGLMMOneLik$clone()`

`InferenceIncidKKCondLogitGLMMOneLik$new()`**:** Initialize inference for the combined conditional-logit (discordant matched pairs) plus random-intercept-GLMM (concordant pairs and reservoir subjects) incidence model; see `InferenceIncidKKCondLogitGLMMOneLik` for the model form. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferenceIncidKKCondLogitGLMMOneLik$new(
  des_obj,
  model_formula = NULL,
  max_abs_reasonable_coef = 50,
  max_abs_reasonable_se = 1.25,
  max_abs_log_sigma = 8,
  verbose = FALSE,
  smart_cold_start_default = NULL,
  optimization_alg = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with an incidence response.

`model_formula` Optional formula for covariate adjustment.

`max_abs_reasonable_coef` Cap for reasonable coefficient estimates.

`max_abs_reasonable_se` Cap for reasonable treatment standard errors.

`max_abs_log_sigma` Cap for reasonable log random effect variance.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart optimizer start values.

`optimization_alg` Character. Optimization algorithm (default "lbfgs").

`InferenceIncidKKCondLogitGLMMOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidKKCondLogitGLMMOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidKKCondLogitGLMMOneLik$new(seq_des)
inf$compute_estimate()
```

InferenceIncidKKCondLogitIVWC

Conditional Logistic IVWC Inference (KK Designs, Binary Response)

Description

Inverse-variance-weighted combination (IVWC) of two independently fit conditional-likelihood pieces for KK matched-pair-plus-reservoir binary designs: matched pairs are analyzed with exact conditional logistic regression (`conditional_logit_fit_matched_pairs()`, which conditions out the pair-specific nuisance intercept and estimates only the treatment log-odds-ratio β_T from discordant pairs, or the joint `clogit`-style likelihood when covariates are present), and reservoir subjects are analyzed with ordinary logistic regression (`conditional_logit_fit_reservoir()`). If $\hat{\beta}_m, \hat{\sigma}_m^2$ and $\hat{\beta}_r, \hat{\sigma}_r^2$ are the matched-pair and reservoir estimates and their variances, the combined estimate is the variance-weighted average

$$\hat{\beta}_T = w^\star \hat{\beta}_m + (1 - w^\star)\hat{\beta}_r, \quad w^\star = \frac{\hat{\sigma}_r^2}{\hat{\sigma}_r^2 + \hat{\sigma}_m^2},$$

with combined variance $\hat{\sigma}_m^2 \hat{\sigma}_r^2 / (\hat{\sigma}_m^2 + \hat{\sigma}_r^2)$. This is the classical fixed-effects inverse-variance meta-analysis pooling formula (see Cochrane Handbook / DerSimonian-Laird), applied here to combine

the two conditionally-independent likelihood contributions of a KK design rather than to pool separate studies. When only one of the two components is estimable the combined estimate falls back to that component alone. Contrast this with `InferenceIncidKKCondLogitOneLik`, which instead fits a single joint likelihood over both pieces (see that class's documentation) – `likelihood_tier = "partial"` here reflects that the matched-pair piece is a genuine conditional (partial) likelihood, but the two-piece combination itself is a closed-form Wald/meta-analytic step, not a further likelihood evaluation.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Details

Legacy class. Not fully tested in `comprehensive_tests.R`.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Super class

`Inference` -> `InferenceIncidKKCondLogitIVWC`

Methods

Public methods:

- `InferenceIncidKKCondLogitIVWC$approximate_randomization_distribution_beta_hat_T()`
- `InferenceIncidKKCondLogitIVWC$supports_rand_pval_for_incidence()`
- `InferenceIncidKKCondLogitIVWC$clone()`

`InferenceIncidKKCondLogitIVWC$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceIncidKKCondLogitIVWC$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
```

```
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`delta` The null difference. Default 0.

`transform_responses` Type of transformation. Default "none".

`show_progress` Show progress bar. Default TRUE.

`permutations` Pre-computed permutations. Default NULL.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceIncidKKCondLogitIVWC$supports_rand_pval_for_incidence():`

Usage:

```
InferenceIncidKKCondLogitIVWC$supports_rand_pval_for_incidence()
```

`InferenceIncidKKCondLogitIVWC$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidKKCondLogitIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Fleiss, J.L., Levin, B., Paik, M.C. (2003). *Statistical Methods for Rates and Proportions*, 3rd ed. Wiley. (conditional logistic regression for matched pairs)

See Also

`InferenceIncidKKCondLogitOneLik` for the one-likelihood alternative combining strategy.

`InferenceIncidKKCondLogitOneLik`
One-Likelihood Conditional-Logistic Inference for KK Binary Designs

---

Description

Estimates a treatment log-odds-ratio β_T for binary (incidence) outcomes collected under a KK matching-on-the-fly design (DesignSeqOneByOneKK14 or subclass) by maximizing one combined likelihood that couples a conditional-logistic (intercept-free, within-matched-pair) likelihood for matched subjects with an ordinary logistic likelihood for reservoir subjects, sharing a single treatment coefficient across both pieces. This is the "one-likelihood" counterpart to InferenceIncidKKCondLogitIVWC, which instead fits the matched and reservoir pieces separately and pools them by inverse-variance weighting; here the treatment coefficient is a single joint MLE, and `likelihood_tier = "full"` exposes likelihood-ratio, score, and gradient inference plus a parametric likelihood bootstrap in addition to Wald.

Estimand. β_T, the treatment coefficient of a logistic mean model $\text{logit}(P(Y = 1 \mid w, x)) = \beta_0 + \beta_T w + x\beta$; $\exp(\hat{\beta}_T)$ is the treatment-vs-control odds ratio.

Model. Matched pairs contribute McFadden-style conditional logistic likelihood terms that condition away the pair-specific nuisance intercept (see `build_matching_combined_clogit_design_cpp`/ `collect_discordant_pairs_cpp`); reservoir subjects contribute an ordinary logistic likelihood with one shared intercept. The combined negative log-likelihood is minimized jointly in $(\beta_0, \beta_T, \beta)$ via `fast_logistic_regression_cpp`/ `fast_logistic_regression_with_var_cpp`. When `get_testing_type() != "wald"`, asymptotic CI/p-value calls are routed through InferenceAsympLik's generic score/likelihood-ratio/gradient dispatch instead of the design's own Wald machinery.

Assumptions. Independence across matched pairs and reservoir subjects given covariates; correct logistic mean specification; a KK matching-on-the-fly design supplying the matched/reservoir partition. No response censoring is supported (checked at construction via `assertNoCensoring()`).

Super class

Inference -> InferenceIncidKKCondLogitOneLik

Methods

Public methods:

- InferenceIncidKKCondLogitOneLik$compute_rand_two_sided_pval()
- InferenceIncidKKCondLogitOneLik$clone()

`InferenceIncidKKCondLogitOneLik$compute_rand_two_sided_pval()`**:** Computes a randomization-based two-sided p-value for the treatment effect, preflighting the observed combined-likelihood treatment statistic (see the class-header note above) before delegating to InferenceRandCI's Zhang-dispatch-aware implementation.

Usage:

```
InferenceIncidKKCondLogitOneLik$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  type = NULL,
  args_for_type = NULL,
```

```
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization (permutation) draws.

`delta` The null treatment effect. Default 0.

`transform_responses` Optional response transform applied before the randomization statistic is computed. Default `"none"`.

`na.rm` Whether to remove non-finite permutation replicates.

`show_progress` Whether to show a progress bar.

`permutations` Optional pre-generated permutation matrix/list to reuse instead of drawing new permutations.

`type` Optional randomization-statistic type override.

`args_for_type` Optional list of extra arguments for `type`.

`zero_one_logit_clamp` Clamp applied to responses at the 0/1 boundary before a logit-scale transform, to avoid infinite values. Default `.Machine$double.eps`.

`InferenceIncidKKCondLogitOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidKKCondLogitOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A., and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388. doi:10.1111/biom.12148. (KK14 in `REFERENCES.md`.)

See Also

Analogous Python API for conditional logistic regression: statsmodels discrete models (`ConditionalLogit`). Logistic regression (orientation).

`InferenceIncidKKGCompRiskDiff`

G-Computation Risk-Difference Inference for KK Designs with Binary Responses

---

Description

Fits an all-subject logistic working model $\operatorname{logit} P(Y = 1 \mid w, x) = \beta_0 + \beta_T w + \beta_X^\top x$ for a KK incidence outcome using treatment w and, optionally, all recorded covariates x, then estimates the marginal (standardized, g-computation) risk difference $\hat{\theta} = n^{-1} \sum_i \{\hat{p}(1, x_i) - \hat{p}(0, x_i)\}$ by averaging the fitted-model predicted risks under all-treated and all-control assignments over the empirical covariate distribution (Robins 1986). Matched pairs are treated as clusters and reservoir subjects are treated as singletons when computing the sandwich covariance of the standardized estimator (the delta-method variance of the empirical mean of the two counterfactual-risk contrasts, not the naive logistic-regression coefficient variance).

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Details

This estimator has `likelihood_tier = "none"`: the fitted logistic model is a working model for standardization only, and reported inference is sandwich/bootstrap-based, not likelihood-based. `compute_estimate()` fits the model and returns $\hat{\theta}$ on the risk-difference (probability) scale; `compute_estimate_with_boots` refits under Bayesian-bootstrap subject weights for `compute_bayesian_bootstrap_confidence_interval()`. Those weights affect the working-model fit only: the resulting predictions are still averaged equally over the retained empirical covariate rows, so every g-computation implementation in EDI targets the same sample-conditional standardized estimand. Jackknife deletes one cluster (matched pair or singleton reservoir subject) at a time. If the working model fails to converge or the design has no treatment-arm variation, the estimate is marked non-estimable via `is_nonestimable()`.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Super class

`Inference` -> `InferenceIncidKKGCompRiskDiff`

Methods

Public methods:

- `InferenceIncidKKGCompRiskDiff$approximate_randomization_distribution_beta_hat_T()`

- InferenceIncidKKGCompRiskDiff$supports_rand_pval_for_incidence()
- InferenceIncidKKGCompRiskDiff$clone()

`InferenceIncidKKGCompRiskDiff$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceIncidKKGCompRiskDiff$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`delta` The null difference. Default 0.

`transform_responses` Type of transformation. Default "none".

`show_progress` Show progress bar. Default TRUE.

`permutations` Pre-computed permutations. Default NULL.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceIncidKKGCompRiskDiff$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceIncidKKGCompRiskDiff$supports_rand_pval_for_incidence()
```

`InferenceIncidKKGCompRiskDiff$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidKKGCompRiskDiff$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Robins, J. (1986). A new approach to causal inference in mortality studies with a sustained exposure period. *Mathematical Modelling*, 7(9-12), 1393-1512. doi:10.1016/02700255(86)900886

See Also

`InferenceIncidKKGCompRiskRatio` for the risk-ratio analog on the same standardization machinery.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidKKGCompRiskDiff$new(seq_des)
inf$compute_estimate()
```

InferenceIncidKKGCompRiskRatio

G-Computation Risk-Ratio Inference for KK Designs with Binary Responses

---

Description

Fits the same all-subject logistic working model as `InferenceIncidKKGCompRiskDiff` for a KK incidence outcome using treatment and, optionally, all recorded covariates, then estimates the marginal (standardized, g-computation) risk ratio $\hat{\theta} = \left(n^{-1} \sum_i \hat{p}(1, X_i)\right) / \left(n^{-1} \sum_i \hat{p}(0, X_i)\right)$ by averaging fitted-model predicted risks under all-treated and all-control assignments over the empirical covariate distribution (Robins 1986). Matched pairs are treated as clusters and reservoir subjects are treated as singletons when computing the sandwich covariance; the delta method is applied on the log-risk-ratio scale to keep the reported ratio and its confidence interval positive, then back-transformed for reporting.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Details

This estimator has `likelihood_tier = "none"`. If the working model fails to converge, has no treatment-arm variation, or the all-control standardized risk is zero (undefined ratio), the estimate is marked non-estimable via `is_nonestimable()`.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Super class

Inference -> InferenceIncidKKGCompRiskRatio

Methods

Public methods:

- InferenceIncidKKGCompRiskRatio$approximate_randomization_distribution_beta_hat_T()
- InferenceIncidKKGCompRiskRatio$supports_rand_pval_for_incidence()
- InferenceIncidKKGCompRiskRatio$clone()

InferenceIncidKKGCompRiskRatio$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceIncidKKGCompRiskRatio$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`delta` The null difference. Default 0.

`transform_responses` Type of transformation. Default "none".

`show_progress` Show progress bar. Default TRUE.

`permutations` Pre-computed permutations. Default NULL.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

InferenceIncidKKGCompRiskRatio$supports_rand_pval_for_incidence():

Usage:

```
InferenceIncidKKGCompRiskRatio$supports_rand_pval_for_incidence()
```

InferenceIncidKKGCompRiskRatio$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidKKGCompRiskRatio$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Robins, J. (1986). A new approach to causal inference in mortality studies with a sustained exposure period. *Mathematical Modelling*, 7(9-12), 1393-1512. doi:10.1016/02700255(86)900886

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidKKGCompRiskRatio$new(seq_des)
inf$compute_estimate()
```

InferenceIncidKKGEE *GEE Inference for KK Designs with Binary Response*

Description

Fits a Generalized Estimating Equations (GEE) model with a binomial family and logit link, $\text{logit}\ \Pr(Y_i = 1 \mid x_i) = x_i^\top \beta$, for binary (incidence) responses under a KK matching-on-the-fly design, using an **exchangeable** working correlation structure where each cluster is either a matched pair (2 members) or a reservoir singleton (1 member) — see `$compute_estimate()`'s method-level documentation for the full fitting contract (internal Rcpp solver vs. **geepack** fallback, hardening/retry behavior). GEE is used here purely to fit one marginal model jointly across matched-pair and reservoir subjects while accounting for the within-pair correlation the matching induces, not as a longitudinal/repeated-measures tool. Inference is quasi-likelihood/estimating-equation based (`likelihood_tier = "quasi"`): standard errors are GEE sandwich (robust) standard errors, not model-likelihood-based.

Super class

`Inference` -> `InferenceIncidKKGEE`

Methods

Public methods:

- `InferenceIncidKKGEE$new()`
- `InferenceIncidKKGEE$clone()`

`InferenceIncidKKGEE$new()`**:** Initialize KK binary-response GEE inference, validate the matched/reservoir design, and prepare the exchangeable-working-correlation binomial (logit-link) GEE fitting machinery used by `InferenceIncidKKGEE`.

Usage:

```
InferenceIncidKKGEE$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with an incidence response.

`model_formula` Optional formula for covariate adjustment.

`use_rcpp` Whether to use the internal Rcpp GEE solver (`TRUE`, default) with automatic fallback to `geepack::geeglm` on failure, or always use `geepack::geeglm` directly (`FALSE`).

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`InferenceIncidKKGEE$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidKKGEE$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Liang, K.-Y., and Zeger, S. L. (1986). "Longitudinal Data Analysis Using Generalized Linear Models." *Biometrika*, 73(1), 13-22, doi:10.1093/biomet/73.1.13, for the GEE estimating-equation framework and sandwich variance estimator used here.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidKKGEE$new(seq_des)
inf$compute_estimate()
```

InferenceIncidKKModifiedPoisson

Modified-Poisson Inference for KK Designs with Binary Responses

---

Description

Fits Zou's (2004) modified-Poisson working model for binary incidence outcomes under a KK matching-on-the-fly design: a log-link Poisson model $\log E[Y_i \mid w_i, x_i] = \beta_0 + \beta_T w_i + x_i^\top \gamma$ is fit to the binary (0/1) response by ordinary Poisson maximum likelihood (a working, misspecified likelihood — the true response is Bernoulli, not Poisson), and the coefficient standard errors are corrected by a cluster-robust sandwich covariance rather than the (invalid, for a misspecified likelihood) model-based Poisson information. Matched pairs are treated as clusters (2 members) and reservoir subjects as singleton clusters when computing the sandwich covariance, so the matched-pair correlation induced by the design is accounted for even though the modified-Poisson working model itself does not encode it directly. $\exp(\hat{\beta}_T)$ is the estimated risk ratio, directly interpretable unlike a logistic regression's odds ratio (which only approximates the risk ratio when the outcome is rare). `likelihood_tier = "none"` (the sandwich-corrected inference is not a normalized model likelihood): only Wald inference is exposed. See `InferenceAbstractKKMarginalIncid` for the shared marginal-incidence fitting contract.

Super classes

`Inference` -> `InferenceAbstractKKMarginalIncid` -> `InferenceAbstractKKModifiedPoisson` -> `InferenceIncidKKModifiedPoisson`

Methods

Public methods:

- `InferenceIncidKKModifiedPoisson$clone()`

`InferenceIncidKKModifiedPoisson$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidKKModifiedPoisson$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Zou, G. (2004). "A Modified Poisson Regression Approach to Prospective Studies with Binary Data." *American Journal of Epidemiology*, 159(7), 702-706, doi:10.1093/aje/kwh090; Kapelner, A. and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388, doi:10.1111/biom.12148, for the KK matching-on-the-fly design this class is built for.

See Also

`InferenceIncidModifiedPoisson` for the non-KK analog.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidKKModifiedPoisson$new(seq_des)
inf$compute_estimate()
```

InferenceIncidKKNewcombeRiskDiff

KK Newcombe Risk-Difference IVWC Inference for Binary Responses

---

Description

Initialize KK Newcombe risk-difference IVWC inference and prepare matched/reservoir paired-binomial components used by `InferenceIncidKKNewcombeRiskDiff`.

Computes the compound Newcombe risk-difference point estimate $\hat{\theta} = w_1\hat{\theta}_1 + w_2\hat{\theta}_2$ on the risk-difference (probability) scale, where $\hat{\theta}_1$ is the paired-Newcombe discordant-pair estimate from matched pairs and $\hat{\theta}_2$ is the independent-Newcombe estimate from reservoir subjects, combined by inverse-variance weighting $w_j \propto 1/\widehat{\mathrm{Var}}(\hat{\theta}_j)$ (falls back to the single available component when one has zero subjects). Caches intermediate match/reservoir statistics for reuse by `compute_asymp_confidence_interval()` and `compute_asymp_two_sided_pval()`.

Usage

```
KKNewcombeRiskDiffIVWCSource
```

Details

Implements a compound Newcombe risk-difference estimator for KK designs. This class pools information from matched pairs (using the Paired Newcombe method) and the reservoir (using the Independent Newcombe method) via inverse-variance weighted combination (IVWC).

The matched-pair component applies the paired Newcombe (Method-10-style Wilson-score) interval to the discordant pairs to estimate the treatment effect and its variance (Newcombe 1998). The reservoir component applies the independent-samples Newcombe interval, treating unmatched subjects as two independent binomial samples. The two component estimates $\hat{\theta}_1, \hat{\theta}_2$ are combined by inverse-variance weighting, $\hat{\theta} = w_1\hat{\theta}_1 + w_2\hat{\theta}_2$, $w_j = (1/\hat{V}_j)/\sum_k(1/\hat{V}_k)$, the standard IVWC framework used throughout the package's KK inference classes. `likelihood_tier = "none"`: this is a closed-form Wilson-score-type estimator, not a fitted likelihood model, so no likelihood-ratio or parametric-bootstrap methods are exposed. If a design has no matched pairs or no reservoir subjects, the single available component is used directly rather than combined.

References

Newcombe, R. G. (1998). Interval Estimation for the Difference Between Independent Proportions: Comparison of Eleven Methods. *Statistics in Medicine*, 17(8), 873-890. doi:10.1002/(SICI)1097-0258(19980430)17:8<873::AIDSIM779>3.0.CO;2I

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidKKNewcombeRiskDiff$new(seq_des)
inf$compute_estimate()
```

InferenceIncidLogBinomial

Log-Binomial Regression Inference for Incidence Responses

Description

Fits a binomial regression with the **log** link for binary (incidence) responses: $\log P(Y_i = 1) = \beta_0 + \beta_T W_i + X_i^\top \gamma$, where W_i is the treatment indicator and X_i are optional recorded covariates, by maximum likelihood (`fast_log_binomial_regression_cpp`/ `fast_log_binomial_regression_weighted_cpp`). $\hat{\beta}_T$ is a **log risk ratio**: $\exp(\hat{\beta}_T)$ is the estimated treatment risk ratio (relative risk) directly, unlike the log-odds-ratio from `InferenceIncidLogRegr`'s logit link. `likelihood_tier = "full"`: Wald, score, gradient, and likelihood-ratio tests are all available when the model converges, plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test. Because the log link does not constrain fitted probabilities to $[0, 1]$ (only to $[0, \infty)$), fits are hardened by QR column-dropping and a coefficient-magnitude cap (`max_abs_reasonable_coef`) and rejected as nonestimable when the fit is implausible — the same practical limitation as the identity-link sibling `InferenceIncidBinomialIdentityRiskDif` here applying to the upper rather than both tails of the probability scale. Validity requires the multiplicative log-linear risk model to be correctly specified over the covariate range observed.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Super class

Inference -> InferenceIncidLogBinomial

Methods

Public methods:

- InferenceIncidLogBinomial$approximate_randomization_distribution_beta_hat_T()
- InferenceIncidLogBinomial$supports_rand_pval_for_incidence()
- InferenceIncidLogBinomial$clone()

InferenceIncidLogBinomial$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceIncidLogBinomial$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

r Number of randomization vectors. Default 501.

delta The null difference. Default 0.

transform_responses Type of transformation. Default "none".

show_progress Show progress bar. Default TRUE.

permutations Pre-computed permutations. Default NULL.

debug If TRUE, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default FALSE.

zero_one_logit_clamp The clamping amount for exact 0 and 1 values when logging

InferenceIncidLogBinomial$supports_rand_pval_for_incidence():

Usage:

```
InferenceIncidLogBinomial$supports_rand_pval_for_incidence()
```

InferenceIncidLogBinomial$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidLogBinomial$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

McCullagh, P., and Nelder, J. A. (1989). *Generalized Linear Models* (2nd ed.). Chapman and Hall/CRC, for the binomial GLM family and log-link relative-risk parameterization.

See Also

`InferenceIncidLogRegr` (logit link, log-odds-ratio estimand), `InferenceIncidBinomialIdentityRiskDiff` (identity link, risk-difference estimand) for alternative link/estimand choices on the same response type. Comparable Python API: statsmodels GLM (`family=Binomial(link=log())`). See also: Generalized linear model (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidLogBinomial$new(seq_des)
inf$compute_estimate()
```

`InferenceIncidLogRegr` *Logistic Regression Inference for Incidence Responses*

Description

Fits a logistic regression model for binary (incidence) responses: $\text{logit}(P(Y_i = 1)) = \beta_0 + \beta_T W_i + X_i^\top \gamma$, where W_i is the treatment indicator and X_i are optional recorded covariates, by maximum likelihood (`fast_logistic_regression_cpp`/ `fast_logistic_regression_weighted_cpp`). $\hat{\beta}_T$ is a log-odds-ratio: $\exp(\hat{\beta}_T)$ is the estimated treatment odds ratio. `likelihood_tier = "full"`: Wald, score, gradient, and likelihood-ratio tests are all available (via the shared `StandardModelCache` model-caching contract), plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test. A fit whose coefficients exceed `max_abs_reasonable_coef` in magnitude (a proxy for near-perfect separation) is cached as nonestimable rather than returned. Validity requires the usual logistic-regression assumptions: correctly specified linear predictor on the logit scale, independence across subjects conditional on covariates, and no perfect/quasi-complete separation.

Estimand. Composes `MarginalEstimand` (`set_estimand()`/`get_estimand()`/`get_supported_estimands()`). Under the default `estimand = "conditional"`, $\hat{\beta}_T$ is the log-odds-ratio above. Under `estimand = "marginal_mean_diff"`, the reported quantity is instead the g-computation marginal risk difference $\frac{1}{n}\sum_i\{\text{plogis}(\hat{\beta}_0 + \hat{\beta}_T + X_i^\top\hat{\gamma}) - \text{plogis}(\hat{\beta}_0 + X_i^\top\hat{\gamma})\}$ — every subject's covariates plugged

in once under treatment and once under control, averaged over the empirical covariate distribution. Under `estimand = "marginal_ratio"`, the log of the corresponding marginal risk ratio. Because there is no latent submodel for this family (unlike e.g. `InferencePropZeroOneInflatedBetaRegr`'s zero/one-inflation mixture), the marginal mean function is exactly the model's own fitted mean; no separate standardization step beyond the g-computation average is needed. Standard errors under a marginal estimand use the delta method against the model's coefficient covariance (degrees of freedom `Inf`); `testing_type` is restricted to `"wald"` whenever the estimand is non-conditional (`set_testing_type()` errors otherwise). The underlying model fit is identical regardless of estimand — switching `estimand` is a pure post-fit transform, never a refit.

Super class

`Inference` -> `InferenceIncidLogRegr`

Methods

Public methods:

- `InferenceIncidLogRegr$compute_estimate()`
- `InferenceIncidLogRegr$compute_asymp_confidence_interval()`
- `InferenceIncidLogRegr$compute_asymp_two_sided_pval()`
- `InferenceIncidLogRegr$clone()`

`InferenceIncidLogRegr$compute_estimate():` Fits the logistic regression model by maximum likelihood. Under the default `estimand = "conditional"`, returns $\hat{\beta}_T$, the treatment log-odds-ratio. Under `estimand = "marginal_mean_diff"/"marginal_ratio"` (set via `set_estimand()`), returns the g-computation marginal risk difference/log-risk-ratio instead — see the class-level `@details` for the formula. The underlying model fit is identical either way (a pure post-fit transform of the same cached fit, no refit).

Usage:

```
InferenceIncidLogRegr$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip standard-error computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferenceIncidLogRegr$compute_asymp_confidence_interval():` Wald confidence interval, dispatched by `testing_type` for the conditional estimand (score/gradient/ likelihood-ratio/Bartlett available); under a marginal estimand `testing_type` is always `"wald"` (the only value `set_estimand()` permits there), so this always resolves to the delta-method interval. Calls `self$compute_estimate()` first (not `private$shared()` directly) so the estimand-aware cache is always current regardless of call order.

Usage:

```
InferenceIncidLogRegr$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Two-sided miscoverage rate; the returned interval targets `1 - alpha` coverage.

`InferenceIncidLogRegr$compute_asymp_two_sided_pval():` Wald two-sided p-value, dispatched by `testing_type` exactly as `compute_asymp_confidence_interval()`; see that method's description for the marginal-estimand always-Wald note.

Usage:

```
InferenceIncidLogRegr$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment-effect value under the current estimand (conditional log-odds-ratio, or marginal risk difference/ log-risk-ratio).

`InferenceIncidLogRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidLogRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

McCullagh, P., and Nelder, J. A. (1989). *Generalized Linear Models* (2nd ed.). Chapman and Hall/CRC, for the logistic regression model and its maximum-likelihood theory.

See Also

Comparable Python API: statsmodels GLM. See also: Logistic regression (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidLogRegr$new(seq_des)
inf$compute_estimate()


inf$set_seed(1)
inf$compute_lik_ratio_bootstrap_two_sided_pval(delta = 0, B = 9, show_progress = FALSE)
```

InferenceIncidMiettinenNurminenRiskDiff

Miettinen-Nurminen Risk-Difference Inference for Binary Responses

---

Description

Fits the classical Miettinen-Nurminen score method for the risk difference in a two-arm binary trial. The point estimate is the observed risk difference, while confidence intervals and p-values are obtained by inverting the constrained score test under the null $p_T - p_C = \delta$.

This class is intentionally unadjusted. It operates on the 2×2 table induced by treatment assignment and incidence response (counts x_T, x_C of events among n_T, n_C treated/control subjects), and is therefore the natural classical binary-endpoint complement to the regression-based incidence methods already in the package. The point estimate is the plain risk difference $\hat{p}_T - \hat{p}_C$; the Miettinen-Nurminen confidence interval and p-value invert a **restricted-maximum-likelihood score test** of $H_0 : p_T - p_C = \delta$ (via `mn_ci_cpp`/`mn_pvalue_cpp`) — under the null, the two arms' event probabilities are re-estimated subject to the constraint $\hat{p}_T - \hat{p}_C = \delta$, and the resulting score statistic is compared to its asymptotic normal distribution, with a small-sample bias correction factor $(n_T + n_C)/(n_T + n_C - 1)$ applied to the naive Wald variance used elsewhere (e.g. in `$compute_estimate_with_bootstrap_weights()`, which never gets this correction since it skips the score-test path entirely). This score-based interval generally has better small-sample coverage than the naive normal-approximation Wald interval on the risk difference.

Super class

`Inference` -> `InferenceIncidMiettinenNurminenRiskDiff`

Methods

Public methods:

- `InferenceIncidMiettinenNurminenRiskDiff$new()`
- `InferenceIncidMiettinenNurminenRiskDiff$compute_estimate()`
- `InferenceIncidMiettinenNurminenRiskDiff$compute_estimate_with_bootstrap_weights()`
- `InferenceIncidMiettinenNurminenRiskDiff$compute_asymp_confidence_interval()`
- `InferenceIncidMiettinenNurminenRiskDiff$compute_asymp_two_sided_pval()`
- `InferenceIncidMiettinenNurminenRiskDiff$clone()`

`InferenceIncidMiettinenNurminenRiskDiff$new()`: Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize a Miettinen-Nurminen risk-difference inference object for a completed design with an incidence response.

Usage:

```
InferenceIncidMiettinenNurminenRiskDiff$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE
)
```

Arguments:

`des_obj` A completed `DesignSeqOneByOne` object with an incidence response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

Examples:

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 20, response_type = "incidence")
for (i in 1:20) {
 x_i = data.frame(x1 = rnorm(1), x2 = rnorm(1))
 w_i = seq_des$add_one_subject_to_experiment_and_assign(x_i)
 p_i = plogis(-0.8 + 0.5 * w_i)
 seq_des$add_one_subject_response(i, rbinom(1, 1, p_i))
}
seq_des_inf = InferenceIncidMiettinenNurminenRiskDiff$new(seq_des)
seq_des_inf$compute_estimate()
```

`InferenceIncidMiettinenNurminenRiskDiff$compute_estimate()`: Computes the observed (unadjusted) risk-difference estimate $\hat{p}_T - \hat{p}_C$ (see class documentation for the full Miettinen-Nurminen inference model). `NA` if either arm is empty.

Usage:

```
InferenceIncidMiettinenNurminenRiskDiff$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`InferenceIncidMiettinenNurminenRiskDiff$compute_estimate_with_bootstrap_weights()`: Recomputes the risk-difference estimate under subject/block bootstrap weights: the weighted event proportions $\hat{p}_T^w = \sum_i r_i y_i \mathbb{1}[w_i = 1] / \sum_i r_i \mathbb{1}[w_i = 1]$ (and analogously for control), differenced. Used by the Bayesian bootstrap and related weighted-resampling machinery; see `InferenceBayesianBootstrap`. Always leaves the standard error and degrees of freedom unavailable (`NA`) regardless of `estimate_only` — this weighted path never computes the Miettinen-Nurminen score-based variance.

Usage:

```
InferenceIncidMiettinenNurminenRiskDiff$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` Present for interface parity; this method never computes variance components regardless of its value.

`InferenceIncidMiettinenNurminenRiskDiff$compute_asymp_confidence_interval()`: Computes a $1-\alpha$ Miettinen-Nurminen restricted-MLE score confidence interval for the risk difference (see class documentation for the full method), by bisection-inverting the score test (`mn_ci_cpp`) to `pval_epsilon` tolerance.

Usage:

```
InferenceIncidMiettinenNurminenRiskDiff$compute_asymp_confidence_interval(
  alpha = 0.05,
  pval_epsilon = 1e-07
)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`pval_epsilon` Bisection tolerance for CI bounds.

`InferenceIncidMiettinenNurminenRiskDiff$compute_asymp_two_sided_pval()`: Computes a two-sided Miettinen-Nurminen restricted-MLE score p-value (`mn_pvalue_cpp`) testing $H_0 : p_T - p_C =$ `delta` (see class documentation for the full method).

Usage:

```
InferenceIncidMiettinenNurminenRiskDiff$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment effect on the risk-difference scale.

`InferenceIncidMiettinenNurminenRiskDiff$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidMiettinenNurminenRiskDiff$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Miettinen, O., and Nurminen, M. (1985). "Comparative Analysis of Two Rates." *Statistics in Medicine*, 4(2), 213-226, doi:10.1002/sim.4780040211, for the restricted-maximum-likelihood score method used here.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidMiettinenNurminenRiskDiff$new(seq_des)
inf$compute_estimate()


## ------------------------------------------------
## Method `InferenceIncidMiettinenNurminenRiskDiff$new()`
## ------------------------------------------------

seq_des = DesignSeqOneByOneBernoulli$new(n = 20, response_type = "incidence")
for (i in 1:20) {
 x_i = data.frame(x1 = rnorm(1), x2 = rnorm(1))
 w_i = seq_des$add_one_subject_to_experiment_and_assign(x_i)
 p_i = plogis(-0.8 + 0.5 * w_i)
 seq_des$add_one_subject_response(i, rbinom(1, 1, p_i))
}
seq_des_inf = InferenceIncidMiettinenNurminenRiskDiff$new(seq_des)
seq_des_inf$compute_estimate()
```

InferenceIncidModifiedPoisson

Modified Poisson Regression Inference for Incidence Responses

Description

Fits Zou's (2004) modified Poisson regression for binary (incidence) responses: $\log E[Y_i \mid w_i, x_i] = \beta_0 + \beta_T w_i + x_i^\top \gamma$, fit by maximizing the ordinary Poisson log-likelihood treating the binary Y_i as if it were Poisson-distributed (a valid estimating equation for the conditional mean regardless of the true outcome distribution, exactly as `InferencePropFractionalLogit`'s quasi-binomial fit is for fractional responses). $\hat{\beta}_T$ is a **log risk ratio**: $\exp(\hat{\beta}_T)$ is the estimated treatment relative risk, the same estimand as `InferenceIncidLogBinomial`'s log-binomial model, but modified Poisson never produces a fit failure from the $[0, 1]$-probability constraint that a genuine binomial log-link model can hit. Standard errors use Zou's (2004) robust/sandwich correction, not the ordinary (model-based) Poisson Fisher information: the misspecified Poisson working model assumes $\mathrm{Var}(Y_i) = \mu_i$, but Y_i is actually Bernoulli with $\mathrm{Var}(Y_i) = p_i(1-p_i) \leq \mu_i$, so the naive Fisher-information SE materially overstates the true sampling variance. The sandwich correction is computed in R from quantities `fast_poisson_regression_with_var_cpp` already returns (`mu` and the unrestricted Fisher information `fisher_information`), not a change to the underlying kernel. **Changed 2026-10:** earlier versions of this class used the naive Fisher-information SE directly, which measured at roughly 0.26 Type-I error against a nominal 5 Bernoulli design, `model_formula = ~.`, asymptotic Wald test); every standard error, confidence interval, and p-value this class has ever returned differs from this version's. `likelihood_tier = "full"` metadata is set for component-composition purposes, but `private$supports_likelihood_tests()` is hard `FALSE` — only Wald inference is exposed (`get_supported_testing_types_impl()` returns `"wald"` only), not likelihood-ratio/score/gradient tests. Fits with implausible coefficients or fitted linear predictors (checked via `private$is_modified_poisson_fit_reason` capped by `max_abs_reasonable_coef`/`max_abs_reasonable_linear_predictor`) are cached as nonestimable rather than returned.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Super class

Inference -> InferenceIncidModifiedPoisson

Methods

Public methods:

- InferenceIncidModifiedPoisson$approximate_randomization_distribution_beta_hat_T()
- InferenceIncidModifiedPoisson$supports_rand_pval_for_incidence()
- InferenceIncidModifiedPoisson$clone()

InferenceIncidModifiedPoisson$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceIncidModifiedPoisson$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`delta` The null difference. Default 0.

`transform_responses` Type of transformation. Default "none".

`show_progress` Show progress bar. Default TRUE.

`permutations` Pre-computed permutations. Default NULL.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

InferenceIncidModifiedPoisson$supports_rand_pval_for_incidence():

Usage:

```
InferenceIncidModifiedPoisson$supports_rand_pval_for_incidence()
```

InferenceIncidModifiedPoisson$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidModifiedPoisson$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Zou, G. (2004). "A Modified Poisson Regression Approach to Prospective Studies with Binary Data." *American Journal of Epidemiology*, 159(7), 702-706, doi:10.1093/aje/kwh090.

See Also

`InferenceIncidLogBinomial` for the genuine log-binomial alternative with the same log-risk-ratio estimand. Comparable Python API: statsmodels discrete models (`Poisson` family on binary data). See also: Poisson regression (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidModifiedPoisson$new(seq_des)
inf$compute_estimate()
```

InferenceIncidNewcombeRiskDiff

Newcombe Risk-Difference Inference for Binary Responses

Description

Fits the Newcombe hybrid score method (Method 10) for the risk difference in a two-arm binary trial. This method constructs a confidence interval for the difference between two independent proportions by combining Wilson score intervals for each group.

This class is unadjusted and assumes independent samples (e.g. from a Bernoulli design). It ignores any matched-pair structure if present; for matched data, use `InferenceIncidKKNewcombeRiskDiff`. The point estimate is the plain risk difference $\hat{p}_T - \hat{p}_C$. The confidence interval (`newcombe_independent_ci_cpp`) is Newcombe's "Method 10" hybrid score interval: separate Wilson score intervals $[\ell_T, u_T]$ and $[\ell_C, u_C]$ are computed for each arm's proportion individually, then combined into a difference interval via $[\hat{p}_T - \hat{p}_C - \sqrt{(\hat{p}_T - \ell_T)^2 + (u_C - \hat{p}_C)^2},\ \hat{p}_T - \hat{p}_C + \sqrt{(u_T - \hat{p}_T)^2 + (\hat{p}_C - \ell_C)^2}]$ — this avoids the boundary/coverage problems of the naive Wald interval on a risk difference while remaining closed-form (no iterative score-test inversion, unlike the Miettinen-Nurminen method in `InferenceIncidMiettinenNurminenRiskDiff`). The two-sided p-value has no closed form here: it is obtained by numerically inverting the confidence interval (bisection via `stats::uniroot`) to find the significance level at which `delta` falls exactly on the interval boundary.

Super class

`Inference` -> `InferenceIncidNewcombeRiskDiff`

Methods

Public methods:

- `InferenceIncidNewcombeRiskDiff$new()`
- `InferenceIncidNewcombeRiskDiff$compute_estimate()`
- `InferenceIncidNewcombeRiskDiff$compute_estimate_with_bootstrap_weights()`
- `InferenceIncidNewcombeRiskDiff$compute_asymp_confidence_interval()`
- `InferenceIncidNewcombeRiskDiff$compute_asymp_two_sided_pval()`
- `InferenceIncidNewcombeRiskDiff$clone()`

`InferenceIncidNewcombeRiskDiff$new()`: Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize a Newcombe risk-difference inference object for a completed design with an uncensored incidence response.

Usage:

```
InferenceIncidNewcombeRiskDiff$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE
)
```

Arguments:

`des_obj` A completed `DesignSeqOneByOne` object with an incidence response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`InferenceIncidNewcombeRiskDiff$compute_estimate()`: Computes the observed (unadjusted) risk-difference estimate $\hat{p}_T - \hat{p}_C$ (see class documentation for the full Newcombe interval method).

Usage:

```
InferenceIncidNewcombeRiskDiff$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`InferenceIncidNewcombeRiskDiff$compute_estimate_with_bootstrap_weights()`: Recomputes the risk-difference estimate under subject/block bootstrap weights: the weighted event proportions $\hat{p}_T^w = \sum_i r_i y_i \mathbb{1}[w_i = 1] / \sum_i r_i \mathbb{1}[w_i = 1]$ (and analogously for control), differenced. Used by the Bayesian bootstrap and related weighted-resampling machinery; see `InferenceBayesianBootstrap`. Always leaves the standard error and degrees of freedom unavailable (`NA`) regardless of `estimate_only` — the Newcombe interval method has no separate variance quantity to compute on this path.

Usage:

```
InferenceIncidNewcombeRiskDiff$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` Present for interface parity; this method never computes variance components regardless of its value.

`InferenceIncidNewcombeRiskDiff$compute_asymp_confidence_interval()`: Computes a $1-\alpha$ Newcombe hybrid Wilson-score confidence interval for the risk difference (see class documentation for the full formula), via `newcombe_independent_ci_cpp`.

Usage:

```
InferenceIncidNewcombeRiskDiff$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The significance level.

`InferenceIncidNewcombeRiskDiff$compute_asymp_two_sided_pval()`: Computes a two-sided p-value testing $H_0 : p_T - p_C =$ `delta` by numerically finding (`stats::uniroot`) the significance level α at which `delta` falls exactly on the boundary of the Newcombe confidence interval (see class documentation) — there is no closed-form p-value for this method. The search is carried out on $\log(\alpha)$ so that very small p-values are resolved accurately. If the null is already outside the interval at the numerical α floor, that floor is returned; if it remains inside as α approaches one, the p-value is 1.

Usage:

```
InferenceIncidNewcombeRiskDiff$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null risk difference.

`InferenceIncidNewcombeRiskDiff$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidNewcombeRiskDiff$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Newcombe, R. G. (1998). "Interval Estimation for the Difference Between Independent Proportions: Comparison of Eleven Methods." *Statistics in Medicine*, 17(8), 873-890, doi:10.1002/(SICI)10970258(19980430)17:8<873::AIDSIM779>3.0.CO;2I, for "Method 10", the hybrid Wilson-score interval used here.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidNewcombeRiskDiff$new(seq_des)
inf$compute_estimate()
```

InferenceIncidProbitRegr

Probit Regression Inference for Incidence Responses

Description

Fits a probit regression model for binary (incidence) responses: $\Phi^{-1}(P(Y_i = 1)) = \beta_0 + \beta_T W_i + X_i^\top \gamma$, where Φ is the standard normal CDF, W_i is the treatment indicator, and X_i are optional recorded covariates, by maximum likelihood (fast_probit_regression_cpp/ fast_probit_regression_weighted_cpp). Unlike InferenceIncidLogRegr's logit link, $\hat{\beta}_T$ here is not an odds-ratio scale parameter: it is the treatment's additive effect on the latent standard-normal index underlying the binary outcome. likelihood_tier = "full": Wald, score, gradient, and likelihood-ratio tests are all available when the model converges, plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test. A fit whose coefficients exceed max_abs_reasonable_coef in magnitude (a proxy for near-perfect separation) is cached as nonestimable rather than returned. Validity requires the usual probit assumptions: correctly specified linear predictor on the latent-normal scale, independence across subjects conditional on covariates, and no perfect/quasi-complete separation.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether compute_rand_two_sided_pval() is actually usable on this instance right now – FALSE exactly when it would stop(): an incidence-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see private$should_use_design_ra TRUE for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so InferenceSuite can check this before attempting the sentinel instead of relying on the stop() being silently swallowed into a pval = NA "ok" row – the single source of truth for both this check and compute_rand_two_sided_pval()'s own guard, so the two can never drift apart.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Super class

Inference -> InferenceIncidProbitRegr

Methods

Public methods:

- InferenceIncidProbitRegr$approximate_randomization_distribution_beta_hat_T()
- InferenceIncidProbitRegr$supports_rand_pval_for_incidence()
- InferenceIncidProbitRegr$clone()

`InferenceIncidProbitRegr$approximate_randomization_distribution_beta_hat_T():`

Usage:

```
InferenceIncidProbitRegr$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`delta` The null difference. Default 0.

`transform_responses` Type of transformation. Default "none".

`show_progress` Show progress bar. Default TRUE.

`permutations` Pre-computed permutations. Default NULL.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceIncidProbitRegr$supports_rand_pval_for_incidence():`

Usage:

```
InferenceIncidProbitRegr$supports_rand_pval_for_incidence()
```

`InferenceIncidProbitRegr$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidProbitRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

McCullagh, P., and Nelder, J. A. (1989). *Generalized Linear Models* (2nd ed.). Chapman and Hall/CRC, for the binomial GLM family and probit link.

See Also

`InferenceIncidLogRegr` for the logit-link alternative with a log-odds-ratio estimand. Comparable Python API: statsmodels GLM (`family=Binomial(link=probit())`). See also: Probit model (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidProbitRegr$new(seq_des)
inf$compute_estimate()
```

InferenceIncidRiskDiff

Risk Difference Inference for Incidence Responses

Description

Fits a linear probability model $E[Y \mid w, x] = \beta_0 + \beta_T w + \beta_X^\top x$ via ordinary least squares for binary (incidence) responses $Y \in \{0, 1\}$, using the treatment indicator w and, optionally, all recorded covariates x as predictors. $\hat{\beta}_T$ is reported directly as the risk-difference estimate: because Y is 0/1, the OLS fit coincides with a saturated/linear model for the conditional risk $P(Y = 1 \mid w, x)$, so the coefficient on w is already on the risk-difference (probability) scale with no back-transformation needed. This is a misspecified working model for a binary response (heteroskedastic, errors not Gaussian: $\mathrm{Var}(Y \mid w, x) = P(w, x)(1 - P(w, x))$ varies by treatment arm and covariates, so a single pooled residual variance is the wrong variance model), so `likelihood_tier = "none"`: standard errors and the Wald CI use the **Huber-White (HC0) sandwich** variance of $\hat{\beta}_T$ – $(X'X)^{-1} X' \,\mathrm{diag}(e_i^2)\, X\, (X'X)^{-1}$, from the OLS residuals e_i – not the classical homoskedastic OLS variance and not a binomial likelihood, and no likelihood-ratio or parametric-bootstrap methods are exposed.

