Package 'deli'

August 28, 2026

Title M-Estimation and Empirical Sandwich Variance Estimation

Version 0.1.0

Description Solves estimating equations to obtain M-estimators together with empirical sandwich variance estimates, providing a general interface for both custom and built-in estimating equations. Built-in equations cover basic statistics, regression, causal inference, survival analysis, measurement error, and more. An 'R' port of the 'Python' library 'delicatessen' described in 'Zivich et al. (2022)' <doi:10.48550/arXiv.2203.11300>.

License MIT + file LICENSE

URL https://github.com/r-causal/deli, https://r-causal.github.io/deli/

BugReports https://github.com/r-causal/deli/issues

Depends R (>= 4.3)

Imports cli, generics, rlang (>= 1.0.0), rootSolve, S7 (>= 0.2.0), stats

Suggests causaldata, jsonlite, knitr, MASS, minpack.lm, nleqslv, nlme, quarto, readxl, rmarkdown, testthat (>= 3.0.0), yaml

VignetteBuilder knitr

Config/Needs/website ggplot2, quarto, reticulate

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first confidence-bands, autodiff, predictions, estimate-solvers, validation, formula-interface

Encoding UTF-8

Language en-US

LazyData true

LazyDataCompression xz

NeedsCompilation no

Author Malcolm Barrett [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0003-0299-5825>),
Paul Zivich [ctb] (ORCID: <https://orcid.org/0000-0002-9932-1095>,
Author of the Python 'delicatessen' library, whose design deli
ports)

Maintainer Malcolm Barrett <malcolmbarrett@gmail.com>

Repository CRAN

Date/Publication 2026-08-28 09:40:02 UTC

Contents

`actg175` *ACTG 175 clinical trial data*

Description

Data from AIDS Clinical Trials Group Study 175, used to demonstrate confidence bands for treatment effects.

Usage

```
actg175
```

Format

A data frame.

References

Hammer SM, et al. (1996). A trial comparing nucleoside monotherapy with combination therapy in HIV-infected adults with CD4 cell counts from 200 to 500 per cubic millimeter. *New England Journal of Medicine*, 335(15), 1081-1090.

`additive_design_matrix` *Build an additive design matrix for GAMs*

Description

Constructs the expanded design matrix for generalized additive models by appending spline basis terms according to per-column specifications. Each column in `X` keeps its linear term; columns with a non-`NULL` specification get additional spline basis columns appended.

This function mirrors `additive_design_matrix()` in Python delicatessen, so code translated from Python can keep its shape. There is no base R equivalent of this construction, so this is the interface for it in deli as well.

Usage

```
additive_design_matrix(X, specifications, return_penalty = FALSE)
```

Arguments

X	Numeric matrix (n-by-b) of input covariates.
specifications	A list of length b (number of columns in X). Each element is either NULL (no spline for that column) or a list with: **knots** Numeric vector of knot locations (required). **natural** Logical, generate restricted splines? Default TRUE. **power** Numeric power for spline. Default 3. **penalty** Numeric penalty for spline terms. Default 0. **normalized** Logical, normalize spline terms? Default FALSE.
return_penalty	Logical. If TRUE, return a list with both the design matrix and the penalty vector. Default FALSE.

Value

If return_penalty = FALSE, a numeric matrix. If TRUE, a list with elements X (the design matrix) and penalty (numeric vector).

Examples

```
set.seed(42)
X <- cbind(rnorm(50), rnorm(50))

# The first column stays linear; the second also gets penalized splines
specs <- list(NULL, list(knots = c(-1, 0, 1), penalty = 5))
out <- additive_design_matrix(X, specs, return_penalty = TRUE)

# Linear terms are unpenalized, spline terms carry the requested penalty
out$penalty

dim(out$X)
```

---

aft_predictions_function

Function-level predicted survival measures from an AFT model

Description

Computes predicted survival analysis measures and point-wise confidence intervals from an accelerated failure time model for a single covariate pattern across a set of time points. The point estimates mirror aft_predictions_individual(); the variance is obtained with the delta method (see delta_method()) and Wald-type intervals are formed on the resulting standard errors. Meant to be used after fitting ee_aft() with MEstimator(), typically to draw a measure and its confidence band over time.

Usage

```
aft_predictions_function(
  X,
  times,
  theta,
  covariance,
  distribution,
  measure = "survival",
  alpha = 0.05,
  deriv_method = "capprox",
  dx = 1e-09
)
```

Arguments

X	Numeric 1-by-b design matrix giving a single covariate pattern. More than one row is an error, since each pattern has its own variance.
times	Numeric vector of time points for prediction.
theta	Numeric vector of estimated parameters from ee_aft.
covariance	Numeric covariance matrix from vcov(m).
distribution	Character string matching the distribution used in ee_aft.
measure	Character string: "survival", "risk", "density", "hazard", or "cumulative_hazard". Default "survival".
alpha	Numeric significance level. Default 0.05 (95% CIs).
deriv_method	Character string for the derivative method used to build the delta-method Jacobian. One of "capprox" (central difference), "fapprox" (forward difference), "bapprox" (backward difference), or "exact" (forward-mode automatic differentiation). Default "capprox". Python Delicatessen uses exact differentiation internally; pass deriv_method = "exact" to reproduce it with exact derivatives and no step-size tuning. See delta_method().
dx	Numeric step size for the finite-difference methods; ignored when deriv_method = "exact". Default 1e-9. Must be a single positive finite number, which is checked whichever deriv_method is in force. The step is absolute and is floored at the floating-point resolution of each estimate, so a large parameter magnitude cannot silently reduce it to nothing; see approx_differentiation().

Details

The survival and hazard for a covariate pattern X are

$$S(t) = S_\epsilon\left(\frac{\log(t) - X\beta^T}{\sigma}\right)$$

$$h(t) = (\sigma t)^{-1} h_\epsilon\left(\frac{\log(t) - X\beta^T}{\sigma}\right)$$

where S_ϵ and h_ϵ are the error survival and hazard functions for the chosen distribution. The requested measure is derived from these through convert_survival_measures().

Value

A data frame with one row per time point and columns: `time`, `predicted`, `variance`, `lower`, `upper`.

Length-one times

A single time point is supported and returns a one-row data frame equal to the corresponding row of a multi-time call. This is a deliberate improvement over Python Delicatessen, whose `aft_predictions_function` raises on a scalar or length-one `times` because it takes the diagonal of a scalar delta-method covariance.

Examples

```
# Weibull AFT fit, then a survival curve for one covariate pattern
set.seed(1)
n <- 200
x <- rbinom(n, 1, 0.5)
Xd <- cbind(1, x)
eps <- log(-log(runif(n)))
t_event <- exp(2 + 0.5 * x + 0.8 * eps)
t_censor <- rexp(n, rate = 0.02)
t_obs <- pmin(t_event, t_censor)
delta <- as.numeric(t_event <= t_censor)

psi <- function(theta) {
  ee_aft(theta, X = Xd, time = t_obs, event = delta, distribution = "weibull")
}
m <- m_estimate(
  stacked_equations = psi,
  init = c(mean(log(t_obs)), 0, 0),
  solver = "nleqslv"
)

aft_predictions_function(
  X = matrix(c(1, 1), nrow = 1), times = c(5, 10, 20),
  theta = coef(m), covariance = vcov(m),
  distribution = "weibull", measure = "risk"
)
```

`aft_predictions_individual`

Predicted survival measures from an AFT model

Description

Computes individual-level predicted survival measures from an accelerated failure time model at specified time points. Meant to be used after fitting `ee_aft()` with `MEstimator()`.

Usage

```
aft_predictions_individual(X, times, theta, distribution, measure = "survival")
```

Arguments

X	Numeric n-by-b design matrix of covariate values.
times	Numeric vector of time points for prediction.
theta	Numeric vector of estimated parameters from ee_aft.
distribution	Character string matching the distribution used in ee_aft.
measure	Character string: "survival", "risk", "density", "hazard", or "cumulative_hazard". Default "survival".

Value

A data frame with n rows and one column per time point.

Examples

```
# Weibull AFT fit, then individual-level survival for the first four people
set.seed(1)
n <- 200
x <- rbinom(n, 1, 0.5)
Xd <- cbind(1, x)
eps <- log(-log(runif(n)))
t_event <- exp(2 + 0.5 * x + 0.8 * eps)
t_censor <- rexp(n, rate = 0.02)
t_obs <- pmin(t_event, t_censor)
delta <- as.numeric(t_event <= t_censor)

psi <- function(theta) {
  ee_aft(theta, X = Xd, time = t_obs, event = delta, distribution = "weibull")
}
m <- m_estimate(
  stacked_equations = psi,
  init = c(mean(log(t_obs)), 0, 0),
  solver = "nleqslv"
)

# Rows are individuals and columns are the requested times. The first two
# people share a covariate pattern, as do the third and fourth, so their
# predictions agree.
aft_predictions_individual(
  X = Xd[1:4, ], times = c(5, 10, 20),
  theta = coef(m), distribution = "weibull", measure = "survival"
)
```

`aggregate_efuncs` *Aggregate estimating function contributions by group*

Description

Collapses unit-level estimating function contributions into group-level contributions for clustered or grouped data. Uses an independent working correlation structure (summing within groups).

This function mirrors `aggregate_efuncs()` in Python delicatessen, so code translated from Python can keep its shape. There is no base R equivalent that operates on estimating function contributions, so this is the interface for them in deli as well.

Usage

```
aggregate_efuncs(est_funcs, group)
```

Arguments

`est_funcs`	A p-by-n matrix of estimating function contributions, where p is the number of parameters and n is the number of observations. A length-n vector is treated as a single parameter observed across n observations, matching a 1-by-n matrix.
`group`	A vector of length n identifying the group (cluster) for each observation.

Details

This function should be called inside the `psi` function after computing unit-level estimating equations but before returning them to `MEstimator()`. This changes the effective sample size used by the empirical sandwich variance estimator.

Value

A p-by-m matrix, where m is the number of unique groups. Row names are those of `est_funcs`, since the rows are the same parameters. Columns are ordered by the sorted unique values of `group` and are labeled with those values, as character. A factor `group` is coerced with `as.vector()` to its character labels before sorting, so its columns sort lexically by label rather than by factor-level order and carry those labels; a level with no observations contributes no column and so no label.

Examples

```
# Fifty clusters of four observations, sharing a cluster-level shift in y
set.seed(42)
n <- 200
group <- rep(1:50, each = 4)
cluster_effect <- rnorm(50, sd = 2)
y <- cluster_effect[group] + rnorm(n)

psi <- function(theta) aggregate_efuncs(ee_mean(theta, y = y), group = group)

m <- m_estimate(stacked_equations = psi, init = mean(y))
```

```
# Cluster-robust standard error, larger than the naive independence version
sqrt(diag(vcov(m)))
```

`bonate_adverse` *Bonate adverse events data*

Description

Adverse events case study data from Bonate (2011), used to demonstrate generalized linear models (logistic, probit, complementary log-log).

Usage

```
bonate_adverse
```

Format

A data frame.

References

Bonate PL. (2011). *Pharmacokinetic-Pharmacodynamic Modeling and Simulation*. 2nd Ed. Springer.

`breast_cancer` *Breast cancer survival data*

Description

Survival data from Collett (2015) on the survival times of 45 women with breast cancer in Middlesex Hospital July 1987 (Table 1.2).

Usage

```
breast_cancer
```

Format

A data frame with 45 rows and 3 columns:

delta Event indicator (1 = event, 0 = censored)

times Observation time, in months

stain Tumor HPA stain indicator (1 = positive, 0 = negative)

References

Collett, D. (2015). Survival analysis. In Modelling survival data in medical research. 3rd Ed. Chapman and Hall/CRC. pg 6-7

collett_bladder *Collett bladder cancer recurrence data*

Description

Bladder cancer recurrence data from Collett (2015), used to demonstrate pooled logistic regression for survival analysis.

Usage

```
collett_bladder
```

Format

A data frame with 86 rows and 6 columns:

patient Patient ID

time Follow-up time, in months

delta Event indicator

treat Treatment group

init Number of initial tumors

size Size of largest initial tumor

References

Collett, D. (2015). Modelling survival data in medical research. 3rd Ed. Chapman and Hall/CRC.

compute_confidence_bands *Compute confidence bands from theta and covariance*

Description

Compute confidence bands from theta and covariance

Usage

```
compute_confidence_bands(
  theta,
  covariance,
  alpha = 0.05,
  method = "supt",
  n_draws = 100000L,
  seed = NULL
)
```

Arguments

theta	Numeric parameter vector.
covariance	Numeric covariance matrix.
alpha	Significance level. Default 0.05.
method	"supt" or "bonferroni". Default "supt".
n_draws	Number of MVN draws for sup-t. Default 1e5. See confidence_bands() for why this differs from Python's 1e6 estimator default and how to match it.
seed	RNG seed. Default NULL.

Value

A p-by-2 matrix with columns "lower" and "upper". Rows take their names from theta, when it has any.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                  model = "linear")

# Bands from the estimates and covariance alone, without the fitted object
compute_confidence_bands(coef(fit), covariance = vcov(fit),
                         method = "supt", seed = 1)
```

---

compute_sandwich *Compute the empirical sandwich variance estimator*

Description

Computes the empirical sandwich variance estimator directly from a set of estimating equations and a vector of parameter estimates. Unlike MEstimator(), this function does not solve for the parameters; it assumes that theta is already the root of the estimating equations and only assembles the sandwich covariance at that point.

Usage

```
compute_sandwich(
  stacked_equations,
  theta,
  deriv_method = "capprox",
  dx = 1e-09,
  allow_pinv = TRUE,
  finite_correction = NULL,
  summed_equations = NULL,
  check_summed_equations = TRUE
)
```

Arguments

`stacked_equations`
: A function that takes a numeric vector `theta` and returns a p-by-n matrix of estimating equation contributions, where p is the number of parameters and n is the number of observations. A list of one element per equation, each holding that equation's contributions across the observations, is accepted as well. `estimate()` has no support for that shape, so an estimating function written in it reaches a variance through this entry point and a Jacobian through `compute_bread()`. The list form holds exactly one element per parameter, each of one length, so an over-identified system reaches this function as a matrix rather than as a list.

`theta`
: Numeric vector of parameter estimates. This function assumes `theta` is the root of `stacked_equations`; it does not solve for it.

`deriv_method`
: Character string selecting the method used to build the bread Jacobian. One of `"capprox"` (central difference), `"fapprox"` (forward difference), `"bapprox"` (backward difference), or `"exact"` (forward-mode automatic differentiation). Default `"capprox"`.

  This default differs from Python delicatessen, whose `compute_sandwich` defaults to `"approx"`, a forward difference computed through SciPy's `approx_fprime`. deli does not replicate the SciPy `"approx"` path; its `"fapprox"` is the hand-implemented forward difference. Code ported across the two libraries should set `deriv_method` explicitly rather than rely on the default.

`dx`
: Numeric step size for the finite-difference methods; ignored when `deriv_method = "exact"`. A small value is recommended, since large steps can give poor approximations. Default `1e-9`. Must be a single positive finite number, which is checked whichever `deriv_method` is in force. The step is absolute and is floored at the floating-point resolution of each estimate, so a large parameter magnitude cannot silently reduce it to nothing; see `approx_differentiation()`.

`allow_pinv`
: Logical. When `TRUE` (default), the Moore-Penrose pseudo-inverse is used where the bread matrix cannot be solved, which is the case for the rectangular bread of an over-identified system as well as for a square one whose solve fails; when `FALSE`, a bread with no inverse raises an error carrying the class `deli_bread_not_invertible`.

  What the two settings choose between is what to do when the solve fails, and nothing else: a bread the solve inverts is inverted under either. An ill-conditioned bread that base R returns an inverse for is one of those, however far its condition number runs, so `allow_pinv = FALSE` is not a conditioning test and a caller who wants one applies it to the returned matrix.

`finite_correction`
: Character string or `NULL`. Finite-sample correction applied to the meat matrix. `NULL` (default) applies no correction; `"HC1"` rescales the meat by $n/(n-p)$, where p is the number of parameters.

`summed_equations`
: A function that takes a numeric vector `theta` and returns the length-p vector of row sums of `stacked_equations` at `theta`, or `NULL` (default) to derive those sums from the full p-by-n return.

  The bread is the Jacobian of the summed estimating equations, so each of the one or two perturbed evaluations it makes per parameter is reduced to one value

per equation as soon as it is built. Deriving the reduction builds the whole p-by-n matrix 2p times for arithmetic that is linear in it; an estimating function whose sums have a closed form, such as the $X^T r$ of a regression score, can supply them and hand the bread only what it uses. The meat is unaffected either way, since it needs the per-observation contributions and takes the one full evaluation it always took.

Under `deriv_method = "exact"` the reduction is called with a tangent-carrying `theta`, so it must be written in operations that carry derivatives: `t(X) %*% r` does, and `base::crossprod()` does not. See `auto_differentiation()` for which operations carry a tangent and where.

An argument that is neither `NULL` nor a function raises an error carrying the class `deli_summed_equations_error`, whatever `check_summed_equations` says. So does a return at `theta` that is not numeric, or one holding fewer values than there are parameters, both of which `compute_bread()` reads before it differentiates anything. That the return holds exactly one value per estimating equation is read by the comparison below, since only a call that has evaluated the estimating functions knows how many of them there are.

`check_summed_equations`

Logical. When `TRUE` (default) and `summed_equations` was supplied, its value at `theta` is compared against the row sums of the one full evaluation the meat is built from, and a disagreement raises an error carrying the class `deli_summed_equations_disagree`. The comparison costs one call to `summed_equations` and one reduction of a matrix the call already holds. Set it to `FALSE` to skip the comparison, which leaves a reduction that sums some other system to return a matrix with the shape of a covariance and no claim to be one.

The comparison is made at `theta`, which the caller states is the root, so both quantities are at rounding there. That is enough to catch a reduction of some other system and not enough to catch a reduction that is a multiple of the right one, which agrees at a root and yields that multiple of the right bread.

Details

The sandwich is built from a bread matrix and a meat matrix. The bread is the negative Jacobian of the summed estimating equations, and the meat is the cross-product of the equation evaluations. Each is scaled by `1/n` internally, so the returned matrix is on the asymptotic scale (see `Value`). The bread Jacobian is obtained either by finite differences or by forward-mode automatic differentiation.

Value

A p-by-p covariance matrix on the asymptotic scale. The bread and meat are each divided by `n` internally, so the returned matrix is the variance that corresponds to the standard deviation. Dividing it by the number of observations gives the standard-error-scale variance, whose square-rooted diagonal is the vector of standard errors.

A covariance matrix is the only thing this function returns. Where the bread has no inverse, and so no sandwich can be assembled from it, the call raises an error carrying the class `deli_bread_not_invertible` rather than returning something that has to be tested for. That covers a bread holding `NA`, and, under `allow_pinv = FALSE`, a rectangular bread and one the solve could not invert. `estimate()` makes

the other choice from the same matrices: a fit whose bread holds NA warns and comes back with no variance, since it still carries the estimates.

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

See Also

MEstimator() and GMMEstimator(), which solve for theta and report this variance internally, and delta_method() for the variance of a transformation of the parameters.

Examples

```
# A generic data set for estimating a mean and variance
y <- c(1, 2, 4, 1, 2, 3, 1, 5, 2)

# The mean and variance are the roots of ee_mean_variance, so they can be
# computed directly rather than solved for
theta <- c(mean(y), stats::var(y) * (length(y) - 1) / length(y))

# Wrap the built-in estimating equation as a function of theta alone
psi <- function(theta) ee_mean_variance(theta, y = y)

# compute_sandwich() returns the asymptotic-scale variance, so dividing by
# n puts it on the standard-error scale
sandwich <- compute_sandwich(psi, theta = theta) / length(y)
sandwich

# The diagonal square roots are the standard errors
sqrt(diag(sandwich))

# The bread only ever needs the sums of the estimating equations, so an
# equation whose sums have a closed form can supply them directly
summed <- function(theta) {
  c(
    sum(y) - length(y) * theta[1],
    sum((y - theta[1])^2) - length(y) * theta[2]
  )
}
compute_sandwich(psi, theta = theta, summed_equations = summed) / length(y)
```

---

confidence_bands *Confidence bands for parameter vectors*

Description

Computes simultaneous confidence bands that provide coverage for the entire parameter vector, adjusting for multiple comparisons. The formula is:

$$\hat{\theta} \pm \hat{c}_{\alpha/2} \times \widehat{SE}(\hat{\theta})$$

where $\hat{c}$ is the adjusted critical value.

Usage

```
confidence_bands(
  object,
  alpha = 0.05,
  method = "supt",
  n_draws = 100000L,
  seed = NULL,
  subset = NULL,
  covariance = NULL,
  ...
)
```

Arguments

`object`	A fitted `MEstimator` object, or a numeric vector of parameter estimates.
`alpha`	Numeric significance level, between 0 and 1. Default `0.05`.
`method`	Character string. `"supt"` (default) for supremum-t or `"bonferroni"` for Bonferroni correction.
`n_draws`	Integer number of MVN draws for the sup-t method. Default `1e5`. The critical value is a Monte Carlo estimate of a fixed sup-t quantile, and at `1e5` draws the band half-width varies by roughly 0.2% across independent draws, which is negligible relative to the band width. Python's estimator method defaults to `1e6`; both defaults target the same quantity and agree to within Monte Carlo error, so `deli` keeps the smaller default for faster computation. Set `n_draws = 1e6` to match Python's estimator default. Note that seeded band values are not reproducible across the two languages because the MVN samplers differ.
`seed`	Integer seed for reproducibility. Default `NULL`.
`subset`	Integer vector of parameter indices to compute bands for. Default `NULL` (all parameters).
`covariance`	Numeric covariance matrix (only when `object` is numeric).
`...`	Not used. Must be empty, so a name that is not one of the documented arguments is an error rather than silently ignored.

Value

A p-by-2 matrix with columns `"lower"` and `"upper"`. For a fitted estimator the rows are named for the parameters, as in `confint()`. For a numeric `object` they take their names from it, when it has any.

Examples

```
# Two independent samples, each contributing one mean parameter
set.seed(42)
n <- 200
y1 <- rnorm(n, 2)
y2 <- rnorm(n, 3)

psi <- function(theta) {
  rbind(y1 - theta[1], y2 - theta[2])
}

m <- m_estimate(stacked_equations = psi, init = c(0, 0))

# Simultaneous bands cover both means at once, so they are wider than the
# pointwise intervals from confint(m)
confidence_bands(m, method = "supt", seed = 1)
```

`confidence_intervals` *Confidence intervals for M-Estimator parameters*

Description

Computes two-sided Wald-type $(1-\alpha)\times 100\%$ confidence intervals using the point estimates and sandwich variance: $\hat{\theta} \pm c_{\alpha/2} \times \widehat{SE}(\hat{\theta})$. The critical value $c_{\alpha/2}$ comes from the standard normal distribution by default. When a `finite_correction` is set on the fit, it comes instead from the t-distribution with $n-p$ degrees of freedom, matching the finite-sample adjustment of the variance.

This function mirrors `m.confidence_intervals()` in Python delicatessen, so code translated from Python can keep its shape.

Usage

```
confidence_intervals(object, alpha = 0.05, ...)
```

Arguments

`object`	A fitted `MEstimator` object (after calling `estimate()`).
`alpha`	Numeric significance level, between 0 and 1. Default `0.05` for 95% confidence intervals.
`...`	Not used. Must be empty, so a name that is not one of the documented arguments is an error rather than silently ignored.

Value

A p-by-2 matrix with columns `"lower"` and `"upper"`.

See Also

confint(), the standard R accessor for the same intervals. It returns identical values but is parameterized by level = 0.95 where this function takes alpha = 0.05.

Examples

```
psi <- function(theta) {
  y <- c(1, 2, 3, 4, 5)
  matrix(y - theta[1], nrow = 1)
}
m <- m_estimate(stacked_equations = psi, init = 0)
confidence_intervals(m)
```

---

convert_survival_measures

Convert between survival analysis measures

Description

Converts a survival probability (and optionally a hazard) to other survival analysis metrics.

Usage

```
convert_survival_measures(survival, hazard = NULL, measure)
```

Arguments

survival	Numeric survival probability or vector of probabilities.
hazard	Numeric hazard value or vector. Required for "hazard" and "density" measures.
measure	Character string specifying the desired measure. One of: "survival", "risk", "cumulative_hazard", "hazard", or "density".

Value

Numeric value or vector of the requested measure.

Examples

```
# Risk is the complement of survival
convert_survival_measures(0.8, measure = "risk")

# The density is the product of the hazard and the survival, so the
# "density" and "hazard" measures need the hazard as well.
convert_survival_measures(
  c(0.9, 0.8, 0.6),
  hazard = c(0.05, 0.06, 0.08),
  measure = "density"
```

