

# Package ‘peperr’

March 25, 2026

**Type** Package

**Date** 2026-03-24

**Title** Parallelised Estimation of Prediction Error

**Version** 1.7

**Depends** snowfall, survival, methods

**Suggests** CoxBoost, glmnet, grpreg, locfit, mboost, ncvreg, penalized,  
randomForestSRC, rlecuyer, SGL, codetools, testthat (>= 3.0.0)

**Author** Christine Porzelius [aut],  
Harald Binder [aut],  
Frederic Bertrand [cre]

**Maintainer** Frederic Bertrand <frederic.bertrand@lecnam.net>

**Description** Designed for prediction error estimation through resampling techniques, possibly accelerated by parallel execution on a compute cluster. Newly developed model fitting routines can be easily incorporated. Methods used in the package are detailed in Porzelius Ch., Binder H. and Schumacher M. (2009) <doi:10.1093/bioinformatics/btp062> and were used, for instance, in Porzelius Ch., Schumacher M. and Binder H. (2011) <doi:10.1007/s00180-011-0236-6>.

**License** GPL (>= 2)

**NeedsCompilation** yes

**RoxygenNote** 7.1.1

**URL** <https://github.com/fbertran/peperr>,  
<https://fbertran.github.io/peperr/>

**BugReports** <https://github.com/fbertran/peperr/issues>

**Config/testthat/edition** 3

**Repository** CRAN

**Date/Publication** 2026-03-25 06:10:46 UTC

## Contents

aggregation.brier	2
aggregation.misclass	3
aggregation.pmpec	4
complexity.cv.penalized	5
complexity.ipec.CoxBoost	6
complexity.mincv.CoxBoost	7
extract.fun	8
fit.CoxBoost	9
fit.coxph	10
fit.glmnet	11
fit.grpsurv	12
fit.mboost.coxph	13
fit.ncvsurv	14
fit.penalized	15
fit.rfsrc	16
fit.SGL.cox	17
ipec	18
peperr	19
perr	26
PLL	28
PLL.CoxBoost	29
PLL.coxph	30
PLL.penfit	30
plot.peperr	31
pmpec	32
predictProb	34
predictProb.CoxBoost	35
predictProb.coxph	36
predictProb.penfit	36
predictProb.survfit	37
resample.indices	38
<b>Index</b>	<b>40</b>

---

aggregation.brier	<i>Determine the Brier score for a fitted model</i>
-------------------	---

---

### Description

Evaluate the Brier score, i.e. prediction error, for a fitted model on new data. To be used as argument `aggregation.fun` in `peperr` call.

### Usage

```
aggregation.brier(full.data=NULL, response, x, model, cplx=NULL,
type=c("apparent", "noinf"), fullsample.attr = NULL, ...)
```

**Arguments**

full.data	passed from peperr, but not used for calculation of the Brier score.
response	vector of binary response.
x	n*p matrix of covariates.
model	model fitted as returned by a fit.fun, as used in a call to peperr.
cplx	passed from peperr, but not necessary for calculation of the Brier score.
type	character.
fullsample.attr	passed from peperr, but not necessary for calculation of the Brier score.
...	additional arguments, passed to predict function.

**Details**

The empirical Brier score is the mean of the squared difference of the risk prediction and the true value of all observations and takes values between 0 and 1, where small values indicate good prediction performance of the risk prediction model.

**Value**

Scalar, indicating the empirical Brier score.

---

aggregation.misclass *Determine the missclassification rate for a fitted model*

---

**Description**

Evaluate the misclassification rate, i.e. prediction error, for a fitted model on new data. To use as argument aggregation.fun in peperr call.

**Usage**

```
aggregation.misclass(full.data=NULL, response, x, model, cplx=NULL,
type=c("apparent", "noinf"), fullsample.attr = NULL, ...)
```

**Arguments**

full.data	passed from peperr, but not used for calculation of the misclassification rate.
response	vector of binary response.
x	n*p matrix of covariates.
model	model fitted with fit.fun.
cplx	passed from peperr, but not necessary for calculation of the misclassification rate.
type	character.

fullsample.attr  
 passed from peperr, but not necessary for calculation of the misclassification rate.  
 ... additional arguments, passed to predict function.

### Details

Misclassification rate is the ratio of observations for which prediction of response is wrong.

### Value

Scalar, indicating the misclassification rate.

---

aggregation.pmppec      *Determine the prediction error curve for a fitted model*

---

### Description

Interface to pmppec, for conforming to the structure required by the argument aggregation.fun in peperr call. Evaluates the prediction error curve, i.e. the Brier score tracked over time, for a fitted survival model.

### Usage

```
aggregation.pmppec(full.data, response, x, model, cplx=NULL, times = NULL,
  type=c("apparent", "noinf"), fullsample.attr = NULL, ...)
```

### Arguments

full.data      data frame with full data set.  
 response      Either a survival object (with Surv(time, status), where time is an n-vector of censored survival times and status an n-vector containing event status, coded with 0 and 1) or a matrix with columns time containing survival times and status containing integers, where 0 indicates censoring, 1 the interesting event and larger numbers other competing risks.  
 x              n\*p matrix of covariates.  
 model          survival model as returned by fit.fun as used in call to peperr.  
 cplx           numeric, number of boosting steps or list, containing number of boosting steps in argument stepno.  
 times          vector of evaluation time points. If given, used as well as in calculation of full apparent and no-information error as in resampling procedure. Not used if fullsample.attr is specified.  
 type           character.  
 fullsample.attr      vector of evaluation time points, passed in resampling procedure. Either user-defined, if times were passed as args.aggregation, or the determined time points from the aggregation.fun call with the full data set.  
 ...            additional arguments passed to pmppec call.

**Details**

If no evaluation time points are passed, they are generated using all uncensored time points if their number is smaller than 100, or 100 time points up to the 95% quantile of the uncensored time points are taken.

pmpec requires a predictProb method for the class of the fitted model, i.e. for a model of class `class predictProb.class`.

**Value**

A matrix with one row. Each column represents the estimated prediction error of the fit at the time points.

**See Also**

peperr, predictProb, pmpec

---

complexity.cv.penalized

*Complexity selection helpers for penalized backends*

---

**Description**

Wrapper functions for choosing the **penalized** tuning parameter used by `fit.penalized`, `fit.LASSO`, and `fit.fusedLASSO`.

**Usage**

```
complexity.cv.penalized(response, x, full.data, ...)
complexity.LASSO(response, x, full.data, ...)
complexity.fusedLASSO(response, x, full.data, ...)
```

**Arguments**

response	response used for model fitting. Survival responses may be passed as a <code>Surv</code> object or as a two-column matrix with time and status.
x	n * p matrix of covariates.
full.data	data frame containing the full data set. This argument is accepted to match the peperr complexity-function contract.
...	additional arguments passed to <a href="#">profl1</a> .

**Details**

`complexity.cv.penalized` profiles the cross-validated `lambda1` values returned by [profl1](#) and keeps the maximizer. `complexity.LASSO` is a compatibility alias. `complexity.fusedLASSO` performs the same search while forwarding `fused1 = TRUE`.

**Value**

A scalar tuning value, typically the selected lambda1.

**See Also**

[fit.penalized](#), [fit.LASSO](#), [fit.fusedLASSO](#), [profl1](#)

---

complexity.ipec.CoxBoost

*Interface function for CoxBoost complexity selection via integrated prediction error curves*

---

**Description**

Determines the number of boosting steps for a survival model fitted by CoxBoost via integrated prediction error curve estimates, conforming to the calling convention required by argument `complexity` in `peperr` call.

**Usage**

```
complexity.ipec.CoxBoost(response, x, boot.n.c = 10, boost.steps = 100,
  eval.times = NULL, smooth = FALSE, full.data, ...)
```

```
complexity.ripec.CoxBoost(response, x, boot.n.c = 10, boost.steps = 100,
  eval.times = NULL, smooth = FALSE, full.data, ...)
```

**Arguments**

<code>response</code>	a survival object (with <code>Surv(time, status)</code> ).
<code>x</code>	<code>n</code> * <code>p</code> matrix of covariates.
<code>boot.n.c</code>	number of bootstrap samples.
<code>boost.steps</code>	maximum number of boosting steps, i.e. number of boosting steps is selected out of the interval <code>[1, boost.steps]</code> .
<code>eval.times</code>	vector of evaluation time points.
<code>smooth</code>	logical. Shall the prediction error curve be smoothed by local polynomial regression before integration?
<code>full.data</code>	data frame containing response and covariates of the full data set.
<code>...</code>	additional arguments passed to CoxBoost call.

**Details**

Plotting the .632+ estimator for each time point given in `eval.times` results in a prediction error curve. A summary measure can be obtained by integrating over time. `complexity.ripec.CoxBoost` computes a Riemann integral, while `complexity.ipec.CoxBoost` uses a Lebesgue-like integral taking Kaplan-Meier estimates as weights. The number of boosting steps for which the minimal integrated error is obtained is returned.