Super class

`Inference` -> `InferenceIncidRiskDiff`

Methods

Public methods:

- `InferenceIncidRiskDiff$new()`
- `InferenceIncidRiskDiff$compute_estimate()`
- `InferenceIncidRiskDiff$compute_asymp_confidence_interval()`
- `InferenceIncidRiskDiff$compute_asymp_two_sided_pval()`
- `InferenceIncidRiskDiff$compute_estimate_with_bootstrap_weights()`
- `InferenceIncidRiskDiff$clone()`

`InferenceIncidRiskDiff$new()`: Uses the randomization-CI layer's two-sided p-value contract (`InferenceRandCI`'s version, not `InferenceRand`'s): for incidence responses it dispatches

to the Zhang exact randomization test rather than refusing outright, matching this class's pre-migration old-ladder behavior. This deliberately differs from the `InferenceAllSimpleAverageDiff`-family precedent of pinning `InferenceRand`'s version, which would have regressed the working Zhang dispatch this class had on the old ladder.

Initialize a risk-difference inference object.

Usage:

```
InferenceIncidRiskDiff$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with an incidence response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`r` Number of randomization vectors. @param delta Null difference.

`transform_responses` Transformation. @param na.rm Remove NAs.

`show_progress` Show progress. @param permutations Pre-computed permutations.

`type` Optional exact-inference type for incidence dispatch.

`args_for_type` Optional arguments for `type`.

`zero_one_logit_clamp` Clamp for exact 0/1 values when logging.

`InferenceIncidRiskDiff$compute_estimate()`**:** Fits the OLS linear-probability model and returns the risk-difference point estimate $\hat{\beta}_T$, the coefficient on treatment. On a hardened design (`private$harden`), or when `estimate_only = FALSE`, delegates to `fast_ols_with_var_cpp()` via the shared model cache so the variance is available for later confidence-interval/p-value calls without refitting.

Usage:

```
InferenceIncidRiskDiff$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If `TRUE`, skip the variance/degrees-of-freedom computation and return only $\hat{\beta}_T$ (cheaper for simulation/randomization callers that never request inference).

`InferenceIncidRiskDiff$compute_asymp_confidence_interval()`**:** Wald confidence interval for the risk difference, $\hat{\beta}_T \pm z_{1-\alpha/2}\, \hat{s}(\hat{\beta}_T)$, using the **Huber-White (HC0) sandwich** standard error from the cached model fit (see the class-level documentation) and a normal-quantile multiplier – `fast_ols_with_var_cpp()` doesn't report residual degrees of freedom, so `private$compute_z_or_t_ci_fr` always falls back to its z branch here, never a t reference, regardless of n. Interval bounds are not clamped to $[-1, 1]$; a linear-probability model can produce out-of-range endpoints near the boundary of the covariate space.

Usage:

`InferenceIncidRiskDiff$compute_asymp_confidence_interval(alpha = 0.05)`

Arguments:

`alpha` Two-sided miscoverage rate; the returned interval has nominal coverage $1 - \alpha$.

`InferenceIncidRiskDiff$compute_asymp_two_sided_pval():` Two-sided Wald test of $H_0 : \beta_T = \delta$ vs. $H_1 : \beta_T \neq \delta$, via the Wald statistic $(\hat{\beta}_T - \delta)/\hat{s}(\hat{\beta}_T)$ ($\hat{s}$ the Huber-White sandwich SE) referred to a standard normal distribution – same z, not t, reference as `$compute_asymp_confidence_interval()`, for the same reason (no residual degrees of freedom are ever cached for this class).

Usage:

`InferenceIncidRiskDiff$compute_asymp_two_sided_pval(delta = 0)`

Arguments:

`delta` Null risk-difference value under H_0 (default `0`, no treatment effect).

`InferenceIncidRiskDiff$compute_estimate_with_bootstrap_weights():` Refits the linear-probability model by weighted least squares (`stats::lm.wfit()`) under subject/block resampling weights (nonparametric-bootstrap replicate weights or Bayesian-bootstrap Dirichlet weights, both expanded to row weights via `expand_subject_or_block_weights_to_row_weights()`) and returns the re-estimated treatment coefficient. Rows with non-finite or non-positive weight are excluded; if too few positive-weight rows remain to identify the design, the replicate estimate is `NA_real_`.

Usage:

```
InferenceIncidRiskDiff$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Numeric weights, one per subject or per resampling block (matched pair/cluster), as produced by the bootstrap or Bayesian-bootstrap resampling machinery.

`estimate_only` Accepted for interface compatibility; standard errors are never computed for a single bootstrap replicate regardless of this flag (only the point estimate is used to build the bootstrap distribution).

`InferenceIncidRiskDiff$clone():` The objects of this class are cloneable with this method.

Usage:

`InferenceIncidRiskDiff$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidRiskDiff$new(seq_des)
inf$compute_estimate()
```

InferenceIncidWald *Wald Incidence Inference*

Description

Unadjusted incidence inference using the empirical risk difference $\hat{\theta} = \bar{Y}_T - \bar{Y}_C$ (sample proportions in the treatment and control arms) together with the standard unpooled Wald standard error $\hat{s}(\hat{\theta}) = \sqrt{\bar{Y}_T(1-\bar{Y}_T)/n_T + \bar{Y}_C(1-\bar{Y}_C)/n_C}$ and normal-approximation confidence interval / hypothesis test $\hat{\theta} \pm z_{1-\alpha/2}\, \hat{s}(\hat{\theta})$. This is the classical two-proportion Wald interval (e.g. Wald 1943; see `InferenceIncidNewcombeRiskDiff` for a small-sample-robust alternative). `likelihood_tier = "none"`: no likelihood is fit, so no likelihood-ratio or parametric-bootstrap methods are exposed; the reservoir/covariate structure is ignored, unlike `InferenceIncidRiskDiff`'s covariate- adjusted linear-probability model. Non-estimable if either arm has zero subjects.

Super class

`Inference` -> `InferenceIncidWald`

Methods

Public methods:

- `InferenceIncidWald$new()`
- `InferenceIncidWald$clone()`

`InferenceIncidWald$new()`: Uses the randomization-CI layer's two-sided p-value contract (`InferenceRandCI`'s version, not `InferenceRand`'s): for incidence responses this dispatches to the Zhang exact randomization test where applicable rather than refusing outright, matching this class's pre-migration old-ladder behavior (it inherited from `InferenceAllSimpleAverageDiff`, whose own pin was already corrected to `InferenceRandCI` – see that file's identical rationale). This class independently composes the same components rather than truly inheriting `InferenceAllSimpleAverageDiff`, so it had its own stale copy of the old `InferenceRand` pin, which silently regressed Zhang dispatch (‘compute_rand_two_sided_pval()‘ started throwing "Randomization tests are not supported for incidence" for the same designs the old ladder handled correctly) – found via `test-incid-wald-migration-golden.R`'s `randomization_pval` case going from ‘"ok"‘ to ‘"unsupported"‘.

Pins asymptotic CI dispatch to the composed `Wald` component's implementation (`InferenceAsymp`), which uses `private$get_standard_error()` (this class's two-proportion Wald SE) and `private$get_degrees_of_free` Without this explicit pin, the `SimpleMeanDifference` component – composed after `Wald` in this class's `components` list – silently wins the assembly-order collision and dispatches its own Welch's t-test on raw `y` instead, making the documented Wald formula dead code. See class documentation.

Pins asymptotic p-value dispatch to the composed `Wald` component's implementation; see `$compute_asymp_confidence_i` for the rationale.

Initialize Wald risk-difference incidence inference and prepare the treatment/control binomial summaries used by `InferenceIncidWald` and related `InferenceIncidRiskDiff` methods.

Usage:

```
InferenceIncidWald$new(des_obj, model_formula = NULL, verbose = FALSE)
```

Arguments:

`des_obj` A completed design object.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`delta` Null treatment effect value.

Returns: A new `InferenceIncidWald` object.

`InferenceIncidWald$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceIncidWald$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'incidence')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(rbinom(10, 1, 0.5))
inf = InferenceIncidWald$new(seq_des)
inf$compute_estimate()
```

InferenceOrdinalAdjCatLogitRegr

Adjacent Category Logit Regression Inference for Ordinal Responses

---

Description

Fits an adjacent-category logit regression for ordinal responses (via `fast_adjacent_category_logit_cpp` — see that page for the full model, an alternative ordinal parameterization to the cumulative-logit proportional-odds model) using the treatment indicator and, optionally, all recorded covariates as predictors. This is a full-likelihood class (`likelihood_tier = "full"`) supporting score, gradient, and likelihood-ratio tests, plus parametric likelihood-ratio bootstrap calibration, in addition to Wald and resampling-based inference. Bayesian-bootstrap inference is temporarily unavailable because the current non-uniform weighted hook fits a cumulative-logit surrogate rather than the adjacent-category likelihood. It will remain disabled until the native weighted adjacent-category backend described in the package implementation plan lands.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether compute_rand_two_sided_pval() is actually usable on this instance right now – FALSE exactly when it would stop(): an incidence-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see private$should_use_design_ra TRUE for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so InferenceSuite can check this before attempting the sentinel instead of relying on the stop() being silently swallowed into a pval = NA "ok" row – the single source of truth for both this check and compute_rand_two_sided_pval()'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceOrdinalAdjCatLogitRegr

Methods

Public methods:

- InferenceOrdinalAdjCatLogitRegr$approximate_randomization_distribution_beta_hat_T()
- InferenceOrdinalAdjCatLogitRegr$supports_rand_pval_for_incidence()
- InferenceOrdinalAdjCatLogitRegr$compute_rand_two_sided_pval()
- InferenceOrdinalAdjCatLogitRegr$clone()

InferenceOrdinalAdjCatLogitRegr$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceOrdinalAdjCatLogitRegr$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

r Number of randomization vectors. Default 501.

r Number of randomization vectors.

delta The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalAdjCatLogitRegr$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceOrdinalAdjCatLogitRegr$supports_rand_pval_for_incidence()
```

`InferenceOrdinalAdjCatLogitRegr$compute_rand_two_sided_pval()`:

Usage:

```
InferenceOrdinalAdjCatLogitRegr$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalAdjCatLogitRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalAdjCatLogitRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalAdjCatLogitRegr$new(seq_des)
inf$compute_estimate()
```

`InferenceOrdinalCauchitRegr`

Cauchit Regression Inference for Ordinal Responses

Description

Cauchit-link cumulative-odds ordinal regression: $P(Y \leq k \mid w, x) = F_{\text{Cauchy}}(\alpha_k - \beta_T w - \beta_X^\top x)$, where F_{Cauchy} is the standard Cauchy CDF, α_k are category-specific cutpoints, and β_T is the treatment log-odds coefficient on the cauchit scale (proportional-odds-style shift common to all categories). Fit by maximum likelihood. The heavy-tailed Cauchy link is markedly less sensitive to outlying/extreme response categories than the logit or probit link, at the cost of a less familiar effect-size interpretation. `likelihood_tier = "full"`: exposes likelihood-ratio, score, gradient, and parametric-likelihood-bootstrap inference in addition to the Wald/asymptotic and Bayesian-bootstrap paths.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one

per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceOrdinalCauchitRegr

Methods

Public methods:

- InferenceOrdinalCauchitRegr$approximate_randomization_distribution_beta_hat_T()
- InferenceOrdinalCauchitRegr$supports_rand_pval_for_incidence()
- InferenceOrdinalCauchitRegr$compute_rand_two_sided_pval()
- InferenceOrdinalCauchitRegr$clone()

InferenceOrdinalCauchitRegr$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceOrdinalCauchitRegr$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

r Number of randomization vectors. Default 501.

r Number of randomization vectors.

delta The null difference. Default 0.

delta Null difference.

transform_responses Type of transformation. Default "none".

transform_responses Transformation.

show_progress Show progress bar. Default TRUE.

show_progress Show progress.

permutations Pre-computed permutations. Default NULL.

permutations Pre-computed permutations.

debug If TRUE, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default FALSE.

zero_one_logit_clamp The clamping amount for exact 0 and 1 values when logging

zero_one_logit_clamp The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalCauchitRegr$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceOrdinalCauchitRegr$supports_rand_pval_for_incidence()
```

`InferenceOrdinalCauchitRegr$compute_rand_two_sided_pval()`:

Usage:

```
InferenceOrdinalCauchitRegr$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalCauchitRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalCauchitRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Agresti, A. (2010). *Analysis of Ordinal Categorical Data* (2nd ed.). Wiley. Ch. 3-4 (cumulative link models).

See Also

https://en.wikipedia.org/wiki/Ordinal_regression, https://www.statsmodels.org/stable/discretemod.html for an analogous Python cumulative-link API.

InferenceOrdinalCloglogRegr

Cumulative Cloglog Inference for Ordinal Responses

Description

Complementary log-log cumulative-odds ordinal regression: $P(Y \leq k \mid w, x) = 1 - \exp\{-\exp(\alpha_k - \beta_T w - \beta_X^\top x)\}$, where α_k are category-specific cutpoints and β_T is the treatment coefficient on the cloglog scale. Fit by maximum likelihood. The cloglog link is asymmetric (unlike logit/probit) and is the natural ordinal generalization of a proportional-hazards/grouped survival-time model, so it is preferred when the underlying process is plausibly a discretized time-to-event or extreme-value mechanism. `likelihood_tier = "full"`: exposes likelihood-ratio, score, gradient, and parametric-likelihood-bootstrap inference in addition to Wald/asymptotic and Bayesian-bootstrap paths.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceOrdinalCloglogRegr`

Methods

Public methods:

- `InferenceOrdinalCloglogRegr$approximate_randomization_distribution_beta_hat_T()`
- `InferenceOrdinalCloglogRegr$supports_rand_pval_for_incidence()`
- `InferenceOrdinalCloglogRegr$compute_rand_two_sided_pval()`
- `InferenceOrdinalCloglogRegr$clone()`

`InferenceOrdinalCloglogRegr$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceOrdinalCloglogRegr$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalCloglogRegr$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceOrdinalCloglogRegr$supports_rand_pval_for_incidence()
```

`InferenceOrdinalCloglogRegr$compute_rand_two_sided_pval()`:

Usage:

```
InferenceOrdinalCloglogRegr$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalCloglogRegr$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalCloglogRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Agresti, A. (2010). *Analysis of Ordinal Categorical Data* (2nd ed.). Wiley. Ch. 3-4 (cumulative link models); McCullagh, P. (1980). "Regression Models for Ordinal Data." *JRSS-B*, 42(2), 109-142.

See Also

https://en.wikipedia.org/wiki/Ordinal_regression

`InferenceOrdinalContRatioRegr`

Continuation Ratio Regression Inference for Ordinal Responses

---

Description

Fits a conditional (stratified) continuation-ratio logit model for ordinal responses: for cut $j = 1, \ldots, K-1$, among subjects who have reached at least category j,

$$\log \frac{\Pr(Y_i > j \mid Y_i \geq j)}{\Pr(Y_i = j \mid Y_i \geq j)} = \alpha_j + \beta_T W_i + X_i^\top \gamma,$$

a discrete-time-hazard-model analog for ordinal data, with a treatment coefficient β_T constrained equal across all cuts. $\exp(\hat{\beta}_T)$ is the common "continue vs. stop here" odds ratio: a positive β_T

means treatment pushes subjects toward higher categories of Y, matching the sign convention of every other ordinal estimator in the package. Fitting proceeds by expand_continuation_ratio_data_cpp's stacked-binary expansion followed by conditional logistic regression on the expanded data. likelihood_tier = "full": likelihood-ratio, score, gradient, and Wald tests are all available when the model converges, plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test. Validity requires the continuation-ratio proportionality assumption (a common β_T across all $K-1$ cuts).

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether compute_rand_two_sided_pval() is actually usable on this instance right now – FALSE exactly when it would stop(): an incidence-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see private$should_use_design_ra TRUE for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so InferenceSuite can check this before attempting the sentinel instead of relying on the stop() being silently swallowed into a pval = NA "ok" row – the single source of truth for both this check and compute_rand_two_sided_pval()'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceOrdinalContRatioRegr

Methods

Public methods:

- InferenceOrdinalContRatioRegr$approximate_randomization_distribution_beta_hat_T()
- InferenceOrdinalContRatioRegr$supports_rand_pval_for_incidence()
- InferenceOrdinalContRatioRegr$compute_rand_two_sided_pval()
- InferenceOrdinalContRatioRegr$clone()

InferenceOrdinalContRatioRegr$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceOrdinalContRatioRegr$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
```

```
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalContRatioRegr$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceOrdinalContRatioRegr$supports_rand_pval_for_incidence()
```

`InferenceOrdinalContRatioRegr$compute_rand_two_sided_pval()`:

Usage:

```
InferenceOrdinalContRatioRegr$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalContRatioRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalContRatioRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Agresti, A. (2010). *Analysis of Ordinal Categorical Data* (2nd ed.). Wiley, for the continuation-ratio model family.

See Also

`InferenceOrdinalStereotypeLogitRegr` and `InferenceOrdinalKKCondAdjCatLogitRegr` for related ordinal-logit expansions. See also: Ordinal regression (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalContRatioRegr$new(seq_des)
inf$compute_estimate()
```

InferenceOrdinalGCompMeanDiff

G-Computation Mean-Difference Inference for Ordinal Responses

---

Description

Fits a proportional-odds working model for an ordinal outcome using treatment and, optionally, all recorded covariates (`fast_ordinal_regression_with_var_cpp`), then estimates the marginal difference in **expected ordinal category score** by G-computation — see `gcomp_ordinal_proportional_odds_post_fit_cpp` for the exact standardization formula (`mean1 - mean0`). Standard errors are obtained by the **delta method**: a central finite-difference gradient of the mean-difference functional with respect to the fitted $[\alpha, \beta]$ parameters, propagated through the model's fitted variance-covariance matrix, $\widehat{\mathrm{Var}}(\widehat{\mathrm{md}}) =$

$\nabla^\top \widehat{\mathrm{Var}}(\hat{\theta})\nabla$. If that delta-method standard error is unavailable or non-finite, the Wald-style methods (`$compute_asymp_confidence_interval()`, `$compute_asymp_two_sided_pval()`, `$compute_wald_confidence_in` `$compute_wald_two_sided_pval()`) all silently fall back to a nonparametric bootstrap interval/p-value instead (with a warning), rather than returning `NA`.

Super class

`Inference` -> `InferenceOrdinalGCompMeanDiff`

Methods

Public methods:

- `InferenceOrdinalGCompMeanDiff$new()`
- `InferenceOrdinalGCompMeanDiff$compute_estimate()`
- `InferenceOrdinalGCompMeanDiff$compute_estimate_with_bootstrap_weights()`
- `InferenceOrdinalGCompMeanDiff$compute_asymp_confidence_interval()`
- `InferenceOrdinalGCompMeanDiff$compute_asymp_two_sided_pval()`
- `InferenceOrdinalGCompMeanDiff$compute_wald_confidence_interval()`
- `InferenceOrdinalGCompMeanDiff$compute_wald_two_sided_pval()`
- `InferenceOrdinalGCompMeanDiff$clone()`

`InferenceOrdinalGCompMeanDiff$new():` Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize the ordinal g-computation (G-Comp) inference object for a completed design with an ordinal, uncensored response.

Usage:

```
InferenceOrdinalGCompMeanDiff$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `DesignSeqOneByOne` object with an ordinal response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values by default.

`InferenceOrdinalGCompMeanDiff$compute_estimate():` Computes the G-computation standardized mean-difference treatment-effect estimate (see class documentation for the full proportional-odds-based standardization).

Usage:

```
InferenceOrdinalGCompMeanDiff$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`InferenceOrdinalGCompMeanDiff$compute_estimate_with_bootstrap_weights()`: Recomputes the G-computation mean-difference estimate under subject/block bootstrap weights (via `fast_ordinal_regression_weighted_cpp` plus `gcomp_ordinal_proportional_odds_post_fit_cpp`), used by the Bayesian bootstrap and related weighted-resampling machinery; see `InferenceNonParamBootstrap`. Runs side-effect free: the ordinary (unweighted) cached fit, warm-start state, and rank-reduced column selection are saved before the weighted refit and restored afterward (`on.exit`), so a weighted bootstrap replicate cannot corrupt the class's own point estimate or subsequent fits.

Usage:

```
InferenceOrdinalGCompMeanDiff$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Row weights for the bootstrap sample.

`estimate_only` If TRUE, skip variance calculations.

`InferenceOrdinalGCompMeanDiff$compute_asymp_confidence_interval()`: Computes a $1 - \alpha$ confidence interval for the G-Comp mean difference using the delta-method standard error (see class documentation), or falls back (with a warning) to a nonparametric bootstrap interval if that standard error is unavailable. Identical to `$compute_wald_confidence_interval()`.

Usage:

```
InferenceOrdinalGCompMeanDiff$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The significance level (default 0.05).

`InferenceOrdinalGCompMeanDiff$compute_asymp_two_sided_pval()`: Computes a two-sided Wald p-value testing $H_0 : \mathrm{md} = \texttt{delta}$ using the delta-method standard error (see class documentation), or falls back (with a warning) to a nonparametric bootstrap p-value if that standard error is unavailable. Identical to `$compute_wald_two_sided_pval()`.

Usage:

```
InferenceOrdinalGCompMeanDiff$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment effect (default 0).

`InferenceOrdinalGCompMeanDiff$compute_wald_confidence_interval()`: Identical to `$compute_asymp_confider` (both compute the same delta-method-based Wald interval, with the same bootstrap fallback); provided as an explicit alias for callers that want to name the Wald method directly rather than via the generic "asymptotic" dispatch.

Usage:

```
InferenceOrdinalGCompMeanDiff$compute_wald_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The significance level (default 0.05).

`InferenceOrdinalGCompMeanDiff$compute_wald_two_sided_pval()`: Identical to `$compute_asymp_two_sided_pva` (both compute the same delta-method-based Wald p-value, with the same bootstrap fallback); provided as an explicit alias for callers that want to name the Wald method directly rather than via the generic "asymptotic" dispatch.

Usage:

```
InferenceOrdinalGCompMeanDiff$compute_wald_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment effect (default 0).

`InferenceOrdinalGCompMeanDiff$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalGCompMeanDiff$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalGCompMeanDiff$new(seq_des)
inf$compute_estimate()
```

InferenceOrdinalJonckheereTerpstraTest

Jonckheere-Terpstra (JT) Test for Ordinal Responses

Description

Two-arm Jonckheere-Terpstra (JT) rank test for an ordinal response — for two groups, this reduces to the Mann-Whitney U statistic. The point estimate is the **stochastic superiority** probability, centered at 0 under the null: $\hat{\beta}_T = \widehat{\Pr}(Y_T > Y_C) + \frac{1}{2}\widehat{\Pr}(Y_T = Y_C) - \frac{1}{2}$, computed from category counts as $U/(n_T n_C) - 1/2$. Asymptotic inference (`$compute_asymp_confidence_interval()`, `$compute_asymp_two_sided_pval()`) uses the classical null variance of the Mann-Whitney U statistic, $\mathrm{Var}(U) = n_T n_C (n_T + n_C + 1)/12$ (no tie correction), matching `clinfun::jonckheere.test()`'s normal approximation. This class also provides an **exact**, permutation-distribution-based two-sided p-value via `$compute_exact_two_sided_pval_for_treatment_effect()` (`exact_jonckheere_terpstra_pval_cpp`). Exact enumeration is capped at one million allocations; above that limit the public method warns and returns the asymptotic normal p-value instead.

Super class

`Inference` -> `InferenceOrdinalJonckheereTerpstraTest`

Methods

Public methods:

- `InferenceOrdinalJonckheereTerpstraTest$new()`
- `InferenceOrdinalJonckheereTerpstraTest$compute_estimate()`
- `InferenceOrdinalJonckheereTerpstraTest$compute_estimate_with_bootstrap_weights()`
- `InferenceOrdinalJonckheereTerpstraTest$compute_exact_two_sided_pval_for_treatment_effect()`
- `InferenceOrdinalJonckheereTerpstraTest$compute_asymp_confidence_interval()`
- `InferenceOrdinalJonckheereTerpstraTest$compute_asymp_two_sided_pval()`
- `InferenceOrdinalJonckheereTerpstraTest$clone()`

`InferenceOrdinalJonckheereTerpstraTest$new()`: Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize the JT test object for a completed design with an ordinal, uncensored response.

Usage:

```
InferenceOrdinalJonckheereTerpstraTest$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE
)
```

Arguments:

`des_obj` A completed `DesignSeqOneByOne` object.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress.

`InferenceOrdinalJonckheereTerpstraTest$compute_estimate()`: Returns the estimated treatment effect: the stochastic superiority measure $\widehat{\Pr}(Y_T > Y_C) + \frac{1}{2}\widehat{\Pr}(Y_T = Y_C) - \frac{1}{2}$, computed from the Mann-Whitney U statistic (see class documentation).

Usage:

```
InferenceOrdinalJonckheereTerpstraTest$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`InferenceOrdinalJonckheereTerpstraTest$compute_estimate_with_bootstrap_weights()`: Recomputes the JT superiority estimate under subject/block bootstrap weights: the weighted version of the same stochastic superiority quantity, $\sum_{i,j} w_i w_j \left(\mathbb{1}[y_{T,i} > y_{C,j}] + \frac{1}{2}\mathbb{1}[y_{T,i} = y_{C,j}]\right) \Big/ \sum_{i,j} w_i w_j - \frac{1}{2}$, used by the Bayesian bootstrap and related weighted-resampling machinery. Always leaves the standard error unavailable (`NA`) regardless of `estimate_only` — this weighted path never computes the null-variance approximation.

Usage:

```
InferenceOrdinalJonckheereTerpstraTest$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject/block level.

`estimate_only` Present for interface parity; this method never computes variance components regardless of its value.

`InferenceOrdinalJonckheereTerpstraTest$compute_exact_two_sided_pval_for_treatment_effect()`: Returns the **exact**, permutation-distribution-based two-sided p-value (`exact_jonckheere_terpstra_pval_cpp`). Exact enumeration is capped at one million allocations; when that limit is exceeded, this method warns and returns `$compute_asymp_two_sided_pval()` instead.

Usage:

```
InferenceOrdinalJonckheereTerpstraTest$compute_exact_two_sided_pval_for_treatment_effect(

)
```

`InferenceOrdinalJonckheereTerpstraTest$compute_asymp_confidence_interval()`: Computes the asymptotic normal confidence interval, using the same Mann-Whitney U null-variance approximation ($n_T n_C(n_T + n_C + 1)/12$, no tie correction) as `clinfun::jonckheere.test()`; see class documentation.

Usage:

```
InferenceOrdinalJonckheereTerpstraTest$compute_asymp_confidence_interval(
  alpha = 0.05
)
```

Arguments:

`alpha` The significance level (default 0.05).

`InferenceOrdinalJonckheereTerpstraTest$compute_asymp_two_sided_pval()`: Computes the asymptotic normal two-sided p-value, using the same Z-approximation as `clinfun::jonckheere.test()`; see class documentation and `$compute_exact_two_sided_pval_for_treatment_effect()` for the exact (non-approximate) alternative.

Usage:

```
InferenceOrdinalJonckheereTerpstraTest$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null treatment effect (default 0).

`InferenceOrdinalJonckheereTerpstraTest$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalJonckheereTerpstraTest$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Jonckheere, A. R. (1954). "A Distribution-Free k-Sample Test Against Ordered Alternatives." *Biometrika*, 41(1-2), 133-145, doi:10.1093/biomet/41.12.133; Terpstra, T. J. (1952). "The Asymptotic Normality and Consistency of Kendall's Test Against Trend, When Ties Are Present in One Ranking." *Indagationes Mathematicae*, 14, 327-333.

Examples

```
set.seed(1)
x_dat <- data.frame(
  x1 = c(-1.2, -0.7, -0.2, 0.3, 0.8, 1.3, 1.8, 2.3),
  x2 = c(0, 1, 0, 1, 0, 1, 0, 1)
)
seq_des <- DesignSeqOneByOneBernoulli$new(n = nrow(x_dat), response_type = "ordinal",
  verbose = FALSE)
for (i in seq_len(nrow(x_dat))) {
  seq_des$add_one_subject_to_experiment_and_assign(x_dat[i, , drop = FALSE])
}
seq_des$add_all_subject_responses(as.integer(c(1, 2, 2, 3, 3, 4, 4, 5)))
infer <- InferenceOrdinalJonckheereTerpstraTest$
  new(seq_des, verbose = FALSE)
infer
```

InferenceOrdinalKKCLMM

Ordinal KK CLMM (Proportional Odds / logit link)

Description

Cumulative-link random-intercept mixed model for ordinal responses under a KK matching-on-the-fly design, using the logit link (proportional odds): $\mathrm{logit}(P(Y_i \leq k)) = \alpha_k - (\beta_T W_i + X_i^\top \gamma) - b_{g(i)}$, $b_g \sim N(0, \sigma_b^2)$, where $g(i)$ is subject i's matched-pair group id (reservoir subjects get singleton groups). $\exp(\hat{\beta}_T)$ is the (conditional-on-b_g) treatment odds ratio. See `InferenceAbstractKKOrdinalCLMM` for the shared model-fitting, caching, and likelihood-tier contract common to all four link-function siblings (`...Probit`, `...Cauchit`, `...Cloglog`); this class supplies only the link-function choice (`private$clmm_link() == "logit"`).

Super classes

`Inference` -> `InferenceAbstractKKOrdinalCLMM` -> `InferenceOrdinalKKCLMM`

Methods

Public methods:

- `InferenceOrdinalKKCLMM$new()`
- `InferenceOrdinalKKCLMM$clone()`

`InferenceOrdinalKKCLMM$new()`: Initialize the logit-link ordinal KK CLMM subclass; see the shared ordinal mixed-model contract in `InferenceAbstractKKOrdinalCLMM`.

Usage:

```
InferenceOrdinalKKCLMM$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object.

`model_formula` Optional formula for covariate adjustment.

`use_rcpp` Use internal Rcpp implementation (default `TRUE`).

`verbose` Print messages?

`smart_cold_start_default` Use smart cold start values?

`InferenceOrdinalKKCLMM$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalKKCLMM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalKKCLMM$new(seq_des)
inf$compute_estimate()
```

`InferenceOrdinalKKCLMMCauchit`

Ordinal KK CLMM (Cauchit link)

Description

Cumulative-link random-intercept mixed model for ordinal responses under a KK matching-on-the-fly design, using the cauchit (inverse-Cauchy-CDF) link: $\tan(\pi(P(Y_i \leq k) - 1/2)) = \alpha_k - (\beta_T W_i + X_i^\top \gamma) - b_{g(i)}$, $b_g \sim N(0, \sigma_b^2)$, where $g(i)$ is subject i's matched-pair group id. The cauchit link's heavy-tailed latent distribution makes it more robust than logit/probit to a small number of subjects near the response's extreme categories. See `InferenceAbstractKKOrdinalCLMM` for the shared model-fitting, caching, and likelihood-tier contract common to all four link-function siblings; this class supplies only the link-function choice (`private$clmm_link() == "cauchit"`).

Super classes

Inference -> InferenceAbstractKKOrdinalCLMM -> InferenceOrdinalKKCLMMCauchit

Methods

Public methods:

- InferenceOrdinalKKCLMMCauchit$new()
- InferenceOrdinalKKCLMMCauchit$clone()

InferenceOrdinalKKCLMMCauchit$new(): Initialize the cauchit-link ordinal KK CLMM subclass; see the shared ordinal mixed-model contract in InferenceAbstractKKOrdinalCLMM.

Usage:

```
InferenceOrdinalKKCLMMCauchit$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

des_obj A completed Design object.

model_formula Optional formula for covariate adjustment.

use_rcpp Use internal Rcpp implementation (default TRUE).

verbose Print messages?

smart_cold_start_default Use smart cold start values?

InferenceOrdinalKKCLMMCauchit$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalKKCLMMCauchit$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalKKCLMMCauchit$new(seq_des)
inf$compute_estimate()
```

InferenceOrdinalKKCLMMCloglog

Ordinal KK CLMM (Complementary log-log link)

Description

Cumulative-link random-intercept mixed model for ordinal responses under a KK matching-on-the-fly design, using the complementary log-log link: $\log(-\log(1 - P(Y_i \leq k))) = \alpha_k - (\beta_T W_i + X_i^\top \gamma) - b_{g(i)}$, $b_g \sim N(0, \sigma_b^2)$, where $g(i)$ is subject i's matched-pair group id. Unlike the symmetric logit/probit/ cauchit links, the cloglog link is asymmetric, making it appropriate when the ordinal categories arise from an underlying continuous-time proportional-hazards process discretized into intervals. See `InferenceAbstractKKOrdinalCLMM` for the shared model-fitting, caching, and likelihood-tier contract common to all four link-function siblings; this class supplies only the link-function choice (`private$clmm_link() == "cloglog"`).

Super classes

`Inference` -> `InferenceAbstractKKOrdinalCLMM` -> `InferenceOrdinalKKCLMMCloglog`

Methods

Public methods:

- `InferenceOrdinalKKCLMMCloglog$new()`
- `InferenceOrdinalKKCLMMCloglog$clone()`

`InferenceOrdinalKKCLMMCloglog$new()`: Initialize the cloglog-link ordinal KK CLMM subclass; see the shared ordinal mixed-model contract in `InferenceAbstractKKOrdinalCLMM`.

Usage:

```
InferenceOrdinalKKCLMMCloglog$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object.

`model_formula` Optional formula for covariate adjustment.

`use_rcpp` Use internal Rcpp implementation (default `TRUE`).

`verbose` Print messages?

`smart_cold_start_default` Use smart cold start values?

`InferenceOrdinalKKCLMMCloglog$clone()`: The objects of this class are cloneable with this method.

Usage:

`InferenceOrdinalKKCLMMCloglog$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalKKCLMMCloglog$new(seq_des)
inf$compute_estimate()
```

`InferenceOrdinalKKCLMMProbit`

Ordinal KK CLMM (Probit link)

Description

Cumulative-link random-intercept mixed model for ordinal responses under a KK matching-on-the-fly design, using the probit link: $\Phi^{-1}(P(Y_i \leq k)) = \alpha_k - (\beta_T W_i + X_i^\top \gamma) - b_{g(i)}, b_g \sim N(0, \sigma_b^2)$, where Φ is the standard normal CDF and $g(i)$ is subject i's matched-pair group id. Unlike the logit-link sibling, $\hat{\beta}_T$ here is not an odds-ratio scale parameter; it is the treatment's effect on the latent standard-normal index underlying the ordinal categories. See `InferenceAbstractKKOrdinalCLMM` for the shared model-fitting, caching, and likelihood-tier contract common to all four link-function siblings; this class supplies only the link-function choice (`private$clmm_link() == "probit"`).

Super classes

`Inference` -> `InferenceAbstractKKOrdinalCLMM` -> `InferenceOrdinalKKCLMMProbit`

Methods

Public methods:

- `InferenceOrdinalKKCLMMProbit$new()`
- `InferenceOrdinalKKCLMMProbit$clone()`

`InferenceOrdinalKKCLMMProbit$new()`: Initialize the probit-link ordinal KK CLMM subclass; see the shared ordinal mixed-model contract in `InferenceAbstractKKOrdinalCLMM`.

Usage:

```
InferenceOrdinalKKCLMMProbit$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
```

```
    verbose = FALSE,
    smart_cold_start_default = NULL
  )
```

Arguments:

`des_obj` A completed `Design` object.

`model_formula` Optional formula for covariate adjustment.

`use_rcpp` Use internal Rcpp implementation (default `TRUE`).

`verbose` Print messages?

`smart_cold_start_default` Use smart cold start values?

`InferenceOrdinalKKCLMMProbit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalKKCLMMProbit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalKKCLMMProbit$new(seq_des)
inf$compute_estimate()
```

InferenceOrdinalKKCondAdjCatLogitRegr

Adjacent Category Logit Inference for KK Matching-on-the-fly Designs

---

Description

Fits a conditional (stratified) adjacent-category logit model for ordinal responses under a KK matching-on-the-fly design:

$$\log \frac{\Pr(Y_i = j+1 \mid Y_i \in \{j, j+1\})}{\Pr(Y_i = j \mid Y_i \in \{j, j+1\})} = \alpha_j + \beta_T W_i + X_i^\top \gamma,$$

for adjacent category comparisons $j = 1, \ldots, K-1$, with cut-specific intercepts α_j and a treatment coefficient β_T constrained equal across all cuts (the parallel/proportional adjacent-category assumption). $\exp(\hat{\beta}_T)$ is the common adjacent-category odds ratio. Fitting proceeds by `expand_adjacent_category_data_cpp`'s stacked-binary expansion (each subject contributes a 0/1 row per adjacent cut they border, stratified

by matched pair) followed by conditional logistic regression on the expanded data — the matched-pair identity becomes the conditioning stratum, so the pair's shared nuisance intercept is conditioned out exactly as in a single binary conditional-logit KK model, and reservoir (unmatched) subjects each form their own singleton stratum. `likelihood_tier = "partial"` (a conditional/partial likelihood, matched-set effects are profiled out rather than estimated); `supports_likelihood_tests()` is hard `FALSE` — only Wald inference is exposed, not likelihood-ratio, score, or gradient tests. Validity requires the adjacent-category proportionality assumption (a common β_T across all $K-1$ cuts) in addition to the usual conditional-logit exchangeability-within-strata assumption induced by the KK design.

Super class

`Inference` -> `InferenceOrdinalKKCondAdjCatLogitRegr`

Methods

Public methods:

- `InferenceOrdinalKKCondAdjCatLogitRegr$new()`
- `InferenceOrdinalKKCondAdjCatLogitRegr$compute_estimate()`
- `InferenceOrdinalKKCondAdjCatLogitRegr$compute_estimate_with_bootstrap_weights()`
- `InferenceOrdinalKKCondAdjCatLogitRegr$compute_asymp_confidence_interval()`
- `InferenceOrdinalKKCondAdjCatLogitRegr$compute_asymp_two_sided_pval()`
- `InferenceOrdinalKKCondAdjCatLogitRegr$clone()`

`InferenceOrdinalKKCondAdjCatLogitRegr$new()`: Initialize inference for the conditional adjacent-category logit model $\log(\Pr(Y_i = j+1 \mid Y_i \in \{j, j+1\}) / \Pr(Y_i = j \mid Y_i \in \{j, j+1\})) = \alpha_j + \beta_T W_i + X_i^\top \gamma$ and prepare KK matched-pair structure for the stratified conditional-logit fit. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferenceOrdinalKKCondAdjCatLogitRegr$new(
  des_obj,
  verbose = FALSE,
  harden = TRUE,
  model_formula = NULL,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed KK `DesignSeqOneByOne` object with an ordinal response.

`verbose` Flag for progress messages.

`harden` Whether to apply robustness measures.

`model_formula` Optional formula for covariate adjustment.

`smart_cold_start_default` Whether to use smart cold start values by default.

`InferenceOrdinalKKCondAdjCatLogitRegr$compute_estimate()`: Fits the conditional adjacent-category logit model via stacked-binary expansion (`expand_adjacent_category_data_cpp`) plus conditional logistic regression, and returns the shared log-odds-ratio estimate $\hat{\beta}_T$.

Usage:

```
InferenceOrdinalKKCondAdjCatLogitRegr$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If `TRUE`, skip standard-error computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferenceOrdinalKKCondAdjCatLogitRegr$compute_estimate_with_bootstrap_weights()`: Recomputes the treatment estimate under subject/block-level bootstrap weights (Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded to row level via `private$expand_subject_or_block_weights_to_r` When weights are effectively constant, this collapses to the unweighted `compute_estimate()` call. Otherwise, rather than refitting the full expanded conditional-logit model under weights, it calls `weighted_ordinal_bootstrap_surrogate_fit()` — a fast weighted ordinal-logistic surrogate fit on the raw (unexpanded) design matrix — as an approximation to the weighted adjacent-category likelihood; this trades exact reweighted refitting for speed across many bootstrap replicates. No standard error is computed (`s_beta_hat_T` is always `NA`); the surrogate returns `NA` if the fit fails.

Usage:

```
InferenceOrdinalKKCondAdjCatLogitRegr$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Subject-, block-, cluster-, or matched-set bootstrap weights.

`estimate_only` If `TRUE`, compute only the weighted point estimate.

`InferenceOrdinalKKCondAdjCatLogitRegr$compute_asymp_confidence_interval()`: Wald confidence interval for the shared adjacent-category log-odds-ratio β_T, using the conditional-logit model's standard error; see `InferenceAsymp` for the shared Wald contract. Fits the model first if not already cached.

Usage:

```
InferenceOrdinalKKCondAdjCatLogitRegr$compute_asymp_confidence_interval(
  alpha = 0.05
)
```

Arguments:

`alpha` Two-sided miscoverage rate; the returned interval targets `1 - alpha` coverage.

`InferenceOrdinalKKCondAdjCatLogitRegr$compute_asymp_two_sided_pval()`: Return the adjacent-category conditional-logit asymptotic p-value for the treatment coefficient, using the shared Wald semantics documented in `InferenceAsymp`.

Usage:

```
InferenceOrdinalKKCondAdjCatLogitRegr$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null hypothesis treatment effect.

`InferenceOrdinalKKCondAdjCatLogitRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalKKCondAdjCatLogitRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Agresti, A. (2010). *Analysis of Ordinal Categorical Data* (2nd ed.). Wiley, for the adjacent-category logit model family; Kapelner, A. and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388, doi:10.1111/biom.12148, for the KK matching-on-the-fly design this class is built for.

