SimulatorZ package vignette

Yuqing Zhang*, Christoph Bernau†, Levi Waldron‡

October 13, 2014

Contents

1 Introduction

simulatorZ is a package for generating realistic simulations of a range of genomic data types, based on real experimental data contained in *ExpressionSet* or *SummarizedExperiment* data containers. It can generate simulations of a single dataset based on a reference experimental dataset, or of a collection of independent datasets based on a reference collection. Details of the simulation methodology are described by Bernau et al. [1] and in section 2.1. Briefly, standard non-parametric bootstrap is used to sample (with replacement) individuals from experiments and to sample experiments from a collection. Optionally, a form of parametric bootstrap is used to simulate censored time-to-event outcomes based on the baseline hazard, follow-up time, and association between outcome and genomic features found in the experimental datasets.

Development of this package was motivated by the need to simulate collections of independent genomic data sets, and to perform training and validation with prediction algorithms across independent datasets. It derived from studies concerning the performance of risk prediction algorithms across independent transcriptome studies of cancer patients.

*zhangyuqing.pkusms@gmail.com
†Christoph.Bernau@lrz.de
‡levi.waldron@hunter.cuny.edu

Cross study validation (CSV) is an alternative to traditional cross-validation (CV) in research domains where predictive modeling is applied to high-dimensional data and where multiple comparable datasets are publicly available. Recent studies have both shown the potential power and cast questions upon this method. For example, Bernau et al. noticed a dramatic drop of CSV performance compared to that of CV. *simulatorZ* enables statistical investigation of potential causes for this discrepancy through realistic simulation studies.

In this vignette, we give a short tour of the package and will show how to use it for interesting tasks.

2 Data Set Simulation

2.1 Approach in the simulation

The *simulatorZ* package implements a resampling scheme illustrated in Bernau et al. [1] to simulate a collection of independent genomic sets with gene expression profiles and time to event outcomes. The resampling procedure involves both parametric and non-parametric bootstrap. The whole process can be divided into three steps:

- non-parametric bootstrap at set level.
  In this step, we sample the set labels with replacement, from a list of sets we use as original ones. This will capture the differences between studies, and the variability of whether the studies are accessible.
- non-parametric bootstrap at patient level.
  We resample observations with replacement from each set corresponding to the randomly drawn labels obtained from step 1.
- parametric bootstrap to simulate time to event.
  The parametric step involves a generative model that combines the truncated inversion method in Bender et al. (2005) [2], the Nelson-Aalen estimator for cumulative hazard functions, and CoxBoost method (Binder and Schumacher, 2008)[3] to simulate survival times.

simulatorZ makes use of this approach and improves it by applying a stratified non-parametric bootstrap at the patient level (step 2), to balance the prevalence of covariates in the simulated sets, thus allowing for more diverse purpose of use in similar researches.

The package provides separate functions for both parametric and non-parametric step in the simulation(`simData`, `simTime`), as well as one driver function to complete the whole process(`simBootstrap`). It supports both *ExpressionSets* and *SummarizedExperiment* object and the examples below will demonstrate how to use the functions for both these types respectively.

Examples in this vignette uses *ExpressionSets* from *curatedOvarianData* [4] and *SummarizedExperiment* from *parathyroidSE* in *Bioconductor*. Users who are not familiar with this two types of object could refer to the corresponding manuals for more information.

2.2 Simulation on ExpressionSets

2.2.1 non-parametric bootstrap

First of all, we use the *curatedOvarianData* package [4] to obtain a list of *ExpressionSets*. A collection of 14 datasets with overall survival information could be obtained as follows, as demonstrated in the *curatedOvarianData* package:

```
> library(curatedOvarianData)
> source(system.file("extdata",
+                    "patientselection.config",package="curatedOvarianData"))
> source(system.file("extdata", "createEsetList.R", package="curatedOvarianData"))
```

However this takes a while, so to get a smaller dataset quickly we simply select three datasets and construct them as a list of called `esets` including 3 *ExpressionSets*.

```
> library(curatedOvarianData)
> data(GSE17260_eset)
> data(E.MTAB.386_eset)
> data(GSE14764_eset)
> esets.orig <- list(GSE17260=GSE17260_eset, E.MTAB.386=E.MTAB.386_eset, GSE14764=GSE14764_eset)
```