Since **CoxBoost** is only suggested by **peperr**, install it before calling these functions. If `smooth = TRUE`, **locfit** is also required.

**Value**

Scalar value giving the number of boosting steps.

**See Also**

`peperr`, [CoxBoost](#)

---

`complexity.mincv.CoxBoost`

*Interface for CoxBoost selection of the optimal number of boosting steps via cross-validation*

---

**Description**

Determines the number of boosting steps for a survival model fitted by CoxBoost via cross-validation, conforming to the calling convention required by argument `complexity` in `peperr` call.

**Usage**

```
complexity.mincv.CoxBoost(response, x, full.data, ...)
```

**Arguments**

<code>response</code>	a survival object ( <code>Surv(time, status)</code> ).
<code>x</code>	$n \times p$ matrix of covariates.
<code>full.data</code>	data frame containing response and covariates of the full data set.
<code>...</code>	additional arguments passed to <code>cv.CoxBoost</code> call.

**Details**

Function is basically a wrapper around `cv.CoxBoost` of package **CoxBoost**. A K-fold cross-validation (default  $K = 10$ ) is performed to search the optimal number of boosting steps, per default in the interval  $[0, \text{maxstepno}]$ . Calling `peperr`, the default arguments of `cv.CoxBoost` can be changed by passing a named list containing these as argument `args.complexity`.

Since **CoxBoost** is only suggested by **peperr**, install it before calling this function.

**Value**

Scalar value giving the optimal number of boosting steps.

**See Also**

peperr, [cv.CoxBoost](#)

---

extract.fun	<i>Extract functions, libraries and global variables to be loaded onto a compute cluster</i>
-------------	--

---

**Description**

Automatic extraction of functions, libraries and global variables employed passed functions. Designed for peperr call, see Details section there.

**Usage**

```
extract.fun(funs = NULL)
```

**Arguments**

funs            list of function names.

**Details**

This function is necessary for compute cluster situations where for computation on nodes required functions, libraries and variables have to be loaded explicitly on each node. Avoids loading of whole global environment which might include the unnecessary loading of huge data sets.

It might have problems in some cases, especially it is not able to extract the library of a function that has no namespace. Similarly, it can only extract a required library if it is loaded, or if the function contains a require or library call.

**Value**

list containing

packages	vector containing quoted names of libraries
functions	vector containing quoted names of functions
variables	vector containing quoted names of global variables

**See Also**

peperr

**Examples**

```
# 1. Simplified example for illustration
## Not run:
library(CoxBoost)
a <- function(){
# some calculation
}

b<-function(){
# some other calculation
x <- cv.CoxBoost()
# z is global variable
y <- a(z)
}

# list with packages, functions and variables required for b:
extract.fun(list(b))

# 2. As called by default in peperr example
extract.fun(list(fit.CoxBoost, aggregation.pmpec))

## End(Not run)
```

---

fit.CoxBoost

*Interface function for fitting a CoxBoost model*


---

**Description**

Interface for fitting survival models by CoxBoost, conforming to the requirements for argument `fit.fun` in `peperr` call.

**Usage**

```
fit.CoxBoost(response, x, cplx, ...)
```

**Arguments**

<code>response</code>	a survival object (with <code>Surv(time, status)</code> ).
<code>x</code>	<code>n</code> * <code>p</code> matrix of covariates.
<code>cplx</code>	number of boosting steps, or a list containing at least <code>stepno</code> and optionally further CoxBoost tuning arguments.
<code>...</code>	additional arguments passed to CoxBoost call.

**Details**

Function is basically a wrapper around CoxBoost of package **CoxBoost**. A Cox proportional hazards model is fitted by componentwise likelihood based boosting, especially suited for models with many covariates and few observations.

Since **CoxBoost** is only suggested by **peperr**, install it before calling this function.

**Value**

CoxBoost object

**See Also**

peperr, [CoxBoost](#)

---

fit.coxph

*Interface function for fitting a Cox proportional hazards model*

---

**Description**

Interface for fitting survival models by Cox proportional hazards model, conforming to the requirements for argument `fit.fun` in `peperr` call.

**Usage**

```
fit.coxph(response, x, cplx, ...)
```

**Arguments**

response	a survival object (with <code>Surv(time, status)</code> ).
x	n*p matrix of covariates.
cplx	not used.
...	additional arguments passed to <code>coxph</code> call.

**Details**

Function is basically a wrapper around `coxph` of package **survival**.

**Value**

coxph object

**See Also**

peperr, [coxph](#)

---

fit.glmnet	<i>glmnet Cox backend</i>
------------	---------------------------

---

## Description

Wrappers that integrate **glmnet** Cox models into the peperr fit/complexity/prediction interface.

## Usage

```
fit.glmnet(response, x, cplx, ...)
complexity.cv.glmnet(response, x, full.data, ...)
## S3 method for class 'coxnet'
predictProb(object, response, x, times, complexity = NULL, ...)
## S3 method for class 'coxnet'
PLL(object, newdata, newtime, newstatus, complexity = NULL, ...)
```

## Arguments

response	survival response as a Surv object or a two-column time/status matrix.
x	covariate matrix.
cplx	selected lambda value for glmnet.
full.data	full data frame, accepted for the peperr complexity-function contract.
object	a fitted coxnet object.
times	evaluation times for survival probabilities.
complexity	selected lambda value.
newdata	new covariate matrix.
newtime	vector of follow-up times.
newstatus	vector of event indicators.
...	additional arguments passed to <a href="#">glmnet</a> , <a href="#">cv.glmnet</a> , <a href="#">survfit.coxnet</a> , or <a href="#">predict</a> .

## Details

The backend stores the training design matrix and survival outcome on the fitted object so that the `survfit.coxnet` method can be reused later when `pmpec` calls `predictProb`.

## Value

`fit.glmnet` returns a fitted coxnet object. `complexity.cv.glmnet` returns a scalar lambda. `predictProb.coxnet` returns an  $n * \text{length}(\text{times})$  survival-probability matrix. `PLL.coxnet` returns a numeric predictive partial log-likelihood.

## See Also

[peperr](#), [predictProb](#), [PLL](#), [glmnet](#), [cv.glmnet](#)

---

fit.grpsurv	<i>grpreg grouped Cox backend</i>
-------------	-----------------------------------

---

## Description

Wrappers for grouped survival models fitted with **grpreg**.

## Usage

```
fit.grpsurv(response, x, cplx, group, ...)
complexity.cv.grpsurv(response, x, full.data, group, ...)
## S3 method for class 'grpsurv'
predictProb(object, response, x, times, complexity = NULL, ...)
## S3 method for class 'grpsurv'
PLL(object, newdata, newtime, newstatus, complexity = NULL, ...)
```

## Arguments

response	survival response as a Surv object or a two-column time/status matrix.
x	covariate matrix.
cplx	selected lambda value.
group	group membership vector for the columns of x.
full.data	full data frame, accepted for the peperr complexity-function contract.
object	a fitted grpsurv object.
times	evaluation times for survival probabilities.
complexity	selected lambda value.
newdata	new covariate matrix.
newtime	vector of follow-up times.
newstatus	vector of event indicators.
...	additional arguments passed to <a href="#">grpsurv</a> , <a href="#">cv.grpreg</a> , or <a href="#">predict</a> .

## Details

The grouped backend exposes grouped lasso, grouped MCP, grouped SCAD, and related penalties through the native **grpreg** arguments passed in ...

## Value

Fitted grpsurv objects, scalar lambda values, survival-probability matrices, and numeric predictive partial log-likelihood values, respectively.

## See Also

[peperr](#), [grpsurv](#), [cv.grpreg](#)

---

fit.mboost.coxph      *mboost Cox backend*


---

## Description

Wrappers for Cox boosting models fitted with **mboost**.

## Usage

```
fit.mboost.coxph(response, x, cplx, ...)
complexity.cvrisk.mboost(response, x, full.data, max.mstop = 100L, folds = NULL, ...)
## S3 method for class 'mboost_coxph'
predictProb(object, response, x, times, complexity = NULL, ...)
## S3 method for class 'mboost_coxph'
PLL(object, newdata, newtime, newstatus, complexity = NULL, ...)
```

## Arguments

response	survival response as a Surv object or a two-column time/status matrix.
x	covariate matrix.
cplx	selected stopping iteration mstop.
full.data	full data frame, accepted for the peperr complexity-function contract.
max.mstop	maximum number of boosting iterations considered during cross-validation.
folds	optional <b>mboost</b> fold specification passed to cvrisk.
object	a fitted mboost_coxph object.
times	evaluation times for survival probabilities.
complexity	selected stopping iteration.
newdata	new covariate matrix.
newtime	vector of follow-up times.
newstatus	vector of event indicators.
...	additional arguments passed to <a href="#">glmboost</a> , <a href="#">cvrisk</a> , or prediction helpers.

