Package 'CoxBoost'

July 2, 2014

Version 1.4

Title Cox models by likelihood based boosting for a single survival endpoint or competing risks

Author Harald Binder <binderh@uni-mainz.de>

Maintainer Harald Binder <binderh@uni-mainz.de>

Depends R (>= 2.14.0), survival, Matrix, prodlim

Suggests parallel, snowfall

Description This package provides routines for fitting Cox models by
likelihood based boosting for a single endpoint or in presence of competing risks

License GPL (>= 2)

NeedsCompilation yes

Repository CRAN

Date/Publication 2013-05-10 10:18:16

R topics documented:

coef.CoxBoost *Coeffients from CoxBoost fit*

Description

Extracts the coefficients from the specified boosting steps of a CoxBoost object fitted by `CoxBoost`.

Usage

```
## S3 method for class 'CoxBoost'
coef(object,at.step=NULL,scaled=TRUE,...)
```

Arguments

`object`	fitted CoxBoost object from a `CoxBoost` call.
`at.step`	scalar or vector of boosting step(s) at which prediction is wanted. If no step is given, the final boosting step is used.
`scaled`	logical value indicating whether coefficients should be returned scaled to be at the level of the original covariate values, or unscaled as used internally when `standardize=TRUE` is used in the `CoxBoost` call.
`...`	miscellaneous arguments, none of which is used at the moment.

Value

For a vector of length `p` (number of covariates) (`at.step` being a scalar) or a `length(at.step) * p` matrix (`at.step` being a vector).

Author(s)

Harald Binder <binderh@uni-mainz.de>

CoxBoost *Fit a Cox model by likelihood based boosting*

Description

`CoxBoost` is used to fit a Cox proportional hazards model by componentwise likelihood based boosting. It is especially suited for models with a large number of predictors and allows for mandatory covariates with unpenalized parameter estimates.

Usage

```
CoxBoost(time,status,x,unpen.index=NULL,standardize=TRUE,subset=1:length(time),
         weights=NULL,stepno=100,penalty=9*sum(status[subset]==1),
         criterion = c("pscore", "score","hpscore","hscore"),
         stepsize.factor=1,sf.scheme=c("sigmoid","linear"),pendistmat=NULL,
         connected.index=NULL,x.is.01=FALSE,return.score=TRUE,trace=FALSE)
```

Arguments

`time`	vector of length `n` specifying the observed times.
`status`	censoring indicator, i.e., vector of length `n` with entries `0` for censored observations and `1` for uncensored observations. If this vector contains elements not equal to `0` or `1`, these are taken to indicate events from a competing risk and a model for the subdistribution hazard with respect to event `1` is fitted (see e.g. Fine and Gray, 1999; Binder et al. 2009a).
`x`	`n * p` matrix of covariates.
`unpen.index`	vector of length `p.unpen` with indices of mandatory covariates, where parameter estimation should be performed unpenalized.
`standardize`	logical value indicating whether covariates should be standardized for estimation. This does not apply for mandatory covariates, i.e., these are not standardized.
`subset`	a vector specifying a subset of observations to be used in the fitting process.
`weights`	optional vector of length `n`, for specifying weights for the individual observations.
`penalty`	penalty value for the update of an individual element of the parameter vector in each boosting step.
`criterion`	indicates the criterion to be used for selection in each boosting step. `"pscore"` corresponds to the penalized score statistics, `"score"` to the un-penalized score statistics. Different results will only be seen for un-standardized covariates (`"pscore"` will result in preferential selection of covariates with larger covariance), or if different penalties are used for different covariates. `"hpscore"` and `"hscore"` correspond to `"pscore"` and `"score"`. However, a heuristic is used for evaluating only a subset of covariates in each boosting step, as described in Binder et al. (2011). This can considerably speed up computation, but may lead to different results.
`stepsize.factor`	determines the step-size modification factor by which the natural step size of boosting steps should be changed after a covariate has been selected in a boosting step. The default (value `1`) implies constant penalties, for a value < 1 the penalty for a covariate is increased after it has been selected in a boosting step, and for a value > 1 the penalty it is decreased. If `pendistmat` is given, penalty updates are only performed for covariates that have at least one connection to another covariate.
`sf.scheme`	scheme for changing step sizes (via `stepsize.factor`). `"linear"` corresponds to the scheme described in Binder and Schumacher (2009b), `"sigmoid"` employs a sigmoid shape.
`pendistmat`	connection matrix with entries ranging between 0 and 1, with entry `(i,j)` indicating the certainty of the connection between covariates `i` and `j`. According to this information penalty changes due to `stepsize.factor` < 1 are propagated, i.e., if entry `(i,j)` is non-zero, the penalty for covariate `j` is decreased after it has been increased for covariate `i`, after it has been selected in a boosting step. This matrix either has to have dimension `(p - p.unpen) * (p - p.unpen)` or the indicices of the `p.connected` connected covariates have to be given in

	`connected.index`, in which case the matrix has to have dimension `p.connected * p.connected`. Generally, sparse matices from package `Matrix` can be used to save memory.
`connected.index`	indices of the `p.connected` connected covariates, for which `pendistmat` provides the connection information for distributing changes in penalties. No overlap with `unpen.index` is allowed. If `NULL`, and a connection matrix is given, all covariates are assumed to be connected.
`stepno`	number of boosting steps (`m`).
`x.is.01`	logical value indicating whether (the non-mandatory part of) `x` contains just values 0 and 1, i.e., binary covariates. If this is the case and indicated by this argument, computations are much faster.
`return.score`	logical value indicating whether the value of the score statistic (or penalized score statistic, depending on `criterion`), as evaluated in each boosting step for every covariate, should be returned. The corresponding element `scoremat` can become very large (and needs much memory) when the number of covariates and boosting steps is large.
`trace`	logical value indicating whether progress in estimation should be indicated by printing the name of the covariate updated.