This function help clean the list of ExpressionSets, providing options to select only features (rows) common to all datasets (keep.common.only=TRUE), and select only samples having certain annotations available (if meta.required is a vector of columns found in phenoData(eset):

```
> cleanEsets <- function(obj, keep.common.only=TRUE, meta.required=NULL){
+     if(keep.common.only){
+         intersect.features <- Reduce(intersect, lapply(obj, featureNames))
+         for (i in 1:length(obj))
+             obj[[i]] <- obj[[i]][intersect.features, ]
+     }
+     if(!is.null(meta.required)){
+         obj <- lapply(obj, function(obj1){
+             for (x in meta.required)
+                 obj1 <- obj1[, !is.na(obj1[[x]])]
+             if(ncol(obj1) > 0){
+                 return(obj1)
+             }else{
+                 return(NULL)
+             }
+         })
+         return(obj[!sapply(obj, is.null)])
+     }
+ }
```

Now create a list of ExpressionSets that contains only samples where overall survival is known, and with only the intersection of features available in all datasets. To speed later simulations, we keep only 1000 features.

```
> esets <- cleanEsets(esets.orig, meta.required=c("days_to_death", "vital_status"))
> esets <- lapply(esets, function(eset) eset[1:1000, ])
> sapply(esets, dim)
```

```
         GSE17260 E.MTAB.386 GSE14764
Features     1000       1000     1000
Samples       110        129       80
```

This will be our list of original sets for further simulation. Noted that here we only take a subset for the convenience. It is not obligatory for the sets to have the same number of observations. However, if performing the parametric bootstrap with multiple datasets it is important that they should have the SAME features.

`simData` is the function to perform non-parametric bootstrap resampling at both set and patient level. So if we want to simulate a collection of sets each containing 500 observations with the same method mentioned above, we can use `simData` like this:

```
> set.seed(8)
> sim.esets <- simData(esets, n.samples=500)
```

```
[1] 3 2 1
[1] "covariate: NULL"
```

```
> names(sim.esets)
```

```
[1] "obj"          "indices"      "setsID"       "y.vars"       "prob.desired"
[6] "prob.real"
```

The output includes another list of the simulated data sets. Elements of this list are:

- `obj`: the simulated data, as a list of *ExpressionSet* or *SummarizedExperiment* objects.
- `setsID` and `indices`: these indicate original set and patient labels selected in the simulation process, making it easy to keep track of the data.
- `prob.desired` and `prob.real` will be discussed in detail later.

By default, `simData` performs a two-step bootstrap: the first step samples complete datasets, and the second step samples individuals from each dataset. If we prefer instead to sample individuals from each original dataset, but keep all datasets without resampling at the dataset, set the `type` argument to "one-step":

```
> sim.esets <- simData(esets, 500, type="one-step")
```

```
[1] 1 2 3
[1] "covariate: NULL"
```

2.2.2 Balance the Prevalence of Covariates

As we mentioned before, *simulatorZ* supports balancing the distribution of covariates which is also done with `simData`. The parameter `balance.variables` is a string or a vector of strings that gives the names of covariates to be balanced. This makes the distribution of the covariate(or joint distribution of all those covariates) resemble that in all the original sets combined, by re-weighting the probability of sampling. For instance, we will balance `"tumorstage"` like this:

First, we eliminate observations missing the value in "tumorstage" (if "tumorstage" is not available in any set, it is ruled out from the original set).

```
> cleaned.esets <- cleanEsets(esets.orig, meta.required="tumorstage")
```