## Details

If **mboost** provides native survival-curve predictions for the fitted model, `predictProb.mboost_coxph` uses them. Otherwise it falls back to a Breslow baseline estimated from the stored training data and linear predictors.

## Value

Fitted `mboost_coxph` objects, selected stopping iterations, survival-probability matrices, and numeric predictive partial log-likelihood values, respectively.

## See Also

[peperr](#), [glmboost](#), [cvrisk](#)

---

fit.ncvsurv	<i>ncvreg survival backend</i>
-------------	--------------------------------

---

### Description

Wrappers for Cox models fitted with **ncvreg**.

### Usage

```
fit.ncvsurv(response, x, cplx, ...)
complexity.cv.ncvsurv(response, x, full.data, ...)
## S3 method for class 'ncvsurv'
predictProb(object, response, x, times, complexity = NULL, ...)
## S3 method for class 'ncvsurv'
PLL(object, newdata, newtime, newstatus, complexity = NULL, ...)
```

### Arguments

response	survival response as a Surv object or a two-column time/status matrix.
x	covariate matrix.
cplx	selected lambda value.
full.data	full data frame, accepted for the peperr complexity-function contract.
object	a fitted ncvsurv object.
times	evaluation times for survival probabilities.
complexity	selected lambda value.
newdata	new covariate matrix.
newtime	vector of follow-up times.
newstatus	vector of event indicators.
...	additional arguments passed to ncvSurv, cv.ncvsurv, or predict.

### Details

The implementation first tries the native **ncvreg** survival prediction method. If that is unavailable for the installed version, it falls back to a Breslow baseline built from stored training data and linear predictors.

### Value

Fitted ncvsurv objects, selected lambda values, survival-probability matrices, and numeric predictive partial log-likelihood values, respectively.

### See Also

[peperr](#), [predictProb](#), [PLL](#)

---

fit.penalized                      *Penalized backend fitting wrappers*

---

## Description

Interface functions for fitting **penalized** models in peperr. The generic wrapper is `fit.penalized`; `fit.LASSO` and `fit.fusedLASSO` are convenience aliases for plain and fused lasso fits.

## Usage

```
fit.penalized(response, x, cplx, ...)  
fit.LASSO(response, x, cplx, ...)  
fit.fusedLASSO(response, x, cplx, ...)
```

## Arguments

response	response. This can be a binary vector, a numeric response, a Surv object, or a two-column matrix with time and status.
x	n * p matrix of covariates.
cplx	complexity value. For the <b>penalized</b> backend this is usually <code>lambda1</code> , or a named list of tuning arguments.
...	additional arguments passed to <a href="#">penalized</a> .

## Details

`fit.penalized` is the general wrapper around [penalized](#). For survival models it also stores the training design matrix and response so that `predictProb.penfit` and `PLL.penfit` can be used later by `pmpec` and `peperr`.

`fit.LASSO` simply calls `fit.penalized`. `fit.fusedLASSO` calls the same backend with `fused1 = TRUE`.

## Value

A fitted `penfit` object.

## See Also

[complexity.cv.penalized](#), [predictProb.penfit](#), [PLL.penfit](#), [penalized](#)

---

 fit.rfsrc

*randomForestSRC survival backend*


---

### Description

Wrappers for random survival forests fitted with **randomForestSRC**.

### Usage

```
fit.rfsrc(response, x, cplx, ...)
complexity.oob.rfsrc(response, x, full.data,
  mtry = unique(pmax(1L, c(floor(sqrt(ncol(x))), floor(ncol(x)/3), ncol(x))),
  nodesize = c(5L, 15L), ntree = 200L, ...)
## S3 method for class 'rfsrc'
predictProb(object, response, x, times, complexity = NULL, ...)
```

### Arguments

response	survival response as a Surv object or a two-column time/status matrix.
x	covariate matrix.
cplx	either a selected mtry value or a named list such as <code>list(mtry = ..., nodesize = ...)</code> .
full.data	full data frame, accepted for the peperr complexity-function contract.
mtry	candidate mtry values for OOB tuning.
nodesize	candidate terminal-node sizes for OOB tuning.
ntree	number of trees used during OOB tuning.
object	a fitted rfsrc object.
times	evaluation times for survival probabilities.
complexity	unused for fitted forests; included for interface compatibility.
...	additional arguments passed to <a href="#">rfsrc</a> or <a href="#">predict.rfsrc</a> .

### Details

No `PLL.rfsrc` method is provided, because random survival forests do not naturally expose a Cox-style partial likelihood. They can still be used with `pmpec` through `predictProb.rfsrc`.

### Value

Fitted `rfsrc` objects, tuning lists containing `mtry` and `nodesize`, and survival-probability matrices.

### See Also

[peperr](#), [rfsrc](#), [predict.rfsrc](#)

---

fit.SGL.cox	<i>Sparse-group-lasso Cox backend</i>
-------------	---------------------------------------

---

## Description

Wrappers for Cox models fitted with **SGL**.

## Usage

```
fit.SGL.cox(response, x, cplx, index, ...)
complexity.cvSGL.cox(response, x, full.data, index, ...)
## S3 method for class 'SGL_cox'
predictProb(object, response, x, times, complexity = NULL, ...)
## S3 method for class 'SGL_cox'
PLL(object, newdata, newtime, newstatus, complexity = NULL, ...)
```

## Arguments

response	survival response as a Surv object or a two-column time/status matrix.
x	covariate matrix.
cplx	selected lambda value.
index	group membership vector for the columns of x.
full.data	full data frame, accepted for the peperr complexity-function contract.
object	a fitted SGL_cox object.
times	evaluation times for survival probabilities.
complexity	selected lambda value.
newdata	new covariate matrix.
newtime	vector of follow-up times.
newstatus	vector of event indicators.
...	additional arguments passed to <a href="#">SGL</a> or <a href="#">cvSGL</a> .

## Details

The **SGL** predictor returns relative risks. The peperr wrapper reconstructs survival probabilities from those risks with a Breslow baseline estimated on the stored training data.

## Value

Fitted SGL\_cox objects, selected lambda values, survival-probability matrices, and numeric predictive partial log-likelihood values, respectively.

## See Also

[peperr](#), [SGL](#), [cvSGL](#)

---

ipecc *Integrated prediction error curve*

---

## Description

Summary measures of prediction error curves

## Usage

```
ipecc(pe, eval.times, type=c("Riemann", "Lebesgue", "relativeLebesgue"), response=NULL)
```

## Arguments

pe	prediction error at different time points. Vector of length of eval.times or matrix (columns correspond to evaluation time points, rows to different prediction error estimates)
eval.times	evaluation time points
type	type of integration. 'Riemann' estimates Riemann integral, 'Lebesgue' uses the probability density as weights, while 'relativeLebesgue' delivers the difference to the null model (using the same weights as for 'Lebesgue').
response	survival object (Surv(time, status)), required only if type is 'Lebesgue' or 'relativeLebesgue'

## Details

For survival data, prediction error at each evaluation time point can be extracted of a peperr object by function perr. A summary measure can then be obtained via integrating over time. Note that the time points used for evaluation are stored in list element attribute of the peperr object.

## Value

ipecc Value of integrated prediction error curve. Integer or vector, if pe is vector or matrix, respectively, i.e. one entry per row of the passed matrix.

## See Also

[perr](#)

## Examples

```
## Not run:
n <- 200
p <- 100
beta <- c(rep(1,10),rep(0,p-10))
x <- matrix(rnorm(n*p),n,p)
real.time <- -(log(runif(n)))/(10*exp(drop(x %*% beta)))
cens.time <- rexp(n,rate=1/10)
```

```

status <- ifelse(real.time <= cens.time,1,0)
time <- ifelse(real.time <= cens.time,real.time,cens.time)

# Example:
# Obtain prediction error estimate fitting a Cox proportional hazards model
# using CoxBoost
# through 10 bootstrap samples
# with fixed complexity 50 and 75
# and aggregate using prediction error curves
peperr.object <- peperr(response=Surv(time, status), x=x,
  fit.fun=fit.CoxBoost, complexity=c(50, 75),
  indices=resample.indices(n=length(time), method="sub632", sample.n=10))
# 632+ estimate for both complexity values at each time point
prederr <- perr(peperr.object)
# Integrated prediction error curve for both complexity values
ipecc(prederr, eval.times=peperr.object$attribute, response=Surv(time, status))

## End(Not run)

```

---

peperr

*Parallelised Estimation of Prediction Error*


---

## Description

Prediction error estimation for regression models via resampling techniques. Potentially parallelised, if compute cluster is available.

