survHDExtra
package vignette

Christoph Bernau [*]
Levi Waldron [†]
Markus Riester[‡]

2012

1 SurvHDExtra–Additonal features for survHD

This package provides additional features for the Bioconductor package **survHD** which is meant for survival analysis on highdimensional data. The most important features currently implemented are additional survival methods (e.g. **customPenalized**, customUniCox, customSuperPc or **customRSF**)which can be included into the **survHD** framework via the **customSurv** interface. Moreover, **survHDExtra** provides several helpful methods for geneset analysis which are also introduced here.

2 Custom Survival Models

The **survHD** package provides an interface for incorporating user defined survival methods into its framework. As mentioned above, several of such 'custom survival functiaons' are implemented in this package. If necessary, however, you can also include your own user-defined survival mathods (you also might consider sending us your function or posting it on github for inclusion into **survHDExtra**). 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

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

- **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.

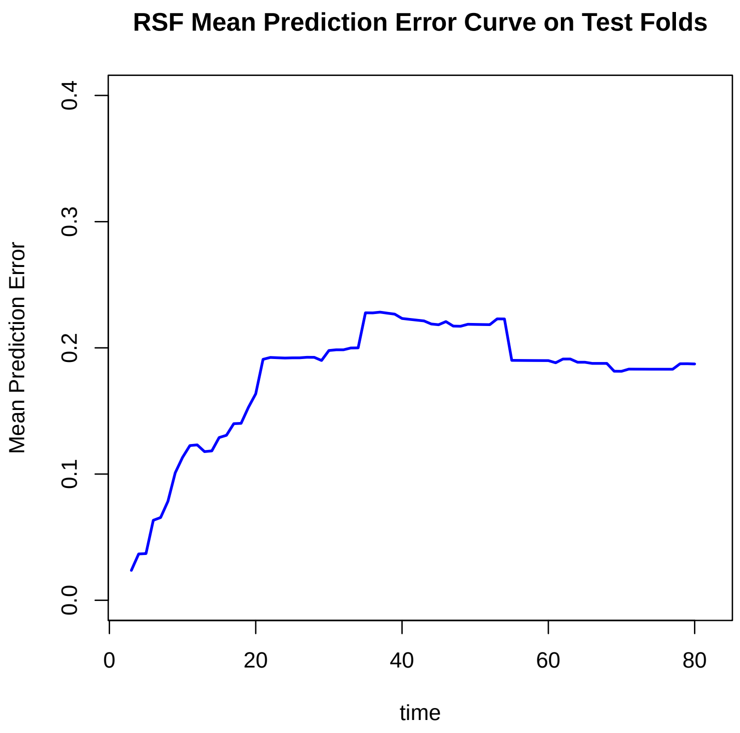
```
> ##learningset
> ls <- generateLearningsets(y=y[, 1], method='CV', fold=5)
> #gene selection
> gsel <- geneSelection(X=X, y=y, method='fastCox', LearningSets=ls,
+                    criterion='coefficient')
> ###tune
> tuneres <- tune(X=X, y=y, GeneSel=gsel, nbgene=30, survmethod='customSurv',
+ 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, 82))
```



3 Gene Set Analysis

It is often of interest to test whether the expression of a particular group of genes is associated with some phenotype. This is commonly a phenotype that differs between two groups, for example between a test and a control group. This test is easily generalized to different phenotype classes like survival outcome by first ranking all genes in a univariate fashion and then testing for a non-random ranking of gene sets. We first create a random data set:

```
> set.seed(100)
> x  <- matrix(rnorm(1000*20), ncol=20)
> dd <- sample(1:1000, size=100)
> u  <- matrix(2*rnorm(100), ncol=10, nrow=100)
> x[dd, 11:20] <- x[dd, 11:20]+u
> y <- Surv(c(rnorm(10)+1, rnorm(10)+2),  rep(TRUE,  20))
> genenames   <- paste("g", 1:1000, sep="")
> rownames(x) <- genenames
```

Now we need some sets of genes which we will test for association with outcome. Here we just create some random ones:

```
> genesets <- vector("list", 50)
> for(i in 1:50){
+     genesets[[i]] <- paste("g", sample(1:1000, size=30), sep="")
+ }
> geneset.names <- paste("set", as.character(1:50), sep="")
```

A well curated resource of gene sets is the MSigDB. It provides gene sets in tab separated files with the suffix gmt. These files can be imported with the `gsaReadGmt` function. For now, we just create such a gmt data structure from the list of random sets:

```
> gmt <- new("gsagenesets",  genesets=genesets,
+                  geneset.names=geneset.names)
```

The focus of this package is survival outcome, so testing gene sets for association with survival is simple:

```
> gsa.res <- gsaWilcoxSurv(gmt, X=x, y=y, cluster=FALSE, p.value=0.3)
```

The `cluster` command can be used to cluster similar gene sets in the ranking. The similarity of gene sets can also be explored with the `plot` function and the results are shown in Figure~1:

```
> plot(gsa.res)
```

Another useful plot is a barplot, which visualizes the ranking of genes in up to two gene sets (Figure~2):

```
> plot(gsa.res,  type="barcode", geneset.id1="set22",
+     geneset.id2="set33")
```

For phenotypes other than survival, `gsaWilcoxSurv` accepts a ranking of genes. In our example, we split the data in two groups and test gene sets for association with this splitting:

```
> library(genefilter)
> genes.ttest <- rowttests(x,  as.factor(c(rep(1, 10), rep(2, 10))))
> gsa.res.tt  <- gsaWilcoxSurv(gmt,  genenames=rownames(x),
+     statistics=genes.ttest[, 1])
```