See Also

`InferenceOrdinalAdjCatLogitRegr` for the non-KK analog. See also: Ordinal regression (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalKKCondAdjCatLogitRegr$new(seq_des)
inf$compute_estimate()
```

`InferenceOrdinalKKGEE` *GEE Inference for KK Designs with Ordinal Response*

Description

Fits a **proportional-odds local-odds-ratio** Generalized Estimating Equations model, via `multgee::ordLORgee`, for ordinal responses under a KK matching-on-the-fly design, using the treatment indicator and, optionally, all recorded covariates as predictors. Each GEE cluster is either a matched pair (2 members) or a reservoir singleton (1 member) — GEE is used here purely to fit one marginal cumulative-logit model jointly across matched-pair and reservoir subjects while accounting for the within-pair correlation the matching induces, not as a longitudinal/repeated-measures tool. Unlike the other `Inference*KKGEE` classes in this family (continuous/count/incidence/proportion, which use an internal Rcpp solver or `geepack::geeglm` with an exchangeable working correlation), this ordinal class always requires the **multgee** package and has no `use_rcpp` option. The raw **multgee** treatment coefficient is negated when reported so that, consistently with EDI's other ordinal estimators, a positive estimate means movement toward higher response categories. Inference is quasi-likelihood/ estimating-equation based (`likelihood_tier = "quasi"`): standard errors are GEE sandwich (robust) standard errors, not model-likelihood-based. Bayesian-bootstrap inference is temporarily unavailable because `multgee::ordLORgee` does not accept the non-uniform observation weights needed to refit the same clustered estimator. It will remain disabled until the weighted ordinal-GEE implementation planned for v1.1.0 is complete.

Super class

Inference -> InferenceOrdinalKKGEE

Methods

Public methods:

- InferenceOrdinalKKGEE$new()
- InferenceOrdinalKKGEE$compute_estimate_with_bootstrap_weights()
- InferenceOrdinalKKGEE$clone()

InferenceOrdinalKKGEE$new(): Initialize KK ordinal GEE inference, validate the ordinal matched/reservoir design, and prepare the multgee::ordLORgee proportional-odds local-odds-ratio GEE fitting machinery used by InferenceOrdinalKKGEE. Requires the **multgee** package; errors at construction if it is not installed.

Usage:

```
InferenceOrdinalKKGEE$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

des_obj A completed Design object with an ordinal response.

model_formula Optional formula for covariate adjustment. If NULL (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

verbose Whether to print progress messages.

smart_cold_start_default Whether to use smart cold start values.

InferenceOrdinalKKGEE$compute_estimate_with_bootstrap_weights(): Recomputes the KK ordinal treatment estimate under subject/block bootstrap weights, used by the Bayesian bootstrap and related weighted-resampling machinery. If the supplied weights are all (numerically) equal, this short-circuits to the unweighted $compute_estimate(estimate_only = TRUE) (the multgee proportional-odds GEE fit) rather than refitting. Otherwise, since multgee::ordLORgee does not support observation weights, this falls back to a **different, approximating** model: a plain (non-GEE, no matched-pair clustering) weighted proportional-odds ordinal logistic regression via fast_ordinal_regression_weighted_cpp, treating the coefficient on the first predictor column as the treatment effect. This always leaves the standard error and degrees of freedom unavailable (s_beta_hat_T = NA, df = Inf) regardless of estimate_only, since it is a point-estimate-only fallback path.

Usage:

```
InferenceOrdinalKKGEE$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Subject-, block-, cluster-, or matched-set bootstrap weights.

`estimate_only` If `TRUE`, compute only the weighted point estimate. Has no effect on the weighted (non-uniform-weight) fallback path, which never computes a standard error regardless.

`InferenceOrdinalKKGEE$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalKKGEE$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Touloumis, A. (2015). "R Package multgee: A Generalized Estimating Equations Solver for Multinomial Responses." *Journal of Statistical Software*, 64(8), 1-14, doi:10.18637/jss.v064.i08, for the local-odds-ratio GEE solver used here; Liang, K.-Y., and Zeger, S. L. (1986). "Longitudinal Data Analysis Using Generalized Linear Models." *Biometrika*, 73(1), 13-22, doi:10.1093/biomet/73.1.13, for the underlying GEE estimating-equation framework.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalKKGEE$new(seq_des)
inf$compute_estimate()
```

InferenceOrdinalKKGLMM

GLMM Inference for KK Designs with Ordinal Response

Description

Fits a cumulative-logit random-intercept mixed model (proportional odds) for ordinal responses under a KK matching-on-the-fly design: $\mathrm{logit}(P(Y_i \le k \mid w_i, x_i, b_{g(i)})) = \alpha_k - (\beta_T w_i + x_i^\top \gamma) - b_{g(i)}$, for cutpoints $\alpha_1 < \cdots < \alpha_{K-1}$, treatment indicator w_i, covariates x_i, and a matched-pair random intercept $b_g \sim N(0, \sigma_b^2)$ that is integrated out of the marginal likelihood (either by adaptive Gauss-Hermite quadrature when `use_rcpp = TRUE`, the default; see `fast_ordinal_glmm_cpp` for the quadrature order and optimizer details, or by **glmmTMB**'s Laplace approximation when `use_rcpp = FALSE`). $g(i)$ is subject i's matched-pair group id; reservoir (unmatched) subjects each get their own singleton group, contributing no within-group correlation but still entering the joint likelihood. The treatment coefficient β_T is a log-odds-ratio: $\exp(\beta_T)$ is the (conditional-on-b_g) odds ratio of being at or above any given response category. `likelihood_tier = "full"`: likelihood-ratio, score, Wald, and gradient tests are available when `use_rcpp = TRUE` and the fit converges;

use_rcpp = FALSE disables likelihood-test support (private$supports_likelihood_tests() returns FALSE) because **glmmTMB**'s Laplace-approximate likelihood is not wired into this package's score/gradient/LR machinery. Validity requires the random-intercept structure to correctly capture the design's matching dependence, proportional odds (the treatment/covariate effect is constant across cutpoints), and correct specification of the fixed-effects formula.

This differs from the GEE sibling InferenceOrdinalKKGEE (documented above) in estimand and inference basis: the GLMM's β_T is a subject-specific (conditional) log-odds-ratio with model-likelihood-based inference, while the GEE's is a population-averaged (marginal) log-odds-ratio with sandwich-based inference; the two need not numerically agree even on the same data, and the correct choice depends on whether a subject-specific/conditional or population-averaged/marginal treatment effect is of interest.

Super class

Inference -> InferenceOrdinalKKGLMM

Methods

Public methods:

- InferenceOrdinalKKGLMM$new()
- InferenceOrdinalKKGLMM$compute_estimate()
- InferenceOrdinalKKGLMM$compute_asymp_confidence_interval()
- InferenceOrdinalKKGLMM$compute_asymp_two_sided_pval()
- InferenceOrdinalKKGLMM$compute_estimate_with_bootstrap_weights()
- InferenceOrdinalKKGLMM$clone()

InferenceOrdinalKKGLMM$new(): Initialize inference for the cumulative-logit random-intercept mixed model $\mathrm{logit}(P(Y_i \le k)) = \alpha_k - (\beta_T W_i + X_i^\top \gamma) - b_{g(i)}$, $b_g \sim N(0, \sigma_b^2)$, where $g(i)$ is subject i's matched-pair group id (reservoir subjects get singleton groups). Does not fit the model; the fit is deferred to the first call to compute_estimate() or a method that requires it.

Usage:

```
InferenceOrdinalKKGLMM$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

des_obj A completed Design object with an ordinal response.

model_formula Optional formula for covariate adjustment. If NULL (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

use_rcpp Logical. If TRUE (default), use internal Rcpp.

verbose Whether to print progress messages.

smart_cold_start_default Whether to use smart cold start values.

`InferenceOrdinalKKGLMM$compute_estimate()`: Fits the cumulative-logit random-intercept mixed model by (adaptive-Gauss-Hermite- or Laplace-)approximate maximum likelihood and returns $\hat{\beta}_T$, the estimated treatment log-odds-ratio, conditional on the matched-pair random intercept. Caches the fitted model object, full parameter vector, and (when `estimate_only = FALSE`) the standard error and degrees of freedom for reuse by `compute_asymp_confidence_interval()`, `compute_asymp_two_sided_pval()`, and likelihood-test methods; a fit that fails the kernel's projected-gradient convergence check, produces non-finite parameters, reaches the upper random-effect variance boundary, exceeds `private$max_abs_reasonable_coef`, or lacks a finite positive treatment-coefficient variance is cached as nonestimable rather than returned. A valid near-zero random-effect variance boundary is accepted using conditional fixed-effect information. The native optimizer retains multistart L-BFGS for basin selection and, only when its finite selected point fails the projected-score tolerance, applies a damped-Newton polish using the numerical Hessian. The polished point is retained only if it remains finite and does not increase the negative log-likelihood; at a valid lower variance boundary, the KKT-satisfied variance coordinate is excluded from that Newton system.

Usage:

```
InferenceOrdinalKKGLMM$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If `TRUE`, skip standard-error/variance-component computation and cache only the point estimate; used by randomization and bootstrap resampling paths where only $\hat{\beta}_T$ is needed per replicate.

`InferenceOrdinalKKGLMM$compute_asymp_confidence_interval()`: Wald confidence interval for β_T using the fitted model's standard error and degrees of freedom; see `InferenceAsymp` for the shared $\hat{\beta}_T \pm t_{\alpha/2,df} \cdot \widehat{se}(\hat{\beta}_T)$ (or z-based when `df = Inf`) contract. Fits the model first if not already cached.

Usage:

```
InferenceOrdinalKKGLMM$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Two-sided miscoverage rate; the returned interval targets `1 - alpha` coverage.

`InferenceOrdinalKKGLMM$compute_asymp_two_sided_pval()`: Two-sided Wald test of $H_0 : \beta_T =$ `delta` against $H_1 : \beta_T \neq$ `delta`, using the fitted model's standard error and degrees of freedom; see `InferenceAsymp` for the shared t/z test contract. Fits the model first if not already cached.

Usage:

```
InferenceOrdinalKKGLMM$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Treatment log-odds-ratio value under the null hypothesis.

`InferenceOrdinalKKGLMM$compute_estimate_with_bootstrap_weights()`: Refits the ordinal random-intercept model with subject/block-level weights applied to each row's contribution to the marginal likelihood (Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded from subject/block level to individual rows via `private$expand_subject_or_block_weights_to_row_weights()`). The weighted refit uses **ordinal**'s cumulative-logit CLMM with 20-point adaptive Gauss-Hermite quadrature, preserving the conditional treatment-coefficient estimand reported by the unweighted

fit. When weights are effectively constant, this collapses to the unweighted `compute_estimate()` call (returns `df = Inf` to signal a degenerate/skipped bootstrap replicate rather than refitting). Rows with non-finite or non-positive weight, or non-finite response, are dropped from the weighted fit; if no rows remain, the estimate is `NA`.

Usage:

```
InferenceOrdinalKKGLMM$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Subject-, block-, cluster-, or matched-set bootstrap weights.

`estimate_only` If `TRUE`, compute only the weighted point estimate.

`InferenceOrdinalKKGLMM$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalKKGLMM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Hedeker, D., and Gibbons, R. D. (1994). "A Random-Effects Ordinal Regression Model for Multilevel Analysis." *Biometrics*, 50(4), 933-944, doi:10.2307/2533433, for the random-effects cumulative-logit model; Pinheiro, J. C., and Bates, D. M. (1995). "Approximations to the Log-Likelihood Function in the Nonlinear Mixed-Effects Model." *Journal of Computational and Graphical Statistics*, 4(1), 12-35, doi:10.1080/10618600.1995.10474663, for the adaptive Gauss-Hermite quadrature approximation used to integrate out the random intercept.

See Also

Comparable Python API: statsmodels MixedLM (continuous analog; no ordinal-GLMM in **statsmodels**). See also: Ordinal regression and Mixed model (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalKKGLMM$new(seq_des)
inf$compute_estimate()
```

InferenceOrdinalOrderedProbitRegr

Ordered Probit Regression Inference for Ordinal Responses

Description

Fits a cumulative-probit ("ordered probit") model for ordinal responses: $\Phi^{-1}(P(Y_i \leq k)) = \alpha_k - (\beta_T W_i + X_i^\top \gamma)$, for cutpoints $\alpha_1 < \cdots < \alpha_{K-1}$, where Φ is the standard normal CDF, W_i is the treatment indicator, and X_i are optional recorded covariates, by maximum likelihood (`fast_ordinal_probit_regression_cpp`/ `fast_ordinal_probit_regression_with_var_cpp`). As with binary probit regression, $\hat{\beta}_T$ is not an odds-ratio-scale parameter: it is the treatment's effect on the latent standard-normal index underlying the ordinal categories. `likelihood_tier = "full"`: likelihood-ratio, score, gradient, and Wald tests are all available when the model converges, plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test. Validity requires the proportional/parallel cutpoints assumption (a single β_T shared across all cutpoints) in addition to the usual latent-normal-index assumption.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceOrdinalOrderedProbitRegr`

Methods

Public methods:

- `InferenceOrdinalOrderedProbitRegr$approximate_randomization_distribution_beta_hat_T()`
- `InferenceOrdinalOrderedProbitRegr$supports_rand_pval_for_incidence()`

- InferenceOrdinalOrderedProbitRegr$compute_rand_two_sided_pval()
- InferenceOrdinalOrderedProbitRegr$clone()

InferenceOrdinalOrderedProbitRegr$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceOrdinalOrderedProbitRegr$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

InferenceOrdinalOrderedProbitRegr$supports_rand_pval_for_incidence():

Usage:

```
InferenceOrdinalOrderedProbitRegr$supports_rand_pval_for_incidence()
```

InferenceOrdinalOrderedProbitRegr$compute_rand_two_sided_pval():

Usage:

```
InferenceOrdinalOrderedProbitRegr$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalOrderedProbitRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalOrderedProbitRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

McCullagh, P. (1980). "Regression Models for Ordinal Data." *Journal of the Royal Statistical Society, Series B*, 42(2), 109-142, doi:10.1111/j.25176161.1980.tb01109.x, for the cumulative-link ordinal model family this class's probit link instantiates.

See Also

`InferenceOrdinalCauchitRegr`, `InferenceOrdinalCloglogRegr` for other cumulative-link function choices on the same ordinal model family. See also: Ordinal regression and Probit model (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalOrderedProbitRegr$new(seq_des)
inf$compute_estimate()
```

InferenceOrdinalPairedSignTest

Paired Sign Test Inference for KK Designs with Ordinal Response

Description

Fits the classical paired sign test for ordinal responses under a KK matching-on-the-fly design. For each matched pair i with treated member response $Y_{i,T}$ and control member response $Y_{i,C}$, only the sign of the within-pair difference $Y_{i,T} - Y_{i,C}$ is used; tied pairs ($Y_{i,T} = Y_{i,C}$) are dropped from the effective sample. The estimand is $\theta = P(Y_T > Y_C \mid \text{pair untied})$, and the reported treatment effect is $\hat{\beta}_T = \hat{p} - 0.5$, where $\hat{p}$ is the sample proportion of untied pairs favoring treatment; $\beta_T = 0$ corresponds to $\theta = 0.5$ (no directional preference). The standard error is the usual binomial-proportion formula $\sqrt{\hat{p}(1-\hat{p})/n_{\text{eff}}}$, where n_{eff} is the number of untied pairs. **Reservoir (unmatched) subjects are not included** — this is a purely within-pair test, unlike the IVWC-style classes elsewhere in the KK family that combine matched-pair and reservoir information. `likelihood_tier = "none"` (`supports_likelihood_tests()` is hard `FALSE`): only Wald inference on the proportion scale is exposed. Bootstrap and jackknife are deliberately unsupported and throw explicit errors (see `approximate_bootstrap_distribution_beta_hat_T()` and `approximate_jackknife_distribution_beta_hat_T()`), since subject-level resampling or deletion would violate the matched-pair design's dependence structure; randomization inference (`compute_rand_two_sided_pval()`) remains available since it permutes treatment assignment within the design's own randomization mechanism rather than resampling subjects. Requires a KK matching-on-the-fly design (`DesignSeqOneByOneKK14/KK21`) or `DesignFixedBinaryMatch`; a design with no discordant (untied) pairs is cached as nonestimable for the standard error (point estimate `0`) or fully nonestimable, per `harden`.

Super class

`Inference` -> `InferenceOrdinalPairedSignTest`

Methods

Public methods:

- `InferenceOrdinalPairedSignTest$new()`
- `InferenceOrdinalPairedSignTest$compute_estimate()`
- `InferenceOrdinalPairedSignTest$compute_estimate_with_bootstrap_weights()`
- `InferenceOrdinalPairedSignTest$compute_asymp_confidence_interval()`
- `InferenceOrdinalPairedSignTest$compute_asymp_two_sided_pval()`
- `InferenceOrdinalPairedSignTest$clone()`

`InferenceOrdinalPairedSignTest$new()`: Uses the shared randomization-test two-sided p-value contract; see `InferenceRand`. Pinned from plain `InferenceRand` (not `InferenceRandCI`) per the established ordinal-class precedent – Zhang dispatch is incidence-only.

Initialize inference for the paired sign test on within-pair response differences $Y_{i,T} - Y_{i,C}$; see `InferenceOrdinalPairedSignTest` for the model form. Requires a KK matching-on-the-fly design (`DesignSeqOneByOneKK14/KK21`) or `DesignFixedBinaryMatch`. Does not compute the

sign-test statistic; that is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferenceOrdinalPairedSignTest$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed KK matching-on-the-fly design object.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`r` Number of randomization draws.

`delta` Null treatment effect.

`transform_responses` Optional response transformation.

`na.rm` Whether to drop non-finite draws.

`show_progress` Whether to show a progress bar.

`permutations` Optional pre-computed permutations.

`zero_one_logit_clamp` Clamp for logit transforms.

`InferenceOrdinalPairedSignTest$compute_estimate()`: Computes the pair-sign counts (pos/neg, ties dropped) from the design's matched-pair structure and returns $\hat{\beta}_T = \hat{p} - 0.5$, where $\hat{p}$ is the proportion of untied pairs favoring treatment. If every pair is tied, the estimate is `0` (no directional preference) and the fit is cached as standard-error-nonestimable (or fully nonestimable when `harden = FALSE`).

Usage:

```
InferenceOrdinalPairedSignTest$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip the standard-error computation and cache only the point estimate.

`InferenceOrdinalPairedSignTest$compute_estimate_with_bootstrap_weights()`: Recomputes $\hat{\beta}_T$ under subject/block-level bootstrap weights (Bayesian-bootstrap draw weights, expanded to row level via `private$expand_subject_or_block_weights_to_row_weights()`): for each matched pair, a weighted vote is cast toward whichever member has the higher response, using the mean bootstrap weight of the pair's two rows; $\hat{\beta}_T^{(w)}$ is the weighted proportion of treatment-favoring pairs minus `0.5`. No standard error is computed (`s_beta_hat_T` is always `NA`). Pairs with no discordant (untied) weighted votes are cached as nonestimable.

Usage:

```
InferenceOrdinalPairedSignTest$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Subject-, block-, cluster-, or matched-set bootstrap weights.

`estimate_only` If `TRUE`, compute only the weighted point estimate.

`InferenceOrdinalPairedSignTest$compute_asymp_confidence_interval()`: Wald confidence interval for $\beta_T = \theta - 0.5$ (equivalently, for $\theta = P(Y_T > Y_C \mid \text{pair untied})$), using the binomial-proportion standard error; see `InferenceAsymp` for the shared Wald contract. Fits (computes the pair-sign counts) first if not already cached.

Usage:

```
InferenceOrdinalPairedSignTest$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Two-sided miscoverage rate; the returned interval targets `1 - alpha` coverage.

`InferenceOrdinalPairedSignTest$compute_asymp_two_sided_pval()`: Two-sided Wald test of $H_0 : \theta = 0.5$ (equal chance of favoring treatment vs. control among untied pairs) against $H_1 : \theta \neq 0.5$, using the binomial-proportion standard error; see `InferenceAsymp` for the shared Wald contract. Only `delta = 0` is supported (the sign test's null is fixed at no directional preference; a non-zero `delta` throws).

Usage:

```
InferenceOrdinalPairedSignTest$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null value for β_T; must be `0`.

`InferenceOrdinalPairedSignTest$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalPairedSignTest$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Dixon, W. J., and Mood, A. M. (1946). "The Statistical Sign Test." *Journal of the American Statistical Association*, 41(236), 557-566, doi:10.2307/2280577, for the classical paired sign test; Kapelner, A. and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388, doi:10.1111/biom.12148, for the KK matching-on-the-fly design this class is built for.

See Also

Sign test (Wikipedia).

Examples

```
set.seed(1)
x_dat <- data.frame(
  x1 = c(-1.2, -0.7, -0.2, 0.3, 0.8, 1.3, 1.8, 2.3),
  x2 = c(0, 1, 0, 1, 0, 1, 0, 1)
)
seq_des <- DesignSeqOneByOneKK21$new(n = nrow(x_dat), response_type = "ordinal",
verbose = FALSE)
for (i in seq_len(nrow(x_dat))) {
  seq_des$add_one_subject_to_experiment_and_assign(x_dat[i, , drop = FALSE])
}
seq_des$add_all_subject_responses(as.integer(c(1, 2, 2, 3, 3, 4, 4, 5)))
infer <- InferenceOrdinalPairedSignTest$
  new(seq_des, verbose = FALSE)
infer
```

InferenceOrdinalPartialProportionalOddsRegr

Partial Proportional-Odds Regression Inference for Ordinal Responses

---

Description

Fits a **partial** proportional-odds cumulative-logit model for an ordinal response: a subset of covariates named in `nonparallel` are allowed a separate coefficient at each cumulative threshold (relaxing the proportional-odds/parallel-lines assumption for exactly those covariates), while every other covariate — including the **treatment** indicator, which is *always* fit as a parallel (proportional) term regardless of `nonparallel` — keeps one shared coefficient across all thresholds. The reported treatment effect is therefore always a single proportional (threshold-invariant) log-odds shift, even when other covariates' effects are allowed to vary by threshold. When `nonparallel` is empty, fitting uses this package's fast Rcpp full-proportional-odds solver (`fast_ordinal_regression_with_var_cpp`); otherwise it falls back, in order, to `VGAM::vglm(family = VGAM::cumulative(parallel = ...))`, `ordinal::clm(nominal = ...)`, and (only when `nonparallel` is empty and the earlier fast/VGAM/clm attempts failed) `MASS::polr`. Each fallback requires its corresponding package to be installed; unavailable packages are silently skipped in favor of the next fallback.

Super class

`Inference` -> `InferenceOrdinalPartialProportionalOddsRegr`

Methods

Public methods:

- `InferenceOrdinalPartialProportionalOddsRegr$new()`
- `InferenceOrdinalPartialProportionalOddsRegr$compute_estimate()`
- `InferenceOrdinalPartialProportionalOddsRegr$compute_estimate_with_bootstrap_weights()`

- InferenceOrdinalPartialProportionalOddsRegr$compute_asymp_confidence_interval()
- InferenceOrdinalPartialProportionalOddsRegr$compute_asymp_two_sided_pval()
- InferenceOrdinalPartialProportionalOddsRegr$compute_wald_confidence_interval()
- InferenceOrdinalPartialProportionalOddsRegr$compute_wald_two_sided_pval()
- InferenceOrdinalPartialProportionalOddsRegr$benchmark_asymp_two_sided_pval_breakdown()
- InferenceOrdinalPartialProportionalOddsRegr$clone()

InferenceOrdinalPartialProportionalOddsRegr$new(): Uses the shared randomization two-sided p-value contract; see InferenceRand.

Initialize partial proportional-odds ordinal regression inference for a completed design with an ordinal, uncensored response.

Usage:

```
InferenceOrdinalPartialProportionalOddsRegr$new(
  des_obj,
  verbose = FALSE,
  harden = TRUE,
  model_formula = NULL,
  nonparallel = character(0),
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed DesignSeqOneByOne object with an ordinal response.

`verbose` Whether to print progress messages.

`harden` Whether to apply robustness measures.

`model_formula` Optional formula for covariate adjustment. If NULL (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`nonparallel` Names of covariates (not including "treatment", which is always fit as a parallel/proportional term) allowed a separate coefficient at each cumulative threshold, relaxing the proportional-odds assumption for those covariates specifically.

`smart_cold_start_default` Whether to use smart cold start values by default.

InferenceOrdinalPartialProportionalOddsRegr$compute_estimate(): Retrieves the estimated (always-parallel) treatment log-odds shift from the partial proportional-odds fit (see class documentation for the fitting backend cascade).

Usage:

```
InferenceOrdinalPartialProportionalOddsRegr$compute_estimate(
  estimate_only = FALSE
)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

Returns: The estimated treatment effect.

InferenceOrdinalPartialProportionalOddsRegr$compute_estimate_with_bootstrap_weights(): Recomputes the partial-proportional-odds treatment estimate under subject/block bootstrap weights,

used by the Bayesian bootstrap and related weighted-resampling machinery. If the weights are effectively constant, short-circuits to the unweighted `$compute_estimate(estimate_only = TRUE)`. Otherwise refits with weights via the same backend cascade as the unweighted fit (`VGAM`/`ordinal`/`MASS::polr`, each weighted), and if **all** of those fail, falls back further to a plain weighted binary-logistic surrogate fit (`weighted_ordinal_bootstrap_surrogate_fit(..., method = "logistic")`) that does not model the ordinal structure at all. Never computes a standard error on any weighted path (`s_beta_hat_T` is always `NA`), regardless of `estimate_only`.

Usage:

```
InferenceOrdinalPartialProportionalOddsRegr$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Subject-, block-, cluster-, or matched-set bootstrap weights.

`estimate_only` Present for interface parity; this method never computes variance components regardless of its value.

`InferenceOrdinalPartialProportionalOddsRegr$compute_asymp_confidence_interval()`**:** Computes a Wald-style confidence interval for the treatment log-odds shift, using the model-based standard error from whichever backend (fast Rcpp solver, `VGAM`, `ordinal`, or `MASS::polr`) successfully fit the unweighted model (see class documentation). If that standard error is unavailable (`NA`, non-finite, or 0) — e.g. because the fit succeeded via a fallback path that doesn't report one — the interval is explicitly marked non-estimable (`c(NA, NA)`) when `private$harden` is `TRUE`, or raises an error otherwise, rather than silently returning a misleading result. Identical to `$compute_wald_confidence_interval()`.

Usage:

```
InferenceOrdinalPartialProportionalOddsRegr$compute_asymp_confidence_interval(
  alpha = 0.05
)
```

Arguments:

`alpha` Significance level for the interval.

Returns: A confidence interval for the treatment effect.

`InferenceOrdinalPartialProportionalOddsRegr$compute_asymp_two_sided_pval()`**:** Computes a Wald-style two-sided p-value testing $H_0 : \beta_T =$ `delta`, using the same model-based standard error as `$compute_asymp_confidence_interval()`; if unavailable, marked non-estimable (`NA`) or an error is raised, per `private$harden` — see that method's documentation. Identical to `$compute_wald_two_sided_pval()`.

Usage:

```
InferenceOrdinalPartialProportionalOddsRegr$compute_asymp_two_sided_pval(
  delta = 0
)
```

Arguments:

`delta` Null treatment effect to test.

Returns: A two-sided p-value.

`InferenceOrdinalPartialProportionalOddsRegr$compute_wald_confidence_interval()`: Identical to `$compute_asymp_confidence_interval()`; provided as an explicit alias for callers that want to name the Wald method directly rather than via the generic "asymptotic" dispatch.

Usage:

```
InferenceOrdinalPartialProportionalOddsRegr$compute_wald_confidence_interval(
  alpha = 0.05
)
```

Arguments:

`alpha` Significance level for the interval.

Returns: A confidence interval for the treatment effect.

`InferenceOrdinalPartialProportionalOddsRegr$compute_wald_two_sided_pval()`: Identical to `$compute_asymp_two_sided_pval()`; provided as an explicit alias for callers that want to name the Wald method directly rather than via the generic "asymptotic" dispatch.

Usage:

```
InferenceOrdinalPartialProportionalOddsRegr$compute_wald_two_sided_pval(
  delta = 0
)
```

Arguments:

`delta` Null treatment effect to test.

Returns: A two-sided p-value.

`InferenceOrdinalPartialProportionalOddsRegr$benchmark_asymp_two_sided_pval_breakdown()`: Diagnostic helper for performance investigation: runs the same computation as `$compute_asymp_two_sided_pval()` (fit the partial proportional-odds model requiring a standard error, cache the estimate/SE/df, compute the two-sided Wald p-value) but separately times each of the three stages — model fit, cache materialization, and final p-value arithmetic — via `proc.time()`. If the fit fails or has no usable standard error, returns immediately with only `fit_time` populated and every other timing/result field `NA`.

Usage:

```
InferenceOrdinalPartialProportionalOddsRegr$benchmark_asymp_two_sided_pval_breakdown(
  delta = 0
)
```

Arguments:

`delta` Null treatment effect to test.

Returns: A named list: `fit_time`, `cache_time`, `pval_math_time`, `total_time` (all in seconds), `pval`, `beta_hat_T`, and `s_beta_hat_T`.

`InferenceOrdinalPartialProportionalOddsRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalPartialProportionalOddsRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Peterson, B., and Harrell, F. E. (1990). "Partial Proportional Odds Models for Ordinal Response Variables." *Journal of the Royal Statistical Society, Series C (Applied Statistics)*, 39(2), 205-217, doi:10.2307/2347760, for the partial (non-parallel-covariate) proportional-odds model fit here. McCullagh, P. (1980). "Regression Models for Ordinal Data." *Journal of the Royal Statistical Society, Series B*, 42(2), 109-142, doi:10.1111/j.25176161.1980.tb01109.x, for the full proportional-odds model this generalizes (see InferenceOrdinalPropOddsRegr).

InferenceOrdinalPropOddsRegr

Proportional Odds Regression Inference for Ordinal Responses

Description

Fits a proportional-odds (cumulative-logit) regression, via fast_ordinal_regression_with_var_cpp (see that page for the full model), for ordinal responses using the treatment indicator and, optionally, all recorded covariates as predictors. This is a full-likelihood class (likelihood_tier = "full") supporting score, gradient, and likelihood-ratio tests, plus parametric likelihood-ratio bootstrap calibration, in addition to Wald and resampling-based inference.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether compute_rand_two_sided_pval() is actually usable on this instance right now – FALSE exactly when it would stop(): an incidence-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see private$should_use_design_ra TRUE for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so InferenceSuite can check this before attempting the sentinel instead of relying on the stop() being silently swallowed into a pval = NA "ok" row – the single source of truth for both this check and compute_rand_two_sided_pval()'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceOrdinalPropOddsRegr

Methods

Public methods:

- InferenceOrdinalPropOddsRegr$approximate_randomization_distribution_beta_hat_T()
- InferenceOrdinalPropOddsRegr$supports_rand_pval_for_incidence()
- InferenceOrdinalPropOddsRegr$compute_rand_two_sided_pval()
- InferenceOrdinalPropOddsRegr$clone()

InferenceOrdinalPropOddsRegr$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceOrdinalPropOddsRegr$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

InferenceOrdinalPropOddsRegr$supports_rand_pval_for_incidence():

Usage:

```
InferenceOrdinalPropOddsRegr$supports_rand_pval_for_incidence()
```

InferenceOrdinalPropOddsRegr$compute_rand_two_sided_pval():

Usage:

```
InferenceOrdinalPropOddsRegr$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalPropOddsRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalPropOddsRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

McCullagh, P. (1980). "Regression Models for Ordinal Data." *Journal of the Royal Statistical Society, Series B*, 42(2), 109-142, doi:10.1111/j.25176161.1980.tb01109.x, for the proportional-odds cumulative-logit model fit here.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'ordinal')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(sample(1:4, 10, replace = TRUE))
inf = InferenceOrdinalPropOddsRegr$new(seq_des)
inf$compute_estimate()
```

```
inf$set_seed(1)
inf$compute_lik_ratio_bootstrap_two_sided_pval(delta = 0, B = 9, show_progress = FALSE)
```

`InferenceOrdinalRidit` *Ridit Analysis for Ordinal Responses*

Description

Performs Ridit analysis (Relative to an Identified Distribution unit) for comparing two groups on an ordinal scale. Every subject's category k is converted to a **ridit score** — its relative rank position within the **reference** distribution's empirical CDF: $r_k = F_{\text{ref}}(k-1) + \frac{1}{2} f_{\text{ref}}(k)$, where F_{ref} and f_{ref} are the reference group's empirical cumulative and point probabilities. The `reference` distribution — controlled by the `reference` constructor argument — may be the **control** arm (default), the **treatment** arm, or the **pooled** sample. The treatment effect is oriented as treated-versus-control stochastic superiority: with a control or pooled reference it is the treated mean ridit minus 0.5; with a treatment reference it is 0.5 minus the control mean ridit. The latter orientation is essential because the reference group's own mean ridit is 0.5 by construction. For a two-arm sample, the control- and treatment-reference formulas are the same empirical Mann-Whitney contrast, while retaining the requested group's scores for inspection. The mean ridit has a direct interpretation as (an estimate of) the probability that a randomly selected treated subject's outcome exceeds a randomly selected control subject's outcome (a Mann-Whitney-type stochastic superiority probability), similar in spirit to `InferenceOrdinalJonckheereTerpstraTest`'s superiority measure but referenced against a chosen distribution rather than always symmetric between the two arms. Standard errors and p-values come from `fast_ridit_analysis_cpp`'s asymptotic formula, not a resampling approximation.

Super class

`Inference` -> `InferenceOrdinalRidit`

Methods

Public methods:

- `InferenceOrdinalRidit$new()`
- `InferenceOrdinalRidit$compute_estimate()`
- `InferenceOrdinalRidit$compute_estimate_with_bootstrap_weights()`
- `InferenceOrdinalRidit$get_mean_ridit_treatment()`
- `InferenceOrdinalRidit$get_ridit_scores()`
- `InferenceOrdinalRidit$compute_asymp_confidence_interval()`
- `InferenceOrdinalRidit$compute_asymp_two_sided_pval()`
- `InferenceOrdinalRidit$clone()`

`InferenceOrdinalRidit$new()`: Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize a Ridit analysis inference object for a completed design with an ordinal, uncensored response.

Usage:

```
InferenceOrdinalRidit$new(
  des_obj,
  model_formula = NULL,
  reference = "control",
  verbose = FALSE,
  max_resample_attempts = 50L
)
```

Arguments:

`des_obj` A DesignSeqOneByOne object whose entire n subjects are assigned and response y is recorded within.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`reference` The group to use as the "Identified Distribution" when constructing ridit scores. Must be one of "control", "treatment", or "pooled". The reported contrast is always oriented so positive values mean higher treated outcomes: treated mean minus 0.5 for control/pooled, and 0.5 minus control mean for treatment. Default is "control".

`verbose` A flag indicating whether messages should be displayed.

`max_resample_attempts` Maximum number of times a single bootstrap replicate may be redrawn when the drawn sample fails validity screening. If all attempts fail the replicate is recorded as `NA`, silently reducing the effective `B`. Must be a positive integer. Default `50L`.

`InferenceOrdinalRidit$compute_estimate()`: Returns the estimated treated-versus-control stochastic superiority contrast. This is treated mean ridit minus 0.5 for a control/pooled reference and 0.5 minus control mean ridit for a treatment reference (see class documentation).

Usage:

```
InferenceOrdinalRidit$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

Returns: The numeric estimate.

`InferenceOrdinalRidit$compute_estimate_with_bootstrap_weights()`: Recomputes the ridit treatment estimate under subject/block bootstrap weights: the reference distribution's category proportions and every subject's ridit score are recomputed using the weights, then the oriented weighted ridit contrast is returned. Used by the Bayesian bootstrap and related weighted-resampling machinery. Always leaves the standard error and degrees of freedom unavailable (`NA`) regardless of `estimate_only` — this weighted path never computes the asymptotic variance.

Usage:

```
InferenceOrdinalRidit$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Subject-, block-, cluster-, or matched-set bootstrap weights.

`estimate_only` Present for interface parity; this method never computes variance components regardless of its value.

`InferenceOrdinalRidit$get_mean_ridit_treatment()`: Returns the mean ridit score among treated subjects (not centered). With a treatment reference this is 0.5 by construction; `$compute_estimate()` therefore uses the control mean instead.

Usage:

```
InferenceOrdinalRidit$get_mean_ridit_treatment()
```

Returns: The numeric Mean Ridit.

`InferenceOrdinalRidit$get_ridit_scores()`: Returns each subject's individual ridit score (see class documentation for the ridit-score formula), in subject order.

Usage:

```
InferenceOrdinalRidit$get_ridit_scores()
```

Returns: A numeric vector of scores.

`InferenceOrdinalRidit$compute_asymp_confidence_interval()`: Computes the asymptotic confidence interval for the oriented ridit treatment effect, using `fast_ridit_analysis_cpp`'s asymptotic standard error.

Usage:

```
InferenceOrdinalRidit$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level.

Returns: A numeric vector of length 2.

`InferenceOrdinalRidit$compute_asymp_two_sided_pval()`: Computes a two-sided Wald p-value for the oriented ridit contrast (i.e. `delta = 0` tests the null of no group difference), using `fast_ridit_analysis_cpp`'s asymptotic standard error.

Usage:

```
InferenceOrdinalRidit$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null value (centered at 0, so delta=0 means Ridit=0.5).

Returns: The p-value.

`InferenceOrdinalRidit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalRidit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Bross, I. D. J. (1958). "How to Use Ridit Analysis." *Biometrics*, 14(1), 18-38, doi:10.2307/2527727, for the ridit transformation and its interpretation used here.

Examples

```
set.seed(1)
x_dat <- data.frame(
  x1 = c(-1.2, -0.7, -0.2, 0.3, 0.8, 1.3, 1.8, 2.3),
  x2 = c(0, 1, 0, 1, 0, 1, 0, 1)
)
seq_des <- DesignSeqOneByOneBernoulli$new(n = nrow(x_dat), response_type = "ordinal",
  verbose = FALSE)
for (i in seq_len(nrow(x_dat))) {
  seq_des$add_one_subject_to_experiment_and_assign(x_dat[i, , drop = FALSE])
}
seq_des$add_all_subject_responses(as.integer(c(1, 2, 2, 3, 3, 4, 4, 5)))
infer <- InferenceOrdinalRidit$
  new(seq_des, verbose = FALSE)
infer
```

InferenceOrdinalStereotypeLogitRegr

Stereotype Logit Regression Inference for Ordinal Responses

Description

Fits Anderson's (1984) stereotype logit model for ordinal responses (see `fast_stereotype_logit_cpp` for the full reduced-rank multinomial-softmax formula and reparameterization): a single linear predictor $\eta_i = \beta_T W_i + X_i^\top \gamma$ is scaled by a category-specific **score** $\phi_k \in [0, 1]$ (jointly estimated, monotone in k) in a softmax over all K categories, rather than assuming a single proportional/parallel effect across cuts as `InferenceOrdinalContRatioRegr`/ `InferenceOrdinalKKCondAdjCatLogitRegr` do. This makes the stereotype model a genuinely more flexible (multinomial-logit-like, reduced-rank) alternative to the standard proportional-odds/adjacent-category/continuation-ratio ordinal families, at the cost of a less directly interpretable treatment coefficient (β_T enters multiplicatively through the ϕ_k scores rather than as a single additive log-odds-ratio). `likelihood_tier = "full"`: likelihood-ratio, score, gradient, and Wald tests are all available when the model converges, plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok"

row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

`compute_lik_ratio_two_sided_pval()` **can be anti-conservative at small n.** The stereotype model's category-score parameters ϕ_k are a Davies (1977) non-regular case, not identified when β_T is at/near zero – exactly the neighborhood every null hypothesis test sits in – so the LR statistic's null distribution need not be the chi-square(1) this method assumes. The size of the problem depends on the constrained refit: before 2026-09-21 the delta-constrained refit started cold and could stall in a much worse local optimum (e.g. neg-log-likelihood 131.3 vs the full fit's 118.9 at δ equal to the MLE itself), which inflated the LR statistic and produced ~18-23% Type-I error at $n = 100$; it also collapsed the bootstrap and inverted-LR confidence intervals to zero width. The refit now also starts from the full-fit parameters and keeps the better fit. Measured after that change (400 simulated null datasets, 5 response categories, a continuous covariate): about 6.5% Type-I error and 0.93 confidence-interval coverage at $n = 100$, with no zero-width intervals, and about 13% Type-I error at $n = 50$ with 3 categories (169 converged fits). Part of the small-sample excess comes from the likelihood being multimodal: in a few percent of $n = 50$ fits the reported point estimate sits at a local optimum whose likelihood is below that of another optimum, which makes the LR statistic at the estimate positive rather than zero. For small samples, prefer the better-calibrated alternatives on this class: `compute_lik_ratio_bootstrap_two_sided_pval()` (parametric-bootstrap calibration via the class's own null simulation; ~6-7% Type-I error) and the cheaper `compute_lik_ratio_bartlett_two_sided_pval()` (Monte-Carlo Bartlett correction; ~4% Type-I error); those two figures predate the refit change and were not re-measured. Both are roughly 20-40x slower per call than the raw chi-square test (a fresh bootstrap/Monte-Carlo refit at every delta candidate), which is why they are excluded from this package's own routine comprehensive test suite (see `comprehensive_slow_paths.R`). `compute_lik_ratio_two_sided_pval()` itself is left unchanged (not silently recalibrated) to avoid an undocumented behavior change to an existing method's contract. Bayesian-bootstrap inference is temporarily unavailable because the current non-uniform weighted hook fits a cumulative-logit surrogate rather than the stereotype likelihood. It will remain disabled until the native weighted stereotype-logit backend described in the package implementation plan lands.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceOrdinalStereotypeLogitRegr`

Methods

Public methods:

- InferenceOrdinalStereotypeLogitRegr$approximate_randomization_distribution_beta_hat_T()
- InferenceOrdinalStereotypeLogitRegr$supports_rand_pval_for_incidence()
- InferenceOrdinalStereotypeLogitRegr$compute_rand_two_sided_pval()
- InferenceOrdinalStereotypeLogitRegr$clone()

`InferenceOrdinalStereotypeLogitRegr$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceOrdinalStereotypeLogitRegr$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalStereotypeLogitRegr$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceOrdinalStereotypeLogitRegr$supports_rand_pval_for_incidence()
```

`InferenceOrdinalStereotypeLogitRegr$compute_rand_two_sided_pval()`:

Usage:

```
InferenceOrdinalStereotypeLogitRegr$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceOrdinalStereotypeLogitRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceOrdinalStereotypeLogitRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Anderson, J. A. (1984). "Regression and Ordered Categorical Variables." *Journal of the Royal Statistical Society, Series B*, 46(1), 1-30, doi:10.1111/j.25176161.1984.tb01276.x, for the stereotype logit model.

See Also

`InferenceOrdinalContRatioRegr` for a proportional (non-reduced-rank) ordinal alternative. See also: Ordinal regression (Wikipedia).

InferencePropBetaRegr *Beta Regression Inference for Proportion Responses*

Description

Fits Ferrari and Cribari-Neto's (2004) beta regression for proportion responses $Y_i \in (0, 1)$: $\text{logit}(E[Y_i \mid w_i, x_i]) = \beta_0 + \beta_T w_i + x_i^\top \gamma$, $Y_i \mid w_i, x_i \sim \text{Beta}(\mu_i \phi, (1 - \mu_i)\phi)$ for fitted mean μ_i and a single (constant, not covariate-dependent) precision parameter ϕ, by maximum likelihood (`fast_beta_regression_cpp`/`fast_beta_regression_weighted_cpp`). $\hat{\beta}_T$ is a log-odds-ratio on the conditional-mean scale: $\exp(\hat{\beta}_T)$ is the odds ratio for the expected proportion. Unlike `InferencePropFractionalLogit`'s quasi-likelihood (which specifies only the conditional mean), beta regression also specifies the conditional variance/shape via ϕ — a correctly specified beta model yields a fully efficient likelihood-based fit and genuine likelihood-ratio/score/gradient tests, at the cost of requiring the beta-distribution shape assumption to actually hold. `likelihood_tier = "full"`: likelihood-ratio, score, gradient, and Wald tests are all available when the model converges, plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test. Y_i values of exactly 0 or 1 are not supported by the beta density and are handled by `sanitize_beta_response()`'s boundary adjustment before fitting.

Estimand. Composes `MarginalEstimand` (`set_estimand()`/`get_estimand()`/`get_supported_estimands()`). Under the default `estimand = "conditional"`, $\hat{\beta}_T$ is the log-odds-ratio above. Under `estimand = "marginal_mean_diff"`, the reported quantity is instead the g-computation marginal mean difference $\frac{1}{n}\sum_i \{\text{plogis}(\hat{\beta}_0 + \hat{\beta}_T + X_i^\top \hat{\gamma}) - \text{plogis}(\hat{\beta}_0 + X_i^\top \hat{\gamma})\}$ (the precision parameter ϕ does not enter the mean, so it plays no role in this functional). Only `"marginal_mean_diff"` is supported — a ratio of two mean proportions, both bounded in $[0, 1]$, is not the standard estimand for a beta-regression treatment effect the way a rate ratio is for count data. Because there is no latent submodel for this family (unlike e.g. `InferencePropZeroOneInflatedBetaRegr`'s zero/one-inflation mixture), the marginal mean function is exactly the model's own fitted mean; no separate standardization step beyond the g-computation average is needed. Standard errors under the marginal estimand use the delta method against the mean-submodel coefficient covariance (degrees of freedom `Inf`); `testing_type` is restricted to `"wald"` whenever the estimand is non-conditional. The underlying model fit is identical regardless of estimand — switching `estimand` is a pure post-fit transform, never a refit.

Super class

`Inference` -> `InferencePropBetaRegr`

Methods

Public methods:

- `InferencePropBetaRegr$new()`
- `InferencePropBetaRegr$compute_estimate()`
- `InferencePropBetaRegr$compute_asymp_confidence_interval()`
- `InferencePropBetaRegr$compute_asymp_two_sided_pval()`
- `InferencePropBetaRegr$compute_estimate_with_bootstrap_weights()`
- `InferencePropBetaRegr$clone()`

`InferencePropBetaRegr$new()`: Initialize inference for the beta regression model $\mathrm{logit}(E[Y_i \mid w_i, x_i]) = \beta_0 + \beta_T w_i + x_i^\top \gamma$, $Y_i \sim \mathrm{Beta}(\mu_i \phi, (1-\mu_i)\phi)$; see `InferencePropBetaRegr` for the model form. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferencePropBetaRegr$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL,
  optimization_alg = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a proportion response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values by default.

`optimization_alg` Character scalar specifying the optimization algorithm. Default is dispatched via policy.

`InferencePropBetaRegr$compute_estimate()`: Fits the beta regression model by maximum likelihood (jointly estimating the mean coefficients and the precision parameter ϕ). Under the default `estimand = "conditional"`, returns the log-odds-ratio estimate $\hat{\beta}_T$ on the conditional-mean scale. Under `estimand = "marginal_mean_diff"` (set via `set_estimand()`), returns the g-computation marginal mean difference instead — see the class-level `@details` for the formula. The underlying model fit is identical either way (a pure post-fit transform of the same cached fit, no refit).

Usage:

```
InferencePropBetaRegr$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip standard-error computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferencePropBetaRegr$compute_asymp_confidence_interval()`: Wald confidence interval, dispatched by `testing_type` for the conditional estimand (score/gradient/ likelihood-ratio/Bartlett available; see `InferenceAsympLik`); under a marginal estimand `testing_type` is always `"wald"` (the only value `set_estimand()` permits there), so this always resolves to the delta-method interval. Calls `self$compute_estimate()` first (not `private$shared()` directly) so the estimand-aware cache is always current regardless of call order.