## Usage

```

peperr(response, x,
  indices = NULL,
  fit.fun, complexity = NULL, args.fit = NULL, args.complexity = NULL,
  parallel = NULL, cpus = 2, clustertype=NULL, clusterhosts=NULL,
  noclusterstart = FALSE, noclusterstop=FALSE,
  aggregation.fun=NULL, args.aggregation = NULL,
  load.list = extract.fun(list(fit.fun, complexity, aggregation.fun)),
  load.vars = NULL, load.all = FALSE,
  trace = FALSE, debug = FALSE,
  peperr.lib.loc=NULL,
  RNG=c("RNGstream", "SPRNG", "fixed", "none"), seed=NULL,
  lb=FALSE, sr=FALSE, sr.name="default", sr.restore=FALSE)

```

## Arguments

**response** Either a survival object (with `Surv(time, status)`, where `time` is an `n`-vector of censored survival times and `status` an `n`-vector containing event status, coded with 0 and 1) or a matrix with columns `time` containing survival times and `status` containing integers, where 0 indicates censoring, 1 the interesting event and larger numbers other competing risks. In case of binary response, vector with entries 0 and 1.

<code>x</code>	<code>n*p</code> matrix of covariates.
<code>indices</code>	named list, with two elements (both expected to be lists) <code>sample.index</code> , containing the vector of indices of observations used to fit the model, and list <code>not.in.sample</code> , containing the vector of indices of observations used for assessment. One list entry per split. Function <code>resample.indices</code> provides the most common resampling methods. If argument <code>indices</code> is not specified (default), the indices are determined as follows: If number of observations in the passed data matrix is smaller than number of covariates, 500 bootstrap samples without replacement are generated ("subsampling"), else 500 bootstrap samples with replacement.
<code>fit.fun</code>	function returning a fitted model, see Details.
<code>complexity</code>	if the choice of a complexity parameter is necessary, for example the number of boosting steps in boosting techniques, a function returning complexity parameter for model fitted with <code>fit.fun</code> , see Details. Alternatively, one explicit value for the complexity or a vector of values can be passed. In the latter case, the model fit is carried out for each of the complexity parameters. Alternatively, a named list can be passed, if complexity is a tuple of different parameter values.
<code>args.fit</code>	named list of arguments to be passed to the function given in <code>fit.fun</code> .
<code>args.complexity</code>	if <code>complexity</code> is a function, a named list of arguments to be passed to this function.
<code>parallel</code>	the default setting corresponds to the case that <code>sfCluster</code> is used or if R runs sequential, i.e. without any parallelisation. If <code>sfCluster</code> is used, settings from <code>sfCluster</code> commandline call are taken, i.e. the required number of nodes has to be specified as option of the <code>sfCluster</code> call (and not using argument <code>cpus</code> ). If another cluster solution (specified by argument <code>clustertype</code> ) shall be used, a cluster with <code>cpus</code> CPUs is started if <code>parallel=TRUE</code> . <code>parallel=FALSE</code> switches back to sequential execution. See Details.
<code>cpus</code>	number of nodes, i.e., number of parallel running R processes, to be set up in a cluster, if not specified by commandline call. Only needed if <code>parallel=TRUE</code> .
<code>clustertype</code>	type of cluster, character. 'SOCK' for socket cluster, 'MPI', 'PVM' or 'NWS'. Only considered if <code>parallel=TRUE</code> . If so, a socket cluster, which does not require any additional installation, is started as default.
<code>clusterhosts</code>	host list for socket and NWS clusters, if <code>parallel=TRUE</code> . Has to be specified only if using more than one machine.
<code>noclusterstart</code>	if function is used in already parallelised code. If set to <code>TRUE</code> , no cluster is initialised even if a compute cluster is available and function works in sequential mode. Additionally usable if calls on the slaves should be executed before calling function <code>peperr</code> , for example to load data on slaves, see Details.
<code>noclusterstop</code>	if <code>TRUE</code> , cluster stop is suppressed. Useful for debugging of sessions on slaves. Note that the next <code>peperr</code> call forces cluster stop, except if called with <code>noclusterstart=TRUE</code> .
<code>aggregation.fun</code>	function that evaluates the prediction error for a model fitted by the function given in <code>fit.fun</code> , see Details. If not specified, function <code>aggregation.pmppec</code> is taken if response is survival object, in case of binary response function <code>aggregation.brier</code> .

<code>args.aggregation</code>	named list of arguments to be passed to the function given in argument <code>aggregation.fun</code> .
<code>load.list</code>	a named list with element packages, functions and variables containing quoted names of libraries, functions and global variables required for computation on cluster nodes. The default extracts automatically the libraries, functions and global variables of the, potentially user-defined, functions <code>fit.fun</code> , <code>complexity</code> and <code>aggregation.fun</code> , see function <code>extract.fun</code> . Can be set to <code>NULL</code> , e.g. if no libraries, functions and variables are needed. Alternatively, use argument <code>load.all</code> . See Details.
<code>load.vars</code>	a named list with global variables required for computation on cluster nodes. See Details. Relict, global variables can now be passed as list element variables of argument <code>load.list</code> .
<code>load.all</code>	logical. If set to <code>TRUE</code> , all variables, functions and libraries of the current global environment are loaded on cluster nodes. See Details.
<code>trace</code>	logical. If <code>TRUE</code> , output about the current execution step is printed (if running parallel: printed on nodes, that means not visible in master R process, see Details).
<code>debug</code>	if <code>TRUE</code> , information concerning export of variables is given.
<code>peperr.lib.loc</code>	location of package <b>peperr</b> if not in standard library search path ( <code>.libPaths()</code> ), to be specified for loading <b>peperr</b> onto the cluster nodes.
<code>RNG</code>	type of RNG. "fixed" requires a specified seed. "RNGstream" uses its default seed if not specified. "SPRNG" is retained only for backward compatibility and now triggers an error explaining that it is no longer supported. See Details.
<code>seed</code>	seed to allow reproducibility of results. Only considered if argument <code>RNG</code> is not "none". See Details.
<code>lb</code>	if <code>TRUE</code> and a compute cluster is used, computation of slaves is executed load balanced. See Details.
<code>sr</code>	if <code>TRUE</code> , intermediate results are saved. If execution is interrupted, they can be restored by setting argument <code>sr.restore</code> to <code>TRUE</code> . See documentation of package <b>snowfall</b> for details
<code>sr.name</code>	if <code>sr</code> is set to <code>TRUE</code> and more than one computation runs simultaneously, unique names need to be used.
<code>sr.restore</code>	if <code>sr</code> is set to <code>TRUE</code> , an interrupted computation is restarted.

## Details

Validation of new model fitting approaches requires the proper use of resampling techniques for prediction error estimation. Especially in high-dimensional data situations the computational demand might be huge. `peperr` accelerates computation through automatic parallelisation of the resampling procedure, if a compute cluster is available. A noticeable speed-up is reached even when using a dual-core processor.

Resampling based prediction error estimation requires for each split in training and test data the following steps: a) selection of model complexity (if desired), using the training data set, b) fitting the model with the selected (or a given) complexity on the training set and c) measurement of prediction error on the corresponding test set.

Functions for fitting the model, determination of model complexity, if required by the fitting procedure, and aggregating the prediction error are passed as arguments `fit.fun`, `complexity` and `aggregation.fun`. Already available functions are

for model fit: `fit.CoxBoost`, `fit.coxph`, `fit.penalized`, `fit.LASSO`, `fit.fusedLASSO`, `fit.glmnet`, `fit.grpsurv`, `fit.ncvsurv`, `fit.rfsrc`, `fit.mboost.coxph`, `fit.SGL.cox`

to determine complexity: `complexity.mincv.CoxBoost`, `complexity.ipec.CoxBoost`, `complexity.ripec.CoxBoost`, `complexity.cv.penalized`, `complexity.LASSO`, `complexity.fusedLASSO`, `complexity.cv.glmnet`, `complexity.cv.grpsurv`, `complexity.cv.ncvsurv`, `complexity.oob.rfsrc`, `complexity.cvrisk.mboost`, `complexity.cvSGL.cox`

to aggregate prediction error: `aggregation.pmpec`, `aggregation.brier`, `aggregation.misclass`

Function `peperr` is especially designed for evaluation of newly developed model fitting routines. For that, own routines can be passed as arguments to the `peperr` call. They are incorporated as follows (also compare existing functions, as named above):

1. Model fitting techniques, which require selection of one or more complexity parameters, often provide routines based on cross-validation or similar to determine this parameter. If this routine is already at hand, the complexity function needed for the `peperr` call is not more than a wrapper around that, which consists of providing the data in the required way, calling the routine and return the selected complexity value(s).
2. For a given model fitting routine the fitting function, which is passed to the `peperr` call as argument `fit.fun`, is not more than a wrapper around that. Explicitly, response and matrix of covariates have to be transformed to the required form, if necessary, the routine is called with the passed complexity value, if required, and the fitted prediction model is returned.
3. Prediction error is estimated using a fitted model and a data set, by any kind of comparison of the true and the predicted response values. In case of survival response, apparent error (type `apparent`), which means that the prediction error is estimated in the same data set as used for model fitting, and no-information error (type `noinf`), which calculates the prediction error in permuted data, have to be provided. Note that the aggregation function returns the error with an additional attribute called `addattr`. The evaluation time points have to be stored there to allow later access.
4. In case of survival response, the user may additionally provide a function for partial log likelihood calculation, if he uses an own function for model fit, called `PLL.class`. If prediction error curves are used for aggregation (`aggregation.pmpec`), a `predictProb` method has to be provided, i.e. for each model of class `class predictProb.class`, see there.