```
)
```

`crime` *US state crime data*

Description

Data on violent crime rates by US state from the 2005 Statistical Abstract of the United States, used to demonstrate robust regression.

Usage

```
crime
```

Format

A data frame with 51 rows (50 states + DC).

References

Agresti A & Finlay B. (2009). *Statistical Methods for the Social Sciences*, 4th Edition. Pearson.

`cutler1995` *Cutler (1995) pharmacodynamic data*

Description

Pharmacodynamic data from Cutler et al. (1995) on acetylcholinesterase inhibition, used to demonstrate E-max dose-response models.

Usage

```
cutler1995
```

Format

A data frame.

References

Cutler NR et al. (1995). Dose-dependent CSF acetylcholinesterase inhibition by SDZ ENA 713 in Alzheimer's disease. *Acta Neurologica Scandinavica*, 91(5), 347-351.

deli-augment *Augment data with predictions from a fitted deli estimator*

Description

An `augment()` method for `MEstimator` and `GMMEstimator` objects fitted through the formula interface. It returns the model frame the fit was built from, or `newdata` when supplied, with the fitted values, their standard errors, a Wald confidence interval, and the residuals as columns beside it.

Usage

```
## S3 method for class '`deli::deli_estimator`'
augment(
  x,
  newdata = NULL,
  type.predict = c("link", "response"),
  conf.level = 0.95,
  ...
)
```

Arguments

`x`	A fitted `MEstimator` or `GMMEstimator` object made with the formula interface (after calling `estimate()`).
`newdata`	A data frame of covariate values to predict at, or `NULL` (default) to report the model frame the fit was built from. That frame holds the variables the formula named, so a transformed term appears as the column the transformation produced and a column of the fitting data the formula did not name is not there.
`type.predict`	Character string. `"link"` (default) puts `.fitted` and its interval on the scale of the linear predictor; `"response"` puts them on the scale of the response.
`conf.level`	Numeric confidence level for `.lower` and `.upper`. Default `0.95`.
`...`	Not used. Must be empty, so that a name that is not one of the documented arguments is an error rather than silently ignored. A misspelled `newdata` would otherwise augment the fitted rows while the caller believed they had asked for rows of their own.

Details

The columns added are `.fitted`, `.se.fit`, `.lower`, `.upper`, and, when `newdata` is not supplied, `.resid`. They are exactly what `predict()` and `residuals()` return for the same fit, so `.fitted` is `predict()`, `.lower` and `.upper` are `predict(interval = "confidence")`, and `.resid` is `residuals()`.

`.fitted` is on the link scale by default, matching both `predict()` and `broom::augment()` on a `glm`, and `type.predict = "response"` puts it and its interval on the scale of the response. `.resid` is the response residual either way, since a residual measured against a linear predictor would be a response minus a quantity the response is not measured in.

Rows the fit dropped for missing data are not reported, so the result has one row per `nobs()` and its row names are those of the retained rows. A `newdata` row with a missing value is kept, with `NA` in the added columns, so that the result lines up with the rows handed in.

`augment()` covers the estimating equations whose linear predictor `predict()` forms, and refuses the same fits with the same reasons; see deli-predict. It has no counterpart to that method's `times` argument: a survival measure is one value per row of the data and time rather than one per row, so the predictions do not go beside the data as columns.

Value

A data frame: the model frame, or `newdata`, followed by the columns `.fitted`, `.se.fit`, `.lower`, `.upper`, and `.resid`. The last is absent when `newdata` is supplied.

See Also

deli-predict for the predictions themselves and the equations they are available for, deli-tidiers for the parameter-level and model-level summaries, and reexports for the generic itself, which deli re-exports so that `augment()` resolves with deli alone attached.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                  model = "linear")

head(augment(fit))

# New covariate patterns come back with no residual column, since they carry
# no response to residualize against.
augment(fit, newdata = data.frame(wt = c(2, 3, 4), hp = 110))
```

`deli-conditions` *The condition classes deli raises*

Description

Where a caller might reasonably want to answer an error or a warning rather than let it stop the work, deli attaches a condition class to it, so that `base::tryCatch()`, `base::withCallingHandlers()`, or `rlang::try_fetch()` can match on the class rather than on the wording of a message. This page lists those classes and says what each of them reports on.

Details

Every class below belongs to an error unless the entry says otherwise. A condition carrying two of them leads with the narrower one, so a handler for either the family or the particular fault catches it.

A condition a user's own estimating function raises reaches the caller with whatever class that function gave it. Nothing here renames one.

What an estimating function returned

- `deli_psi_return_error`: the estimating function returned something no sandwich can be built from, either at the starting values or at the point handed to `compute_sandwich()`. Every refusal of a return carries it, so recognizing one is a match on this rather than on four messages.
- `deli_psi_shape_error`: the number of estimating equations returned cannot be solved against the number of parameters. Leads `deli_psi_return_error`.
- `deli_psi_list_unsupported`: the estimating function returned a list of one element per equation. `compute_bread()` and `compute_sandwich()` take that shape, and no fit can solve it, so `estimate()` refuses it rather than failing on the summed equations. Leads `deli_psi_return_error`.

The estimating equations deli supplies

- `deli_mlogit_binary_outcome`: the indicator matrix handed to `ee_mlogit()` holds two columns. Two categories leave a single non-reference residual whose reference residual is its exact negative, so the equations cancel to zero at every value of the parameters and a fit of them reports the values it started from with no variance. A two-level outcome is logistic regression, which `ee_glm()` fits with `distribution = "binomial"`.

The formula interface

- `deli_formula_ee_lookup_error`: `.ee` is neither a function nor the name of one. Either it arrived as some other type entirely, or it arrived as a character vector naming no one function, because no function of that name was found or because the vector is not of length one. A handler that means to read the name it failed to resolve should confirm it has a character vector first.
- `deli_formula_ee_signature_error`: the arguments of `.ee` leave something the interface fills nowhere to go: `theta`, the model matrix it passes as `X`, the response it passes by position, or the offset an `offset()` term in the formula supplies. It also covers an equation that writes formals after its `...`, which can be filled by name and by nothing else, so the positional response would fall into the dots and leave them unfilled.
- `deli_formula_ee_argument_error`: something in the `...` the caller wrote cannot be forwarded to `.ee`. On its own it is a name matching no argument of `.ee` exactly, which is what keeps an abbreviation from reaching an argument the caller did not mean.
- `deli_formula_ee_unnamed_argument`: an argument was forwarded with no name, and so by position, into whichever argument the response did not take. Leads `deli_formula_ee_argument_error`.
- `deli_formula_ee_reserved_argument`: a name the caller supplied is one the interface fills itself, either `theta`, `X`, or the argument the response is passed to. Leads `deli_formula_ee_argument_error`.
- `deli_formula_auto_init_error`: the `init` the formula interface generated itself, one zero per model-matrix column, does not fit the estimating equation.

Solving

- `deli_solver_not_converged`, a warning: the solver stopped without solving the estimating equations, either because it reported a failure of its own or because the point it returned does not solve them.

- deli_nested_solver_error: a rootSolve::multiroot() solve was asked for while another one was running, which it cannot be. An error rather than a warning, because the solve cannot be attempted at all.
- deli_gmm_moments_rejected, a warning: the Hansen J-statistic of an over-identified GMM fit stands so far out against its reference distribution as to all but rule the fit out, which usually means the moment conditions cannot all hold at one value of the parameters.
- deli_gmm_moments_dependent, a warning: the moment conditions are linearly dependent at the estimated values, so the two-step weight matrix is a pseudo-inverse and what comes back is the fit of the independent conditions alone. It also carries the rarer covariance that has no inverse for want of conditioning rather than of rank, which the message tells apart.
- deli_meat_not_invertible: the covariance of the moment conditions cannot be inverted, so the two-step GMM weight update has no weight matrix to take, and allow_pinv = FALSE refused the pseudo-inverse. The counterpart of deli_bread_not_invertible for the other half of the sandwich.

The sandwich variance

- deli_bread_na, a warning: the bread matrix holds NA, so no inverse of it exists. A fit records no variance and carries on; compute_sandwich() has nothing but a matrix to return and converts it into the error below.
- deli_bread_not_invertible: the bread cannot be inverted and allow_pinv = FALSE refused the pseudo-inverse, or it is rectangular and has no inverse at any rank, or it holds NA.
- deli_summed_equations_error: the summed_equations handed to compute_sandwich() or compute_bread(), or carried as a property of MEstimator() or GMMEstimator(), is not one the bread can be differentiated from. It is neither NULL nor a function, or its return at the point in hand is not numeric, or that return does not hold one value per estimating equation. Every refusal of the argument carries it, as does a check_summed_equations that is not a single TRUE or FALSE.
- deli_summed_equations_disagree: the reduction is a function of the right shape and does not sum the stacked_equations it is paired with, either by differing from their row sums or by returning a value that is not a number where they sum to one. The bread is differentiated from the one and the meat is built from the other, so a sandwich assembled from two systems would carry the shape of a covariance matrix without the meaning, and a fit would report the estimates of the other system as well. Leads deli_summed_equations_error.

Differentiation

- deli_exact_unsupported_function: under deriv_method = "exact", a function reached with a tangent-carrying argument has no rule, either because it hands its argument to compiled code without dispatching or because deli declines to differentiate it. auto_differentiation() names the replacements to use instead.
- deli_exact_unsupported_shape: the estimating equations arrived in a container the summing step has no rule for, either because the tangents survived in a shape it cannot reduce or because a per-equation list does not hold one element per parameter, each of one length. Nothing was lost; the shape is the problem.

- `deli_exact_tangent_lost`: derivative information is gone or would be, either because a tangent-carrying value was asked to become a plain double or because a result arrived with no tangent and evidence that one had been dropped on the way.
- `deli_finite_difference_lost`, a warning: a finite-difference step changed the function by less than the floating-point spacing of its values, so an entry of the Jacobian carries none of the digits of the values it came from.

Parameter names

- `deli_param_name_collision`: filling the unnamed entries of a partly named parameter vector with positional `theta_1`, `theta_2`, ... labels would repeat a label another parameter in the same vector already carries.

`deli-display` *Display methods for deli estimators*

Description

Methods for `base::print()` and `base::summary()` so that a fitted estimator shows its coefficients at the console and reports its parameter-level inference as a table.

Usage

```
## S3 method for class '`deli::deli_estimator`'
print(x, ..., subset = NULL)

## S3 method for class '`deli::deli_estimator`'
summary(object, alpha = 0.05, subset = NULL, ...)
```

Arguments

`x`, `object`	A fitted `MEstimator` or `GMMEstimator` object. Named `x` for `print()` and `object` for `summary()`, because `base::print()` and `base::summary()` name their first argument that.
`...`	Not used. Must be empty, so that a name neither method recognizes is an error rather than silently ignored: a misspelled `alpha` would report limits at the default width and a misspelled `subset` would display every parameter, each while the call still read as the one that was meant.
`subset`	Integer vector of parameter indices to display, or `NULL` (default) to display all of them.
`alpha`	Numeric significance level for the confidence limits reported by `summary()`, between 0 and 1. Default `0.05` for 95% limits.

Details

`print()` reports the parameter and observation counts followed by the estimates, rounded to four decimal places. An estimator that has not been through `estimate()` has no estimates to show, so it reports its parameter count and says so.

`summary()` collects the estimates, their standard errors, Z-scores, confidence limits, P-values, and S-values into one object, which prints as a table with one row per parameter. The values are those `confint()`, `z_scores()`, `p_values()`, and `s_values()` return individually, so `alpha` here means what it means there: `0.05` gives 95% limits.

An over-identified `GMMEstimator` fit reports Hansen's J-statistic above the table, with its degrees of freedom and its P-value, because it judges the fit as a whole rather than any one parameter. See `GMMEstimator()` for what it means and where its reference distribution holds. No other fit has one, so no other output carries the line.

The `S` column reads `Inf` when a P-value underflows to exactly zero, for the reason `s_values()` gives. The `P` column reports that same underflow as `<2e-16`: `base::format.pval()` stops printing digits below the `eps` it is given, and the table gives it `2.2e-16`. `tidy()` returns the literal `0` instead.

`subset` restricts which parameters are displayed and nothing else. The reported parameter count and every reported value are computed from the whole fit, so displaying a subset of a stacked estimator is a way to read the parameters of interest without the nuisance parameters, not a way to refit without them. Row labels keep the names the full fit gave them.

Value

- `print()`: its input, invisibly.
- `summary()`: an object holding the estimates, standard errors, Z-scores, confidence limits, P-values, and S-values, which prints as a table.

See Also

deli-generics for the accessors that return these quantities as plain vectors and matrices, and deli-tidiers for the same results as a data frame.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                  model = "linear")

fit

# `subset` is an argument of the method rather than of the fit, so showing it
# here means calling the generic by name.
print(fit, subset = 2:3)

summary(fit)

# `summary()` takes `subset` as well, alongside `alpha` for the width of the
# reported limits.
summary(fit, subset = 2:3, alpha = 0.1)
```

deli-generics *Standard S3 generics for deli estimators*

Description

Methods for base R generics `stats::coef()`, `stats::vcov()`, `stats::confint()`, `stats::nobs()`, `stats::df.residual()`, `stats::fitted()`, `stats::residuals()`, `stats::weights()`, `stats::model.frame()`, `stats::model.matrix()`, `stats::formula()`, `stats::terms()`, `stats::sigma()`, `stats::logLik()`, and `stats::deviance()` so that deli estimator objects interoperate with the broader R modeling ecosystem.

Usage

```
## S3 method for class '`deli::deli_estimator`'
coef(object, ...)

## S3 method for class '`deli::deli_estimator`'
vcov(object, ...)

## S3 method for class '`deli::deli_estimator`'
confint(object, parm, level = 0.95, ...)

## S3 method for class '`deli::deli_estimator`'
nobs(object, ...)

## S3 method for class '`deli::deli_estimator`'
df.residual(object, ...)

## S3 method for class '`deli::deli_estimator`'
fitted(object, ...)

## S3 method for class '`deli::deli_estimator`'
residuals(object, type = "response", ...)

## S3 method for class '`deli::deli_estimator`'
weights(object, ...)

## S3 method for class '`deli::deli_estimator`'
model.frame(
  formula,
  data = NULL,
  subset = NULL,
  na.action,
  drop.unused.levels = FALSE,
  xlev = NULL,
  ...
)
```

```
## S3 method for class '`deli::deli_estimator`'
model.matrix(object, data = NULL, ...)

## S3 method for class '`deli::deli_estimator`'
formula(x, ...)

## S3 method for class '`deli::deli_estimator`'
terms(x, ...)

## S3 method for class '`deli::deli_estimator`'
sigma(object, ...)

## S3 method for class '`deli::deli_estimator`'
logLik(object, ...)

## S3 method for class '`deli::deli_estimator`'
deviance(object, ...)
```

Arguments

`object`	A fitted `MEstimator` or `GMMEstimator` object (after calling `estimate()`).
`...`	Not used. `confint()`, `residuals()`, `model.frame()`, and `model.matrix()` require them to be empty, so that a name none of them recognizes is an error rather than silently ignored: a misspelled `level` would report the default limits, a misspelled `type` would report response residuals as though they had been asked for, and a misspelled `data` would report on the fitted rows while the caller believed they had asked for rows of their own. The rest ignore them, which is the convention for a base generic with no optional argument for a wrong name to displace.
`parm`	A specification of which parameters are to be given confidence intervals, either a vector of numbers or a vector of names. If missing, all parameters are considered.
`level`	The confidence level required. Default `0.95`.
`type`	Character string naming the residual `residuals()` returns. Only `"response"` is available, since it is the one residual defined for every estimating equation `predict()` supports. Any other value is an error rather than a response residual under another name.
`formula`	A fitted `MEstimator` or `GMMEstimator` object. Named `formula` because `stats::model.frame()` names its first argument that; the method takes a fit, not a formula.
`data`	A data frame to build the model frame or the design matrix of, or `NULL` (default) to report the one the fit was solved on. `model.frame()` needs it to carry the response as well as the predictors; `model.matrix()` names no response, so the predictors alone are enough there.
`subset`, `na.action`, `drop.unused.levels`, `xlev`	Passed to `stats::model.frame()`, and meaning there what they mean for any other model frame. `xlev` defaults to the factor levels the fit recorded. All four

describe how to build a frame from `data`, so supplying one without `data` is an error rather than a silent no-op.

`x` A fitted `MEstimator` or `GMMEstimator` object. Named `x` because `stats::formula()` and `stats::terms()` name their first argument that.

Details

`fitted()`, `residuals()`, `weights()`, `model.frame()`, `model.matrix()`, `formula()`, and `terms()` report on the model a fit was specified as, which only the formula interface records, so each of them is an error for a fit built from a `stacked_equations` function. `fitted()` and `residuals()` are also an error for a formula fit of an estimating equation `predict()` does not support, since a fitted value is a prediction; see deli-predict for the equations it covers.

`fitted()` is the conditional mean of the response, so it is on the response scale rather than the link scale, matching `stats::fitted()` on a `glm` object. `residuals()` is the response minus that mean, the residual a GLM calls a response residual, and the only `type` it takes. Deviance and Pearson residuals are not offered: the first needs a likelihood and the second a variance function, and an M-estimator need not state either, while the response residual is defined the same way for every equation `predict()` supports. Under a non-identity link it is heteroscedastic by construction.

`model.frame()` returns the frame the fit was solved on, or builds the frame of `data` when one is supplied, through the terms and factor levels the fit recorded. A data-dependent term such as `poly(x, 2)` is therefore evaluated at its fitted basis and a factor gets its fitted levels, as they are under `predict()`. `data` has to carry the response, since a model frame holds every variable the formula names; covariate values that carry no response are what `predict()` and deli-augment take a `newdata` for.

`model.matrix()` returns the design the fit was solved on, or the design of `data` when one is supplied, coded with the contrasts and factor levels the fit recorded rather than with whatever `getOption("contrasts")` says when the call is made. A design matrix is a property of the fit, so a fit made under one setting of that option answers with the coding it was solved on under any other, which is what `predict()` does as well. A design names no response, so `data` needs only the predictors.

`df.residual()` is the number of observations less the number of parameters. Both counts belong to the fit rather than to a specification, so it answers for a fit built from a `stacked_equations` function as readily as for a formula one. `weights()` returns the observation weights the fit was solved with, which reach a fit only through the formula interface. A formula fit specified without weights returns `NULL`, as `stats::weights()` does for an unweighted `stats::lm()` fit: a vector of ones would be indistinguishable from a fit weighted by ones.

`sigma()`, `logLik()`, and `deviance()` are errors for every fit, and say why. An M-estimator is defined by its estimating equations, which need come from no likelihood at all, so there is no log-likelihood to return and no deviance to take from one; `stats::AIC()` and `stats::BIC()` reach a fit through `logLik()` and are refused with it. A residual standard deviation belongs to the model an equation states rather than to the solve: a linear equation has one, a logistic equation has none, and nothing in the estimates says which was solved. Reporting an error is the only answer that does not invent a quantity the fit never had.

Value

- `coef()`: Named numeric vector of parameter estimates.

- `vcov()`: Named variance-covariance matrix.
- `confint()`: Matrix with columns `"lower"` and `"upper"`.
- `nobs()`: Integer number of observations.
- `df.residual()`: Integer residual degrees of freedom, the number of observations less the number of parameters.
- `fitted()`: Named numeric vector of fitted values on the response scale, one per observation the fit was solved on.
- `residuals()`: Named numeric vector of response residuals.
- `weights()`: The observation weights the fit was solved with, as they were recorded, or `NULL` for a formula fit specified without any. Weights that vary over time are reported as the matrix they were supplied as rather than as a vector: `ee_plogit()` takes an n-by-K matrix carrying one column per time interval, and an observation weighted differently in each interval has no one weight for a vector to hold.
- `model.frame()`: The model frame the fit was built from, with the rows dropped for missing data already removed, or the model frame of `data` when one is supplied.
- `model.matrix()`: The design matrix the fit was solved on, or the design matrix of `data` when one is supplied, coded with the contrasts and factor levels the fit recorded.
- `formula()`: The model formula.
- `terms()`: The `terms` object of the model frame, carrying the response index, any offset, and the `predvars` of a data-dependent term.
- `sigma()`, `logLik()`, `deviance()`: Nothing. Each raises an error saying that an M-estimator states no likelihood and records no residual scale.

See Also

deli-predict for predictions at new covariate values, and deli-augment for the fitted values, intervals, and residuals as columns beside the data.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                  model = "linear")

coef(fit)

vcov(fit)

confint(fit)

nobs(fit)

df.residual(fit)

head(fitted(fit))

head(residuals(fit))
```

```
formula(fit)

# Weights reach a fit through the formula interface, and `weights()` reports
# the vector the fit was solved with.
weighted <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                       model = "linear", weights = 1 / hp)

head(weights(weighted))
```

---

deli-predict *Predictions from a fitted deli estimator*

Description

A method for stats::predict() that predicts from a fit made through the formula interface, with sandwich standard errors and Wald confidence intervals. It forms the linear predictor of a regression fit on either the link or the response scale, and evaluates a survival measure at a set of `times` for a fit of ee_aft() or ee_plogit().

Usage

```
## S3 method for class '`deli::deli_estimator`'
predict(
  object,
  newdata = NULL,
  type = c("link", "response"),
  se.fit = FALSE,
  interval = c("none", "confidence"),
  level = 0.95,
  times = NULL,
  measure = "survival",
  deriv_method = "capprox",
  dx = 1e-09,
  ...
)
```

Arguments

`object`	A fitted `MEstimator` or `GMMEstimator` object made with the formula interface (after calling estimate()).
`newdata`	A data frame of covariate values to predict at, or `NULL` (default) to predict the rows the fit was made on. An offset written into the formula with `offset()` is evaluated on `newdata`; an offset supplied through `...` at fit time is one value per fitted observation, so predicting on `newdata` is an error rather than a prediction at an offset of zero.

type	Character string. "link" (default) returns the linear predictor; "response" applies the inverse link, giving the conditional mean of the response. Cannot be supplied beside times.
se.fit	Logical. Return standard errors beside the predictions? Default FALSE.
interval	Character string. "none" (default) or "confidence" for Wald intervals at level.
level	The confidence level for interval. Default 0.95. The critical value comes from the standard normal distribution, or from the t-distribution with $n - p$ degrees of freedom when a finite_correction is set on the fit, matching confint().
times	Numeric vector of times to predict a survival measure at, or NULL (default) to predict the linear predictor instead. Supported for a fit of ee_aft() or ee_plogit().
measure	Character string naming the survival measure, one of "survival" (default), "risk", "cumulative_hazard", "hazard", or "density"; see convert_survival_measures(). Only meaningful beside times.
deriv_method	Character string for the derivative method used to build the Jacobian of the survival measure. One of "capprox" (central difference, the default), "fapprox", "bapprox", or "exact" (forward-mode automatic differentiation, available for an AFT fit and an error for a pooled logistic one). Only meaningful beside times.
dx	Numeric step size for the finite-difference methods, ignored when deriv_method = "exact". Default 1e-9. Only meaningful beside times. Must be a single positive finite number, which is checked wherever it is supplied beside times, including a prediction asking for no standard error, which takes no step at all.
...	Not used. Must be empty, so a name that is not one of the documented arguments is an error rather than silently ignored.

Value

Without times, a named numeric vector of predictions, one per row of newdata or of the fitted design, and with interval = "confidence" a matrix with columns "fit", "lwr", and "upr".

With times, a data frame with one row per row of the design and time, every time for the first row before any time for the second, and columns .row, the row label of the design, time, and fit. With interval = "confidence" it also has "lwr" and "upr".

With se.fit = TRUE, either shape becomes a list whose fit element is whichever of the two the other arguments call for and whose se.fit element is a numeric vector of standard errors.

Two surfaces

times chooses what is predicted. Without it, predict() returns the linear predictor on the scale type names. With it, predict() returns the survival measure measure names at each of the times, one prediction per row of the design at each time.

The two cover disjoint sets of fits. An equation reaches the first surface by having a linear predictor that is a conditional mean of the response, and the two equations on the second surface are there because theirs is not: ee_aft() puts its linear predictor on the log-time scale, and ee_plogit() has one linear predictor per person and time interval rather than one per person. Supplying type beside times, or measure, deriv_method, or dx without times, is an error rather than an argument silently ignored, since no fit takes both sets.

The linear predictor

The design for `newdata` is rebuilt through the terms, factor levels, and contrasts the fit recorded, so a factor whose new values cover only some of the fitted levels still produces the fitted set of columns, and a data-dependent term such as `poly(x, 2)` or `scale(x)` is evaluated with the coefficients it was fitted with rather than refitted to `newdata`. A factor level the fit never saw, or a predictor `newdata` does not carry, is an error.

