Package: coxdev (via r-universe)

November 8, 2026

Title Cox Model Deviance, Gradient, Information and Score Sampling

Version 0.4.0

Description Fast computation of the deviance of the Cox proportional hazards model (Cox (1972) <doi:10.1111/j.2517-6161.1972.tb00899.x>), its gradient and the diagonal of its Hessian with respect to the linear predictor, the information and its derivative in a direction as matrix-free linear operators, and draws of the score (multinomial, Bernoulli, Fisher noncentral hypergeometric or wild), with the ties of Breslow (1974) <doi:10.2307/2529620> or Efron (1977) <doi:10.1080/01621459.1977.10480613>, start times (left truncation), strata and sample weights. The 'C++' code is header-only, shared with the 'Python' package 'coxdev', and installed for packages that use it through 'LinkingTo'.

License GPL (>= 3)

URL https://github.com/jonathan-taylor/coxdev

BugReports https://github.com/jonathan-taylor/coxdev/issues

Encoding UTF-8

Depends R (>= 4.1.0)

Imports R6, Rcpp, stats

LinkingTo Rcpp, RcppEigen

Suggests knitr, Matrix, rmarkdown, survival, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

NeedsCompilation yes

Author Jonathan Taylor [aut, cre], Trevor Hastie [aut], Balasubramanian Narasimhan [aut]

Maintainer Jonathan Taylor <jtaylo@stanford.edu>

Repository https://jonathan-taylor-stanford.r-universe.dev

Date/Publication 2026-11-07 20:59:59 UTC

RemoteUrl https://github.com/cran/coxdev

RemoteRef HEAD

RemoteSha ec20f6ef07430244d757b7552cc8abc8898f1768

Contents

CoxDeviance 2
CoxInformation 9
CoxInformationDerivative 11
make_cox_deviance 13

Index **14**

`CoxDeviance` *Cox model deviance, gradient and information*

Description

An R6 class computing the deviance of the Cox proportional hazards model, its gradient and the diagonal of its Hessian with respect to the linear predictor, the information as a matrix-free linear operator, and draws of the score. It handles Efron's and Breslow's tie-breaking, start times (left truncation), strata and sample weights. It is the R version of the `CoxDeviance` class of the Python package, and runs the same C++ code.

The deviance is `2 * (loglik_sat - loglik)`, with `loglik` the (weighted) log partial likelihood and `loglik_sat` its value for the saturated model. The `gradient` and `diag_hessian` are of the deviance with respect to the linear predictor. The information is half the Hessian of the deviance, the negative Hessian of `loglik`: for a design `X`, `crossprod(X, I %*% X)` is the information of the coefficients.

Results are cached: a second call with the same linear predictor and weights does no work.

Value

`CoxDeviance$new()` returns a `CoxDeviance` object, whose methods are documented below: `compute()` returns a `CoxDevianceResult` (a list with the deviance, its gradient and the diagonal of its Hessian), `information()` a CoxInformation operator and `information_derivative()` a CoxInformation-Derivative operator.

Public fields

`tie_breaking` The tie-breaking method, `"efron"` or `"breslow"`.

Active bindings

`n` The number of observations.

`stratified` Whether the model has strata.

Methods

Public methods:

- CoxDeviance$new()
- CoxDeviance$compute()
- CoxDeviance$c_index()
- CoxDeviance$information()
- CoxDeviance$information_derivative()
- CoxDeviance$sample_score()
- CoxDeviance$wild_score()
- CoxDeviance$score_residuals()
- CoxDeviance$schoenfeld_residuals()
- CoxDeviance$risk_set_variance()
- CoxDeviance$expected_events()
- CoxDeviance$survival_at_exit()
- CoxDeviance$baseline_hazard()
- CoxDeviance$print()

`CoxDeviance$new()`: Set up the computations for a survival response, given as vectors or as a `survival::Surv()` object.

Usage:

```
CoxDeviance$new(
  event,
  status,
  start = NULL,
  strata = NULL,
  sample_weight = NULL,
  tie_breaking = c("efron", "breslow")
)
```

Arguments:

`event` Event (failure or censoring) times; or the response as a `Surv` object, `Surv(time, status)` or `Surv(start, stop, status)`, in place of `event`, `status` and `start`. Strata set on it by `glmnet::stratifySurv()` are used.

`status` Event indicators: 1 for a failure, 0 for censoring.