Then we can balance the distribution of covariates like:

```
> sim.sets <- simData(cleaned.esets, 500,
+                     balance.variables="tumorstage")
```

```
[1] 3 2 1
[1] "covariate: tumorstage"
```

```
> sim.sets$prob.desired
```

```
covariate_all
          1           2           3           4
0.025078370 0.006269592 0.849529781 0.119122257
```

```
> sim.sets$prob.real[[1]]
```

```
covariate_all
          3           4           3           3           3           3           3
0.009430351 0.007233962 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351
          3           3           3           3           3           3           3
0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351
          3           3           3           3           3           4           3
0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.007233962 0.009430351
          3           3           3           3           3           3           3
0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351
          3           3           3           4           3           4           3
0.009430351 0.009430351 0.009430351 0.007233962 0.009430351 0.007233962 0.009430351
          3           3           3           3           3           3           3
0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351
          3           3           3           3           3           3           3
0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351
          3           4           4           4           3           3           3
```

```
0.009430351 0.007233962 0.007233962 0.007233962 0.009430351 0.009430351 0.009430351
          3           3           3           3           3           3           3
0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351
          3           3           3           4           3           3           3
0.009430351 0.009430351 0.009430351 0.007233962 0.009430351 0.009430351 0.009430351
          3           3           3           3           3           3           3
0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351
          3           4           4           3           3           3           3
0.009430351 0.007233962 0.007233962 0.009430351 0.009430351 0.009430351 0.009430351
          3           3           4           3           3           3           3
0.009430351 0.009430351 0.007233962 0.009430351 0.009430351 0.009430351 0.009430351
          3           3           3           3           3           3           3
0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351 0.009430351
          3           4           3           4           4           3           3
0.009430351 0.007233962 0.009430351 0.007233962 0.007233962 0.009430351 0.009430351
          4           3           4           3           4
0.007233962 0.009430351 0.007233962 0.009430351 0.007233962
```

Now we will explain the output `prob.desired` and `prob.real`. These two output are only useful when we consider the distribution of covariates. `prob.desired` is the overall distribution calculated by combining all observations together. `prob.real` is a list of probablities. Each elements in the list corresponds to the covariate value of patients in certain original set. For example, in the codes above, `prob.real[[1]]` prints out the tumorstage value, and the probability of being sampled in the first original data set.

We can also consider the joint distribution of two covariates at the same time. Like if we want to re-weight the resampling probability considering both tumorstage and grade, first we need to do similar cleaning:

```
> cleaned.esets <- cleanEsets(esets.orig, meta.required=c("tumorstage", "grade"))
```

Then we set `balance.variables` as a character vector:

```
> sim.sets <- simData(cleaned.esets, 500,
+                     balance.variables=c("tumorstage", "grade"))
```

```
[1] 2 1
[1] "covariate: tumorstage" "covariate: grade"
```

```
> sim.sets$prob.desired
```

```
covariate_all
        1 1         1 2         1 3         2 3         3 1         3 2         3 3
0.005263158 0.005263158 0.031578947 0.005263158 0.131578947 0.315789474 0.405263158
        4 1         4 2         4 3
0.015789474 0.015789474 0.068421053
```

```
> sim.sets$prob.real[[1]]
```

```
covariate_all
        3 3         4 3         3 3         3 3         3 3         3 1         3 3
0.013294199 0.006529382 0.013294199 0.013294199 0.013294199 0.006005285 0.013294199
        3 2         3 3         3 2         3 2         3 1         3 1         3 3
0.008723466 0.013294199 0.008723466 0.008723466 0.006005285 0.006005285 0.013294199
        3 1         3 2         3 2         3 3         3 3         4 2         3 2
0.006005285 0.008723466 0.008723466 0.013294199 0.013294199 0.005524862 0.008723466
        3 2         3 3         3 3         3 3         3 1         3 3         3 2
0.008723466 0.013294199 0.013294199 0.013294199 0.006005285 0.013294199 0.008723466
        3 2         3 3         3 3         4 3         3 2         4 3         3 3
0.008723466 0.013294199 0.013294199 0.006529382 0.008723466 0.006529382 0.013294199
        3 3         3 3         3 2         3 2         3 2         3 1         3 2
```

```
0.013294199 0.013294199 0.008723466 0.008723466 0.008723466 0.006005285 0.008723466
        3 3         3 2         3 1         3 2         3 2         3 3         3 3
0.013294199 0.008723466 0.006005285 0.008723466 0.008723466 0.013294199 0.013294199
        3 2         4 3         4 3         4 3         3 1         3 2         3 2
0.008723466 0.006529382 0.006529382 0.006529382 0.006005285 0.008723466 0.008723466
        3 2         3 3         3 3         3 2         3 2         3 2         3 2
0.008723466 0.013294199 0.013294199 0.008723466 0.008723466 0.008723466 0.008723466
        3 1         3 1         3 3         4 1         3 2         3 2         3 2
0.006005285 0.006005285 0.013294199 0.005524862 0.008723466 0.008723466 0.008723466
        3 1         3 3         3 2         3 3         3 1         3 2         3 3
0.006005285 0.013294199 0.008723466 0.013294199 0.006005285 0.008723466 0.013294199
        3 2         4 3         4 3         3 1         3 3         3 1         3 3
0.008723466 0.006529382 0.006529382 0.006005285 0.013294199 0.006005285 0.013294199
        3 1         3 3         4 2         3 1         3 2         3 2         3 1
0.006005285 0.013294199 0.005524862 0.006005285 0.008723466 0.008723466 0.006005285
        3 1         3 1         3 1         3 1         3 2         3 3         3 1
0.006005285 0.006005285 0.006005285 0.006005285 0.008723466 0.013294199 0.006005285
        3 2         4 2         3 1         4 1         4 3         3 3         3 2
0.008723466 0.005524862 0.006005285 0.005524862 0.006529382 0.013294199 0.008723466
        4 3         3 2         4 1         3 2         4 3
0.006529382 0.008723466 0.005524862 0.008723466 0.006529382
```