Concerning parallelisation, there are three possibilities to run `peperr`:

- Start R on commandline with `sfCluster` and preferred options, for example number of `cpus`. Leave the three arguments `parallel`, `clustertype` and `nodes` unchanged.
- Use any other cluster solution supported by **snowfall**, i.e. LAM/MPI, socket, PVM, NWS (set argument `clustertype`). Argument `parallel` has to be set to `TRUE` and number of `cpus` can be chosen by argument `nodes`)
- If no cluster is used, R works sequentially. Keep `parallel=NULL`. No parallelisation takes place and therefore no speed up can be obtained.

In general, if `parallel=NULL`, all information concerning the cluster set-up is taken from commandline, else, it can be specified using the three arguments `parallel`, `clustertype`, `nodes`, and, if necessary, `clusterhosts`.

`sfCluster` is a Unix tool for flexible and comfortable management of parallel R processes. However, **peperr** is usable with any other cluster solution supported by **snowfall**, i.e. `sfCluster` has not to be installed to use package **peperr**. Note that this may require cluster handling by the user, e.g. manually shut down with `'lamhalt'` on commandline for `type="MPI"`. But, using a socket cluster (argument `parallel=TRUE` and `clustertype="SOCK"`), does not require any extra installation.

Note that the run time cannot speed up anymore if the number of nodes is chosen higher than the number of passed training/test samples plus one, as parallelisation takes place in the resampling procedure and one additional run is used for computation on the full sample.

If not running in sequential mode, a specified number of R processes called `nodes` is spawned for parallel execution of the resampling procedure (see above). This requires to provide all variables, functions and libraries necessary for computation on each of these R processes, so explicitly all variables, functions and libraries required by the, potentially user-defined, functions `fit.fun`, `complexity` and `aggregation.fun`. The simplest possibility is to load the whole content of the global environment on each node and all loaded libraries. This is done by setting argument `load.all=TRUE`. This is not the default, as a huge amount of data is potentially loaded to each node unnecessarily. Function `extract.fun` is provided to extract the functions and libraries needed, automatically called at each call of function `peperr`. Note that all required libraries have to be located in the standard library search path (obtained by `.libPaths()`). Another alternative is to load required data manually on the slaves, using **snowfall** functions `sfLibrary`, `sfExport` and `sfExportAll`. Then, argument `noclusterstart` has to be switched to `TRUE`. Additionally, argument `load.list` could be set to `NULL`, to avoid potentially overwriting of functions and variables loaded to the cluster nodes automatically.

Note that a `set.seed` call before calling function `peperr` is not sufficient to allow reproducibility of results when running in parallel mode, as the slave R processes are not affected as they are own R instances. `peperr` provides two possibilities to make results reproducible:

- Use `RNG="RNGstream"` for independent parallel random number streams initialized on the cluster nodes via `sfClusterSetupRNG` from package **snowfall**. This requires package **rlcuyer**. A seed can be specified using argument `seed`, else the default values are taken. A `set.seed` call on the master is required additionally and argument `lb=FALSE`, see below.
- `RNG="SPRNG"` is kept as a recognized legacy value so existing code fails with an informative message, but it is no longer supported because **rsprng** has been removed from CRAN.
- If `RNG="fixed"`, a seed has to be specified. This can be either an integer or a vector of length number of samples +2. In the second case, the first entry is used for the main R process, the next number of samples ones for each sample run (in parallel execution mode on slave R processes) and the last one for computation on full sample (as well on slave R process in parallel execution mode). Passing integer `x` is equivalent to passing vector `x+(0:(number of samples+1))`. This procedure allows reproducibility in any case, i.e. also if the number of parallel processes changes as well as in sequential execution.

Load balancing (argument `lb`) means, that a slave gets a new job immediately after the previous is finished. This speeds up computation, but may change the order of jobs. Due to that, results are only reproducible, if `RNG="fixed"` is used.

**Value**

Object of class peperr

<code>indices</code>	list of resampling indices.
<code>complexity</code>	passed complexity. If argument <code>complexity</code> not specified, 0.
<code>selected.complexity</code>	selected complexity for the full data set, if <code>complexity</code> was passed as function. Else equal to value <code>complexity</code> .
<code>response</code>	passed response.
<code>full.model.fit</code>	List, one entry per complexity value. Fitted model of the full data set by passed <code>fit.fun</code> .
<code>full.apparent</code>	full apparent error of the full data set. Matrix: One row per complexity value. In case of survival response, columns correspond to evaluation timepoints, which are returned in value <code>attribute</code> .
<code>noinf.error</code>	No information error of the full data set, i. e. evaluation in permuted data. Matrix: One row per complexity value. Columns correspond to evaluation timepoints, which are returned in <code>attribute</code> .
<code>attribute</code>	if response is survival: Evaluation time points. Passed in <code>args.aggregation</code> or automatically determined by aggregation function. Otherwise, if available, extra attribute returned by aggregation function, else NULL, see Details.
<code>sample.error</code>	list. Each entry contains matrix of prediction error for one resampling test sample. One row per complexity value.
<code>sample.complexity</code>	vector of complexity values. Equals <code>value.complexity</code> , if complexity value was passed explicitly, otherwise by function <code>complexity</code> selected complexity value for each resampling sample. If argument <code>complexity</code> not specified, 0.
<code>sample.lipec</code>	only, if response is survival. Lebesgue integrated prediction error curve for each sample. List with one entry per sample, each a matrix with one row per complexity value.
<code>sample.pll</code>	only, if response is survival and <code>PLL.class</code> function available. Predictive partial log likelihood for each sample. List with one entry per sample, each a matrix with one row per complexity value.
<code>null.model.fit</code>	only, if response is survival or binary. Fit of null model, i.e. fit without information of covariates. In case of survival response Kaplan-Meier, else logistic regression model.
<code>null.model</code>	only, if response is survival or binary. Vector or scalar: Prediction error of the null model, in case of survival response at each evaluation time point.
<code>sample.null.model</code>	list. Prediction error of the null model for one resampling test sample. Matrix, one row per complexity value.

**Author(s)**

Christine Porzeliuss <cp@fdm.uni-freiburg.de>, Harald Binder

## References

Binder, H. and Schumacher, M. (2008) Adapting prediction error estimates for biased complexity selection in high-dimensional bootstrap samples. *Statistical Applications in Genetics and Molecular Biology*, 7:1.

Porzelius, C., Binder, H., Schumacher, M. (2008) Parallelised prediction error estimation for evaluation of high-dimensional models. Manuscript.

## See Also

[perr](#), [resample.indices](#), [extract.fun](#)

## Examples

```
# Generate survival data with 10 informative covariates
n <- 200
p <- 100
beta <- c(rep(1,10),rep(0,p-10))
x <- matrix(rnorm(n*p),n,p)
real.time <- -(log(runif(n)))/(10*exp(drop(x %*% beta)))
cens.time <- rexp(n,rate=1/10)
status <- ifelse(real.time <= cens.time,1,0)
time <- ifelse(real.time <= cens.time,real.time,cens.time)

## Not run:
# A: R runs sequential or R is started on commandline with desired options
# (for example using sfCluster: sfCluster -i --cpus=5)
# Example A1:
# Obtain prediction error estimate fitting a Cox proportional hazards model
# using CoxBoost
# through 10 bootstrap samples
# with fixed complexity 50 and 75
# and aggregate using prediction error curves (default setting)

peperr.object1 <- peperr(response=Surv(time, status), x=x,
  fit.fun=fit.CoxBoost, complexity=c(50, 75),
  indices=resample.indices(n=length(time), method="sub632", sample.n=10))
peperr.object1

# Diagnostic plots
plot(peperr.object1)

# Extraction of prediction error curves (.632+ prediction error estimate),
# blue line corresponds to complexity 50,
# red one to complexity 75
plot(peperr.object1$attribute,
  perr(peperr.object1)[1,], type="l", col="blue",
  xlab="Evaluation time points", ylab="Prediction error")
lines(peperr.object1$attribute,
  perr(peperr.object1)[2,], col="red")

# Example A2:
```

```

# As Example A1, but
# with complexity selected through a cross-validation procedure
# and extra argument 'penalty' passed to fit function and complexity function
peperr.object2 <- peperr(response=Surv(time, status), x=x,
  fit.fun=fit.CoxBoost, args.fit=list(penalty=100),
  complexity=complexity.mincv.CoxBoost, args.complexity=list(penalty=100),
  indices=resample.indices(n=length(time), method="sub632", sample.n=10),
  trace=TRUE)
peperr.object2

# Diagnostic plots
plot(peperr.object2)

# Example A3:
# As Example A2, but
# with extra argument 'times', specifying the evaluation times passed to aggregation.fun
# and seed, for reproducibility of results
# Note: set.seed() is required additional to argument 'seed',
# as function 'resample.indices' is used in peperr call.
set.seed(123)
peperr.object3 <- peperr(response=Surv(time, status), x=x,
  fit.fun=fit.CoxBoost, args.fit=list(penalty=100),
  complexity=complexity.mincv.CoxBoost, args.complexity=list(penalty=100),
  indices=resample.indices(n=length(time), method="sub632", sample.n=10),
  args.aggregation=list(times=seq(0, quantile(time, probs=0.9), length.out=100)),
  trace=TRUE, RNG="fixed", seed=321)
peperr.object3

# Diagnostic plots
plot(peperr.object3)

# B: R is started sequential, desired cluster options are given as arguments
# Example B1:
# As example A1, but using a socket cluster and 3 CPUs
peperr.object4 <- peperr(response=Surv(time, status), x=x,
  fit.fun=fit.CoxBoost, complexity=c(50, 75),
  indices=resample.indices(n=length(time), method="sub632", sample.n=10),
  parallel=TRUE, clustertype="SOCK", cpus=3)

## End(Not run)