`start` Start (entry) times, for left-truncated data, or `NULL`.

`strata` Stratum labels (integers, a factor or character), or `NULL` for an unstratified model.

`sample_weight` Sample weights, or `NULL` for all one. They are needed here only to know which are 0; each computation takes its own.

`tie_breaking` `"efron"` (the default) or `"breslow"`.

`CoxDeviance$compute()`: Compute the deviance and its derivatives (the `__call__` of the Python class).

Usage:

```
CoxDeviance$compute(linear_predictor, sample_weight = NULL)
```

Arguments:

`linear_predictor` The linear predictor, one entry per observation.

`sample_weight` Sample weights, or `NULL` for all one.

Returns: A `CoxDevianceResult`: a list with the `linear_predictor`, the `sample_weight`, `loglik_sat` (the saturated log-likelihood), the `deviance`, its `gradient` with respect to the linear predictor and the diagonal `diag_hessian` of its Hessian.

`CoxDeviance$c_index()`**:** Harrell's concordance index (C-index) of a linear predictor. For each failure `i` at time `t`, every `j` at risk at `t` (`start[j] < t <= event[j]`) is compared with `i`, except the other failures at `t`. The pair has weight `w_i * w_j`, and is concordant if `linear_predictor[i] > linear_predictor[j]`, a higher risk failing first; tied predictions count half. With strata, only pairs within a stratum are compared.

This is `survival::concordance(y ~ -linear_predictor, weights = w)` (with `+ strata(...)` for strata) and, without strata, `glmnet::Cindex` (Harrell et al., 1982; Harrell, Lee and Mark, 1996). It is computed in O(n log n) per column, by the algorithm of the survival package's concordance vignette (Therneau and Atkinson), with the weights of the risk set in a Fenwick tree (Fenwick, 1994). It does not need `compute()`.

Usage:

```
CoxDeviance$c_index(linear_predictor, sample_weight = NULL)
```

Arguments:

`linear_predictor` The linear predictor (a risk score): a vector, or a matrix with one column per predictor.

`sample_weight` As for `compute()`.

Returns: The C-index, one per column of a matrix; `NaN` if no pair is comparable.

`CoxDeviance$information()`**:** The information at a linear predictor: half the Hessian of the deviance, as a matrix-free linear operator.

Usage:

```
CoxDeviance$information(linear_predictor, sample_weight = NULL)
```

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

Returns: A CoxInformation operator.

`CoxDeviance$information_derivative()`**:** The derivative of the information at a linear predictor in a direction of the linear predictor, `d/dt information(linear_predictor + t * direction)` at `t = 0`, as a matrix-free linear operator. It is symmetric, and a product costs `O(n)`, a few times a product with the information. `crossprod(X, dI %*% X)` is the derivative of the information of the coefficients of a design `X`, as needed for the derivative of a log determinant (e.g. in REML or a Laplace approximation), where `direction` is the derivative of the linear predictor. The derivative of the Hessian of the deviance is twice it.

Usage:

```
CoxDeviance$information_derivative(
  linear_predictor,
  direction,
  sample_weight = NULL
)
```

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

`direction` The direction of the derivative, one entry per observation.

Returns: A CoxInformationDerivative operator.

`CoxDeviance$sample_score()`**:** Draws of the score at a linear predictor: from the Cox model with that linear predictor, or (`"wild"`) from its score residuals.

Each draw `r` has one entry per observation, on the scale of the score `-gradient / 2`, so that `crossprod(X, r)` is a draw of the score of the coefficients of a design `X`. The model-based methods draw the failures at each event time from the model and return their score: the sum over the failures of `x_j` minus its expectation, with risk weights `sample_weight * exp(linear_predictor)`. They differ only at tied event times, and need equal positive weights.

- `"multinomial"`: the `d` failures drawn with replacement, with probabilities proportional to the risk weights: the sampling model of Breslow's likelihood. Its covariance is the Breslow information.
- `"bernoulli"`: each subject at risk fails independently with probability `min(1, d p)`, a Cox-model analogue of the weird bootstrap, centered with the number of failures drawn. Its mean is zero only while `d p < 1`.
- `"fisher"`: the set of `d` failures drawn from Fisher's noncentral hypergeometric distribution, the sampling model of the exact discrete partial likelihood, centered at its exact mean. Its covariance is the information of the exact likelihood.
- `"wild"`: the score residuals (of the likelihood set by `tie_breaking`) times independent standard normals. Its covariance is the sandwich, which does not need the model to be right.

