survHD
package vignette

Christoph Bernau *
Levi Waldron †
Markus Riester‡

2012

1 SurvHD - Survival Analysis for High-Dimensional Data (Gene Expression Data)

Survival analysis on high-dimensional data is a complex and computationally intensive task. Many standard algorithms for survival data are not applicable in the "$p >> n$"-case, e.g. with several thousands of variables and only hundreds of samples. Many adaptions of the Cox-Model and other standard approaches for high-dimensional data have been developed in the last years. Furthermore, methods for tuning and evaluation of models are distinct from classification problems. Our R-package **survHD** provides a framework for the application, optimization, and evaluation of high-dimensional survival models. In this vignette, we will introduce the basic work flow of the package using a well-known cancer microarray data set. Additionally, we will explain the built-in interface for including user-defined custom survival model functions into the **survHD** framework.

1.1 Input Data

The package accepts three different formats for predictor data **X**: data-frame, matrix, or the Bioconductor-defined **ExpressionSet** class. The corresponding survival outcome can be either a **Surv** object from the package **survival** or in the case of the **ExpressionSet**-format a character vector specifying the name of this **surv** object in the **phenoData**-slot of the **ExpressionSet** object. In our example, we use the matrix format for analyzing the gene expressions of 86 lung adenocarcinoma patients published by Beer et al. (2002), which can be found in the data folder of the package (here we just use the first 500 probe sets for illustration purposes):

```
> set.seed(321)
> data(beer.exprs)
> data(beer.survival)
> X <- t(as.matrix(beer.exprs))
> y <- Surv(beer.survival$os, beer.survival$status)
```

*bernau@ibe.med.uni-muenchen.de
†lwaldron.research@gmail.com
‡markus@jimmy.harvard.edu

1.2 Fitting Models

The most important function in **survHD** is called **learnSurvival**. It is used to fit the survival model and store its essential information for later evaluation on test data. Apart from the input data mentioned in the last section, the only additional required argument is the name of survival method that shall be performed: in this example we will use **coxBoostSurv**, a wrapper to the **CoxBoost** package. As a first example of training on a full dataset followed by independent evaluation on a predefined dataset, we exclusively use the first 60 observations for model fitting because we want to evaluate our model on the last 26 observations of our data. The parameter **stepno** is a method specific tuning parameter (the number of Boosting steps) which is simply passed to the **CoxBoost** function.

```
> ytrain <- y[1:60,]
> Xtrain <- X[1:60,]
> coxboostmodel <- learnSurvival(y=ytrain,
+                                X=Xtrain,
+                                survmethod='coxBoostSurv',
+                                stepno=20)

> str(coxboostmodel, max.level=2)

Formal class 'LearnOut' [package "survHD"] with 7 slots
  ..@ ModelLearnedlist:List of 1
  ..@ X               :'data.frame':        60 obs. of  500 variables:
  .. .. [list output truncated]
  ..@ y               : Surv [1:60, 1:2]  84.1   91.8+  93.7+ 108.2+  34.6   68.1+  34.2+  47.0+  39.1+
  .. ..- attr(*, "dimnames")=List of 2
  .. ..- attr(*, "type")= chr "right"
  ..@ GeneSel         :Formal class 'GeneSel' [package "survHD"] with 4 slots
  ..@ nbgene          : int 500
  ..@ LearningSets    :Formal class 'LearningSets' [package "survHD"] with 4 slots
  ..@ threshold       : num -1
```

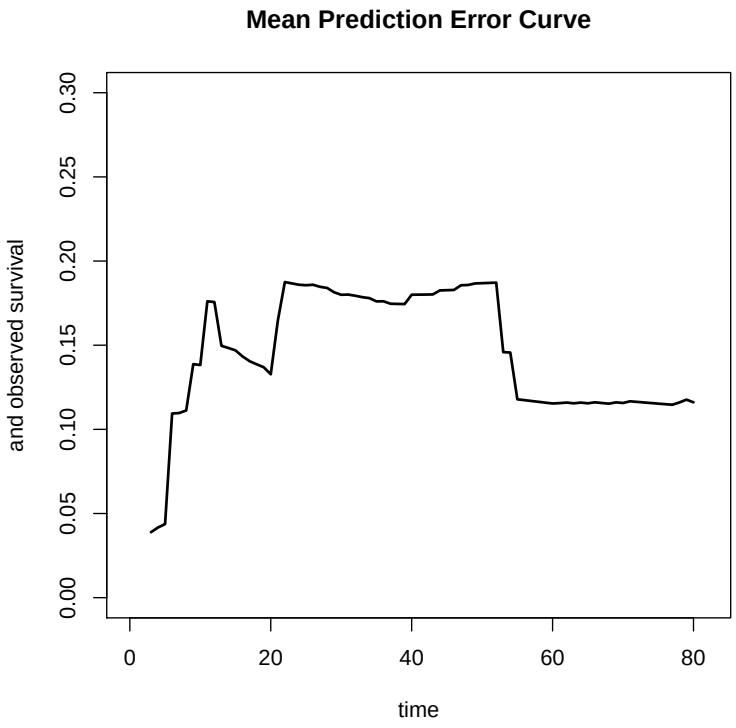
The call to **str()** shows the slot of the output object which stores all the information necessary to perform predictions for new data and evaluate the model.