Note that this function is only suitable for discrete covariates. We need to discretize the continuous variables first if we wish to consider them.

Before we move to the next part, there is one more comment on this function. `simData` does not ask for a specific response variable, even though it has a parameter for it. This is useful when we use `simBootstrap` directly. If a y variable is specified, `simData` will pass it to the output without dealing with it.

2.2.3 Parametric Boostrap

To simulated corresponding time to event, first we need to obtain the true model from original data sets. `getTrueModel` function first fit CoxBoost to the original sets for coefficients, then calculates the base cumulative survival and censoring hazard. These will all be used by `simTime`, which takes the output of `getTrueModel` to simulate the time and censoring status in simulated data sets. These will require a response variable, which we can extract from the data sets in this example.

```
> y.list <- lapply(esets, function(eset){
+     return( Surv(eset$days_to_death, eset$vital_status=="deceased") )
+   })
```

```
> true.mod <- getTrueModel(esets, y.list, 100)
```

```
> names(true.mod)
```

```
[1] "beta"  "grid"  "survH" "censH" "lp"
```

`beta` represents coefficients obtained from fitting CoxBoost to the original data sets. `lp` is linear predictor calculated by multiplying coefficients with the gene expression matrix. `survH` and `censH` are cumulative survival and censoring hazard while `grid` is the corresponding time span.

```
> simmodel <- simData(esets, 500, y.list)
> new.esets <- simTime(simmodel, true.mod)
```

`simTime` output the same form of list as input, only different in that the y.vars is updated. Both parameters of `simTime` can be constructed from scratch. Please refer to the examples sim the help manuel.

2.2.4 The driver function

In order to achieve the three-step simulation, we can complete it step-by-step in the following way:

```
> true.mod <- getTrueModel(esets, y.list, 100)
> sim.sets <- simData(esets, 500, y.list)
> sim.sets <- simTime(sim.sets, true.mod)
```

Or, we can use the driver function `simBootstrap` directly:

```
> sim.sets <- simBootstrap(esets, y.list, 500, 100)
```

This driver function is capable of skipping resampling set labels and balancing covariates. We can use its `type` and `balance.variables` the same way as we use them in `simData`. However, separating the functions allows users to use different true models. For example, we can combine all the data sets into one big set and use it to simulate one true model, then use it for every simulated independent set.

```
> y.vars=y.list
> balance.variables=c("tumorstage", "grade")
> X.list <- lapply(esets, function(eset){
+   newx <- t(exprs(eset))
+   return(newx)
+ })
> all.X <- do.call(rbind, X.list)
> cov.list <- lapply(esets, function(eset){
+   return(pData(eset)[, balance.variables])
+ })
> all.cov <- as(data.frame(do.call(rbind, cov.list)), "AnnotatedDataFrame")
> rownames(all.cov) <- colnames(t(all.X))
> all.yvars <- do.call(rbind, y.vars)
> whole_eset <- ExpressionSet(t(all.X), all.cov)
```

Then we use this big merged ExpressionSet to fit a parametric model of survival that will be used as the "true" model:

```
> lp.index <- c()
> for(i in 1:length(esets)){
+   lp.index <- c(lp.index, rep(i, length(y.vars[[i]][, 1])))
+ }
> truemod <- getTrueModel(list(whole_eset), list(all.yvars), parstep=100)
> simmodels <- simData(esets, 150, y.vars)
> beta <- grid <- survH <- censH <- lp <- list()
> for(listid in 1:length(esets)){
+   beta[[listid]] <- truemod$beta[[1]]
+   grid[[listid]] <- truemod$grid[[1]]
+   survH[[listid]] <- truemod$survH[[1]]
+   censH[[listid]] <- truemod$censH[[1]]
+   lp[[listid]] <- truemod$lp[[1]][which(lp.index==listid)]
+ }
> res <- list(beta=beta, grid=grid, survH=survH, censH=censH, lp=lp)
> simmodels <- simTime(simmodels, res)
```

Users can also build up their own true model based on the example in `simTime`, which ensures flexibility of this package.