Details

In contrast to gradient boosting (implemented e.g. in the `glmboost` routine in the R package `mboost`, using the `CoxPH` loss function), `CoxBoost` is not based on gradients of loss functions, but adapts the offset-based boosting approach from Tutz and Binder (2007) for estimating Cox proportional hazards models. In each boosting step the previous boosting steps are incorporated as an offset in penalized partial likelihood estimation, which is employed for obtain an update for one single parameter, i.e., one covariate, in every boosting step. This results in sparse fits similar to Lasso-like approaches, with many estimated coefficients being zero. The main model complexity parameter, which has to be selected (e.g. by cross-validation using `cv.CoxBoost`), is the number of boosting steps `stepno`. The penalty parameter `penalty` can be chosen rather coarsely, either by hand or using `optimCoxBoostPenalty`.

The advantage of the offset-based approach compared to gradient boosting is that the penalty structure is very flexible. In the present implementation this is used for allowing for unpenalized mandatory covariates, which receive a very fast coefficient build-up in the course of the boosting steps, while the other (optional) covariates are subjected to penalization. For example in a microarray setting, the (many) microarray features would be taken to be optional covariates, and the (few) potential clinical covariates would be taken to be mandatory, by including their indices in `unpen.index`.

If a group of correlated covariates has influence on the response, e.g. genes from the same pathway, componentwise boosting will often result in a non-zero estimate for only one member of this group. To avoid this, information on the connection between covariates can be provided in `pendistmat`. If then, in addition, a penalty updating scheme with `stepsize.factor` < 1 is chosen, connected covariates are more likely to be chosen in future boosting steps, if a directly connected covariate has been chosen in an earlier boosting step (see Binder and Schumacher, 2009b).

Value

`CoxBoost` returns an object of class `CoxBoost`.

`n, p`	number of observations and number of covariates.
`stepno`	number of boosting steps.
`xnames`	vector of length `p` containing the names of the covariates. This information is extracted from `x` or names following the scheme `V1, V2, ...`

are used.

`coefficients`	`(stepno+1) * p` matrix containing the coefficient estimates for the (standardized) optional covariates for boosting steps `0` to `stepno`. This will typically be a sparse matrix, built using package `Matrix`

.

`scoremat`	`stepno * p` matrix containing the value of the score statistic for each of the optional covariates before each boosting step.
`meanx, sdx`	vector of mean values and standard deviations used for standardizing the covariates.
`unpen.index`	indices of the mandatory covariates in the original covariate matrix `x`.
`penalty`	If `stepsize.factor != 1`, `stepno * (p - p.unpen)` matrix containing the penalties used for every boosting step and every penalized covariate, otherwise a vector containing the unchanged values of the penalty employed in each boosting step.
`time`	observed times given in the `CoxBoost` call.
`status`	censoring indicator given in the `CoxBoost` call.
`event.times`	vector with event times from the data given in the `CoxBoost` call.
`linear.predictors`	`(stepno+1) * n` matrix giving the linear predictor for boosting steps `0` to `stepno` and every observation.
`Lambda`	matrix with the Breslow estimate for the cumulative baseline hazard for boosting steps `0` to `stepno` for every event time.
`logplik`	partial log-likelihood of the fitted model in the final boosting step.

Author(s)

Written by Harald Binder <`binderh@uni-mainz.de`>.

References

Binder, H., Benner, A., Bullinger, L., and Schumacher, M. (2013). Tailoring sparse multivariable regression techniques for prognostic single-nucleotide polymorphism signatures. Statistics in Medicine, doi: 10.1002/sim.5490.

Binder, H., Allignol, A., Schumacher, M., and Beyersmann, J. (2009). Boosting for high-dimensional time-to-event data with competing risks. Bioinformatics, 25:890-896.

Binder, H. and Schumacher, M. (2009). Incorporating pathway information into boosting estimation of high-dimensional risk prediction models. BMC Bioinformatics. 10:18.

Binder, H. and Schumacher, M. (2008). Allowing for mandatory covariates in boosting estimation of sparse high-dimensional survival models. BMC Bioinformatics. 9:14.

Tutz, G. and Binder, H. (2007) Boosting ridge regression. Computational Statistics \& Data Analysis, 51(12):6044-6059.

Fine, J. P. and Gray, R. J. (1999). A proportional hazards model for the subdistribution of a competing risk. Journal of the American Statistical Association. 94:496-509.