Usage:

```
InferencePropBetaRegr$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Two-sided miscoverage rate; the returned interval targets `1 - alpha` coverage.

`InferencePropBetaRegr$compute_asymp_two_sided_pval()`: Wald two-sided p-value, dispatched by `testing_type` exactly as `compute_asymp_confidence_interval()`; see that method's description for the marginal-estimand always-Wald note.

Usage:

```
InferencePropBetaRegr$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment-effect value under the current estimand (conditional log-odds-ratio, or marginal mean difference).

`InferencePropBetaRegr$compute_estimate_with_bootstrap_weights()`: Refits the beta model with subject/block-level weights applied to the fitting log-likelihood (Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded to row level via `private$expand_subject_or_block_weights_to_row` via `fast_beta_regression_weighted_cpp`, and returns the reweighted estimate $\hat{\beta}_T^{(w)}$. Uses the same QR column-dropping hardening as `compute_estimate()`; a hardened-but-still-unreasonable fit is cached as nonestimable.

Usage:

```
InferencePropBetaRegr$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferencePropBetaRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferencePropBetaRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Ferrari, S., and Cribari-Neto, F. (2004). "Beta regression for modelling rates and proportions." *Journal of Applied Statistics*, 31(7), 799-815, doi:10.1080/0266476042000214501.

See Also

`InferencePropFractionalLogit` for a quasi-likelihood proportion model that specifies only the conditional mean. Comparable Python API: no direct beta-regression equivalent in **statsmodels**; see statsmodels GLM for the general exponential-family GLM framework. See also: Beta distribution (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'proportion')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferencePropBetaRegr$new(seq_des)
inf$compute_estimate()
```

InferencePropFractionalLogit

Fractional Logit Inference for Proportion Responses

Description

Fits Papke and Wooldridge's (1996) fractional logistic (quasi-binomial) regression for proportion responses $Y_i \in [0, 1]$ (not restricted to $\{0, 1\}$): $E[Y_i \mid w_i, x_i] = \text{logit}^{-1}(\beta_0 + \beta_T w_i + x_i^{\top}\gamma)$, fit by maximizing the Bernoulli quasi-log-likelihood $\sum_i \{Y_i \log \mu_i + (1-Y_i)\log(1-\mu_i)\}$ treating Y_i as if it were binary (a valid estimating equation for the conditional mean even though Y_i is fractional — the Bernoulli log-likelihood's score is unbiased for the true mean regardless of the actual distribution of Y_i on $[0, 1]$). $\hat{\beta}_T$ is a log-odds-ratio on the conditional-mean scale: $\exp(\hat{\beta}_T)$ is the odds ratio for the expected proportion. Standard errors use the model-based (non-robust/non-sandwich) Fisher information from this quasi-likelihood, scaled by an estimated quasi-binomial dispersion parameter $\hat{\phi}$ (Papke & Wooldridge's own prescription; fixed 2026-09-06 – the unscaled Bernoulli-based variance systematically overstates $\text{Var}(\hat{\beta}_T)$ for a genuinely fractional response, since Bernoulli is the maximum-variance distribution on [0,1] for a given mean); only Wald inference is exposed (`private$supports_likelihood_tests()` is hard `FALSE` here even though `likelihood_tier = "full"` metadata is set for component-composition purposes — this class deliberately does not compose `ParametricLikelihoodBootstrap`, so no likelihood-ratio/score/gradient test surface is exposed). Validity requires that the conditional mean is correctly specified on the logit scale; unlike beta regression, no assumption is made about the conditional variance or shape of Y_i's distribution.

Super class

`Inference` -> `InferencePropFractionalLogit`

Methods

Public methods:

- `InferencePropFractionalLogit$new()`
- `InferencePropFractionalLogit$compute_estimate()`
- `InferencePropFractionalLogit$compute_estimate_with_bootstrap_weights()`
- `InferencePropFractionalLogit$clone()`

`InferencePropFractionalLogit$new()`: Initialize inference for the fractional logit model $E[Y_i \mid w_i, x_i] = \mathrm{logit}^{-1}(\beta_0 + \beta_T w_i + x_i^\top \gamma)$; see `InferencePropFractionalLogit` for the model form. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferencePropFractionalLogit$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  harden = TRUE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a proportion response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`harden` Whether to apply robustness measures.

`smart_cold_start_default` Whether to use smart cold start values.

`InferencePropFractionalLogit$compute_estimate()`: Fits the fractional logit model by maximizing the Bernoulli quasi-log-likelihood on the fractional response and returns the log-odds-ratio estimate $\hat{\beta}_T$. When `estimate_only = TRUE` and hardening is disabled (`harden = FALSE`), uses a fast path via base R's `glm.fit(family = quasibinomial())` instead of the package's own fitting routine; otherwise dispatches through the shared hardened-fit path.

Usage:

```
InferencePropFractionalLogit$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations; when combined with `harden = FALSE`, also switches to the `quasibinomial()` fast path.

`InferencePropFractionalLogit$compute_estimate_with_bootstrap_weights()`: Refits the fractional logit model with subject/block-level weights applied to the fitting quasi-log-likelihood (Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded to row level via `private$expand_subject_or_b`
and returns the reweighted estimate $\hat{\beta}_T^{(w)}$. Uses the same QR column-dropping hardening as `compute_estimate()`'s hardened path; a hardened-but-still-unreasonable fit is cached as nonestimable.

Usage:

```
InferencePropFractionalLogit$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferencePropFractionalLogit$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferencePropFractionalLogit$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Papke, L. E., and Wooldridge, J. M. (1996). "Econometric Methods for Fractional Response Variables with an Application to 401(K) Plan Participation Rates." *Journal of Applied Econometrics*, 11(6), 619-632, doi:10.1002/(SICI)10991255(199611)11:6<619::AIDJAE418>3.0.CO;21.

See Also

`InferencePropBetaRegr` for a proportion model that also specifies the conditional variance/shape. Comparable Python API: statsmodels GLM (`family=Binomial()` on fractional response data). See also: Logistic regression (Wikipedia).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'proportion')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferencePropFractionalLogit$new(seq_des)
inf$compute_estimate()
```

`InferencePropGCompMeanDiff`

G-Computation Mean-Difference Inference for Proportion Responses

---

Description

Fits a fractional-logit working model, $\text{logit}\, E[Y_i \mid x_i] = x_i^\top \hat{\beta}$ (via `fast_logistic_regression_cpp`, treating the continuous-in-$(0,1)$ proportion as a quasi-binomial mean model — the same mean-model idea as `InferencePropFractionalLogit`), for a proportion outcome using treatment and, optionally, all recorded covariates, then estimates the marginal mean difference by G-computation: standardizing predicted mean proportions under all-treated and all-control assignments over the empirical covariate distribution (see `gcomp_fractional_logit_point_estimate_cpp` for the exact standardization formula). Inference uses a Huber-White sandwich-robust covariance for the regression coefficients and the **delta method** (analytic gradient of the standardized mean-difference

functional with respect to $\hat{\beta}$, by default) to propagate that covariance onto the mean-difference scale, $\widehat{\mathrm{Var}}(\widehat{\mathrm{md}}) = \nabla^\top \widehat{\mathrm{Var}}(\hat{\beta})\nabla$.

The implementation is optimized for resampling-based inference. It utilizes a fast C++ IRLS solver for the underlying fractional logit regression. During resampling draws, it bypasses the calculation of the sandwich covariance matrix and delta-method standard errors, providing a significant speedup when computing bootstrap or randomization distributions.

Variance fallback cascade. If the primary analytic-gradient/sandwich-covariance variance is non-finite (e.g. near-boundary fitted probabilities), up to eight progressively more conservative fallback strategies are tried in order (see the `variance_fallback_methods` constructor argument for the full list and their individual definitions): stabilized (PSD-projected) sandwich covariance, model-based (Fisher information) covariance, finite-difference gradients in place of the analytic delta-method gradient, and combinations with progressively stronger probability clipping. The first strategy in the ordered list that yields a finite, positive variance is used; an empty `variance_fallback_methods` vector always returns `NA` variance rather than erroring.

Super class

`Inference` -> `InferencePropGCompMeanDiff`

Methods

Public methods:

- `InferencePropGCompMeanDiff$new()`
- `InferencePropGCompMeanDiff$compute_estimate()`
- `InferencePropGCompMeanDiff$get_standard_error()`
- `InferencePropGCompMeanDiff$compute_estimate_with_bootstrap_weights()`
- `InferencePropGCompMeanDiff$compute_asymp_confidence_interval()`
- `InferencePropGCompMeanDiff$compute_asymp_two_sided_pval()`
- `InferencePropGCompMeanDiff$compute_wald_two_sided_pval()`
- `InferencePropGCompMeanDiff$compute_wald_confidence_interval()`
- `InferencePropGCompMeanDiff$compute_bootstrap_two_sided_pval()`
- `InferencePropGCompMeanDiff$compute_bootstrap_confidence_interval()`
- `InferencePropGCompMeanDiff$approximate_bootstrap_distribution_beta_hat_T()`
- `InferencePropGCompMeanDiff$clone()`

`InferencePropGCompMeanDiff$new()`**:** Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize the g-computation inference object.

Usage:

```
InferencePropGCompMeanDiff$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  prob_clip_eps = 1e-06,
  prob_clip_strong_eps = 1e-04,
  max_resample_attempts = 50L,
```

```
  smart_cold_start_default = NULL,
  harden = TRUE,
  variance_fallback_methods = c("robust", "stabilized_robust", "model_based",
   "stabilized_robust_fd", "model_based_fd", "stabilized_robust_strong_clip",
    "model_based_strong_clip", "model_based_fd_strong_clip")
)
```

Arguments:

`des_obj` A completed `DesignSeqOneByOne` object with a proportion response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` Whether to print progress messages.

`prob_clip_eps` Primary probability clamp applied to fitted values during model-based variance computation. Predicted probabilities are clipped to `[prob_clip_eps, 1 - prob_clip_eps]` before computing IWLS weights. Must be in `[0, 0.5)`. Default `1e-6`.

`prob_clip_strong_eps` Stronger clamp used as a fallback when the primary variance strategy fails. Predicted probabilities and gradients are clipped to `[prob_clip_strong_eps, 1 - prob_clip_strong_eps]`. Must be in `[0, 0.5)` and should be $\geq$ `prob_clip_eps`. Default `1e-4`.

`max_resample_attempts` Maximum number of times a single bootstrap replicate may be redrawn when the drawn sample fails validity screening (e.g. near-perfect separation, too few observations per arm, excessive boundary mass). If all attempts fail the replicate is recorded as `NA`, silently reducing the effective `B`. Can be overridden per-call in `approximate_bootstrap_distribution_beta` Must be a positive integer. Default `50L`.

`smart_cold_start_default` Whether to use smart cold start values.

`harden` Whether to apply robustness measures.

`variance_fallback_methods` Ordered character vector of variance strategies to attempt in sequence. Each name corresponds to a (gradient, covariance-matrix) pair; the first strategy that yields a finite, positive variance is used. Allowed values (in their default order) are:

- `"robust"` Analytic delta-method gradient with the sandwich (HC) covariance.
- `"stabilized_robust"` Same gradient; covariance projected to the nearest PSD matrix.
- `"model_based"` Same gradient; Fisher-information (IWLS) covariance. Fitted probabilities are first clipped to `[prob_clip_eps, 1 - prob_clip_eps]`, then the binomial variance weight $w_i = \hat{\mu}_i(1 - \hat{\mu}_i)$ is further capped at `0.25`. The cap is the global maximum of $p(1-p)$, attained at $p = 0.5$; it prevents a near-boundary fitted probability that slips through the clip from inflating the information matrix, which is mathematically correct because no Bernoulli variance can exceed 0.25.
- `"stabilized_robust_fd"` Finite-difference (central-difference) gradient; stabilized sandwich covariance. The step size for coefficient j is $h_j = \varepsilon^{1/3}(|\hat{\beta}_j| + 1)$, where $\varepsilon =$ `.Machine$double.eps`. The cube-root of machine epsilon is the theoretically optimal step that balances truncation error ($O(h^2)$ for central differences) against floating-point cancellation ($O(\varepsilon/h)$), giving a total error of $O(\varepsilon^{2/3})$.
- `"model_based_fd"` Finite-difference gradient (same step rule as `"stabilized_robust_fd"`); model-based covariance with the 0.25 weight cap.
- `"stabilized_robust_strong_clip"` Strong-clipped analytic gradient; stabilized sandwich covariance.

"model_based_strong_clip" Strong-clipped analytic gradient; model-based covariance (strong-clipped) with the 0.25 weight cap.

"model_based_fd_strong_clip" Strong-clipped finite-difference gradient (same step rule); model-based covariance (strong-clipped) with the 0.25 weight cap.

Pass a shorter vector or a single string to restrict which strategies are tried. An empty vector always returns NA variance.

InferencePropGCompMeanDiff$compute_estimate(): Computes the g-computation treatment-effect estimate.

Usage:

```
InferencePropGCompMeanDiff$compute_estimate(estimate_only = FALSE)
```

Arguments:

estimate_only If TRUE, skip variance component calculations.

InferencePropGCompMeanDiff$get_standard_error(): Returns the standard error of the g-computation mean-difference estimate (NA if it is unavailable).

Usage:

```
InferencePropGCompMeanDiff$get_standard_error()
```

Returns: A single numeric standard error, or NA_real_.

InferencePropGCompMeanDiff$compute_estimate_with_bootstrap_weights(): Recomputes the g-computation mean-difference estimate under the supplied subject- or block-level bootstrap weights and caches it; used by the Bayesian-bootstrap and resampling paths.

Usage:

```
InferencePropGCompMeanDiff$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

subject_or_block_weights Numeric vector of bootstrap weights, one per subject (or per block when the design is blocked).

estimate_only If TRUE, only the point estimate is required (no variance).

Returns: The weighted mean-difference estimate, or NA_real_ if the weighted fit is unusable.

InferencePropGCompMeanDiff$compute_asymp_confidence_interval(): Computes a 1 - alpha confidence interval.

Usage:

```
InferencePropGCompMeanDiff$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

alpha The confidence level in the computed confidence interval is 1 - alpha.

InferencePropGCompMeanDiff$compute_asymp_two_sided_pval(): Uses the shared asymptotic two-sided p-value contract; see InferenceAsymp.

Usage:

```
InferencePropGCompMeanDiff$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null mean difference. Defaults to 0.

`InferencePropGCompMeanDiff$compute_wald_two_sided_pval():` Computes a Wald two-sided p-value for the treatment effect.

Usage:

```
InferencePropGCompMeanDiff$compute_wald_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null mean difference. Defaults to 0.

`InferencePropGCompMeanDiff$compute_wald_confidence_interval():` Computes a Wald confidence interval for the treatment effect.

Usage:

```
InferencePropGCompMeanDiff$compute_wald_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The significance level. Default 0.05.

`InferencePropGCompMeanDiff$compute_bootstrap_two_sided_pval():` Computes a bootstrap two-sided p-value for the treatment effect.

Usage:

```
InferencePropGCompMeanDiff$compute_bootstrap_two_sided_pval(
  delta = 0,
  B = 501,
  type = "symmetric",
  na.rm = FALSE,
  boundary_tol = 0.02,
  max_boundary_mass = 0.95,
  sep_tol = 0.02,
  min_group_n = 5L,
  show_progress = TRUE,
  min_number_usable_samples = 5L
)
```

Arguments:

`delta` The null mean difference. Defaults to 0.

`B` Number of bootstrap samples.

`type` Bootstrap p-value type. See `InferenceNonParamBootstrap$compute_bootstrap_two_sided_pval`.

`na.rm` Whether to remove non-finite bootstrap replicates.

`boundary_tol` Resample screening threshold for boundary mass near 0/1.

`max_boundary_mass` Reject a resample when at least this fraction is near the boundary.

`sep_tol` Separation tolerance used to reject nearly perfectly separated resamples.

`min_group_n` Minimum number of observations required in each treatment arm.

`show_progress` Whether to show a progress bar.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`InferencePropGCompMeanDiff$compute_bootstrap_confidence_interval()`: Generic (non-screening-modified) bootstrap two-sided p-value, aliased directly from ‘InferenceNonParamBootstrap‘ so ‘compute_bootstrap_two_sided_pval()‘ can dispatch to it without relying on ‘super$‘, which does not resolve under flat component composition.

Computes a bootstrap confidence interval.

Usage:

```
InferencePropGCompMeanDiff$compute_bootstrap_confidence_interval(
  alpha = 0.05,
  B = 501,
  type = NULL,
  na.rm = TRUE,
  show_progress = TRUE,
  boundary_tol = 0.02,
  max_boundary_mass = 0.95,
  sep_tol = 0.02,
  min_group_n = 5L,
  min_number_usable_samples = 5L
)
```

Arguments:

`alpha` The confidence level 1 - `alpha`.

`B` Number of bootstrap samples.

`type` Bootstrap CI type.

`na.rm` Whether to remove non-finite bootstrap replicates.

`show_progress` Whether to show bootstrap progress.

`boundary_tol` Resample screening threshold for boundary mass near 0/1.

`max_boundary_mass` Reject a resample when at least this fraction is near the boundary.

`sep_tol` Separation tolerance used to reject nearly perfectly separated resamples.

`min_group_n` Minimum number of observations required in each treatment arm.

`min_number_usable_samples` Minimum number of finite bootstrap samples required.

`InferencePropGCompMeanDiff$approximate_bootstrap_distribution_beta_hat_T()`: Generic (non-screening-modified) bootstrap confidence interval; see ‘compute_bootstrap_two_sided_pval_generic‘.

Abbreviated bootstrap sampler that reuses a bootstrap worker.

Usage:

```
InferencePropGCompMeanDiff$approximate_bootstrap_distribution_beta_hat_T(
  B = 501,
  show_progress = TRUE,
  max_resample_attempts = NULL,
  boundary_tol = 0.02,
  max_boundary_mass = 0.95,
  sep_tol = 0.02,
  min_group_n = 5L,
  debug = FALSE
)
```

Arguments:

B The number of bootstrap samples (default 501).

show_progress Whether to show a progress bar.

max_resample_attempts Maximum redraw attempts per bootstrap replicate before the replicate is recorded as NA. NULL (default) uses the value set at construction time.

boundary_tol Resample screening threshold for boundary mass near 0/1.

max_boundary_mass Reject a resample when at least this fraction is near the boundary.

sep_tol Separation tolerance used to reject nearly perfectly separated resamples.

min_group_n Minimum number of observations required in each treatment arm.

debug If TRUE, return per-replicate diagnostics (values, errors, warnings) instead of just the bootstrap values.

InferencePropGCompMeanDiff$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferencePropGCompMeanDiff$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'proportion')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferencePropGCompMeanDiff$new(seq_des)
inf$compute_estimate()
```

InferencePropKKGEE *GEE Inference for KK Designs with Proportion Response*

Description

Fits a Generalized Estimating Equations model with a binomial (quasi-likelihood, fractional-response) family and logit link, $\operatorname{logit} E[Y_i \mid x_i] = x_i^\top \beta$, for proportion (continuous values in (0, 1)) responses under a KK matching-on-the-fly design — the same fractional-logit mean-model idea as InferencePropFractionalLogit, extended to jointly account for matched-pair and reservoir clustering via GEE. Each GEE cluster is either a matched pair (2 members) or a reservoir singleton (1 member), with an **exchangeable** working correlation structure — see $compute_estimate()'s method-level documentation for the full fitting contract (internal Rcpp solver vs. **geepack** fallback, hardening/retry behavior). Inference is quasi-likelihood/estimating-equation based (likelihood_tier = "quasi"): standard errors are GEE sandwich (robust) standard errors, not model-likelihood-based.

Super class

`Inference` -> `InferencePropKKGEE`

Methods

Public methods:

- `InferencePropKKGEE$new()`
- `InferencePropKKGEE$clone()`

`InferencePropKKGEE$new()`: Initialize KK proportion-response GEE inference, validate the matched/reservoir design, and prepare the exchangeable-working-correlation fractional-logit GEE fitting machinery used by `InferencePropKKGEE`.

Usage:

```
InferencePropKKGEE$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a proportion response.

`model_formula` Optional formula for covariate adjustment.

`use_rcpp` Whether to use the internal Rcpp GEE solver (`TRUE`, default) with automatic fallback to `geepack::geeglm` on failure, or always use `geepack::geeglm` directly (`FALSE`, requires **geepack** to be installed).

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`InferencePropKKGEE$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferencePropKKGEE$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Liang, K.-Y., and Zeger, S. L. (1986). "Longitudinal Data Analysis Using Generalized Linear Models." *Biometrika*, 73(1), 13-22, doi:10.1093/biomet/73.1.13, for the GEE estimating-equation framework and sandwich variance estimator used here.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'proportion')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferencePropKKGEE$new(seq_des)
inf$compute_estimate()
```

InferencePropKKGLMM *KK GLMM Inference for Proportion Responses*

Description

Fits a combined conditional-logit-plus-random-intercept-GLMM likelihood for proportion responses under a KK matching-on-the-fly design. Matched pairs with a discordant pair-difference are handled by a conditional (fixed pair-effect) logistic term, while concordant/reservoir subjects are handled by a random-intercept logistic mixed model with intercept $b_g \sim N(0, \sigma_b^2)$ per matched-set/reservoir group g; both terms share the same treatment coefficient β_T, jointly maximized by the internal `fast_clogit_plus_glmm_cpp` routine. This combines the design-exact conditional-logit treatment of matched pairs (no nuisance pair-intercept to estimate) with a GLMM's ability to still contribute information from concordant pairs and reservoir subjects, which a pure conditional-logit-on-discordant-pairs-only approach would discard. $\exp(\hat{\beta}_T)$ is the common treatment odds ratio. `likelihood_tier = "full"`: likelihood-ratio, score, and Wald tests are all available when the model converges. See `InferenceAbstractKKCondLogitGLMM` for the shared model-fitting and caching contract used by this class's incidence-response siblings (`InferenceIncidKKCondLogitGLMMIVWC`, `InferenceIncidKKCondLogitGLMMOneLik`).

Super classes

`Inference` -> `InferenceAbstractKKCondLogitGLMM` -> `InferencePropKKGLMM`

Methods

Public methods:

- `InferencePropKKGLMM$new()`
- `InferencePropKKGLMM$clone()`

`InferencePropKKGLMM$new()`**:** Initialize inference for the combined conditional-logit (discordant matched pairs) plus random-intercept-GLMM (concordant pairs/reservoir) proportion model; see `InferencePropKKGLMM` for the model form. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferencePropKKGLMM$new(
  des_obj,
  model_formula = NULL,
  max_abs_reasonable_coef = 10000,
  max_abs_log_sigma = 8,
  verbose = FALSE,
  smart_cold_start_default = NULL,
  optimization_alg = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a proportion response.

`model_formula` Optional formula for covariate adjustment.

`max_abs_reasonable_coef` Cap for reasonable coefficient estimates.

`max_abs_log_sigma` Cap for reasonable log random effect variance.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`optimization_alg` Character. Optimization algorithm (default "lbfgs").

`InferencePropKKGLMM$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferencePropKKGLMM$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kapelner, A. and Krieger, A. M. (2014). "Matching on-the-fly: Sequential allocation with higher power and efficiency." *Biometrics*, 70(2), 378-388, doi:10.1111/biom.12148, for the KK matching-on-the-fly design this class is built for; Breslow, N. E., and Clayton, D. G. (1993). "Approximate Inference in Generalized Linear Mixed Models." *Journal of the American Statistical Association*, 88(421), 9-25, doi:10.2307/2290687, for the GLMM likelihood framework combined with the conditional-logit term here.

`InferencePropKKQuantileRegrIVWC`

Quantile Regression Compound Estimator for KK Matching-on-the-Fly Designs (Proportion Outcomes)

Description

A variance-weighted compound quantile regression estimator for KK matching-on-the-fly designs with proportion responses. Inference is performed on the **logit (log-odds) scale**: responses $y \in (0, 1)$ are transformed via $\text{logit}(y) = \log(y/(1-y))$ before quantile regression.

The estimator combines:

1. Quantile regression on logit-scale within-pair differences $\text{logit}(y_T) - \text{logit}(y_C)$ (matched pairs)
2. Quantile regression of $\text{logit}(y)$ on treatment and covariates (reservoir)

using the same variance-weighted combination logic as the OLS compound estimator.

The estimated treatment effect is a **log-odds-ratio shift** at quantile `tau`. At `beta_T = 1` (one log-odds-ratio unit of treatment effect), the population treatment effect on the logit scale is exactly 1, so no `skip_ci` is needed.

Default quantile: `tau = 0.5` **(median regression).** To target a different quantile — for example the 25th or 75th percentile — pass `tau = 0.25` or `tau = 0.75` to the constructor:

```
inf = InferencePropKKQuantileRegrIVWC$
new(seq_des, tau = 0.75)
```

Any value strictly between 0 and 1 is accepted.

Standard errors use Powell's "nid" sandwich estimator (non-iid), falling back to "iid" on failure. Asymptotic z-based inference is used throughout.

This class requires the **quantreg** package, which is listed in Suggests and is not installed automatically with **EDI**. Install **quantreg** before using this class.

Legacy class. Not fully tested in `comprehensive_tests.R`.

Super class

`Inference` -> `InferencePropKKQuantileRegrIVWC`

Methods

Public methods:

- `InferencePropKKQuantileRegrIVWC$new()`
- `InferencePropKKQuantileRegrIVWC$clone()`

`InferencePropKKQuantileRegrIVWC$new()`**:** Initialize proportion-response KK IVWC quantile-regression inference on the logit response scale; see `InferencePropKKQuantileRegrIVWC`.

Usage:

```
InferencePropKKQuantileRegrIVWC$new(
  des_obj,
  model_formula = NULL,
  tau = 0.5,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A DesignSeqOneByOne object whose entire n subjects are assigned and response y is recorded within.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`tau` The quantile level for regression on the logit scale, strictly between 0 and 1. The default `tau = 0.5` estimates the median log-odds-ratio treatment effect. Pass a different value (e.g. `tau = 0.25` or `tau = 0.75`) to target a different percentile of the treatment effect distribution.

`verbose` A flag indicating whether messages should be displayed to the user. Default is `FALSE`.

`smart_cold_start_default` Whether to use smart cold start values.

`InferencePropKKQuantileRegrIVWC$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferencePropKKQuantileRegrIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'proportion')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferencePropKKQuantileRegrIVWC$new(seq_des)
inf$compute_estimate()
```

`InferencePropKKQuantileRegrOneLik` *Quantile Regression Combined-Likelihood Compound Estimator for KK Designs (Proportion)*

Description

Fits the combined stacked quantile regression (matched-pair differences + reservoir) using the treatment indicator and all recorded covariates for proportion responses. Responses $y \in (0, 1)$ are transformed via $\mathrm{logit}(y) = \log(y/(1-y))$ before regression; the estimated treatment effect $\hat{\beta}_T$ is a log-odds-ratio shift at quantile `tau` of the logit-transformed response. Minimizes the joint check-function (pinball) loss $\rho_\tau(u) = u(\tau - \mathbb{1}\{u < 0\})$ over both data sources simultaneously in one **quantreg** fit, unlike the `IVWC` sibling, which fits matched-pair and reservoir quantile regressions separately and pools them by inverse-variance weighting. Standard errors use Powell's sandwich estimator. `likelihood_tier = "none"`: quantile regression minimizes an asymmetric-loss objective, not a proper likelihood, so no likelihood-ratio or parametric-bootstrap methods are exposed. Requires the **quantreg** package.

Super class

`Inference` -> `InferencePropKKQuantileRegrOneLik`

Methods

Public methods:

- InferencePropKKQuantileRegrOneLik$new()
- InferencePropKKQuantileRegrOneLik$clone()

InferencePropKKQuantileRegrOneLik$new(): Initialize proportion-response KK combined-likelihood quantile-regression inference. Responses are fitted on the logit scale; the shared stacked quantile-regression fit is documented in InferenceContinKKQuantileRegrOneLik (this class's continuous-response sibling, sharing the same KKQuantileRegrOneLik component).

Usage:

```
InferencePropKKQuantileRegrOneLik$new(
  des_obj,
  model_formula = NULL,
  tau = 0.5,
  verbose = FALSE
)
```

Arguments:

des_obj A DesignSeqOneByOne object whose entire n subjects are assigned and response y is recorded within.

model_formula Optional formula for covariate adjustment. If NULL (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

tau The quantile level on the logit scale, strictly between 0 and 1. Default is 0.5.

verbose Whether to print progress messages.

InferencePropKKQuantileRegrOneLik$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferencePropKKQuantileRegrOneLik$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

References

Koenker, R. (2005). *Quantile Regression*. Cambridge University Press. doi:10.1017/CBO9780511754098

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'proportion')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferencePropKKQuantileRegrOneLik$new(seq_des)
inf$compute_estimate()
```

InferencePropQuantileRegr

Quantile Regression Inference for Proportion Responses

Description

Fits a quantile regression for proportion responses (constrained to (0, 1)) using the treatment indicator and, optionally, all recorded covariates as predictors. Inference is performed on the **logit (log-odds) scale**: responses $y \in (0, 1)$ are transformed via $\text{logit}(y) = \log(y/(1-y))$ before quantile regression, so the estimated treatment effect is a **log-odds-ratio shift** at quantile `tau`; by default `tau = 0.5`, so this is a median log-odds-ratio shift.

Fitting is via `rq` (method `"br"`, the Barrodale-Roberts simplex algorithm, for the point estimate; the default Frisch-Newton-adjacent interior-point path for the variance-computing fit) on `logit(y) ~ w + covariates` with no intercept column (the design matrix already carries one). Standard errors use **quantreg**'s Powell (1991) kernel sandwich `"nid"` estimator (heteroskedasticity- and design-robust, valid under non-i.i.d. errors) when available, falling back to the i.i.d.-errors `"iid"` estimator if `"nid"` extraction fails; inference on the resulting standard error uses the asymptotic normal (Wald) approximation, not a resampling-based reference distribution, for the asymptotic CI/p-value paths. `compute_asymp_confidence_interval`/`compute_asymp_two_sided_pval` use the fit's residual degrees of freedom $n - p$ in a t-reference (via `compute_z_or_t_ci_from_s_and_df`) rather than a plain normal reference, so the interval/test remain slightly conservative in small samples relative to a bare Wald z.

This class requires the **quantreg** package, which is listed under `Suggests` and is not installed automatically with **EDI**. Install **quantreg** manually before use. Only uncensored proportion responses are supported (checked via `assertNoCensoring` at construction).

Super class

`Inference` -> `InferencePropQuantileRegr`

Methods

Public methods:

- `InferencePropQuantileRegr$new()`
- `InferencePropQuantileRegr$compute_estimate()`
- `InferencePropQuantileRegr$compute_estimate_with_bootstrap_weights()`
- `InferencePropQuantileRegr$compute_asymp_confidence_interval()`
- `InferencePropQuantileRegr$compute_asymp_two_sided_pval()`
- `InferencePropQuantileRegr$clone()`

`InferencePropQuantileRegr$new()`: Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize a quantile-regression inference object for a completed design with a proportion response.

Usage:

```
InferencePropQuantileRegr$new(
  des_obj,
  model_formula = NULL,
  tau = 0.5,
  verbose = FALSE
)
```

Arguments:

`des_obj` A completed `Design` object with a proportion response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`tau` The quantile to estimate (default 0.5).. Default 0.5.

`verbose` Whether to print progress messages.. Default FALSE.

`InferencePropQuantileRegr$compute_estimate()`**:** Computes the fitted treatment coefficient of the `tau`-quantile regression of `logit(y)` on the treatment indicator (plus any adjustment covariates) — a **log-odds-ratio shift at quantile** `tau` of the proportion response, not a difference in means or in the raw-scale quantile. Caches `beta_hat_T` (and, unless `estimate_only`, the standard error and residual degrees of freedom) so repeated calls are cheap; returns `NA_real_` if the reduced design matrix is degenerate (fewer usable rows than columns) or the **quantreg** fit fails/errors.

Usage:

```
InferencePropQuantileRegr$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`InferencePropQuantileRegr$compute_estimate_with_bootstrap_weights()`**:** Recomputes `compute_estimate`'s treatment log-odds-ratio-shift coefficient with each subject's (or block's) contribution to the `tau`-quantile fit reweighted by `subject_or_block_weights` (expanded to per-row weights and passed as `quantreg::rq(..., weights = ...)`), for the Bayesian bootstrap contract; see `InferenceBayesianBootstrap`. **Writes into the same** `beta_hat_T`/`s_beta_hat_T`/`df` **cache fields that** `compute_estimate` **reads from** — a call to this method overwrites the cached original-data estimate with the bootstrap-reweighted one, so a subsequent `compute_estimate()` call will return the *bootstrap replicate's* value from cache rather than recomputing on the original data, until the cache is reset by whatever higher-level bootstrap driver owns this object's lifecycle. Returns `NA_real_` under the same degenerate-design/fit-failure conditions as `compute_estimate`.

Usage:

```
InferencePropQuantileRegr$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferencePropQuantileRegr$compute_asymp_confidence_interval()`**:** Uses the shared asymptotic confidence-interval contract; see `InferenceAsymp`.

Usage:

```
InferencePropQuantileRegr$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`InferencePropQuantileRegr$compute_asymp_two_sided_pval():` Uses the shared asymptotic two-sided p-value contract; see `InferenceAsymp`.

Usage:

```
InferencePropQuantileRegr$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null difference to test against. Default is zero.

`InferencePropQuantileRegr$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferencePropQuantileRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Koenker, R. and Bassett, G. (1978). "Regression Quantiles." *Econometrica*, 46(1), 33-50, doi:10.2307/1913643, for quantile regression itself. Powell, J. L. (1991). "Estimation of Monotonic Regression Models under Quantile Restrictions," in *Nonparametric and Semiparametric Methods in Econometrics and Statistics*, Cambridge University Press, for the `"nid"` sandwich standard error.

See Also

`InferenceContinQuantileRegr` for the untransformed (continuous-scale) analogue of this class.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'proportion')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferencePropQuantileRegr$new(seq_des)
inf$compute_estimate()
```

InferencePropZeroOneInflatedBetaRegr

Zero/One-Inflated Beta Inference for Proportion Responses

Description

Internal class for non-KK zero/one-inflated beta regression models. The response is modeled as a three-component mixture with point masses at 0 and 1 plus a beta-distributed interior component on $(0, 1)$. The reported treatment effect is the treatment coefficient from the beta mean submodel, on the logit scale, **conditional on the response falling strictly inside** $(0, 1)$: it is not, and should not be read as, the effect on the unconditional mean $E[Y]$, which also depends on how treatment shifts the zero/one inflation probabilities. A design where treatment moves mass between the point masses and the interior, with no shift in the interior beta mean, will report a null treatment coefficient here even though $E[Y]$ changed.

The beta mean submodel uses treatment alone in the univariate class and treatment plus covariates in the multivariate class. The zero/one inflation submodels use `model_formula_zero_one`, which defaults to `~ .` so that treatment plus all available covariates enter those auxiliary pieces. See the class-level description above for what the reported coefficient does and does not represent.

Marginal estimand. This class composes `MarginalEstimand` (`set_estimand()`/`get_estimand()`/`get_supported_estima` in addition to the default `"conditional"` estimand described above, it supports `"marginal_mean_diff"`: the model-implied unconditional mean recombines all three mixture components, $E[Y \mid x, w] = \pi_1(x, w) \cdot 1 + (1 - \pi_0(x, w) - \pi_1(x, w)) \cdot \text{logit}^{-1}(x^\top \beta + \beta_T w)$ (the zero mass contributes nothing), where π_0/π_1 are the normalized zero/one-inflation mixture probabilities. The reported treatment effect under `"marginal_mean_diff"` is the g-computation average $\hat{\tau} = n^{-1} \sum_i [\hat{E}(Y \mid x_i, w = 1) - \hat{E}(Y \mid x_i, w = 0)]$, on the response's natural $[0, 1]$ scale (not the log-odds scale of the conditional estimand). Standard errors are delta-method, against the joint covariance of $[\beta, \log \phi, \gamma_0, \gamma_1]$ already returned by `fast_zero_one_inflated_beta_cpp`, using a numerical (central-difference) gradient of $\hat{\tau}$ — see `marginal_estimand_report.md` → `TODO-4` for why analytic differentiation was not used. This is a pure post-fit transform of the same cached maximum-likelihood fit (no refit), so only Wald-via-delta-method inference is available under a marginal estimand (no likelihood-ratio/ score/gradient test — see `set_estimand()`'s testing-type interaction).

Super class

`Inference` -> `InferencePropZeroOneInflatedBetaRegr`

Methods

Public methods:

- `InferencePropZeroOneInflatedBetaRegr$new()`
- `InferencePropZeroOneInflatedBetaRegr$compute_estimate()`
- `InferencePropZeroOneInflatedBetaRegr$compute_asymp_confidence_interval()`
- `InferencePropZeroOneInflatedBetaRegr$compute_asymp_two_sided_pval()`
- `InferencePropZeroOneInflatedBetaRegr$compute_estimate_with_bootstrap_weights()`
- `InferencePropZeroOneInflatedBetaRegr$clone()`

`InferencePropZeroOneInflatedBetaRegr$new()`**:** Initialize inference for the three-component zero/one-inflated beta mixture model; see `InferencePropZeroOneInflatedBetaRegr` for the model form and the important caveat that the reported treatment coefficient is conditional on the interior $(0, 1)$ component, not an unconditional-mean effect. Does not fit the model; the fit is deferred to the first call to `compute_estimate()` or a method that requires it.

Usage:

```
InferencePropZeroOneInflatedBetaRegr$new(
  des_obj,
  model_formula = NULL,
  model_formula_zero_one = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a proportion response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`model_formula_zero_one` Formula for the zero/one inflation submodels. Defaults to `~ .`, meaning treatment plus all available covariates.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`InferencePropZeroOneInflatedBetaRegr$compute_estimate()`**:** Fits the zero/one-inflated beta mixture model by maximum likelihood (jointly the beta mean submodel, the zero/one inflation submodels, and the beta precision). Under the default `estimand = "conditional"`, returns $\hat{\beta}_T$, the treatment log-odds-ratio from the beta mean submodel **conditional on the interior** $(0, 1)$ **component** — see `InferencePropZeroOneInflatedBetaRegr`'s estimand caveat. Under `estimand = "marginal_mean_diff"` (set via `set_estimand()`), returns the g-computation marginal mean difference instead — see the class-level `@details` for the formula. The underlying model fit is identical either way (a pure post-fit transform of the same cached fit, no refit).

Usage:

```
InferencePropZeroOneInflatedBetaRegr$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip standard-error computation and cache only the point estimate; used by randomization and bootstrap resampling paths.

`InferencePropZeroOneInflatedBetaRegr$compute_asymp_confidence_interval()`**:** Wald confidence interval, dispatched by `testing_type` for the conditional estimand (score/gradient/likelihood-ratio/Bartlett available; see `InferenceAsympLik`); under a marginal estimand `testing_type` is always `"wald"` (the only value `set_estimand()` permits there), so this always resolves to the delta-method interval. Calls `self$compute_estimate()` first (not `private$shared()` directly) so the estimand-aware cache is always current regardless of call order.

Usage:

```
InferencePropZeroOneInflatedBetaRegr$compute_asymp_confidence_interval(
  alpha = 0.05
)
```

Arguments:

`alpha` Two-sided miscoverage rate; the returned interval targets `1 - alpha` coverage.

`InferencePropZeroOneInflatedBetaRegr$compute_asymp_two_sided_pval():` Wald two-sided p-value, dispatched by `testing_type` exactly as `compute_asymp_confidence_interval()`; see that method's description for the marginal-estimand always-Wald note.

Usage:

```
InferencePropZeroOneInflatedBetaRegr$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment-effect value under the current estimand (conditional log-odds-ratio, or marginal mean difference).