See the package README for references.

Usage:

```
CoxDeviance$sample_score(
  linear_predictor,
  sample_weight = NULL,
  method = c("multinomial", "bernoulli", "fisher", "wild"),
  n_draws = NULL,
  seed = NULL
)
```

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

`method` One of `"multinomial"`, `"bernoulli"`, `"fisher"` and `"wild"`.

`n_draws` The number of draws, or `NULL` for one draw as a vector.

`seed` The seed of the random numbers: a nonnegative whole number (up to `2^53`), or a string of decimal digits for any seed below `2^64`. The same seed gives the same draws, here and from Python: the same random numbers on every platform, and draws equal to rounding (the last bits of `exp`, `log`, `sin` and `cos` can differ between math libraries). If `NULL`, a seed is drawn from R's random number generator, so that `set.seed()` makes the draws reproducible.

Returns: A vector of length `n` if `n_draws` is `NULL`, else an `n` by `n_draws` matrix.

`CoxDeviance$wild_score()`**:** The wild draw of the score for given multipliers: `r` with `crossprod(X, r)` equal to `sum_i multipliers[i] * U_i`, where `U_i` is the score residual of observation `i` (its contribution to the score, centered within each risk set, with Efron's weights if `tie_breaking = "efron"`). With `multipliers` all one, `r` is the score `-gradient / 2`.

Usage:

```
CoxDeviance$wild_score(linear_predictor, multipliers, sample_weight = NULL)
```

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

`multipliers` One multiplier per observation.

Returns: A vector of length `n`.

`CoxDeviance$score_residuals()`**:** The score residuals of a design at a linear predictor.

Row `i` is `U_i`, the contribution of observation `i` to the score of the coefficients, `crossprod(X, -gradient / 2)`: the sum over the failure times of `(w_i 1{i fails} / d - wbar p_i) (x_i - xbar)`, centered within each risk set, with Efron's weights for ties if `tie_breaking = "efron"` (Lin and Wei, 1989). The rows sum to the score, `drop(G %*% U)` is `crossprod(X, wild_score(eta, G))`, and `crossprod(U)` is the middle of the robust (sandwich) variance. They are `sample_weight` times `residuals(fit, type = "score")` of a `survival::coxph` fit, which are per unit weight.

The residuals are built a column at a time, each in O(n). `U` is dense even for a sparse `X`, since the risk-set means spread each covariate over the subjects at risk. For draws `crossprod(U, G)` (a wild bootstrap) there is no need to form `U`: `crossprod(X, wild_score(eta, G))` is the same, in one O(n) call.

Usage:

```
CoxDeviance$score_residuals(linear_predictor, X, sample_weight = NULL)
```

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

`X` The design: a matrix with one row per observation, a sparse `Matrix` (made dense a column at a time), or a vector.

Returns: A dense matrix with the dimnames of `X` (a vector for a vector `X`).

`CoxDeviance$schoenfeld_residuals()`**:** The Schoenfeld residuals of a design `X` at a linear predictor. Row `i`, for an observation that fails (with positive weight), is `x_i - xbar`, with `xbar` the mean of the covariates over its risk set with weights `sample_weight * exp(eta)`. With `tie_breaking = "efron"`, `xbar` is the average over the failures tied with `i` of their Efron means, as in `survival`. The other rows are zero. Weighted by `sample_weight`, the rows sum to the score: `crossprod(U, sample_weight)` is `crossprod(X, -gradient / 2)`.

`residuals(fit, type = "schoenfeld")` of a `survival::coxph` fit has the rows of the failures only, in order of failure time: the rows with `status == 1` here, in the order of the data.

Introduced by Schoenfeld (1982). Plotted against time, they check the proportional hazards assumption (Grambsch and Therneau, 1994; `survival::cox.zph`).

The residuals are built a column at a time, each in O(n).

Usage:

```
CoxDeviance$schoenfeld_residuals(linear_predictor, X, sample_weight = NULL)
```

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

`X` As for `score_residuals()`.

Returns: A dense matrix with the dimnames of `X` (a vector for a vector `X`), with a row for every observation.