The standard error is the delta-method standard error of the linear predictor, $\sqrt{\mathrm{diag}(X\hat{V}X^T)}$, formed from the coefficient block of the sandwich variance. On the response scale it is scaled by the derivative of the inverse link, which is exact because the inverse link is applied elementwise. The intervals are Wald intervals on the scale asked for, so they are symmetric about `fit` on that scale rather than transformed from the link scale.

`predict()` needs to know which parameters are coefficients on the design and what takes the linear predictor to the mean of the response. It supports `ee_regression()`, `ee_glm()`, `ee_robust_regression()`, `ee_beta_regression()`, and the five penalized regressions `ee_bridge_regression()`, `ee_ridge_regression()`, `ee_lasso_regression()`, `ee_dlasso_regression()`, and `ee_elasticnet_regression()`, whose parameters are one coefficient per design column followed by at most one parameter of the outcome distribution. Any other estimating equation, and any fit built from a `stacked_equations` function, is an error naming the reason; `regression_predictions()` takes a design, estimates, and a covariance matrix directly and can be used wherever this method declines.

Survival measures at a set of times

`times` is supported for a fit of `ee_aft()` or of `ee_plogit()`, and `measure` names any of the measures `convert_survival_measures()` defines. The point estimates are those of `aft_predictions_individual()` and `plogit_predict()` for the same fit.

`ee_survival_model()` has no surface here because the formula interface cannot drive it: it takes no design matrix, and the interface always passes the one it built. Predict from such a fit with `survival_predictions()`.

Each covariate pattern has its own variance, so each row gets its own interval. The variance of the measure for a row at a time is the delta-method variance $G\hat{V}G^T$ of that one prediction, where G is its row of the Jacobian of the whole grid with respect to the parameters, built by `deriv_method`. A pooled logistic fit is predicted through `plogit_predict()`, whose matrix products cannot carry tangents, so `deriv_method = "exact"` is available for an AFT fit alone. Asking for it on a pooled logistic fit is an error whether or not a standard error is wanted, since the Jacobian that could not be built is only built when one is.

See Also

deli-augment, which returns the linear-predictor predictions as columns beside the data; `regression_predictions()`, which computes the same quantities from a design matrix, a vector of estimates, and a covariance matrix, for fits this method does not cover; and `aft_predictions_individual()`, `aft_predictions_function()`, `plogit_predict()`, and `survival_predictions()`, which compute the survival measures from estimates directly.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
```

```
                  model = "linear")

head(predict(fit))

# New covariate patterns, with sandwich standard errors.
at <- data.frame(wt = c(2, 3, 4), hp = 110)

predict(fit, newdata = at, se.fit = TRUE)

predict(fit, newdata = at, interval = "confidence")

# A logistic fit predicts log-odds by default and probabilities on the
# response scale.
vs_fit <- m_estimate(vs ~ wt, data = mtcars, .ee = ee_regression,
                     model = "logistic")

predict(vs_fit, newdata = data.frame(wt = c(2, 3)), type = "response")

# A pooled logistic fit has no linear predictor to put on a scale, and
# predicts a survival measure at a set of times instead. Every row of the
# design gets its own interval.
bladder <- collett_bladder
bladder$novel <- bladder$treat - 1
k <- length(unique(bladder$time[bladder$delta == 1]))

plogit_fit <- m_estimate(
  time ~ novel + init + size - 1, data = bladder, .ee = ee_plogit,
  event = delta, init = c(rep(0, 3), -4, rep(0, k - 1))
)

head(predict(plogit_fit, times = c(12, 24), interval = "confidence"))
```

`deli-tidiers` *Broom tidiers for deli estimators*

Description

`tidy()` and `glance()` methods for `MEstimator` and `GMMEstimator` objects. These allow deli results to flow into tidyverse pipelines.

Usage

```
## S3 method for class '`deli::deli_estimator`'
tidy(x, conf.int = FALSE, conf.level = 0.95, ...)

## S3 method for class '`deli::deli_estimator`'
glance(x, ...)
```

Arguments

x	A fitted MEstimator or GMMEstimator object.
conf.int	Logical. Include confidence intervals? Default FALSE.
conf.level	Numeric confidence level for intervals. Default 0.95.
...	Not used. tidy() requires them to be empty, so that a misspelled conf.int or conf.level is an error rather than a table silently returned without intervals or at the default level. glance() has no optional argument for a wrong name to displace and ignores them.

Value

- tidy(): A data.frame with columns term, estimate, std.error, statistic, p.value, s.value. If conf.int = TRUE, also includes conf.low and conf.high. A p.value that underflows to exactly zero is reported as 0 alongside an infinite s.value; see s_values().
- glance(): A single-row data.frame with model-level summaries: nobs, npar, estimator, finite_correction, and the Hansen J-statistic of an over-identified GMM fit in j_statistic, j_df and j_p_value. The three J columns are present on every fit and hold the typed missing value of their own type where there is no such statistic, which is every M-estimation fit and every just-identified or subset GMM fit; see GMMEstimator() for what the statistic reads and where it is left unset.

See Also

deli-augment, the third broom generic, which returns the observation-level fitted values, intervals, and residuals, and reexports for the generics themselves, which deli re-exports so that tidy() and glance() resolve with deli alone attached.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                  model = "linear")

tidy(fit, conf.int = TRUE)

glance(fit)
```

---

deli_digamma *Digamma function*

Description

Evaluates the digamma function, the first derivative of the log gamma function.

This is the equivalent of base::digamma(), and unlike most of the deli utilities it is not required anywhere: digamma() belongs to the Math group generic and has a tangent rule, so it differentiates under deriv_method = "exact" just as deli_digamma() does. The two agree value for

value, including at the poles and on missing input. The only difference is that `deli_digamma()` returns `NaN` at the non-positive integers, where the digamma function has poles, without base R's `NaNs produced` warning. `deli_digamma()` exists so that code translated from Python delicatessen, where the function is called `digamma()`, can keep its shape.

Usage

```
deli_digamma(z)
```

Arguments

z	Numeric value or vector.

Value

Numeric digamma values.

See Also

`base::digamma()`, which is usable in the same positions, and `deli_polygamma()` for the higher-order derivatives. `inverse_logit()` has the full list of deli utilities and their base R counterparts, including the five that base R cannot stand in for under exact differentiation.

Examples

```
deli_digamma(1)
deli_digamma(c(0.5, 1, 2))

# The same values as the base R counterpart
all.equal(deli_digamma(c(0.5, 1, 2)), digamma(c(0.5, 1, 2)))

# NaN at the poles, where base R warns, and missing values pass through
deli_digamma(c(-1, 0, 2.5, NA))
```

---

deli_polygamma *Polygamma function*

Description

Evaluates the nth derivative of the digamma function.

This is the equivalent of `base::psigamma()` and returns identical values for numeric input, except that it carries derivatives. Exact differentiation (`deriv_method = "exact"`) propagates a tangent alongside each value, and `deli_polygamma()` recognizes a tangent-carrying argument and applies the analytic rule itself, the polygamma function of order `n + 1`. `psigamma()` hands its argument to compiled code without dispatching, and errors on such an argument. Use `deli_polygamma()` inside estimating equations and inside transforms passed to `delta_method()`, and `psigamma()` for ordinary numeric work.

The two take their arguments in opposite orders. `deli_polygamma(n, x)` takes the derivative order first, matching Python's `scipy.special.polygamma(n, x)`; `psigamma(x, deriv = n)` takes it second. `deli_polygamma(1, 3)` is `psigamma(3, deriv = 1)`, not `psigamma(1, deriv = 3)`. A positional substitution between the two computes a different quantity and raises no error.

Usage

```
deli_polygamma(n, x)
```

Arguments

`n`	Integer order of the derivative of the digamma function.
`x`	Numeric value or vector.

Value

Numeric polygamma values.

Exact differentiation

`deriv_method = "exact"` is forward-mode automatic differentiation: it replaces each value with an object carrying both the value and its derivative. deli supports those objects through S3 methods: the `Ops`, `Math`, and `Summary` group generics, plus non-group methods such as `[`, `%*%`, `t()`, `c()`, and `mean()`. `standard_normal_cdf()`, `standard_normal_pdf()`, `deli_polygamma()`, and `deli_digamma()` recognize a tangent-carrying argument themselves and apply their own analytic rule. Support within the group generics is partial; see `vignette("getting-started")` for the operations deli differentiates.

`plogis()`, `qlogis()`, `pnorm()`, `dnorm()`, and `psigamma()` take none of these paths. Each hands its argument straight to compiled code through `.Call()` or `.Internal()` without dispatching, so the tangent reaches C code that requires a plain number. deli catches the resulting failure and raises its own error, naming the function that stopped the computation and the deli function to write in its place. The same applies to every other distribution function in `stats`, such as `qnorm()`, which is named in the error even though deli exports no counterpart for it.

Each deli utility below returns the same values as its base R counterpart for numeric input. What separates them is whether the counterpart survives exact mode:

deli function	base R counterpart	base R under `deriv_method = "exact"`
`inverse_logit()`	`stats::plogis()`	errors
`logit()`	`stats::qlogis()`	errors
`standard_normal_cdf()`	`stats::pnorm()`	errors
`standard_normal_pdf()`	`stats::dnorm()`	errors
`deli_polygamma()`	`base::psigamma()`	errors
`deli_digamma()`	`base::digamma()`	works
`identity_transform()`	`base::identity()`	works

For the first five rows, use the deli function inside estimating equations and inside transforms passed to `delta_method()`, and the base R function everywhere else: simulating data, post-fit

display, plain numeric work. For the last two rows the base R function is usable everywhere, because `digamma()` is a `Math` group member with a tangent rule and `identity()` passes its argument through untouched.

The polygamma row is the one place where the arguments do not line up. `deli_polygamma(n, x)` takes the derivative order first and `psigamma(x, deriv = n)` takes it second, so a positional substitution between the two computes a different quantity and raises no error.

See Also

`base::psigamma()`, the base R equivalent for ordinary numeric work, which takes its arguments in the other order, and `deli_digamma()` for the order-zero case.

Examples

```
deli_polygamma(0, 1)
deli_polygamma(1, c(1, 2, 5))

# The same values as the base R counterpart, with the arguments swapped
all.equal(deli_polygamma(1, c(1, 2, 5)), psigamma(c(1, 2, 5), deriv = 1))

# A gamma GLM, whose last parameter is the log of the shape
m <- m_estimate(
  mpg ~ wt,
  data = mtcars,
  .ee = ee_glm,
  distribution = "gamma",
  link = "log",
  init = c(3, 0, 0)
)

# Variance of the trigamma function at the estimated shape. Writing
# `psigamma(exp(theta[3]), deriv = 1)` here instead would error, because
# exact differentiation hands the transform a tangent-carrying argument.
delta_method(
  m,
  transform = function(theta) deli_polygamma(1, exp(theta[3])),
  deriv_method = "exact"
)
```

`deli_spline` *Generate polynomial spline basis terms*

Description

Generates polynomial spline terms for a numeric vector at pre-specified knot locations. Default is restricted (natural) cubic splines, but unrestricted splines with different polynomial terms can also be generated.

This function mirrors `spline()` in Python delicatessen, so code translated from Python can keep its shape. Despite the name it has nothing to do with `stats::spline()`, which interpolates a curve through data points and returns the interpolated values. `deli_spline()` builds a truncated power basis, a matrix of design columns to be used as covariates in a regression. The `deli_` prefix is there so that the two names do not collide.

No base R function returns these truncated power columns, but the `splines` package, which installs with every copy of R, builds the same spline spaces in a B-spline parameterization. `splines::bs()` is the counterpart to `restricted = FALSE`: for knots `k`, `cbind(1, x, x^2, x^3, deli_spline(x, k, restricted = FALSE))` and `cbind(1, splines::bs(x, knots = k))` span the same space and give identical fitted values. `splines::ns()` is the nearest counterpart to `restricted = TRUE` but not an exact one, since it also constrains the fit to be linear beyond the boundary knots and so spans a subspace of the columns here. `deli_spline()` is what deli offers for parity with Python delicatessen, for the truncated power form itself, and because `additive_design_matrix()` builds on it.

Usage

```
deli_spline(x, knots, power = 3, restricted = TRUE, normalized = FALSE)
```

Arguments

`x`	Numeric vector of observed values.
`knots`	Numeric vector of knot locations. Should be between the min and max of `x`.
`power`	Numeric power for the spline terms. Default `3` (cubic).
`restricted`	Logical. If `TRUE` (default), generate restricted (natural) splines. If `FALSE`, generate unrestricted splines.
`normalized`	Logical. If `TRUE`, divide spline terms by the range of knots (largest minus smallest) raised to `power`. With a single knot the range is zero, so the divisor is that knot raised to `power` instead. Default `FALSE`.

Details

Unrestricted splines for knot k are:

$$s_k(X) = I(X > k)(X - k)^a$$

Restricted (natural) splines subtract the last spline term:

$$r_k(X) = s_k(X) - s_K(X)$$

where K is the largest knot. Restricted splines return one fewer column than the number of knots.

Value

A matrix with `length(x)` rows. Number of columns is `length(knots)` for unrestricted or `length(knots) - 1` for restricted.

Examples

```
# Restricted quadratic splines at four knots, so three basis columns
s <- deli_spline(1:59, knots = c(10, 20, 30, 40), power = 2)
dim(s)

# Each term stays at zero below its knot and grows above it
s[c(5, 15, 25, 35, 45, 55), ]
```

`delta_method` *Delta method for variance of transformed parameters*

Description

Computes the variance-covariance matrix for a transformation of parameters using the Delta Method:

$$Var[g(\theta)] \approx G\Sigma G^T$$

where G is the Jacobian of g and Σ is the covariance matrix of θ.

Usage

```
delta_method(
  object,
  transform,
  covariance = NULL,
  deriv_method = "capprox",
  dx = 1e-09,
  ...
)
```

Arguments

`object`	A fitted `MEstimator` object, or a numeric vector of parameter estimates.
`transform`	Function that takes `theta` and returns a numeric vector.
`covariance`	Numeric covariance matrix (only used when `object` is a numeric vector).
`deriv_method`	Character string for the derivative method used to build the Jacobian of `transform`. One of `"capprox"` (central difference), `"fapprox"` (forward difference), `"bapprox"` (backward difference), or `"exact"` (forward-mode automatic differentiation). Default `"capprox"`.
`dx`	Numeric step size for the finite-difference methods; ignored when `deriv_method = "exact"`. Default `1e-9`. Must be a single positive finite number, which is checked whichever `deriv_method` is in force. The step is absolute and is floored at the floating-point resolution of each estimate, so a large parameter magnitude cannot silently reduce it to nothing; see `approx_differentiation()`.

`...` Not used. Must be empty, so a name that is not one of the documented arguments is an error rather than silently ignored. Exact names matter here because `deriv_method` selects how the Jacobian is built, and a dropped misspelling would leave the default in place and return a different variance with nothing to signal the substitution.

Value

A covariance matrix for `transform(theta)`.

Examples

```
fit <- m_estimate(vs ~ mpg, data = mtcars, .ee = ee_regression,
                  model = "logistic")

# Variance of the odds ratio for mpg, exponentiating the log-odds coefficient
delta_method(fit, transform = function(theta) exp(theta[2]))

# The same variance from the estimates and covariance alone
delta_method(coef(fit), transform = function(theta) exp(theta[2]),
             covariance = vcov(fit))
```

---

ee_2sls *Estimating equations for Two-Stage Least Squares (2SLS)*

Description

Estimates the causal effect using two-stage least squares for instrumental variable analysis. The first stage regresses the treatment on the instruments (and exogenous variables), and the second stage regresses the outcome on predicted treatment (and exogenous variables).

Usage

```
ee_2sls(theta, y, A, Z, W = NULL, weights = NULL)
```

Arguments

`theta`	Numeric vector of length `1 + b + 2c`, where `b` is the number of instruments in `Z` and `c` is the number of exogenous variables in `W`. The first `1 + c` parameters are for the second-stage model; the remainder are for the first-stage model.
`y`	Numeric vector of n observed outcomes.
`A`	Numeric vector of n observed treatment values.
`Z`	Numeric n-by-b matrix of instrumental variable(s).
`W`	Optional n-by-c matrix of exogenous variables included in both stages. Default `NULL`.
`weights`	Optional numeric vector of n weights. Default `NULL`.

Value

A (1+b+2c)-by-n matrix of estimating equation contributions. The second-stage rows are named stage2_A for the fitted treatment and stage2_W_1 through stage2_W_c; the first-stage rows are named stage1_Z_1 through stage1_Z_b and stage1_W_1 through stage1_W_c. The columns of W appear in both stages, so each name carries the stage it belongs to.

Examples

```
# A continuous treatment confounded by an unmeasured U, a strong instrument
# Z, and one measured exogenous covariate. The true causal effect is 2.
set.seed(42)
n <- 500
W1 <- rnorm(n)
Z <- cbind(rbinom(n, 1, 0.5))
U <- rnorm(n)
A <- 1.5 * Z[, 1] + 0.3 * W1 + U + rnorm(n, sd = 0.5)
Y <- 2 * A - U + 0.5 * W1 + rnorm(n)
W <- cbind(1, W1)

# At a first stage of all zeros the fitted treatment is identically zero, so
# the second stage degenerates: its leading design column vanishes and the
# coefficient on it, the causal effect, has nothing left to be estimated
# from. Seed the starting values with the ordinary least squares fit of each
# stage instead: the first stage regresses A on the instrument and the
# exogenous covariates, the second regresses Y on the fitted A and the same
# covariates.
alpha_init <- as.numeric(coef(lm(A ~ cbind(Z, W) - 1)))
a_hat <- as.numeric(cbind(Z, W) %*% alpha_init)
beta_init <- as.numeric(coef(lm(Y ~ cbind(a_hat, W) - 1)))

psi <- function(theta) ee_2sls(theta, y = Y, A = A, Z = Z, W = W)

# theta holds the three second-stage coefficients followed by the three
# first-stage coefficients.
m <- m_estimate(
  stacked_equations = psi,
  init = c(beta_init, alpha_init)
)

# theta_1 is the coefficient on the fitted treatment, the causal effect.
coef(m)
```

---

ee_additive_regression

Estimating equation for additive regression (GAM)

Description

Generalized Additive Model via L2-penalized splines. Internally expands `X` using `additive_design_matrix()` and delegates to `ee_bridge_regression()` with `gamma = 2` (ridge penalty). The penalty only applies to the spline basis terms, not to the original linear terms.

Usage

```
ee_additive_regression(
  theta,
  X,
  y,
  specifications,
  model,
  weights = NULL,
  offset = NULL
)
```

Arguments

`theta`	Numeric vector of length equal to the number of columns in the expanded additive design matrix.
`X`	Numeric n-by-b design matrix (before spline expansion).
`y`	Numeric vector of n observed outcome values.
`specifications`	A list of length b controlling spline generation. Each element is either `NULL` (no spline) or a list with keys `knots`, and optionally `natural`, `power`, `penalty`, `normalized`. See `additive_design_matrix()` for details.
`model`	Character string: `"linear"`, `"logistic"`, or `"poisson"`.
`weights`	Optional numeric vector of n weights. Default `NULL`.
`offset`	Optional numeric vector of n offsets. Default `NULL`.

Value

A p-by-n matrix, where p is the number of columns in the expanded additive design matrix.

Examples

```
set.seed(42)
n <- 200
x <- runif(n, -3, 3)
y <- sin(x) + rnorm(n, sd = 0.3)
X <- cbind(1, x)

# No spline on the intercept column, a penalized spline on x.
specs <- list(NULL, list(knots = c(-2, -1, 0, 1, 2), penalty = 5))

psi <- function(theta) {
  ee_additive_regression(
    theta,
```

```
    X = X,
    y = y,
    specifications = specs,
    model = "linear"
  )
}

# One parameter per column of the expanded design matrix.
m <- m_estimate(
  stacked_equations = psi,
  init = rep(0, ncol(additive_design_matrix(X, specs)))
)
coef(m)
```

`ee_aft` *Estimating equation for accelerated failure time models*

Description

Returns a p-by-n matrix of estimating equation contributions for accelerated failure time (AFT) models. Supports Weibull, exponential, log-logistic, and log-normal distributions.

Usage

```
ee_aft(theta, X, time, event, distribution, weights = NULL, offset = NULL)
```

Arguments

`theta`	Numeric vector of length `b + 1` (or `b` for exponential). The first `b` elements are regression coefficients and the last element is `log(1/sigma)` (the log-inverse scale).
`X`	Numeric n-by-b design matrix (should include intercept column).
`time`	Numeric vector of n observed (possibly censored) times.
`event`	Numeric vector of n event indicators (1 = event, 0 = censored).
`distribution`	Character string: `"weibull"`, `"exponential"`, `"log-logistic"`, or `"log-normal"`.
`weights`	Optional numeric vector of n weights. Default `NULL`.
`offset`	Optional numeric vector of n offsets. Default `NULL`.

Details

The AFT model uses the parameterization where $Z_i = (\log(t_i) - X_i\beta)/\sigma$ and the last element of theta is $\log(1/\sigma)$.

Value

A p-by-n matrix where p is the number of parameters. For the exponential distribution, whose parameters are all regression coefficients, the rows are unnamed. For every other distribution the regression rows are named `X_1` through `X_b` for the columns of `X` and the final row is named `log_inv_scale`.

Examples

```
# A Weibull AFT model for times generated from one binary covariate, with
# some of the times right censored by an independent exponential.
set.seed(1)
n <- 200
x <- rbinom(n, 1, 0.5)
Xd <- cbind(1, x)
eps <- log(-log(runif(n)))
t_event <- exp(2 + 0.5 * x + 0.8 * eps)
t_censor <- rexp(n, rate = 0.02)
t_obs <- pmin(t_event, t_censor)
delta <- as.numeric(t_event <= t_censor)

psi <- function(theta) {
  ee_aft(theta, X = Xd, time = t_obs, event = delta, distribution = "weibull")
}

# The default rootSolve solver does not converge here, so nleqslv is used.
# The last parameter is log(1/sigma), not a regression coefficient.
m <- m_estimate(
  stacked_equations = psi,
  init = c(mean(log(t_obs)), 0, 0),
  solver = "nleqslv"
)
coef(m)
```

---

ee_aipw *Estimating equations for augmented inverse probability weighting (AIPW)*

Description

Estimates the average causal effect using AIPW, which combines a propensity score model and an outcome model for doubly-robust estimation.

Usage

```
ee_aipw(theta, y, A, W, X, X1, X0, truncate = NULL, force_continuous = FALSE)
```

Arguments

theta	Numeric vector of length 3 + b + c, where b is the number of propensity score model parameters and c is the number of outcome model parameters.
y	Numeric vector of n observed outcomes.
A	Numeric vector of n binary treatment indicators (0/1).
W	Numeric n-by-b design matrix for the propensity score model.
X	Numeric n-by-c design matrix for the outcome model.
X1	Numeric n-by-c design matrix under A=1 for all units.
X0	Numeric n-by-c design matrix under A=0 for all units.
truncate	Optional length-2 numeric vector c(lower, upper) to clip propensity scores. Bounds must be in ascending order (lower <= upper). Default NULL.
force_continuous	Logical. Force linear regression for outcome model? Default FALSE.

Value

A (3+b+c)-by-n matrix of estimating equation contributions, with the first three rows named ACE, E[Y^1], and E[Y^0], the propensity score rows named W_1 through W_b, and the outcome model rows named X_1 through X_c.

Examples

```
# A binary treatment, two confounders, and a continuous outcome whose true
# average causal effect is 1.5.
set.seed(42)
n <- 1000
W1 <- rnorm(n)
W2 <- rbinom(n, 1, 0.4)
A <- rbinom(n, 1, inverse_logit(-0.5 + 0.5 * W1 + 0.3 * W2))
Y <- 2 + 1.5 * A + W1 - 0.5 * W2 + rnorm(n)

W_ps <- cbind(1, W1, W2) # Propensity score design matrix
X <- cbind(1, A, W1, W2) # Outcome model, observed design matrix
X1 <- cbind(1, 1, W1, W2) # Outcome model, everyone treated
X0 <- cbind(1, 0, W1, W2) # Outcome model, everyone untreated

psi <- function(theta) {
  ee_aipw(theta, y = Y, A = A, W = W_ps, X = X, X1 = X1, X0 = X0)
}

# theta holds the average causal effect, the mean under treatment, and the
# mean under no treatment, followed by the three propensity score
# coefficients and the four outcome model coefficients.
m <- m_estimate(stacked_equations = psi, init = rep(0, 10))
summary(m, subset = 1:3)
```

`ee_beta_regression` *Estimating equation for beta regression*

Description

Beta regression for outcomes in (0, 1) using mean-precision parameterization. The last element of theta is log(phi), the log precision parameter.