1.3 Evaluation on independent Data

Evaluation of survival models is a complicated task for which no gold standard exists. One of the biggest problems is the time-dependency of the model fit which is difficult to include or weight appropriately. Many reasonable measures for evaluating survival models have been proposed in the literature. Currently, three of these measures are implemented in the function **evaluate**: Harrell's C-Index, Uno's C-Index, and Prediction Error Curves. Prediction error curves compare the model-based estimated survival curves with the actual survival indicator function of the test observations. For evaluating our model with regard to this measure ("PErrC") we have to pass the output of **learnSurvival** (coxboostmodel) and the validation data to the function **evaluate**. Additionally, we must define a grid of time points for which the comparison of estimated and observed survival shall be performed:

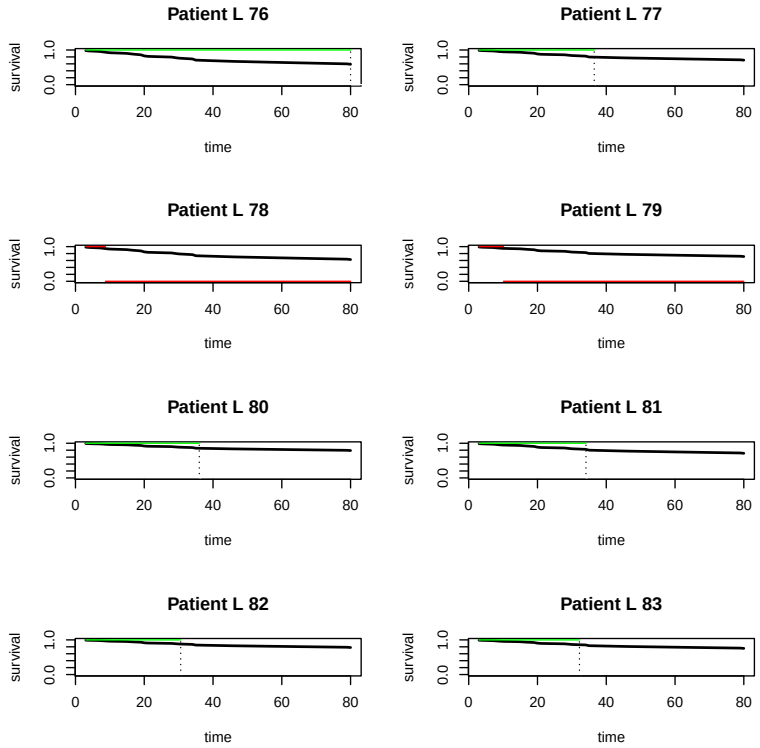
```
> ytest <- y[61:86, ]
> Xtest <- X[61:86, ]
```

```
> timegrid <- 3:80
> ev1 <- evaluate(coxboostmodel, measure='PErrC', newy=ytest, newdata=Xtest,
+               timegrid=timegrid, addtrain=F)
> plot(timegrid, ev1@result[[1]], type='l', lwd=2, col='black', xlim=c(0, 82),
+      ylim=c(0, 0.3), xlab='time', ylab='mean difference between estimated
+ and observed survival', main='Mean Prediction Error Curve')
```



We can see that the prediction performance for the validation set is worse in the time interval between 20 and 55 months. For a more detailed evaluation of the model fit we can compare the observed and estimated survival for each patient separately. For that purpose we can use the **predict** function on the **LearnOut** object in order to obtain observationwise predictions and compare them to the survival indicator function.

```
> probs <- predict(coxboostmodel, newdata=Xtest, type='SurvivalProbs', timegrid=3:80)
> par(mfrow=c(4, 2))
> for(i in 6:13){
+   followup <- xlim2 <- 80
+   col1 <- 'red'
+   col2 <- 'red'
+   if(ytest[i, 2]==0){followup <- min(80, ytest[i, 1]);col1 <- 'green';col2 <- 'white'}
+ plot(timegrid, probs[[1]]@SurvivalProbs@curves[i, ], col='black', lwd=2,
+      main=paste('Patient L', i+70, sep=' '), xlab='time', ylab='survival',
+      type='l', ylim=c(0, 1), xlim=c(3, xlim2))
+ segments(c(3, ytest[i, 1]), c(1, 0), c(min(followup, ytest[i, 1]), followup),
+          c(1, 0), col=c(col1, col2))
+   if(ytest[i, 2]==0)
+     abline(v=followup, lty=3)
+ }
```



This plot illustrates a moderate prediction accuracy on this patient selection. For the two patients with events, the survival curves are indeed lower than e.g. in the first graph representing the survival curve of a patient who was still alive after 80 months when he was censored.

Using **predict** (*type* =**'lp'**), we can also see that the linear predictors are indeed higher for the two patients with events. Please note the (maybe unexpected) structure of the output object. Since **survHD** is actually designed to fit several models at a time – e.g. in a cross-validation – the output object is a list containing a single object of class **LinearPrediction** whose slot **lp** contains the linear predictor which we wanted to obtain:

```
List of 1
 $ :Formal class 'LinearPrediction' [package "survHD"] with 1 slots
  .. ..@ lp: Named num [1:26] 0.6036 -0.0743 0.8752 -0.3733 -0.1395 ...
  .. .. ..- attr(*, "names")= chr [1:26] "L59" "L61" "L62" "L64" ...

        L76          L78          L79          L80
 0.001252846 -0.425616282 -0.126311315 -0.452779637
        L81          L82          L83          L84
-0.806717017 -0.438309036 -0.757825924 -0.632640920
```

Based on the linear predictor one can compute other evaluation metrics like Harrell's C-Statistic. In **survHD**, it can be computed by specifying **measure="HarrellC"** in the call to **evaluate**:

```
> ev2 <- evaluate(coxboostmodel, measure='HarrellC', newy=ytest,
+               newdata=Xtest)
> ev2@result
```

```
[[1]]
  C Index
0.6451613
```

```
> ev3 <- evaluate(coxboostmodel, measure='HarrellC', newy=ytrain,
+               newdata=Xtrain)
> ev3@result
```

```
[[1]]
  C Index
0.8595166
```

Similarly to the prediction error curves this measure suggests a moderate prediction performance on the independent validation set. As we can see from the second call to **evaluate**, the performance on the independent test data is clearly worse than on the training data, indicating overfitting on the training data. Consequently, evaluation should always be performed using independent test observations.