`CoxDeviance$risk_set_variance()`: The risk-set variances of the columns of `Z` at a linear predictor. Row `i`, for an observation that fails (with positive weight), is the variance of each covariate over its risk set with weights `sample_weight * exp(eta)`: the `V_z` by which Grambsch and Therneau (1994) scale the Schoenfeld residuals, `beta + r / V_z` for one covariate. With `tie_breaking = "efron"`, it is the average over the failures tied with `i` of their Efron variances, as for `schoenfeld_residuals()`. The other rows are zero. Weighted by `sample_weight`, the rows sum to the diagonal of the information, `diag(crossprod(Z, I %*% Z))`.

The covariance of two covariates `y` and `z` follows by polarization, `(V(y + z) - V(y - z)) / 4`.

The variances are computed a column at a time, each in O(n).

Usage:

```
CoxDeviance$risk_set_variance(linear_predictor, Z, sample_weight = NULL)
```

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

`Z` The covariates, as `X` for `score_residuals()`.

Returns: A dense matrix with the dimnames of `Z` (a vector for a vector `Z`), with a row for every observation.

`CoxDeviance$expected_events()`: The expected number of events of each observation at a linear predictor: `exp(linear_predictor[i])` times the increase of the baseline cumulative hazard (see `baseline_hazard()`) over `(start[i], event[i]]`. With `tie_breaking = "efron"`, a failure gets Efron's share of the hazard at its own failure time. This is `predict(fit, type = "expected")` of a `survival::coxph` fit: `status - expected_events` is the martingale residual (Barlow and Prentice, 1988; Therneau, Grambsch and Fleming, 1990).

Usage:

```
CoxDeviance$expected_events(linear_predictor, sample_weight = NULL)
```

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

Returns: A vector, one entry per observation.

`CoxDeviance$survival_at_exit()`: The expected survival probability of each observation at its exit: the probability of surviving to `event[i]`, given survival to `start[i]` (to 0 without start times), at a linear predictor. It is `exp(-expected_events)`, the `predict(fit, type = "survival")` of a `survival::coxph` fit (Therneau and Grambsch, 2000, Chapter 4).

Usage:

```
CoxDeviance$survival_at_exit(linear_predictor, sample_weight = NULL)
```

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

Returns: A vector, one entry per observation.

`CoxDeviance$baseline_hazard()`: The estimate of the baseline hazard at a linear predictor: Breslow's, or with `tie_breaking = "efron"` Efron's, which spreads the tied failures at a time over Efron's shrinking risk sets. It is the `survival::basehaz(fit, centered = FALSE)` of a `coxph` fit, at the failure times.

Usage:

`CoxDeviance$baseline_hazard(linear_predictor, sample_weight = NULL)`

Arguments:

`linear_predictor, sample_weight` As for `compute()`.

Returns: A data frame with one row per distinct failure time (of positive weight), in increasing time within each stratum: the `time`, the `hazard` (the increment of the cumulative hazard there) and the `cumulative_hazard`, for the linear predictor as given, and with strata the `stratum`. The cumulative hazard of observation `i` at time `t` is the `cumulative_hazard` at the last failure time `<= t` (in its stratum) times `exp(linear_predictor[i])`.

`CoxDeviance$print()`**:** Print a summary.

Usage:

`CoxDeviance$print(...)`

Arguments:

`...` Ignored.

References

Barlow, W. E. and Prentice, R. L. (1988). Residuals for relative risk regression. *Biometrika*, 75(1), 65–74.

Breslow, N. E. (1972). Discussion of Professor Cox's paper. *Journal of the Royal Statistical Society, Series B*, 34(2), 216–217.

Fenwick, P. M. (1994). A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3), 327–336.

Grambsch, P. M. and Therneau, T. M. (1994). Proportional hazards tests and diagnostics based on weighted residuals. *Biometrika*, 81(3), 515–526.

Harrell, F. E., Califf, R. M., Pryor, D. B., Lee, K. L. and Rosati, R. A. (1982). Evaluating the yield of medical tests. *JAMA*, 247(18), 2543–2546.

Harrell, F. E., Lee, K. L. and Mark, D. B. (1996). Multivariable prognostic models: issues in developing models, evaluating assumptions and adequacy, and measuring and reducing errors. *Statistics in Medicine*, 15(4), 361–387.

Lin, D. Y. and Wei, L. J. (1989). The robust inference for the Cox proportional hazards model. *Journal of the American Statistical Association*, 84(408), 1074–1078.