Usage

```
ee_beta_regression(theta, X, y, weights = NULL, offset = NULL)
```

Arguments

`theta`	Numeric vector of length b + 1.
`X`	Numeric n-by-b design matrix.
`y`	Numeric vector of n outcomes in (0, 1).
`weights`	Optional numeric vector of n weights. Default `NULL`.
`offset`	Optional numeric vector of n offsets. Default `NULL`.

Value

A `(b+1)`-by-n matrix. The rows are named `X_1` through `X_b` for the columns of `X`, and the final row is named `log_phi`.

Examples

```
set.seed(42)
n <- 50
W <- rnorm(n)
mu <- 1 / (1 + exp(-(0.5 + 0.3 * W)))
y <- rbeta(n, shape1 = mu * 10, shape2 = (1 - mu) * 10)
X <- cbind(1, W)

psi <- function(theta) ee_beta_regression(theta, X = X, y = y)

# The last parameter is log(phi), started here at a precision of 10. The
# default solver diverges on this equation, so use nleqslv.
m <- m_estimate(
  stacked_equations = psi,
  init = c(0, 0, log(10)),
  solver = "nleqslv"
)
coef(m)
```

`ee_bridge_regression` *Estimating equation for bridge penalized regression*

Description

Returns a p-by-n matrix for bridge penalized regression. Bridge is the general case: ridge is `gamma = 2`, LASSO approximation is `gamma = 1 + epsilon`.

Usage

```
ee_bridge_regression(
  theta,
  X,
  y,
  model,
  penalty,
  gamma,
  weights = NULL,
  center = 0,
  offset = NULL
)
```

Arguments

`theta`	Numeric vector of length p.
`X`	Numeric n-by-p design matrix.
`y`	Numeric vector of n observed outcome values.
`model`	Character string: `"linear"`, `"logistic"`, or `"poisson"`.
`penalty`	Numeric scalar or vector of length p. Must be non-negative.
`gamma`	Numeric bridge exponent. Must be at least 1. Values below 2 yield a penalty that is not everywhere differentiable, so the sandwich variance is not defined in all settings and a warning is issued.
`weights`	Optional numeric vector of n weights. Default `NULL`.
`center`	Numeric scalar or vector. Default `0`.
`offset`	Optional numeric vector of n offsets. Default `NULL`.

Value

A p-by-n matrix.

Examples

```
# A penalty vector gives one value per column of the design matrix. A scalar
# penalty would shrink the intercept along with the slopes. The estimating
# equation carries the penalty's derivative, which varies like
# |theta|^(gamma - 1) near the penalty center. That derivative is itself
# differentiable only once gamma reaches 2, so gamma = 2.3 issues no
# warning while a value below 2 would.
fit <- m_estimate(
  mpg ~ wt + hp,
  data = mtcars,
  .ee = ee_bridge_regression,
  model = "linear",
  penalty = c(0, 5, 5),
  gamma = 2.3
)
coef(fit)
```

---

ee_dlasso_regression *Estimating equation for differentiable LASSO regression*

Description

Uses a smooth approximation to the L1 penalty based on the standard normal CDF and PDF.

Usage

```
ee_dlasso_regression(
  theta,
  X,
  y,
  model,
  penalty,
  s = 1e-06,
  weights = NULL,
  center = 0,
  offset = NULL
)
```

Arguments

theta	Numeric vector of length p.
X	Numeric n-by-p design matrix.
y	Numeric vector of n observed outcome values.
model	Character string: "linear", "logistic", or "poisson".
penalty	Numeric scalar or vector of length p. Must be non-negative.

s	Numeric smoothing parameter. Must be greater than zero. Default 1e-6.
weights	Optional numeric vector of n weights. Default NULL.
center	Numeric scalar or vector. Default 0.
offset	Optional numeric vector of n offsets. Default NULL.

Value

A p-by-n matrix.

Examples

```
# A penalty vector gives one value per column of the design matrix. A scalar
# penalty would shrink the intercept along with the slopes. The estimating
# equation carries the penalty's derivative. Here that derivative is smooth,
# so the estimating equation is differentiable and no warning is issued,
# unlike ee_lasso_regression() whose derivative has unbounded slope at the
# penalty center.
fit <- m_estimate(
  mpg ~ wt + hp,
  data = mtcars,
  .ee = ee_dlasso_regression,
  model = "linear",
  penalty = c(0, 5, 5)
)
coef(fit)
```

---

ee_elasticnet_regression

Estimating equation for elastic net regression

Description

Combines L1 (approximate LASSO via bridge) and L2 (ridge) penalties at a given ratio. When ratio = 1, this is LASSO; when ratio = 0, ridge.

Usage

```
ee_elasticnet_regression(
  theta,
  X,
  y,
  model,
  penalty,
  ratio,
  epsilon = 0.003,
  weights = NULL,
  center = 0,
```

```
  offset = NULL
)
```

Arguments

theta	Numeric vector of length p.
X	Numeric n-by-p design matrix.
y	Numeric vector of n observed outcome values.
model	Character string: "linear", "logistic", or "poisson".
penalty	Numeric scalar or vector of length p. Must be non-negative.
ratio	Numeric between 0 and 1. Proportion of L1 vs L2 penalty.
epsilon	Numeric LASSO approximation parameter. Default 0.003.
weights	Optional numeric vector of n weights. Default NULL.
center	Numeric scalar or vector. Default 0.
offset	Optional numeric vector of n offsets. Default NULL.

Value

A p-by-n matrix.

Examples

```
# A penalty vector gives one value per column of the design matrix. A scalar
# penalty would shrink the intercept along with the slopes.
#
# The L1 half of the penalty enters the estimating equation as its own
# derivative, and that derivative has unbounded slope at the penalty center.
# The estimating equation is therefore not differentiable there, so the
# bread matrix is undefined and the fit warns once that the sandwich
# variance should not be trusted here.
fit <- m_estimate(
  mpg ~ wt + hp,
  data = mtcars,
  .ee = ee_elasticnet_regression,
  model = "linear",
  penalty = c(0, 5, 5),
  ratio = 0.5
)
coef(fit)
```

ee_emax *Estimating equation for E-max dose-response model*

Description

Implements the hyperbolic E-max (Hill) model:

$$R_i = \theta_0 + \frac{\theta_{max} D_i}{\theta_{50} + D_i}$$

Usage

```
ee_emax(theta, dose, response, loss = NULL, k = NULL)
```

Arguments

theta	Numeric vector of length 3: zero-dose response (e0), maximum change in response (emax), ED50. emax is the change the response approaches as the dose grows without bound, not the response itself, so the asymptote is e0 + emax.
dose	Numeric vector of n dose values.
response	Numeric vector of n response values.
loss	Optional character string for robust loss function. Default NULL (no robust loss). See robust_loss_functions().
k	Optional numeric tuning parameter for robust loss.

Value

A 3-by-n matrix, with rows named e0, emax, and ed50.

See Also

ee_emax_ed() for the effective dose at a given level, which is stacked with this equation to give it a sandwich standard error.

Examples

```
# Dose-response of a herbicide on ryegrass root length. The response falls
# with dose, so the maximum change in response is initialized negative.
psi <- function(theta) {
  ee_emax(theta, dose = inderjit$dose, response = inderjit$response)
}

m <- m_estimate(stacked_equations = psi, init = c(8, -8, 2))

# Zero-dose response, maximum change in response, and ED50
coef(m)
```

ee_emax_ed *Estimating equation for delta-effective dose (E-max)*

Description

Computes the effective dose at level delta from the E-max model. Should be stacked with ee_emax().

Usage

```
ee_emax_ed(theta, dose, delta, ed50)
```

Arguments

theta	Numeric scalar. The ED(delta) parameter.
dose	Numeric vector (used for dimension only).
delta	Numeric effective dose level of interest.
ed50	Numeric estimated ED50 from E-max model.

Value

A 1-by-n matrix.

See Also

ee_emax(), the dose-response equation this one is stacked with.

Examples

```
# This equation carries no information on its own, so stack it with the
# E-max equation that supplies the ED50. Here delta = 0.9 requests the ED90.
psi <- function(theta) {
  emax <- ee_emax(
    theta[1:3],
    dose = inderjit$dose,
    response = inderjit$response
  )
  ed90 <- ee_emax_ed(
    theta[4],
    dose = inderjit$dose,
    delta = 0.9,
    ed50 = theta[3]
  )
  rbind(emax, ed90)
}

m <- m_estimate(stacked_equations = psi, init = c(8, -8, 2, 10))

# theta_4 is the ED90. Stacking is what gives it a standard error of its own,
```

```
# so summary() rather than coef() is what shows the gain.
summary(m)
```

`ee_gestimation_snmm` *Estimating equations for g-estimation of structural nested mean models*

Description

Estimates the parameters of a structural nested mean model via g-estimation. Supports both inefficient (X = NULL) and efficient (X provided) g-estimators, and both linear and log-linear (Poisson) structural mean models.

Usage

```
ee_gestimation_snmm(
  theta,
  y,
  A,
  W,
  V,
  X = NULL,
  model = "linear",
  weights = NULL
)
```

Arguments

`theta`	Numeric vector. For the inefficient g-estimator, length is `b + c` (SMM parameters + PS model parameters). For the efficient g-estimator, length is `b + c + d` (SMM + PS + outcome model parameters).
`y`	Numeric vector of n observed outcomes.
`A`	Numeric vector of n binary treatment indicators (0/1).
`W`	Numeric n-by-c design matrix for the propensity score model.
`V`	Numeric n-by-b design matrix for the structural mean model. Should NOT include A itself.
`X`	Optional n-by-d design matrix for the outcome model (efficient g-estimator). Default `NULL` (inefficient g-estimator).
`model`	Character string: `"linear"` or `"poisson"`. Default `"linear"`.
`weights`	Optional numeric vector of n weights. Default `NULL`.

Value

A matrix of estimating equation contributions. The structural mean model rows are named `SNM phi_0` through `SNM phi_(b-1)`, matching the zero-based subscripts the literature gives those parameters. The propensity score rows are named `W_1` through `W_c`, and, for the efficient g-estimator, the outcome model rows are named `X_1` through `X_d`.

Examples

```
# A confounded binary treatment whose true effect on the outcome is -2.
set.seed(42)
n <- 500
W <- rbinom(n, 1, 0.5)
A <- rbinom(n, 1, 0.25 + 0.5 * W)
Y <- 5 + 2 * W - 2 * A + rnorm(n)

W_ps <- cbind(1, W) # Propensity score design matrix

# An intercept-only structural mean model gives a single causal contrast.
# Build it with rep() so the column has one entry per observation.
V <- cbind(rep(1, n))

psi <- function(theta) {
  ee_gestimation_snmm(theta, y = Y, A = A, W = W_ps, V = V, model = "linear")
}

# theta holds the structural mean model coefficient, which is the causal
# effect, followed by the two propensity score coefficients.
m <- m_estimate(stacked_equations = psi, init = rep(0, 3))
coef(m)
```

`ee_gformula` *Estimating equations for the g-formula (g-computation)*

Description

Returns a stacked set of estimating equations for the g-formula. When `X0 = NULL`, estimates a single causal mean under the plan encoded by `X1`. When `X0` is provided, estimates the average causal effect (difference between two plans).

Usage

```
ee_gformula(theta, y, X, X1, X0 = NULL, force_continuous = FALSE)
```

Arguments

`theta`	Numeric vector. If `X0 = NULL`, length is `1 + p` (causal mean + regression coefficients). If `X0` is provided, length is `3 + p` (ACE, mean under X1, mean under X0, regression coefficients).
`y`	Numeric vector of n observed outcomes.
`X`	Numeric n-by-p design matrix (observed data).
`X1`	Numeric n-by-p design matrix under action plan 1.
`X0`	Optional n-by-p design matrix under action plan 0. Default `NULL`.
`force_continuous`	Logical. Force linear regression even when y is binary? Default `FALSE`.

Value

A matrix of estimating equation contributions. When `X0 = NULL` the first row is named `causal_mean`; when `X0` is provided the first three rows are named `ACE`, `E[Y^1]`, and `E[Y^0]`, where 1 and 0 index the two plans. The outcome model rows are named `X_1` through `X_p` for the columns of `X`.

Examples

```
# A binary treatment, two confounders, and a continuous outcome whose true
# average causal effect is 1.5.
set.seed(42)
n <- 1000
W1 <- rnorm(n)
W2 <- rbinom(n, 1, 0.4)
A <- rbinom(n, 1, inverse_logit(-0.5 + 0.5 * W1 + 0.3 * W2))
Y <- 2 + 1.5 * A + W1 - 0.5 * W2 + rnorm(n)

X <- cbind(1, A, W1, W2) # Observed design matrix
X1 <- cbind(1, 1, W1, W2) # Everyone treated
X0 <- cbind(1, 0, W1, W2) # Everyone untreated

psi <- function(theta) ee_gformula(theta, y = Y, X = X, X1 = X1, X0 = X0)

# theta holds the average causal effect, the mean under treatment, and the
# mean under no treatment, followed by the four outcome model coefficients.
m <- m_estimate(stacked_equations = psi, init = rep(0, 7))
coef(m)[1:3]
```

`ee_glm` *Estimating equation for generalized linear models*

Description

Returns a p-by-n matrix of estimating equation contributions for a GLM with specified distribution and link function:

$$\psi_i(\theta) = \{Y_i - g^{-1}(X_i^T\theta)\}\frac{D(\theta)}{v(\theta)}X_i$$

Usage

```
ee_glm(
  theta,
  X,
  y,
  distribution,
  link,
  hyperparameter = NULL,
  weights = NULL,
  offset = NULL
)
```

Arguments

`theta`	Numeric vector of length p.
`X`	Numeric n-by-p design matrix.
`y`	Numeric vector of n observed outcome values.
`distribution`	Character string: `"normal"` (or `"gaussian"`), `"binomial"` (or `"bernoulli"`, or `"bin"`), `"poisson"`, `"gamma"`, `"negative_binomial"` (or `"nb"`), `"inverse_normal"` (or `"inverse_gaussian"`), `"tweedie"`. Each alias is one Python delicatessen accepts, and this equation accepts the same set.
`link`	Character string: `"identity"`, `"log"`, `"logit"` (or `"logistic"`), `"probit"`, `"cauchit"` (or `"cauchy"`), `"loglog"`, `"cloglog"`, `"inverse"`, `"sqrt"` (or `"square_root"`).
`hyperparameter`	Numeric scalar power p for the variance function of the `"tweedie"` distribution. Must be non-negative. Ignored by every other distribution. Default `NULL`.
`weights`	Optional numeric vector of n weights. Default `NULL`.
`offset`	Optional numeric vector of n offsets. Default `NULL`.

Value

A p-by-n matrix. For the gamma and negative binomial distributions the matrix has `ncol(X) + 1` rows, the final row being the nuisance equation for that distribution's extra parameter (the gamma shape or the negative binomial dispersion). Those two distributions name their rows `X_1` through `X_p` for the columns of `X`, followed by `log_shape` or `log_dispersion`. Every other distribution estimates one coefficient per design column and leaves its rows unnamed.

Gamma

The gamma distribution estimates one additional shape parameter beyond the regression coefficients, so `theta` has length `ncol(X) + 1`. The final element is the log of the shape α (the reciprocal of the GLM dispersion $\phi = 1/\alpha$), and the variance function is μ^2. An extra nuisance estimating equation for $\log(\alpha)$ is appended:

$$\left(1 - \frac{Y_i}{\mu_i}\right) + \log\left(\frac{\alpha Y_i}{\mu_i}\right) - \psi(\alpha)$$

where ψ is the digamma function. The returned matrix therefore has `ncol(X) + 1` rows.

Negative binomial

The negative binomial distribution estimates one additional dispersion parameter beyond the regression coefficients, so `theta` has length `ncol(X) + 1`. The final element is the log of the dispersion α, and the variance function is $\mu + \alpha\mu^2$. An extra nuisance estimating equation for $\log(\alpha)$ is appended so that the uncertainty in the dispersion is carried honestly through the sandwich variance:

$$-\alpha^{-2}\psi\left(Y_i + \alpha^{-1}\right) + \alpha^{-2}\psi\left(\alpha^{-1}\right) + \frac{Y_i}{\alpha^2\mu_i + \alpha} - \frac{\frac{\alpha\mu_i}{\alpha\mu_i+1} + \log\left(\frac{1}{\alpha\mu_i+1}\right)}{\alpha^2}$$

where ψ is the digamma function. The returned matrix therefore has `ncol(X) + 1` rows.

Tweedie

The tweedie distribution uses the variance function $v(\mu) = \mu^p$, where the power p is the fixed `hyperparameter` rather than an estimated parameter. No nuisance estimating equation is appended, so `theta` has length `ncol(X)` and the returned matrix has `ncol(X)` rows. The power recovers familiar special cases: $p = 1$ gives the Poisson variance μ and $p = 2$ gives the gamma variance μ^2, so those choices share the corresponding beta score equations. The `hyperparameter` must be non-negative.

Examples

```
# Negative binomial GLM with a log link estimates the regression
# coefficients plus one log-dispersion parameter, so init has ncol(X) + 1
# entries.
set.seed(1)
n <- 200
X <- cbind(1, rnorm(n), rnorm(n))
mu <- exp(0.5 + 0.5 * X[, 2] - 0.3 * X[, 3])
y <- rnbinom(n, size = 2, mu = mu)

psi <- function(theta) {
  ee_glm(theta, X = X, y = y, distribution = "negative_binomial",
         link = "log")
}

m <- m_estimate(stacked_equations = psi, init = c(0, 0, 0, 0))
coef(m)

# Tweedie GLM with a log link. The power p is a fixed hyperparameter, not an
# estimated parameter, so init has ncol(X) entries. Here p = 1.5 sits in the
# compound Poisson-gamma regime, appropriate for non-negative data with a
# point mass at zero.
set.seed(2)
mu <- exp(0.5 + 0.5 * X[, 2] - 0.3 * X[, 3])
y_tw <- rpois(n, lambda = mu)

psi_tw <- function(theta) {
  ee_glm(theta, X = X, y = y_tw, distribution = "tweedie", link = "log",
         hyperparameter = 1.5)
}

m_tw <- m_estimate(stacked_equations = psi_tw, init = c(0, 0, 0))
coef(m_tw)
```

---

ee_ipw *Estimating equations for inverse probability weighting (IPW)*

Description

Estimates the average causal effect using IPW with a logistic propensity score model.

Usage

```
ee_ipw(theta, y, A, W, truncate = NULL, weights = NULL)
```

Arguments

theta	Numeric vector of length 3 + b, where b is the number of propensity score model parameters.
y	Numeric vector of n observed outcomes.
A	Numeric vector of n binary treatment indicators (0/1).
W	Numeric n-by-b design matrix for the propensity score model.
truncate	Optional length-2 numeric vector c(lower, upper) to clip estimated propensity scores. Bounds must be in ascending order (lower <= upper). Default NULL.
weights	Optional numeric vector of n weights. Default NULL.

Value

A (3+b)-by-n matrix of estimating equation contributions, with the first three rows named ACE, E[Y^1], and E[Y^0] and the propensity score rows named W_1 through W_b for the columns of W.

Examples

```
# A binary treatment, two confounders, and a continuous outcome whose true
# average causal effect is 1.5.
set.seed(42)
n <- 1000
W1 <- rnorm(n)
W2 <- rbinom(n, 1, 0.4)
A <- rbinom(n, 1, inverse_logit(-0.5 + 0.5 * W1 + 0.3 * W2))
Y <- 2 + 1.5 * A + W1 - 0.5 * W2 + rnorm(n)

W_ps <- cbind(1, W1, W2) # Propensity score design matrix

psi <- function(theta) ee_ipw(theta, y = Y, A = A, W = W_ps)

# theta holds the average causal effect, the mean under treatment, and the
# mean under no treatment, followed by the three propensity score
# coefficients.
m <- m_estimate(stacked_equations = psi, init = rep(0, 6))
coef(m)[1:3]
```

---

ee_ipw_msm *Estimating equations for IPW marginal structural model*

Description

Estimates the parameters of a marginal structural model using inverse probability weighting with a logistic propensity score model.

Usage

```
ee_ipw_msm(
  theta,
  y,
  A,
  W,
  V,
  distribution,
  link,
  hyperparameter = NULL,
  truncate = NULL,
  weights = NULL
)
```

Arguments

theta	Numeric vector of length c + b, where c is the number of MSM parameters and b is the number of propensity score model parameters.
y	Numeric vector of n observed outcomes.
A	Numeric vector of n binary treatment indicators (0/1).
W	Numeric n-by-b design matrix for the propensity score model.
V	Numeric n-by-c design matrix for the marginal structural model.
distribution	Character string for the GLM distribution of the outcome model. Every name ee_glm() takes except the three that estimate a nuisance parameter, which the theta partition here reserves no slot for: "normal" (or "gaussian"), "binomial" (or "bernoulli", or "bin"), "poisson", "inverse_normal" (or "inverse_gaussian"), and "tweedie". See hyperparameter for what naming one of the three does.
link	Character string for the GLM link function. See ee_glm() for options.
hyperparameter	Optional numeric hyperparameter passed straight through to the marginal structural model's ee_glm() call. Used by the tweedie outcome distribution, where it fixes the variance power v(mu) = mu^hyperparameter. Default NULL. Note that the theta partition reserves exactly ncol(V) slots for the MSM and no slot for an estimated nuisance parameter, so outcome families that carry one ("gamma", "negative_binomial", "nb") cannot be used as the MSM outcome model. Naming one is refused by name, before the outcome model is formed and whatever hyperparameter is set to. Python Delicatessen cannot fit those families either, where the attempt fails as a shape error.
truncate	Optional length-2 numeric vector c(lower, upper) to clip estimated propensity scores. Bounds must be in ascending order (lower <= upper). Default NULL.
weights	Optional numeric vector of n weights. Default NULL.

Value

A (c+b)-by-n matrix of estimating equation contributions. The marginal structural model rows are named MSM alpha_0 through MSM alpha_(c-1), matching the zero-based subscripts the literature gives those parameters. The propensity score rows are named W_1 through W_b for the columns of W.

Examples

```
# A confounded binary treatment whose true effect on the outcome is -2.
set.seed(42)
n <- 500
W <- rbinom(n, 1, 0.5)
A <- rbinom(n, 1, 0.25 + 0.5 * W)
Y <- 5 + 2 * W - 2 * A + rnorm(n)

W_ps <- cbind(1, W) # Propensity score design matrix
V <- cbind(1, A) # Marginal structural model design matrix

psi <- function(theta) {
  ee_ipw_msm(
    theta,
    y = Y,
    A = A,
    W = W_ps,
    V = V,
    distribution = "normal",
    link = "identity"
  )
}

# theta holds the two marginal structural model coefficients, whose slope is
# the causal effect, followed by the two propensity score coefficients.
m <- m_estimate(stacked_equations = psi, init = rep(0, 4))
coef(m)
```

---

ee_iv_causal *Estimating equations for instrumental variable (IV) estimation*

Description

Estimates the causal effect using the usual IV / Wald estimator. The parameter of interest is the additive effect of treatment A on outcome Y leveraging instrument Z.

Usage

```
ee_iv_causal(theta, y, A, Z, weights = NULL)
```

Arguments

theta	Numeric vector of length 2: the causal effect and the mean of the instrument.
y	Numeric vector of n observed outcomes.
A	Numeric vector of n observed treatment values.
Z	Numeric vector of n binary instrument values (0/1).
weights	Optional numeric vector of n weights. Default NULL.

Value

A 2-by-n matrix of estimating equation contributions, with rows named `causal_effect` and `mean_Z`.

Examples

```
# An unmeasured confounder U biases the association between A and Y, but the
# instrument Z affects Y only through A. The true causal effect is 3.
set.seed(123)
n <- 500
Z <- rbinom(n, 1, 0.5)
U <- rnorm(n)
A <- rbinom(n, 1, inverse_logit(-1 + 3 * Z + U))
Y <- 3 * A - U + rnorm(n, sd = 0.5)

psi <- function(theta) ee_iv_causal(theta, y = Y, A = A, Z = Z)

# theta holds the causal effect followed by the mean of the instrument.
m <- m_estimate(stacked_equations = psi, init = c(0, 0.5))
coef(m)
```

---

ee_lasso_regression *Estimating equation for approximate LASSO regression*

Description

Uses the bridge penalty with `gamma = 1 + epsilon` to approximate LASSO (L1 penalty). The true LASSO is not differentiable at zero, so an approximation is used.

Usage

```
ee_lasso_regression(
  theta,
  X,
  y,
  model,
  penalty,
  epsilon = 0.003,
  weights = NULL,
  center = 0,
  offset = NULL
)
```

Arguments

`theta`	Numeric vector of length p.
`X`	Numeric n-by-p design matrix.
`y`	Numeric vector of n observed outcome values.

model	Character string: "linear", "logistic", or "poisson".
penalty	Numeric scalar or vector of length p. Must be non-negative.
epsilon	Numeric approximation parameter. Default 0.003.
weights	Optional numeric vector of n weights. Default NULL.
center	Numeric scalar or vector. Default 0.
offset	Optional numeric vector of n offsets. Default NULL.

