

Developing a module for model evaluation based on the parallelization techniques of snowfall

Christine Porzelius

Institut für Medizinische Biometrie und Medizinische Informatik, Universitätsklinikum Freiburg
supported by Deutsche Forschungsgemeinschaft FOR 534

May 9, 2008

Overview: Parallelization of R code

1. Identify parts of code that are
 - computationally intensive, considering computational costs of parallelization
 - parallelizable, i.e., independent of each other, for example for-loops, functions of the apply family, repeated simulations
2. Convert code using `snowfall` functions
3. Use *sfCluster* to run R code successively in
 - a) sequential,
 - b) interactive,
 - c) monitoring and
 - d) batch mode

Code conversion using snowfall functions

- Initialization:
`sfInit()`
- Export of libraries, variables, functions, data and code required for computation on nodes:
`sfLibrary(package)`
`sfExport(list)`
`sfSource(file)`
- Parallel computing on cluster:
`sfLapply(list, fun),`
`sfClusterApplyLB(list, fun),`
`sfClusterApplySR(list, fun), etc.`
- Stop cluster:
`sfStop()`

Load Balancing: Function sfClusterApplyLB

- Wrapper around snow function `clusterApplyLB`
- Suppose:
Number of available CPUs $n_{CPU_s} <$ number of jobs to execute in parallel
- Usual procedure: First n_{CPU_s} jobs are started, after finishing of **all** jobs the next n_{CPU_s} jobs, and so on
- Not efficient, if cluster nodes inhomogeneous or jobs computation times differ
- Load balancing: Immediately after one job completes, the next is placed on that node

Intermediate result saving: Function `sfClusterApplySR`

- After each block of n_{CPU_s} jobs, automatically saving of results in temporary folder possible
- Restart of computation possible after interruption of program with restored results
- Multiple calls possible, as access via specified names
- Useful in case of long computations and potential shutdown of nodes
- Not compatible with load balancing, as additional function to `snow` functions

Intermediate result saving: Function sfClusterApplySR

- After each block of n_{CPU_s} jobs, automatically saving of results in temporary folder possible
- Restart of computation possible after interruption of program with restored results
- Multiple calls possible, as access via specified names
- Useful in case of long computations and potential shutdown of nodes
- Not compatible with load balancing, as additional function to snow functions

```
res1 <- sfClusterApplySR(list1, fun1)
res2 <- sfClusterApplySR(list2, fun2, name="calc2")
# interruption during second call

res1 <- sfClusterApplySR(list1, fun1, restore=TRUE)
# complete result loaded

res2 <- sfClusterApplySR(list2, fun2, restore=TRUE,
                          name="calc2")
# start at point of interruption
```

Introduction

Situation

Resampling techniques for data based model evaluation, such as cross-validation or the bootstrap, are

- essential, though often not (properly) applied, as not trivial,
- very time consuming, especially in high-dimensional data settings,
- but predestined for parallelization.

Aim

Developing a module that provides

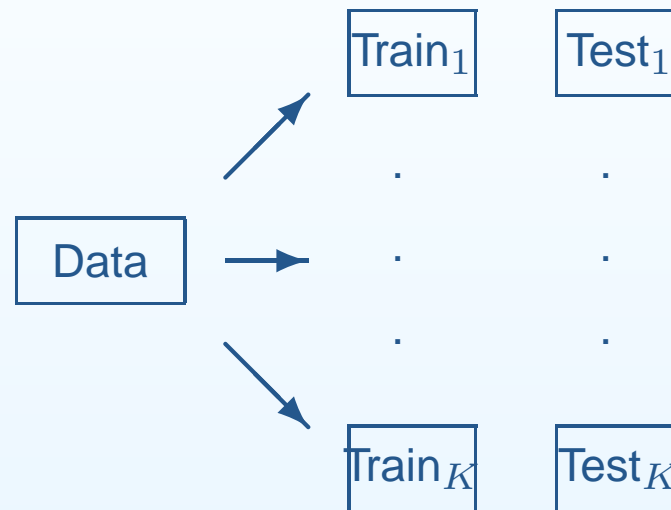
- simple parallelization of the resampling procedure
- flexibility to allow evaluation of new model fitting techniques