```

---

perr

*Prediction error estimates*


---

## Description

Extracts prediction error estimates from peperr objects.

**Usage**

```
perr(peperobject,
     type = c("632p", "632", "apparent", "NoInf", "resample", "nullmodel"))
```

**Arguments**

`peperobject`    `peperr` object obtained by call to function `peperr`.

`type`            "632p" for the .632+ prediction error estimate (default), "632" for the .632 prediction error estimate. "apparent", "NoInf", "resample" and "nullmodel" return the apparent error, the no-information error, the mean sample error and the nullmodel fit, see Details.

**Details**

The .632 and the .632+ prediction error estimates are weighted combinations of the apparent error and bootstrap cross-validation error estimate, for survival data at given time points.

**Value**

If `type="632p"` or `type="632"`: Prediction error: Matrix, with one row per complexity value.

If `type="apparent"`: Apparent error of the full data set. Matrix: One row per complexity value. In case of survival response, columns correspond to evaluation timepoints, which are given in attribute `addattr`.

If `type="NoInf"`: No-information error of the full data set, i. e. evaluation in permuted data. Matrix: One row per complexity value. Columns correspond to evaluation timepoints, which are given in attribute `addattr`.

If `type="resample"`: Matrix. Mean prediction error of resampling test samples, one row per complexity value.

If `type="nullmodel"`: Vector or scalar: Null model prediction error, i.e. of fit without information of covariates. In case of survival response Kaplan-Meier estimate at each time point, if response is binary logistic regression model, else not available.

**References**

Binder, H. and Schumacher, M. (2008) Adapting prediction error estimates for biased complexity selection in high-dimensional bootstrap samples. *Statistical Applications in Genetics and Molecular Biology*, 7:1.

Gerds, T. and Schumacher, M. (2007) Efron-type measures of prediction error for survival analysis. *Biometrics*, 63, 1283–1287.

Schumacher, M. and Binder, H., and Gerds, T. (2007) Assessment of Survival Prediction Models in High-Dimensional Settings. *Bioinformatics*, 23, 1768-1774.

**See Also**

[peperr](#), [ipec](#)

**Examples**

```

## Not run:
n <- 200
p <- 100
beta <- c(rep(1,10),rep(0,p-10))
x <- matrix(rnorm(n*p),n,p)
real.time <- -(log(runif(n)))/(10*exp(drop(x %>% beta)))
cens.time <- rexp(n,rate=1/10)
status <- ifelse(real.time <= cens.time,1,0)
time <- ifelse(real.time <= cens.time,real.time,cens.time)

# Example:
# Obtain prediction error estimate fitting a Cox proportional hazards model
# using CoxBoost
# through 10 bootstrap samples
# with fixed complexity 50 and 75
# and aggregate using prediction error curves
peperr.object <- peperr(response=Surv(time, status), x=x,
  fit.fun=fit.CoxBoost, complexity=c(50, 75),
  indices=resample.indices(n=length(time), method="sub632", sample.n=10))
# 632+ estimate for both complexity values at each time point
perr(peperr.object)

## End(Not run)

```

---

 PLL

*Generic function for extracting the predictive partial log-likelihood*


---

**Description**

Generic function for extracting the predictive partial log-likelihood from a fitted survival model.

**Usage**

```
PLL(object, newdata, newtime, newstatus, ...)
```

**Arguments**

object	fitted model of class class.
newdata	n_new*p matrix of covariates.
newtime	n_new-vector of censored survival times.
newstatus	n_new-vector of event status, coded with 0 for censoring and 1, if an event occurred.
...	additional arguments, for example complexity value, if necessary.

**Details**

The predictive partial log-likelihood measures the prediction performance of each model fitted in a bootstrap sample, using the data not in this sample. Multiplying by (-2) leads to a deviance-like measure, which means that small values indicate good prediction performance.

peperr requires function PLL.class in case of survival response, for each model fit of class class. Available methods include PLL.CoxBoost, PLL.coxph, PLL.penfit, PLL.coxnet, PLL.grpsurv, PLL.ncvsurv, PLL.mboost\_coxph, and PLL.SGL\_cox.

**Value**

Numeric predictive partial log-likelihood returned by the method implementation.

---

PLL.CoxBoost	<i>Predictive partial log-likelihood for a CoxBoost model fit</i>
--------------	---

---

**Description**

Extracts the predictive partial log-likelihood from a CoxBoost model fit.

**Usage**

```
## S3 method for class 'CoxBoost'
PLL(object, newdata, newtime, newstatus, complexity, ...)
```

**Arguments**

object	fitted model of class CoxBoost.
newdata	n_new*p matrix of covariates.
newtime	n_new-vector of censored survival times.
newstatus	n_new-vector of survival status, coded with 0 and 1.
complexity	complexity, either one value, which is the number of boosting steps, or a list containing stepno.
...	additional arguments passed to predict.CoxBoost.

**Details**

Used by function peperr, if function fit.CoxBoost is used for model fit.

Since **CoxBoost** is only suggested by **peperr**, install it before calling this function.

**Value**

Numeric predictive partial log-likelihood returned by predict.CoxBoost.

---

 PLL.coxph

*Predictive partial log-likelihood for Cox proportional hazards model*


---

**Description**

Extracts the predictive partial log-likelihood from a coxph model fit.

**Usage**

```
## S3 method for class 'coxph'
PLL(object, newdata, newtime, newstatus, complexity, ...)
```

**Arguments**

object	fitted model of class coxph.
newdata	n_new*p matrix of covariates.
newtime	n_new-vector of censored survival times.
newstatus	n_new-vector of survival status, coded with 0 and .1
complexity	not used.
...	additional arguments, not used.

**Details**

Used by function peperr, if function fit.coxph is used for model fit.

**Value**

Vector of length n\_new

---

 PLL.penfit

*Predictive partial log-likelihood for penalized Cox models*


---

**Description**

Computes the predictive partial log-likelihood for penfit survival models.

**Usage**

```
## S3 method for class 'penfit'
PLL(object, newdata, newtime, newstatus, complexity = NULL, ...)
```

**Arguments**

object	a fitted penfit object.
newdata	new covariate matrix.
newtime	vector of follow-up times.
newstatus	vector of event indicators.
complexity	unused for fitted penfit objects; included for interface compatibility.
...	unused additional arguments.

**Value**

Numeric predictive partial log-likelihood.

**See Also**

[PLL](#), [fit.penalized](#), [predictProb.penfit](#)

---

plot.peperr	<i>Plot method for peperr object</i>
-------------	--------------------------------------

---

**Description**

Plots, allowing to get a first impression of the prediction error estimates and to check complexity selection in bootstrap samples.

**Usage**

```
## S3 method for class 'peperr'
plot(x, y, ...)
```

**Arguments**

x	peperr object.
y	not used.
...	additional arguments, not used.

**Details**

The plots provide a simple and fast overview of the results of the estimation of the prediction error through resampling. Which plots are shown depends on if complexity was selected, i.e., a function was passed in the peperr call for complexity, or explicitly passed. In case of survival response, prediction error curves are shown. In case of binary response, where one complexity value is passed explicitly, no plot is available. Especially in the case that complexity is selected in each bootstrap sample, these diagnostic plots help to check whether the resampling procedure works adequately and to detect specific problems due to high-dimensional data structures.