Value

A p-by-n matrix.

Examples

```
# A penalty vector gives one value per column of the design matrix. A scalar
# penalty would shrink the intercept along with the slopes.
#
# The approximate L1 penalty enters the estimating equation as its own
# derivative, and that derivative has unbounded slope at the penalty center.
# The estimating equation is therefore not differentiable there, so the
# bread matrix is undefined and the fit warns once that the sandwich
# variance should not be trusted here.
fit <- m_estimate(
  mpg ~ wt + hp,
  data = mtcars,
  .ee = ee_lasso_regression,
  model = "linear",
  penalty = c(0, 5, 5)
)
coef(fit)
```

---

ee_loglogistic *Estimating equation for 4-parameter log-logistic dose-response model*

Description

Implements the 4-parameter log-logistic model:

$$R_i = \theta_0 + \frac{\theta_m - \theta_0}{1 + \exp[\theta_s(\log D_i - \log \theta_{50})]}$$

Usage

```
ee_loglogistic(theta, dose, response, loss = NULL, k = NULL)
```

Arguments

theta	Numeric vector of length 4: lower limit, upper limit, ED50, steepness.
dose	Numeric vector of n dose values.
response	Numeric vector of n response values.
loss	Optional character string for robust loss function.
k	Optional numeric tuning parameter for robust loss.

Value

A 4-by-n matrix, with rows named `lower`, `upper`, `ed50`, and `steepness`.

See Also

`ee_loglogistic_ed()` for the effective dose at a given level, which is stacked with this equation to give it a sandwich standard error.

Examples

```
# Four-parameter log-logistic dose-response for the ryegrass data.
psi <- function(theta) {
  ee_loglogistic(theta, dose = inderjit$dose, response = inderjit$response)
}

# The default rootSolve solver does not converge here, so nleqslv is used.
m <- m_estimate(
  stacked_equations = psi,
  init = c(0.2, 8, 2, 1),
  solver = "nleqslv"
)

# Lower limit, upper limit, ED50, and steepness
coef(m)
```

---

ee_loglogistic_ed *Estimating equation for delta-effective dose (log-logistic)*

Description

Computes the effective dose at level delta from the log-logistic model. Should be stacked with `ee_loglogistic()`.

Usage

```
ee_loglogistic_ed(theta, dose, delta, lower, upper, ed50, steepness)
```

Arguments

theta	Numeric scalar. The ED(delta) parameter.
dose	Numeric vector (used for dimension only).
delta	Numeric effective dose level of interest.
lower	Numeric lower limit parameter.
upper	Numeric upper limit parameter.
ed50	Numeric estimated ED50.
steepness	Numeric steepness parameter.

Value

A 1-by-n matrix.

See Also

ee_loglogistic(), the dose-response equation this one is stacked with.

Examples

```
# The lower limit is held at zero instead of being estimated: root length
# cannot fall below zero, and the five-parameter stack diverges on these
# data. The lower-limit row of the log-logistic equation is therefore
# dropped, leaving the upper limit, ED50, and steepness to be estimated
# alongside the ED90.
psi <- function(theta) {
  loglogistic <- ee_loglogistic(
    c(0, theta[1:3]),
    dose = inderjit$dose,
    response = inderjit$response
  )
  ed90 <- ee_loglogistic_ed(
    theta[4],
    dose = inderjit$dose,
    delta = 0.9,
    lower = 0,
    upper = theta[1],
    ed50 = theta[2],
    steepness = theta[3]
  )
  rbind(loglogistic[-1, , drop = FALSE], ed90)
}

# This stacked system is sensitive to its starting values, so a reasonable
# init matters more here than it does for most equations.
m <- m_estimate(stacked_equations = psi, init = c(8, 2, 1, 5))

# Upper limit, ED50, steepness, and the ED90. Stacking is what gives the ED90
# a standard error of its own, so summary() rather than coef() shows the
# gain.
```

```
summary(m)
```

`ee_mean` *Estimating equation for the mean*

Description

Returns a 1-by-n matrix of estimating equation contributions for the mean: $\psi_i(\theta) = w_i(Y_i - \theta)$.

Usage

```
ee_mean(theta, y, weights = NULL)
```

Arguments

`theta`	Numeric vector of length 1.
`y`	Numeric vector of observed values.
`weights`	Optional numeric vector of weights (same length as `y`). Default `NULL` assigns weight 1 to all observations.

Value

A 1-by-n matrix.

Examples

```
y <- c(1, 2, 3, 1, 4, 5, 3, 2, 6, 7)
psi <- function(theta) ee_mean(theta, y = y)
m <- m_estimate(stacked_equations = psi, init = 0)
coef(m)
```

`ee_mean_geometric` *Estimating equation for the geometric mean*

Description

Returns a 1-by-n matrix of estimating equation contributions for the geometric mean. When `log_theta = TRUE` (default), solves $\log(Y_i) - \log(\theta)$. When `log_theta = FALSE`, solves $\log(Y_i) - \theta$ where θ is the log of the geometric mean.

Usage

```
ee_mean_geometric(theta, y, weights = NULL, log_theta = TRUE)
```

Arguments

theta	Numeric vector of length 1.
y	Numeric vector of positive observed values.
weights	Optional numeric vector of weights. Default NULL.
log_theta	Logical. If TRUE (default), internally log-transforms theta. Default TRUE.

Value

A 1-by-n matrix.

Examples

```
y <- c(1, 2, 3, 1, 4, 5, 3, 2, 6, 7)
psi <- function(theta) ee_mean_geometric(theta, y = y)
m <- m_estimate(stacked_equations = psi, init = 1)
coef(m)
```

---

ee_mean_robust *Estimating equation for the robust mean*

Description

Returns a 1-by-n matrix of estimating equation contributions for the robust mean using the specified loss function: $\psi_i(\theta) = f_k(Y_i - \theta)$.

Usage

```
ee_mean_robust(theta, y, k, loss = "huber")
```

Arguments

theta	Numeric vector of length 1.
y	Numeric vector of observed values.
k	Numeric tuning parameter for the loss function.
loss	Character string specifying the loss function. Default "huber". See robust_loss_functions() for options.

Value

A 1-by-n matrix.

Examples

```
# Forty-nine standard normal observations plus one gross outlier.
set.seed(1)
y <- c(rnorm(49), 50)
psi <- function(theta) ee_mean_robust(theta, y = y, k = 1.345, loss = "huber")
m <- m_estimate(stacked_equations = psi, init = 0)

# The Huber estimate stays near the uncontaminated mean, unlike mean(y).
coef(m)
mean(y)
```

---

ee_mean_sensitivity_analysis

Estimating equations for weighted sensitivity analysis of the mean

---

Description

Estimates the mean of an outcome with missing data using a weighted sensitivity analysis approach. Handles MCAR, MAR, and MNAR mechanisms by specifying a user-defined sensitivity function `q_eval` and a monotone increasing distribution function `H_function`.

Usage

```
ee_mean_sensitivity_analysis(theta, y, delta, X, q_eval, H_function)
```

Arguments

`theta`	Numeric vector of length `1 + b`, where `b` is the number of columns in `X`. The first element is the corrected mean; the remainder are regression coefficients.
`y`	Numeric vector of n outcome values. Missing values should be indicated via the `delta` parameter.
`delta`	Numeric vector of n indicators: 1 if `y` is observed, 0 if missing. Must not contain `NA`.
`X`	Numeric n-by-b design matrix for the missingness model. Should include an intercept column. Must not contain `NA`.
`q_eval`	Numeric vector of n evaluated sensitivity function values, i.e. `q(Y, alpha)`.
`H_function`	A function mapping real values to `[0, 1]` that is monotone increasing (e.g., `inverse_logit()`).

Value

A `(1+b)`-by-n matrix of estimating equation contributions, with the first row named `corrected_mean` and the missingness model rows named `X_1` through `X_b` for the columns of `X`.

Examples

```
# An outcome observed for only part of the sample, with missingness driven by
# the measured covariate W.
set.seed(42)
n <- 500
W <- rbinom(n, 1, 0.5)
Y_full <- 200 - 35 * W + rnorm(n, sd = 5)
delta <- rbinom(n, 1, inverse_logit(2 + W))

# Missing outcomes never enter the estimating equation, so any placeholder
# value works; a zero keeps the arithmetic finite.
Y <- ifelse(delta == 1, Y_full, 0)
X <- cbind(1, W) # Missingness model design matrix

# A sensitivity function of zero everywhere assumes the outcome is missing at
# random given W. Nonzero values encode departures from that assumption.
psi <- function(theta) {
  ee_mean_sensitivity_analysis(
    theta,
    y = Y,
    delta = delta,
    X = X,
    q_eval = rep(0, n),
    H_function = inverse_logit
  )
}

# theta holds the corrected mean, started near the observed outcomes,
# followed by the two missingness model coefficients.
m <- m_estimate(stacked_equations = psi, init = c(180, 0, 0))
coef(m)
```

`ee_mean_variance` *Estimating equations for the mean and variance*

Description

Returns a 2-by-n matrix of estimating equation contributions for the mean and variance:

$$\psi_i(\theta) = \begin{pmatrix} Y_i - \theta_1 \\ (Y_i - \theta_1)^2 - \theta_2 \end{pmatrix}$$

Usage

```
ee_mean_variance(theta, y)
```

Arguments

`theta`	Numeric vector of length 2. `theta[1]` is the mean, `theta[2]` is the variance.
`y`	Numeric vector of observed values.

Value

A 2-by-n matrix, with rows named `mean` and `variance`.

Examples

```
y <- c(1, 2, 3, 1, 4, 5, 3, 2, 6, 7)
psi <- function(theta) ee_mean_variance(theta, y = y)
m <- m_estimate(stacked_equations = psi, init = c(0, 1))
coef(m)
```

---

ee_mlogit *Estimating equation for multinomial logistic regression*

Description

Supports unranked categorical outcomes. `y` must be an n-by-k indicator matrix where the first column is the reference category, and k must be at least three: a two-level outcome is logistic regression, which `ee_glm()` fits with `distribution = "binomial"`. Returns a `(b * (k-1))`-by-n matrix.

Usage

```
ee_mlogit(theta, X, y, weights = NULL, offset = NULL)
```

Arguments

`theta`	Numeric vector of length `b * (k-1)`.
`X`	Numeric n-by-b design matrix.
`y`	Numeric n-by-k indicator matrix (first column = reference), with k at least three.
`weights`	Optional numeric vector of n weights. Default `NULL`.
`offset`	Optional numeric vector of n offsets. Default `NULL`.

Value

A `(b * (k-1))`-by-n matrix.

Examples

```
set.seed(123)
n <- 50
W <- rbinom(n, 1, 0.5)
probs <- cbind(0.5 - 0.2 * W, 0.3 + 0.1 * W, 0.2 + 0.1 * W)
y_cat <- sapply(seq_len(n), function(i) sample(1:3, 1, prob = probs[i, ]))

# The outcome is an indicator matrix whose first column is the reference
# category.
```

```
y <- cbind(
  as.integer(y_cat == 1),
  as.integer(y_cat == 2),
  as.integer(y_cat == 3)
)
X <- cbind(1, W)

psi <- function(theta) ee_mlogit(theta, X = X, y = y)

# Two columns of X and two non-reference categories give four parameters.
m <- m_estimate(stacked_equations = psi, init = rep(0, 4))
coef(m)
```

`ee_percentile` *Estimating equation for the percentile*

Description

Returns a 1-by-n matrix for the q-th percentile: $\psi_i(\theta) = q - I(Y_i \leq \theta)$.

Usage

```
ee_percentile(theta, y, q)
```

Arguments

`theta`	Numeric vector of length 1.
`y`	Numeric vector of observed values.
`q`	Numeric percentile, must be in `(0, 1)`.

Details

The derivative of this estimating equation is not defined at $\hat{\theta}$, so the bread matrix and sandwich variance cannot be used to estimate the variance. The function warns for this reason. A direct call warns every time; a call from within `m_estimate()`, `gmm_estimate()`, `estimate()` or `compute_sandwich()`, each of which evaluates the estimating function many times, delivers the warning once for the operation. It is offered for completeness but is not generally recommended for applications.

Pass the sample quantile as `init`. The estimating function is a step function of `theta`, so its derivative is zero wherever it is defined and a root finder has no direction in which to search: it returns the starting values it was given. Starting from zero therefore returns zero rather than the quantile.

Value

A 1-by-n matrix.

Examples

```
y <- c(1, 2, 3, 1, 4, 5, 3, 2, 6, 7)
psi <- function(theta) ee_percentile(theta, y = y, q = 0.5)

# The root finder cannot move away from its starting values here, so start at
# the sample quantile, which is the solution. The fit warns once that the
# estimating equation is not differentiable, so the sandwich variance should
# not be trusted.
m <- m_estimate(stacked_equations = psi, init = median(y))
coef(m)
```

---

ee_plogit *Estimating equation for pooled logistic regression*

Description

Returns a p-by-n matrix of estimating equation contributions for pooled logistic regression with discrete-time survival data. This implementation does not require creation of a long data set.

Usage

```
ee_plogit(
  theta,
  X,
  time,
  event,
  S = NULL,
  unique_times = NULL,
  weights = NULL,
  offset = NULL
)
```

Arguments

theta	Numeric vector of length b + p_s, where b is the number of covariate columns in X. When S is supplied, p_s is ncol(S); when S = NULL, p_s is K, the number of unique event times.
X	Numeric n-by-b design matrix for baseline covariates.
time	Numeric vector of n observed (possibly censored) times.
event	Numeric vector of n event indicators (1 = event, 0 = censored).
S	Optional time design matrix with K rows (one per time step) and p_s columns. Default NULL uses disjoint indicators for unique event times. When supplied, time is modeled over the unit-time intervals from one to the maximum observed time, so K has to be the number of those intervals.

`unique_times` Optional numeric vector of unique event times. Default `NULL`. When `S = NULL` it names the time steps the disjoint indicators are built for. When `S` is supplied the grid is the unit-time intervals up to the maximum observed time, and that grid is also the binning of the person-periods the equation is solved on, so `unique_times` may only agree with it: a value equal to it is accepted and changes nothing, and any other value is an error rather than the silently ignored argument Python Delicatessen documents. `plogit_predict()` validates it the same way.

`weights` Optional numeric vector of n weights, or an n-by-K matrix of time-varying weights with one column per time interval (K must equal the number of unit-time intervals). Default `NULL`.

`offset` Optional numeric vector of n offsets. Default `NULL`.

Value

A `(b + p_s)`-by-n matrix.

Examples

```
# Bladder tumor recurrence, comparing the novel treatment to placebo while
# adjusting for the number and size of the initial tumors.
W <- cbind(
  novel = collett_bladder$treat - 1,
  as.matrix(collett_bladder[, c("init", "size")])
)

# Time is modeled with disjoint indicators, one per distinct event time,
# which ee_plogit builds by default.
k <- length(unique(collett_bladder$time[collett_bladder$delta == 1]))

psi <- function(theta) {
  ee_plogit(
    theta,
    X = W,
    time = collett_bladder$time,
    event = collett_bladder$delta
  )
}
m <- m_estimate(
  stacked_equations = psi,
  init = c(rep(0, ncol(W)), -4, rep(0, k - 1))
)

# The first three parameters are the covariate coefficients, which
# approximate log hazard ratios. The rest describe the baseline hazard over
# time: the first of them is the log-odds of an event at the earliest event
# time, and each one after it is that time's departure from it.
summary(m, subset = 1:3)
```

`ee_positive_mean_deviation` *Estimating equations for the positive mean deviation*

Description

Returns a 2-by-n matrix for the positive mean deviation and median.

Usage

```
ee_positive_mean_deviation(theta, y)
```

Arguments

`theta`	Numeric vector of length 2. `theta[1]` is the positive mean deviation, `theta[2]` is the median.
`y`	Numeric vector of observed values.

Details

The derivative of the estimating equation for the median is not defined at $\hat{\theta}$, so the bread matrix and sandwich variance cannot be used to estimate the variance. The function warns for this reason. A direct call warns every time; a call from within `m_estimate()`, `gmm_estimate()`, `estimate()` or `compute_sandwich()`, each of which evaluates the estimating function many times, delivers the warning once for the operation. It is offered for completeness but is not generally recommended for applications.

Start `theta[2]` at the sample median and `theta[1]` at the matching positive mean deviation, `mean(2 * (y - median(y)) * (y > median(y)))`, and use `solver = "lm"`. The median equation is a step function of `theta[2]`, so its derivative is zero wherever it is defined, the finite-difference approximation the solvers work from is zero along that row as well, and the Jacobian is singular. Neither solver recovers the median from that, and they fail differently. The default `rootSolve` solver cannot move at all and returns `init` unchanged for both parameters. Whether it warns that it did not converge depends on the starting values, so a fit can come back holding the values it was given with nothing reported. The Levenberg-Marquardt solver drives the first equation to zero. That equation is itself a function of `theta[2]`, so the solver can reduce its residual by trading `theta[2]` against `theta[1]`. Whether it makes that trade depends on the starting values: from some it leaves the median at the value it was given, and from others it moves the median to whatever value the trade leaves it at. It reports no failure of its own either way, so the second outcome is caught by the fit rather than by the solver: a median equation that the solve left further from zero than it was at the starting values, in a stack whose bread carries its row as zeros, warns that the estimating equations are not solved at the returned values. Neither outcome recovers the sample median on its own, and when the median does move, the positive mean deviation returned belongs to that median rather than to the sample median.

Those starting values come from measurement: forty exponential samples per configuration, each fit with the Levenberg-Marquardt solver. Started at the sample median with `theta[1]` at the matching deviation, the fit returned the sample median for all forty at every one of the sizes 9, 10, 25, 40, 41,

100, and 101. Started at the sample median with `theta[1]` at zero, it returned the sample median for about half of them at the even sizes `n = 10` and `n = 40`, and for all forty at `n = 9`, `n = 25`, and `n = 41`. Started with `theta[1]` at zero and `theta[2]` half a unit, one unit, or three units to either side of the sample median, it returned the sample median for none of the forty, at each of those six offsets and each of the sizes 10, 25, 40, and 41.

Value

A 2-by-n matrix, with rows named `positive_mean_deviation` and `median`.

Examples

```
y <- c(1, 2, 3, 1, 4, 5, 3, 2, 6, 7)
psi <- function(theta) ee_positive_mean_deviation(theta, y = y)

# Start the median at the sample median and the deviation at the value that
# matches it, since the solver cannot search for the median, and use the
# Levenberg-Marquardt solver, which holds there. The fit warns once that the
# median estimating equation is not differentiable, so the sandwich variance
# should not be trusted.
init <- c(mean(2 * (y - median(y)) * (y > median(y))), median(y))
m <- m_estimate(stacked_equations = psi, init = init, solver = "lm")
coef(m)
```

---

ee_regression *Estimating equation for regression*

Description

Returns a p-by-n matrix of estimating equation contributions for regression models. Supports linear, logistic, and Poisson regression:

$$\psi_i(\theta) = \{Y_i - g(X_i^T\theta)\}X_i$$

Usage

```
ee_regression(theta, X, y, model, weights = NULL, offset = NULL)
```

Arguments

`theta`	Numeric vector of length p (number of covariates).
`X`	Numeric n-by-p design matrix.
`y`	Numeric vector of n observed outcome values.
`model`	Character string: `"linear"`, `"logistic"`, or `"poisson"`.
`weights`	Optional numeric vector of n weights. Default `NULL`.
`offset`	Optional numeric vector of n offsets. Default `NULL`.

Value

A p-by-n matrix.

Examples

```
fit <- m_estimate(
  mpg ~ wt + hp,
  data = mtcars,
  .ee = ee_regression,
  model = "linear"
)
summary(fit)

# The same equation fits a logistic regression through the model argument.
fit_logit <- m_estimate(
  vs ~ mpg,
  data = mtcars,
  .ee = ee_regression,
  model = "logistic"
)
coef(fit_logit)
```

---

ee_regression_calibration

Estimating equation for regression calibration

Description

Corrects for measurement error in a binary predictor using external validation data. Scales the naive coefficient by the calibration factor.

Usage

```
ee_regression_calibration(theta, beta, a, a_star, r, X = NULL, weights = NULL)
```

Arguments

theta	Numeric vector: the corrected coefficient first, then the calibration coefficients for the design `cbind(a_star, X)` (the `a_star` coefficient first). Length `2 + ncol(X)` when `X` is supplied, or length `3` when `X = NULL` (an intercept-only calibration model).
beta	Numeric scalar. External estimate of the coefficient for the mismeasured predictor on the outcome.
a	Numeric vector of gold-standard action measurements (validation sample only, where `r = 0`).
a_star	Numeric vector of mismeasured action values.

r	Numeric indicator: 0 for validation, 1 for main study.
X	Optional design matrix for calibration model. Default NULL uses intercept only.
weights	Optional numeric vector of n weights. Default NULL.

Value

A length(theta)-by-n matrix (2 + ncol(X) rows, or 3 rows when X = NULL). The first row is named corrected_beta and the second a_star. The remaining rows are named X_1 through X_p for the columns of X, or intercept when X = NULL.

Examples

```
# A binary exposure is measured with error in the main study. The external
# validation study regresses the gold-standard exposure on the mismeasured
# one, and that calibration slope rescales the naive outcome coefficient.
set.seed(789)
n <- 500
a_true <- rbinom(n, 1, 0.5)
a_star <- ifelse(a_true == 1, rbinom(n, 1, 0.85), rbinom(n, 1, 0.1))
r <- c(rep(0, 200), rep(1, 300))

# The gold standard is observed only in the validation sample, so the main
# study positions carry a 0 placeholder. It never reaches an estimate: the
# calibration model is multiplied by (1 - r).
a <- ifelse(r == 0, a_true, 0)

# `beta` is the naive coefficient for the mismeasured exposure, supplied here
# as a fixed external value. Stack an outcome model and pass its coefficient
# instead to propagate the uncertainty in that estimate as well.
psi <- function(theta) {
  ee_regression_calibration(theta, beta = 0.8, a = a, a_star = a_star, r = r)
}

m <- m_estimate(stacked_equations = psi, init = c(1, 0.1, 0.5))

# Corrected coefficient, then the calibration slope and intercept
coef(m)
```

---

ee_ridge_regression *Estimating equation for ridge regression*

Description

Returns a p-by-n matrix of estimating equation contributions for ridge (L2-penalized) regression:

$$\psi_i(\theta) = \{Y_i - g(X_i^T\theta)\}X_i - \frac{\lambda}{n}\theta$$

Usage

```
ee_ridge_regression(
  theta,
  X,
  y,
  model,
  penalty,
  weights = NULL,
  center = 0,
  offset = NULL
)
```

Arguments

theta	Numeric vector of length p.
X	Numeric n-by-p design matrix.
y	Numeric vector of n observed outcome values.
model	Character string: "linear", "logistic", or "poisson".
penalty	Numeric scalar or vector of length p. Must be non-negative. Penalty terms scaled by n internally.
weights	Optional numeric vector of n weights. Default NULL.
center	Numeric scalar or vector. Center for the penalty. Default 0.
offset	Optional numeric vector of n offsets. Default NULL.

Value

A p-by-n matrix.

Examples

```
# A penalty vector gives one value per column of the design matrix. A scalar
# penalty would shrink the intercept along with the slopes.
fit <- m_estimate(
  mpg ~ wt + hp,
  data = mtcars,
  .ee = ee_ridge_regression,
  model = "linear",
  penalty = c(0, 5, 5)
)
coef(fit)
```

`ee_robust_regression` *Estimating equation for robust regression*

Description

Returns a p-by-n matrix for robust regression using the specified loss function (linear regression only):

$$\psi_i(\theta) = f_k(Y_i - X_i^T\theta)X_i$$

Usage

```
ee_robust_regression(
  theta,
  X,
  y,
  model,
  k,
  loss = "huber",
  weights = NULL,
  offset = NULL
)
```

Arguments

`theta`	Numeric vector of length p.
`X`	Numeric n-by-p design matrix.
`y`	Numeric vector of n observed outcome values.
`model`	Character string. Currently only `"linear"` is supported.
`k`	Numeric tuning parameter for the loss function.
`loss`	Character string specifying the loss function. Default `"huber"`. See `robust_loss_functions()`.
`weights`	Optional numeric vector of n weights. Default `NULL`.
`offset`	Optional numeric vector of n offsets. Default `NULL`.

Value

A p-by-n matrix.