Result

peperr: An R package for **p**arallelized **e**stimation of **p**rediction **e**rror

Overview of procedure

1. Repeated **split** of the data set in training and test data sets



2. **Resampling procedure**, for each split consisting of the
 - a) fit of the model on the training data set
 - b) measurement of prediction error on the corresponding test set
3. **Aggregation** of the obtained prediction error measurements into one overall measure

Development: Model evaluation

<code>x</code>	data set of n observations
<code>y</code>	time-to-event or binary response
<code>new.x</code>	new data
<code>fit</code>	fitted prediction model
<code>fit.fun</code>	function to fit predictive model
<code>pred.fun</code>	function to determine prediction error

```
fit <- fit.fun(x, y)
pred.fun(fit, x.new)
```

Development: Resampling procedure

<code>x</code>	data set of n observations
<code>y</code>	time-to-event or binary response
<code>fit</code>	fitted prediction model
<code>fit.fun</code>	function to fit predictive model
<code>pred.fun</code>	function to determine prediction error
<code>n.resample</code>	number of resampled data sets
<code>x.sample</code>	data sets obtained by resampling
<code>y.sample</code>	corresponding response

Resampling:

```
for (i in 1:n.resample){  
  fit <- fit.fun(x.sample[[i]]$train, y.sample[[i]]$train)  
  pred.fun(fit, x.sample[[i]]$test)  
}
```

Development: Improved resampling

<code>x</code>	data set of n observations
<code>y</code>	time-to-event or binary response
<code>fit</code>	fitted prediction model
<code>fit.fun</code>	function to fit predictive model
<code>pred.fun</code>	function to determine prediction error
<code>n.resample</code>	number of resampled data sets
<code>x.sample</code>	data sets obtained by resampling
<code>y.sample</code>	corresponding response

Improved resampling:

```
resampling.fun <- function(i){  
  fit <- fit.fun(x.sample[[i]]$train, y.sample[[i]]$train)  
  pred.fun(fit, x.sample[[i]]$test)  
}  
  
lapply(1:n.resample, resampling.fun)
```

Development: Parallelization of the resampling procedure

Conversion for using snowfall functions:

```
sfInit()

sfExport("x.sample", "y.sample", "fit.fun", "pred.fun")

resampling.fun <- function(i){
  fit <- fit.fun(x.sample[[i]]$train, y.sample[[i]]$train)
  pred.fun(fit, x.sample[[i]]$test)
}

sfClusterApplyLB(1:n.resample, resampling.fun)

sfStop()
```

Development: Parallelization of the resampling procedure

Conversion for using snowfall functions:

```
sfInit()

sfExport("x.sample", "y.sample", "fit.fun", "pred.fun")

resampling.fun <- function(i){
  fit <- fit.fun(x.sample[[i]]$train, y.sample[[i]]$train)
  pred.fun(fit, x.sample[[i]]$test)
}

sfClusterApplyLB(1:n.resample, resampling.fun)

sfStop()
```

Problem: Exporting all resampled data sets to all nodes is not efficient.

Improvement: Export whole data set plus a list of indices obtained by function `get.indices`.