All the examples above assume that the rownames of `X` correspond to the names of the genes in the gene sets. Here an example in which the rownames of `X` are Affymetrix probe ids and the genes in the gene set official gene symbols:

```
> data(beer.exprs)
> data(beer.survival)
> library(hu6800.db)
> library(annotate)
> gmt <- gsaReadGmt(system.file("extdata/ovarian_gene_signatures.gmt",
+     package = "survHD"))
```

We convert the symbols to Affymetrix ids with the `gsaTranslateGmt` function and Bioconductor's annotate package:

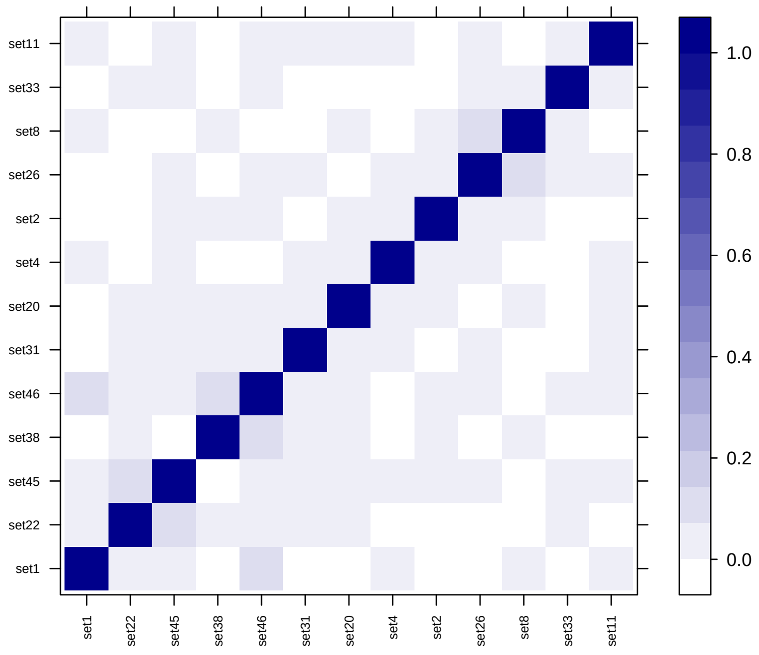


Figure 1: GSA overlap plot. This plot visualizes the similarity of gene sets that are associated with the phenotype of interest. Note that the p-values are not adjusted for multiple testing.

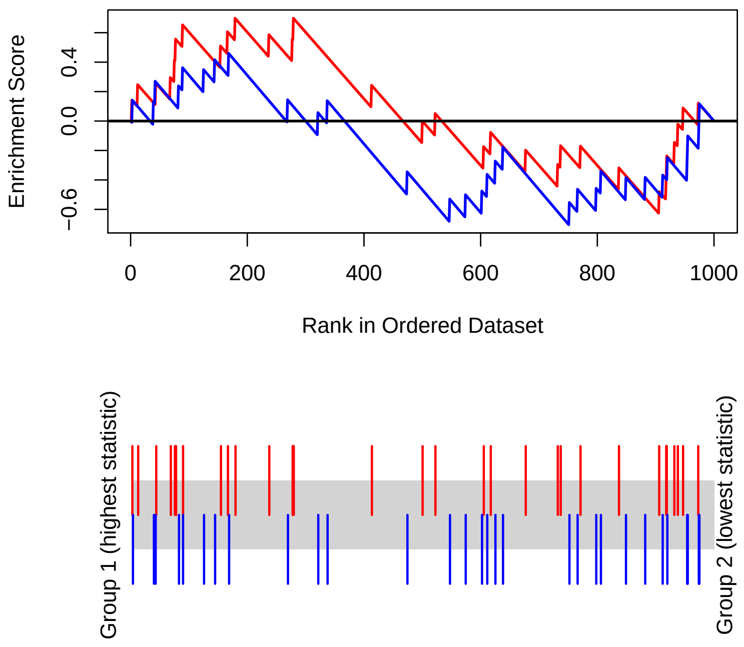


Figure 2: GSA barplot. The horizontal lines on the bottom visualize the positions of up to two different gene sets in a univariate ranking, colored here in red and blue. The plot on the top visualizes the local enrichment. It makes sense to compare two sets in one plot if the gene sets are devided in up- and down-regulated genes. This has the advantage that not only non-randomness is shown, but also that the gene expression direction is consistent. Red are typically the up-regulated genes, blue down-regulated genes. Genes with high statistic have a high hazard ratio and vice versa.

```
> genes    <- getSYMBOL(rownames(beer.exprs),  "hu6800")
> gmt.affy <- gsaTranslateGmt(gmt,  beer.exprs,  genes)
> gsa.res  <- gsaWilcoxSurv(gmt.affy,  beer.exprs,
+                  Surv(beer.survival[, 2],  beer.survival[, 1]))
```

Note that if a gene is represented by more than one probe set, this code will use the first match in the GSA. It is recommended to use for example the `collapseRows` function of the WGNCA package to pick reasonable probe sets.

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=C`, `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, grDevices, graphics, methods, splines, stats, utils
- Other packages: AnnotationDbi~1.18.1, Biobase~2.16.0, BiocGenerics~0.2.0, DBI~0.2-5, Hmisc~3.9-3, RSQLite~0.11.1, annotate~1.34.1, gbm~1.6-3.2, genefilter~1.38.0, hu6800.db~2.7.1, lattice~0.20-6, limma~3.12.1, org.Hs.eg.db~2.7.1, penalized~0.9-41, randomSurvivalForest~3.6.3, survC1~1.0-1, survHD~0.99.0, survHDExtra~0.1.0, survival~2.36-14
- Loaded via a namespace (and not attached): IRanges~1.14.4, XML~3.9-4, cluster~1.14.2, grid~2.15.0, stats4~2.15.0, tools~2.15.0, xtable~1.7-0