Two other standard evaluation measure is available in **survHD** - Uno's C-Statistic (**measure='UnoC'**), which models the censoring distribution in the validation set. Cross-validated partial log likelihood (**measure='CvPLogL'**) is also available, and normally used during cross-validation rather than independent validation. Cross-validation will be covered in the next section on "resampling based evaluation."

```
> ev4 <- evaluate(coxboostmodel, measure='UnoC', newy=ytest,
+               newdata=Xtest, tau=9)
> ev4@result
```

```
[[1]]
[1] 0.6826321
```

```
> ev5 <- evaluate(coxboostmodel, measure='CvPLogL', newy=ytrain,
+               newdata=Xtrain)
> ev5@result
```

```
[[1]]
[1] -73.59193
```

1.4 Resampling based evaluation

Often, there are not enough data to set aside an independent validation set, and in examples such as above, evaluation estimates using a single validation fold are sensitive to the selection of that fold. Cross-validation or the bootstrap are better alternatives where data for all samples are available.

The work flow for resampling is almost the same as for evaluation on independent data. However, one first has to create an object of class **LearningSets** using the function **generateLearningsets**, which defines the validation scheme. In our example we will use 10 fold cross-validation:

```
> ls <- generateLearningsets(y=y, method='CV', fold=10, strat=TRUE)
> str(ls,  max.level=2)
```

```
Formal class 'LearningSets' [package "survHD"] with 4 slots
  ..@ learnmatrix: num [1:10, 1:78] 2 2 2 0 3 2 2 2 2 0 ...
  ..@ method     : chr "stratified CV"
  ..@ ntrain     : int 78
  ..@ iter       : int 10
```

By setting the argument **strat** to **TRUE**, we cause **survHD** to create folds with similar proportions of censored to uncensored observations. The most important slot of the returned object (**ls**) is called **learnmatrix** which contains the indices of the training folds in it rows.

The actual model fitting is again performed by the function **learnSurvival** by passing the return-object of **generateLearningsets (ls)** as argument **LearningSets**:

```
> outcv <- learnSurvival(y=y, X=X, survmethod='coxBoostSurv', LearningSets=ls,
+                        stepno=20)
```

This call automatically fits a CoxBoost model on each training fold. Its return object is of class **LearnOut**. The fitted models can be found in the slot **ModelLearnedlist** and will be by **survHD** used lateron for model evaluation.

```
> str(outcv, max.level=2)
```

```
Formal class 'LearnOut' [package "survHD"] with 7 slots
  ..@ ModelLearnedlist:List of 10
  ..@ X               :'data.frame':        86 obs. of  500 variables:
  .. .. [list output truncated]
  ..@ y               : Surv [1:86, 1:2]  84.1   91.8+  93.7+ 108.2+  34.6   68.1+  34.2+  47.0+  39.1+
  .. ..- attr(*, "dimnames")=List of 2
  .. ..- attr(*, "type")= chr "right"
  ..@ GeneSel         :Formal class 'GeneSel' [package "survHD"] with 4 slots
  ..@ nbgene          : int 500
  ..@ LearningSets    :Formal class 'LearningSets' [package "survHD"] with 4 slots
  ..@ threshold       : num -1
```

The call to **evaluate** is exactly the same as for independent evaluation. The only difference is that the slot **result** of the output object is now a list of the foldwise evaluation statistics as here for measure **HarrellC**:

```
> evcv <- evaluate(outcv, measure='HarrellC')
> unlist(evcv@result)
```

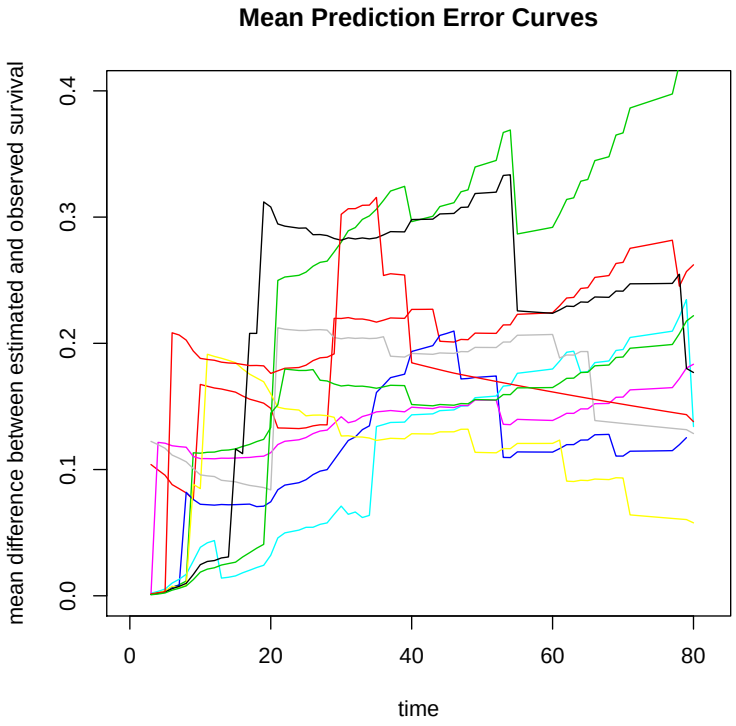
```
  C Index   C Index   C Index   C Index   C Index   C Index
0.7647059 0.5384615 0.9333333 0.5000000 0.7777778 0.9090909
  C Index   C Index   C Index   C Index
0.6363636 0.6250000 0.6111111 0.5000000
```

```
> mean(unlist(evcv@result))
```

```
[1] 0.6795844
```

The average C-Index again indicates moderate prediction ability of the model. As can be seen from the vector of fold statistics it variance is small, i.e. the performance on was similar in each different fold.

```
> evcv2 <- evaluate(outcv, measure='PErrC', timegrid=3:80)
> plot(0, 0, col='white', xlim=c(0, 82), ylim=c(0, 0.4), xlab='time',
+      ylab='mean difference between estimated and observed survival',
+      main='Mean Prediction Error Curves')
> for(i in 1:10){
+ lines(3:80, evcv2@result[[i]], col=i+1)
+ }
>
```



Also here, one can observe that the prediction performance of the models on the different training folds decreases after 20 months.