Development: Parallelization of the resampling procedure

```
resample.indices <- get.indices(n, n.resample)

sfInit()

sfExport("x", "y", "resample.indices", "fit.fun",
        "pred.fun")

resampling.fun <- function(i){
  fit <- fit.fun(x[resample.indices[[i]]$train,],
                y[resample.indices[[i]]$train,])
  pred.fun(fit, x[resample.indices[[i]]$test])
}

sfClusterApplyLB(1:n.resample, resampling.fun)

sfStop()
```

Development: Parallelization of the resampling procedure

Wrap into a function:

```
peperr <- function(x, y, resample.indices, fit.fun,
                  pred.fun){
  resample.indices <- get.indices(n, n.resample)
  sfInit()
  sfExport("x", "y", "resample.indices", "fit.fun",
          "pred.fun")
  resampling.fun <- function(i){
    fit <- fit.fun(x[resample.indices[[i]]$train,],
                  y[resample.indices[[i]]$train,])
    pred.fun(fit, x[resample.indices[[i]]$test])
  }
  sfClusterApplyLB(1:n.resample, resampling.fun)
  sfStop()
}
```

New feature: Function sfExport

- snow function `clusterExport`: Only global variables are exportable
- Improved `sfExport`: Two possibilities
 - Argument `'local=FALSE'`: Variable to export taken from global environment
 - Argument `'local=TRUE'` (default): Variable searched from local to global environment
Additionally argument `'debug'` for information from which scope variable is taken.

Development: Parallelization of the resampling procedure

Build function:

```
peperr <- function(x, y, resample.indices, fit.fun,
                  pred.fun){
  resample.indices <- get.indices(n, n.resample)
  sfInit()
  sfExport("x", "y", "resample.indices", "fit.fun",
          "pred.fun")
  resampling.fun <- function(i){
    fit <- fit.fun(x[resample.indices[[i]]$train,],
                  y[resample.indices[[i]]$train,])
    pred.fun(fit, x[resample.indices[[i]]$test])
  }
  sfClusterApplyLB(1:n.resample, resampling.fun)
  sfStop()
}
```

Problem: Functions `fit.fun` and `pred.fun` can be user defined.
What if additional libraries, functions or global variables are required?

Extraction of required libraries and functions: `extract.fun`

- Input: User-defined functions for model fit and prediction error determination
- Output: List of libraries and additional user-defined functions required for functions passed as argument to `peperr`
- Recursive implementation:
Check for each element of the body of a function, whether it is
 - library/require call → add to returned list of libraries
 - function itself → Check, whether it is
 - part of package → add to returned list of libraries
 - user defined function →
 - add to returned list of functions and
 - call `extract.fun` with this function
- Additional variables can be passed to each function by list.

Operating modes

snowfall functions work

- sequential, i.e. without compute cluster and even without snow
- with LAM/MPI cluster and
- with *sfCluster* (recommended)

⇒ Second point requires small modification of our module:

```
peperr <- function(..., parallel=FALSE, cpus=2, ...){
  if(parallel==TRUE){
    sfInit(parallel=TRUE, cpus=cpus)
  }
  else {
    sfInit()
  }
  ...
}
```

Multiple parallelization instances

- Consecutive parallelization: possible
 - Use `sfRemove/sfRemoveAll`, if necessary
- Nested parallelization: not possible/reasonable
 - No further speed-up obtainable
 - In running cluster no further subcluster can be set up

⇒ Modification of our module:

```
peperr <- function(..., noclusterstart=FALSE, ...){
  if(parallel==TRUE){
    sfInit(parallel=TRUE, cpus=cpus,
           nostart=noclusterstart)
  }
  else {
    sfInit(nostart=noclusterstart)
  }
  ...
}
```

Summary

Important points concerning parallelization with `snowfall` functions:

- Export of data
- Export of local/global variables
- Export of libraries and functions
- Compatibility to all operating modes

Overview: R package peperr

- (Potentially) parallelized prediction error estimation, relieving user of almost all issues of parallelization
- Various implemented routines, e.g.:
 - Resampling: Bootstrap, subsampling, k -fold cross-validation
 - Model fit: Boosting techniques
 - Error measures: Misclass. rate, Brier score, prediction error curve
 - Estimation techniques: Bootstrap .632+, cross-validation
- Easy integration of newly developed model fitting routines
- Additional features:
 - Incorporated model complexity selection / simultaneous fit of multiple complexity values
 - Diagnostic plots

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