See Also

predict.CoxBoost, cv.CoxBoost.

Examples

```
#   Generate some 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)
obs.time <- ifelse(real.time <= cens.time,real.time,cens.time)

#   Fit a Cox proportional hazards model by CoxBoost

cbfit <- CoxBoost(time=obs.time,status=status,x=x,stepno=100,penalty=100)
summary(cbfit)

#   ... with covariates 1 and 2 being mandatory

cbfit.mand <- CoxBoost(time=obs.time,status=status,x=x,unpen.index=c(1,2),
                       stepno=100,penalty=100)
summary(cbfit.mand)
```

cv.CoxBoost *Determines the optimal number of boosting steps by cross-validation*

Description

Performs a K-fold cross-validation for CoxBoost in search for the optimal number of boosting steps.

Usage

```
cv.CoxBoost(time,status,x,subset=1:length(time),maxstepno=100,K=10,
type=c("verweij","naive"),
             parallel=FALSE,upload.x=TRUE,multicore=FALSE,
             folds=NULL,trace=FALSE,...)
```

Arguments

`time`	vector of length `n` specifying the observed times.
`status`	censoring indicator, i.e., vector of length `n` with entries `0` for censored observations and `1` for uncensored observations. If this vector contains elements not equal to `0` or `1`, these are taken to indicate events from a competing risk and a model for the subdistribution hazard with respect to event `1` is fitted (see e.g. Fine and Gray, 1999).
`x`	`n * p` matrix of covariates.
`subset`	a vector specifying a subset of observations to be used in the fitting process.
`maxstepno`	maximum number of boosting steps to evaluate, i.e, the returned “optimal” number of boosting steps will be in the range `[0,maxstepno]`.
`K`	number of folds to be used for cross-validation. If `K` is larger or equal to the number of non-zero elements in `status`, leave-one-out cross-validation is performed.
`type`	way of calculating the partial likelihood contribution of the observation in the hold-out folds: `"verweij"` uses the more appropriate method described in Verweij and van Houwelingen (1996), `"naive"` uses the approach where the observations that are not in the hold-out folds are ignored (often found in other R packages).
`parallel`	logical value indicating whether computations in the cross-validation folds should be performed in parallel on a compute cluster, using package `snowfall`. Parallelization is performed via the package `snowfall` and the initialization function of of this package, `sfInit`, should be called before calling `cv.CoxBoost`.
`multicore`	indicates whether computations in the cross-validation folds should be performed in parallel, using package `parallel`. If `TRUE`, package `parallel` is employed using the default number of cores. A value larger than `1` is taken to be the number of cores that should be employed.
`upload.x`	logical value indicating whether `x` should/has to be uploaded to the compute cluster for parallel computation. Uploading this only once (using `sfExport(x)` from library `snowfall`) can save much time for large data sets.
`folds`	if not `NULL`, this has to be a list of length `K`, each element being a vector of indices of fold elements. Useful for employing the same folds for repeated runs.
`trace`	logical value indicating whether progress in estimation should be indicated by printing the number of the cross-validation fold and the index of the covariate updated.
`...`	miscellaneous parameters for the calls to `CoxBoost`

Value

List with the following components:

`mean.logplik`	vector of length `maxstepno+1` with the mean partial log-likelihood for boosting steps `0` to `maxstepno`
`se.logplik`	vector with standard error estimates for the mean partial log-likelihood criterion for each boosting step.

`optimal.step`	optimal boosting step number, i.e., with minimum mean partial log-likelihood.
`folds`	list of length `K`, where the elements are vectors of the indices of observations in the respective folds.

Author(s)

Harald Binder <`binderh@uni-mainz.de`>

References

Verweij, P. J. M. and van Houwelingen, H. C. (1993). Cross-validation in survival analysis. Statistics in Medicine, 12(24):2305-2314.

See Also

`CoxBoost`, `optimCoxBoostPenalty`

Examples

```
## Not run:
#   Generate some 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)
obs.time <- ifelse(real.time <= cens.time,real.time,cens.time)


#  10-fold cross-validation

cv.res <- cv.CoxBoost(time=obs.time,status=status,x=x,maxstepno=500,
                      K=10,type="verweij",penalty=100)

#   examine mean partial log-likelihood in the course of the boosting steps
plot(cv.res$mean.logplik)

#   Fit with optimal number of boosting steps

cbfit <- CoxBoost(time=obs.time,status=status,x=x,stepno=cv.res$optimal.step,
                  penalty=100)
summary(cbfit)


## End(Not run)
```

`cvcb.control` *Control paramters for cross-validation in* `iCoxBoost`

Description

This function allows to set the control parameters for cross-validation to be passed into a call to `iCoxBoost`.