`InferencePropZeroOneInflatedBetaRegr$compute_estimate_with_bootstrap_weights():` Refits the zero/one-inflated beta model with subject/block-level weights applied to the fitting log-likelihood (Bayesian-bootstrap or nonparametric-bootstrap draw weights, expanded to row level via `private$expand_subject_or_block_weights_to_row_weights()`), and returns the reweighted conditional log-odds-ratio estimate $\hat{\beta}_T^{(w)}$.

Usage:

```
InferencePropZeroOneInflatedBetaRegr$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferencePropZeroOneInflatedBetaRegr$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferencePropZeroOneInflatedBetaRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Ospina, R., and Ferrari, S. L. P. (2010). "Inflated beta distributions." *Statistical Papers*, 51(1), 111-126, doi:10.1007/s0036200801254, for the zero/one-inflated beta mixture density; Ferrari, S., and Cribari-Neto, F. (2004). "Beta regression for modelling rates and proportions." *Journal of Applied Statistics*, 31(7), 799-815, doi:10.1080/0266476042000214501, for the interior beta-regression submodel.

See Also

`InferencePropBetaRegr` for the plain (non-inflated) beta regression model.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'proportion')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferencePropZeroOneInflatedBetaRegr$new(seq_des)
inf$compute_estimate()
```

InferenceRandCustom *Randomization test/CI on a user-supplied statistic*

Description

Runs a randomization test (and, via `compute_rand_confidence_ interval()`, a randomization confidence interval) for an arbitrary user-supplied statistic, without writing an `Inference` subclass. The statistic function is called once per permutation as `fn(y, w, dead)` – plain numeric/integer vectors: the current (possibly permuted) response, treatment assignment, and event indicator. It must return one scalar. For maximum speed, supply `custom_randomization_statistic_cpp` instead: C++ source defining a function of `(NumericVector y, IntegerVector w)` or `(NumericVector y, IntegerVector w, IntegerVector dead)` returning a scalar `double`, using the same convention as `DesignFixedOptimal`'s `custom_objective`.

Only a randomization test and randomization confidence interval are available – there is no package point-estimator, Wald path, or bootstrap machinery on this class, so no other action needs disabling.

Super classes

`Inference` -> `InferenceCustomRand` -> `InferenceRandCustom`

Methods

Public methods:

- `InferenceRandCustom$new()`
- `InferenceRandCustom$fit()`
- `InferenceRandCustom$clone()`

`InferenceRandCustom$new()`**:** Initialize.

Usage:

```
InferenceRandCustom$new(
  des_obj,
  custom_randomization_statistic_function = NULL,
  custom_randomization_statistic_cpp = NULL,
  verbose = FALSE
)
```

Arguments:

des_obj See class description.

custom_randomization_statistic_function See class description.

custom_randomization_statistic_cpp See class description.

verbose See class description.

InferenceRandCustom$fit(): Calls the user-supplied statistic on the observed data.

Usage:

```
InferenceRandCustom$fit(estimate_only = FALSE)
```

Arguments:

estimate_only Unused; present for the ‘fit()‘ contract.

InferenceRandCustom$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferenceRandCustom$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

InferenceSuite *Inference Suite: Discover and Bundle Every Applicable Inference Class for a Design*

Description

A lightweight coordinator (not itself an Inference subclass, and not part of the Inference R6 hierarchy) that pairs a single completed Design object with the full set of concrete Inference classes compatible with it. On construction, the suite consults package-level inference metadata to discover every exported, non-abstract Inference subclass whose declared response-type, matched-design (KK), blocking, and censoring requirements are all satisfied by des_obj, storing the resulting sorted class-name vector in applicable_design_classes. Because discovery is driven by metadata lookups rather than by actually attempting to construct each candidate class, the applicable list automatically stays current as new inference classes are registered elsewhere in the package, without this class needing any changes, and without risking side effects (e.g. an optional-package load failure inside some class's constructor) from a doomed construction attempt.

Compatibility rules (see is_inference_class_compatible_with_design_metadata(), also used by Design's own applicable_inference_class_names()): a candidate class is excluded if it is abstract or not exported; if it declares no compatible response types, or none match des_obj's response type; if its name contains "KK" (a matched-design-only class) but des_obj does not support KK matching; if the class's requires_blocking_design() is TRUE (currently only InferenceIncidExtendedRobins – InferenceIncidCMH works on both blocking and non-blocking designs via different standard-error estimators, so it is **not** excluded) but des_obj does not support blocking; or if des_obj has any left-/interval-censored subjects (a finite y_R) but the class's supports_interval_or_left_censored_data() is FALSE – both of the latter two mirror Inference$initialize()'s own construction-time gate exactly, via each class's registered requires_blocking_design/supports_general_censoring metadata (see infer_inference_requires_blocking_design()/ infer_inference_supports_general_censoring()

in `inference_class_registry.R`). Ordinary right-censoring alone never excludes a class. The KK-name-pattern rule is still hardcoded in this class rather than looked up from a central registry.

A class that is design-compatible but whose registered `required_packages` are not all installed is excluded from `applicable_design_classes` and reported separately, by class name, in `unavailable_due_to_missing_pa` – so callers can tell "not applicable to this design" apart from "applicable, but an optional dependency isn't installed" (see the `Discovery` section of `fix_inference_hierarchy.md`). Package availability is never a reason a class is treated as design-incompatible.

Construction itself does not compute any estimates, p-values, or confidence intervals – it only discovers and validates which inference classes are applicable and does not eagerly construct any of them. This class's `run_all_inference()` method does that: it constructs and fits every applicable class and returns a uniform comparison across them (see that method's own documentation for the output schema, the CI/p-value method selection policy, and the `screen`/`html`/`plots`/`pdf`/`save_results_as_JSON` output options). `lock_objects = FALSE` allows ad hoc fields to be attached to an instance after construction.

Every row this class discovers and fits is a test about the same outcome variable, by construction. `response_type` is a required, immutable constructor argument on `Design` (read-only thereafter via `get_response_type()`), and this class discovers every candidate in `applicable_design_classes` from one attached `Design` object's one `response_type` – there is no code path here that spans two response types in a single instance. This matters beyond bookkeeping: a planned comparison-across-classes feature (a single combined-evidence p-value summarizing every row, via a dependence-robust combination test) relies on every combined class sharing one sharp null of "no treatment effect on this outcome" – which only holds when every test concerns the same outcome variable (combining, say, a survival model's p-value with an unrelated continuous-outcome model's p-value would not be valid, since a real effect on one with none on the other is entirely plausible). That precondition is guaranteed here structurally, not by caller discipline.

Same-`Y` does not mean same estimand, and that matters for interpretation. Rows discovered here routinely test genuinely different θ_i on the same outcome – a mean difference, a log-odds ratio under one link, a rank-based effect, a quantile shift – each θ_i its own parameterization's own null. The combined $H_0 : \theta_1 = 0 \cap \cdots \cap \theta_k = 0$ is one coherent claim (Fisher's unit-level sharp null: no treatment effect *whatsoever*) only under a randomization-based/exact procedure, where every possible summary of "effect" is simultaneously zero by construction. Asymptotic/likelihood-based procedures instead test a *weak*, population-level null specific to their own parameterization, and weak nulls for genuinely different summaries are not generally nested – treatment can shift a distribution's variance or a high quantile while leaving its mean exactly unchanged, so "mean difference = 0" does not imply "quantile effect = 0" outside location-shift-style models. This is comparatively safe across different *link functions for the same latent effect* (e.g. cauchit/probit/cloglog/logit on the same binary/ordinal response, which typically share one underlying latent-variable null) and comparatively riskier across different *kinds* of summary (a mean-difference test combined with a rank-based test combined with a quantile-regression test). Under that weak-null reading, a rejection is more honestly read as "at least one specific summary of this outcome's distribution differs" rather than "there is one coherent nonzero effect" – reinforcing, not loosening, the interpretation caveat below.

Combined Evidence interpretation caveat (read before using `combined_evidence$pval`): the Cauchy combination test is a union-intersection test of $H_0 : \theta_1 = 0 \cap \theta_2 = 0 \cap \cdots \cap \theta_k = 0$ against the alternative that **at least one** $\theta_i \neq 0$. A significant `combined_evidence$pval` is therefore **evidence of an effect in at least one of these senses, not evidence for a specific estimate or direction** – it does not say which class's estimand is nonzero, nor does a small combined p-value imply every

(or even most) constituent p-values were small. Do not report `combined_evidence$pval` as if it estimated a single effect size, and do not treat it as validating any one class's estimate over another's; its only valid use is as evidence that *some* legitimate way of looking for a treatment effect on this outcome found one. This guarantee assumes every constituent `p_i` is itself valid (i.e. uniform under its own null) – the Cauchy combination's dependence-robust size control says nothing about **which** such `p_i` is small, so a single miscalibrated or misspecified procedure (e.g. an asymptotic approximation that breaks down for this sample) can dominate the combined result the same way it would dominate a plain minimum-p-value test, even when every other procedure shows nothing. A significant `combined_evidence$pval` is worth cross-checking against the per-estimand CI-forest plot (`plots$ci_forest`) before trusting it – one outlying interval sitting apart from a cluster of concordant ones is a miscalibration flag, not confirmed evidence.

This same-`Y` precondition is guaranteed *within* one `InferenceSuite` instance structurally (one `Design`, one `response_type`), but the architecture cannot stop a caller from manually combining raw `pvals` pulled from two separate `InferenceSuite` objects' `results_tables` outside `cct_combine_pvalues()` – doing so is outside this function's validity guarantee and is not a supported use of this metric.

Public fields

`applicable_design_classes` Character vector of applicable inference class names derived during initialization.

`unavailable_due_to_missing_packages` A named list, one entry per otherwise-design-compatible class whose registered `required_packages` are not all installed: names are class names, values are the character vector of missing package names. These classes are excluded from `applicable_design_classes` but are reported here separately from plain design/response-type incompatibility (see the class-level docs' "Discovery" rules), so callers can distinguish "not applicable to this design" from "applicable, but an optional dependency isn't installed." Empty named list if every design-compatible class has all required packages available.

Methods

Public methods:

- `InferenceSuite$new()`
- `InferenceSuite$run_all_inference()`
- `InferenceSuite$clone()`

`InferenceSuite$new()`: Discover every `Inference` class applicable to `des_obj` (see the class-level documentation for the compatibility rules) and validate any per-class constructor overrides in `inference_params`, storing `applicable_design_classes` for later use. This constructor does **not** instantiate any `Inference` object itself; callers are expected to construct the specific classes they need (optionally passing the validated `inference_params`) from the discovered list.

Usage:

```
InferenceSuite$new(des_obj, model_formula = NULL, inference_params = list())
```

Arguments:

`des_obj` A completed `Design` object (validated via `is(des_obj, "Design")` when assertions are enabled; see `toggle_asserts`).

`model_formula` Accepted for interface/future-extension purposes but currently **not used anywhere in this method's body** – supplying a non-`NULL` value has no effect on discovery, validation, or any stored state. Do not rely on this parameter to affect covariate adjustment; a design's own model formula and design matrix are what individual `Inference` classes actually consult when later constructed from this suite's discovered class list.

`inference_params` A named list of lists supplying additional constructor arguments for specific inference classes. Each name must be the name of a concrete `Inference` subclass that is applicable to `des_obj` (checked against `applicable_design_classes` once discovered – an inapplicable class name raises an error); the corresponding list contains keyword arguments (beyond `des_obj`) forwarded to that class's `initialize`, and every argument name supplied must match a formal parameter of that class's `initialize` method (other than `des_obj` and `...`) or an error is raised naming the unknown argument(s) and the valid ones. Defaults to an empty list (no extra arguments for any class).

`InferenceSuite$run_all_inference()`**:** Construct and fit every class in `applicable_design_classes`, and report one uniform comparison row per class – estimate, SE, CI, p-value (each via the highest-priority available method; see `inference_suite_inspect.md`'s "Method Selection Policy"), likelihood tier, estimand (where declared), fit time, captured warnings, and status. Unlike the constructor, this method fits models and is not free; call it explicitly when you want the comparison, not automatically.

A single class's construction or fit failure never aborts the report – it is caught and recorded as that class's `status/message` (see "Per-Class Failure Isolation" in the design doc).

Side effects (v1.0.0 slice): `screen` prints each row as its class finishes fitting (computation order), not buffered to the end, with a percent-done/estimated-time-remaining progress bar line underneath each row (the ETA is the mean per-class elapsed time so far times classes remaining), followed by a footer listing classes excluded for missing optional packages. `html = TRUE` writes a self-contained, timestamped HTML report (the same table plus the same footer) to `output_dir` and opens it via `browseURL`; it requires the **knitr** package. The `plots` ggplot2 visualizations, and their embedding into this HTML report, are not yet implemented (`inference_suite_inspect.md` TODO-7); `pdf` output is not yet a parameter of this method.

Usage:

```
InferenceSuite$run_all_inference(
  screen = TRUE,
  html = FALSE,
  alpha = 0.05,
  save_results_as_JSON = FALSE,
  plots = screen,
  pdf = FALSE,
  classes = NULL,
  exclude_classes = character(),
  max_secs_per_class = NULL,
  num_cores = 1L,
  formulas = NULL,
  methods = NULL,
  basic_bootstrap = FALSE,
  compute_conf_intervals = FALSE,
  output_dir = "~",
  combined_evidence_estimands = NULL,
```

```
  combined_evidence_weighting = c("estimand_grouped", "equal", "custom"),
  combined_evidence_weights = NULL
)
```

Arguments:

`screen` Print results to the console as each class finishes. At least one of `screen`/`html` must be `TRUE`.

`html` Render, save (`output_dir`, timestamped filename), and auto-open a self-contained HTML report of the results.

`alpha` Significance level: confidence intervals are computed at `1 - alpha` and `alpha` is the significance threshold used anywhere the report flags significance. Default `0.05`.

`save_results_as_JSON` If `TRUE`, serialize the return object (excluding plot objects) to a timestamped JSON file in `output_dir`. Requires the optional **jsonlite** package; if it is not installed, a `warning()` is issued and this artifact is skipped rather than erroring. Default `FALSE`.

`plots` If `TRUE`, build and display (on the current graphics device) one visualization per estimand: an annotated confidence-interval forest plot (p-value left of each interval, interval width right of it, class/method label right-aligned on each row, color-keyed to significance at `alpha`) stacked over its own "Estimates" box-and-whisker subplot – a free x-axis (same label, same log10/linear choice as the forest, but its own limits) summarizing the point estimates, one point per inference class/formula, collapsed over method/type since those share one estimate. That subplot scales with how many distinct estimates there are: none for a single estimate (redundant with the forest's own dot), dots alone for 2-5, and dots over a box-and-whisker for more than 5. Built with **ggplot2** and stacked into a single **gtable** grob (draw with `grid::grid.draw()`); requires the optional **ggplot2** package, if it is not installed, a `warning()` is issued and plotting is skipped rather than erroring. Defaults to the value of `screen`.

`pdf` If `TRUE`, save the visualization to one timestamped multi-page PDF file in `output_dir` (one page per estimand; page height scales with the largest estimand's number of CI rows). Same **ggplot2** dependency and missing-package handling as `plots`. Default `FALSE`.

`classes` Optional character vector restricting which applicable classes to fit – e.g. re-running against only the few classes a user is actually deciding between, without reconstructing the suite. Every name must already be in `applicable_design_classes` or this errors, naming the unknown name(s) and the valid ones. `NULL` (default) fits every applicable class.

`exclude_classes` Optional character vector of applicable classes to skip, applied after `classes`. Same validation as `classes`. Default none.

`max_secs_per_class` Optional per-class elapsed-time limit in seconds (via `setTimeLimit`), after which that class's row gets `status = "timeout"` instead of hanging the whole report. Protects against one pathological class (e.g. a bootstrap/randomization method with many replicates) blocking every other class. **Known limitation:** R's time limits are checked at R-level interrupt points, so this reliably cuts off slow R-level work but is not guaranteed to interrupt one very slow single native (C/C++/BLAS) call with no intervening R-level check. `NULL` (default) means no limit.

`num_cores` If greater than `1`, fit classes in parallel across this many forked workers (`makeForkCluster`) – Unix/Linux only; on other platforms this falls back to sequential (`num_cores = 1`) with a `warning()`. **Screen output changes under parallel execution:** fitting is a single blocking call that only returns once every worker has finished, so there is no meaningful per-class ETA to show while running – `screen = TRUE` instead prints a "fitting N classes across

K workers" message up front, then every result row together once complete, then a total-elapsed-time summary line (a deliberate design choice, not a degraded default – see `inference_suite_inspect.md`'s TODO-13). Default `1L` (sequential, with the normal incremental streaming/progress bar).

`formulas` `NULL` (default), a single formula (`~ .`), a single formula string (`"~ ."`), or a collection of either – including `c(~ 1, ~ .)`, which base R already returns as a plain `list` of `formula` objects (formulas have no `c()` method of their own), or a character vector (`c("~ .", "~ age + sex * smoking")`). `NULL` means each class fits once with its own default formula – identical to omitting this argument entirely, since `model_formula = NULL` at construction already resolves to `des_obj$get_design_formula()` (default `~ .`). When non-`NULL`, only classes whose constructor **syntactically** accepts a `model_formula` argument are fit once per formula in `formulas` (one `results_table` row each, disambiguated in `results` by `"<class>[<formula>]"` names); classes without a `model_formula` constructor argument at all still fit exactly once, ignoring `formulas`. Note this is a syntactic check (does the constructor accept one), not a semantic one (does the fit actually use it) – some classes accept-and-ignore `model_formula` (e.g. `InferenceAllSimpleAverageDiff`'s unadjusted Welch's t-test); see `fix_inference_hierarchy.md`'s `adjusts_for_covariates` registry-metadata audit, which makes the `cov_model` column semantics-aware wherever that audit has landed.

`methods` `NULL` (default), a character vector of method sentinel strings, or (TODO-22) a named list, `sentinel` to a character vector of requested `type` values, or `NULL`, restricting which inference method(s) – and, for the three resampling sentinels marked "typed" below, which resampling/CI- construction `type` flavor(s) – get fit and reported per class. `NULL` considers every sentinel in `EDI_INFERENCE_SUITE_METHOD_SENTINELS`, and for each typed sentinel, every `type` value that class supports (queried at runtime via its own `get_supported_bootstrap_{pval,ci}_types()` / `get_supported_bayesian_bootstrap_{pval,ci}_types()` / `get_supported_rand_boot{pval,ci}_types()` accessor – never a hardcoded type table in this package), except class/method/type combinations declared in `EDI_COMPREHENSIVE_SLOW_PATHS`. Those implemented but prohibitively slow paths are omitted only from this default selection. Supplying `methods` explicitly opts into the named sentinel/type combinations even when the registry marks them slow. Thus the default remains broad without allowing known multi-minute paths to dominate a routine report; it is not a single "best available" cascade. List-shaped example: `methods = list(bootstrap = c("percentile","bca"), rand_bootstrap = NULL)` fits only `"bootstrap"` (restricted to the `"percentile"`/`"bca"` types that class actually supports) and `"rand_bootstrap"` (every type that class supports); a sentinel present as a list name with value `NULL` still means "every valid type" for it, exactly like the flat-vector shape – to *not* fit a sentinel at all, simply don't name it. Requesting a `type` for a sentinel with no `type` axis (any sentinel not marked "typed" below) errors. Valid sentinels, each corresponding to one asymptotic/exact/ randomization/resampling inference family a class may (or may not) implement:

- `"wald"` Asymptotic Wald inference – `compute_asymp_confidence_interval()` / `compute_asymp_two_sided_pva` (capability `"wald"`). The standard closed-form normal-approximation CI/test.
- `"exact"` Exact inference – `compute_exact_confidence_interval()` / `compute_exact_two_sided_pval_for_tr` (capability `"exact_test"`). Finite-sample-exact methods (e.g. Fisher's exact test, exact binomial).
- `"rand"` Randomization inference – `compute_rand_confidence_interval()` / `compute_rand_two_sided_pval()` (capabilities `"randomization_ci"`/`"randomization_test"` – distinct capability names for the CI vs. p-value side, since a class can support one without the other). Design-based inference via re-randomizing the observed treatment assignment.

"rand_bootstrap" **(typed)** Randomization-bootstrap inference – compute_rand_bootstrap_confidence_interv compute_rand_bootstrap_two_sided_pval() (capabilities "randomization_bootstrap_ci"/"randomizatio – distinct capability names for the CI vs. p-value side, since a class can support one without the other). Resamples under the randomization null rather than the usual iid-resampling bootstrap. type (both sides agree on the same four values, unlike "bootstrap"/"bayes_boot" below): "percentile", "studentized", "symmetric-percentile-t", "smoothed".

"jackknife" Jackknife-Wald inference – compute_jackknife_wald_confidence_interval()/ compute_jackknife_wald_two_sided_pval() (capability "jackknife"). Leave-one-out variance estimate feeding a Wald-style CI/test.

"score" Score (Rao) test – compute_score_confidence_interval()/ compute_score_two_sided_pval() (capability "likelihood_tests", one of its three independent sub-procedures).

"lik_ratio" Likelihood-ratio test – compute_lik_ratio_confidence_interval()/ compute_lik_ratio_two_s (capability "likelihood_tests").

"gradient" Gradient test – compute_gradient_confidence_interval()/ compute_gradient_two_sided_pval (capability "likelihood_tests"). (The plain, "best available" auto-selecting compute_lik_ratio_bartlett_tw compute_lik_ratio_bartlett_confidence_interval() dispatcher is deliberately not its own sentinel – it only picks between the two explicit variants below depending on which this class implements, so it is never a distinct inference procedure.)

"lik_ratio_bartlett_approx" Bartlett-corrected likelihood-ratio test, Monte-Carlo-approximated correction factor pinned explicitly (for reproducibility) – compute_lik_ratio_bartlett_approx_confidence_i compute_lik_ratio_bartlett_approx_two_sided_pval() (capability "likelihood_tests"; degrades to NA for classes without an approximate Bartlett factor).

"lik_ratio_bartlett_exact" Bartlett-corrected likelihood-ratio test, closed-form analytic correction factor pinned explicitly – compute_lik_ratio_bartlett_exact_confidence_interval()/ compute_lik_ratio_bartlett_exact_two_sided_pval() (capability "likelihood_tests"; degrades to NA for classes without an exact Bartlett factor).

"param_boot" Bootstrap-calibrated likelihood-ratio test – compute_lik_ratio_bootstrap_confidence_interva compute_lik_ratio_bootstrap_two_sided_pval() (capability "parametric_likelihood_bootstrap").

"param_boot_direct" Direct parametric-bootstrap estimate/CI/pval for the treatment coefficient itself – compute_param_bootstrap_confidence_interval()/ compute_param_bootstrap_pval() (capability "parametric_likelihood_bootstrap"; distinct from "param_boot" above, which is a bootstrap-calibrated likelihood-ratio *test*, not a direct estimate).

"bayes_boot" **(typed)** Bayesian bootstrap inference – compute_bayesian_bootstrap_confidence_interval()/ compute_bayesian_bootstrap_two_sided_pval() (capability "bayesian_bootstrap"). CI-side type: "percentile", "basic", "wald", "studentized", "bootstrap-t", "bca"; pval-side type swaps "basic" for "symmetric" (all others the same).

"bootstrap" **(typed)** Nonparametric bootstrap inference – compute_bootstrap_confidence_interval()/ compute_bootstrap_two_sided_pval() (capability "nonparametric_bootstrap"). CI-side type: "percentile", "basic", "studentized", "bootstrap-t", "symmetric-percentile-t", "bca", "prepivoted", "double-bootstrap", "calibrated", "smoothed"; pval-side type is a smaller set – "percentile", "symmetric", "studentized", "bootstrap-t", "bca" – neither "basic" nor the other CI-only variants apply on the pval side.

For the three "typed" sentinels above, an exhaustive type list is documented here for orientation only – the actual set consulted at runtime always comes from that class's own accessor (see the top of this section), so a class need not support every value listed. ("likelihood_ratio"/"estimating_equa are deliberately not separate sentinels – both capabilities gate the exact same method pair "lik_ratio" above already covers.) For each class, only sentinels the class has *any* CI or

p-value capability for (among the requested methods) get a row; for a typed sentinel, one row per type that class actually supports (intersected with any requested type subset) – a class with zero applicable sentinels, or a typed sentinel with zero resulting types, still gets exactly one row with method/type = NA_character_ (mirrors the pre-methods "no capability" row) rather than being silently dropped. A class contributing more than one applicable-sentinel row is disambiguated in results/results_table by "<class>{<method>}" or "<class>{<method>:<type>}" (or with a "[<formula>]" tag too under simultaneous formulas fan-out) names. Unlike the removed cascade, ci_method/pval_method on a given row now always match that row's own method (or are NA if this class lacks that half of the sentinel's capability, **including** when type is valid on one side but not the other) – there is no fallback to a different sentinel within one row.

basic_bootstrap FALSE (default). Convenience flag: when TRUE, restricts every typed sentinel ("bootstrap"/ "bayes_boot"/"rand_bootstrap") to just that class's first (i.e. default) type value instead of fitting every type it supports – "just run the default bootstrap flavor for nonparametric/Bayesian/randomization resampling" without having to spell out methods = list(bootstrap = ..., bayes_boot = ..., rand_bootstrap = ...) by hand. Only takes effect for a typed sentinel the caller didn't already restrict via an explicit methods list entry – an explicit type request there always wins over this flag. No effect on non-typed sentinels ("param_boot"/"param_boot_direct" included – neither has a type axis, so they already run their one procedure).

compute_conf_intervals FALSE (default). When FALSE, every task's confidence-interval side (ci_a/ci_b/ ci_method) is skipped entirely – only the p-value side runs. Several sentinels' CI search (Bartlett-approx, "rand", "rand_bootstrap", "param_boot") re-invokes the same expensive machinery as its own p-value roughly 15-40 times per bound during root-finding, by far the dominant cost of a full run_all_inference() run for those sentinels; skipping it can cut total runtime dramatically. ci_a/ci_b/ci_method stay present but always NA in results_table (stable schema either way) and are omitted entirely from the live/print/HTML display tables when FALSE. Set TRUE to compute CIs as before.

output_dir Directory for the html/pdf/ save_results_as_JSON output files. Default "~" (the user's home directory), not the current working directory – these calls are routinely made from inside the package's own source tree (a demo/dev script run from R/EDI/), and a stray timestamped report left in getwd() there is exactly the kind of untracked file that can end up bundled into a source tarball (R CMD build) or committed by accident. Pass an explicit path (e.g. the current directory, or a temp directory) to write elsewhere.

combined_evidence_estimands NULL (default: include every declared estimand), or a character vector of estimand values to restrict the Combined Evidence p-value/weights to. Validated argument-time against the estimand values actually declared among classes/exclude_classes-filtered candidates.

combined_evidence_weighting One of "estimand_grouped" (default – w_i = 1 / (G * m_i)), "equal" (flat w_i = 1/k), or "custom" (caller supplies combined_evidence_weights). See inference_suite_inspect.md's TODO-15.

combined_evidence_weights Named numeric vector (inference_class name -> weight), required when and only when combined_evidence_weighting = "custom". Names must be a subset of the classes being fit; an unnamed usable class defaults to weight 0 (excluded). Need not pre-sum to 1.

Returns: Invisibly, an object of class c("EDIInferenceSuiteResults", "list") with elements results (one named sub-list per class, in computation order), results_table (the same

rows as a flat `data.frame`, sorted/grouped by `estimand` – `NA_character_` last – with a secondary sort by `inference_class`; includes the `weight` column driven by `combined_evidence_weighting/combined_evidence_estimands`), `combined_evidence` (`list(pval, stat, method = "cauchy_combination", n_classes_used, n_estimand_groups, estimands_used, weighting, weights_used, classes_used)` – the Cauchy-combination-test p-value/statistic across all usable rows under the resolved weighting policy; `weights_used/classes_used` are keyed/valued by each row's `results` name, not `results_table$inference_class` directly, since that column can repeat under `formulas`; `pval = stat = NA_real_` if fewer than 2 rows are usable), `design`, `alpha`, `unavailable_due_to_missing_packages`, `plots` (`list(ci_forest)`; `ci_forest` is a named list of one **gtable** grob per `estimand` – the CI forest stacked over its "Estimates" box-and-whisker subplot; draw with `grid::grid.draw()` or pass to `ggplot2::ggsave()` – possibly empty – rather than a single plot, since the visualization is split one-per-estimand, per user request, 2026-08-19; the former separate `estimates` plot became that subplot, 2026-08-21), `files` (`list(html, pdf, json)`, each a path or `NULL`; `pdf` is one multi-page PDF with one page per estimand), `timestamp`, `total_secs`, and `edi_version`.

`InferenceSuite$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSuite$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Madigan, D., Ryan, P. B., and Schuemie, M. (2013), "Does design matter? Systematic evaluation of the impact of analytical choices on effect estimates in observational studies," *Therapeutic Advances in Drug Safety*, 4(2), 53-62, PMID 25083251 – the motivating finding behind this class's "every legitimate way to look for an effect, compared honestly" default output.

Liu, Y. and Xie, J. (2020), "Cauchy combination test: a powerful test with analytic p-value calculation under arbitrary dependency structures," *Journal of the American Statistical Association*, 115(529), 393-402 – `run_all_inference()`'s Combined Evidence Metric.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 20, response_type = "continuous")
for (i in 1:20) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x = rnorm(1)))
}
seq_des$add_all_subject_responses(rnorm(20))

suite = InferenceSuite$new(seq_des)
suite$applicable_design_classes

# Fit and compare every applicable class:
results = suite$run_all_inference(screen = TRUE)
results$results_table
```

InferenceSurvivalCoxPHRegr

Cox Proportional Hazards Regression Inference for Survival Responses

Description

Fits a Cox proportional hazards model, $\lambda(t \mid x_i) = \lambda_0(t)\exp(x_i^\top \beta)$, for survival responses using the treatment indicator and, optionally, all recorded covariates as predictors, by maximizing the **Breslow-tie-corrected partial likelihood**. For exact/right-censored data, fitting uses this package's internal Newton-Raphson C++ solver (`fast_coxph_regression_prebuilt_cpp`) by default (`use_rcpp = TRUE`), falling back to `survival::coxph.fit()`/`survival::coxph()` if that fails to converge or `use_rcpp = FALSE`. For **left- or interval-censored** data (a genuinely different likelihood, with no closed-form partial-likelihood score/information), fitting instead dispatches to `icenReg::ic_sp(model = "ph")`, a semiparametric NPMLE Cox fit whose standard errors come from **icenReg**'s own internal bootstrap (not a closed-form covariance) — only Wald inference (`testing_type = "wald"`) is supported on that path; score/gradient/likelihood-ratio/Bartlett testing types raise an informative error for such data. This is a partial-likelihood class (`likelihood_tier = "partial"`) supporting score, gradient, and likelihood-ratio tests, plus parametric likelihood-ratio bootstrap calibration (both only for the exact/right-censored path), in addition to Wald and resampling-based inference. Fitted coefficients exceeding a fixed magnitude threshold (20, on the log-hazard-ratio scale) are treated as non-estimable (a numerical-divergence guard) rather than returned.

Super class

`Inference` -> `InferenceSurvivalCoxPHRegr`

Methods

Public methods:

- `InferenceSurvivalCoxPHRegr$new()`
- `InferenceSurvivalCoxPHRegr$compute_estimate()`
- `InferenceSurvivalCoxPHRegr$compute_asymp_confidence_interval()`
- `InferenceSurvivalCoxPHRegr$compute_asymp_two_sided_pval()`
- `InferenceSurvivalCoxPHRegr$compute_estimate_with_bootstrap_weights()`
- `InferenceSurvivalCoxPHRegr$clone()`

`InferenceSurvivalCoxPHRegr$new()`**:** Uses the shared randomization two-sided p-value contract; see `InferenceRand`. Deliberately pulled from `InferenceRand`, not `InferenceRandCI` – despite `InferenceRandCI`'s override having a richer signature (`type`/`args_for_type`), its body calls `super$...()`, which resolves against this class's *actual* R6 superclass (`Inference`, which has no such method) once the method body is extracted and merged flatly by the component system – not against `InferenceRand` the way it would inside `InferenceRandCI`'s own real inheritance chain. `InferenceRandCI`'s only other content is an incidence-response special case (Zhang exact test) that never applies to survival data anyway, so the two are behaviorally identical

for this class – confirmed by tracing the body, not assumed. Matches the pattern already used by the sibling migrated classes (LogRank/ GehanWilcox/KMDiff/RestrictedMeanDiff).

Initialize a Cox PH inference object for a completed design with a survival response. Unlike most survival inference classes in this package, this one accepts left- and interval-censored data (via an **icenReg**-backed fallback fit; see class documentation), not only exact/right-censored.

Usage:

```
InferenceSurvivalCoxPHRegr$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a survival response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`use_rcpp` Logical. If `TRUE` (default), enable internal Rcpp score/information helpers for likelihood inference.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values by default.

`InferenceSurvivalCoxPHRegr$compute_estimate()`**:** Computes the Cox PH treatment coefficient $\hat{\beta}_T$ (log hazard ratio) — see class documentation for the fitting backend used (partial-likelihood C++/`survival` solver for exact/ right-censored data; **icenReg** NPMLE for left-/interval-censored data). `NA` if the fit fails or the fitted coefficients are numerically extreme.

Usage:

```
InferenceSurvivalCoxPHRegr$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`InferenceSurvivalCoxPHRegr$compute_asymp_confidence_interval()`**:** Computes an asymptotic confidence interval using the configured likelihood-backed test.

Usage:

```
InferenceSurvivalCoxPHRegr$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level 1 - `alpha`. Default 0.05.

`InferenceSurvivalCoxPHRegr$compute_asymp_two_sided_pval()`**:** Computes an asymptotic two-sided p-value using the configured likelihood-backed test.

Usage:

```
InferenceSurvivalCoxPHRegr$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect to test against. Default 0.

`InferenceSurvivalCoxPHRegr$compute_estimate_with_bootstrap_weights()`: Recomputes the Cox PH treatment estimate under subject/block bootstrap weights (via `weighted_cox_bootstrap_surrogate_fi` which assumes ordinary right-censoring semantics), used by the Bayesian bootstrap and related weighted-resampling machinery. If the weights are effectively constant, short-circuits to the unweighted `$compute_estimate(estimate_only = TRUE)`. **Not supported** for left- or interval-censored data — raises an error immediately, since the surrogate weighted fit has no extension for that likelihood. Always leaves the standard error unavailable (`NA`) regardless of `estimate_only` — this weighted path never computes a variance.

Usage:

```
InferenceSurvivalCoxPHRegr$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Subject-, block-, cluster-, or matched-set bootstrap weights.

`estimate_only` Present for interface parity; this method never computes variance components regardless of its value.

`InferenceSurvivalCoxPHRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalCoxPHRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Cox, D. R. (1972). "Regression Models and Life-Tables." *Journal of the Royal Statistical Society, Series B*, 34(2), 187-220, for the proportional hazards model and partial likelihood; Breslow, N. E. (1974). "Covariance Analysis of Censored Survival Data." *Biometrics*, 30(1), 89-99, doi:10.2307/2529620, for the tied-event partial-likelihood approximation used for exact/right-censored data.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'survival')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalCoxPHRegr$new(seq_des)
inf$compute_estimate()
```

`InferenceSurvivalDepCensTransformRegr`
Dependent-Censoring Transformation Inference for Survival Responses

Description

Fits a joint bivariate log-normal transformation model for a latent event time T_i^E and a latent censoring time T_i^C that are allowed to be **dependent** (a violation of the usual independent-censoring assumption): $\log T_i^E = X_i^\top \beta_{\text{event}} + \sigma_{\text{event}} \epsilon_i^E$, $\log T_i^C = X_i^\top \beta_{\text{cens}} + \sigma_{\text{cens}} \epsilon_i^C$, with $(\epsilon_i^E, \epsilon_i^C)$ jointly standard bivariate normal with correlation ρ (estimated via an `atanh`-reparameterized, clamped nuisance parameter). X_i includes the treatment indicator W_i as its first column, so $\hat{\beta}_T$ (the first entry of $\hat{\beta}_{\text{event}}$) is a log-time-ratio for the **event** submodel, on the same AFT interpretation scale as `InferenceSurvivalWeibullRegr` but log-normal rather than Weibull, and jointly modeling the censoring mechanism rather than assuming it independent. This is the correct tool when censoring is suspected to depend on the same latent factors driving the event time (e.g. sicker patients are both more likely to be censored — dropout — and more likely to fail early), a scenario under which ordinary Kaplan-Meier/Cox/AFT methods (which assume independent censoring) are biased. `likelihood_tier = "full"`: likelihood-ratio, score, gradient, and Wald tests are available when the model converges, plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test. **Substantial method-support limitations, all deliberate**: randomization inference and jackknife bias correction/standard errors are hard-unsupported (each randomization draw would require a full dependent-censoring likelihood refit, too unstable/slow for the comprehensive test suite; jackknife bias correction is unstable for this likelihood on small censored samples) — every jackknife/randomization method returns `NA` and marks the result nonestimable rather than computing a value. Nonparametric-bootstrap confidence intervals are computed and returned whenever both endpoints are finite; they are never replaced based on width, sign, or agreement with a Wald interval. Numerical failures return `NA`. Bayesian-bootstrap weighted re-estimation uses a fast Cox-model surrogate fit (`weighted_cox_bootstrap_surrogate_fit()`) as an approximation rather than a full weighted joint-likelihood refit.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Value

A single logical.

Super class

`Inference` -> `InferenceSurvivalDepCensTransformRegr`

Methods

Public methods:

- `InferenceSurvivalDepCensTransformRegr$supports_rand_pval_for_incidence()`
- `InferenceSurvivalDepCensTransformRegr$clone()`

`InferenceSurvivalDepCensTransformRegr$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceSurvivalDepCensTransformRegr$supports_rand_pval_for_incidence()
```

`InferenceSurvivalDepCensTransformRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalDepCensTransformRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

`InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik` for a different (Clayton-copula, KK-design) approach to dependence between two survival-type quantities. Survival analysis (Wikipedia, general orientation; no direct Wikipedia page for dependent-censoring copula/transformation models specifically).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'survival')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalDepCensTransformRegr$new(seq_des)
inf$compute_estimate()
```

`InferenceSurvivalGehanWilcox`

Gehan-Wilcoxon (Peto-Prentice) Inference for Survival Data with Censoring

---

Description

Non-parametric inference for survival outcomes supporting censored data, using the Peto-Prentice modification of the Gehan-Wilcoxon test. The treatment effect estimate is the mean difference in Peto-Prentice weighted martingale residuals between the treatment and control groups. Specifically, for each subject the weighted residual is $M_i^w = \hat{S}(t_i^-) \cdot M_i$, where $M_i = \delta_i - \hat{\Lambda}_0(t_i)$ is the martingale residual and $\hat{S}(t_i^-)$ is the overall Kaplan-Meier survival estimate just before time t_i. These weights downweight late events, analogously to the Wilcoxon rank-sum test for uncensored data (which also weights early observations more heavily via their larger rank denominator).

The p-value uses `survival::survdiff(rho = 1)` (Peto-Prentice / Fleming-Harrington p=1, q=0), which is distinct from the log-rank test (`rho = 0`) used in `InferenceSurvivalKMDiff`.

Super class

`Inference` -> `InferenceSurvivalGehanWilcox`

Methods

Public methods:

- `InferenceSurvivalGehanWilcox$new()`
- `InferenceSurvivalGehanWilcox$compute_estimate()`
- `InferenceSurvivalGehanWilcox$compute_estimate_with_bootstrap_weights()`
- `InferenceSurvivalGehanWilcox$compute_asymp_confidence_interval()`
- `InferenceSurvivalGehanWilcox$compute_asymp_two_sided_pval()`
- `InferenceSurvivalGehanWilcox$compute_rand_confidence_interval()`
- `InferenceSurvivalGehanWilcox$clone()`

`InferenceSurvivalGehanWilcox$new():` Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize Gehan-Wilcoxon survival inference and prepare the rank-based treatment statistic used by `InferenceSurvivalGehanWilcox`.

Usage:

```
InferenceSurvivalGehanWilcox$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE
)
```

Arguments:

`des_obj` The design object.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` If TRUE, print additional information.

`InferenceSurvivalGehanWilcox$compute_estimate():` Returns the mean difference in Peto-Prentice weighted martingale residuals between the treatment and control groups. Positive values

indicate that treatment subjects experienced fewer early events than expected. For left- or interval-censored data, dispatches instead to `interval::ictest(...,scores = "wmw")` (the Wilcoxon-Mann-Whitney interval-censored generalization of the Peto-Prentice test) and returns its estimate.

Usage:

```
InferenceSurvivalGehanWilcox$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

Returns: A numeric scalar (the Peto-Prentice weighted score treatment effect estimate).

Examples:

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "survival")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2:10])
seq_des$add_all_subject_responses(
  ys = c(4.71, NA, 4.78, 6.11, NA, 8.43),
  y_Ls = c(NA, 1.23, NA, NA, 5.95, NA),
  y_Rs = c(NA, Inf, NA, NA, Inf, NA)
)

seq_des_inf = InferenceSurvivalGehanWilcox$new(seq_des)
seq_des_inf$compute_estimate()
```

`InferenceSurvivalGehanWilcox$compute_estimate_with_bootstrap_weights()`**:** Recomputes the class-specific treatment estimate under bootstrap weights; see `InferenceBayesianBootstrap`.

Usage:

```
InferenceSurvivalGehanWilcox$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferenceSurvivalGehanWilcox$compute_asymp_confidence_interval()`**:** Computes a (1 - alpha)-level confidence interval based on the asymptotic normality of the Peto-Prentice weighted martingale residual mean difference. Falls back to bootstrap if the SE is unavailable.

Usage:

```
InferenceSurvivalGehanWilcox$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level. Default is 0.05.

Returns: A numeric vector of length 2: (lower, upper) confidence bounds.

Examples:

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "survival")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2:10])
seq_des$add_all_subject_responses(
  ys = c(4.71, NA, 4.78, 6.11, NA, 8.43),
  y_Ls = c(NA, 1.23, NA, NA, 5.95, NA),
  y_Rs = c(NA, Inf, NA, NA, Inf, NA)
)

seq_des_inf = InferenceSurvivalGehanWilcox$new(seq_des)
seq_des_inf$compute_asymp_confidence_interval()
```

`InferenceSurvivalGehanWilcox$compute_asymp_two_sided_pval()`**:** Computes the Peto-Prentice (Gehan-Wilcoxon) two-sided p-value via `survival::survdiff(rho = 1)`, which puts greater weight on early events relative to the standard log-rank test (`rho = 0`). For delta != 0, not yet implemented.

Usage:

```
InferenceSurvivalGehanWilcox$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect to test against. Default is 0.

Returns: A p-value in [0, 1].

Examples:

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "survival")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2:10])
seq_des$add_all_subject_responses(
  ys = c(4.71, NA, 4.78, 6.11, NA, 8.43),
  y_Ls = c(NA, 1.23, NA, NA, 5.95, NA),
  y_Rs = c(NA, Inf, NA, NA, Inf, NA)
)

seq_des_inf = InferenceSurvivalGehanWilcox$new(seq_des)
seq_des_inf$compute_asymp_two_sided_pval()
```

`InferenceSurvivalGehanWilcox$compute_rand_confidence_interval()`**:** Randomization confidence intervals are not supported for this class because the Peto-Prentice weighted score scale is not commensurate with the time-ratio null used by the randomization CI bisection algorithm.