**Examples**

```

## Not run:
n <- 200
p <- 100
beta <- c(rep(1,10),rep(0,p-10))
x <- matrix(rnorm(n*p),n,p)
real.time <- -(log(runif(n)))/(10*exp(drop(x %*% beta)))
cens.time <- rexp(n,rate=1/10)
status <- ifelse(real.time <= cens.time,1,0)
time <- ifelse(real.time <= cens.time,real.time,cens.time)

peperr.object1 <- peperr(response=Surv(time, status), x=x,
  fit.fun=fit.CoxBoost, complexity=c(50, 75),
  indices=resample.indices(n=length(time), method="sub632", sample.n=10))
plot(peperr.object1)

peperr.object2 <- peperr(response=Surv(time, status), x=x,
  fit.fun=fit.CoxBoost, args.fit=list(penalty=100),
  complexity=complexity.mincv.CoxBoost, args.complexity=list(penalty=100),
  indices=resample.indices(n=length(time), method="sub632", sample.n=10),
  trace=TRUE)
plot(peperr.object2)

peperr.object3 <- peperr(response=Surv(time, status), x=x,
  fit.fun=fit.CoxBoost, args.fit=list(penalty=100),
  complexity=complexity.mincv.CoxBoost, args.complexity=list(penalty=100),
  indices=resample.indices(n=length(time), method="sub632", sample.n=10),
  args.aggregation=list(times=seq(0, quantile(time, probs=0.9), length.out=100)),
  trace=TRUE)
plot(peperr.object3)

## End(Not run)

```

---

pmpec

*Calculate prediction error curves*


---

**Description**

Calculation of prediction error curve from a survival response and predicted probabilities of survival.

**Usage**

```

pmpec(object, response=NULL, x=NULL, times, model.args=NULL,
  type=c("PErr","NoInf"), external.time=NULL, external.status=NULL,
  data=NULL)

```

**Arguments**

<code>object</code>	fitted model of a class for which the interface function <code>predictProb.class</code> is available.
<code>response</code>	Either a survival object (with <code>Surv(time, status)</code> , where <code>time</code> is an <code>n</code> -vector of censored survival times and <code>status</code> an <code>n</code> -vector containing event status, coded with 0 and 1) or a matrix with columns <code>time</code> containing survival times and <code>status</code> containing integers, where 0 indicates censoring, 1 the interesting event and larger numbers other competing risks.
<code>x</code>	<code>n</code> * <code>p</code> matrix of covariates.
<code>times</code>	vector of time points at which the prediction error is to be estimated.
<code>model.args</code>	named list of additional arguments, e.g. <code>complexity</code> value, which are to be passed to <code>predictProb</code> function.
<code>type</code>	type of output: Estimated prediction error (default) or no information error (prediction error obtained by permuting the data).
<code>external.time</code>	optional vector of time points, used for censoring distribution.
<code>external.status</code>	optional vector of status values, used for censoring distribution.
<code>data</code>	Data frame containing <code>n</code> -vector of observed times ( <code>'time'</code> ), <code>n</code> -vector of event status ( <code>'status'</code> ) and <code>n</code> * <code>p</code> matrix of covariates (remaining entries). Alternatively to <code>response</code> and <code>x</code> , for compatibility to <b>pec</b> .

**Details**

Prediction error of survival data is measured by the Brier score, which considers the squared difference of the true event status at a given time point and the predicted event status by a risk prediction model at that time. A prediction error curve is the weighted mean Brier score as a function of time at time points in `times` (see References).

`pmpec` requires a `predictProb` method for the class of the fitted model, i.e. for a model of class `class predictProb.class`.

`pmpec` is implemented to behave similar to function `pec` of package **pec**, which provides several `predictProb` methods.

In bootstrap framework, `data` contains only a part of the full data set. For censoring distribution, the full data should be used to avoid extreme variance in case of small data sets. For that, the observed times and status values can be passed as argument `external.time` and `external.status`.

**Value**

Vector of prediction error estimates at each time point given in `time`.

**Author(s)**

Harald Binder

## References

Gerds, A. and Schumacher, M. (2006) Consistent estimation of the expected Brier score in general survival models with right-censored event times. *Biometrical Journal*, 48, 1029–1040.

Schoop, R. (2008) Predictive accuracy of failure time models with longitudinal covariates. PhD thesis, University of Freiburg. <http://www.freidok.uni-freiburg.de/volltexte/4995/>.

## See Also

predictProb, **pec**

---

predictProb

*Generic function for extracting predicted survival probabilities*

---

## Description

Generic function for extraction of predicted survival probabilities from a fitted survival model conforming to the interface required by `pmpec`.

## Usage

```
predictProb(object, response, x, ...)
```

## Arguments

<code>object</code>	a fitted survival model.
<code>response</code>	Either a survival object (with <code>Surv(time, status)</code> , where <code>time</code> is an <code>n</code> -vector of censored survival times and <code>status</code> an <code>n</code> -vector containing event status, coded with 0 and 1) or a matrix with columns <code>time</code> containing survival times and <code>status</code> containing integers, where 0 indicates censoring, 1 the interesting event and larger numbers other competing risks. In case of binary response, vector with entries 0 and 1.
<code>x</code>	<code>n</code> * <code>p</code> matrix of covariates.
<code>...</code>	additional arguments, for example model complexity or, in case of survival response, argument <code>times</code> , a vector containing evaluation times.

## Details

`pmpec` requires a `predictProb.class` function for each model fit of class `class`. It extracts the predicted probability of survival from this model.

See existing `predictProb` functions, at the time `predictProb.CoxBoost`, `predictProb.coxph`, `predictProb.survfit`, `predictProb.penfit`, `predictProb.coxnet`, `predictProb.grpsurv`, `predictProb.ncvsurv`, `predictProb.rfsrc`, `predictProb.mboost_coxph`, and `predictProb.SGL_cox`.

If desired `predictProb` function for class `class` is not available in **peperr**, but implemented in package **pec** as `predictSurvProb.class`, it can easily be transformed as `predictProb` method.

**Value**

Matrix with predicted probabilities for each evaluation time point in `times` (columns) and each new observation (rows).

---

`predictProb.CoxBoost` *Extract predicted survival probabilities from a CoxBoost fit*

---

**Description**

Extracts predicted survival probabilities from a survival model fitted by `CoxBoost`, providing an interface as required by `pmpec`.

**Usage**

```
## S3 method for class 'CoxBoost'  
predictProb(object, response, x, times, complexity, ...)
```

**Arguments**

<code>object</code>	a fitted model of class <code>CoxBoost</code> .
<code>response</code>	survival object (with <code>Surv(time, status)</code> , where <code>time</code> is an <code>n</code> -vector of censored survival times and <code>status</code> an <code>n</code> -vector containing survival status, coded with 0 and 1).
<code>x</code>	<code>n</code> * <code>p</code> matrix of covariates.
<code>times</code>	vector of evaluation time points.
<code>complexity</code>	complexity value, or a list containing <code>stepno</code> .
<code>...</code>	additional arguments passed to <code>predict.CoxBoost</code> .

**Details**

Since `CoxBoost` is only suggested by `peperr`, install it before calling this function.

**Value**

Matrix with probabilities for each evaluation time point in `times` (columns) and each new observation (rows).

---

predictProb.coxph      *Extract predicted survival probabilities from a coxph object*

---

### Description

Extracts predicted survival probabilities for survival models fitted by Cox proportional hazards model, providing an interface as required by pmpec.

### Usage

```
## S3 method for class 'coxph'
predictProb(object, response, x, times, ...)
```

### Arguments

object	a fitted model of class coxph.
response	survival object (with Surv(time, status), where time is an n-vector of censored survival times and status an n-vector containing survival status, coded with 0 and 1.
x	n*p matrix of covariates.
times	vector of evaluation time points.
...	additional arguments, currently not used.

### Value

Matrix with probabilities for each evaluation time point in times(columns) and each new observation (rows).

---

predictProb.penfit      *Predicted survival probabilities for penalized Cox models*

---

### Description

Extracts survival probabilities from a penfit survival model for use in pmpec.

### Usage

```
## S3 method for class 'penfit'
predictProb(object, response, x, times, complexity = NULL, ...)
```

**Arguments**

object	a fitted penfit object.
response	survival response, passed for consistency with the predictProb interface.
x	new covariate matrix.
times	evaluation times.
complexity	unused for fitted penfit objects; included for interface compatibility.
...	additional arguments passed to the <b>penalized</b> prediction method.

**Value**

Matrix of survival probabilities with one row per observation and one column per evaluation time.

**See Also**

[predictProb](#), [fit.penalized](#), [PLL.penfit](#)

---

predictProb.survfit *Extract predicted survival probabilities from a survfit object*

---

**Description**

Extracts predicted survival probabilities for survival models fitted by `survfit`, providing an interface as required by `pmpec`.