Usage

```
cvcb.control(K=10,type=c("verweij","naive"),parallel=FALSE,
 upload.x=TRUE,multicore=FALSE,folds=NULL)
```

Arguments

`K`	number of folds to be used for cross-validation. If `K` is larger or equal to the number of events in the data to be analyzed, leave-one-out cross-validation is performed.
`type`	way of calculating the partial likelihood contribution of the observation in the hold-out folds: `"verweij"` uses the more appropriate method described in Verweij and van Houwelingen (1996), `"naive"` uses the approach where the observations that are not in the hold-out folds are ignored (often found in other R packages).
`parallel`	logical value indicating whether computations in the cross-validation folds should be performed in parallel on a compute cluster, using package `snowfall`. Parallelization is performed via the package `snowfall` and the initialization function of of this package, `sfInit`, should be called before calling `iCoxBoost`.
`multicore`	indicates whether computations in the cross-validation folds should be performed in parallel, using package `parallel`. If `TRUE`, package `parallel` is employed using the default number of cores. A value larger than `1` is taken to be the number of cores that should be employed.
`upload.x`	logical value indicating whether `x` should/has to be uploaded to the compute cluster for parallel computation. Uploading this only once (using `sfExport(x)` from library `snowfall`) can save much time for large data sets.
`folds`	if not `NULL`, this has to be a list of length `K`, each element being a vector of indices of fold elements. Useful for employing the same folds for repeated runs.

Value

List with elements corresponding to the call arguments.

Author(s)

Written by Harald Binder <`binderh@uni-mainz.de`>.

References

Verweij, P. J. M. and van Houwelingen, H. C. (1993). Cross-validation in survival analysis. Statistics in Medicine, 12(24):2305-2314.

See Also

`iCoxBoost`, `cv.CoxBoost`

`estimPVal` *Estimate p-values for a model fitted by CoxBoost*

Description

Performs permutation-based p-value estimation for the optional covariates in a fit from `CoxBoost`.

Usage

```
estimPVal(object,x,permute.n=10,per.covariate=FALSE,parallel=FALSE,
          multicore=FALSE,trace=FALSE,...)
```

Arguments

`object`	fit object obtained from `CoxBoost`.
`x`	`n * p` matrix of covariates. This has to be the same that was used in the call to `CoxBoost`.
`permute.n`	number of permutations employed for obtaining a null distribution.
`per.covariate`	logical value indicating whether a separate null distribution should be considered for each covariate. A larger number of permutations will be needed if this is wanted.
`parallel`	logical value indicating whether computations for obtaining a null distribution via permutation should be performed in parallel on a compute cluster. Parallelization is performed via the package `snowfall` and the initialization function of of this package, `sfInit`, should be called before calling `estimPVal`.
`multicore`	indicates whether computations in the permuted data sets should be performed in parallel, using package `parallel`. If `TRUE`, package `parallel` is employed using the default number of cores. A value larger than `1` is taken to be the number of cores that should be employed.
`trace`	logical value indicating whether progress in estimation should be indicated by printing the number of the permutation that is currently being evaluated.
`...`	miscellaneous parameters for the calls to `CoxBoost`

Details

As p-value estimates are based on permutations, random numbers are drawn for determining permutation indices. Therfore, the results depend on the state of the random number generator. This can be used to explore the variability due to random variation and help to determine an adequate value for `permute.n`. A value of 100 should be sufficient, but this can be quite slow. If there is a considerable number of covariates, e.g., larger than 100, a much smaller number of permutations, e.g., 10, might already work well. The estimates might also be negatively affected, if only a small number of boosting steps (say <50) was employed for the original fit.

Value

Vector with p-value estimates, one value for each optional covariate specificed in the original call to `CoxBoost`.

Author(s)

Harald Binder `<binderh@uni-mainz.de>`

References

Binder, H., Porzelius, C. and Schumacher, M. (2009). Rank-based p-values for sparse high-dimensional risk prediction models fitted by componentwise boosting. FDM-Preprint Nr. 101, University of Freiburg, Germany.

See Also

`CoxBoost`

Examples

```
## Not run:
#   Generate some 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)
obs.time <- ifelse(real.time <= cens.time,real.time,cens.time)

#   Fit a Cox proportional hazards model by CoxBoost

cbfit <- CoxBoost(time=obs.time,status=status,x=x,stepno=100,
                  penalty=100)

#   estimate p-values

p1 <- estimPVal(cbfit,x,permute.n=10)

#   get a second vector of estimates for checking how large
#   random variation is
```

```
p2 <- estimPVal(cbfit,x,permute.n=10)

plot(p1,p2,xlim=c(0,1),ylim=c(0,1),xlab="permute 1",ylab="permute 2")

## End(Not run)
```

---

iCoxBoost *Interface for cross-validation and model fitting using a formula description*

Description

Formula interface for fitting a Cox proportional hazards model by componentwise likelihood based boosting (via a call to `CoxBoost`), where cross-validation can be performed automatically for determining the number of boosting steps (via a call to `cv.CoxBoost`).