The `y.vars` parameters also support *Surv*, *matrix* and *data.frame* objects for additional flexibility.

2.3 Simulation on SummarizedExpreriment

Simulation over *SummarizedExperiment* is similar to that over *ExpressionSets*. So we will illustrate that the *simulatorZ* is suitable for simulating from one orginal set as well. First, we use *parathysoidSE* package to obtain a *SummarizedExpreriment* object.

```
> library(parathyroidSE)
> data("parathyroidGenesSE")
```

Enclose parathyroidGenesSE in a list for compatibility with simData.

```
> sim.sets <- simData(list(parathyroidGenesSE), 100)
```

```
[1] 1
[1] "covariate: NULL"
```

```
> names(sim.sets)
```

```
[1] "obj"          "indices"      "setsID"       "y.vars"       "prob.desired"
[6] "prob.real"
```

```
> sim.sets$obj[[1]]   #The simulated SummarizedExperiment
```

```
class: SummarizedExperiment
dim: 63193 100
exptData(1): MIAME
assays(1): counts
rownames(63193): ENSG00000000003 ENSG00000000005 ... LRG_98 LRG_99
rowData metadata column names(0):
colnames: NULL
colData names(8): run experiment ... study sample
```

In this simple example the simulated object is simply a bootstrap sample (with replacement) of the original object. In the original object, "run" is unique:

```
> table(colData(parathyroidGenesSE)$run)
```

```
SRR479052 SRR479053 SRR479054 SRR479055 SRR479056 SRR479057 SRR479058 SRR479059 SRR479060
        1         1         1         1         1         1         1         1         1
SRR479061 SRR479062 SRR479063 SRR479064 SRR479065 SRR479066 SRR479067 SRR479068 SRR479069
        1         1         1         1         1         1         1         1         1
SRR479070 SRR479071 SRR479072 SRR479073 SRR479074 SRR479075 SRR479076 SRR479077 SRR479078
        1         1         1         1         1         1         1         1         1
```

But "run" is not unique in the simulated object due to the bootstrap sampling:

```
> table(colData(sim.sets$obj[[1]])$run)
```

```
SRR479052 SRR479053 SRR479054 SRR479055 SRR479056 SRR479057 SRR479058 SRR479059 SRR479060
        4         5         2         3         3         6         3         6         3
SRR479061 SRR479062 SRR479063 SRR479064 SRR479065 SRR479066 SRR479067 SRR479068 SRR479069
        1         3         4         1         2         7         5         6         4
SRR479070 SRR479071 SRR479072 SRR479073 SRR479075 SRR479076 SRR479077 SRR479078
        3         3         3         2         6         9         3         3
```

3 Training and Validation on genomic data sets

Another important feature of *simulatorZ* is to perform predictive training and validation over data sets.

3.1 Independent within study validation (Superpc)

The following example shows how to use SuperPC (Blair and Tibshirani, 2004) algorithm to train and validate on one ExpressionSet

3.1.1 create training set and large validation set

We use the first element in `esets` to do the within study validation. First we generate a training set with 450 observations:

```
> tr.size <- 450
> simmodel.tr <- simBootstrap(esets[1], y.list[1], tr.size, 100)
> tr.set <- simmodel.tr$obj.list[[1]]
> X.tr <- t(exprs(tr.set))
> y.tr <- simmodel.tr$y.vars.list[[1]]
```

Then we simulate the validation set with the same original set, so that the training and validation set are independent, yet from the same study.

```
> val.size <- 450
> simmodel.val <- simBootstrap(esets[1], y.list[1], val.size, 100)
> val.set <- simmodel.val$obj.list[[1]]
> X.val <- t(exprs(val.set))
> y.val <- simmodel.val$y.vars.list[[1]]
```