Examples

```
# The Huber loss is convex, so its estimating function has a single root and
# seeding the search is about reaching that root rather than choosing among
# several. What makes the seed necessary is that the Huber psi is bounded:
# far from the solution every residual is past the tuning constant k, every
# contribution saturates at k, and the estimating function is constant with a
# Jacobian of exactly zero. Starting from zero lands in that flat region,
# where the solver has no slope to follow, so start from a least-squares fit.
```

```
start <- coef(lm(weight ~ height, data = robust_regress))

fit <- m_estimate(
  weight ~ height,
  data = robust_regress,
  .ee = ee_robust_regression,
  model = "linear",
  k = 1.345,
  loss = "huber",
  init = start
)
coef(fit)
```

`ee_rogan_gladen` *Estimating equation for Rogan-Gladen correction*

Description

Corrects for mismeasured binary outcomes using external validation data to estimate sensitivity and specificity.

Usage

```
ee_rogan_gladen(theta, y, y_star, r, weights = NULL)
```

Arguments

`theta`	Numeric vector of length 4: corrected proportion, naive proportion, sensitivity, specificity.
`y`	Numeric vector of gold-standard measurements (only available in external validation sample where `r = 0`).
`y_star`	Numeric vector of mismeasured outcome values (all observations).
`r`	Numeric indicator: 1 for main study data, 0 for external validation.
`weights`	Optional numeric vector of n weights. Default `NULL`.

Value

A 4-by-n matrix, with rows named `corrected_proportion`, `naive_proportion`, `sensitivity`, and `specificity`.

Examples

```
# A main study measures a binary outcome with an imperfect test. An external
# validation study measures both the test and the gold standard, and so
# informs the sensitivity and specificity used to correct the main study.
set.seed(2)
n_main <- 500
```

```
n_validation <- 400
n <- n_main + n_validation
y_true <- rbinom(n, 1, 0.25)
y_star <- ifelse(y_true == 1, rbinom(n, 1, 0.9), 1 - rbinom(n, 1, 0.85))
r <- c(rep(1, n_main), rep(0, n_validation))

# The gold standard is observed only in the validation sample, so the main
# study positions carry a 0 placeholder. It never reaches an estimate: the
# sensitivity and specificity equations are multiplied by (1 - r).
y <- ifelse(r == 0, y_true, 0)

psi <- function(theta) {
  ee_rogan_gladen(theta, y = y, y_star = y_star, r = r)
}

m <- m_estimate(
  stacked_equations = psi,
  init = c(0.5, 0.5, 0.75, 0.75)
)

# Corrected prevalence, naive prevalence, sensitivity, specificity
coef(m)
```

`ee_rogan_gladen_extended` *Estimating equation for extended Rogan-Gladen correction*

Description

Extended version that conditions sensitivity and specificity on covariates using logistic regression models.

Usage

```
ee_rogan_gladen_extended(theta, y, y_star, r, X, weights = NULL)
```

Arguments

`theta`	Numeric vector of length `1 + 2*p`: corrected proportion, then `p` sensitivity model parameters, then `p` specificity model parameters.
`y`	Numeric vector of gold-standard measurements (validation sample where `r = 0`).
`y_star`	Numeric vector of mismeasured outcome values.
`r`	Numeric indicator: 1 for main study, 0 for validation.
`X`	Numeric n-by-p design matrix for sensitivity/specificity models.
`weights`	Optional numeric vector of n weights. Default `NULL`.

Value

A (1+2*p)-by-n matrix, with rows named corrected_proportion, then sens_1 through sens_p for the sensitivity model, then spec_1 through spec_p for the specificity model.

Examples

```
# A validation design of the kind ee_rogan_gladen() takes, with sensitivity
# and specificity now modeled by logistic regression. The design matrix here
# is intercept only, so both models estimate a single log-odds.
set.seed(1)
n <- 500
y_true <- rbinom(n, 1, 0.3)
y_star <- ifelse(y_true == 1, rbinom(n, 1, 0.9), 1 - rbinom(n, 1, 0.85))
r <- c(rep(0, 200), rep(1, 300))

# The gold standard is observed only in the validation sample, so the main
# study positions carry a 0 placeholder, as on ee_rogan_gladen().
y <- ifelse(r == 0, y_true, 0)
X <- cbind(rep(1, n))

psi <- function(theta) {
  ee_rogan_gladen_extended(theta, y = y, y_star = y_star, r = r, X = X)
}

m <- m_estimate(stacked_equations = psi, init = c(0.5, 1, 1))

# Corrected prevalence, then the sensitivity and specificity intercepts
coef(m)
```

---

ee_survival_model *Estimating equation for parametric survival models*

Description

Returns a p-by-n matrix of estimating equation contributions for parametric survival models. Supports exponential, Weibull, and Gompertz distributions.

Usage

```
ee_survival_model(theta, time, event, distribution)
```

Arguments

theta	Numeric vector of distribution parameters. For exponential, a single parameter (lambda). For Weibull and Gompertz, two parameters (lambda, gamma).
time	Numeric vector of n observed (possibly censored) times.
event	Numeric vector of n event indicators (1 = event, 0 = censored).
distribution	Character string: "exponential", "weibull", or "gompertz".

Details

The estimating equations are based on the score equations of the corresponding parametric survival model, accounting for right censoring. For event observations, the contribution comes from the log-density; for censored observations, the contribution comes from the log-survival function.

Value

A p-by-n matrix where p is the number of parameters. The row is named `lambda` for the exponential distribution, and the rows are named `lambda` and `gamma` for the Weibull and Gompertz distributions.

Examples

```
# Weibull survival times for 45 women with breast cancer, with no covariates.
# The default rootSolve solver does not converge here, so nleqslv is used.
psi <- function(theta) {
  ee_survival_model(
    theta,
    time = breast_cancer$times,
    event = breast_cancer$delta,
    distribution = "weibull"
  )
}
m <- m_estimate(
  stacked_equations = psi,
  init = c(0.1, 0.1),
  solver = "nleqslv"
)

# The first parameter is the scale, the second the shape. A shape near 1
# means the Weibull fits about as well as the simpler exponential.
coef(m)
```

`ee_tobit` *Estimating equation for Tobit regression (Type I)*

Description

Handles left and/or right censored outcomes using standard normal PDF/CDF. Theta is `(beta, log(sigma))`, a vector of length `b + 1`.

Usage

```
ee_tobit(
  theta,
  X,
  y,
  lower = NULL,
```

```
  upper = NULL,
  weights = NULL,
  offset = NULL
)
```

Arguments

theta	Numeric vector of length b + 1.
X	Numeric n-by-b design matrix.
y	Numeric vector of n observed (possibly censored) outcome values.
lower	Numeric lower censoring limit, or NULL (default, no left censoring).
upper	Numeric upper censoring limit, or NULL (default, no right censoring).
weights	Optional numeric vector of n weights. Default NULL.
offset	Optional numeric vector of n offsets. Default NULL.

Value

A (b+1)-by-n matrix. The rows are named X_1 through X_b for the columns of X, and the final row is named log_sigma.

Examples

```
# A latent outcome observed only down to zero, so the negative values are
# left censored at the limit.
set.seed(123)
n <- 200
X <- cbind(1, rnorm(n))
y <- pmax(1 + 0.5 * X[, 2] + rnorm(n), 0)

psi <- function(theta) ee_tobit(theta, X = X, y = y, lower = 0)

# The last parameter is log(sigma), started at the log of the observed
# standard deviation.
m <- m_estimate(
  stacked_equations = psi,
  init = c(mean(y), 0, log(sd(y)))
)
coef(m)
```

---

estimate *Estimate parameters and sandwich variance*

Description

Solves the estimating equations for the parameter vector theta and computes the empirical sandwich variance estimator.

Usage

```
estimate(
  object,
  solver = NULL,
  maxiter = 5000,
  tolerance = 1e-09,
  deriv_method = "capprox",
  dx = 1e-09,
  allow_pinv = TRUE,
  ...
)

## S7 method for class <deli::GMMEstimator>
estimate(
  object,
  solver = NULL,
  maxiter = 5000,
  tolerance = 1e-09,
  deriv_method = "capprox",
  dx = 1e-09,
  allow_pinv = TRUE,
  ...
)
```

Arguments

`object`	An `MEstimator` or `GMMEstimator` object.
`solver`	Character string specifying the solver algorithm, or a custom function. When `NULL` (default), uses `"rootSolve"` for `MEstimator` (`rootSolve::multiroot()`) and `"BFGS"` for `GMMEstimator` (`stats::optim()`). Other options for `MEstimator`: `"lm"`, the Levenberg-Marquardt algorithm (`minpack.lm::nls.lm()`), which mirrors the default solver of Python `delicatessen` (`scipy.optimize.root(method = "lm")`); and `"nleqslv"` (uses `nleqslv::nleqslv()`). A custom function must accept `stacked_equations` and `init` arguments and return the solved theta vector. Whichever solver is used, the point it returns is judged against the estimating equations themselves and a warning is raised when they are not solved there. A custom function reports no status of its own, so its point is judged exactly as a built-in solver's is. Two `GMMEstimator` fits are exceptions. An over-identified one cannot drive every moment to zero, so its moments are read against the J-statistic instead, which `GMMEstimator()` describes. A `subset` one is judged neither way and does not warn, because what it minimizes is neither driven to zero nor read against that statistic; see `subset` in `gmm_estimate()` for what such a fit estimates, and inspect `rowSums()` of the estimating functions at the returned values. `rootSolve::multiroot()` cannot run inside itself, so a fit whose estimating function fits a second M-estimator cannot leave both on the default solver; that is refused rather than attempted, and naming `"nleqslv"` for either of the two fits resolves it. That refusal covers the solves deli makes and cannot cover a solve made inside

a custom function, whose body deli does not read. **A custom solver that calls `rootSolve::multiroot()` itself must not be used while another `rootSolve` solve is running.** The inner call overwrites the environment the outer one is still using, and neither reports it: the outer solve was measured returning the inner fit's estimates under nothing louder than a generic non-convergence warning. Give either the outer fit or the custom solver a different algorithm, `"nleqslv"` or `"lm"`, so that no two `rootSolve` solves are open at once.

What every solver is given is the estimating equations summed across the observations, since a root of the system is a root of those sums, and the bread is differentiated from the same reduction. A `GMMEstimator` minimizes a quadratic form in them instead and reduces them just as often. Both reductions are derived from the full p-by-n return unless the estimator carries a `summed_equations` property, which replaces them with the closed form the caller supplied and leaves the full evaluation to the meat and to the validation at the starting values; see `MEstimator()` and `GMMEstimator()` for what each class reduces and what it cannot.

A `"rootSolve"` solve discards anything printed to standard output while it runs, because `rootSolve::multiroot()` prints Fortran diagnostics there that report on its internals rather than on the fit. The sink covers the whole solve rather than the solver alone, so a `print()` or a `cat()` in the estimating function is discarded along with them and its output does not reach the console. Warnings and messages are untouched, since neither goes to standard output, so `message()` traces an estimating function under every solver.

`maxiter` Integer maximum iterations for the solver (default 5000). Must be a single positive whole number, which is checked whichever solver is in force.

`tolerance` Numeric tolerance for the solver (default 1e-9).

`deriv_method` Character string for the derivative method used to compute the bread matrix. One of `"capprox"` (central difference), `"fapprox"` (forward difference), `"bapprox"` (backward difference), or `"exact"` (forward-mode automatic differentiation). Default `"capprox"`. Exact differentiation removes the finite-difference step size but requires that the estimating equation is composed of operations the autodiff supports (see `auto_differentiation()`). For post-estimation transforms and the survival prediction helpers, exact differentiation is also available through `delta_method()`.

`dx` Numeric step size for numerical differentiation; ignored when `deriv_method = "exact"` (default 1e-9). Must be a single positive finite number, which is checked whichever `deriv_method` is in force. The step is absolute and is floored at the floating-point resolution of each estimate, so a large parameter magnitude cannot silently reduce it to nothing; see `approx_differentiation()`.

`allow_pinv` Logical. Use pseudo-inverse if bread is singular? Default `TRUE`.

`...` Not used. Must be empty, so a name that is not one of the documented arguments is an error rather than silently ignored.

Details

The estimates and every matrix built from them carry parameter names, taken from the first of two channels that supplies them. Names on `init` come first: a parameter the caller named keeps

that name, and one left unnamed is numbered by position, so `init = c(mu = 0, 1)` gives `c("mu", "theta_2")`. Where `init` carries no names at all, the row names of the estimating functions are read instead. That is how a `stacked_equations` function names the parameters it defines, since the estimator otherwise sees only an opaque closure returning a matrix. The formula interface names `init` from the model matrix columns where it can account for its length, one parameter per design column or one more for the parameter the equation appends, so the row names carry the function interface and every formula fit of another length. `ee_gformula()` is one of those: its extra parameter leads rather than trails, so the formula interface leaves its `init` unnamed and the fit takes the labels the equation writes on its own rows.

Row names are read only when they label every parameter distinctly: one name per parameter, none empty, none missing, and no two alike. An incomplete or repetitive set is discarded rather than patched up, because it is usually an accident of how the stack was built. `rbind()` pads the rows of an unlabeled block with empty strings, and `t(X * resid)` on a design whose intercept column has no name produces the same shape; `rbind()` also names each row after the variable that supplied it, so two blocks that each begin with a variable of the same name repeat a label. The count has to match as well, which is what keeps an over-identified `GMMEstimator` out: its rows are moment conditions and outnumber the parameters, so their labels describe the equations. Where neither channel applies, the parameters are numbered `theta_1` through `theta_p`.

Row names survive exact differentiation. Assigning them is ignored while a value carries derivatives, since the labels are read from the plain evaluation at the solved values. `rownames<-` hands its work to `dimnames<-`, which is a generic, so deli registers the setter there and an estimating function can label its rows from anywhere.

Many of the built-in estimating equations name their own rows, so a fit that passes one of them a wholly unnamed `init` comes back labeled rather than numbered. Each documents its labels under **Value**. The rule above still governs a stack built from them: two blocks that name the same parameter, as two `ee_ipw()` blocks do, repeat a label and the fit is numbered instead, and one named block stacked with an unnamed one is incomplete and is numbered as well. Name `init` where a stack needs labels the blocks cannot agree on.

Value

A modified `MEstimator` object with populated `theta`, `bread`, `meat`, `variance`, and `asymptotic_variance` properties.

Examples

```
psi <- function(theta) {
  y <- c(1, 2, 3, 4, 5)
  matrix(y - theta[1], nrow = 1)
}
m <- MEstimator(stacked_equations = psi, init = 0) |>
  estimate()
coef(m)
```

get_tested *GetTested randomized trial data*

Description

Data from the GetTested randomized trial, used to demonstrate standardization and inverse probability weighting for treatment effects.

Usage

```
get_tested
```

Format

A data frame.

References

Morris TP, Walker AS, Williamson EJ, & White IR. (2022). Planning a method for covariate adjustment in randomised trials. *BMJ*, 376.

GMMEstimator *GMM Estimator*

Description

S7 class for Generalized Method of Moments (GMM) estimation via minimization of estimating equations with empirical sandwich variance estimation.

Usage

```
GMMEstimator(
  stacked_equations,
  init,
  subset = NULL,
  finite_correction = NULL,
  overid_maxiter = 200L,
  overid_tolerance = 1e-09,
  summed_equations = NULL,
  check_summed_equations = TRUE
)
```

Arguments

`stacked_equations`
: A function that takes a numeric vector `theta` and returns a p-by-n matrix of estimating equation contributions, where p is the number of estimating equations and n is the number of observations. The number of equations p must be greater than or equal to the number of parameters (length of `init`). Row names on that matrix name the parameters when `init` has none and there is exactly one row per parameter, so an over-identified system is numbered instead; see `estimate()`.

`init`
: Numeric vector of initial parameter values for the minimization algorithm. Names on it label the parameters and take precedence over the row names of `stacked_equations`.

`subset`
: Integer vector of parameter indices to solve for, or `NULL` (default) to solve for all parameters. Indices are 1-based; parameters not listed are held fixed at their `init` values while the rest are solved. The objective is a quadratic form in every moment condition and `subset` changes only which parameters are free to move within it, so the conditions outside the subset are still summed in and still pull on the free parameters. A subset fit is therefore not the fit of the subset equations on their own, which is what `MEstimator()` and `m_estimate()` return, and the same stack and the same `subset` give the two different values. The variance estimator ignores `subset`.

`finite_correction`
: Character string for finite-sample correction (e.g., `"HC1"`), or `NULL` (default) for no correction.

`overid_maxiter`
: Integer maximum iterations for the two-step iterative procedure for over-identified problems. Default `200L`. The update converges linearly rather than quadratically, so a well-identified system commonly needs tens of passes to reach `overid_tolerance` and a weakly identified one can need hundreds.

`overid_tolerance`
: Numeric tolerance for convergence of the two-step iterative procedure for over-identified problems. Default `1e-9`.

`summed_equations`
: A function that takes a numeric vector `theta` and returns the length-p vector of row sums of `stacked_equations` at `theta`, or `NULL` (default) to derive those sums from the full p-by-n return.

  Two of the three things a GMM fit does with the estimating functions want nothing but those sums. The objective is a quadratic form in the mean moments, $\bar{g}(\theta)'W\bar{g}(\theta)$, so every evaluation the minimizer makes reduces the whole p-by-n matrix to the reduction's own output; the bread is the Jacobian of the same sums. Both take a supplied reduction and skip the matrix.

  The third does not. The two-step weight matrix is the inverse of the covariance of the moment conditions, which is a cross-product of the per-observation contributions, so each pass of the update over an over-identified system evaluates `stacked_equations` in full whatever this property holds. So do the validation at the starting values and the meat. A just-identified fit runs no weight update, so it makes exactly those two full evaluations.

  Under `deriv_method = "exact"` the reduction is called with a tangent-carrying `theta`, so it must be written in operations that carry derivatives: `t(X) %*% r`

does, and base::crossprod() does not. See auto_differentiation() for which operations carry a tangent and where.

Anything that is neither NULL nor a function is refused here, with an error carrying the class deli_summed_equations_error. So is a return at the estimated values that is not numeric or holds fewer values than the system has moment conditions, which estimate() reads.

gmm_estimate() and m_estimate() do not offer it, for the reason MEstimator() gives: both build stacked_equations themselves, so the moment conditions a reduction would have to match are ones the caller never writes.

check_summed_equations

Logical. When TRUE (default) and summed_equations was supplied, its value at the estimated values is compared against the row sums of the one full evaluation the meat is built from, and a disagreement raises an error carrying the class deli_summed_equations_disagree. See MEstimator() for what the comparison catches, what it cannot, and what setting it to FALSE leaves behind.

Value

A GMMEstimator S7 object. Call estimate() to minimize the estimating equations and compute the sandwich variance.

Moment quality of an over-identified fit

A just-identified system has as many moment conditions as parameters, so the moments vanish at a solution and the size of what is left over says whether the fit succeeded. An over-identified system has no such reading: no value of the parameters drives every condition to zero, and a residual moment is expected rather than diagnostic. Hansen's J-statistic is the reading that is available there. It is n times the GMM objective at the minimum, $J = n\bar{g}(\hat{\theta})'W\bar{g}(\hat{\theta})$, where $\bar{g}$ averages the moment conditions over the observations and W is the weight matrix the fit finished with. Under correct specification it is asymptotically chi-squared on as many degrees of freedom as the system has moment conditions beyond parameters, so its size can be judged against a reference distribution rather than against the scale of the data.

estimate() records it in the j_statistic property of an over-identified fit, and summary() reports it with its degrees of freedom and its P-value. A just-identified fit has no degrees of freedom left over and leaves the property NULL; its moments are judged directly instead, as estimate() describes. A subset fit holds the parameters outside the subset at their initial values rather than estimating them, which the reference distribution does not allow for, so it is left NULL too.

A P-value the reference distribution all but rules out warns with the class deli_gmm_moments_rejected, which usually means the moment conditions cannot all hold at one value of the parameters. The weight matrix is what makes J comparable across problems, so the warning is raised only where the two-step update settled: a fit that exhausted overid_maxiter has already warned about that, and its J has no reference distribution to be judged against. The property still records the statistic in that case, as it does for overid_maxiter = 0, which leaves the identity weight matrix in place and so leaves J an unstandardized sum of squared moments.

The reading J cannot make is the opposite failure. Moment conditions that are linearly dependent, one of them repeating what the others already say, leave the covariance the weight matrix inverts singular, and the update falls through to the pseudo-inverse; the fit that comes back is the fit of the independent conditions alone. J is silent about it, because a condition the others account for agrees

with them wherever the parameters sit and so adds nothing for J to measure, which drives J toward zero rather than away from it. That case warns with the class `deli_gmm_moments_dependent` instead, naming the conditions the factorization found redundant.

Examples

```
# The constructor builds the estimator and `estimate()` solves it, so an
# object that has not been through `estimate()` reports only what it was
# given. `gmm_estimate()` does both steps in one call.
y <- c(1, 2, 3, 4, 5)
psi <- function(theta) {
  matrix(y - theta[1], nrow = 1)
}
GMMEstimator(stacked_equations = psi, init = 0)

# One moment condition for one parameter is just-identified, so the minimizer
# lands where `MEstimator()` would have found the root.
GMMEstimator(stacked_equations = psi, init = 0) |>
  estimate()

# A Poisson mean is identified twice over, by the mean and by the variance,
# so these two moment conditions estimate one parameter and the system is
# over-identified. That is the case `MEstimator()` cannot solve, and the case
# the `overid_maxiter` and `overid_tolerance` properties govern: they stop
# the two-step weight matrix update that reconciles the two conditions.
set.seed(42)
counts <- rpois(200, lambda = 3)

psi_pois <- function(theta) {
  rbind(
    counts - theta[1],
    (counts - theta[1])^2 - theta[1]
  )
}

g <- GMMEstimator(stacked_equations = psi_pois, init = 1) |>
  estimate()

# With more conditions than parameters neither is solved exactly. The weight
# matrix is what decides how the disagreement between them is split.
summary(g)
```

`gmm_estimate` *One-step GMM estimation*

Description

Creates a `GMMEstimator` and estimates it in one call. Supports both a formula interface and a function interface, parallel to `m_estimate()`.

Usage

```
gmm_estimate(stacked_equations, ...)

## S3 method for class 'formula'
gmm_estimate(
  stacked_equations,
  data,
  .ee,
  ...,
  init = NULL,
  subset = NULL,
  finite_correction = NULL,
  solver = NULL,
  maxiter = 5000,
  tolerance = 1e-09,
  deriv_method = "capprox",
  dx = 1e-09,
  allow_pinv = TRUE,
  overid_maxiter = 200L,
  overid_tolerance = 1e-09
)

## Default S3 method:
gmm_estimate(
  stacked_equations,
  ...,
  init,
  subset = NULL,
  finite_correction = NULL,
  solver = NULL,
  maxiter = 5000,
  tolerance = 1e-09,
  deriv_method = "capprox",
  dx = 1e-09,
  allow_pinv = TRUE,
  overid_maxiter = 200L,
  overid_tolerance = 1e-09
)
```

Arguments

`stacked_equations`

A formula or a function. When a formula, `data` and `.ee` must also be provided. When a function, it should take a numeric vector `theta` and return a p-by-n matrix, whose row names name the parameters when `init` has none; see `estimate()`. A two-level factor or character response is converted to a 0/1 indicator against its first level; an `offset()` term in the formula is passed to `.ee` through its `offset` argument.

`...` For the formula interface, additional arguments passed to `.ee`. These are evaluated with tidy evaluation in the context of `data`, so column names can be used directly (e.g., `event = status`). If the model frame drops rows for missing data, any such argument that spans the full data is subset to the same rows so it stays aligned with the design matrix and response. The function interface forwards nothing, so it requires `...` to be empty.

`data` A data frame (required when `stacked_equations` is a formula).

`.ee` An estimating equation function that accepts `theta`, `X`, and the response as its third argument, plus optionally additional arguments (required when `stacked_equations` is a formula). The formula response is passed positionally, so it reaches whatever the function calls that argument (`y` for ee_regression or ee_glm, `time` for ee_aft). An equation whose arguments leave any of those nowhere to go cannot be driven by a formula and is refused before anything is estimated: ee_survival_model takes no design matrix, so it is fitted through the function interface instead.

`init` Numeric vector of initial parameter values. When `NULL` (default) and using the formula interface, a zero vector with names from the model matrix columns is generated automatically. Names on it label the parameters and take precedence over the row names of `stacked_equations`. An explicit `init` with no names of its own takes the same model matrix names on the formula interface, together with the parameter the estimating equation estimates beyond the design coefficients (`log_shape` for ee_glm with `"gamma"`, `log_dispersion` with `"negative_binomial"`, `log_inv_scale` for a non-exponential ee_aft, `log_sigma` for ee_tobit, and `log_phi` for ee_beta_regression). An `init` of any other length is left unnamed, which passes the labeling to the row names of the estimating functions; see `estimate()` for that channel and for when the parameters are numbered instead.