1.5 Univariate feature selection

survHD allows the incorporation of a univariate pre-filter with any of the learning algorithms. This can help interpretability of models generated from algorithms which otherwise do not perform feature selection, as well as dramatically reducing computation times. **survHD** also provides a fast version of coxph **rowCoxTests**, for fitting the univariate Cox models to many features.

In **survHD**, variable selection is implemented in the function **geneSelection**. Parallel to **learnSurvival**, this function performs a variable selection for each resampling step if the argument **LearningSets** is passed. In our example, we use the method **fastCox**, i.e. univariate Cox models for each gene expression in **X**:

```
> gsel <- geneSelection(X=X, y=y, LearningSets=ls, method='fastCox',
+                       crit='coefficient')
```

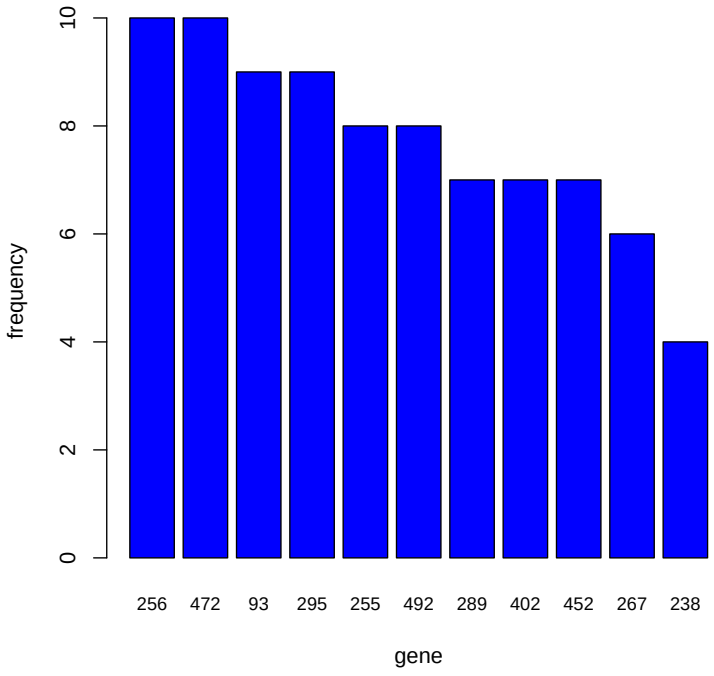
If the argument **crit** is set to **pvalue survHD** will store the p-values instead of the

coefficients in the return-object. This might be useful if one wants to set a certain threshold for the p-value of the variables included in a model lateron.

```
> str(gsel, max.level=2)

Formal class 'GeneSel' [package "survHD"] with 4 slots
  ..@ rankings  :List of 1
  ..@ importance:List of 1
  ..@ method    : chr "fastCox"
  ..@ criterion : chr "coefficient"
```

As can be seen from the previous code, **geneSelection** returns an object of class **GeneSel** which contains the rankings of the different gene expressions as well as their importance. Using the slot **rankings** we can easily see that there is only a single gene which belongs to the Top 10 in each fold.

```
> ranks <- c()
> for(i in 1:10)
+   ranks <- c(ranks, gsel@rankings[[1]][i, 1:10])
> genetab <- sort(table(ranks), decreasing=T)
> bp <- barplot(genetab[genetab>=3], col='blue', xlab='gene',
+             ylab='frequency',  cex.names=0.8)
> abs(gsel@importance[[1]][1, 1:10])

 gene472  gene256  gene255  gene238  gene492  gene402
4.869502 4.451491 3.707571 3.624200 3.449433 3.435399
 gene452  gene295  gene173  gene289
3.423935 3.393521 3.193918 3.104868

>
```



The last line of code illustrates the absolute coefficients for the top 10 genes in the first cross-validation fold.

1.6 Hyperparameter Tuning

Survival models for high-dimensional data usually incorporate a hyperparameter in order to adjust model complexity to the requirements of the data at hand. For most of these hyperparameters no good rule of thumb exists, and one has to resort to resampling approaches to optimize or tune the hyperparameter(s). As well as variable selection, the hyperparameter optimization has to be performed on each training set, resulting in a nested cross-validation or comparable resampling schemes (an inner or nested layer for tuning, and an outer layer for evaluation).

The function **tune** performs hyperparameter optimization for each training set of an (outer) resampling procedure by conducting an internal cross-validation loop. The hyperparameter grid is passed as a list of vectors containing the candidate values. Since we solely tune the number of boosting steps here the arguments **grids** is a one-element list with the vector of candidate numbers for the boosting steps. Please note, that the name of the vector has to exactly match the argument name of the tuned hyperparameter (here **stepno** as in the corresponding **CoxBoost**-function).

```
> tunecoxboost <- tune(y=y, X=X, survmethod='coxBoostSurv', LearningSets=ls,
+                 GeneSel=gsel, nbgene=30, grids=list(stepno=(1:5)*20))
```