Usage:

```
InferenceSurvivalGehanWilcox$compute_rand_confidence_interval(
  alpha = 0.05,
  r = 501,
  pval_epsilon = 0.005,
  show_progress = TRUE,
  ci_search_control = NULL
)
```

Arguments:

`alpha` Unused.

`r` Unused.

`pval_epsilon` Unused.

`show_progress` Unused.

`ci_search_control` Unused.

`InferenceSurvivalGehanWilcox$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalGehanWilcox$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Gehan, E. A. (1965). "A generalized Wilcoxon test for comparing arbitrarily singly-censored samples." *Biometrika*, 52(1-2), 203-223, doi:10.1093/biomet/52.12.203, for the original generalized (Gehan) Wilcoxon test for censored data. Peto, R., and Peto, J. (1972). "Asymptotically Efficient Rank Invariant Test Procedures." *Journal of the Royal Statistical Society, Series A*, 135(2), 185-207, doi:10.2307/2344317, for the survival-weighted (Peto-Prentice) modification this class implements via $\rho = 1$.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'survival')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalGehanWilcox$new(seq_des)
inf$compute_estimate()


## ------------------------------------------------
## Method `InferenceSurvivalGehanWilcox$compute_estimate()`
## ------------------------------------------------

seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "survival")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2:10])
```

```
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2:10])
seq_des$add_all_subject_responses(
  ys = c(4.71, NA, 4.78, 6.11, NA, 8.43),
  y_Ls = c(NA, 1.23, NA, NA, 5.95, NA),
  y_Rs = c(NA, Inf, NA, NA, Inf, NA)
)

seq_des_inf = InferenceSurvivalGehanWilcox$new(seq_des)
seq_des_inf$compute_estimate()

## ------------------------------------------------
## Method `InferenceSurvivalGehanWilcox$compute_asymp_confidence_interval()`
## ------------------------------------------------

seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "survival")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2:10])
seq_des$add_all_subject_responses(
  ys = c(4.71, NA, 4.78, 6.11, NA, 8.43),
  y_Ls = c(NA, 1.23, NA, NA, 5.95, NA),
  y_Rs = c(NA, Inf, NA, NA, Inf, NA)
)

seq_des_inf = InferenceSurvivalGehanWilcox$new(seq_des)
seq_des_inf$compute_asymp_confidence_interval()

## ------------------------------------------------
## Method `InferenceSurvivalGehanWilcox$compute_asymp_two_sided_pval()`
## ------------------------------------------------

seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "survival")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2:10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2:10])
seq_des$add_all_subject_responses(
  ys = c(4.71, NA, 4.78, 6.11, NA, 8.43),
  y_Ls = c(NA, 1.23, NA, NA, 5.95, NA),
  y_Rs = c(NA, Inf, NA, NA, Inf, NA)
)

seq_des_inf = InferenceSurvivalGehanWilcox$new(seq_des)
seq_des_inf$compute_asymp_two_sided_pval()
```

InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC

Clayton Copula / Standard Weibull Compound Inference for KK Designs

Description

This class implements a compound estimator for KK matching-on-the-fly designs with survival responses using a Clayton copula with Weibull AFT margins for matched pairs and a standard Weibull AFT model for the reservoir. The two treatment-effect estimates (on the log-time ratio scale) are combined by inverse-variance weighting.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Frailty distribution. The Clayton copula for a matched pair, `S(t1,t2) = (S1(t1)^-theta + S2(t2)^-theta - 1)^(-1/theta)` with Weibull margins `S_i`, is exactly the closed-form bivariate survival function obtained by multiplying two conditionally-independent Weibull hazards by a shared **gamma** frailty term `Z ~ Gamma(1/theta, 1/theta)` and integrating `Z` out analytically (Clayton 1978; Oakes 1989); `theta` is the frailty variance / dependence parameter (see `ClaytonWeibullLikelihood` in `fast_survival_models_optim.cpp`, which builds the likelihood from the per-subject Weibull cumulative hazards `H1, H2`). This is the classic textbook Weibull-gamma shared-frailty model, fit here in its closed form (no numerical integration required) rather than as an AFT Gaussian-random-intercept model.

This is a different (and equally standard) frailty assumption from the **log-normal**-frailty Weibull AFT GLMM implemented by `InferenceSurvivalGLMMWeibullFrailtyNormalIVWC` / `InferenceSurvivalGLMMWeibullF` which instead places a Gaussian random intercept on the log-time (AFT) scale and integrates it out by Gauss-Hermite quadrature. Prefer this Clayton-copula class for the classic gamma-frailty / proportional-hazards dependence structure; prefer the Weibull-frailty class for a Gaussian-random-intercept / GLMM-style dependence structure.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC

Methods

Public methods:

- InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$approximate_randomization_distribution_beta_ha
- InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$supports_rand_pval_for_incidence()
- InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$compute_rand_two_sided_pval()
- InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$clone()

InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$approximate_randomization_distribution_beta_hat_T(

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$approximate_randomization_distribution_beta_hat_
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$supports_rand_pval_for_incidence():

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$supports_rand_pval_for_incidence(

)
```

`InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$compute_rand_two_sided_pval()`:

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Clayton DG (1978). "A Model for Association in Bivariate Life Tables and Its Application in Epidemiological Studies of Familial Tendency in Chronic Disease Incidence." Biometrika, 65(1), 141-151. doi:10.2307/2335289

Oakes D (1989). "Bivariate Survival Models Induced by Frailties." Journal of the American Statistical Association, 84(406), 487-493. doi:10.2307/2289934

Legacy class. Not fully tested in `comprehensive_tests.R`.

See Also

InferenceSurvivalGLMMWeibullFrailtyNormalIVWC for the corresponding log-normal-frailty IVWC estimator.

InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik

One-Likelihood Clayton-Copula Weibull AFT Inference for KK Survival Designs

---

Description

Estimates a treatment log-time-ratio β_T for right-censored survival outcomes collected under a KK matching-on-the-fly design (DesignSeqOneByOneKK14 or subclass) by maximizing a single combined likelihood: matched-pair survival times are modeled with a Weibull accelerated-failure-time (AFT) margin joined by a Clayton copula (dependence parameter θ) to account for within-pair correlation induced by shared matching covariates, while unmatched reservoir subjects are modeled by the same Weibull AFT margin marginally (no dependence term). All subjects share one treatment coefficient, estimated jointly.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Estimand. β_T, the treatment coefficient of a Weibull AFT model $\log T = \beta_0 + \beta_T W + X\beta + \sigma\epsilon$ with ϵ extreme-value-distributed; $\exp(\hat{\beta}_T)$ is the treatment-vs-control survival-time ratio (an acceleration factor). This is a distinct scale from the log hazard ratio reported by Cox-based KK survival classes.

Model. `.fit_clayton_weibull_aft()` jointly optimizes the AFT regression coefficients, the Weibull shape ($\log\sigma$), and the Clayton copula dependence parameter ($\log\theta$) by direct maximum likelihood over the combined matched-pair-copula / reservoir-marginal log-likelihood; right-censoring enters as the usual survival contribution (density for observed failures, survival function for censored times). `likelihood_tier = "full"`, so a parametric likelihood bootstrap (`simulate_under_lik_null`, which draws new pair times from the fitted Clayton copula and new singleton times from the marginal Weibull) is available alongside Wald inference.

Assumptions. Weibull AFT margin correctly specified; Clayton copula correctly captures within-pair dependence (a positive-dependence, single-parameter Archimedean copula); independent censoring given covariates; a KK matching-on-the-fly design supplying the matched/ reservoir partition.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik

Methods

Public methods:

- InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$approximate_randomization_distribution_beta_
- InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$supports_rand_pval_for_incidence()
- InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$compute_rand_two_sided_pval()
- InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$clone()

InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$approximate_randomization_distribution_beta_hat_

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$approximate_randomization_distribution_beta_ha
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

r Number of randomization vectors. Default 501.

r Number of randomization vectors.

delta The null difference. Default 0.

delta Null difference.

transform_responses Type of transformation. Default "none".

transform_responses Transformation.

show_progress Show progress bar. Default TRUE.

show_progress Show progress.

permutations Pre-computed permutations. Default NULL.

permutations Pre-computed permutations.

debug If TRUE, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default FALSE.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$supports_rand_pval_for_incidence():`

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$supports_rand_pval_for_incidence(

)
```

`InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$compute_rand_two_sided_pval():`

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Clayton, D. G. (1978). "A model for association in bivariate life tables and its application in epidemiological studies of familial tendency in chronic disease incidence." *Biometrika*, 65(1), 141-151. doi:10.1093/biomet/65.1.141. (Clayton1978 in `REFERENCES.md`.)

Oakes, D. (1989). "Bivariate survival models induced by frailties." *Journal of the American Statistical Association*, 84(406), 487-493. doi:10.1080/01621459.1989.10478795. (Oakes1989 in `REFERENCES.md`.)

See Also

Analogous Python API for AFT/copula survival models: lifelines WeibullAFTFitter, copulas. Copula (probability theory) (orientation).

`InferenceSurvivalGLMMWeibullFrailtyNormalIVWC`
Weibull Frailty IVWC Inference for KK Designs

Description

Log-normal (Gaussian random-intercept) frailty Weibull AFT estimator; see `InferenceSurvivalGLMMWeibullFrailtyNorm` for the frailty-distribution details and contrast with the gamma-frailty `InferenceSurvivalGLMMWeibullFrailtyLoggammaI` (Clayton copula) alternative.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Legacy class. Not fully tested in `comprehensive_tests.R`.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceSurvivalGLMMWeibullFrailtyNormalIVWC

Methods

Public methods:

- InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$approximate_randomization_distribution_beta_hat_
- InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$supports_rand_pval_for_incidence()
- InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$compute_rand_two_sided_pval()
- InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$clone()

InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$supports_rand_pval_for_incidence():

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$supports_rand_pval_for_incidence(

)
```

`InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$compute_rand_two_sided_pval():`

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyNormalIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

InferenceSurvivalGLMMWeibullFrailtyNormalOneLik

Weibull Frailty Combined-Likelihood Inference for KK Designs

Description

Log-normal (Gaussian random-intercept) frailty Weibull AFT estimator; see InferenceSurvivalGLMMWeibullFrailtyNorm for the frailty-distribution details and contrast with the gamma-frailty InferenceSurvivalGLMMWeibullFrailtyLoggammaO (Clayton copula) alternative.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether compute_rand_two_sided_pval() is actually usable on this instance right now – FALSE exactly when it would stop(): an incidence-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see private$should_use_design_ra TRUE for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so InferenceSuite can check this before attempting the sentinel instead of relying on the stop() being silently swallowed into a pval = NA "ok" row – the single source of truth for both this check and compute_rand_two_sided_pval()'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceSurvivalGLMMWeibullFrailtyNormalOneLik

Methods

Public methods:

- InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$approximate_randomization_distribution_beta_ha
- InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$supports_rand_pval_for_incidence()
- InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$compute_rand_two_sided_pval()
- InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$clone()

InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$approximate_randomization_distribution_beta_hat_T(

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$approximate_randomization_distribution_beta_hat_
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$supports_rand_pval_for_incidence(

)
```

`InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$compute_rand_two_sided_pval()`:

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalGLMMWeibullFrailtyNormalOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`InferenceSurvivalKKLWACoxPHIVWC`
LWA-style Marginal Cox IVWC Compound Inference for KK Designs

---

Description

Fits a compound (IVWC) estimator for KK matching-on-the-fly designs with survival responses: matched pairs are analyzed with a marginal Cox model $\lambda(t \mid w) = \lambda_0(t)\exp(\beta_T w)$ whose robust variance uses the Lee-Wei-Amato (1992) cluster-robust sandwich (treating each matched pair as an independent cluster of correlated failure times), while reservoir subjects are analyzed with a standard (independent-subjects) Cox partial likelihood; the two log-hazard-ratio estimates are then combined by inverse-variance weighting. `likelihood_tier = "partial"` (Cox partial likelihood), but likelihood-ratio/score/gradient tests are not exposed on this IVWC compound (only on the `OneLik` sibling, which fits one combined partial likelihood across both sources instead of pooling two separate fits).

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Legacy class. Not fully tested in `comprehensive_tests.R`.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceSurvivalKKLWACoxPHIVWC

Methods

Public methods:

- InferenceSurvivalKKLWACoxPHIVWC$approximate_randomization_distribution_beta_hat_T()
- InferenceSurvivalKKLWACoxPHIVWC$supports_rand_pval_for_incidence()
- InferenceSurvivalKKLWACoxPHIVWC$compute_rand_two_sided_pval()
- InferenceSurvivalKKLWACoxPHIVWC$clone()

InferenceSurvivalKKLWACoxPHIVWC$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceSurvivalKKLWACoxPHIVWC$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

r Number of randomization vectors. Default 501.

r Number of randomization vectors.

delta The null difference. Default 0.

delta Null difference.

transform_responses Type of transformation. Default "none".

transform_responses Transformation.

show_progress Show progress bar. Default TRUE.

show_progress Show progress.

permutations Pre-computed permutations. Default NULL.

permutations Pre-computed permutations.

debug If TRUE, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default FALSE.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalKKLWACoxPHIVWC$supports_rand_pval_for_incidence():`

Usage:

```
InferenceSurvivalKKLWACoxPHIVWC$supports_rand_pval_for_incidence()
```

`InferenceSurvivalKKLWACoxPHIVWC$compute_rand_two_sided_pval():`

Usage:

```
InferenceSurvivalKKLWACoxPHIVWC$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalKKLWACoxPHIVWC$clone():` The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalKKLWACoxPHIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Lee, E. W., Wei, L. J., and Amato, D. A. (1992). "Cox-Type Regression Analysis for Large Numbers of Small Groups of Correlated Failure Time Observations." In *Survival Analysis: State of the Art*, 237-247. Springer. doi:10.1007/9789401579834_14

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'survival')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalKKLWACoxPHIVWC$new(seq_des)
inf$compute_estimate()
```

InferenceSurvivalKKLWACoxPHOneLik

LWA-style Marginal Cox Combined-Likelihood Inference for KK Designs

---

Description

Fits a single combined Cox partial likelihood $\lambda(t \mid w, x) = \lambda_0(t)\exp(\beta_T w + \beta_X^\top x)$ jointly over matched-pair and reservoir subjects for KK matching-on-the-fly designs with survival responses (a marginal, not stratified, Cox model: matched pairs do not get pair-specific baseline hazards). This is the one-likelihood combined-fit analog of `InferenceSurvivalKKLWACoxPHIVWC`, which instead fits and pools two separate estimators. `likelihood_tier = "partial"`: exposes likelihood-ratio and parametric-likelihood-bootstrap inference in addition to Wald/asymptotic and Bayesian-bootstrap paths.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceSurvivalKKLWACoxPHOneLik

Methods

Public methods:

- InferenceSurvivalKKLWACoxPHOneLik$approximate_randomization_distribution_beta_hat_T()
- InferenceSurvivalKKLWACoxPHOneLik$supports_rand_pval_for_incidence()
- InferenceSurvivalKKLWACoxPHOneLik$compute_rand_two_sided_pval()
- InferenceSurvivalKKLWACoxPHOneLik$clone()

InferenceSurvivalKKLWACoxPHOneLik$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceSurvivalKKLWACoxPHOneLik$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

InferenceSurvivalKKLWACoxPHOneLik$supports_rand_pval_for_incidence():

Usage:

```
InferenceSurvivalKKLWACoxPHOneLik$supports_rand_pval_for_incidence()
```

InferenceSurvivalKKLWACoxPHOneLik$compute_rand_two_sided_pval():

Usage:

```
InferenceSurvivalKKLWACoxPHOneLik$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalKKLWACoxPHOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalKKLWACoxPHOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Cox, D. R. (1972). "Regression Models and Life-Tables." *Journal of the Royal Statistical Society, Series B*, 34(2), 187-220.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'survival')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalKKLWACoxPHOneLik$new(seq_des)
inf$compute_estimate()
```

InferenceSurvivalKKRankRegrIVWC

Rank Regression Inference for Survival Responses under KK Designs

Description

Fits a multivariate Gehan-Wilcoxon rank regression for survival outcomes under a KK matching-on-the-fly design. The model adjusts for the treatment indicator and, optionally, all recorded covariates.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Legacy class. Not fully tested in `comprehensive_tests.R`.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceSurvivalKKRankRegrIVWC`

Methods

Public methods:

- `InferenceSurvivalKKRankRegrIVWC$approximate_randomization_distribution_beta_hat_T()`
- `InferenceSurvivalKKRankRegrIVWC$supports_rand_pval_for_incidence()`
- `InferenceSurvivalKKRankRegrIVWC$compute_rand_two_sided_pval()`
- `InferenceSurvivalKKRankRegrIVWC$clone()`

`InferenceSurvivalKKRankRegrIVWC$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceSurvivalKKRankRegrIVWC$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalKKRankRegrIVWC$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceSurvivalKKRankRegrIVWC$supports_rand_pval_for_incidence()
```

`InferenceSurvivalKKRankRegrIVWC$compute_rand_two_sided_pval()`:

Usage:

```
InferenceSurvivalKKRankRegrIVWC$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalKKRankRegrIVWC$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalKKRankRegrIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneKK14$new(n = 10, response_type = 'survival')
for (i in 1:10) {
 seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1), x2 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalKKRankRegrIVWC$new(seq_des)
inf$compute_estimate()
```

InferenceSurvivalKKStratCoxPHIVWC

Stratified Cox / Standard Cox Compound Inference for KK Designs

---

Description

This class implements a compound estimator for KK matching-on-the-fly designs with survival responses. For matched pairs, it uses stratified Cox proportional hazards regression (each pair is a stratum). For reservoir subjects, it uses standard Cox regression. The two estimates (both log-hazard ratios) are combined via a variance-weighted linear combination.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra`

TRUE for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so InferenceSuite can check this before attempting the sentinel instead of relying on the stop() being silently swallowed into a pval = NA "ok" row – the single source of truth for both this check and compute_rand_two_sided_pval()'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Details

Under harden = TRUE, multivariate fits preserve the treatment column and progressively retry reduced covariate sets after QR-based rank reduction and correlation-based pruning. Extreme finite coefficients / standard errors are rejected and treated as non-estimable.

The matched-pair sub-estimate treats each pair as its own stratum (a pair-specific baseline hazard, exactly canceling shared-frailty effects within the pair via Cox's partial likelihood) and is a special case of the Lee-Wei-Amato (1992) large-numbers-of-small-groups stratified Cox approach; the reservoir sub-estimate is a standard unstratified Cox partial-likelihood fit (Cox 1972). The two log-hazard-ratio estimates are combined by inverse-variance weighting, the same rule used throughout the KK IVWC family.

Legacy class. Not fully tested in comprehensive_tests.R.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Randomization p-value.

Super class

Inference -> InferenceSurvivalKKStratCoxPHIVWC

Methods

Public methods:

- InferenceSurvivalKKStratCoxPHIVWC$approximate_randomization_distribution_beta_hat_T()
- InferenceSurvivalKKStratCoxPHIVWC$supports_rand_pval_for_incidence()
- InferenceSurvivalKKStratCoxPHIVWC$compute_rand_two_sided_pval()
- InferenceSurvivalKKStratCoxPHIVWC$clone()

InferenceSurvivalKKStratCoxPHIVWC$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceSurvivalKKStratCoxPHIVWC$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
```

```
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalKKStratCoxPHIVWC$supports_rand_pval_for_incidence()`**:**

Usage:

```
InferenceSurvivalKKStratCoxPHIVWC$supports_rand_pval_for_incidence()
```

`InferenceSurvivalKKStratCoxPHIVWC$compute_rand_two_sided_pval()`**:**

Usage:

```
InferenceSurvivalKKStratCoxPHIVWC$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalKKStratCoxPHIVWC$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalKKStratCoxPHIVWC$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Cox, D. R. (1972). "Regression Models and Life-Tables." Journal of the Royal Statistical Society, Series B, 34(2), 187-220.

Lee, E. W., Wei, L. J., and Amato, D. A. (1992). "Cox-Type Regression Analysis for Large Numbers of Small Groups of Correlated Failure Time Observations." In Survival Analysis: State of the Art, 237-247. Springer. doi:10.1007/9789401579834_14

`InferenceSurvivalKKStratCoxPHOneLik`

Stratified Cox Combined-Likelihood Compound Inference for KK Designs

---

Description

Fits a single combined partial likelihood for KK matching-on-the-fly designs with survival responses: matched pairs contribute a stratified Cox term (one stratum per pair, canceling shared-frailty effects within the pair) and reservoir subjects contribute a standard unstratified Cox term, summed into one joint partial log-likelihood and optimized jointly for a single shared treatment coefficient. This differs from the two-stage IVWC sibling `InferenceSurvivalKKStratCoxPHIVWC`, which fits the matched and reservoir sub-models separately and combines the two log-hazard-ratio estimates by inverse-variance weighting; this class instead estimates one coefficient from the combined likelihood directly, which additionally supports likelihood-ratio tests and parametric likelihood bootstrap (`likelihood_tier = "partial"`).

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only

reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Computes a randomization-based p-value.

Value

When `debug = FALSE` (default), a numeric vector of length `r`. When `debug = TRUE`, a list with: `values`, `errors` (list of character vectors, one per iteration), `warnings` (list of character vectors, one per iteration), `num_errors`, `num_warnings`, `prop_iterations_with_errors`, `prop_iterations_with_warnings`, and `prop_illegal_values`.

A single logical.

Randomization p-value.

Super class

`Inference` -> `InferenceSurvivalKKStratCoxPHOneLik`

Methods

Public methods:

- `InferenceSurvivalKKStratCoxPHOneLik$approximate_randomization_distribution_beta_hat_T()`
- `InferenceSurvivalKKStratCoxPHOneLik$supports_rand_pval_for_incidence()`
- `InferenceSurvivalKKStratCoxPHOneLik$compute_rand_two_sided_pval()`
- `InferenceSurvivalKKStratCoxPHOneLik$clone()`

`InferenceSurvivalKKStratCoxPHOneLik$approximate_randomization_distribution_beta_hat_T()`:

Usage:

```
InferenceSurvivalKKStratCoxPHOneLik$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`debug` If `TRUE`, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default `FALSE`.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalKKStratCoxPHOneLik$supports_rand_pval_for_incidence()`:

Usage:

```
InferenceSurvivalKKStratCoxPHOneLik$supports_rand_pval_for_incidence()
```

`InferenceSurvivalKKStratCoxPHOneLik$compute_rand_two_sided_pval()`:

Usage:

```
InferenceSurvivalKKStratCoxPHOneLik$compute_rand_two_sided_pval(
  r = 501,
  delta = 0,
  transform_responses = "none",
  na.rm = TRUE,
  show_progress = TRUE,
  permutations = NULL,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

`r` Number of randomization vectors. Default 501.

`r` Number of randomization vectors.

`delta` The null difference. Default 0.

`delta` Null difference.

`transform_responses` Type of transformation. Default "none".

`transform_responses` Transformation.

`na.rm` Remove NAs.

`show_progress` Show progress bar. Default TRUE.

`show_progress` Show progress.

`permutations` Pre-computed permutations. Default NULL.

`permutations` Pre-computed permutations.

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`zero_one_logit_clamp` The clamping amount for exact 0 and 1 values when logging

`InferenceSurvivalKKStratCoxPHOneLik$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalKKStratCoxPHOneLik$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Cox, D. R. (1972). "Regression Models and Life-Tables." Journal of the Royal Statistical Society, Series B, 34(2), 187-220.

Lee, E. W., Wei, L. J., and Amato, D. A. (1992). "Cox-Type Regression Analysis for Large Numbers of Small Groups of Correlated Failure Time Observations." In Survival Analysis: State of the Art, 237-247. Springer. doi:10.1007/9789401579834_14

`InferenceSurvivalKKWeibullMarginal`

Marginal (Cluster-Robust) Weibull Inference for KK Matched-Pair Survival Designs

---

Description

Initialize the marginal (cluster-robust) Weibull inference object.

Returns the pooled treatment effect estimate (log-time-ratio scale).

Recomputes the treatment estimate under Bayesian-bootstrap weights.

Computes the asymptotic (cluster-robust) confidence interval.

Computes the asymptotic (cluster-robust) two-sided p-value.

Duplicates this subclass while preserving fit caches; see `Inference`.

Usage

```
SurvivalKKWeibullMarginalSource
```

Details

Fits a single pooled Weibull Accelerated Failure Time (AFT) model across all subjects (treatment plus, optionally, all recorded covariates), ignoring the matched-pair structure in the mean model. Standard errors are computed via a cluster-robust (sandwich) covariance estimator: matched pairs from a KK matching-on-the-fly or binary-match design form size-2 clusters, and unmatched reservoir subjects each form their own singleton cluster.

This is the "marginal" competitor to `InferenceSurvivalGLMMWeibullFrailtyNormalOneLik`: rather than modeling the within-pair correlation explicitly via a frailty term, it fits an ordinary (working-independence) Weibull AFT model and corrects the treatment-effect standard error post hoc for the within-pair dependence. The model is fit via the package's fast C++ Weibull AFT backend (`fast_weibull_regression_general_cpp`) and the cluster-robust sandwich is assembled from per-subject dfbeta contributions collapsed within clusters, which is numerically equivalent to `survival::survreg(..., cluster = ..., robust = TRUE)` (retained as a fallback if the C++ fit fails to converge).

Examples

```
des = DesignSeqOneByOneKK14$new(n = 20, response_type = 'survival')
for (i in 1:20) {
  des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
des$add_all_subject_responses(rexp(20))
inf = InferenceSurvivalKKWeibullMarginal$new(des)
inf$compute_estimate()
```

InferenceSurvivalKMDiff

Kaplan-Meier Median-Difference Inference for Survival Responses

---

Description

Fits a non-parametric treatment-effect estimator for censored survival responses: the difference in Kaplan-Meier **median** survival times between the treated and control arms, $\hat{m}_T - \hat{m}_C$. Standard errors are obtained by **back-calculating** from each arm's separate Brookmeyer-Crowley confidence interval for its own median (via `survival::survfit`'s default log-log-transformed interval): $\hat{\sigma}_i = (\text{upper}_i - \text{lower}_i)/(2z_{\alpha/2})$, combined (the two arms are independent by design) as $\sqrt{\hat{\sigma}_T^2 + \hat{\sigma}_C^2}$. When either arm's median is inestimable (its Kaplan-Meier curve never reaches 0.5) or the back-calculated bounds are non-finite, the Wald-style confidence interval and p-value methods (`$compute_asymp_confidence_interval()`, `$compute_asymp_two_sided_pval()`) silently fall back to a nonparametric bootstrap instead of returning `NA`. A convenience method, `$compute_asymp_log_rank_two_sided_p` is also provided for the log-rank p-value on the same fitted survival curves. For **left- or interval-censored** data, the point estimate instead comes from a Turnbull NPMLE median contrast (`interval::icfit()`, via `turnbull_npmle_stat_diff()`), which has no closed-form standard error — inference on that path relies entirely on the bootstrap fallback described above. Randomization confidence intervals are not supported (the median difference's units are not commensurate with the randomization CI bisection algorithm's transformed-scale null search).

Super class

`Inference` -> `InferenceSurvivalKMDiff`

Methods

Public methods:

- `InferenceSurvivalKMDiff$new()`
- `InferenceSurvivalKMDiff$compute_estimate()`
- `InferenceSurvivalKMDiff$compute_estimate_with_bootstrap_weights()`
- `InferenceSurvivalKMDiff$compute_bayesian_bootstrap_two_sided_pval()`
- `InferenceSurvivalKMDiff$compute_asymp_confidence_interval()`
- `InferenceSurvivalKMDiff$compute_asymp_two_sided_pval()`
- `InferenceSurvivalKMDiff$compute_asymp_log_rank_two_sided_pval_for_treatment_effect()`

- InferenceSurvivalKMDiff$compute_rand_confidence_interval()
- InferenceSurvivalKMDiff$clone()

InferenceSurvivalKMDiff$new(): Uses the shared randomization two-sided p-value contract; see InferenceRand.

Initialize Kaplan-Meier median-difference survival inference and prepare treatment-group survival curves used by InferenceSurvivalKMDiff.

Usage:

```
InferenceSurvivalKMDiff$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

des_obj The design object.

model_formula Optional formula for covariate adjustment. If NULL (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

verbose If TRUE, print additional information.

smart_cold_start_default Whether to use smart cold start values by default.

InferenceSurvivalKMDiff$compute_estimate(): Computes the class-specific mean or survival contrast; see InferenceMLEorKMSummaryTable.

Usage:

```
InferenceSurvivalKMDiff$compute_estimate(estimate_only = FALSE)
```

Arguments:

estimate_only If TRUE, skip variance component calculations.

Returns: The setting-appropriate (see description) numeric estimate of the treatment effect

Examples:

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "survival")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2 : 10])
seq_des$add_all_subject_responses(
  ys = c(4.71, NA, 4.78, 6.11, NA, 8.43),
  y_Ls = c(NA, 1.23, NA, NA, 5.95, NA),
  y_Rs = c(NA, Inf, NA, NA, Inf, NA)
)

seq_des_inf = InferenceSurvivalKMDiff$new(seq_des)
seq_des_inf$compute_estimate()
```

`InferenceSurvivalKMDiff$compute_estimate_with_bootstrap_weights()`: Recomputes the class-specific treatment estimate for a bootstrap sample; see `InferenceNonParamBootstrap`.

Usage:

```
InferenceSurvivalKMDiff$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Row weights for the bootstrap sample.

`estimate_only` If TRUE, skip variance calculations.

`InferenceSurvivalKMDiff$compute_bayesian_bootstrap_two_sided_pval()`: Computes a Bayesian-bootstrap two-sided p-value for the difference in Kaplan-Meier medians.

A weighted Kaplan-Meier curve can fail to reach 0.5 even when the original sample has finite medians in both arms. Such a draw has no finite median-difference estimate. This class therefore drops non-finite weighted-median draws by default and requires at least `min_number_usable_samples` finite draws. If too few remain, the result is `NA_real_` with nonestimability reason `"bayesian_bootstrap_too_few_finit` Set `na.rm = FALSE` to retain the generic fail-on-any-nonfinite policy.

Usage:

```
InferenceSurvivalKMDiff$compute_bayesian_bootstrap_two_sided_pval(
  delta = 0,
  B = 501,
  type = NULL,
  na.rm = TRUE,
  show_progress = TRUE,
  min_number_usable_samples = 5L,
  weighting_unit_type = NULL
)
```

Arguments:

`delta` Null median difference. Default 0.

`B` Number of Bayesian-bootstrap replicates. Default 501.

`type` Bayesian-bootstrap p-value type.

`na.rm` Whether to discard non-finite weighted-median draws. Defaults to `TRUE` for this estimator.

`show_progress` Whether to display a progress bar.

`min_number_usable_samples` Minimum number of finite draws required.

`weighting_unit_type` Optional Bayesian-bootstrap weighting-unit scheme.

Returns: A numeric two-sided p-value, or `NA_real_` when too few finite draws remain.

`InferenceSurvivalKMDiff$compute_asymp_confidence_interval()`: Computes a (1 - alpha)-level confidence interval for the difference in Kaplan-Meier median survival times (treatment minus control).

The Brookmeyer-Crowley confidence interval is obtained for each group's median separately via `survival::survfit` (using a log-log transformation of the survival function by default). The per-group SE is back-calculated from the CI half-width as $\hat{\sigma}_i = (\text{upper}_i - \text{lower}_i)/(2z_{\alpha/2})$. The

two groups are independent by design, so the SE of the difference is $\sqrt{\hat{\sigma}_T^2 + \hat{\sigma}_C^2}$, and the CI is $(\hat{m}_T - \hat{m}_C) \pm z_{\alpha/2} \cdot \sqrt{\hat{\sigma}_T^2 + \hat{\sigma}_C^2}$.
Falls back to `compute_bootstrap_confidence_interval` when either group's median is not estimable (i.e., the Kaplan-Meier curve does not reach 0.5) or when the Brookmeyer-Crowley CI bounds are `NA`.

Usage:

```
InferenceSurvivalKMDiff$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` The significance level; the confidence level is 1 - `alpha`. Default is 0.05.

Returns: A numeric vector of length 2 giving the (lower, upper) confidence bounds for the difference in median survival times, on the original time scale.

`InferenceSurvivalKMDiff$compute_asymp_two_sided_pval()`: Computes a Wald-style 2-sided p-value based on the median difference and its back-calculated standard error.

Usage:

```
InferenceSurvivalKMDiff$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null difference to test against. Default is 0.

Returns: The approximate frequentist p-value

`InferenceSurvivalKMDiff$compute_asymp_log_rank_two_sided_pval_for_treatment_effect()`: Computes a 2-sided p-value via the log rank test

Usage:

```
InferenceSurvivalKMDiff$compute_asymp_log_rank_two_sided_pval_for_treatment_effect(
  delta = 0
)
```

Arguments:

`delta` The null difference to test against. For any treatment effect at all this is set to zero (the default).

Returns: The approximate frequentist p-value

`InferenceSurvivalKMDiff$compute_rand_confidence_interval()`: Uses the shared randomization confidence-interval contract; see `InferenceRandCI`.

Usage:

```
InferenceSurvivalKMDiff$compute_rand_confidence_interval(
  alpha = 0.05,
  r = 501,
  pval_epsilon = 0.005,
  show_progress = TRUE,
  ci_search_control = NULL
)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`r` The number of randomization vectors. The default is 501.

`pval_epsilon` The bisection algorithm tolerance. The default is 0.005.

`show_progress` Show a text progress indicator.

`ci_search_control` Unused.

Returns: A 1 - alpha sized frequentist confidence interval

`InferenceSurvivalKMDiff$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalKMDiff$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kaplan, E. L., and Meier, P. (1958). "Nonparametric Estimation from Incomplete Observations." *Journal of the American Statistical Association*, 53(282), 457-481, doi:10.2307/2281868, for the Kaplan-Meier survival curve estimator each arm's median is read from. Brookmeyer, R., and Crowley, J. (1982). "A Confidence Interval for the Median Survival Time." *Biometrics*, 38(1), 29-41, doi:10.2307/2530286, for the per-arm median confidence interval this class's standard error is back-calculated from. Turnbull, B. W. (1976). "The Empirical Distribution Function with Arbitrarily Grouped, Censored and Truncated Data." *Journal of the Royal Statistical Society, Series B*, 38(3), 290-295, doi:10.1111/j.25176161.1976.tb01597.x, for the NPMLE used on the left-/interval-censored path.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'survival')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalKMDiff$new(seq_des)
inf$compute_estimate()


## ------------------------------------------------
## Method `InferenceSurvivalKMDiff$compute_estimate()`
## ------------------------------------------------

seq_des = DesignSeqOneByOneBernoulli$new(n = 6, response_type = "survival")
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[1, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[2, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[3, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[4, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[5, 2 : 10])
seq_des$add_one_subject_to_experiment_and_assign(MASS::biopsy[6, 2 : 10])
seq_des$add_all_subject_responses(
  ys = c(4.71, NA, 4.78, 6.11, NA, 8.43),
  y_Ls = c(NA, 1.23, NA, NA, 5.95, NA),
```

```
  y_Rs = c(NA, Inf, NA, NA, Inf, NA)
)

seq_des_inf = InferenceSurvivalKMDiff$new(seq_des)
seq_des_inf$compute_estimate()
```

InferenceSurvivalLogRank

Log-Rank Inference for Survival Data with Censoring

Description

Non-parametric all-subject inference for survival outcomes supporting right censoring, based on the standard two-sample log-rank test. The treatment effect estimate is the difference in mean martingale residuals between the treatment and control groups under the pooled null hazard. The p-value uses the classic log-rank score statistic with its hypergeometric tie-adjusted variance. For left- or interval-censored data, this class dispatches instead to `interval::ictest(..., scores = "logrank1")` (Sun's-scores interval-censored generalization of the log-rank test) for both the point estimate and testing.

Super class

`Inference` -> `InferenceSurvivalLogRank`

Methods

Public methods:

- `InferenceSurvivalLogRank$new()`
- `InferenceSurvivalLogRank$compute_estimate()`
- `InferenceSurvivalLogRank$compute_estimate_with_bootstrap_weights()`
- `InferenceSurvivalLogRank$compute_asymp_confidence_interval()`
- `InferenceSurvivalLogRank$compute_asymp_two_sided_pval()`
- `InferenceSurvivalLogRank$compute_asymp_log_rank_two_sided_pval_for_treatment_effect()`
- `InferenceSurvivalLogRank$compute_rand_confidence_interval()`
- `InferenceSurvivalLogRank$clone()`

`InferenceSurvivalLogRank$new()`: Uses the shared randomization two-sided p-value contract; see `InferenceRand`.

Initialize log-rank survival inference and prepare the treatment-group survival data used by `InferenceSurvivalLogRank`.

Usage:

```
InferenceSurvivalLogRank$new(des_obj, model_formula = NULL, verbose = FALSE)
```

Arguments:

`des_obj` The design object.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

`verbose` If TRUE, print additional information.

`InferenceSurvivalLogRank$compute_estimate()`: Computes the treatment-effect estimate on the martingale-residual mean-difference scale. Under left-/interval-censored data, dispatched instead through `interval::ictest()`'s Sun's-scores log-rank test (TODO-7, interval_censored_survival_response.md); its `estimate` field (mean score difference between groups) is on the same "difference of group-mean scores" scale as the right-censored martingale-residual difference this method otherwise returns.

Usage:

```
InferenceSurvivalLogRank$compute_estimate(estimate_only = FALSE)
```

Arguments:

`estimate_only` If TRUE, skip variance component calculations.

`InferenceSurvivalLogRank$compute_estimate_with_bootstrap_weights()`: Recomputes the class-specific treatment estimate under bootstrap weights; see `InferenceBayesianBootstrap`.

Usage:

```
InferenceSurvivalLogRank$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Bootstrap weights at the subject or block level.

`estimate_only` If TRUE, skip variance calculations.

`InferenceSurvivalLogRank$compute_asymp_confidence_interval()`: Computes a (1 - alpha)-level confidence interval based on the asymptotic normality of the martingale-residual mean-difference estimate. Falls back to bootstrap if the estimated standard error is unavailable.

Usage:

```
InferenceSurvivalLogRank$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level.

`InferenceSurvivalLogRank$compute_asymp_two_sided_pval()`: Computes a Wald-style 2-sided p-value by inverting the confidence interval.

Usage:

```
InferenceSurvivalLogRank$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null difference to test against. Default is 0.

Returns: The approximate frequentist p-value

`InferenceSurvivalLogRank$compute_asymp_log_rank_two_sided_pval_for_treatment_effect()`: Computes the standard two-sided log-rank p-value for a zero treatment effect. Under left-/interval-censored data, this is `interval::ictest()`'s own p-value (TODO-7) rather than a re-derived chi-squared statistic.

Usage:

```
InferenceSurvivalLogRank$compute_asymp_log_rank_two_sided_pval_for_treatment_effect(
  delta = 0
)
```

Arguments:

`delta` Null treatment effect to test against. Only `0` is supported.

`InferenceSurvivalLogRank$compute_rand_confidence_interval()`: Randomization confidence intervals are not supported for this class because the martingale-residual score scale is not commensurate with the transformed time-ratio null used by the randomization CI algorithm.

Usage:

```
InferenceSurvivalLogRank$compute_rand_confidence_interval(
  alpha = 0.05,
  r = 501,
  pval_epsilon = 0.005,
  show_progress = TRUE,
  ci_search_control = NULL
)
```

Arguments:

`alpha` Unused.

`r` Unused.

`pval_epsilon` Unused.

`show_progress` Unused.

`ci_search_control` Unused.

`InferenceSurvivalLogRank$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalLogRank$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Mantel, N. (1966). "Evaluation of survival data and two new rank order statistics arising in its consideration." *Cancer Chemotherapy Reports*, 50(3), 163-170, for the log-rank test. Peto, R., and Peto, J. (1972). "Asymptotically Efficient Rank Invariant Test Procedures." *Journal of the Royal Statistical Society, Series A*, 135(2), 185-207, doi:10.2307/2344317, for its asymptotic-efficiency properties and the $\rho = 0$ case of the Fleming-Harrington family this class corresponds to (see also `InferenceSurvivalGehanWilcox` for the $\rho = 1$ member of the same family). Sun, J. (1996). "A non-parametric test for interval-censored failure time data with application to AIDS studies." *Statistics in Medicine*, 15(13), 1387-1395, for the interval-censored generalization used on the left-/interval-censored path.

Examples

```
set.seed(1)
x_dat <- data.frame(
  x1 = c(-1.2, -0.7, -0.2, 0.3, 0.8, 1.3, 1.8, 2.3),
  x2 = c(0, 1, 0, 1, 0, 1, 0, 1)
)
seq_des <- DesignSeqOneByOneBernoulli$
  new(
  n = nrow(x_dat),
  response_type = "survival",
  verbose = FALSE
)
for (i in seq_len(nrow(x_dat))) {
  seq_des$
  add_one_subject_to_experiment_and_assign(x_dat[i, , drop = FALSE])
}
seq_des$
  add_all_subject_responses(
  ys = c(1.2, 2.4, NA, 3.1, NA, 4.0, 3.3, NA),
  y_Ls = c(NA, NA, 1.8, NA, 2.7, NA, NA, 4.5),
  y_Rs = c(NA, NA, Inf, NA, Inf, NA, NA, Inf)
)
infer <- InferenceSurvivalLogRank$
  new(
  seq_des,
  verbose = FALSE
)
infer
```

InferenceSurvivalRestrictedMeanDiff

Restricted Mean Survival Time (RMST) Difference Inference for Survival Responses

---

Description

Fits a non-parametric treatment-effect estimator for censored survival responses: the difference in **restricted mean survival time** (RMST) between the treated and control arms, $\hat{\mu}_T(\tau) - \hat{\mu}_C(\tau)$, where both arms use the same truncation horizon τ, chosen as the smaller of their maximum follow-up times, and each RMST is the area under its Kaplan-Meier survival curve up to that horizon ($\hat{\mu}(\tau) = \int_0^\tau \hat{S}(t)\,dt$), computed by trapezoidal integration of the step-function KM curve. The standard error of the difference comes from the Greenwood-type variance of each arm's RMST, combined across the two (independent) arms via `get_restricted_mean_se_diff()`. When that standard error is unavailable or non-finite, `$compute_asymp_confidence_interval()` falls back to a nonparametric bootstrap interval rather than returning `NA`. Randomization confidence intervals are not supported (the RMST-difference units are not commensurate with the randomization CI bisection algorithm's transformed-scale null search).

Super class

Inference -> InferenceSurvivalRestrictedMeanDiff

Methods

Public methods:

- InferenceSurvivalRestrictedMeanDiff$new()
- InferenceSurvivalRestrictedMeanDiff$compute_estimate()
- InferenceSurvivalRestrictedMeanDiff$compute_estimate_with_bootstrap_weights()
- InferenceSurvivalRestrictedMeanDiff$compute_asymp_confidence_interval()
- InferenceSurvivalRestrictedMeanDiff$compute_asymp_two_sided_pval()
- InferenceSurvivalRestrictedMeanDiff$compute_rand_confidence_interval()
- InferenceSurvivalRestrictedMeanDiff$clone()

InferenceSurvivalRestrictedMeanDiff$new(): Uses the shared randomization two-sided p-value contract; see InferenceRand.

Initialize restricted-mean-survival-time difference inference and prepare treatment-group survival summaries used by InferenceSurvivalRestrictedMeanDiff.

Usage:

```
InferenceSurvivalRestrictedMeanDiff$new(
  des_obj,
  model_formula = NULL,
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

des_obj The design object.

model_formula Optional formula for covariate adjustment. If NULL (default), the formula from the design object is used and its pre-computed design matrix is reused. If a formula is provided, a new design matrix is constructed from the design's imputed covariates.

verbose If TRUE, print additional information.

smart_cold_start_default Whether to use smart cold start values by default.