Schoenfeld, D. (1982). Partial residuals for the proportional hazards regression model. *Biometrika*, 69(1), 239–241.

Therneau, T. M. and Atkinson, E. J. The concordance statistic. Vignette of the R package survival, `vignette("concordance", package = "survival")`.

Therneau, T. M. and Grambsch, P. M. (2000). *Modeling Survival Data: Extending the Cox Model*. Springer, New York.

Therneau, T. M., Grambsch, P. M. and Fleming, T. R. (1990). Martingale-based residuals for survival models. *Biometrika*, 77(1), 147–160.

Examples

```
set.seed(1)
n <- 100
X <- matrix(rnorm(n * 3), n, 3)
eta <- drop(X %*% c(0.5, -0.5, 0.25))
event <- ceiling(rexp(n, exp(eta)) * 4) / 4   # with ties
status <- rbinom(n, 1, 0.7)
cox <- CoxDeviance$new(event = event, status = status, tie_breaking = "efron")
# or, the same, with the response as a Surv object
if (requireNamespace("survival", quietly = TRUE)) {
  cox <- CoxDeviance$new(survival::Surv(event, status), tie_breaking = "efron")
}

# deviance, gradient and diagonal of the Hessian
res <- cox$compute(eta)
res$deviance
head(res$gradient)

# the information, as an operator
I <- cox$information(eta)
info_beta <- crossprod(X, I %*% X)
sqrt(diag(solve(info_beta)))   # standard errors

# draws of the score of the coefficients
draws <- crossprod(X, cox$sample_score(eta, n_draws = 2000, seed = 1))
diag(cov(t(draws))) / diag(info_beta)   # near 1
```

`CoxInformation` *The information of the Cox model as a linear operator*

Description

The information at a linear predictor (half the Hessian of the deviance with respect to the linear predictor, the negative Hessian of the log partial likelihood), as an `n` by `n` symmetric linear operator that is never formed: a product with a vector costs `O(n)`, after the sort done once by CoxDeviance. Made by `CoxDeviance$information()`.

Besides its methods, the operator works with `%*%` (on either side, with a vector or a matrix; R >= 4.4.0), `dim()`, `nrow()`, `ncol()`, `t()` and `as.matrix()`; its diagonal is `$diagonal()`.

The operator keeps its linear predictor and weights: it gives the information there even after the CoxDeviance has computed at others.

Value

Made by `CoxDeviance$information()`, a `CoxInformation` object, an R6 linear operator: `multiply()` returns the product with a vector (a vector) or with a matrix (a matrix), `diagonal()` the diagonal (a vector), and `to_dense()` the `n` by `n` matrix.

Public fields

`result` The CoxDeviance result at which this is the information.

Methods

Public methods:

- `CoxInformation$new()`
- `CoxInformation$multiply()`
- `CoxInformation$multiply_transpose()`
- `CoxInformation$rows()`
- `CoxInformation$cols()`
- `CoxInformation$diagonal()`
- `CoxInformation$to_dense()`
- `CoxInformation$print()`

`CoxInformation$new()`: Made by `CoxDeviance$information()`, not directly.

Usage:

```
CoxInformation$new(result, matmat)
```

Arguments:

`result` A `CoxDevianceResult`.

`matmat` A function multiplying each column of a matrix by the information at `result`.

`CoxInformation$multiply()`: The product of the information with `v`.

Usage:

```
CoxInformation$multiply(v)
```

Arguments:

`v` A vector of length `n`, or a matrix with `n` rows.

Returns: A vector of length `n` if `v` is a vector, else a matrix.

`CoxInformation$multiply_transpose()`: The product of the transpose with `v`: as the information is symmetric, the same as `multiply(v)`.

Usage:

```
CoxInformation$multiply_transpose(v)
```

Arguments:

`v` A vector of length `n`, or a matrix with `n` rows.

`CoxInformation$rows()`: The number of rows, `n`.

Usage:

```
CoxInformation$rows()
```

`CoxInformation$cols()`: The number of columns, `n`.

Usage:

```
CoxInformation$cols()
```

`CoxInformation$diagonal()`: The diagonal of the information, `diag_hessian / 2`.

Usage:

```
CoxInformation$diagonal()
```

`CoxInformation$to_dense()`: The `n` by `n` matrix, by `n` products: for small problems and testing.

Usage:

```
CoxInformation$to_dense()
```

`CoxInformation$print()`: Print a summary.