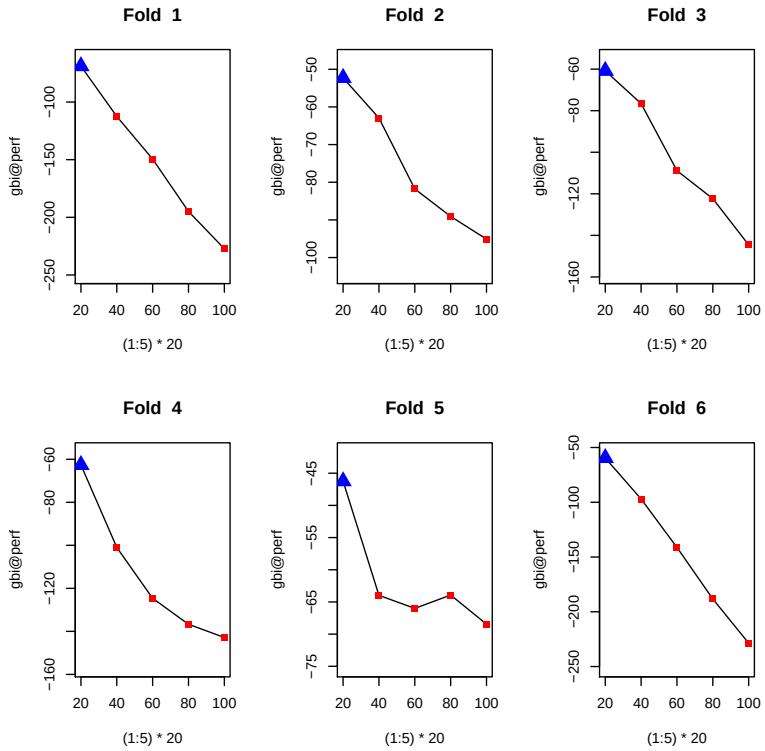
By additionally specifying the **GeneSel** argument and **nbgene=100**, **survHD** will tune the models based on the top 100 genes in each fold.

The best parameter of each fold **(res.ind)** as well as its index in the tuning grid can be obtained using the function **getBestParameters**. Parallel to the function **evaluate**, one can choose between several measures according to which the best parameter shall be determined. Here we use the cross-validated partial log likelihood.

```
> gb <- getBestParameters(tunecoxboost, res.ind=1, measure='CvPLogL')
>  gb@par

$stepno
[1] 20

> gb@bestind

[1] 1

> str(gb, max.level=2)

Formal class 'BestParameters' [package "survHD"] with 4 slots
  ..@ par    :List of 1
  ..@ bestind: int 1
  ..@ perf   : num [1:5] -68.9 -112.5 -149.5 -194.9 -227.2
  ..@ measure:Formal class 'CvPLogL' [package "survHD"] with 2 slots
```

We can also check how stable and distinct the tuning process has been, since **getBestParameters** returns the performance of all hyperparameter values:

```
> par(mfrow=c(2, 3))
> for(i in 1:6){
+   gbi <- getBestParameters(tunecoxboost, res.ind=i, measure='CvPLogL')
+  plot((1:5)*20, gbi@perf, type='l', col='black',
+       ylim=range(gbi@perf)*c(1.1, 0.9), main=paste('Fold ', i, sep=' '))
+   points((1:5)*20, gbi@perf, col='red', pch=15)
+    points(((1:5)*20)[gbi@bestind], gbi@perf[gbi@bestind],
+           col='blue', pch=17, cex=2)
+ }
```



In our case, the lowest number of boost-steps achieves the best result in all of the first 6 folds.

1.7 Complete Workflow

So, let us summarize the steps needed for a complete work flow. After loading the data, we have to choose the resampling scheme and create the corresponding LearningSets:

```
> ls <- generateLearningsets(y=y, method='CV', fold=10, strat=TRUE)
```

If necessary, the next step is variable selection using a filter method like **fastCox**. The generated resampling folds are passed via the argument **LearningSets**:

```
> gsel <- geneSelection(X=X, y=y, LearningSets=ls, method='fastCox',
+                       crit='coefficient')
```

Subsequently, for each training set a suitable hyperparameter has to be determined. If a filter method has been applied, the corresponding **GeneSel** object must be passed as well as the **LearningSets**:

```
> tunecoxboost <- tune(y=y, X=X, survmethod='coxBoostSurv', LearningSets=ls,
+                 GeneSel=gsel, nbgene=100, grids=list(stepno=(1:5)*20))
```

Finally, all three previously obtained outputs can be combined in a call to **learnSurvival**, which will fit the requested survival model on each learningset using the variables selected in **GeneSel** and tune the hyperparameters according to their performance in **tuneres**. Regarding the incorporation of a **tuneres** argument, one must additionally specify by which measure to evaluate the tuning results. This measure is specified in the argument **addtune** which is a list of arguments which is internally passed to function **evaluate**. Cross-validated partial log-likelihood is standard, but any available evaluation metrics may be used.

```
> cv10 <- learnSurvival(y=y, X=X, tuneres=tunecoxboost,
+           survmethod='coxBoostSurv', LearningSets=ls, GeneSel=gsel,
+                     nbgene=30, addtune=list(measure='CvPLogL'))
```

The function **evaluate** can be used to assess the performance of the survival method. The performance is returned separately for each fold, and here we combine these using a simple average:

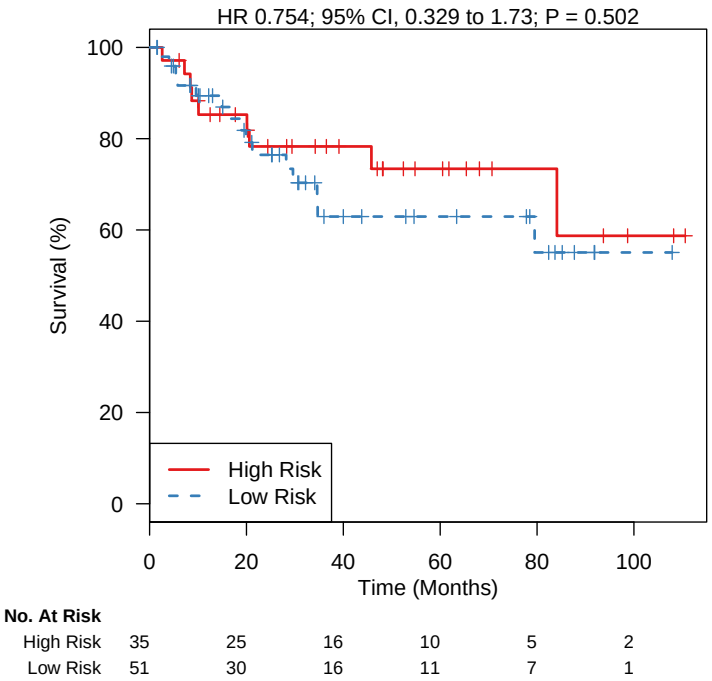
```
> mean(unlist(evaluate(cv10,  measure='CvPLogL')@result))

[1] -12.30361

> mean(unlist(evaluate(cv10,  measure='HarrellC')@result))

[1] 0.501919
```

We also illustrate the performance of the models computed in the learnSurvival-step. Using the **predict** function on an object of class **LearnOut (with type='cp')** causes **survHD** to use the linear predictors –obtained for all test observations– to categorize samples into two risk classes.

```
> preds <- predict(cv10, type='cp', voting_scheme='first')
> table(preds)

preds
High Risk  Low Risk
       35        51
```

Subsequently, one can plot the corresponding Kaplan Meier curves. In **survHD** this can be achieved by simply using the function **plot** on the previous **LearnOut**-object.

```
>  plot(cv10)
```



1.8 Comparison of two models

In **survHD** two models can also be compared in a more direct way than only evaluating them according to a certain measure. The function **IDI.INF** from the package **survIDINRI** –among other measures– computes the median improvement in riskscore of a model in comparison with a null model. This feature is implemented in the function **compare**. We first have to fit two models using **learnSurvival**. The first one is the CoxBoost model described above whereas the second one is constructed using the plusMinus-algorithm:

```
> Xtrain <- Xtrain[1:60, ]
> ytrain <- y[1:60, ]
> coxboostmodel1 <- learnSurvival(y=ytrain, X=Xtrain,
+  survmethod='coxBoostSurv', stepno=20)
> coxboostmodel2 <- learnSurvival(y=ytrain, X=Xtrain,
+ survmethod='coxBoostSurv',  stepno=40)
```

After fitting the models, they can be compared with regard to certain test data. Here we use the Integrated Discrimination Improvement Index (**IDI**):

```
> ytest <- y[61:86, ]
> Xtest <- X[61:86, ]
> comp1<-compare(coxboostmodel1, coxboostmodel2, measure='IDI',newdata=Xtest,
+         newy=ytest, t0=50)
> comp1[[1]]@estimate

-0.003147978
```

```
> comp2<-compare(coxboostmodel2, coxboostmodel1,measure='IDI',
+         newdata=Xtest, newy=ytest, t0=50)
> comp2[[1]]@estimate

0.003147978

> str(comp2[[1]],max.level=2)

Formal class 'IDI' [package "survHD"] with 2 slots
  ..@ estimate: Named num 0.00315
  .. ..- attr(*, "names")= chr ""
  ..@ conf.int: Named num [1:2] -0.0229 0.0414
  .. ..- attr(*, "names")= chr [1:2] "2.5%" "97.5%"
```

The class bf MeasureOut also contains a confidence interval for the computed measure as can be seen from the output. Other measures for comparison of two survival models in **survHD** are:

- **NRI** Category-less Net Reclassification Index
- **MIRS** Median Improvement in Risk Score
- **CDelta** Difference in C-Statstic

2 Custom Survival Models

The **survHD** package provides an interface for incorporating user defined survival methods into its framework. For that purpose, two functionalities are needed:

- model fitting
- prediction from previously fitted models

Both features are implemented by defining a single function. In our example we ill add the method **rsf** from the package **randomSurvivalForest** to **survHD** as a custom function **customRSF**. The user-defined function has to accept at least three predefined inputs:

- **Xlearn:** A **data.frame** of gene expressions for the current training data(columns are genes)
- **Ylearn:** the survival response for the current training data
- **learnind**: the indices of the current training observations inside the complete data set **X** which is usually passed to functions like **learnSurvival**. Using this argument, one can e.g. extract the relevant observations from additional clinical covariates which can be passed using the ... argument.

```
customRSF <- function(Xlearn, Ylearn, learnind, ...){
                ###load required packages
                require(randomSurvivalForest)
                ###handle inputs
                ll <- list(...)
                datarsf <- data.frame(Xlearn, time=Ylearn[, 1], status=Ylearn[, 2])
```

```
ll$data <- datarsf
ll$formula <- as.formula('Surv(time, status)~.')

##call actual model function rsf from randomSurvivalForest
output.rsf <- do.call("rsf",  args = ll)
...}
```

First, one typically has to load or source a function or package which performs the actual model fitting. The next step is the processing of the inputs. In our case we do not use any additional covariates, so we only have to convert **Xlearn** and **Ylearn** into the input format of the function **rsf**. As can be seen from the code, this function accepts a **formula** and a **data.frame** containing all necessary variables. Moreover, we pass all the arguments represented by **...** argument and eventually call the function **rsf** using **do.call**. After this the model fitting is complete.