Usage

```
iCoxBoost(formula,data=NULL,weights=NULL,subset=NULL,mandatory=NULL,
  cause=1,standardize=TRUE,stepno=200,
  criterion=c("pscore","score","hpscore","hscore"),
  nu=0.1,stepsize.factor=1,varlink=NULL,
  cv=cvcb.control(),trace=FALSE,...)
```

Arguments

`formula`	A formula describing the model to be fitted, similar to a call to `coxph`. The response must be a survival object, either as returned by `Surv` or `Hist` (in a competing risks application).
`data`	data frame containing the variables described in the formula.
`weights`	optional vector, for specifying weights for the individual observations.
`subset`	a vector specifying a subset of observations to be used in the fitting process.
`mandatory`	vector containing the names of the covariates whose effect is to be estimated un-regularized.
`cause`	cause of interest in a competing risks setting, when the response is specified by `Hist` (see e.g. Fine and Gray, 1999; Binder et al. 2009a).
`standardize`	logical value indicating whether covariates should be standardized for estimation. This does not apply for mandatory covariates, i.e., these are not standardized.
`stepno`	maximum number of boosting steps to be evaluated when determining the number of boosting steps by cross-validation, otherwise the number of boosting seps itself.

`criterion`	indicates the criterion to be used for selection in each boosting step. `"pscore"` corresponds to the penalized score statistics, `"score"` to the un-penalized score statistics. Different results will only be seen for un-standardized covariates (`"pscore"` will result in preferential selection of covariates with larger covariance), or if different penalties are used for different covariates. `"hpscore"` and `"hscore"` correspond to `"pscore"` and `"score"`. However, a heuristic is used for evaluating only a subset of covariates in each boosting step, as described in Binder et al. (2011). This can considerably speed up computation, but may lead to different results.
`nu`	(roughly) the fraction of the partial maximum likelihood estimate used for the update in each boosting step. This is converted into a penalty for the call to `CoxBoost`. Use smaller values, e.g., 0.01 when there is little information in the data, and larger values, such as 0.1, with much information or when the number of events is larger than the number of covariates. Note that the default for direct calls to `CoxBoost` corresponds to `nu=0.1`.
`stepsize.factor`	determines the step-size modification factor by which the natural step size of boosting steps should be changed after a covariate has been selected in a boosting step. The default (value `1`) implies constant `nu`, for a value < 1 the value `nu` for a covariate is decreased after it has been selected in a boosting step, and for a value > 1 the value `nu` is increased. If `pendistmat` is given, updates of `nu` are only performed for covariates that have at least one connection to another covariate.
`varlink`	list for specifying links between covariates, used to re-distribute step sizes when `stepsize.factor != 1`. The list needs to contain at least two vectors, the first containing the name of the source covariates, the second containing the names of the corresponding target covariates, and a third (optional) vector containing weights between 0 and 1 (defaulting to 1). If `nu` is increased/descreased for one of the source covariates according to `stepsize.factor`, the `nu` for the corresponding target covariate is descreased/increased accordingly (multiplied by the weight). If `formula` contains interaction terms, als rules for these can be set up, using variable names such as `V1:V2` for the interaction term between covariates `V1` and `V2`.
`cv`	`TRUE`, for performing cross-validation, with default parameters, `FALSE` for not performing cross-validation, or list containing the parameters for cross-validation, as obtained from a call to `cvcb.control`.
`trace`	logical value indicating whether progress in estimation should be indicated by printing the name of the covariate updated.
`...`	miscellaneous arguments, passed to the call to `cv.CoxBoost`.

Details

In contrast to gradient boosting (implemented e.g. in the `glmboost` routine in the R package `mboost`, using the `CoxPH` loss function), `CoxBoost` is not based on gradients of loss functions, but adapts the offset-based boosting approach from Tutz and Binder (2007) for estimating Cox proportional hazards models. In each boosting step the previous boosting steps are incorporated as an offset in penalized partial likelihood estimation, which is employed for obtain an update for one single parameter, i.e., one covariate, in every boosting step. This results in sparse fits similar to Lasso-like approaches, with many estimated coefficients being zero. The main model complexity parameter, the number of boosting steps, is automatically selected by cross-validation using a call to `cv.CoxBoost`). Note that this will introduce random variation when repeatedly calling `iCoxBoost`, i.e. it is advised to set/save the random number generator state for reproducible results.

The advantage of the offset-based approach compared to gradient boosting is that the penalty structure is very flexible. In the present implementation this is used for allowing for unpenalized mandatory covariates, which receive a very fast coefficient build-up in the course of the boosting steps, while the other (optional) covariates are subjected to penalization. For example in a microarray setting, the (many) microarray features would be taken to be optional covariates, and the (few) potential clinical covariates would be taken to be mandatory, by including their names in `mandatory`.

If a group of correlated covariates has influence on the response, e.g. genes from the same pathway, componentwise boosting will often result in a non-zero estimate for only one member of this group. To avoid this, information on the connection between covariates can be provided in `varlink`. If then, in addition, a penalty updating scheme with `stepsize.factor < 1` is chosen, connected covariates are more likely to be chosen in future boosting steps, if a directly connected covariate has been chosen in an earlier boosting step (see Binder and Schumacher, 2009b).

Value

`iCoxBoost` returns an object of class `iCoxBoost`, which also has class `CoxBoost`. In addition to the elements from `CoxBoost` it has the following elements:

`call, formula, terms`	call, formula and terms from the formula interface.
`cause`	cause of interest.
`cv.res`	result from `cv.CoxBoost`, if cross-validation has been performed.

Author(s)

Written by Harald Binder <`binderh@uni-mainz.de`>.