Here we use C-Index(Gnen and Heller,2005) as a validation measure. First we can use the true linear predictor to calculated the C-Index. No algorithms is supposed to generate a C-Index larger than this one.

```
> #check C-Index for true lp
> val.par <- getTrueModel(esets, y.list, 100)
> lpboot <- val.par$lp[[1]][simmodel.val$indices[[1]]]
> library(Hmisc)
> c.ind <- rcorr.cens(-lpboot, y.val)[1]
```

```
> print(c.ind)
```

```
  C Index
0.7161614
```

3.1.2 Fit Superpc

Now we fit Superpc on training set with parameter tuning step to get the optimal parameters which works best for the validation.

```
> library(superpc)
> tr.data<- data<-list(x=t(X.tr),y=y.tr[,1], censoring.status=y.tr[,2],
+                      featurenames=colnames(X.tr))
> fit.tr<-superpc.train(data=tr.data,type='survival')
> #tuning
> cv.tr<-superpc.cv(fit.tr,data=tr.data)
> n.comp<-which.max(apply(cv.tr$scor, 1, max, na.rm = TRUE))
> thresh<-cv.tr$thresholds[which.max(cv.tr$scor[n.comp, ])]
```

Then we can compute linear predictors using the optimal parameters:

```
> lp.tr<- superpc.predict(fit.tr, tr.data, tr.data, threshold=thresh,
+                         n.components=n.comp)$v.pred.1df
> boxplot(lp.tr)
```

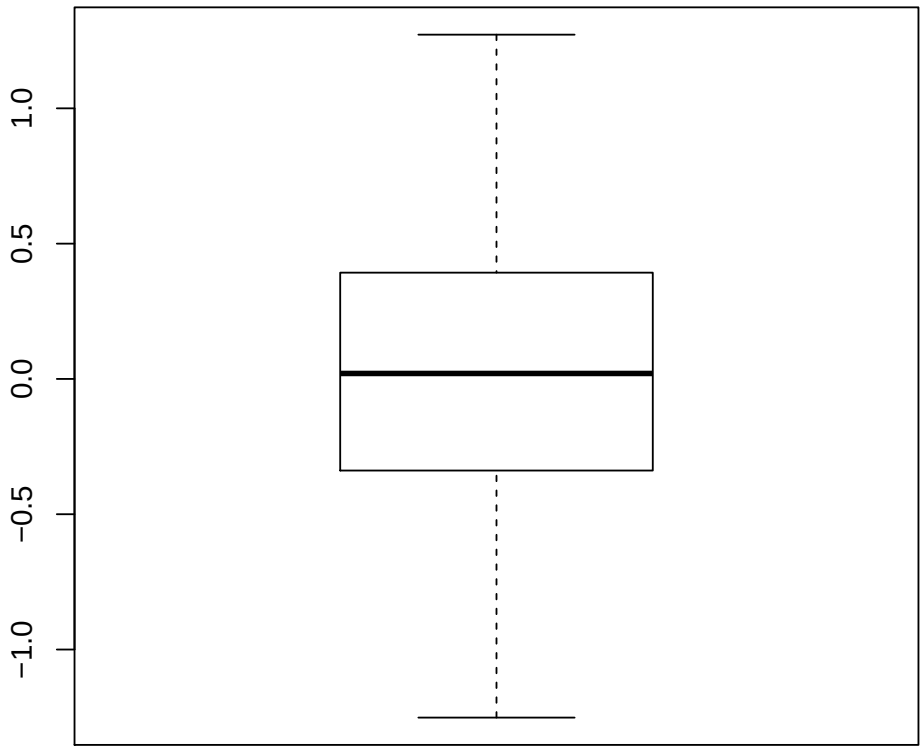

Figure 1: **Distribution of Linear Predictors Calculated by Training**

3.1.3 validation on large validation set

```
> data.val<- data<-list(x=t(X.val),y=y.val[,1], censoring.status=y.val[,2],
+                       featurenames=colnames(X.tr))
> lp.val<-superpc.predict(fit.tr, tr.data, data.val, threshold=thresh,
+                         n.components=n.comp)$v.pred.1df
```

Now we get can compute the C-Index for the validation data:

```
> print('C-Index')
```

```
[1] "C-Index"
```

```
> (c.ind<-rcorr.cens(-lp.val,y.val)[1])
```

```
  C Index
0.6360767
```

Also, the correlation of this linear predictor to the true one is another important measure for the quality of validation.

```
> print('correlation to true lp')
```

```
[1] "correlation to true lp"
```

```
> (corlps<-cor(lp.val,lpboot,method='pearson'))
```

```
[1] 0.643116
```

simulatorZ contains a function for the Mas-o-menus algorithm(Zhao et al., 2013) to support simple training and validation task. Please see the help document for `plusMinus` function.