`subset` Integer vector of parameter indices to solve for, or `NULL` (default) to solve for all parameters. Indices are 1-based; parameters not listed are held fixed at their `init` values while the rest are solved. The objective is a quadratic form in every moment condition and `subset` changes only which parameters are free to move within it, so the conditions outside the subset are still summed in and still pull on the free parameters. A subset fit is therefore not the fit of the subset equations on their own, which is what `MEstimator()` and `m_estimate()` return, and the same stack and the same `subset` give the two different values. The variance estimator ignores `subset`.

`finite_correction`
Character string for finite-sample correction (e.g., `"HC1"`), or `NULL` (default) for no correction. When set, the meat matrix is rescaled and inference switches to the t-distribution with `df = n_obs - n_params`. Passed straight through to the estimator constructor.

`solver` Character string or function for the solver. Default `NULL` uses `"rootSolve"` for M-estimation and `"BFGS"` for GMM. See `estimate()` for the full list of solvers and for how the returned point is judged against the estimating equations.

`maxiter` Integer maximum iterations (default 5000). Must be a single positive whole number.

`tolerance` Numeric convergence tolerance (default 1e-9).

`deriv_method` Character string for numerical differentiation method (default `"capprox"`).

`dx` Numeric step size for differentiation (default 1e-9). Must be a single positive finite number, which is checked whichever `deriv_method` is in force. The step is absolute and is floored at the floating-point resolution of each estimate, so a large parameter magnitude cannot silently reduce it to nothing; see `approx_differentiation()`.

`allow_pinv` Logical. Use pseudo-inverse if bread is singular? Default `TRUE`.

`overid_maxiter` Integer maximum iterations for the two-step iterative procedure for over-identified problems. Default `200L`. The update converges linearly rather than quadratically, so a well-identified system commonly needs tens of passes to reach `overid_tolerance` and a weakly identified one can need hundreds.

`overid_tolerance` Numeric tolerance for convergence of the two-step iterative procedure. Default `1e-9`.

Details

Both interfaces place `...` ahead of `init`, `subset`, `tolerance`, and the rest of the settings, so each of those must be named in full: R does not partially match a supplied name against an argument that follows `....` `estimate()` and the inference generics take `...` last, so they still accept R's usual abbreviations.

What becomes of a name that matches no argument depends on the interface. The function interface has no estimating equation to forward `...` to, so it requires `...` to be empty and reports an unrecognized name as an error rather than silently ignoring it. The formula interface forwards `...` to `.ee` and matches each name against that function's arguments exactly, naming the argument a refused name was probably meant for. A name that merely abbreviates one is refused too, rather than partially matched to it, since a fit that quietly took a misspelling for `weights` reports different numbers and says nothing about it. An `.ee` that takes `...` of its own accepts any name.

Value

A fitted `GMMEstimator` object.

Moment quality of an over-identified fit

A just-identified system has as many moment conditions as parameters, so the moments vanish at a solution and the size of what is left over says whether the fit succeeded. An over-identified system has no such reading: no value of the parameters drives every condition to zero, and a residual moment is expected rather than diagnostic. Hansen's J-statistic is the reading that is available there. It is n times the GMM objective at the minimum, $J = n\bar{g}(\hat{\theta})' W \bar{g}(\hat{\theta})$, where $\bar{g}$ averages the moment conditions over the observations and W is the weight matrix the fit finished with. Under correct specification it is asymptotically chi-squared on as many degrees of freedom as the system has moment conditions beyond parameters, so its size can be judged against a reference distribution rather than against the scale of the data.

`estimate()` records it in the `j_statistic` property of an over-identified fit, and `summary()` reports it with its degrees of freedom and its P-value. A just-identified fit has no degrees of freedom left over and leaves the property `NULL`; its moments are judged directly instead, as `estimate()` describes. A `subset` fit holds the parameters outside the subset at their initial values rather than estimating them, which the reference distribution does not allow for, so it is left `NULL` too.

A P-value the reference distribution all but rules out warns with the class `deli_gmm_moments_rejected`, which usually means the moment conditions cannot all hold at one value of the parameters. The weight matrix is what makes J comparable across problems, so the warning is raised only where the two-step update settled: a fit that exhausted `overid_maxiter` has already warned about that, and its J has no reference distribution to be judged against. The property still records the statistic in that case, as it does for `overid_maxiter = 0`, which leaves the identity weight matrix in place and so leaves J an unstandardized sum of squared moments.

The reading J cannot make is the opposite failure. Moment conditions that are linearly dependent, one of them repeating what the others already say, leave the covariance the weight matrix inverts singular, and the update falls through to the pseudo-inverse; the fit that comes back is the fit of the independent conditions alone. J is silent about it, because a condition the others account for agrees with them wherever the parameters sit and so adds nothing for J to measure, which drives J toward zero rather than away from it. That case warns with the class `deli_gmm_moments_dependent` instead, naming the conditions the factorization found redundant.

Examples

```
# Two instruments for a single treatment effect, confounded by an
# unmeasured U. Two moment conditions for one parameter leave the system
# over-identified, which is the case GMM is for: `m_estimate()` requires one
# estimating equation per parameter.
set.seed(42)
n <- 200
d <- data.frame(Z1 = rbinom(n, 1, 0.5), Z2 = rnorm(n))
U <- rnorm(n)
d$A <- 0.5 * d$Z1 + 0.3 * d$Z2 + U + rnorm(n)
d$Y <- 2 * d$A - U + rnorm(n)

# One moment condition per instrument: an instrument should be uncorrelated
# with the residual of the outcome on the treatment.
ee_iv_moments <- function(theta, X, y, Z) {
  t(Z * (y - as.numeric(X %*% theta)))
}

# The formula interface reads the design and the starting values off the
# model and passes anything else, here the instruments, on to `.ee`.
g <- gmm_estimate(
  Y ~ A - 1,
  data = d,
  .ee = ee_iv_moments,
  Z = cbind(Z1, Z2)
)

# The instruments move the estimate toward the treatment effect of 2 that
# generated the data. The least-squares fit of Y on A ignores the confounding
# and stays further from it.
coef(g)
coef(lm(Y ~ A - 1, data = d))

# The function interface takes a `stacked_equations` closure instead, for a
# system no formula describes. It has no design to read parameter names from,
```

```
# so names on `init` are what label the results.
psi_iv <- function(theta) {
  residual <- d$Y - theta[1] * d$A
  rbind(d$Z1 * residual, d$Z2 * residual)
}

g2 <- gmm_estimate(
  stacked_equations = psi_iv,
  init = c(effect = 0)
)
coef(g2)
```

`identity_transform` *Identity transformation*

Description

Returns the input unchanged. Used as a no-op transformation in contexts that accept an arbitrary transformation function.

This is `base::identity()` under a name that does not mask it. The two are interchangeable everywhere, including under exact differentiation (`deriv_method = "exact"`), since returning the argument untouched also returns any tangent it carries. `identity_transform()` exists so that code translated from Python delicatessen, where the function is called `identity()`, can keep its shape. In R code, prefer `identity()`.

Usage

```
identity_transform(value)
```

Arguments

`value`	Any value.

Value

The input `value`, unchanged.

See Also

`base::identity()`, which does the same thing and is the idiomatic choice in R. `inverse_logit()` has the full list of deli utilities and their base R counterparts.

Examples

```
identity_transform(42)
identity_transform(c(1, 2, 3))

# The same value as base R, and the same object
identical(identity_transform(c(1, 2, 3)), identity(c(1, 2, 3)))
```

inderjit *Inderjit dose-response data*

Description

Example data from Inderjit et al. (2002) on the dose-response of herbicide on perennial ryegrass growth.

Usage

```
inderjit
```

Format

A data frame with 24 rows and 2 columns:

response Ryegrass root length

dose Herbicide dose

References

Inderjit, Streibig JC, & Olofsdotter M. (2002). Joint action of phenolic acid mixtures and its significance in allelopathy research. *Physiologia Plantarum*, 114(3), 422-428.

influence_functions *Influence functions for M-Estimator*

Description

Computes the influence function values for individual observations:

$$IF(O_i;\theta) = B_n(\theta)^{-1}\psi(O_i;\theta)$$

This function mirrors `m.influence_functions()` in Python delicatessen, so code translated from Python can keep its shape. There is no base R accessor for influence function values, so this is the interface for them in deli as well.

Usage

```
influence_functions(object, allow_pinv = TRUE, ...)
```

Arguments

`object`	A fitted `MEstimator` object (after calling `estimate()`).
`allow_pinv`	Logical. Use pseudo-inverse if bread is singular? Default `TRUE`.
`...`	Not used. Must be empty, so a name that is not one of the documented arguments is an error rather than silently ignored.

Value

An n-by-p matrix of influence function values, where n is the number of observations and p is the number of parameters. Columns are named for the parameters, as in `coef()`. The rows take whatever labels the estimating function put on the columns of its own return, so a fit whose estimating function collapses its contributions with `aggregate_efuncs()` has one row per group, labeled with the group value, and every row is that group's influence rather than an observation's.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                  model = "linear")

# One row per observation, showing its contribution to each estimate
head(influence_functions(fit))
```

---

inverse_logit *Inverse logistic transformation*

Description

Transforms log-odds into probabilities: $1/(1 + \exp(-x))$.

This is the equivalent of `stats::plogis()` and returns identical values for numeric input, except that it carries derivatives. Exact differentiation (`deriv_method = "exact"`) propagates a tangent alongside each value through the arithmetic and `Math` group generics. `inverse_logit()` is written as `1 / (1 + exp(-logodds))`, so those group generic methods carry a tangent through it; `plogis()` hands its argument to compiled code without dispatching, and errors on a tangent-carrying argument. Use `inverse_logit()` inside estimating equations and inside transforms passed to `delta_method()`, and `plogis()` for ordinary numeric work.

Usage

```
inverse_logit(logodds)
```

Arguments

`logodds`	A numeric value or vector of log-odds.

Value

Numeric probability values.

Exact differentiation

`deriv_method = "exact"` is forward-mode automatic differentiation: it replaces each value with an object carrying both the value and its derivative. deli supports those objects through S3 methods: the `Ops`, `Math`, and `Summary` group generics, plus non-group methods such as `[`, `%*%`, `t()`, `c()`, and `mean()`. `standard_normal_cdf()`, `standard_normal_pdf()`, `deli_polygamma()`, and `deli_digamma()` recognize a tangent-carrying argument themselves and apply their own analytic rule. Support within the group generics is partial; see `vignette("getting-started")` for the operations deli differentiates.

`plogis()`, `qlogis()`, `pnorm()`, `dnorm()`, and `psigamma()` take none of these paths. Each hands its argument straight to compiled code through `.Call()` or `.Internal()` without dispatching, so the tangent reaches C code that requires a plain number. deli catches the resulting failure and raises its own error, naming the function that stopped the computation and the deli function to write in its place. The same applies to every other distribution function in `stats`, such as `qnorm()`, which is named in the error even though deli exports no counterpart for it.

Each deli utility below returns the same values as its base R counterpart for numeric input. What separates them is whether the counterpart survives exact mode:

deli function	base R counterpart	base R under `deriv_method = "exact"`
`inverse_logit()`	`stats::plogis()`	errors
`logit()`	`stats::qlogis()`	errors
`standard_normal_cdf()`	`stats::pnorm()`	errors
`standard_normal_pdf()`	`stats::dnorm()`	errors
`deli_polygamma()`	`base::psigamma()`	errors
`deli_digamma()`	`base::digamma()`	works
`identity_transform()`	`base::identity()`	works

For the first five rows, use the deli function inside estimating equations and inside transforms passed to `delta_method()`, and the base R function everywhere else: simulating data, post-fit display, plain numeric work. For the last two rows the base R function is usable everywhere, because `digamma()` is a `Math` group member with a tangent rule and `identity()` passes its argument through untouched.

The polygamma row is the one place where the arguments do not line up. `deli_polygamma(n, x)` takes the derivative order first and `psigamma(x, deriv = n)` takes it second, so a positional substitution between the two computes a different quantity and raises no error.

See Also

`stats::plogis()`, the base R equivalent for ordinary numeric work, and `logit()`, which inverts this transformation.

Examples

```
inverse_logit(0)
inverse_logit(c(-2, 0, 2))

# The same values as the base R counterpart
all.equal(inverse_logit(c(-2, 0, 2)), plogis(c(-2, 0, 2)))
```

```
m <- m_estimate(
  vs ~ mpg,
  data = mtcars,
  .ee = ee_regression,
  model = "logistic"
)

# Variance of the fitted probability at mpg = 20. Writing
# `plogis(theta[1] + theta[2] * 20)` here instead would error, because exact
# differentiation hands the transform a tangent-carrying argument.
delta_method(
  m,
  transform = function(theta) inverse_logit(theta[1] + theta[2] * 20),
  deriv_method = "exact"
)
```

`lau_wihs` *Lau WIHS HIV/CD4 data*

Description

CD4 T cell count data from the Women's Interagency HIV Study (WIHS), used to demonstrate sensitivity analysis for missing data.

Usage

```
lau_wihs
```

Format

A data frame with columns: id, black, age, cd4, cd41-cd44.

References

Cole SR, Zivich PN, Edwards JK, Shook-Sa BE, & Hudgens MG. (2023). Sensitivity analyses for means or proportions with missing outcome data. *Epidemiology*, 34(5), 645-651.

`logit` *Logistic transformation*

Description

Transforms probabilities into log-odds: $\log(p/(1-p))$.

This is the equivalent of `stats::qlogis()` and returns identical values for numeric input, except that it carries derivatives. Exact differentiation (`deriv_method = "exact"`) propagates a tangent alongside each value through the arithmetic and `Math` group generics. `logit()` is written as `log(prob / (1 - prob))`, so those group generic methods carry a tangent through it; `qlogis()` hands its argument to compiled code without dispatching, and errors on a tangent-carrying argument. Use `logit()` inside estimating equations and inside transforms passed to `delta_method()`, and `qlogis()` for ordinary numeric work.

Usage

```
logit(prob)
```

Arguments

`prob`	A numeric value or vector of probabilities.

Value

Numeric log-odds values.

Exact differentiation

`deriv_method = "exact"` is forward-mode automatic differentiation: it replaces each value with an object carrying both the value and its derivative. deli supports those objects through S3 methods: the `Ops`, `Math`, and `Summary` group generics, plus non-group methods such as `[`, `%*%`, `t()`, `c()`, and `mean()`. `standard_normal_cdf()`, `standard_normal_pdf()`, `deli_polygamma()`, and `deli_digamma()` recognize a tangent-carrying argument themselves and apply their own analytic rule. Support within the group generics is partial; see `vignette("getting-started")` for the operations deli differentiates.

`plogis()`, `qlogis()`, `pnorm()`, `dnorm()`, and `psigamma()` take none of these paths. Each hands its argument straight to compiled code through `.Call()` or `.Internal()` without dispatching, so the tangent reaches C code that requires a plain number. deli catches the resulting failure and raises its own error, naming the function that stopped the computation and the deli function to write in its place. The same applies to every other distribution function in `stats`, such as `qnorm()`, which is named in the error even though deli exports no counterpart for it.

Each deli utility below returns the same values as its base R counterpart for numeric input. What separates them is whether the counterpart survives exact mode:

deli function	base R counterpart	base R under `deriv_method = "exact"`
`inverse_logit()`	`stats::plogis()`	errors

`logit()`	`stats::qlogis()`	errors
`standard_normal_cdf()`	`stats::pnorm()`	errors
`standard_normal_pdf()`	`stats::dnorm()`	errors
`deli_polygamma()`	`base::psigamma()`	errors
`deli_digamma()`	`base::digamma()`	works
`identity_transform()`	`base::identity()`	works

For the first five rows, use the deli function inside estimating equations and inside transforms passed to `delta_method()`, and the base R function everywhere else: simulating data, post-fit display, plain numeric work. For the last two rows the base R function is usable everywhere, because `digamma()` is a `Math` group member with a tangent rule and `identity()` passes its argument through untouched.

The polygamma row is the one place where the arguments do not line up. `deli_polygamma(n, x)` takes the derivative order first and `psigamma(x, deriv = n)` takes it second, so a positional substitution between the two computes a different quantity and raises no error.

See Also

`stats::qlogis()`, the base R equivalent for ordinary numeric work, and `inverse_logit()`, which inverts this transformation.

Examples

```
logit(0.5)
logit(c(0.1, 0.5, 0.9))

# The same values as the base R counterpart
all.equal(logit(c(0.1, 0.5, 0.9)), qlogis(c(0.1, 0.5, 0.9)))

# The proportion of cars with a manual transmission
m <- m_estimate(
  stacked_equations = function(theta) ee_mean(theta, y = mtcars$am),
  init = 0.5
)

# Variance of the log-odds of that proportion. Writing `qlogis(theta[1])`
# here instead would error, because exact differentiation hands the
# transform a tangent-carrying argument.
delta_method(
  m,
  transform = function(theta) logit(theta[1]),
  deriv_method = "exact"
)
```

`MEstimator` *M-Estimator*

Description

S7 class for M-estimation via solving estimating equations with empirical sandwich variance estimation.

Usage

```
MEstimator(
  stacked_equations,
  init,
  subset = NULL,
  finite_correction = NULL,
  summed_equations = NULL,
  check_summed_equations = TRUE
)
```

Arguments

`stacked_equations`
: A function that takes a numeric vector `theta` and returns a p-by-n matrix of estimating equation contributions, where p is the number of parameters and n is the number of observations. Row names on that matrix name the parameters when `init` has none and every parameter is labeled; see `estimate()`.

`init`
: Numeric vector of initial parameter values for the root-finding algorithm. Names on it label the parameters and take precedence over the row names of `stacked_equations`.

`subset`
: Integer vector of parameter indices to solve for, or `NULL` (default) to solve for all parameters. Indices are 1-based; parameters not listed are held fixed at their `init` values while the rest are solved. The equations outside the subset are set aside along with the parameters they estimate, so the subset parameters are the root of the subset equations alone and the rest of the stack has no say in where they land: give a three-equation linear regression stack `subset = 1L` and the intercept comes back as the mean of the response less what the slopes held at their `init` values account for, because the first equation on its own is the estimating equation for a mean. Held at zero, which is what an unset `init` usually means, they account for nothing and the intercept is the mean of the response itself. `GMMEstimator()` and `gmm_estimate()` read the argument differently, since the GMM objective sums every equation whether the subset lists it or not, so the same stack and the same `subset` give the two different values. The variance estimator ignores `subset`.

`finite_correction`
: Character string for finite-sample correction (e.g., `"HC1"`), or `NULL` (default) for no correction.

summed_equations

A function that takes a numeric vector theta and returns the length-p vector of row sums of stacked_equations at theta, or NULL (default) to derive those sums from the full p-by-n return.

A fit reduces the estimating functions to those sums everywhere but the meat. The solver is given the summed equations to find a root of, and the bread is their Jacobian, so a fit builds the whole p-by-n matrix once per solver evaluation and once or twice more per parameter, all of it for arithmetic that is linear in it. An estimating function whose sums have a closed form, such as the $X^T r$ of a regression score, can supply them here and give both steps only what they use. The meat is unaffected either way: it needs the per-observation contributions and takes the one full evaluation it always took, as does the validation at the starting values.

Under deriv_method = "exact" the reduction is called with a tangent-carrying theta, so it must be written in operations that carry derivatives: t(X) %*% r does, and base::crossprod() does not. See auto_differentiation() for which operations carry a tangent and where.

Anything that is neither NULL nor a function is refused here, with an error carrying the class deli_summed_equations_error. So is a return at the estimated values that is not numeric or does not hold one value per estimating equation, which estimate() reads.

m_estimate() and gmm_estimate() do not offer it. Both build stacked_equations themselves from a formula and the equation named in .ee, so the estimating functions a reduction would have to match are ones the caller never writes; this is for a system assembled by hand.

check_summed_equations

Logical. When TRUE (default) and summed_equations was supplied, its value at the estimated values is compared against the row sums of the one full evaluation the meat is built from, and a disagreement raises an error carrying the class deli_summed_equations_disagree. The comparison costs one call to summed_equations and one reduction of a matrix the fit already holds.

What it is for is a reduction that sums some other system. The solver drives whatever it is given to zero, so such a reduction sends the fit to that other system's root and leaves the bread the Jacobian of one system and the meat the cross-product of another. Set it to FALSE to skip the comparison, which leaves a fit that reports estimates nobody solved for and a covariance with the shape of one and no claim to be one.

A reduction that is a multiple of the right one is not what the comparison sees. It vanishes where the right one does, so the fit lands at the same estimates and both quantities are at rounding where they are compared; the bread comes back as that multiple of the right bread. See compute_sandwich(), whose argument of the same name reads the same comparison at a point the caller supplies.

Value

An MEstimator S7 object. Call estimate() to solve the estimating equations and compute the sandwich variance.

Examples

```
# Estimating equations for the mean
y <- c(1, 2, 3, 4, 5)
psi <- function(theta) {
  matrix(y - theta[1], nrow = 1)
}
m <- MEstimator(stacked_equations = psi, init = 0) |>
  estimate()
coef(m)
summary(m)

# The solver and the bread only ever need the sums of the estimating
# equations, so an equation whose sums have a closed form can supply them
# and leave the whole matrix to the meat
summed <- function(theta) sum(y) - length(y) * theta[1]
MEstimator(stacked_equations = psi, init = 0, summed_equations = summed) |>
  estimate() |>
  summary()
```

---

mroz *Mroz labor force participation data*

Description

Data from Mroz (1987) on married women's labor force participation, used to demonstrate OLS, 2SLS, and instrumental variable estimation.

Usage

```
mroz
```

Format

A data frame with 753 rows.

References

Mroz TA. (1987). The sensitivity of an empirical model of married women's hours of work to economic and statistical assumptions. *Econometrica*, 55(4), 765-799.

`m_estimate` *One-step M-estimation*

Description

Creates an `MEstimator` and estimates it in one call, analogous to how `stats::lm()` creates and fits a model in a single step. Supports both a formula interface (for regression-family estimating equations) and a function interface (for custom estimating equations).

Usage

```
m_estimate(stacked_equations, ...)