InferenceSurvivalRestrictedMeanDiff$compute_estimate(): Computes the class-specific mean or survival contrast; see InferenceMLEorKMSummaryTable.

Usage:

```
InferenceSurvivalRestrictedMeanDiff$compute_estimate(estimate_only = FALSE)
```

Arguments:

estimate_only If TRUE, skip variance component calculations.

Returns: The setting-appropriate (see description) numeric estimate of the treatment effect

InferenceSurvivalRestrictedMeanDiff$compute_estimate_with_bootstrap_weights(): Recomputes the class-specific treatment estimate for a bootstrap sample; see InferenceNonParamBootstrap.

Usage:

```
InferenceSurvivalRestrictedMeanDiff$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Row weights for the bootstrap sample.

`estimate_only` If TRUE, skip variance calculations.

`InferenceSurvivalRestrictedMeanDiff$compute_asymp_confidence_interval()`: Computes a $1-\alpha$ level Wald confidence interval for the RMST-difference treatment effect $\hat{\mu}_T(\tau) - \hat{\mu}_C(\tau)$, using its Greenwood-based standard error (see class documentation). Falls back to a nonparametric bootstrap interval if that standard error is unavailable or non-finite.

Usage:

```
InferenceSurvivalRestrictedMeanDiff$compute_asymp_confidence_interval(
  alpha = 0.05
)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

Returns: A (1 - alpha)-sized frequentist confidence interval for the treatment effect

`InferenceSurvivalRestrictedMeanDiff$compute_asymp_two_sided_pval()`: Computes a two-sided Wald p-value testing $H_0 : \mu_T(\tau) - \mu_C(\tau) = 0$ (only `delta = 0` is currently supported; a non-zero null raises an error), using the RMST-difference estimate and its Greenwood-based standard error — see class documentation. Falls back to a nonparametric bootstrap p-value if that standard error is unavailable.

Usage:

```
InferenceSurvivalRestrictedMeanDiff$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` The null difference to test against. For any treatment effect at all this is set to zero (the default).

Returns: The approximate frequentist p-value

`InferenceSurvivalRestrictedMeanDiff$compute_rand_confidence_interval()`: Uses the shared randomization confidence-interval contract; see `InferenceRandCI`.

Usage:

```
InferenceSurvivalRestrictedMeanDiff$compute_rand_confidence_interval(
  alpha = 0.05,
  r = 501,
  pval_epsilon = 0.005,
  show_progress = TRUE,
  ci_search_control = NULL
)
```

Arguments:

`alpha` The confidence level in the computed confidence interval is 1 - `alpha`. The default is 0.05.

`r` The number of randomization vectors. The default is 501.

`pval_epsilon` The bisection algorithm tolerance. The default is 0.005.

`show_progress` Show a text progress indicator.

`ci_search_control` Unused.

Returns: A 1 - alpha sized frequentist confidence interval

`InferenceSurvivalRestrictedMeanDiff$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalRestrictedMeanDiff$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Royston, P., and Parmar, M. K. B. (2013). "Restricted mean survival time: an alternative to the hazard ratio for the design and analysis of randomized trials with a time-to-event outcome." *BMC Medical Research Methodology*, 13, 152, doi:10.1186/1471228813152, for RMST as a treatment-effect summary. Kaplan, E. L., and Meier, P. (1958). "Nonparametric Estimation from Incomplete Observations." *Journal of the American Statistical Association*, 53(282), 457-481, doi:10.2307/2281868, for the underlying survival curve estimator each arm's RMST is integrated from.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'survival')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalRestrictedMeanDiff$new(seq_des)
inf$compute_estimate()
```

InferenceSurvivalStratCoxPHRegr

Stratified Cox PH Inference for Survival Responses

Description

Fits an **auto-stratified** Cox proportional hazards regression: rather than the plain single-baseline-hazard model of `InferenceSurvivalCoxPHRegr`, this class allows a separate baseline hazard per stratum, $\lambda(t \mid x_i, s_i) = \lambda_{0,s_i}(t)\exp(x_i^\top \beta)$, relaxing the proportional-hazards assumption across strata while keeping it within each. Stratification variables are chosen **automatically** from the recorded low-cardinality (categorical-like) covariates (`compute_survival_strata_ids_cpp`) —

no stratification variables are specified explicitly by the caller. If no suitable stratification covariates are found, the fit falls back to the corresponding standard (unstratified) Cox PH model. Fitting uses `survival::coxph.fit()`/`survival::coxph()` with `strata` passed through when applicable. This is a partial-likelihood class (`likelihood_tier = "partial"`) supporting Wald, score, gradient, and likelihood-ratio tests, plus parametric likelihood-ratio bootstrap calibration. Randomization confidence intervals are not supported (the log-hazard-ratio estimator units are not commensurate with the randomization CI bisection algorithm's log-time-ratio/AFT-effect null search).

Super class

`Inference` -> `InferenceSurvivalStratCoxPHRegr`

Methods

Public methods:

- `InferenceSurvivalStratCoxPHRegr$new()`
- `InferenceSurvivalStratCoxPHRegr$compute_asymp_confidence_interval()`
- `InferenceSurvivalStratCoxPHRegr$compute_asymp_two_sided_pval()`
- `InferenceSurvivalStratCoxPHRegr$compute_estimate_with_bootstrap_weights()`
- `InferenceSurvivalStratCoxPHRegr$compute_rand_confidence_interval()`
- `InferenceSurvivalStratCoxPHRegr$clone()`

`InferenceSurvivalStratCoxPHRegr$new()`: Uses the shared randomization two-sided p-value contract; see `InferenceRand`. Pinned from `InferenceRand` for the same traced reason as `InferenceSurvivalCoxPHRegr` (see that factory call's comment): `InferenceRandCI`'s richer override calls `super$...()`, which resolves against `Inference` once flattened, and its only other behavior is an incidence-only Zhang special case that never applies to survival data.

Initialize stratified Cox proportional-hazards inference and prepare the partial-likelihood fit used by `InferenceSurvivalStratCoxPHRegr`.

Usage:

```
InferenceSurvivalStratCoxPHRegr$new(
  des_obj,
  model_formula = NULL,
  use_rcpp = TRUE,
  optimization_alg = "lbfgs",
  verbose = FALSE,
  smart_cold_start_default = NULL
)
```

Arguments:

`des_obj` A completed `Design` object with a survival response.

`model_formula` Optional formula for covariate adjustment. If `NULL` (default), covariates from the design object are included. Use `~ 1` for univariate.

`use_rcpp` Logical. If `TRUE` (default), enable internal Rcpp score/information helpers for likelihood inference. Cox optimization uses **survival::coxph.fit**.

`optimization_alg` Optimization algorithm: `"newton_raphson"` (default) or `"lbfgs"`.

`verbose` Whether to print progress messages.

`smart_cold_start_default` Whether to use smart cold start values.

`InferenceSurvivalStratCoxPHRegr$compute_asymp_confidence_interval()`: Computes an asymptotic confidence interval using the configured likelihood-backed test.

Usage:

```
InferenceSurvivalStratCoxPHRegr$compute_asymp_confidence_interval(alpha = 0.05)
```

Arguments:

`alpha` Significance level 1 - `alpha`. Default 0.05.

`InferenceSurvivalStratCoxPHRegr$compute_asymp_two_sided_pval()`: Computes an asymptotic two-sided p-value using the configured likelihood-backed test.

Usage:

```
InferenceSurvivalStratCoxPHRegr$compute_asymp_two_sided_pval(delta = 0)
```

Arguments:

`delta` Null treatment effect to test against. Default 0.

`InferenceSurvivalStratCoxPHRegr$compute_estimate_with_bootstrap_weights()`: Recomputes the stratified Cox PH treatment estimate under Bayesian-bootstrap weights.

Usage:

```
InferenceSurvivalStratCoxPHRegr$compute_estimate_with_bootstrap_weights(
  subject_or_block_weights,
  estimate_only = FALSE
)
```

Arguments:

`subject_or_block_weights` Subject-, block-, cluster-, or matched-set bootstrap weights.

`estimate_only` If `TRUE`, compute only the weighted point estimate.

`InferenceSurvivalStratCoxPHRegr$compute_rand_confidence_interval()`: Compute a randomization-based confidence interval for the stratified Cox treatment effect by inverting the class-specific randomization p-value. See `InferenceRandCI`.

Usage:

```
InferenceSurvivalStratCoxPHRegr$compute_rand_confidence_interval(
  alpha = 0.05,
  r = 501,
  pval_epsilon = 0.005,
  show_progress = TRUE,
  ci_search_control = NULL
)
```

Arguments:

`alpha` The significance level (default 0.05).

`r` Number of vectors to draw.

`pval_epsilon` The bisection convergence tolerance.

`show_progress` Whether to show a progress bar.

`ci_search_control` Unused.

`InferenceSurvivalStratCoxPHRegr$clone()`: The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalStratCoxPHRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'survival')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalStratCoxPHRegr$new(seq_des)
inf$compute_estimate()
```

`InferenceSurvivalWeibullRegr`

Weibull AFT Inference for Survival Responses

Description

Fits a Weibull Accelerated Failure Time (AFT) model for survival responses: $\log T_i = \beta_0 + \beta_T W_i + X_i^\top \gamma + \sigma \epsilon_i$, $\epsilon_i \sim$ standard extreme-value (Gumbel-minimum), so that T_i is marginally Weibull-distributed with shape $1/\sigma$ and treatment-dependent scale, by maximum likelihood (`fast_weibull_regression_cpp`). $\hat{\beta}_T$ is a **log-time-ratio** (log acceleration factor): $\exp(\hat{\beta}_T)$ is the estimated multiplicative effect of treatment on survival time (an AFT model, not a proportional-hazards model — the Weibull distribution is the one location where AFT and proportional-hazards parameterizations coincide, since $\exp(-\beta_T/\sigma)$ also equals the treatment hazard ratio). `likelihood_tier = "full"`: likelihood-ratio, score, gradient, and Wald tests are all available when the model converges, plus parametric-likelihood-bootstrap calibration of the likelihood-ratio test. Right-censored and interval-censored observations enter the likelihood via their appropriate survival/density contributions. Validity requires the Weibull shape assumption for the (log-)survival-time distribution and, when interpreted causally, the usual design-based/model-based assumptions.

Computes the randomization distribution of the treatment effect estimate under the sharp null.

Whether `compute_rand_two_sided_pval()` is actually usable on this instance right now – `FALSE` exactly when it would `stop()`: an `incidence`-response instance with no custom randomization statistic and a design not eligible for design-randomization-based incidence inference (see `private$should_use_design_ra` `TRUE` for every other case, including every non-incidence response type. Public, self-contained (only reads already-set instance state, no side effects), so `InferenceSuite` can check this before attempting the sentinel instead of relying on the `stop()` being silently swallowed into a `pval = NA` "ok" row – the single source of truth for both this check and `compute_rand_two_sided_pval()`'s own guard, so the two can never drift apart.

Value

When debug = FALSE (default), a numeric vector of length r. When debug = TRUE, a list with: values, errors (list of character vectors, one per iteration), warnings (list of character vectors, one per iteration), num_errors, num_warnings, prop_iterations_with_errors, prop_iterations_with_warnings, and prop_illegal_values.

A single logical.

Super class

Inference -> InferenceSurvivalWeibullRegr

Methods

Public methods:

- InferenceSurvivalWeibullRegr$approximate_randomization_distribution_beta_hat_T()
- InferenceSurvivalWeibullRegr$supports_rand_pval_for_incidence()
- InferenceSurvivalWeibullRegr$clone()

InferenceSurvivalWeibullRegr$approximate_randomization_distribution_beta_hat_T():

Usage:

```
InferenceSurvivalWeibullRegr$approximate_randomization_distribution_beta_hat_T(
  r = 501,
  delta = 0,
  transform_responses = "none",
  show_progress = TRUE,
  permutations = NULL,
  debug = FALSE,
  zero_one_logit_clamp = .Machine$double.eps
)
```

Arguments:

r Number of randomization vectors. Default 501.

delta The null difference. Default 0.

transform_responses Type of transformation. Default "none".

show_progress Show progress bar. Default TRUE.

permutations Pre-computed permutations. Default NULL.

debug If TRUE, return a list with the distribution values and per-iteration diagnostics including error messages, warning messages, counts of each, and summary proportions for iterations with errors, warnings, and illegal (non-finite) values. Runs serially. Default FALSE.

zero_one_logit_clamp The clamping amount for exact 0 and 1 values when logging

InferenceSurvivalWeibullRegr$supports_rand_pval_for_incidence():

Usage:

```
InferenceSurvivalWeibullRegr$supports_rand_pval_for_incidence()
```

InferenceSurvivalWeibullRegr$clone(): The objects of this class are cloneable with this method.

Usage:

```
InferenceSurvivalWeibullRegr$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

Kalbfleisch, J. D., and Prentice, R. L. (2002). *The Statistical Analysis of Failure Time Data* (2nd ed.). Wiley, for the Weibull AFT model and its equivalence to a proportional-hazards model. For the randomization confidence interval (`compute_rand_confidence_interval()`), which inverts an AFT sharp null by rescaling the recorded times of treated units — event and censoring times alike, censoring indicators unchanged — the residual construction and its validity under independent censoring are from Tsiatis, A. A. (1990). Estimating regression parameters using linear rank tests for censored data. *The Annals of Statistics*, 18(1), 354-372, doi:10.1214/aos/1176347504; Wei, L. J., Ying, Z., and Lin, D. Y. (1990). Linear regression analysis of censored survival data based on rank tests. *Biometrika*, 77(4), 845-851, doi:10.1093/biomet/77.4.845; and Jin, Z., Lin, D. Y., Wei, L. J., and Ying, Z. (2003). Rank-based inference for the accelerated failure time model. *Biometrika*, 90(2), 341-353, doi:10.1093/biomet/90.2.341.

See Also

Comparable Python API: lifelines WeibullAFTFitter. See also: Proportional hazards model (Wikipedia, for the AFT/PH equivalence note).

Examples

```
seq_des = DesignSeqOneByOneBernoulli$new(n = 10, response_type = 'survival')
for (i in 1:10) {
  seq_des$add_one_subject_to_experiment_and_assign(data.frame(x1 = rnorm(1)))
}
seq_des$add_all_subject_responses(runif(10))
inf = InferenceSurvivalWeibullRegr$new(seq_des)
inf$compute_estimate()


inf$set_seed(1)
inf$compute_lik_ratio_bootstrap_two_sided_pval(delta = 0, B = 9, show_progress = FALSE)
```

`inv_logit` *Inverse Logit (Logistic) Function*

Description

Computes the inverse logit (standard logistic sigmoid) function, $\mathrm{logit}^{-1}(x) = 1/(1 + e^{-x})$, the canonical mean function for binomial/logistic-family models throughout this package (mapping a linear predictor on the log-odds scale back to a probability). The result is clamped to $[$`zero_one_logit_clamp`, $1-$

zero_one_logit_clamp] before being returned, so an extreme x (e.g. from a poorly identified or diverging fit) cannot produce an exact 0 or 1 probability that would later cause a -Inf/ NaN when log-transformed downstream (e.g. in a log-likelihood).

Usage

```
inv_logit(x, zero_one_logit_clamp = .Machine$double.eps)
```

Arguments

x	Any real number (or vector), typically a fitted linear predictor $\eta = x_i^\top \beta$ on the log-odds scale.
zero_one_logit_clamp	The clamping distance from the 0/1 boundaries applied to the result. Default .Machine$double.eps.

Value

The inverse-logit-transformed value(s), in (0, 1), the same length as x.

See Also

logit for the forward transform.

Examples

```
inv_logit(0)
```

---

logit *Logit (Log-Odds) Transform*

Description

Computes the logit (log-odds) function $\mathrm{logit}(p) = \log(p/(1-p))$, the canonical link function for binomial/logistic-family models throughout this package. p is first clamped to [zero_one_logit_clamp, 1− zero_one_logit_clamp] before transforming, so exact 0 or 1 inputs (which would otherwise map to $-\infty/\infty$) instead return a large but finite value; this is what lets proportion/fractional responses with mass exactly at the boundary be used as pseudo-continuous inputs to logit-scale machinery elsewhere in the package (e.g. fast_ols_cpp-backed logit-transform-then-OLS shortcuts) without producing non-finite values.

Usage

```
logit(p, zero_one_logit_clamp = .Machine$double.eps)
```

Arguments

p	The value(s) to transform, nominally in `(0, 1)` (values outside that range, or exactly 0/1, are clamped rather than rejected).
zero_one_logit_clamp	The clamping distance from the 0/1 boundaries applied to p before transforming. Default `.Machine$double.eps`.

Value

The logit-transformed value(s) as a real number (or vector), the same length as p.

See Also

`inv_logit` for the inverse transform.

Examples

```
logit(0.25)
```

---

mn_pvalue_cpp *Export of C++ function mn_pvalue_cpp*

Description

Evaluates the two-sided asymptotic p-value for the Miettinen-Nurminen restricted-maximum-likelihood score test of $H_0 : p_T - p_C = \delta$ in two independent binomial samples (see `InferenceIncidMiettinenNurminenRiskDiff` for the class that consumes this function). Internally: `mn_z_statistic_cpp` computes the score z statistic using the constrained MLEs $\tilde{p}_C, \tilde{p}_T = \tilde{p}_C + \delta$ (`mn_constrained_mle_pc_cpp`, found by bisecting the constrained score equation to zero) in place of the unconstrained sample proportions in the variance formula, with a small-sample correction factor $(n_T + n_C)/(n_T + n_C - 1)$ applied to the naive binomial variance; this function then returns $2\,\Phi(-|z|)$, the two-sided normal-tail p-value. Returns `NA` if either arm is empty, δ is outside $(-1, 1)$, or the resulting z is not finite (e.g. the constrained variance estimate is non-positive).

Usage

```
mn_pvalue_cpp(x_t, n_t, x_c, n_c, delta, p_t_obs, p_c_obs)
```

Arguments

x_t	Number of events in treatment.
n_t	Number of subjects in treatment.
x_c	Number of events in control.
n_c	Number of subjects in control.
delta	Null risk difference.
p_t_obs	Observed treatment-arm risk.
p_c_obs	Observed control-arm risk.

Value

The two-sided p-value.

`print.EDIInferenceSuiteResults`

Prints the results table from an `InferenceSuite` `run_all_inference()` *call – the same table* `screen = TRUE` *prints during the call itself, so a user who assigned the return value and later types its name (or calls* `print()` *on it) sees a readable table rather than a raw nested list dump. The table itself is rendered by* `run_all_inference_format_pretty_table()`*: rows sorted by* `estimand`*, with a double rule under the header and a single rule between* `estimand` *groups and at the bottom, class names and* `estimand` *values shortened for display (never the underlying* `results_table` *values), and a* `cov_model` *letter-key legend appended when applicable – see that function's own documentation for the exact column-by-column rendering rules.*

Description

Prints the results table from an `InferenceSuite` `run_all_inference()` call – the same table `screen = TRUE` prints during the call itself, so a user who assigned the return value and later types its name (or calls `print()` on it) sees a readable table rather than a raw nested list dump. The table itself is rendered by `run_all_inference_format_pretty_table()`: rows sorted by `estimand`, with a double rule under the header and a single rule between `estimand` groups and at the bottom, class names and `estimand` values shortened for display (never the underlying `results_table` values), and a `cov_model` letter-key legend appended when applicable – see that function's own documentation for the exact column-by-column rendering rules.

Usage

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

Arguments

`x`	An `EDIInferenceSuiteResults` object, as returned by `InferenceSuite$run_all_inference()`.
`...`	Ignored; present for S3 consistency with the generic.

Value

`x`, invisibly.

robust_negbinreg — *Robust Negative Binomial Regression with Backward Column-Dropping Fallback*

Description

Fits a negative-binomial GLM via `glm.nb` (log link, joint ML estimation of the regression coefficients and the dispersion parameter θ), falling back to a smaller model when the fit throws an error (typically non-convergence of θ, or a singular design). On each failure, the **last** column of `data_obj` is dropped and the fit is retried against the same `form_obj` (which must resolve to `y ~ .` or similar so that its right-hand side tracks the shrinking column set); this repeats until a fit succeeds or every predictor column has been removed, at which point `NA` is returned. Because columns are dropped strictly from the right, callers should order `data_obj`'s columns from most to least important *a priori*, or accept that this is a best-effort robustness measure rather than a principled model-selection procedure.

Usage

```
robust_negbinreg(form_obj, data_obj)
```

Arguments

`form_obj`	The model formula, typically `y ~ .` so its right-hand side automatically tracks `data_obj`'s shrinking column set across retries.
`data_obj`	The data frame to run negative-binomial regression on; its **last** column is dropped on each retry, in order, until a fit converges or no columns remain.

Value

The fitted `glm.nb` model object, or `NA` if no column subset (down to and including the response alone) produced a successful fit.

Examples

```
dat = data.frame(y = rpois(10, 2), x1 = rnorm(10), x2 = rnorm(10))
robust_negbinreg(y ~ ., dat)
```

robust_survreg — *Robust Parametric Survival Regression from Response/Censoring Vectors*

Description

Convenience wrapper around `robust_survreg_with_surv_object` that builds the `Surv` object from separate response and censoring vectors first. See that function for the full description of the warm-start-then-random-restart fitting strategy used to make `survreg` converge reliably even from poor or near-singular starting points.

Usage

```
robust_survreg(
  y,
  dead,
  cov_matrix_or_vector,
  dist = "weibull",
  num_max_iter = 50
)
```

Arguments

y	The (possibly right-censored) response vector (event/censoring time).
dead	The event indicator (1 if the event was observed/uncensored, 0 if right-censored at y).
cov_matrix_or_vector	The design matrix (or a single covariate vector) of predictors, **excluding** the intercept (one is added by the internal ~ . formula).
dist	The parametric AFT distribution family passed to survreg (default "weibull"); see that function's dist argument for the full list of supported families.
num_max_iter	Maximum number of random-restart attempts if the direct fit fails or does not converge (default 50); see robust_survreg_with_surv_object.

Value

The fitted survreg model object, or NULL if no attempt converged to a fit with no NA coefficients within num_max_iter tries.

Examples

```
X = matrix(rnorm(500), 100, 5)
y = runif(100)
dead = rbinom(100, 1, 0.5)
robust_survreg(y, dead, X)
```

---

robust_survreg_with_surv_object

Robust Parametric Survival Regression (AFT) with Warm-Start and Random-Restart Fallback

---

Description

Fits a parametric accelerated-failure-time (AFT) survival regression via survreg on surv_object ~ . over the columns of cov_matrix_or_vector, with two layers of robustness against survreg's well-known sensitivity to starting values and near-collinear design matrices:

1. **Preprocessing**: near-collinear columns of the design matrix are dropped first via drop_highly_correlated_cols then drop_linearly_dependent_cols, before any fitting is attempted.

2. **Warm start (Weibull only)**: when `dist = "weibull"`, a fast closed-form-gradient Weibull fit (`fast_weibull_regression`) is attempted first; if it succeeds and returns a finite log-likelihood, its coefficients and $\log(\hat{\sigma})$ are passed to `survreg` as the `init` vector, which typically converges the true MLE in a single `survreg` call. If this warm-started fit is unavailable, fails, or produces `NA` coefficients, fitting falls through to the general random-restart loop below (for all other `dist` values, this warm start is skipped entirely).
3. **Random-restart loop**: starting from an all-zero `init` vector, `survreg` is called repeatedly (perturbing `init` by an independent standard-normal jitter, `init + rnorm(length(init))`, after every failed attempt) until a fit with no `NA` coefficients is obtained or `num_max_iter` attempts are exhausted, at which point `NULL` is returned.

`survreg.control(maxiter = 100, rel.tolerance = 1e-9, outer.max = 10)` is used throughout (tighter than `survreg`'s own defaults) to reduce the chance of a spuriously "converged" fit at a poor optimum.

Usage

```
robust_survreg_with_surv_object(
  surv_object,
  cov_matrix_or_vector,
  dist = "weibull",
  num_max_iter = 50
)
```

Arguments

`surv_object` The survival object (built from the response vector and censoring vector via `Surv`).

`cov_matrix_or_vector` The design matrix (or a single covariate vector) of predictors, **excluding** the intercept (one is added by the internal `~ .` formula).

`dist` The parametric AFT distribution family passed to `survreg` (default `"weibull"`); only `"weibull"` triggers the closed-form warm start.

`num_max_iter` Maximum number of random-restart attempts if the (possibly warm-started) direct fit fails or does not converge (default 50).

Value

The fitted `survreg` model object, or `NULL` if no attempt converged to a fit with no `NA` coefficients within `num_max_iter` tries.

Examples

```
X = matrix(rnorm(500), 100, 5)
y = runif(100)
dead = rbinom(100, 1, 0.5)
surv = survival::Surv(y, dead)
robust_survreg_with_surv_object(surv, X)
```

sample_mode *Sample Mode*

Description

Thin R wrapper around `sample_mode_cpp()`, which returns the most frequently occurring value in `data`. Integer, logical, double, character, and factor vectors are all supported (dispatched internally on `TYPEOF(data)`; factors preserve their `class/levels` attributes on the returned value). `NA` (and, for doubles, `NaN` as a category distinct from `NA`) is counted like any other value and can itself be returned as "the mode" if it is the most frequent entry. Ties are broken by **first occurrence**: among values tied for the highest count, the one that appears earliest in `data` is returned — this is a positional, not a numeric/lexicographic, tie-break rule.

Usage

```
sample_mode(data)
```

Arguments

`data`	A vector (integer, logical, double, character, or factor) to compute the mode of.

Value

A length-1 vector (same type as `data`) holding the most frequently occurring value, with ties broken in favor of whichever tied value occurs earliest in `data`.

Examples

```
sample_mode(c(1, 2, 2, 3))
```

set_cold_start_dispatch_policy *Update the cold-start dispatch policy*

Description

Overrides, queries, or resets the runtime policy consulted by `edi_cold_start_dispatch_policy()` for whether an inference class's C++ fitting backend defaults to an OLS-based `smart_cold_start` or a plain zero-vector start; see `get_cold_start_dispatch_policy` for the built-in default table and the rationale behind it.

Usage

```
set_cold_start_dispatch_policy(policy = NULL, reset = FALSE)
```

Arguments

policy	Either NULL (no change) or a named list of policy overrides merged into the current configuration (see Details).
reset	If TRUE, discard all overrides and restore the built-in default policy.

Details

Call with no arguments (policy = NULL, reset = FALSE) to retrieve the current configuration without changing it. Pass a named list to policy to merge new/overriding entries into the current configuration via modifyList (so default and/or inference_class_overrides can each be supplied independently, and entries not mentioned are left untouched — this cannot *remove* an existing override pattern, only add or replace one). Pass reset = TRUE to discard any accumulated overrides and restore the package's built-in default policy exactly as returned by get_cold_start_dispatch_policy.

Value

Invisible NULL when policy is supplied (a mutation), or invisibly the current policy configuration list when called for its side-effect-free query/reset value.

See Also

get_cold_start_dispatch_policy for the policy schema and built-in defaults; set_warm_start_dispatch_policy and set_optimization_dispatch_policy for the analogous setters governing warm-starting and optimizer-algorithm choice; tune_EDI_for_this_machine, which calls this setter with machine-measured overrides rather than hand-picked ones.

Examples

```
set_cold_start_dispatch_policy(reset = TRUE)
```

---

set_num_cores *Set the number of cores for parallelization*

Description

This function initializes a persistent parallel cluster (either a fork cluster on Unix-like systems or a mirai cluster on others) to be used by all Design and Inference objects. This avoids the overhead of creating clusters repeatedly.

Usage

```
set_num_cores(num_cores, force_mirai = FALSE)
```

Arguments

num_cores	Integer number of worker processes to make available.
force_mirai	If TRUE, forces the use of the mirai package even on systems where forking is available.

Details

set_num_cores() sets a global upper bound for parallel work. It does not guarantee that every inference routine will use all requested workers. EDI's inference dispatcher applies a blocklist-first heuristic informed by package benchmarks: workloads that have shown consistent multicore slowdowns are forced to run serially, while the remaining workloads are allowed to use their method-specific warmup heuristics and native thread caps.

The default forced-serial blocklist covers incidence randomization confidence intervals, bootstrap for non-regression KK Wilcoxon inference, bootstrap for non-KK survival procedures, and bootstrap for incidence procedures. Do not expect a universal "more cores is faster" rule.

If you want to change the default policy, use set_parallel_dispatch_policy().

Value

Invisible NULL.

Examples

```
set_num_cores(2)
unset_num_cores()
```

---

set_optimization_dispatch_policy

Update the optimization dispatch policy

Description

Overrides, queries, or resets the runtime policy consulted by edi_optimization_dispatch_policy() for which optimization algorithm ("newton_raphson", "lbfgs", or "irls") an inference class's C++ model-fitting backend uses by default; see get_optimization_dispatch_policy for the built-in default table and the empirical rationale behind its per-family choices.

Usage

```
set_optimization_dispatch_policy(policy = NULL, reset = FALSE)
```

Arguments

policy	Either NULL (no change) or a named list of policy overrides merged into the current configuration (see Details).
reset	If TRUE, discard all overrides and restore the built-in default policy.

Details

Call with no arguments (`policy = NULL`, `reset = FALSE`) to retrieve the current configuration without changing it. Pass a named list to `policy` to merge new/overriding entries into the current configuration via `modifyList` (`default_alg` and/or `inference_class_overrides` can each be supplied independently). Pass `reset = TRUE` to discard any accumulated overrides and restore the package's built-in default policy exactly as returned by `get_optimization_dispatch_policy`. Note that this only changes the *default* algorithm consulted when a fitting call does not itself specify `optimization_alg` explicitly; an explicit per-call argument still takes precedence.

Value

Invisible `NULL` when `policy` is supplied (a mutation), or invisibly the current policy configuration list when called for its side-effect-free query/reset value.

See Also

`get_optimization_dispatch_policy` for the policy schema and built-in defaults; `set_cold_start_dispatch_policy` and `set_warm_start_dispatch_policy` for the analogous setters governing cold-start and warm-start behavior; `tune_EDI_for_this_machine`, which calls this setter with machine-measured, convergence-checked overrides rather than hand-picked ones.

Examples

```
set_optimization_dispatch_policy(reset = TRUE)
```

`set_parallel_dispatch_policy`

Update the parallel dispatch policy

Description

EDI uses an empirical, blocklist-first dispatch policy to decide when an inference routine's `"bootstrap"` or `"rand_ci"` resampling should be forced serial even if multiple cores are available (see `get_parallel_dispatch_policy` for the built-in table and the PCRE pattern/response-type matching rules it encodes). This function lets the user update that policy at runtime without editing package internals.

Usage

```
set_parallel_dispatch_policy(policy = NULL, reset = FALSE)
```

Arguments

`policy`	Either `NULL` (no change), a named list of policy-section overrides, or a custom dispatch function (see Details).
`reset`	If `TRUE`, restore the built-in default policy and remove any custom function override.

Details

The policy can be updated in two mutually exclusive ways:

- Pass a named list to `policy` to merge with the current policy configuration via `modifyList`, one section at a time. Supported top-level keys are `bootstrap` and `rand_ci`, and each key's value is itself a list that may contain `serial_inference_class_patterns` (character vector of PCRE regular expressions) and/or `serial_response_types` (character vector of exact response-type strings); an unrecognized top-level key raises an error rather than being silently ignored.
- Pass a custom function with signature `function(inference_class, response_type, operation)` to `policy` to **replace** the entire pattern-table dispatch logic with your own rule; it must return a list with at least `force_serial` (logical) and `reason` (character). Supplying a function clears any list-based overrides accumulated so far and is stored separately from the pattern-table configuration.

Use `reset = TRUE` to discard both the list-based overrides and any custom function, restoring the package's built-in default policy exactly as returned by `get_parallel_dispatch_policy`. Do not expect a universal "more cores is faster" rule — this policy exists because several resampling workloads have shown consistent multicore *slowdowns* in package benchmarks.

Value

Invisible `NULL` when `policy` is supplied (a mutation), or invisibly the current policy configuration list when called for its side-effect-free query/reset value.

See Also

`get_parallel_dispatch_policy` for the policy schema and built-in defaults (including why this table is a safety blocklist, not a performance one); `set_num_cores` for setting the actual worker-count upper bound this policy operates within; `tune_EDI_for_this_machine` for the separate, machine-dependent question of when parallel execution is *worthwhile* (never applied here — this policy is only ever overridden explicitly, by you).

Examples

```
set_parallel_dispatch_policy(reset = TRUE)
```

---

set_warm_start_dispatch_policy

Update the warm-start dispatch policy

Description

Overrides, queries, or resets the runtime policy consulted by `edi_warm_start_dispatch_policy()` for whether an inference class reuses a previous fit's parameters/curvature to seed the next fit during resampling; see `get_warm_start_dispatch_policy` for the built-in default table and its `jackknife/non_param_boot/ bayesian_boot/param_boot/rand` operation schema (each with an `inference_class_overrides` layer and an `n_conditioned_overrides` layer).

Usage

```
set_warm_start_dispatch_policy(policy = NULL, reset = FALSE)
```

Arguments

policy	Either NULL (no change) or a named list of per-operation policy overrides merged into the current configuration (see Details).
reset	If TRUE, discard all overrides and restore the built-in default policy.

Details

Call with no arguments (policy = NULL, reset = FALSE) to retrieve the current configuration without changing it. Pass a named list to policy to merge new/overriding entries into the current configuration via modifyList (per-operation sub-lists, e.g. list(rand = list(inference_class_overrides = ...)) or list(rand = list(n_conditioned_overrides = ...)), are merged rather than replaced wholesale — note n_conditioned_overrides is a plain list of rules, so overriding it replaces the whole list for that operation, not a per-rule merge). Pass reset = TRUE to discard any accumulated overrides and restore the package's built-in default policy exactly as returned by get_warm_start_dispatch_policy. **This function controls the full dispatch policy**, including the sample-size-conditioned n_conditioned_overrides layer.

Value

Invisible NULL when policy is supplied (a mutation), or invisibly the current policy configuration list when called for its side-effect-free query/reset value.

See Also

get_warm_start_dispatch_policy for the policy schema and built-in defaults; set_cold_start_dispatch_policy for the analogous, simpler single-layer setter governing the initial cold-start heuristic; tune_EDI_for_this_machine, which calls this setter with machine-measured overrides rather than hand-picked ones.

Examples

```
set_warm_start_dispatch_policy(reset = TRUE)
```

SimulationFramework — *Simulation Framework for Experimental Designs and Inference Methods*

Description

An R6 class for benchmarking experimental designs and inference methods by Monte Carlo simulation. Each replication generates synthetic covariates and responses, runs every requested (design, inference) pair, and records point estimates, confidence intervals, and p-values. Raw and aggregated results are available through SimulationFrameworkReport.

Details

Covariates are drawn independently from $\mathrm{Uniform}(0, 1)$.

- `cond_exp_func_model = "linear"`: the base continuous signal is $y = X\beta$ where β is evenly spaced from 1 to -1.
- `cond_exp_func_model = "nonlinear"`: the Friedman (1991) function $10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5$; requires $p \geq 5$.

The continuous base signal is transformed to the scale appropriate for `response_type`. Treatment effects are applied per-subject: additive on the linear/logit/ordinal scale, log-multiplicative for count and survival.

For each `(design, inference)` pair the framework runs whichever of the following are supported by the inference class:

- **asymptotic** (`InferenceAsymp` subclasses): Wald CI and p-value.
- **bootstrap** (`InferenceNonParamBootstrap` subclasses): percentile CI and p-value.
- **randomisation** (`InferenceRand` subclasses): p-value; additionally a test-inversion CI for `continuous`, `proportion`, and `count` response types (`InferenceRandCI` subclasses).

Incompatible `(design, inference)` pairs (e.g.\ a KK-specific inference class with a non-KK design) are silently skipped via `tryCatch`.

Reported summary metrics include:

- **MSE**: $\overline{(\hat{\beta}_T - \beta_T)^2}$ over reps with a finite point estimate.
- **coverage**: proportion of reps where β_T lies inside the CI (`NA` when no CI is available for that inference type).
- **power**: proportion of p-values $< \alpha$; equals the empirical type-I error rate when `betaT = 0`.

Methods

Public methods:

- `SimulationFramework$new()`
- `SimulationFramework$run()`
- `SimulationFramework$get_all_intermediate_data()`
- `SimulationFramework$clear_all_intermediate_data_and_gc()`
- `SimulationFramework$clone()`

`SimulationFramework$new()`: Create a new `SimulationFramework` object that stores simulation design settings, response-generation settings, inference methods, and replication controls.

Usage:

```
SimulationFramework$new(
  response_type,
  design_classes_and_params = NULL,
  inference_classes_and_params = NULL,
  n = 100L,
  p = 5L,
```

```
  cond_exp_func_model = "linear",
  Nrep_W = 100L,
  Nrep_Y_w = 1L,
  betaT = 1,
  alpha = 0.05,
  B_boot = 201L,
  r_rand = 201L,
  pval_epsilon = 0.02,
  sd_noise = 1,
  n_ordinal_levels = 4L,
  proportion_epsilon = 1e-06,
  phi_proportion = 100,
  k_survival = 2,
  incidence_clamp = 1e-09,
  proportion_clamp = 1e-09,
  count_clamp = 1e-09,
  survival_clamp = 1e-09,
  survival_min_time = 0.1,
  count_min_rate = 0L,
  count_shift = 0,
  norm_sq_beta_vec = 1,
  X_mat = NULL,
  num_cores = 1L,
  seed = NULL,
  cov_draw_method = stats::rnorm,
  cov_draw_method_args = list(mean = 0, sd = 1),
  random_X_draws = TRUE,
  prob_censoring = 0.25,
  custom_replication_data_generator = NULL,
  custom_apply_treatment_and_noise = NULL,
  make_estimand_fn = NULL,
  dgp_params = list(),
  custom_dgp = NULL,
  verbose = TRUE,
  keep_all_intermediate_data = FALSE,
  turn_off_asserts_for_speed = TRUE,
  inference_types_and_params = NULL,
  results_filename = "simulation_framework_results.csv.bz2",
  continue_from_last_result_row = TRUE,
  reuse_cache = TRUE,
  stop_on_error = TRUE,
  save_to_disk_every_n_rep = 25L,
  save_model_control_fits = TRUE
)
```

Arguments:

`response_type` **(required)** Character scalar or vector. The type of outcome variable. One of `"continuous"`, `"incidence"`, `"proportion"`, `"count"`, `"survival"`, `"ordinal"`.

`design_classes_and_params` `NULL` (default) or a list describing design classes and optional

constructor parameters. Unnamed R6 class generators use default parameters, for example `list(DesignSeqOneByOneKK21, DesignFixedBernoulli)`. Named entries use the entry name as the design class and the value as the parameter list, for example `list(DesignSeqOneByOneUrn = list(alpha = 2, beta = 2))`. Duplicate named entries are allowed for repeated designs with different parameters. Each generator must be constructable with only `response_type` and `n` plus any extra params supplied in this list. `NULL` uses the package's standard design set. Designs requiring `strata_cols`, `cluster_col`, or `factors` have sensible defaults auto-injected (first covariate column; second for `cluster_col`; `list(treatment=2)` for `factors`) when not supplied in the parameter list. Example:

```
design_classes_and_params = list(
  DesignSeqOneByOneKK21 = list(lambda = 0.5, t_0_pct = 0.1),
  DesignSeqOneByOneUrn  = list(alpha  = 2,   beta    = 2),
  DesignFixedBernoulli  # default params
)
```

Commonly useful design constructor parameters:

`lambda` Matching-weight decay for `KK14` / `KK21` / `KK21stepwise`.

`t_0_pct` Burn-in fraction for `KK14` / `KK21` / `KK21stepwise`.

`morrison` Logical; Morrison correction for `KK14`.

`alpha`, `beta` Shape parameters for `DesignSeqOneByOneUrn`.

`preferred_num_bins_for_continuous_covariate` Bin count for `DesignFixedBlocking` and `DesignFixedBlockedCluster`.

`B_target` Target number of blocks for `DesignFixedBlocking`.

`inference_classes_and_params` `NULL` (default) or a list describing inference classes and optional constructor parameters. Unnamed R6 class generators use default parameters, for example `list(InferenceContinOLS, InferenceContinKKOLSIVWC)`. Named entries use the entry name as the inference class and the value as the constructor parameter list, for example `list(InferenceContinOLS = list(max_resample_attempts = 25L))`. Duplicate named entries are allowed for repeated inference classes with different parameters. Supplied parameters must be accepted by the inference class constructor. `NULL` selects a curated set for the given `response_type`: several universal classes that work with any design, plus representative KK-specific classes (silently skipped for non-KK designs at runtime).

`n` Integer scalar or vector. Sample size per simulation replication. Default `100`.

`p` Integer scalar or vector. Number of covariates. Must be ≥ 5 when `cond_exp_func_model = "nonlinear"`. Default `5`.

`cond_exp_func_model` Character scalar or vector. How the latent continuous signal is constructed before transformation to the `response_type` scale.

`"linear"` Linear combination $X\beta$ with coefficients evenly spaced from 1 to -1.

`"nonlinear"` Friedman (1991) function $10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5$; requires $p \geq 5$.

Default `"linear"`.

`Nrep_W` Positive integer. Number of treatment-assignment draws (w-reps). Each w-rep generates a fresh covariate matrix `X` and a new treatment assignment vector `w`. Default `100L`.

`Nrep_Y_w` Positive integer. Number of draws of the response per draw of `w`, the allocation vector. For each w-rep, `Nrep_Y_w` independent response vectors `y` are drawn from the same `(X, w)`. The effective total number of replications recorded is `Nrep_W * Nrep_Y_w`. Default `1` (standard behaviour: one outcome draw per allocation-vector draw).

`betaT` Numeric scalar or vector. True treatment effect added to treated subjects' outcomes. The scale is response-type specific: additive for `continuous`, `proportion`, and `ordinal`; on the logit scale for `incidence`; log-multiplicative for `count` and `survival`. Default `1`. Set `betaT = 0` to check type-I error.

`alpha` Numeric in $(0, 1)$. Significance level used for all confidence intervals and for computing power ($p < \alpha$). Default `0.05`.

`B_boot` Positive integer. Bootstrap resamples per CI / p-value call. Default `201`.

`r_rand` Positive integer. Randomisation draws per rand p-value call, and per bisection step of the rand CI. Default `201`.

`pval_epsilon` Numeric. Bisection convergence tolerance for randomisation-based CIs (`compute_rand_confidence_in` Default `0.02`.

`sd_noise` Numeric > 0. Standard deviation of independent Gaussian noise added to each subject's outcome. Default `1`.

`n_ordinal_levels` Positive integer. Number of ordinal categories when `response_type = "ordinal"`. Default `4L`.

`proportion_epsilon` Numeric scalar. Small value added to proportion base responses to avoid 0 and 1. Default `1e-6`.

`phi_proportion` Positive numeric scalar. Precision parameter for beta-distributed observed proportion outcomes. The beta mean is `y_linear_model[i] + betaT * w[i]`. Default `100`.

`k_survival` Positive numeric scalar. Scale parameter passed to the Weibull draw for observed survival outcomes. Default `2`.

`incidence_clamp` Numeric scalar in $(0, 0.5)$. Clamp applied to the Bernoulli probability for observed incidence outcomes. Default `1e-9`.