3.2 Cross Study Validation

With enough data sets, we can perform training and validation on one set as illustrated in the last section, or do cross study validation with multiple sets. If we hope to perform cross study validation between each pair of data sets, the *simulatorZ* uses `zmatrix` to generate a matrix of C-Index. For the diagnostic elements, it performs cross validation. An example is presented as following:

```
> z <- zmatrix(obj=esets, y.vars=y.list,
+              fold=3,trainingFun=plusMinus)
```

```
> print(z)
```

```
          [,1]      [,2]      [,3]
[1,] 0.4441851 0.5117889 0.5349544
[2,] 0.5168701 0.6010359 0.4701114
[3,] 0.5649677 0.5507111 0.5202838
```

Combining the two main features of *simulatorZ*, we can first simulate a collection of independent genomic data sets, then perform the training and prediction with these data sets. The last example in this vignette will show how to complete this whole process.

```
> Z.list <- list()
> CV <- CSV <- c()
> for(b in 1:20){
+   print(paste("iteration: ", b, sep=""))
+   sim2.esets <- simBootstrap(obj=esets, n.samples=150, y.vars=y.list,
+                              parstep=100, type="two-steps")
+   Z.list[[b]] <- zmatrix(obj=sim2.esets$obj.list,
+                          y.vars=sim2.esets$y.vars.list, fold=4,
+                          trainingFun=plusMinus)
+   sum.cv <- 0
+   for(i in 1:length(esets)){
+     sum.cv <- sum.cv + Z.list[[b]][i, i]
+   }
+   CV[b] <- sum.cv / length(esets)
+   CSV[b] <- (sum(Z.list[[b]]) - sum.cv) / (length(esets)*(length(esets)-1))
+ }
```

So far we have done 20 simulations, during each of them we get a matrix of C-Index, the average of the CV and CSV validation statistics. This provides an insight of the difference between cross-study validation and the traditional cross validation.

```
> average.Z <- Z.list[[1]]
> for(i in 2:length(Z.list)){
+   average.Z <- average.Z + Z.list[[i]]
+ }
> average.Z <- average.Z / 20
> print(average.Z)
```