References

Binder, H., Benner, A., Bullinger, L., and Schumacher, M. (2013). Tailoring sparse multivariable regression techniques for prognostic single-nucleotide polymorphism signatures. Statistics in Medicine, doi: 10.1002/sim.5490.

Binder, H., Allignol, A., Schumacher, M., and Beyersmann, J. (2009). Boosting for high-dimensional time-to-event data with competing risks. Bioinformatics, 25:890-896.

Binder, H. and Schumacher, M. (2009). Incorporating pathway information into boosting estimation of high-dimensional risk prediction models. BMC Bioinformatics. 10:18.

Binder, H. and Schumacher, M. (2008). Allowing for mandatory covariates in boosting estimation of sparse high-dimensional survival models. BMC Bioinformatics. 9:14.

Tutz, G. and Binder, H. (2007) Boosting ridge regression. Computational Statistics \& Data Analysis, 51(12):6044-6059.

Fine, J. P. and Gray, R. J. (1999). A proportional hazards model for the subdistribution of a competing risk. Journal of the American Statistical Association. 94:496-509.

See Also

predict.iCoxBoost, CoxBoost, cv.CoxBoost.

Examples

```
#   Generate some survival data with 10 informative covariates
n <- 200; p <- 100
beta <- c(rep(1,2),rep(0,p-2))
x <- matrix(rnorm(n*p),n,p)
actual.data <- as.data.frame(x)
real.time <- -(log(runif(n)))/(10*exp(drop(x %*% beta)))
cens.time <- rexp(n,rate=1/10)
actual.data$status <- ifelse(real.time <= cens.time,1,0)
actual.data$time <- ifelse(real.time <= cens.time,real.time,cens.time)

#   Fit a Cox proportional hazards model by iCoxBoost

cbfit <- iCoxBoost(Surv(time,status) ~ .,data=actual.data)
summary(cbfit)
plot(cbfit)

#   ... with covariates 1 and 2 being mandatory

cbfit.mand <- iCoxBoost(Surv(time,status) ~ .,data=actual.data,mandatory=c("V1"))
summary(cbfit.mand)
plot(cbfit.mand)
```

---

optimCoxBoostPenalty *Coarse line search for adequate penalty parameter*

Description

This routine helps in finding a penalty value that leads to an “optimal” number of boosting steps for CoxBoost, determined by cross-validation, that is not too small/in a specified range.

Usage

```
optimCoxBoostPenalty(time,status,x,minstepno=50,maxstepno=200,
                       start.penalty=9*sum(status==1),iter.max=10,
                       upper.margin=0.05,parallel=FALSE,
                       trace=FALSE,...)
```

Arguments

time	vector of length n specifying the observed times.

`status`	censoring indicator, i.e., vector of length `n` with entries `0` for censored observations and `1` for uncensored observations. If this vector contains elements not equal to `0` or `1`, these are taken to indicate events from a competing risk and a model for the subdistribution hazard with respect to event `1` is fitted (see e.g. Fine and Gray, 1999).
`x`	`n * p` matrix of covariates.
`minstepno, maxstepno`	range of boosting steps in which the "optimal" number of boosting steps is wanted to be.
`start.penalty`	start value for the search for the appropriate penalty.
`iter.max`	maximum number of search iterations.
`upper.margin`	specifies the fraction of `maxstepno` which is used as an upper margin in which a cross-validation minimum is not taken to be one. This is necessary because of random fluctuations of cross-validated partial log-likelihood.
`parallel`	logical value indicating whether computations in the cross-validation folds should be performed in parallel on a compute cluster. Parallelization is performed via the package `snowfall` and the initialization function of of this package, `sfInit`, should be called before calling `cv.CoxBoost`.
`trace`	logical value indicating whether information on progress should be printed.
`...`	miscellaneous parameters for `cv.CoxBoost`.

Details

The penalty parameter for `CoxBoost` has to be chosen only very coarsely. In Tutz and Binder (2006) it is suggested for likelihood based boosting just to make sure, that the optimal number of boosting steps, according to some criterion such as cross-validation, is larger or equal to 50. With a smaller number of steps, boosting may become too "greedy" and show sub-optimal performance. This procedure uses a very coarse line search and so one should specify a rather large range of boosting steps.

Value

List with element `penalty` containing the "optimal" penalty and `cv.res` containing the corresponding result of `cv.CoxBoost`.

Author(s)

Written by Harald Binder <`binderh@uni-mainz.de`>.

References

Tutz, G. and Binder, H. (2006) Generalized additive modelling with implicit variable selection by likelihood based boosting. *Biometrics*, 62:961-971.

See Also

`CoxBoost`, `cv.CoxBoost`

Examples

```
## Not run:
#   Generate some 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)
obs.time <- ifelse(real.time <= cens.time,real.time,cens.time)

#  determine penalty parameter

optim.res <- optimCoxBoostPenalty(time=obs.time,status=status,x=x,
                                  trace=TRUE,start.penalty=500)

#   Fit with obtained penalty parameter and optimal number of boosting
#   steps obtained by cross-validation

cbfit <- CoxBoost(time=obs.time,status=status,x=x,
                  stepno=optim.res$cv.res$optimal.step,
                  penalty=optim.res$penalty)
summary(cbfit)


## End(Not run)
```

`optimStepSizeFactor` *Coarse line search for optimum step-size modification factor*

Description

This routine helps in finding an optimum step-size modification factor for `CoxBoost`, i.e., that results in an optimum in terms of cross-validated partial log-likelihood.