**Usage**

```
## S3 method for class 'survfit'
predictProb(object, response, x, times, train.data, ...)
```

**Arguments**

object	a fitted model of class <code>survfit</code> .
response	survival object (with <code>Surv(time, status)</code> , where <code>time</code> is an <code>n</code> -vector of censored survival times and <code>status</code> an <code>n</code> -vector containing survival status, coded with 0 and 1).
x	<code>n</code> * <code>p</code> matrix of covariates.
times	vector of evaluation time points.
train.data	not used.
...	additional arguments, currently not used.

**Value**

Matrix with probabilities for each evaluation time point in `times`(columns) and each new observation (rows).

---

resample.indices      *Generation of indices for resampling Procedure*

---

### Description

Generates training and test set indices for use in resampling estimation of prediction error, e.g. cross-validation or bootstrap (with and without replacement).

### Usage

```
resample.indices(n, sample.n = 100, method = c("no", "cv", "boot", "sub632"))
```

### Arguments

n	number of observations of the full data set.
sample.n	the number of bootstrap samples in case of method="boot" and the number of cross-validation subsets in case of method="cv", e.g. 10 for 10-fold cross-validation. Not considered if method="no", where number of samples is one (the full data set) by definition.
method	by default, the training set indices are the same as the test set indices, i.e. the model is assessed in the same data as fitted ("no"). "cv": Cross-validation, "boot": Bootstrap (with replacement), "sub632": Bootstrap without replacement, also called subsampling. In the latter case, the number of observations in each sample equals $\text{round}(0.632 * n)$ , see Details.

### Details

As each bootstrap sample should be taken as if new data, complexity selection should be carried out in each bootstrap sample. Binder and Schumacher show that when bootstrap samples are drawn with replacement, often too complex models are obtained in high-dimensional data settings. They recommend to draw bootstrap samples without replacement, each of size  $\text{round}(0.632 * n)$ , which equals the expected number of unique observations in one bootstrap sample drawn with replacement, to avoid biased complexity selection and improve predictive power of the resulting models.

### Value

A list containing two lists of length `sample.n`:

sample.index	contains in each element the indices of observations of one training set.
not.in.sample	contains in each element the indices of observations of one test set, corresponding to the training set in listelement <code>sample.index</code> .

### References

Binder, H. and Schumacher, M. (2008) Adapting prediction error estimates for biased complexity selection in high-dimensional bootstrap samples. *Statistical Applications in Genetics and Molecular Biology*, 7:1.

**See Also**

peperr

**Examples**

```
# generate dataset: 100 patients, 20 covariates
data <- matrix(rnorm(2000), nrow=100)

# generate indices for training and test data for 10-fold cross-validation
indices <- resample.indices(n=100, sample.n = 10, method = "cv")

# create training and test data via indices
trainingsample1 <- data[indices$sample.index[[1]],]
testsample1 <- data[indices$not.in.sample[[1]],]
```

# Index

## \* **models**

- aggregation.brier, 2
- aggregation.misclass, 3
- aggregation.pmpec, 4
- complexity.cv.penalized, 5
- complexity.ipec.CoxBoost, 6
- complexity.mincv.CoxBoost, 7
- extract.fun, 8
- fit.CoxBoost, 9
- fit.coxph, 10
- fit.glmnet, 11
- fit.grpsurv, 12
- fit.mboost.coxph, 13
- fit.ncvsurv, 14
- fit.penalized, 15
- fit.rfsrc, 16
- fit.SGL.cox, 17
- ipec, 18
- peperr, 19
- perr, 26
- PLL, 28
- PLL.CoxBoost, 29
- PLL.coxph, 30
- PLL.penfit, 30
- plot.peperr, 31
- pmpec, 32
- predictProb, 34
- predictProb.CoxBoost, 35
- predictProb.coxph, 36
- predictProb.penfit, 36
- predictProb.survfit, 37
- resample.indices, 38

## \* **regression**

- aggregation.brier, 2
- aggregation.misclass, 3
- aggregation.pmpec, 4
- complexity.cv.penalized, 5
- complexity.ipec.CoxBoost, 6
- complexity.mincv.CoxBoost, 7

- extract.fun, 8
- fit.CoxBoost, 9
- fit.coxph, 10
- fit.glmnet, 11
- fit.grpsurv, 12
- fit.mboost.coxph, 13
- fit.ncvsurv, 14
- fit.penalized, 15
- fit.rfsrc, 16
- fit.SGL.cox, 17
- ipec, 18
- peperr, 19
- perr, 26
- PLL, 28
- PLL.CoxBoost, 29
- PLL.coxph, 30
- PLL.penfit, 30
- plot.peperr, 31
- pmpec, 32
- predictProb, 34
- predictProb.CoxBoost, 35
- predictProb.coxph, 36
- predictProb.penfit, 36
- predictProb.survfit, 37
- resample.indices, 38

## \* **survival**

- aggregation.brier, 2
- aggregation.misclass, 3
- aggregation.pmpec, 4
- complexity.cv.penalized, 5
- complexity.ipec.CoxBoost, 6
- complexity.mincv.CoxBoost, 7
- extract.fun, 8
- fit.CoxBoost, 9
- fit.coxph, 10
- fit.glmnet, 11
- fit.grpsurv, 12
- fit.mboost.coxph, 13
- fit.ncvsurv, 14

- fit.penalized, 15
  - fit.rfsrc, 16
  - fit.SGL.cox, 17
  - ipec, 18
  - peperr, 19
  - perr, 26
  - PLL, 28
  - PLL.CoxBoost, 29
  - PLL.coxph, 30
  - PLL.penfit, 30
  - plot.peperr, 31
  - pmpec, 32
  - predictProb, 34
  - predictProb.CoxBoost, 35
  - predictProb.coxph, 36
  - predictProb.penfit, 36
  - predictProb.survfit, 37
  - resample.indices, 38
- aggregation.brier, 2
- aggregation.misclass, 3
- aggregation.pmpec, 4
- complexity.cv.glmnet (fit.glmnet), 11
- complexity.cv.grpsurv (fit.grpsurv), 12
- complexity.cv.ncvsurv (fit.ncvsurv), 14
- complexity.cv.penalized, 5, 15
- complexity.cvrisk.mboost  
    (fit.mboost.coxph), 13
- complexity.cvSGL.cox (fit.SGL.cox), 17
- complexity.fusedLASSO  
    (complexity.cv.penalized), 5
- complexity.ipec.CoxBoost, 6
- complexity.LASSO  
    (complexity.cv.penalized), 5
- complexity.mincv.CoxBoost, 7
- complexity.oob.rfsrc (fit.rfsrc), 16
- complexity.ripec.CoxBoost  
    (complexity.ipec.CoxBoost), 6
- CoxBoost, 7, 10
- coxph, 10
- cv.CoxBoost, 8
- cv.glmnet, 11
- cv.grpreg, 12
- cvrisk, 13
- cvSGL, 17
- extract.fun, 8, 25
- fit.CoxBoost, 9
- fit.coxph, 10
- fit.fusedLASSO, 6
- fit.fusedLASSO (fit.penalized), 15
- fit.glmnet, 11
- fit.grpsurv, 12
- fit.LASSO, 6
- fit.LASSO (fit.penalized), 15
- fit.mboost.coxph, 13
- fit.ncvsurv, 14
- fit.penalized, 6, 15, 31, 37
- fit.rfsrc, 16
- fit.SGL.cox, 17
- glmboost, 13
- glmnet, 11
- grpsurv, 12
- ipec, 18, 27
- penalized, 15
- peperr, 11–14, 16, 17, 19, 27
- perr, 18, 25, 26
- PLL, 11, 14, 28, 31
- PLL.CoxBoost, 29
- PLL.coxnet (fit.glmnet), 11
- PLL.coxph, 30
- PLL.grpsurv (fit.grpsurv), 12
- PLL.mboost\_coxph (fit.mboost.coxph), 13
- PLL.ncvsurv (fit.ncvsurv), 14
- PLL.penfit, 15, 30, 37
- PLL.SGL\_cox (fit.SGL.cox), 17
- plot.peperr, 31
- pmpec, 32
- predict.rfsrc, 16
- predictProb, 11, 14, 34, 37
- predictProb.CoxBoost, 35
- predictProb.coxnet (fit.glmnet), 11
- predictProb.coxph, 36
- predictProb.grpsurv (fit.grpsurv), 12
- predictProb.mboost\_coxph  
    (fit.mboost.coxph), 13
- predictProb.ncvsurv (fit.ncvsurv), 14
- predictProb.penfit, 15, 31, 36
- predictProb.rfsrc (fit.rfsrc), 16
- predictProb.SGL\_cox (fit.SGL.cox), 17
- predictProb.survfit, 37
- profL1, 5, 6
- resample.indices, 25, 38

rfsrc, [16](#)

SGL, [17](#)