In order to provide outputs which can be evaluated and processed by **survHD** we also need a **predict** function. This function should either provide an object of class **LinearPrediction** or **SurvivalProbs** whose slots can be found below:

```
>  LinearPrediction <- predict(coxboostmodel, newdata=Xtest, type='lp')[[1]]
>  str(LinearPrediction, max.level=2)

Formal class 'LinearPrediction' [package "survHD"] with 1 slots
  ..@ lp: Named num [1:26] 1.952 -0.1976 -0.0529 -0.1154 -0.5334 ...
  .. ..- attr(*,  "names")= chr [1:26] "L59" "L61" "L62" "L64" ...

>  SurvivalProbs <- predict(coxboostmodel, newdata=Xtest, type='SurvivalProbs',
+  timegrid=3:80)[[1]]
>  str(SurvivalProbs, max.level=2)
```

The prediction function has to accept four arguments:

- **object:** an object of class **ModelCustom** which contain is created by the **survHD** automatically and contains the fitted model (here: **output.rsf**) in its slot **mod**.
- **newdata: data.frame** of geneexpressions for the observations for which predictions shall be performed.
- **type:** indicates whether linear predictors (**'lp'** or survival probabilities (**'SurvivalProbs'**) shall be predicted
- **timegrid:** if **type='SurvivalProbs'** this argument specifies the time points at which predictions shall be performed

```
##define prediction function which will be stored in slot predfun
##and called by predictsurvhd (signature(ModelCustom))
###
predfun <- function(object, newdata, type, timegrid=NULL, ...){
require(randomSurvivalForest)
#either type lp or type SurvivalProbs must be implemented
#for typ lp the obligatory return class is LinearPrediction
        if (type == "lp") {
       stop("Random Forests don't provide linear predictors,  sorry.")
        }
```

```
#for typ SurvivalProbs the obligatory return class is Breslow
        else if (type == "SurvivalProbs") {
            modelobj <- object@mod
            if (is.null(timegrid)) {
                stop("No timegrid specified.")
            }

###create data for which predictions are to be performed
###function checks for response but does not use it (fake response)
predsrsf <- predict.rsf(object = modelobj,  test=data.frame(newdata,
time=rexp(n=nrow(newdata)), status=sample(c(0, 1), nrow(newdata), replace=T)))
###predict-function provides predictions for training times only
###->interpolate for timepoints in timegrid
curves <- exp(-t(apply(predsrsf$ensemble, 1, FUN=function(z)
approx(x=predsrsf$timeInterest, y=z, xout=timegrid)$y)))
###create breslow-object
pred <- new("breslow",  curves = curves,  time = timegrid)
###create SurvivalProbs-object embedding the breslow-object
            pred <- new("SurvivalProbs",  SurvivalProbs = pred)
        }
else stop('Invalid "type" argument.')
return(pred)
}
```

This function is structured into two sections which correspond to predicting linear predictors and survival probabilities respectively. Since random survival forests are not capable of providing linear predictors the first section simply returns an error. In the survival probability section at first the input data have to be preprocessed for the function **predict.rsf** which can perform predictions on the basis of objects of class **randomSurvivalForest** created by the function **rsf**. Subsequently, this predictions are estimated and eventually an object of class **SurvivalProbs** is created which is the predefined class for survival probabilities in **survHD**. Its only slot **SurvivalProbs** is an object of class **Breslow** which not only stores the survival probabilities in the slot **curves** but also the time points in slot **time**. This **survprob** object is finally returned by the custom prediction function.

The definition of this prediction function was the second part of the user-defined survival function **customRSF**. In the end, we still have to create the obligatory output object of class **ModelCustom** which primarily consist of the fitted model object **output.rsf** in slot **mod** and the user-defined prediction function in slot **predfun**. Additional information can be stored in the slot **extraData** which must be of class **list**:

```
  ###now create customsurvhd-object (which is the obligatory output-object)
custommod <- new("ModelCustom", mod=output.rsf, predfun=predfun, extraData=list())

 return(custommod)
```

Just for the sake of completeness, the complete user defined survival method looks like this. It basically consists of three parts: model fitting, prediction function, creating of the **customsurvhd** object:

```
 ###random survival forest as a custom survival model function
```

```
##Xlearn and Ylearn are obligatory inputs
customRSF <- function(Xlearn, Ylearn, learnind, ...){
  ###load required packages
  require(randomSurvivalForest)
  ###handle inputs

  ll <- list(...)
  datarsf <- data.frame(Xlearn, time=Ylearn[, 1], status=Ylearn[, 2])
  ll$data <- datarsf
  ll$formula <- as.formula('Surv(time, status)~.')

  ##call actual model function rsf from randomSurvivalForest
  output.rsf <- do.call("rsf",  args = ll)

  ##define prediction function which will be stored in slot predfun
  ##and called by predictsurvhd (signature(ModelCustom))
  predfun <- function(object, newdata, type, timegrid=NULL, ...){
  require(randomSurvivalForest)
  #either type lp or type SurvivalProbs must be implemented
  #for typ lp the obligatory return class is LinearPrediction
        if (type == "lp") {
        stop("Random Forests don't provide linear predictors,  sorry.")
        }
  #for typ SurvivalProbs the obligatory return class is Breslow
  else if (type == "SurvivalProbs") {
  modelobj <- object@mod
  if (is.null(timegrid)) {
    stop("No timegrid specified.")
        }

        ###create data for which predictions are to be performed
        ###function checks for response but does not use it (fake response)
        predsrsf <- predict.rsf(object = modelobj,  test=data.frame(newdata,
        time=rexp(n=nrow(newdata)), status=sample(c(0, 1), nrow(newdata), replace=T)))
        ###predict-function provides predictions for training times only
        ###->interpolate for timepoints in timegrid
        curves <- exp(-t(apply(predsrsf$ensemble, 1, FUN=function(z) approx(
        x=predsrsf$timeInterest, y=z, xout=timegrid)$y)))
        ###create breslow-object
        pred <- new("breslow",  curves = curves,  time = timegrid)
        ###create SurvivalProbs-object embedding the breslow-object
        pred <- new("SurvivalProbs",  SurvivalProbs = pred)
        }

        else stop('Invalid "type" argument.')
        return(pred)
        }


        ###now create customsurvhd-object (obligatory output-object)
        custommod <- new("ModelCustom", mod=output.rsf,
        predfun=predfun, extraData=list())

        return(custommod)
```

```
}
```