Usage

```
optimStepSizeFactor(time,status,x,
                      direction=c("down","up","both"),start.stepsize=0.1,
                      iter.max=10,constant.cv.res=NULL,
                      parallel=FALSE,trace=FALSE,...)
```

Arguments

`time`	vector of length `n` specifying the observed times.

`status`	censoring indicator, i.e., vector of length `n` with entries `0` for censored observations and `1` for uncensored observations. If this vector contains elements not equal to `0` or `1`, these are taken to indicate events from a competing risk and a model for the subdistribution hazard with respect to event `1` is fitted (see e.g. Fine and Gray, 1999).
`x`	`n * p` matrix of covariates.
`direction`	direction of line search for an optimal step-size modification factor (starting from value 1).
`start.stepsize`	step size used for the line search. A final step is performed using half this size.
`iter.max`	maximum number of search iterations.
`constant.cv.res`	result of `cv.CoxBoost` for `stepsize.factor=1`, that can be provided for saving computing time, if it already is available.
`parallel`	logical value indicating whether computations in the cross-validation folds should be performed in parallel on a compute cluster. Parallelization is performed via the package `snowfall` and the initialization function of of this package, `sfInit`, should be called before calling `cv.CoxBoost`.
`trace`	logical value indicating whether information on progress should be printed.
`...`	miscellaneous parameters for `cv.CoxBoost`.

Details

A coarse line search is performed for finding the best parameter `stepsize.factor` for `CoxBoost`. If an `pendistmat` argument is provided (which is passed on to `CoxBoost`), a search for factors smaller than 1 is sensible (corresponding to `direction="down"`). If no connection information is provided, it is reasonable to employ `direction="both"`, for avoiding restrictions without subject matter knowledge.

Value

List with the following components:

`factor.list`	array with the evaluated step-size modification factors.
`critmat`	matrix with the mean partial log-likelihood for each step-size modification factor in the course of the boosting steps.
`optimal.factor.index`	index of the optimal step-size modification factor.
`optimal.factor`	optimal step-size modification factor.
`optimal.step`	optimal boosting step number, i.e., with minimum mean partial log-likelihood, for step-size modification factor `optimal.factor`.

Author(s)

Written by Harald Binder <`binderh@uni-mainz.de`>.

References

Binder, H. and Schumacher, M. (2009). Incorporating pathway information into boosting estimation of high-dimensional risk prediction models. BMC Bioinformatics. 10:18.

See Also

`CoxBoost`, `cv.CoxBoost`

Examples

```
## Not run:
#   Generate some 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)
obs.time <- ifelse(real.time <= cens.time,real.time,cens.time)

#  Determine step-size modification factor. As there is no connection matrix,
#  perform search into both directions

optim.res <- optimStepSizeFactor(direction="both",
                                time=obs.time,status=status,x=x,
                                trace=TRUE)

#   Fit with obtained step-size modification parameter and optimal number of boosting
#   steps obtained by cross-validation

cbfit <- CoxBoost(time=obs.time,status=status,x=x,
                  stepno=optim.res$optimal.step,
                  stepsize.factor=optim.res$optimal.factor)
summary(cbfit)


## End(Not run)
```

plot.CoxBoost   *Plot coefficient paths from CoxBoost fit*

Description

Plots coefficient paths, i.e. the parameter estimates plotted against the boosting steps as obtained from a CoxBoost object fitted by `CoxBoost`.

Usage

```
## S3 method for class 'CoxBoost'
plot(x,line.col="dark grey",label.cex=0.6,xlab=NULL,ylab=NULL,xlim=NULL,ylim=NULL,...)
```

Arguments

`x`	fitted CoxBoost object from a `CoxBoost` call.
`line.col`	color of the lines of the coefficient path
`label.cex`	scaling factor for the variable labels.
`xlab`	label for the x axis, default label when `NULL`.
`ylab`	label for the y axis, default label when `NULL`.
`xlim,ylim`	plotting ranges, default label when `NULL`.
`...`	miscellaneous arguments, passed to the plot routine.

Value

No value is returned, but a plot is generated.

Author(s)

Harald Binder <binderh@uni-mainz.de>

`predict.CoxBoost` *Predict method for CoxBoost fits*

Description

Obtains predictions at specified boosting steps from a CoxBoost object fitted by `CoxBoost`.

Usage

```
## S3 method for class 'CoxBoost'
predict(object,newdata=NULL,newtime=NULL,newstatus=NULL,
        subset=NULL,at.step=NULL,times=NULL,
        type=c("lp","logplik","risk","CIF"),...)
```

Arguments

`object`	fitted CoxBoost object from a `CoxBoost` call.
`newdata`	`n.new * p` matrix with new covariate values. If just prediction for the training data is wanted, it can be omitted.