`proportion_clamp` Numeric scalar in $(0, 0.5)$. Clamp applied to the beta mean for observed proportion outcomes. Default `1e-9`.

`count_clamp` Positive numeric scalar. Minimum Poisson mean for observed count outcomes. Default `1e-9`.

`survival_clamp` Positive numeric scalar. Minimum Weibull shape for observed survival outcomes. Default `1e-9`.

`survival_min_time` Numeric scalar. Minimum survival time and shift for base responses. Default `0.1`.

`count_min_rate` Integer scalar. Minimum baseline rate for count responses. Default `0L`.

`count_shift` Numeric scalar. Constant added to counts after zero-centering for base responses. Default `0`.

`norm_sq_beta_vec` Positive numeric scalar. The desired squared Euclidean norm of the latent linear coefficient vector β. The generated vector is scaled to match this norm. Default `1`.

`X_mat` Numeric matrix of dimensions `n x p`, or `NULL` (default). If provided, these fixed covariates are used for every replication. In this case, `cov_draw_method` must be `NULL`.

`num_cores` Positive integer. Number of worker processes for parallel execution of Monte Carlo replications. Note that when `num_cores > 1`, parallelization *within* individual inference routines (e.g. bootstrap, randomization) is automatically disabled to prevent thread oversubscription.

Unix/Linux (recommended): A `makeForkCluster` pool is created once at `run()` start. Workers inherit all pre-generated design and SE caches via copy-on-write with zero serialization overhead. Parallelism operates at the *replication* level: `num_cores` replications run simultaneously, each executing all DGP cells serially. This eliminates the per-batch dispatch

overhead that would arise from cycling through cells within every replication, and keeps all cores fully subscribed regardless of the number of DGP cells. For best performance, run on a Unix/Linux machine and set `num_cores` to the number of physical cores available.
Non-Unix (Windows/macOS): mirai daemons are used when available. Every active (replication, DGP-cell) pair is one work unit and `num_cores` units are kept in flight continuously, so all cores stay subscribed even when the grid has fewer DGP cells than cores. Cell state is pushed to the daemons once at `run()` start rather than re-serialized per dispatch. If **mirai** is not installed, execution falls back to serial with a warning. Default `1`.

`seed` Integer or `NULL` (default). Random seed for the entire simulation run.

`cov_draw_method` A function used to draw `n * p` i.i.d. covariate values for every replication. The function must accept the total number of values as its first argument, followed by arguments in `cov_draw_method_args`. Default `stats::rnorm`. Must be `NULL` when `X_mat` is supplied.

`cov_draw_method_args` Named list of additional arguments forwarded to `cov_draw_method` beyond the sample-size first argument. Default is `list(mean = 0, sd = 1)`.

`random_X_draws` Logical. If `TRUE` (default), a new set of covariates is drawn for every single replication. If `FALSE`, one set is drawn per `(n, p)` cell and shared across its replications.

`prob_censoring` Numeric in $[0, 1]$. Per-subject independent censoring probability; applied only when `response_type = "survival"`. Default `0.25`.

`custom_replication_data_generator` Optional function for custom replication data. When supplied, it is called as `fn(state, rep)` and must return a list containing at least `X` and `y_linear_model`. Any additional fields in the returned list (e.g. latent frailty draws) are passed forward as `rep_data` to `custom_apply_treatment_and_noise` and the function built by `make_estimand_fn`.

`custom_apply_treatment_and_noise` Optional function for custom response generation. Signature: `fn(y_linear_model, w, rep_data, state)`. `w` uses {0, 1} encoding, with 1 for treatment and 0 for control. `rep_data` is the full list returned by `custom_replication_data_generator` (or `NULL` for the standard path). Must return a list with components `y` and `dead`. Three-argument functions `fn(y_linear_model, w, state)` are still accepted for backwards compatibility.

`make_estimand_fn` Optional factory function for a custom true estimand. Signature: `fn(beta_T)`, returning a function with signature `fn(y_linear_model, X, w, rep_data, state)`. Called once per grid cell with that cell's `beta_T` so the returned estimand function is always tied to the right effect size (important when `betaT` is a vector of multiple values). The returned function is invoked once per design class per replication *after* the design completes, so `w` (in {0, 1} encoding) and `X` reflect the realized assignment. Must return a numeric scalar. When supplied, its return value is used as the ground truth for *all* inference classes (overriding the `is_mean_diff` gate). Three-argument functions `fn(y_linear_model, state)` are still accepted for backwards compatibility (they will not receive `X` or `w`). Default `NULL` uses `beta_T` directly as the ground truth.

`dgp_params` Optional named list of DGP configuration values (e.g. `list(frailty_dist = "gamma", censoring_rate = 0.8)`). Injected into `state` as `state\$dgp_params` and accessible in all three custom-DGP hooks. Recommended over using closures to pass DGP parameters.

`custom_dgp` Optional function for a fully custom DGP. Signature: `fn(n, p, rep, state)` returning a list with components `X` (data.frame, `n` x `p`), `w` (integer vector in {0, 1}, length `n`), `y` (numeric, length `n`), `dead` (integer {0,1} or `NULL` for non-survival), `true_estimand`

(numeric scalar, optional). When supplied, the design class acts as a data container only; it does *not* run its own randomization or matching. Requires a fixed design class (not `DesignSeqOneByOne` variants). Cannot be combined with `custom_replication_data_generator` or `custom_apply_treatment_and_noise`.

`verbose` Logical. If `TRUE`, prints a message for every replication and for every `(design, inference)` pair that is skipped due to an error. Default `TRUE`.

`keep_all_intermediate_data` Logical. If `TRUE`, the framework saves the instantiated design and inference objects for every replication. These can be retrieved after the run using `$get_all_intermediate_data()`. Warning: this can consume a lot of memory for many replications. Default `FALSE`.

`turn_off_asserts_for_speed` Logical. If `TRUE` (default), all **checkmate** assertions across the package are globally disabled during the simulation run to improve performance.

`inference_types_and_params` `NULL` (default) or a named list from inference type to a named list of arguments for that type's function invocation. The list names control which inference outputs are computed. Valid names are `"asymp_ci"`, `"asymp_pval"`, `"exact_ci"`, `"exact_pval"`, `"boot_ci"`, `"boot_pval"`, `"rand_ci"`, and `"rand_pval"`. Each value must be a named list whose names are accepted by the corresponding inference function. `NULL` runs all eight types with default invocation arguments. Example:

```
inference_types_and_params = list(
  asymp_pval = list(delta = 0),
  boot_ci    = list(B = 99, type = "perc"),
  rand_pval  = list(r = 999, transform_responses = TRUE)
)
```

When no `*_ci` type is requested, `coverage` is omitted from `SimulationFrameworkReport$summarize()`. When no `*_pval` type is requested, `power` is omitted.

`results_filename` Character scalar. The filename for the results file. Supported extensions are `.csv` and `.csv.bz2`. Default `"simulation_framework_results.csv.bz2"`.

`continue_from_last_result_row` Logical. If `TRUE` (default), the framework loads existing results from `results_filename` and skips previously completed replications.

`reuse_cache` Logical. If `TRUE` (default), expensive pre-generated design / SE cache objects are loaded from disk when available. If `FALSE`, these cache objects are regenerated from scratch, but each regenerated object is still saved to disk for later restarts.

`stop_on_error` Logical. If `TRUE` (default), any error raised during a simulation path aborts the run immediately. If `FALSE`, the framework records the error, skips the failing path, and continues with the remaining replications / design / inference combinations. Use `$get_errors()` after `$run()` to inspect the captured errors.

`save_to_disk_every_n_rep` Positive integer. Results are flushed to the on-disk staging file only once every this many replications, and always after the final replication. Larger values reduce disk I/O overhead at the cost of losing more progress if the run is interrupted. Default `25L`.

`save_model_control_fits` Logical. If `TRUE` (default), after the design/SE cache is built, saves the per-subject model-implied potential outcomes under treatment and control as CSV files in a subfolder named `<stem>_response_values/` (where `<stem>` is `results_filename` with its `.csv`/`.csv.bz2` extension stripped) next to `results_filename`. One file is written per unique `(response_type, cond_exp_func_model, n, p, betaT)` cell. Only meaningful when `random_X_draws = FALSE`; silently skipped otherwise. Column names depend on `response_type`:

"continuous", "survival" columns yt and yc

"incidence", "proportion" columns pt and pc

"count" columns rt and rc

Default TRUE.

SimulationFramework$run(): Execute the configured simulation replications, run each requested design and inference method, collect estimates/p-values/CIs and errors, and return a SimulationFrameworkReport.

Usage:

```
SimulationFramework$run()
```

Returns: The SimulationFramework object itself (invisibly).

SimulationFramework$get_all_intermediate_data(): Retrieve the stored intermediate data (design and inference objects) for every replication. Only available if keep_all_intermediate_data = TRUE was passed to the constructor.

Usage:

```
SimulationFramework$get_all_intermediate_data()
```

Returns: A nested list containing the intermediate data for each replication, or NULL if not recorded.

SimulationFramework$clear_all_intermediate_data_and_gc(): Release all stored intermediate data and invoke the garbage collector. Useful after inspecting intermediate results to free memory before further processing. Sets the internal store to NULL and calls gc().

Usage:

```
SimulationFramework$clear_all_intermediate_data_and_gc()
```

Returns: The SimulationFramework object itself (invisibly).

SimulationFramework$clone(): The objects of this class are cloneable with this method.

Usage:

```
SimulationFramework$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
# Simple simulation with two designs and two inference methods.
# n/Nrep_W/num_boot/B_boot/r_rand are kept small so this example runs in a
# few seconds; a real simulation would use much larger values (this
# class's defaults, or larger still) for adequately precise estimates.
sim = SimulationFramework$new(
  response_type = "continuous",
  design_classes_and_params = list(
    DesignSeqOneByOneKK21 = list(lambda = 0.5, num_boot = 50L),
    DesignSeqOneByOneBernoulli = list()
  ),
  inference_classes_and_params = list(
```

```
    InferenceContinOLS = list(),
    InferenceContinKKOLSIVWC = list()
  ),
  n = 20, p = 3, Nrep_W = 2L, betaT = 1, B_boot = 50L, r_rand = 101L,
  results_filename = tempfile(fileext = ".csv.bz2"),
  continue_from_last_result_row = FALSE
)
sim$run()
report = SimulationFrameworkReport$new(sim)
report$summarize()
```

SimulationFrameworkReport

Reporting class for SimulationFramework results

Description

An R6 class for accessing and summarizing the results of a `SimulationFramework` run. It can be constructed either from a completed `SimulationFramework` object (via `SimulationFrameworkReport$new(sim)`) or by loading results from a previously saved CSV / CSV.BZ2 file (via `SimulationFrameworkReport$new("path/to/resul`

Details

When constructed from a `SimulationFramework` object all design/inference parameter metadata is preserved, so `$summarize()` can annotate each row with human-readable parameter strings. When constructed from a file only the raw results are available; parameter annotation columns will be empty strings.

Methods

Public methods:

- `SimulationFrameworkReport$new()`
- `SimulationFrameworkReport$get_results()`
- `SimulationFrameworkReport$get_errors()`
- `SimulationFrameworkReport$summarize()`
- `SimulationFrameworkReport$print()`
- `SimulationFrameworkReport$clone()`

`SimulationFrameworkReport$new()`: Create a new `SimulationFrameworkReport` object that stores simulation results, captured errors, and summary helpers returned by `SimulationFramework`.

Usage:

```
SimulationFrameworkReport$new(sim_or_filename, alpha = NULL)
```

Arguments:

`sim_or_filename` Either a completed `SimulationFramework` object or a character string giving the path to a `.csv` or `.csv.bz2` results file written by `SimulationFramework`.

alpha Numeric in $(0, 1)$. Significance level for coverage and power calculations. When sim_or_filename is a SimulationFramework object and alpha is NULL (default), the framework's own alpha is used. When loading from a file, defaults to 0.05.

SimulationFrameworkReport$get_results(): Get the raw per-replication results.

Usage:

```
SimulationFrameworkReport$get_results()
```

Returns: A data.table with one row per (replication, design, inference class, inference type).

SimulationFrameworkReport$get_errors(): Return all errors captured during the simulation run.

Usage:

```
SimulationFrameworkReport$get_errors()
```

Returns: A list of named lists, one per captured error. Each element includes the simulation cell metadata, replication number, design / inference path, user-supplied parameters, error stage, and error message. Empty when constructed from a file.

SimulationFrameworkReport$summarize(): Aggregate and summarize simulation results.

Usage:

```
SimulationFrameworkReport$summarize()
```

Returns: A data.table with one row per unique (response_type, cond_exp_func_model, n, p, betaT, design, inference, inference_type) combination. Columns include MSE, coverage, ci_length, and coverage_pval (when CI types were run; coverage_pval is the exact two-sided binomial test p-value of H0: true coverage = 1 - alpha), power (when betaT != 0 and p-value types were run), size and size_pval (when betaT == 0 and p-value types were run; size_pval is the exact two-sided binomial test p-value of H0: true size = alpha, suitable for multiplicity-corrected calibration checks across settings), and parameter annotation strings.

SimulationFrameworkReport$print(): Print a concise summary of the report.

Usage:

```
SimulationFrameworkReport$print()
```

SimulationFrameworkReport$clone(): The objects of this class are cloneable with this method.

Usage:

```
SimulationFrameworkReport$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
sim <- SimulationFramework$new(
  response_type = "continuous",
  design_classes_and_params = list(DesignFixedBernoulli),
  inference_classes_and_params = list(InferenceAllSimpleAverageDiff),
  n = 20L, Nrep_W = 5L, betaT = 1,
  results_filename = tempfile(fileext = ".csv"),
```

```
  verbose = FALSE, continue_from_last_result_row = FALSE
)
sim$run()
report <- SimulationFrameworkReport$new(sim)
report$get_results()
report$summarize()
```

---

summary.EDIInferenceSuiteResults

Summarizes an InferenceSuite run_all_inference() *result: counts by* status*, the estimate range across* status == "ok" *classes, and how many reject at* alpha*.*

Description

Summarizes an InferenceSuite run_all_inference() result: counts by status, the estimate range across status == "ok" classes, and how many reject at alpha.

Usage

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

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

Arguments

object	An EDIInferenceSuiteResults object, as returned by InferenceSuite$run_all_inference().
...	Ignored; present for S3 consistency with the generic.
x	A summary.EDIInferenceSuiteResults object, as returned by summary.EDIInferenceSuiteResults.

Value

An object of class summary.EDIInferenceSuiteResults, printable via its own print method.

x, invisibly.

summary_glm_lean *Lean GLM Summary (Skips Deviance Residual Quantiles)*

Description

A drop-in replacement for summary.glm that produces the identical coefficient table, dispersion estimate, and (optionally) correlation matrix, but **omits the five-number summary of the deviance residuals** (summary(object$deviance.resid)) that summary.glm() always computes and stores in its deviance.resid component. That residual summary is cheap for a single fit but adds up when summarizing thousands of GLM fits in a resampling loop (e.g. bootstrap or randomization replicates elsewhere in this package), so this function skips it entirely; the returned object's deviance.resid component is simply absent rather than populated, which will matter to code that calls print.summary.glm() on the result or otherwise inspects that field. Every other computation — dispersion estimation (Pearson X^2/df for Gaussian/Gamma/inverse-Gaussian families, fixed at 1 for Poisson/binomial, unless dispersion is supplied explicitly), the coefficient table (Wald z tests when dispersion is fixed/known, t tests with df.residual degrees of freedom when dispersion is estimated), and the optional correlation/symbolic.cor outputs, is identical to summary.glm.

Usage

```
summary_glm_lean(
  object,
  dispersion = NULL,
  correlation = FALSE,
  symbolic.cor = FALSE,
  ...
)
```

Arguments

object	A fitted glm object.
dispersion	The dispersion parameter for the fitting family; if NULL (default), estimated as in summary.glm (fixed at 1 for poisson/binomial, else the Pearson-residual-based moment estimate).
correlation	Logical; if TRUE, the estimated correlation matrix of the coefficients is returned and printed. Default FALSE.
symbolic.cor	Logical; if TRUE and correlation = TRUE, the correlation matrix is printed in symbolic form (see symnum) rather than as numbers. Default FALSE.
...	Currently unused; present only for signature compatibility with summary.glm.

Value

An object of class c("summary.glm") with the same components as summary.glm's return value **except** deviance.resid, which is not computed and is absent from the result.

See Also

summary.glm, of which this is a residual-summary-skipping variant.

Examples

```
fit = glm(rbinom(10, 1, 0.5) ~ rnorm(10), family = binomial)
summary_glm_lean(fit)
```

---

transform_cont_y_based_on_response_type

Transform continuous latent signal to the response type scale

---

Description

A helper function to transform a latent continuous signal to the scale appropriate for a given response_type, identical to the logic used within SimulationFramework but not used within SimulationFramework.

Usage

```
transform_cont_y_based_on_response_type(
  y_cont,
  response_type,
  n_ordinal_levels = 4L,
  proportion_epsilon = 1e-06,
  survival_min_time = 0.1,
  count_min_rate = 0L,
  count_shift = 0
)
```

Arguments

y_cont	Numeric vector. The latent continuous response signal.
response_type	Character scalar. One of "continuous", "incidence", "proportion", "count", "survival", "ordinal".
n_ordinal_levels	Positive integer. Number of ordinal categories when response_type = "ordinal". Default 4L.
proportion_epsilon	Numeric scalar. Small value added to proportion to avoid 0 and 1. Default 1e-6.
survival_min_time	Numeric scalar. Minimum survival time and shift. Default 0.1.
count_min_rate	Integer scalar. Minimum baseline rate for count response. Default 0L.
count_shift	Numeric scalar. Constant added to counts after zero-centering. Default 0.

Value

A numeric vector of transformed responses on the appropriate scale.

Examples

```
transform_cont_y_based_on_response_type(rnorm(10), 'incidence')
```

---

tune_EDI_for_this_machine

Benchmark this machine and tune EDI's performance-policy defaults to it

---

Description

Every performance-policy default shipped in EDI (whether an inference class uses a smart cold start, whether resampling reuses warm starts and at what sample sizes, which optimizer algorithm a family uses, and at what sample size parallel bootstrapping starts to beat serial) was measured empirically on the maintainer's machine. Yours differs – core count, cache, BLAS, compiler flags – so a policy that is net-positive there can be net-negative here, and vice versa. This function re-runs those benchmarks **on your hardware**, decides the winning setting per axis, saves the result to a per-user config file, and applies it immediately; every later `library(EDI)` re-applies it. Hardware changed? Re-run; it overwrites.

Usage

```
tune_EDI_for_this_machine(
  effort = c("standard", "quick", "thorough"),
  axes = NULL,
  families = NULL,
  n_grid = NULL,
  reps = NULL,
  num_cores_grid = NULL,
  converged_fn = NULL,
  quiet = FALSE,
  dry_run = FALSE,
  force = FALSE
)
```

Arguments

`effort`	One of `"standard"` (default; moderate sample-size grid and replicate count), `"quick"` (coarser grid, fewer replicates, warm-start families narrowed to those the shipped tables already name, parallel axis on the bootstrap operation only), or `"thorough"` (full grid, more replicates).
`axes`	Which axes to tune: any subset of `"cold_start"`, `"warm_start"`, `"optimizer"`, `"parallel"`. `NULL` (default) means all that are available here – see Details for when the optimizer and parallel axes are included.

`families`	Optional character vector of inference class names to restrict every axis to; `NULL` (default) tunes every class each axis governs.
`n_grid`	Optional integer vector of sample sizes overriding the effort tier's grid.
`reps`	Optional replicate count per timed cell overriding the effort tier's.
`num_cores_grid`	Optional integer vector of core counts (each $\geq$ 2) for the parallel axis; default `c(2, detectCores())`.
`converged_fn`	`function(inf) -> logical(1)`, required for the optimizer axis (see Details).
`quiet`	If `TRUE`, print nothing (no preamble, no progress bar, no summary).
`dry_run`	If `TRUE`, run every benchmark and print the would-be policy changes, but write no file and apply nothing.
`force`	If `FALSE` (default), refuse to run when the machine looks busy (see Details: "Idle machine") – in an interactive session you are asked whether to proceed anyway; non-interactively it is an error. `TRUE` skips the check. Applies under `dry_run` too, since a dry run still benchmarks.

Details

What is tuned. Four axes, each against the corresponding `get_*_dispatch_policy()` table: cold start (`get_cold_start_dispatch_policy`), warm start per resampling operation (`get_warm_start_dispatch_policy`), optimizer algorithm (`get_optimization_dispatch_policy`), and the parallel-vs-serial crossover sample size per family (`get_parallel_dispatch_policy`). The bootstrap confidence-interval *type* policy is a statistical-validity table, not a performance one, and is never touched; nor are entries in the parallel policy's serial blocklist that exist for parallel safety.

How a deviation is accepted. Per axis, per (family, sample size) cell, both settings are timed on identical synthetic data – interleaved (A/B/A/B) for the cold-start/warm-start/optimizer axes, and blocked (all serial reps, then all parallel reps) for the parallel axis, whose fork-cluster setup cost rules out per-replicate interleaving. A candidate displaces the shipped default only if its median time is at least 5% better *and* that improvement exceeds twice the candidate's own interquartile spread; ties keep the shipped default. The optimizer axis additionally requires the candidate to have converged on every replicate of the cell – speed never trumps a convergence failure. Only *deviations* from the shipped defaults are stored, in the exact shape the matching `set_*_dispatch_policy()` setter accepts, and they are merged into (never replacing) the shipped tables.

Progress. A single progress bar with a running estimated-time-left is redrawn in place as each benchmark cell completes – the same bar `InferenceSuite$run_all_inference()` shows.

Correctness gate. A timing win alone does not displace a shipped default: every accepted deviation is re-fit once under *both* settings on identical synthetic data and the outputs compared (point estimates for cold start/optimizer/parallel; the resampling operation's own output, RNG-matched, for warm start). A disagreement – or an unverifiable comparison – discards the deviation, with a `warning()` naming it; discarded deviations are available on the returned object via `attr(x, "discarded_by_correctness_gate")` and are never written to the config file or applied.

The optimizer axis and `converged_fn`. There is not yet a generic, class-independent accessor on an inference object that reports whether its fit converged, so the optimizer axis needs you to supply one as `converged_fn(inf)`. When `axes` is left `NULL` the optimizer axis is included only if `converged_fn` is given; asking for it explicitly without one is an error. An always-`TRUE` `converged_fn` disables the convergence guard and is **not** appropriate for a real tuning run.

The parallel axis. Runs only on Unix-alikes with at least two logical cores (it benchmarks a real fork cluster). Its preferred core count is *recorded only* – never applied at package load – you still opt into parallelism with `set_num_cores`.

Idle machine. Run this on an otherwise idle machine; a tuning run under contention measures the contention, not the hardware. Before benchmarking, the function checks the 1-minute load average against the core count and times a small fixed calibration operation for noise; if either says the machine is busy it refuses to run (interactively, it asks first) unless `force = TRUE`.

Value

Invisibly, an `EDILocalMachineTuning` object: the policy diffs (`policy_diffs`), the raw per-axis deviations (`raw_deviations`), the hardware fingerprint, the effort/grid/reps used, timing, and the config file path – printable via `print()`.

See Also

`get_local_EDI_optimization` to see what is saved, `clear_local_EDI_optimization` to return to shipped defaults; the underlying tables: `get_cold_start_dispatch_policy`, `get_warm_start_dispatch_policy`, `get_optimization_dispatch_policy`, `get_parallel_dispatch_policy`.

Examples

```
# See what would change without writing anything. force = TRUE skips the
# idle-machine contention guard (see Details/`force` above) -- an example
# must not fail just because the machine running R CMD check happens to
# be busy (e.g. a shared CI runner); the guard itself has its own
# dedicated tests (test-local-machine-tuning-assembly.R). Scoped to one
# axis/family/n rather than the full live registry effort = "quick"
# otherwise walks (narrowing "quick" itself to a handful of
# high-effect-size families across every axis is still open, see
# edi_tuning_effort_presets()'s docs) -- this keeps the example a
# few-second sanity check instead of a multi-minute benchmark run.
res = tune_EDI_for_this_machine(effort = "quick", dry_run = TRUE, force = TRUE,
                                axes = "cold_start", families = "InferenceIncidLogRegr",
                                 n_grid = 50L, reps = 1L)
print(res)
```

`unset_num_cores` *Unset the number of cores and stop parallel clusters*

Description

This function stops any global fork or mirai clusters stored in the package environment and resets the core count to serial execution.

Usage

```
unset_num_cores()
```

Value

Invisible NULL.

Examples

```
set_num_cores(2)
unset_num_cores()
```

Index

∗ **Cox proportional hazards**
InferenceSurvivalCoxPHRegr, 382
∗ **Cox regression**
InferenceSurvivalCoxPHRegr, 382
∗ **Gehan-Wilcoxon test**
InferenceSurvivalGehanWilcox, 386
∗ **Kaplan-Meier**
InferenceSurvivalKMDiff, 418
∗ **Miettinen-Nurminen**
InferenceIncidMiettinenNurminenRiskDiff, 282
∗ **Peto-Prentice test**
InferenceSurvivalGehanWilcox, 386
∗ **RMST**
InferenceSurvivalRestrictedMeanDiff, 426
∗ **Wilson score interval**
InferenceIncidNewcombeRiskDiff, 288
∗ **cumulative logit model**
InferenceOrdinalPropOddsRegr, 338
∗ **design**
EDI, 79
∗ **log-rank test**
InferenceSurvivalLogRank, 423
∗ **median regression**
InferenceContinQuantileRegr, 174
∗ **median survival time**
InferenceSurvivalKMDiff, 418
∗ **ordinal logistic regression**
InferenceOrdinalPropOddsRegr, 338
∗ **pair-matching**
EDI, 79
∗ **proportional hazards regression**
InferenceSurvivalCoxPHRegr, 382
∗ **proportional odds**
InferenceOrdinalPropOddsRegr, 338
∗ **quantile regression**
InferenceContinQuantileRegr, 174
InferencePropQuantileRegr, 366
∗ **restricted mean survival time**
InferenceSurvivalRestrictedMeanDiff, 426
∗ **ridit analysis**
InferenceOrdinalRidit, 341
∗ **risk difference**
InferenceIncidMiettinenNurminenRiskDiff, 282
InferenceIncidNewcombeRiskDiff, 288
∗ **survival analysis**
InferenceSurvivalLogRank, 423
...Cauchit, *314*
...Cloglog, *314*
...IVWC, *186*
...Probit, *314*
.normalize_optimizer_algorithm, *84*, *86*, *95*, *97*, *100*, *105*, *120*

InferenceIncidBinomialIdentityRiskDiff, *278*

block_and_cluster_ra, *19*
block_ra, *21*, *45*
browseURL, *376*
build_cox_data_cache_cpp, 10, *11–13*, *87–92*
build_stratified_cox_data_cache_cpp, *10*, 11, *11*, *91*, *92*

clear_local_EDI_optimization, 13, *119*, *461*
cluster_ra, *24*

Design, *14*, *17*, *19*, *22*, *24*, *26*, *29*, *33*, *36*, *38*, *40*, *42*, *47*, *50*, *52*, *54*, *56*, *58*, *61*, *64*, *68*, *71*, *74*, *76*, *77*, *373–375*
DesignFixed, *14*, *16*, *17*, *19*, *21*, *22*, *24*, *26*, *28–30*, *33*, *35–40*, *45*, *47*, *49*, *50*

DesignFixed$assign_w_to_all_subjects(), *18*
DesignFixedBernoulli, 14, *35, 36*, *54*
DesignFixedBinaryMatch, 16, *19*, *37*, *45*, *60*
DesignFixedBlockedCluster, 19, *21*, *24, 25*
DesignFixedBlocking, 21, *45*, *47*
DesignFixedCluster, 24
DesignFixedFactorial, 26
DesignFixedGreedy, 28, *31*, *34, 35*, *37*, *39, 40*, *49*
DesignFixedGreedyDOptimal, *14–16*, *28*, 30, *35*, *39*, *49*
DesignFixediBCRD, *14, 15*, 35, *58*, *60*
DesignFixedMatchingGreedyPairSwitching, 37
DesignFixedOptimal, 39
DesignFixedOptimalBlocks, 45
DesignFixedRerandomization, 49
DesignSeqOneByOne, *52*, *54*, *56*, *58*, *60, 61*, *64*, *68*, *71*, *73–77*, *123*, *126*
DesignSeqOneByOne$add_one_subject_to_experiment_and_assign(), *59*
DesignSeqOneByOneAtkinson, 52, *54*, *56*, *71*
DesignSeqOneByOneBernoulli, *52*, 54, *56*, *58*
DesignSeqOneByOneEfron, 56, *77*
DesignSeqOneByOneiBCRD, 58
DesignSeqOneByOneKK14, 60, *63, 64*, *68*, *71*, *123*, *126*, *143*, *155*, *186*, *190*, *197*, *268*, *395*
DesignSeqOneByOneKK21, *56*, 63, *67, 68*, *123*, *146*
DesignSeqOneByOneKK21stepwise, 67
DesignSeqOneByOnePocockSimon, 71
DesignSeqOneByOneRandomBlockSize, 73, *75*, *77*
DesignSeqOneByOneSPBR, 75
DesignSeqOneByOneUrn, 77

EDI, 79
EDI-package (EDI), 79
edi_build_info_cpp, 80
EDI_COMPREHENSIVE_SLOW_PATHS, 81, *378*
exact_jonckheere_terpstra_pval_cpp, 82
expand.grid, *27*
expand_adjacent_category_data_cpp, *319, 320*
expand_continuation_ratio_data_cpp, *306*
fast_adjacent_category_logit_cpp, *297*
fast_beta_regression, 84, *85*, *88*
fast_beta_regression_cpp, *84, 85*, *348*
fast_beta_regression_weighted_cpp, *348*, *350*
fast_beta_regression_with_var, 85
fast_beta_regression_with_var_cpp, *85*
fast_coxph_regression, *10–13*, 87, *89, 90*
fast_coxph_regression_cpp, *10, 11*, *87, 88*, 89, *91, 92*
fast_coxph_regression_prebuilt_cpp, *10, 11*, *13*, *89, 90*, 91
fast_dnbinom_mu_vec_cpp, *101, 102*
fast_hurdle_negbin_with_var_cpp, *179*
fast_identity_binomial_regression_cpp, *94, 95*, *222*
fast_identity_binomial_regression_weighted_cpp, *222*
fast_log_binomial_regression_cpp, *94, 95*, *278*
fast_log_binomial_regression_weighted_cpp, *278*
fast_logistic_regression, 92, *95*
fast_logistic_regression_cpp, *93*, 94, *96, 97*, *99, 100*, *280*, *353*
fast_logistic_regression_weighted_cpp, 96, *280*
fast_logistic_regression_with_var, *93*, 97
fast_logistic_regression_with_var_cpp, *95*, *97*, 99, *99*
fast_neg_bin_cpp, *101, 102*, *199*
fast_neg_bin_weighted_cpp, *200*
fast_neg_bin_with_var_cpp, *102, 103*, *199*
fast_negbin_regression, 100, *103*
fast_negbin_regression_with_var, *101*, 102, *102*
fast_ols_cpp, 103, *104, 105*, *171, 172*, *435*
fast_ols_with_var_cpp, 104, *104*, *171, 172*
fast_ordinal_glmm_cpp, *324*
fast_ordinal_probit_regression_cpp, *328*
fast_ordinal_probit_regression_with_var_cpp, *328*
fast_ordinal_regression_weighted_cpp, *310*, *323*
fast_ordinal_regression_with_var_cpp, *308*, *334*, *338*

fast_poisson_regression_cpp, 105, *107–112*, *203*, *215, 216*
fast_poisson_regression_weighted_cpp, 107, *107*, *209*, *213*, *216*
fast_poisson_regression_with_var_cpp, *107*, 109, *112*, *203*, *286*
fast_probit_regression_cpp, *291*
fast_quasipoisson_regression_with_var_cpp, *107*, 111, *111*, *212–214*
fast_robust_regression_cpp, *177, 178*
fast_stereotype_logit_cpp, *344*
fast_weibull_regression, 113, *440*
fast_weibull_regression_cpp, *432*
fast_weibull_regression_general_cpp, *113, 114*
fisher.test, *229, 230*

gcomp_fractional_logit_point_estimate_cpp, *353*
gcomp_logistic_point_estimate_cpp, *235*, *249*
gcomp_ordinal_proportional_odds_post_fit_cpp, *308*, *310*
generate_covariate_dataset, 115
get_bootstrap_dispatch_policy, 116, *118*, *120, 121*
get_cold_start_dispatch_policy, 118, *120–122*, *441, 442*, *460, 461*
get_local_EDI_optimization, *13*, 119, *461*
get_optimization_dispatch_policy, *117, 118*, 119, *121*, *443, 444*, *460, 461*
get_parallel_dispatch_policy, *117*, 120, *444, 445*, *460, 461*
get_warm_start_dispatch_policy, 121, *121*, *445, 446*, *460, 461*
glm, *457*
glm.nb, *438*

Inference, *26*, *124*, *128*, *132*, *136*, *139*, *144*, *147*, *149*, *153*, *156*, *159*, *161*, *163*, *166*, *169*, *171*, *174*, *177*, *180*, *184*, *187*, *190*, *194*, *197*, *200*, *204*, *210*, *212*, *215*, *217*, *220*, *223*, *225*, *227*, *229*, *231*, *233*, *235*, *249*, *263, 264*, *266*, *268*, *270*, *273, 274*, *276*, *279*, *281*, *283*, *287, 288*, *291*, *293*, *296*, *298*, *301*, *303*, *306*, *309*, *312*, *314*, *316–318*, *320*, *323*, *325*, *328*, *331*, *334*, *338*, *341*, *345*, *348*, *351*, *354*, *360, 361*, *363, 364*, *366*, *369*, *372*, *382*, *385*, *387*, *393*, *396*, *399*, *401*, *404*, *407*, *409*, *412*, *415*, *417, 418*, *423*, *427*, *430*, *433*
InferenceAbstractKKCondLogitGLMM, *263, 264*, *361*
InferenceAbstractKKMarginalIncid, *276*
InferenceAbstractKKModifiedPoisson, *276*
InferenceAbstractKKOrdinalCLMM, *314–318*
InferenceAllKKMeanDiffIVWC, 123, *128*, *152*
InferenceAllKKWilcoxIVWC, 126, *126*
InferenceAllSimpleAverageDiff, 131, *135*, *138*
InferenceAllSimpleMeanDiffPooledVar, 135, *135*
InferenceAllSimpleWilcox, *127*, 138, *138*
InferenceAsymp, *131, 132*, *135, 136*, *170*, *173*, *175, 176*, *178, 179*, *186*, *191–193*, *205*, *214*, *216*, *225*, *234*, *321*, *326*, *333*, *356*, *367, 368*
InferenceAsympLik, *197*, *268*, *349*, *370*
InferenceBaiAdjustedTKK14, 143, *146*
InferenceBaiAdjustedTKK21, *143*, 146
InferenceBayesianBootstrap, *137*, *175*, *178*, *213*, *216*, *284*, *289*, *367*, *388*, *424*
InferenceContinKKGLMM, 149
InferenceContinKKOLSIVWC, *124*, 152, *158*
InferenceContinKKOLSOneLik, 155, *160*, *166*
InferenceContinKKQuantileRegrIVWC, 158, *159*, *162*
InferenceContinKKQuantileRegrOneLik, 160, *161*, *365*
InferenceContinKKRobustRegrIVWC, 163, *168*
InferenceContinKKRobustRegrOneLik, 165
InferenceContinLin, 168, *169*, *171*
InferenceContinOLS, 171, *171, 172*
InferenceContinQuantileRegr, 174, *368*
InferenceContinRobustRegr, 176
InferenceCountHurdleNegBin, 179, *180*, *184*, *186*
InferenceCountHurdlePoisson, 183, *184*, *221*

InferenceCountKKCondPoissonOneLik, 186
InferenceCountKKGLMM, 189
InferenceCountKKHurdlePoissonIVWC, 194, *196*
InferenceCountKKHurdlePoissonOneLik, 196
InferenceCountLikelihood, *180, 181*
InferenceCountNegBin, *183*, 199, *200*, *203*, *219*
InferenceCountPoisson, *186*, *199*, 203, *204–206*, *208*, *215*, *221, 222*
InferenceCountPoissonKKGEE, 210, *210*
InferenceCountQuasiPoisson, 212, *215*
InferenceCountRobustPoisson, 215
InferenceCountZeroInflatedNegBin, 217, *218*, *221*
InferenceCountZeroInflatedPoisson, *203*, 219, *219, 220*
InferenceCustomRand, *372*
InferenceIncidBinomialIdentityRiskDiff, 222
InferenceIncidCMH, 224, *226*
InferenceIncidExactBinomial, 227, *231*
InferenceIncidExactFisher, 229, *231*
InferenceIncidExactZhang, 231
InferenceIncidExtendedRobins, 233
InferenceIncidGCompRiskDiff, 235, *262*
InferenceIncidGCompRiskRatio, 248, *248*
InferenceIncidKKCondLogitGLMMIVWC, 262, *264*, *361*
InferenceIncidKKCondLogitGLMMOneLik, *263*, 264, *264*, *361*
InferenceIncidKKCondLogitIVWC, 265, *268*
InferenceIncidKKCondLogitOneLik, 267, *267*
InferenceIncidKKGCompRiskDiff, 269, *272*
InferenceIncidKKGCompRiskRatio, *271*, 272
InferenceIncidKKGEE, 274, *274*
InferenceIncidKKModifiedPoisson, 275
InferenceIncidKKNewcombeRiskDiff, 277, *277*
InferenceIncidLogBinomial, *224*, 278, *286*, *288*
InferenceIncidLogRegr, *222*, *224*, *278*, 280, *280*, *291, 292*
InferenceIncidMiettinenNurminenRiskDiff, 282, *288*, *436*
InferenceIncidModifiedPoisson, *276*, 286
InferenceIncidNewcombeRiskDiff, 288, *296*
InferenceIncidProbitRegr, 291
InferenceIncidRiskDiff, 293, *296*
InferenceIncidWald, 296, *296*
InferenceJackknife, *127*, *139*, *182*, *201, 202*
InferenceMLEorKMSummaryTable, *132*, *419*, *427*
InferenceNonParamBootstrap, *132*, *235*, *249*, *310*, *420*, *427*
InferenceOrdinalAdjCatLogitRegr, 297, *322*
InferenceOrdinalCauchitRegr, 300, *330*
InferenceOrdinalCloglogRegr, 303, *330*
InferenceOrdinalContRatioRegr, 305, *344*, *347*
InferenceOrdinalGCompMeanDiff, 308
InferenceOrdinalJonckheereTerpstraTest, 311, *341*
InferenceOrdinalKKCLMM, 314
InferenceOrdinalKKCLMMCauchit, 315
InferenceOrdinalKKCLMMCloglog, 317
InferenceOrdinalKKCLMMProbit, 318
InferenceOrdinalKKCondAdjCatLogitRegr, *308*, 319, *344*
InferenceOrdinalKKGEE, 322, *323*, *325*
InferenceOrdinalKKGLMM, 324
InferenceOrdinalOrderedProbitRegr, 328
InferenceOrdinalPairedSignTest, 331, *331*
InferenceOrdinalPartialProportionalOddsRegr, 334
InferenceOrdinalPropOddsRegr, 338, *338*
InferenceOrdinalRidit, 341
InferenceOrdinalStereotypeLogitRegr, *308*, 344
InferencePropBetaRegr, 348, *349*, *353*, *371*
InferencePropFractionalLogit, *286*, *348*, *350*, 351, *352, 353*, *359*
InferencePropGCompMeanDiff, 353
InferencePropKKGEE, 359, *360*
InferencePropKKGLMM, *263, 264*, 361, *361*
InferencePropKKQuantileRegrIVWC, 362, *363*
InferencePropKKQuantileRegrOneLik, 364
InferencePropQuantileRegr, 366

InferencePropZeroOneInflatedBetaRegr, *281*, *348*, 369, *370*
InferenceRand, *174*, *177*, *212*, *215*, *283*, *289*, *309*, *312*, *331*, *335*, *342*, *354*, *366*, *382*, *387*, *419*, *423*, *427*, *430*
InferenceRandCI, *268*
InferenceRandCustom, 372
InferenceSuite, 373, *437*, *456*
InferenceSurvivalCoxPHRegr, 382, *429*
InferenceSurvivalDepCensTransformRegr, 385
InferenceSurvivalGehanWilcox, *138*, 386, *387*, *425*
InferenceSurvivalGLMMWeibullFrailtyLoggammaIVWC, 392, *398*
InferenceSurvivalGLMMWeibullFrailtyLoggammaOneLik, *386*, 395, *401*
InferenceSurvivalGLMMWeibullFrailtyNormalIVWC, *392*, *395*, 398, *398*
InferenceSurvivalGLMMWeibullFrailtyNormalOneLik, *392*, 400, *401*, *417*
InferenceSurvivalKKLWACoxPHIVWC, 403, *406*
InferenceSurvivalKKLWACoxPHOneLik, 406
InferenceSurvivalKKRankRegrIVWC, 409
InferenceSurvivalKKStratCoxPHIVWC, 411, *414*
InferenceSurvivalKKStratCoxPHOneLik, 414
InferenceSurvivalKKWeibullMarginal, 417
InferenceSurvivalKMDiff, 418, *419*
InferenceSurvivalLogRank, 423, *423*
InferenceSurvivalRestrictedMeanDiff, 426, *427*
InferenceSurvivalStratCoxPHRegr, 429, *430*
InferenceSurvivalWeibullRegr, *385*, 432
inv_logit, 434, *436*
is, *117*, *375*
IVWC, *364*

KKNewcombeRiskDiffIVWCSource *(*InferenceIncidKKNewcombeRiskDiff*)*, 277

lm.fit, *155*
logit, 435, *435*

makeForkCluster, *377*
mantelhaen.test, *229*
MarginalEstimand, *203*, *212*, *222*, *280*, *348*, *369*
mn_constrained_mle_pc_cpp, *436*
mn_pvalue_cpp, 436
mn_z_statistic_cpp, *436*
modifyList, *442*, *444–446*

nonbimatch, *16*

ols_hc2_post_fit_cpp, *168*, *170*
OneLik, *403*

print.EDIInferenceSuiteResults, 437
print.summary.EDIInferenceSuiteResults *(*summary.EDIInferenceSuiteResults*)*, 456

qr, *101*, *102*

robust_negbinreg, 438
robust_survreg, 438
robust_survreg_with_surv_object, *438*, 439, *439*
rq, *366*

sample, *26*, *73*
sample_mode, 441
set_cold_start_dispatch_policy, *118*, 441, *444*, *446*
set_estimand(), *184*, *219*
set_num_cores, *50*, 442, *445*, *461*
set_optimization_dispatch_policy, *442*, 443
set_parallel_dispatch_policy, 444
set_warm_start_dispatch_policy, *122*, *442*, *444*, 445
setTimeLimit, *377*
SimulationFramework, 446, *447*, *454*
SimulationFrameworkReport, *453*, 454, *454*
solve, *103*
stats, *169*
summary.EDIInferenceSuiteResults, 456, *456*
summary.glm, *457*, *458*
summary_glm_lean, 457
Surv, *438*, *440*

SurvivalKKWeibullMarginalSource
(InferenceSurvivalKKWeibullMarginal), 417
survreg, *113*, *438–440*
symnum, *457*

toggle_asserts, *375*
transform_cont_y_based_on_response_type, 458
tune_EDI_for_this_machine, *13*, *118–122*, *442*, *444–446*, 459

unset_num_cores, 461