It is practical to make sure that the custom function in the global environment of **R** such that it is available for all functions of **survHD**.

```
> ###define function in global envir
> assign(x="customRSF", value=customRSF, envir=.GlobalEnv)
```

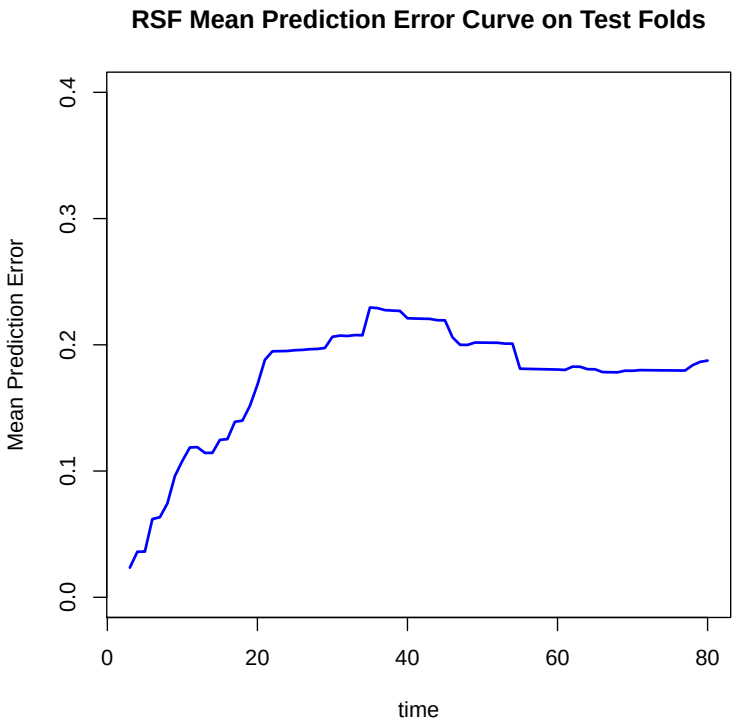
Using the custom function **customRSF** we can easily implement a complete work flow of generating LearningSets, gene selection, tuning, resampling and a final evaluation.

```
> ##learningset
> ls <- generateLearningsets(y=y[, 1], method='CV', fold=5)
> #gene selection
> gsel <- geneSelection(X=X, y=y, method='fastCox', LearningSets=ls,
+                       criterion='coefficient')
> ###tune
> tuneres <- tune(X=X, y=y, GeneSel=gsel, nbgene=30, survmethod='customSurv',
+ customSurvModel=customRSF, LearningSets=ls, grids = list(ntree = 20*(1:10)))
> ###use tuneres in learnSurvival
> svaggr <- learnSurvival(X=X, y=y, GeneSel=gsel, nbgene=30, survmethod='customSurv',
+ customSurvModel=customRSF, LearningSets=ls, tuneres=tuneres, measure="PErrC",
+ timegrid=4:10, gbm=FALSE, addtune=list(GeneSel=gsel, nbgene=30))
```

The only essential difference is the argument **survmethod** which is set to **'customSurv'**.

Note that even tuning does not need any additional code apart from defining a reasonable tuning grid (argument **grids**) for the **rsf**-specific argument **ntree**. The last step is the evaluation where we have to resort to **measure='PErrC'** since it is the only one which can be computed without a linear predictor:

```
> ###error expected because rsf does not provide lp
> try(evaluate(svaggr, measure='CvPLogL'))
> ###error expected because rsf does not provide lp
> try(evaluate(svaggr, measure='PErrC', timegrid=3:80, gbm=T))
> ###works without lp
> evcust <- evaluate(svaggr, measure='PErrC', timegrid=3:80, gbm=F)

> perrcs <- c()
> for(i in 1:5){
+   perrcs <- cbind(perrcs, evcust@result[[i]])
+ }
> plot(3:80, rowMeans(perrcs, na.rm=TRUE), , col='blue', lwd=2,
+      main='RSF Mean Prediction Error Curve on Test Folds', xlab='time',
+      ylab='Mean Prediction Error',
+      type='l', ylim=c(0, 0.4), xlim=c(3, xlim2))
```



The lower mean prediction error curve suggests a better performance on the data compared to the previous results for **CoxBoostSurv**.

A Session Info

- R version 2.15.0 (2012-03-30), `i486-pc-linux-gnu`
- Locale: `LC_CTYPE=de_DE.utf8`, `LC_NUMERIC=C`, `LC_TIME=de_DE.utf8`, `LC_COLLATE=de_DE.utf8`, `LC_MONETARY=de_DE.utf8`, `LC_MESSAGES=de_DE.utf8`, `LC_PAPER=C`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=de_DE.utf8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, graphics, grDevices, methods, splines, stats, utils
- Other packages: Biobase 2.16.0, BiocGenerics 0.2.0, CoxBoost 1.3, gbm 1.6-3.2, Hmisc 3.9-3, KernSmooth 2.23-8, lattice 0.20-6, Matrix 1.0-6, penalized 0.9-41, prodlim 1.3.2, randomSurvivalForest 3.6.3, RColorBrewer 1.0-5, survC1 1.0-1, survcomp 1.6.0, survHD 0.99.0, survIDINRI 1.0-1, survival 2.36-14
- Loaded via a namespace (and not attached): bootstrap 2012.04-0, cluster 1.14.2, compiler 2.15.0, grid 2.15.0, rmeta 2.16, SuppDists 1.1-8, survivalROC 1.0.0, tools 2.15.0