## S3 method for class 'formula'
m_estimate(
  stacked_equations,
  data,
  .ee,
  ...,
  init = NULL,
  subset = NULL,
  finite_correction = NULL,
  solver = NULL,
  maxiter = 5000,
  tolerance = 1e-09,
  deriv_method = "capprox",
  dx = 1e-09,
  allow_pinv = TRUE
)

## Default S3 method:
m_estimate(
  stacked_equations,
  ...,
  init,
  subset = NULL,
  finite_correction = NULL,
  solver = NULL,
  maxiter = 5000,
  tolerance = 1e-09,
  deriv_method = "capprox",
  dx = 1e-09,
  allow_pinv = TRUE
)
```

Arguments

`stacked_equations`
: A formula or a function. When a formula, `data` and `.ee` must also be provided. When a function, it should take a numeric vector `theta` and return a p-by-n matrix, whose row names name the parameters when `init` has none; see `estimate()`. A two-level factor or character response is converted to a 0/1 indicator against its first level; an `offset()` term in the formula is passed to `.ee` through its `offset` argument.

`...`
: For the formula interface, additional arguments passed to `.ee`. These are evaluated with tidy evaluation in the context of `data`, so column names can be used directly (e.g., `event = status`). If the model frame drops rows for missing data, any such argument that spans the full data is subset to the same rows so it stays aligned with the design matrix and response. The function interface forwards nothing, so it requires `...` to be empty.

`data`
: A data frame (required when `stacked_equations` is a formula).

`.ee`
: An estimating equation function that accepts `theta`, `X`, and the response as its third argument, plus optionally additional arguments (required when `stacked_equations` is a formula). The formula response is passed positionally, so it reaches whatever the function calls that argument (`y` for ee_regression or ee_glm, `time` for ee_aft). An equation whose arguments leave any of those nowhere to go cannot be driven by a formula and is refused before anything is estimated: ee_survival_model takes no design matrix, so it is fitted through the function interface instead.

`init`
: Numeric vector of initial parameter values. When `NULL` (default) and using the formula interface, a zero vector with names from the model matrix columns is generated automatically. Names on it label the parameters and take precedence over the row names of `stacked_equations`. An explicit `init` with no names of its own takes the same model matrix names on the formula interface, together with the parameter the estimating equation estimates beyond the design coefficients (`log_shape` for ee_glm with `"gamma"`, `log_dispersion` with `"negative_binomial"`, `log_inv_scale` for a non-exponential ee_aft, `log_sigma` for ee_tobit, and `log_phi` for ee_beta_regression). An `init` of any other length is left unnamed, which passes the labeling to the row names of the estimating functions; see `estimate()` for that channel and for when the parameters are numbered instead.

`subset`
: Integer vector of parameter indices to solve for, or `NULL` (default) to solve for all parameters. Indices are 1-based; parameters not listed are held fixed at their `init` values while the rest are solved. The equations outside the subset are set aside along with the parameters they estimate, so the subset parameters are the root of the subset equations alone and the rest of the stack has no say in where they land: give a three-equation linear regression stack `subset = 1L` and the intercept comes back as the mean of the response less what the slopes held at their `init` values account for, because the first equation on its own is the estimating equation for a mean. Held at zero, which is what an unset `init` usually means, they account for nothing and the intercept is the mean of the response itself. `GMMEstimator()` and `gmm_estimate()` read the argument differently, since the GMM objective sums every equation whether the subset lists it or not, so the

same stack and the same `subset` give the two different values. The variance estimator ignores `subset`.

`finite_correction`
: Character string for finite-sample correction (e.g., `"HC1"`), or `NULL` (default) for no correction. When set, the meat matrix is rescaled and inference switches to the t-distribution with `df = n_obs - n_params`. Passed straight through to the estimator constructor.

`solver`
: Character string or function for the solver. Default `NULL` uses `"rootSolve"` for M-estimation and `"BFGS"` for GMM. See `estimate()` for the full list of solvers and for how the returned point is judged against the estimating equations.

`maxiter`
: Integer maximum iterations (default 5000). Must be a single positive whole number.

`tolerance`
: Numeric convergence tolerance (default 1e-9).

`deriv_method`
: Character string for numerical differentiation method (default `"capprox"`).

`dx`
: Numeric step size for differentiation (default 1e-9). Must be a single positive finite number, which is checked whichever `deriv_method` is in force. The step is absolute and is floored at the floating-point resolution of each estimate, so a large parameter magnitude cannot silently reduce it to nothing; see `approx_differentiation()`.

`allow_pinv`
: Logical. Use pseudo-inverse if bread is singular? Default `TRUE`.

Details

Both interfaces place `...` ahead of `init`, `subset`, `tolerance`, and the rest of the settings, so each of those must be named in full: R does not partially match a supplied name against an argument that follows `...`. `estimate()` and the inference generics take `...` last, so they still accept R's usual abbreviations.

What becomes of a name that matches no argument depends on the interface. The function interface has no estimating equation to forward `...` to, so it requires `...` to be empty and reports an unrecognized name as an error rather than silently ignoring it. The formula interface forwards `...` to `.ee` and matches each name against that function's arguments exactly, naming the argument a refused name was probably meant for. A name that merely abbreviates one is refused too, rather than partially matched to it, since a fit that quietly took a misspelling for `weights` reports different numbers and says nothing about it. An `.ee` that takes `...` of its own accepts any name.

Value

A fitted `MEstimator` object with populated `theta`, `variance`, etc. Use `coef()`, `vcov()`, `confint()`, `summary()`, or `tidy()` to extract results.

Examples

```
# Formula interface
m <- m_estimate(mpg ~ wt + hp, data = mtcars,
                .ee = ee_regression, model = "linear")
coef(m)
summary(m)
```

```
# Function interface
y <- c(1, 2, 3, 4, 5)
m2 <- m_estimate(
  stacked_equations = function(theta) matrix(y - theta[1], nrow = 1),
  init = c(mean = 0)
)
coef(m2)
```

---

nsduh *NSDUH substance use survey data*

Description

Data from the National Survey on Drug Use and Health, used to demonstrate causal mediation analysis.

Usage

```
nsduh
```

Format

A data frame with approximately 40,000 rows.

References

Coffman DL, Zhong W. (2021). Assessing mediation using marginal structural models in the presence of confounding and moderation. *Psychological Methods*, 17(4), 532.

---

plogit_predict *Predicted survival measures from a pooled logistic regression model*

Description

Computes predicted survival analysis measures from a pooled logistic regression model at specified time points. Meant to be used after fitting `ee_plogit()` with `MEstimator()`.

Usage

```
plogit_predict(
  theta,
  time,
  event,
  X,
  S = NULL,
  times_to_predict = NULL,
  measure = "survival",
  unique_times = NULL
)
```

Arguments

theta	Numeric vector of estimated parameters from ee_plogit.
time	Numeric vector of n observed (possibly censored) times (same as in ee_plogit).
event	Numeric vector of n event indicators (same as in ee_plogit).
X	Numeric n-by-b design matrix for covariates.
S	Optional time design matrix, with one row per unit-time interval up to the maximum observed time. Default NULL uses disjoint indicators.
times_to_predict	Optional numeric vector of specific times to predict at. Default NULL returns all time steps.
measure	Character string: "survival", "risk", "density", "hazard", or "cumulative_hazard". Default "survival".
unique_times	Optional numeric vector of unique event times. Default NULL. When S is supplied it may only agree with the time grid the function builds; see the section on a supplied time design matrix.

Value

A K-by-n matrix (or selected subset of rows if times_to_predict is specified).

A supplied time design matrix

A supplied S models time parametrically over the unit-time intervals from one to the maximum observed time, one row of S per interval. That grid is the function's to build, and the two arguments that describe it have to agree with it rather than replace it: nrow(S) counts its steps and unique_times, when supplied, names them. A mismatch in either is an error.

Neither can be honored in place of the built grid, because the grid is also the binning of the person-periods ee_plogit() solves on, so predictions on any other grid would come from coefficients that were never fitted to it. A maximum observed time falling between two whole times names no further whole interval, so the grid stops at the last whole one and an S sized past it is an error as well. ee_plogit() validates both arguments the same way, so a grid the equation refuses is not one predictions come back from.

This is a deliberate divergence from Python Delicatessen, which documents unique_times as ignored when a time design matrix is supplied. An ignored argument leaves a caller believing the grid was theirs to choose, so deli validates it instead.

Examples

```
# Bladder tumor recurrence, fit with disjoint indicators for time
W <- cbind(
  novel = collett_bladder$treat - 1,
  as.matrix(collett_bladder[, c("init", "size")])
)
k <- length(unique(collett_bladder$time[collett_bladder$delta == 1]))

psi <- function(theta) {
  ee_plogit(
```

```
    theta,
    X = W,
    time = collett_bladder$time,
    event = collett_bladder$delta
  )
}
m <- m_estimate(
  stacked_equations = psi,
  init = c(rep(0, ncol(W)), -4, rep(0, k - 1))
)

# Rows are the requested times and columns are individuals, so this shows
# disease-free survival at 12, 24, and 59 months for the first five people.
plogit_predict(
  coef(m),
  time = collett_bladder$time,
  event = collett_bladder$delta,
  X = W,
  times_to_predict = c(12, 24, 59),
  measure = "survival"
)[, 1:5]
```

`pparg` *PPARg cheminformatics data*

Description

Virtual screening data for the PPARg receptor, used to demonstrate hit enrichment curves and confidence bands.

Usage

```
pparg
```

Format

A data frame.

References

Ash JR & Hughes-Oliver JM. (2022). Confidence bands for hit enrichment curves. *Journal of Cheminformatics*, 14(1), 1-15.

p_values *P-values for M-Estimator parameters*

Description

Computes two-sided Wald-type P-values from Z-scores. The Z-scores are compared to the standard normal distribution by default. When a `finite_correction` is set on the fit, they are compared instead to the t-distribution with $n - p$ degrees of freedom.

This function mirrors `m.p_values()` in Python delicatessen, so code translated from Python can keep its shape.

Usage

```
p_values(object, null = 0, ...)
```

Arguments

`object`	A fitted `MEstimator` object (after calling `estimate()`).
`null`	Numeric null hypothesis value(s). Default `0`.
`...`	Not used. Must be empty, so a name that is not one of the documented arguments is an error rather than silently ignored.

Value

A numeric vector of P-values.

See Also

`summary()` and `tidy()`, which report the same P-values in table form alongside the other parameter-level results.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                  model = "linear")

p_values(fit)
```

`regression_predictions` *Generate predicted values from a regression model*

Description

Computes predicted outcomes, their variance, and Wald-type confidence intervals from estimated regression coefficients and their covariance matrix. This is a post-processing utility meant to be used after `MEstimator()` has been fitted.

Usage

```
regression_predictions(X, theta, covariance, offset = NULL, alpha = 0.05)
```

Arguments

`X`	Numeric n-by-p design matrix of covariate values for prediction.
`theta`	Numeric vector of p estimated coefficients (from `coef(m)`).
`covariance`	Numeric p-by-p covariance matrix (from `vcov(m)`).
`offset`	Optional numeric vector of n offsets. Default `NULL`.
`alpha`	Numeric significance level for confidence intervals. Default `0.05` (95% CIs).

Details

No transformations are applied. For logistic models this returns log-odds (not probabilities). Apply `stats::plogis()` for the probability scale, or `inverse_logit()` if the values feed a transform passed to `delta_method()` with `deriv_method = "exact"`.

Value

A data frame with n rows and columns: `predicted`, `variance`, `lower`, `upper`. The rows are labeled by position regardless of the row names of `X`, which say nothing about the predictions made from it.

Examples

```
set.seed(1)
n <- 200
dat <- data.frame(x = rnorm(n), z = rbinom(n, 1, 0.5))
dat$y <- 1 + 0.5 * dat$x + 2 * dat$z + rnorm(n)

m <- m_estimate(y ~ x + z, data = dat, .ee = ee_regression, model = "linear")

# Predict along a small grid of x, holding z at 1. The columns of the grid
# must match the order of the coefficients, intercept first.
X_new <- cbind(1, x = c(-1, 0, 1), z = 1)
regression_predictions(X_new, theta = coef(m), covariance = vcov(m))
```

robust_loss_functions *Robust loss function derivatives*

Description

Computes the first derivative (psi function) of robust loss functions, evaluated at the given residuals. Used internally by `ee_mean_robust()` and `ee_robust_regression()`.

This function mirrors `robust_loss_functions()` in Python delicatessen, so code translated from Python can keep its shape. There is no base R equivalent for these score functions, so this is the interface for them in deli as well.

Usage

```
robust_loss_functions(residuals, loss, k)
```

Arguments

`residuals`	Numeric vector of residuals.
`loss`	Character string specifying the loss function. One of: `"huber"`, `"tukey"`, `"andrew"`, `"hampel"`, `"fair"`, `"cauchy"`, `"ullah"`, `"welsch"`.
`k`	Numeric tuning constant. For `"hampel"`, a length-3 vector `c(a, b, c)` where `a < b < c`.

Value

Numeric vector the same length as `residuals`.

Examples

```
r <- c(-5, -1, 0, 1, 5)
robust_loss_functions(r, "huber", k = 1.345)
robust_loss_functions(r, "tukey", k = 4.685)
```

robust_regress *Robust regression example data*

Description

Illustrative example of robust linear regression from Zivich et al. (2022).

Usage

```
robust_regress
```

Format

A data frame with 15 rows and 3 columns:

height Height measurement

weight Weight measurement (with outlier induced at observation 9)

weight_no_outlier Weight measurement (without outlier)

References

Zivich PN, Klose M, Cole SR, Edwards JK, & Shook-Sa BE. (2022). Delicatessen: M-estimation in Python. *arXiv:2203.11300*.

`sdss_quasar` *SDSS quasar survey data*

Description

Quasar photometric data from the Sloan Digital Sky Survey 3rd Data Release (n=46,420 quasars), used to demonstrate generalized additive models.

Usage

```
sdss_quasar
```

Format

A data frame with columns for redshift (z) and magnitude bands.

References

Weinstein MA, et al. (2004). An empirical algorithm for broadband photometric redshifts of quasars from the Sloan Digital Sky Survey. *The Astrophysical Journal Supplement Series*, 155(2), 243.

`shaq_free_throws` *Shaquille O'Neal free throw data*

Description

Example data from Boos and Stefanski (2013) on Shaquille O'Neal free throws in the 2000 NBA playoffs (Table 7.1 on pg 324).

Usage

```
shaq_free_throws
```

Format

A data frame with 23 rows and 3 columns:

game Game number

ft_success Free throws made during game

ft_attempt Free throws attempted during game

References

Boos DD, & Stefanski LA. (2013). M-estimation (estimating equations). In Essential Statistical Inference (pp. 297-337). Springer, New York, NY.

---

standard_normal_cdf *Standard normal CDF*

Description

Evaluates the cumulative distribution function of the standard normal distribution.

This is the equivalent of `stats::pnorm()` at the default mean and standard deviation, and returns identical values for numeric input, except that it carries derivatives. Exact differentiation (`deriv_method = "exact"`) propagates a tangent alongside each value, and `standard_normal_cdf()` recognizes a tangent-carrying argument and applies the analytic rule itself, the standard normal density. `pnorm()` hands its argument to compiled code without dispatching, and errors on such an argument. Use `standard_normal_cdf()` inside estimating equations and inside transforms passed to `delta_method()`, and `pnorm()` for ordinary numeric work.

Usage

```
standard_normal_cdf(x)
```

Arguments

`x`	Numeric value or vector of quantiles.

Value

Numeric CDF values.

Exact differentiation

`deriv_method = "exact"` is forward-mode automatic differentiation: it replaces each value with an object carrying both the value and its derivative. deli supports those objects through S3 methods: the `Ops`, `Math`, and `Summary` group generics, plus non-group methods such as `[`, `%*%`, `t()`, `c()`, and `mean()`. `standard_normal_cdf()`, `standard_normal_pdf()`, `deli_polygamma()`, and `deli_digamma()` recognize a tangent-carrying argument themselves and apply their own analytic rule. Support within the group generics is partial; see `vignette("getting-started")` for the operations deli differentiates.

`plogis()`, `qlogis()`, `pnorm()`, `dnorm()`, and `psigamma()` take none of these paths. Each hands its argument straight to compiled code through `.Call()` or `.Internal()` without dispatching, so the tangent reaches C code that requires a plain number. deli catches the resulting failure and raises its own error, naming the function that stopped the computation and the deli function to write in its place. The same applies to every other distribution function in `stats`, such as `qnorm()`, which is named in the error even though deli exports no counterpart for it.

Each deli utility below returns the same values as its base R counterpart for numeric input. What separates them is whether the counterpart survives exact mode:

deli function	base R counterpart	base R under `deriv_method = "exact"`
`inverse_logit()`	`stats::plogis()`	errors
`logit()`	`stats::qlogis()`	errors
`standard_normal_cdf()`	`stats::pnorm()`	errors
`standard_normal_pdf()`	`stats::dnorm()`	errors
`deli_polygamma()`	`base::psigamma()`	errors
`deli_digamma()`	`base::digamma()`	works
`identity_transform()`	`base::identity()`	works

For the first five rows, use the deli function inside estimating equations and inside transforms passed to `delta_method()`, and the base R function everywhere else: simulating data, post-fit display, plain numeric work. For the last two rows the base R function is usable everywhere, because `digamma()` is a `Math` group member with a tangent rule and `identity()` passes its argument through untouched.

The polygamma row is the one place where the arguments do not line up. `deli_polygamma(n, x)` takes the derivative order first and `psigamma(x, deriv = n)` takes it second, so a positional substitution between the two computes a different quantity and raises no error.

See Also

`stats::pnorm()`, the base R equivalent for ordinary numeric work, and `standard_normal_pdf()` for the density.

Examples

```
standard_normal_cdf(0)
standard_normal_cdf(c(-1.96, 0, 1.96))

# The same values as the base R counterpart
all.equal(standard_normal_cdf(c(-1.96, 0, 1.96)), pnorm(c(-1.96, 0, 1.96)))

# A probit regression, whose mean model is the standard normal CDF
m <- m_estimate(
  vs ~ mpg,
  data = mtcars,
  .ee = ee_glm,
  distribution = "binomial",
  link = "probit"
)
```

```
# Variance of the fitted probability at mpg = 20. Writing
# `pnorm(theta[1] + theta[2] * 20)` here instead would error, because exact
# differentiation hands the transform a tangent-carrying argument.
delta_method(
  m,
  transform = function(theta) standard_normal_cdf(theta[1] + theta[2] * 20),
  deriv_method = "exact"
)
```

`standard_normal_pdf` *Standard normal PDF*

Description

Evaluates the probability density function of the standard normal distribution.

This is the equivalent of `stats::dnorm()` at the default mean and standard deviation, and returns identical values for numeric input, except that it carries derivatives. Exact differentiation (`deriv_method = "exact"`) propagates a tangent alongside each value, and `standard_normal_pdf()` recognizes a tangent-carrying argument and applies the analytic rule itself, $-x$ times the density. `dnorm()` hands its argument to compiled code without dispatching, and errors on such an argument. Use `standard_normal_pdf()` inside estimating equations and inside transforms passed to `delta_method()`, and `dnorm()` for ordinary numeric work.

Usage

```
standard_normal_pdf(x)
```

Arguments

`x`	Numeric value or vector of quantiles.

Value

Numeric density values.

Exact differentiation

`deriv_method = "exact"` is forward-mode automatic differentiation: it replaces each value with an object carrying both the value and its derivative. deli supports those objects through S3 methods: the `Ops`, `Math`, and `Summary` group generics, plus non-group methods such as `[`, `%*%`, `t()`, `c()`, and `mean()`. `standard_normal_cdf()`, `standard_normal_pdf()`, `deli_polygamma()`, and `deli_digamma()` recognize a tangent-carrying argument themselves and apply their own analytic rule. Support within the group generics is partial; see `vignette("getting-started")` for the operations deli differentiates.

`plogis()`, `qlogis()`, `pnorm()`, `dnorm()`, and `psigamma()` take none of these paths. Each hands its argument straight to compiled code through `.Call()` or `.Internal()` without dispatching, so

the tangent reaches C code that requires a plain number. deli catches the resulting failure and raises its own error, naming the function that stopped the computation and the deli function to write in its place. The same applies to every other distribution function in `stats`, such as `qnorm()`, which is named in the error even though deli exports no counterpart for it.

Each deli utility below returns the same values as its base R counterpart for numeric input. What separates them is whether the counterpart survives exact mode:

deli function	base R counterpart	base R under `deriv_method = "exact"`
`inverse_logit()`	`stats::plogis()`	errors
`logit()`	`stats::qlogis()`	errors
`standard_normal_cdf()`	`stats::pnorm()`	errors
`standard_normal_pdf()`	`stats::dnorm()`	errors
`deli_polygamma()`	`base::psigamma()`	errors
`deli_digamma()`	`base::digamma()`	works
`identity_transform()`	`base::identity()`	works

For the first five rows, use the deli function inside estimating equations and inside transforms passed to `delta_method()`, and the base R function everywhere else: simulating data, post-fit display, plain numeric work. For the last two rows the base R function is usable everywhere, because `digamma()` is a `Math` group member with a tangent rule and `identity()` passes its argument through untouched.

The polygamma row is the one place where the arguments do not line up. `deli_polygamma(n, x)` takes the derivative order first and `psigamma(x, deriv = n)` takes it second, so a positional substitution between the two computes a different quantity and raises no error.

See Also

`stats::dnorm()`, the base R equivalent for ordinary numeric work, and `standard_normal_cdf()` for the distribution function.

Examples

```
standard_normal_pdf(0)
standard_normal_pdf(c(-1, 0, 1))

# The same values as the base R counterpart
all.equal(standard_normal_pdf(c(-1, 0, 1)), dnorm(c(-1, 0, 1)))

m <- m_estimate(
  vs ~ mpg,
  data = mtcars,
  .ee = ee_glm,
  distribution = "binomial",
  link = "probit"
)

# Variance of the marginal effect of mpg at mpg = 20, the standard normal
# density at the linear predictor times the coefficient. Writing
# `dnorm(theta[1] + theta[2] * 20)` here instead would error, because exact
```

```
# differentiation hands the transform a tangent-carrying argument.
delta_method(
  m,
  transform = function(theta) {
    standard_normal_pdf(theta[1] + theta[2] * 20) * theta[2]
  },
  deriv_method = "exact"
)
```

`survival_predictions` *Generate predicted survival measures from a parametric survival model*

Description

Computes predicted survival analysis measures and point-wise confidence intervals using the delta method. Meant to be used after fitting `ee_survival_model()` with `MEstimator()`.

Usage

```
survival_predictions(
  times,
  theta,
  covariance,
  distribution,
  measure = "survival",
  alpha = 0.05,
  deriv_method = "capprox",
  dx = 1e-09
)
```

Arguments

`times`	Numeric vector of time points for prediction.
`theta`	Numeric vector of estimated parameters from `ee_survival_model`.
`covariance`	Numeric covariance matrix from `vcov(m)`.
`distribution`	Character string matching the distribution used in `ee_survival_model`: `"exponential"`, `"weibull"`, or `"gompertz"`. Any other value is an error.
`measure`	Character string: `"survival"`, `"risk"`, `"density"`, `"hazard"`, or `"cumulative_hazard"`. Default `"survival"`.
`alpha`	Numeric significance level. Default `0.05`.
`deriv_method`	Character string for the derivative method used to build the delta-method Jacobian. One of `"capprox"` (central difference), `"fapprox"` (forward difference), `"bapprox"` (backward difference), or `"exact"` (forward-mode automatic differentiation). Default `"capprox"`. Python Delicatessen uses exact differentiation internally; pass `deriv_method = "exact"` to reproduce it with exact derivatives and no step-size tuning. See `delta_method()`.

dx — Numeric step size for the finite-difference methods; ignored when `deriv_method = "exact"`. Default `1e-9`. Must be a single positive finite number, which is checked whichever `deriv_method` is in force. The step is absolute and is floored at the floating-point resolution of each estimate, so a large parameter magnitude cannot silently reduce it to nothing; see `approx_differentiation()`.

Value

A data frame with columns: `time`, `predicted`, `variance`, `lower`, `upper`.

Gompertz distribution

The Gompertz survival and hazard follow the `ee_survival_model()` parameterization,

$$S(t) = \exp\left(-\frac{\lambda}{\gamma}\left(e^{\gamma t} - 1\right)\right), \qquad h(t) = \lambda e^{\gamma t}.$$

This is a deliberate divergence from Python Delicatessen, whose `survival_predictions` branches only on the exponential distribution and otherwise applies the Weibull formulas, so a `"gompertz"` request there silently returns Weibull values.

Examples

```
# Weibull survival times for 45 women with breast cancer, with no covariates.
# The default rootSolve solver does not converge here, so nleqslv is used.
psi <- function(theta) {
  ee_survival_model(
    theta,
    time = breast_cancer$times,
    event = breast_cancer$delta,
    distribution = "weibull"
  )
}
m <- m_estimate(
  stacked_equations = psi,
  init = c(0.1, 0.1),
  solver = "nleqslv"
)

# The survival function at three follow-up times, with delta-method
# confidence intervals. Observed times run from 5 to 225 months.
survival_predictions(
  times = c(50, 100, 150),
  theta = coef(m),
  covariance = vcov(m),
  distribution = "weibull",
  measure = "survival"
)
```

`s_values` *S-values (surprisal) for M-Estimator parameters*

Description

Computes Shannon Information values (S-values) as $S = -\log_2(P)$, where P is the corresponding P-value.

This function mirrors `m.s_values()` in Python delicatessen, so code translated from Python can keep its shape.

Usage

```
s_values(object, null = 0, ...)
```

Arguments

`object`	A fitted `MEstimator` object (after calling `estimate()`).
`null`	Numeric null hypothesis value(s). Default `0`.
`...`	Not used. Must be empty, so a name that is not one of the documented arguments is an error rather than silently ignored.

Details

A P-value small enough to underflow to exactly zero has an S-value of `Inf`. That is the limit the surprisal is heading toward rather than a defect. The smallest P a double can hold is 2^{-1074}, so a P that arrives as zero stands for more than a thousand bits of surprisal, past the range a double can name. The infinity reports evidence beyond measurement, where any finite substitute would name a number the fit does not support.

Value

A numeric vector of S-values.

See Also

`summary()` and `tidy()`, which report the same S-values in table form alongside the other parameter-level results.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                  model = "linear")

s_values(fit)
```

`z_scores` *Z-scores for M-Estimator parameters*

Description

Computes Wald-type Z-scores: $(\hat{\theta} - \theta_0)/\widehat{SE}(\hat{\theta})$.

This function mirrors `m.z_scores()` in Python delicatessen, so code translated from Python can keep its shape.

Usage

```
z_scores(object, null = 0, ...)
```

Arguments

`object`	A fitted `MEstimator` object (after calling `estimate()`).
`null`	Numeric null hypothesis value(s). Default `0`.
`...`	Not used. Must be empty, so a name that is not one of the documented arguments is an error rather than silently ignored.

Value

A numeric vector of Z-scores.

See Also

`summary()` and `tidy()`, which report the same Z-scores in table form alongside the other parameter-level results.

Examples

```
fit <- m_estimate(mpg ~ wt + hp, data = mtcars, .ee = ee_regression,
                  model = "linear")

z_scores(fit)
```

Index