`newtime, newstatus`	vectors with observed time and censoring indicator (0 for censoring, 1 for no censoring, and any other values for competing events in a competing risks setting) for new observations, where prediction is wanted. Only required if predicted partial log-likelihood is wanted, i.e., if `type="logplik"`. This can also be omitted when prediction is only wanted for the training data, i.e., `newdata=NULL`.
`subset`	an optional vector specifying a subset of observations to be used for evaluation.
`at.step`	scalar or vector of boosting step(s) at which prediction is wanted. If `type="risk"` is used, only one step is admissible. If no step is given, the final boosting step is used.
`times`	vector with `T` time points where prediction is wanted. Only needed for `type="risk"`
`type`	type of prediction to be returned: `"lp"` gives the linear predictor, `"logplik"` the partial log-likelihood, `"risk"` the predicted probability of not yet having had the event at the time points given in `times`, and `"CIF"` the predicted cumulative incidence function, i.e., the predicted probability of having had the event of interest.
`...`	miscellaneous arguments, none of which is used at the moment.

Value

For `type="lp"` and `type="logplik"` a vector of length `n.new` (`at.step` being a scalar) or a `n.new * length(at.step)` matrix (`at.step` being a vector) with predictions is returned. For `type="risk"` or `type="CIF"` a `n.new * T` matrix with predicted probabilities at the specific time points is returned.

Author(s)

Harald Binder <`binderh@uni-mainz.de`>

Examples

```
#   Generate some 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)
obs.time <- ifelse(real.time <= cens.time,real.time,cens.time)

#   define training and test set

train.index <- 1:100
test.index <- 101:200

#   Fit CoxBoost to the training data

cbfit <- CoxBoost(time=obs.time[train.index],status=status[train.index],
                  x=x[train.index,],stepno=300,penalty=100)
```

```
#   mean partial log-likelihood for test set in every boosting step

step.logplik <- predict(cbfit,newdata=x[test.index,],
                        newtime=obs.time[test.index],
                        newstatus=status[test.index],
                        at.step=0:300,type="logplik")

plot(step.logplik)

#   names of covariates with non-zero coefficients at boosting step
#   with maximal test set partial log-likelihood

print(cbfit$xnames[cbfit$coefficients[which.max(step.logplik),] != 0])
```

---

predict.iCoxBoost *Predict method for iCoxBoost fits*

Description

Obtains predictions at specified boosting steps from a iCoxBoost object fitted by `iCoxBoost`.

Usage

```
## S3 method for class 'iCoxBoost'
predict(object,newdata=NULL,
        subset=NULL,at.step=NULL,times=NULL,
        type=c("lp","logplik","risk","CIF"),...)
```

Arguments

`object`	fitted CoxBoost object from a `CoxBoost` call.
`newdata`	data frame with new covariate values (for `n.new` observations). If just prediction for the training data is wanted, it can be omitted. If the predictive partial log-likelihood is wanted (`type=logplik`), this frame also has to contain the response information.
`subset`	an optional vector specifying a subset of observations to be used for evaluation.
`at.step`	scalar or vector of boosting step(s) at which prediction is wanted. If `type="risk"` is used, only one step is admissible. If no step is given, the final boosting step is used.
`times`	vector with `T` time points where prediction is wanted. Only needed for `type="risk"`
`type`	type of prediction to be returned: `"lp"` gives the linear predictor, `"logplik"` the partial log-likelihood, `"risk"` the predicted probability of not yet having had the event at the time points given in `times`, and `"CIF"` the predicted cumulative incidence function, i.e., the predicted probability of having had the event of interest.
`...`	miscellaneous arguments, none of which is used at the moment.

Value

For `type="lp"` and `type="logplik"` a vector of length `n.new` (`at.step` being a scalar) or a `n.new * length(at.step)` matrix (`at.step` being a vector) with predictions is returned. For `type="risk"` or `type="CIF"` a `n.new * T` matrix with predicted probabilities at the specific time points is returned.

Author(s)

Harald Binder <`binderh@uni-mainz.de`>

Examples

```
n <- 200; p <- 100
beta <- c(rep(1,2),rep(0,p-2))
x <- matrix(rnorm(n*p),n,p)
actual.data <- as.data.frame(x)
real.time <- -(log(runif(n)))/(10*exp(drop(x %*% beta)))
cens.time <- rexp(n,rate=1/10)
actual.data$status <- ifelse(real.time <= cens.time,1,0)
actual.data$time <- ifelse(real.time <= cens.time,real.time,cens.time)

#   define training and test set

train.index <- 1:100
test.index <- 101:200

#   Fit a Cox proportional hazards model by iCoxBoost

cbfit <- iCoxBoost(Surv(time,status) ~ .,data=actual.data[train.index,],
   stepno=300,cv=FALSE)

#   mean partial log-likelihood for test set in every boosting step

step.logplik <- predict(cbfit,newdata=actual.data[test.index,],
                        at.step=0:300,type="logplik")

plot(step.logplik)

#   names of covariates with non-zero coefficients at boosting step
#   with maximal test set partial log-likelihood

print(coef(cbfit,at.step=which.max(step.logplik)-1))
```

Index