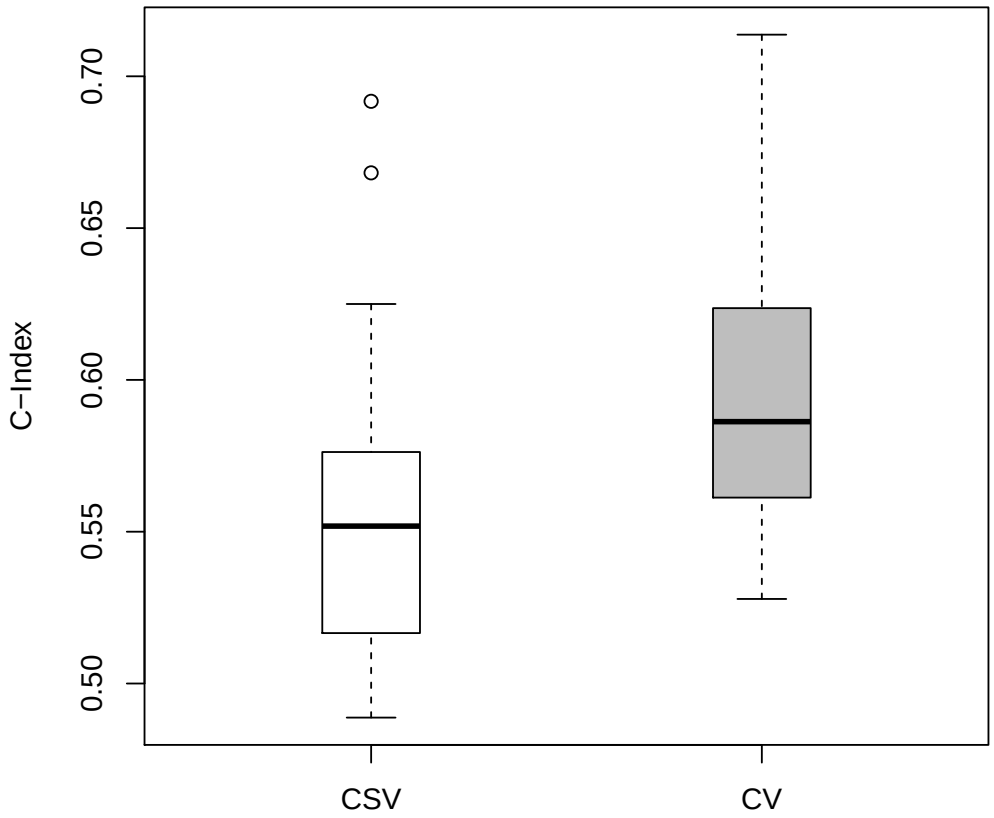

Figure 2: **Boxplots of C-Index performance with cross validation and cross study validation.** There is a dramatic drop of the CSV statistics compared to CV.

```
          [,1]      [,2]      [,3]
[1,] 0.5876904 0.5441265 0.5853218
[2,] 0.5585839 0.5957865 0.5541208
[3,] 0.5632464 0.5366644 0.6019338
```

```
> resultlist <- list(CSV=CSV, CV=CV)
> boxplot(resultlist, col=c("white", "grey"), ylab="C-Index",
+         boxwex = 0.25, xlim=c(0.5, 2.5))
```

4 Session Info

```
R version 3.1.1 Patched (2014-09-25 r66681)
Platform: x86_64-apple-darwin13.1.0 (64-bit)

locale:
```

```
[1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
 [1] grid      stats4    splines   parallel  stats     graphics  grDevices utils
 [9] datasets  methods   base

other attached packages:
 [1] superpc_1.09             Hmisc_3.14-5             Formula_1.1-2
 [4] lattice_0.20-29          parathyroidSE_1.3.1      GenomicRanges_1.18.0
 [7] GenomeInfoDb_1.2.0       IRanges_2.0.0            S4Vectors_0.4.0
[10] curatedOvarianData_1.3.3 affy_1.44.0              simulatorZ_1.0.0
[13] CoxBoost_1.4             prodlim_1.4.5            Matrix_1.1-4
[16] survival_2.37-7          Biobase_2.26.0           BiocGenerics_0.12.0

loaded via a namespace (and not attached):
 [1] BiocInstaller_1.16.0  BiocStyle_1.4.0       RColorBrewer_1.0-5
 [4] XVector_0.6.0         acepack_1.3-3.3       affyio_1.34.0
 [7] cluster_1.15.3        foreign_0.8-61        gbm_2.1
[10] latticeExtra_0.6-26   lava_1.2.6            nnet_7.3-8
[13] preprocessCore_1.28.0 rpart_4.1-8           tools_3.1.1
[16] zlibbioc_1.12.0
```

References

[1] Christoph Bernau, Markus Riester, Anne-Laure Boulesteix, Giovanni Parmigiani, Curtis Huttenhower, Levi Waldron, and Lorenzo Trippa. Cross-study validation for the assessment of prediction algorithms. *Bioinformatics*, 30(12):i105–i112, 15 June 2014. doi:10.1093/bioinformatics/btu279.

[2] Ralf Bender, Thomas Augustin, and Maria Blettner. Generating survival times to simulate cox proportional hazards models. *Stat. Med.*, 24(11):1713–1723, 15 June 2005. doi:10.1002/sim.2059.

[3] Harald Binder and Martin Schumacher. Allowing for mandatory covariates in boosting estimation of sparse high-dimensional survival models. *BMC Bioinformatics*, 9:14, 10 January 2008. doi:10.1186/1471-2105-9-14.

[4] Benjamin Frederick Ganzfried, Markus Riester, Benjamin Haibe-Kains, Thomas Risch, Svitlana Tyekucheva, Ina Jazic, Xin Victoria Wang, Mahnaz Ahmadifar, Michael J Birrer, Giovanni Parmigiani, Curtis Huttenhower, and Levi Waldron. curatedOvarianData: clinically annotated data for the ovarian cancer transcriptome. *Database*, 2013:bat013, 2 April 2013. doi:10.1093/database/bat013.