Usage:

```
CoxInformation$print(...)
```

Arguments:

`...` Ignored.

Examples

```
set.seed(1)
n <- 50
X <- matrix(rnorm(n * 2), n, 2)
eta <- drop(X %*% c(0.5, -0.5))
cox <- CoxDeviance$new(event = rexp(n, exp(eta)), status = rbinom(n, 1, 0.7))
I <- cox$information(eta)
I
crossprod(X, I %*% X)     # the information of the coefficients
t(X) %*% I %*% X          # the same
all.equal(I$diagonal(), diag(as.matrix(I)))
```

CoxInformationDerivative

The derivative of the information of the Cox model in a direction

---

Description

The derivative of the information at a linear predictor in a direction of the linear predictor, `d/dt I(linear_predictor + t * d` at `t = 0`, as an `n` by `n` symmetric linear operator that is never formed: a product with a vector costs `O(n)`. Made by `CoxDeviance$information_derivative()`.

It works as a CoxInformation does, with `%*%`, `dim()`, `t()` and `as.matrix()`, except that it has no `$diagonal()`.

Value

Made by `CoxDeviance$information_derivative()`, a `CoxInformationDerivative` object, an R6 linear operator: `multiply()` returns the product with a vector (a vector) or with a matrix (a matrix), and `to_dense()` the `n` by `n` matrix.

Super class

CoxInformation -> CoxInformationDerivative

Public fields

direction The direction of the derivative.

Methods

Public methods:

- CoxInformationDerivative$new()
- CoxInformationDerivative$diagonal()
- CoxInformationDerivative$print()

CoxInformationDerivative$new(): Made by CoxDeviance$information_derivative(), not directly.

Usage:

```
CoxInformationDerivative$new(result, direction, matmat)
```

Arguments:

result A CoxDevianceResult.

direction The direction of the derivative.

matmat A function multiplying each column of a matrix by the derivative of the information at result in the direction direction.

CoxInformationDerivative$diagonal(): Not available in O(n): use diag(as.matrix(dI)).

Usage:

```
CoxInformationDerivative$diagonal()
```

CoxInformationDerivative$print(): Print a summary.

Usage:

```
CoxInformationDerivative$print(...)
```

Arguments:

... Ignored.

Examples

```
set.seed(1)
n <- 50
X <- matrix(rnorm(n * 2), n, 2)
eta <- drop(X %*% c(0.5, -0.5))
cox <- CoxDeviance$new(event = rexp(n, exp(eta)), status = rbinom(n, 1, 0.7))
# the derivative of the information of the coefficients as they move
# along b
b <- c(1, 0)
dI <- cox$information_derivative(eta, drop(X %*% b))
crossprod(X, dI %*% X)
```

`make_cox_deviance` *Make a Cox deviance object (superseded)*

Description

Superseded by CoxDeviance, which this wraps; kept for code written for earlier versions of the package.

Usage

```
make_cox_deviance(
  event,
  start = NA,
  status,
  tie_breaking = c("efron", "breslow"),
  weight = rep(1, length(event))
)
```

Arguments

`event`	The event times.
`start`	The start times, or `NA` for right-censored data only.
`status`	The event indicators: 1 for a failure, 0 for censoring.
`tie_breaking`	`"efron"` (the default) or `"breslow"`.
`weight`	The sample weights, default all one.

Value

A list of two functions: `coxdev(linear_predictor, sample_weight = NULL)`, giving the result of `CoxDeviance$compute()`, and `information(linear_predictor, sample_weight = NULL)`, giving a function that multiplies a vector or matrix by the information, returning a matrix.

Examples

```
set.seed(10101)
nobs <- 100; nvars <- 10
x <- matrix(rnorm(nobs * nvars), nobs, nvars)
fx <- drop(x[, 1:3] %*% rnorm(3)) / 3
cox_deviance <- make_cox_deviance(event = rexp(nobs, exp(fx)),
                                  status = rbinom(nobs, 1, 0.7))
result <- cox_deviance$coxdev(linear_predictor = fx)
h <- cox_deviance$information(fx)
I <- crossprod(x, h(x))
```

Index

CoxDeviance, 2, *9, 10*, *13*
CoxInformation, *2*, *4*, 9, *11, 12*
CoxInformationDerivative, *2*, *5*, 11

make_cox_deviance, 13

survival::Surv(), *3*