SuperLearner

Eric C. Polley and **Mark J. van der Laan**
Division of Biostatistics
University of California, Berkeley

Abstract

An R package for the Super Learner ? is presented.

Keywords: prediction models, cross-validation, R.

1. Introduction

This vignette explains how to use the **SuperLearner** R package. The goal of the **SuperLearner** package is to make it easy to run the super learner algorithm ?. A strength of the super learner algorithm is the ability to combine many different prediction algorithms together and let the data decide on the optimal ensemble. R is the perfect language for such an algorithm because of the wealth of available prediction algorithms already available as packages. One problem is that prediction algorithms do not have a common syntax, so one of the goals of the SuperLearner package is to translate these prediction algorithms into a common syntax to allow for easy programming of the super learner.

2. SuperLearner

(insert details on the super learner here) ?

3. Using the package

```
> library(SuperLearner)
```

```
Super Learner
Version:  1.1-18
package created on 2010-04-14

Use SuperLearnerNews() to see changes from previous versions and latest news

Suggested packages to install for the Super Learner library:
glmnet, randomForest, class, gam, gbm, nnet, polspline,
 MASS, e1071, stepPlr, arm, party, DSA, spls, LogicReg,
```

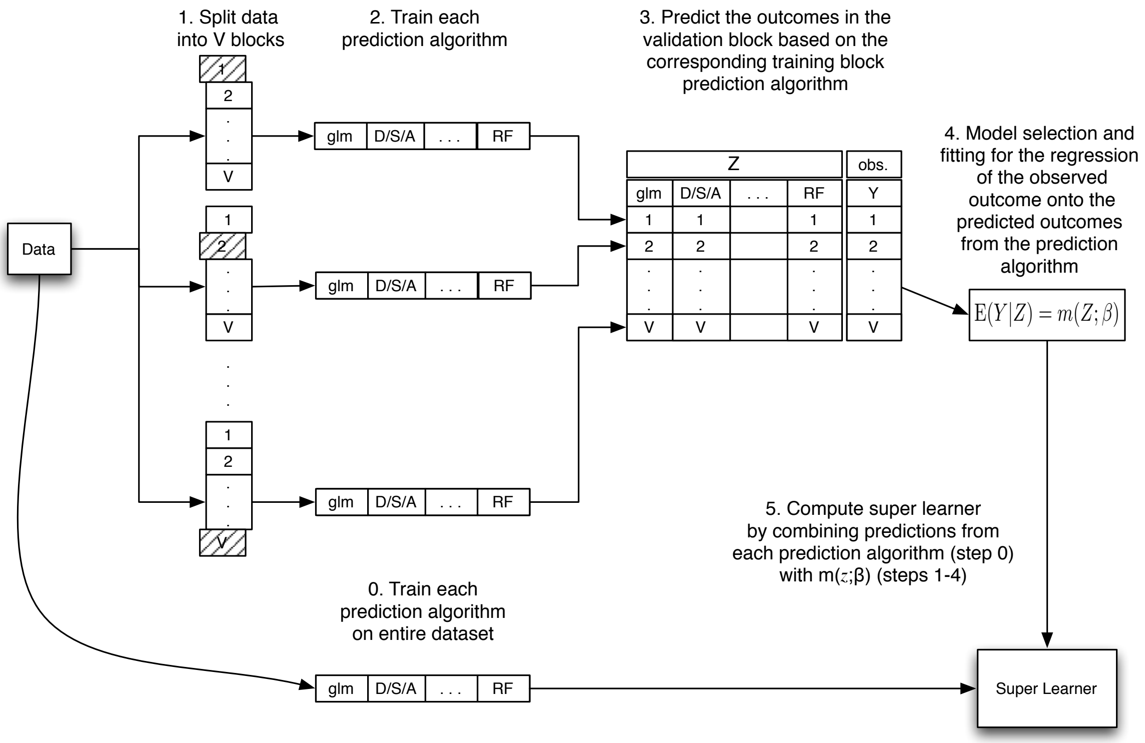
1. Split data into V blocks
2. Train each prediction algorithm
3. Predict the outcomes in the validation block based on the corresponding training block prediction algorithm
4. Model selection and fitting for the regression of the observed outcome onto the predicted outcomes from the prediction algorithm
Data
glm
D/S/A
RF
Z
obs.
Y
$E(Y|Z) = m(Z;\beta)$
5. Compute super learner by combining predictions from each prediction algorithm (step 0) with m(z;β) (steps 1-4)
0. Train each prediction algorithm on entire dataset
Super Learner

Figure 1: Super Learner algorithm diagram

```
multicore, SIS, BayesTree, ipred, mlbench, rpart, caret,
mda, earth, Hmisc
```

When you load the **SuperLearner** package a few helpful details will be printed. The version number and date are presented along with a message that the only required package, **nnls**, is also loaded. We do not know what prediction algorithms you will require for the super learner you are running in each session so the additional packages for prediction algorithms are not pre-loaded. A message with a list of suggested packages is printed and it is recommended to have these packages installed but they do not need to be loaded each time **SuperLearner** is loaded.

3.1. Creating prediction algorithms

The **SuperLearner** package includes many built-in wrapper functions for prediction algorithms. These wrappers can be directly used or easy adapted for specific examples. For a list of the wrapper functions included in the package, you can use the functions `listWrappers(what = "SL")`

```
> listWrappers(what = "SL")
```

```
All prediction algorithm wrappers in SuperLearner:
 [1] "SL.DSA"                 "SL.DSA.2"
 [3] "SL.bagTree"             "SL.bagTree.unit"
 [5] "SL.bagging"           "SL.bart"
 [7] "SL.bayesglm"          "SL.caret"
 [9] "SL.caret.rpart"       "SL.cforest"
[11] "SL.cv.spls"           "SL.earth"
[13] "SL.gam"               "SL.gam.3"
[15] "SL.gam.4"             "SL.gam.5"
[17] "SL.gbm.1"             "SL.gbm.2"
[19] "SL.glm"               "SL.glmnet"
[21] "SL.glmnet.alpha25"    "SL.glmnet.alpha50"
[23] "SL.glmnet.alpha75"    "SL.knn"
[25] "SL.knn100"            "SL.knn20"
[27] "SL.knn200"            "SL.knn30"
[29] "SL.knn40"             "SL.knn50"
[31] "SL.knn60"             "SL.knn70"
[33] "SL.knn80"             "SL.knn90"
[35] "SL.loess"             "SL.logreg"
[37] "SL.mars"              "SL.mean"
[39] "SL.nnet"              "SL.nnet.3"
[41] "SL.nnet.4"            "SL.nnet.5"
[43] "SL.polymars"          "SL.polymars.dimreduce"
[45] "SL.rF"                "SL.randomForest"
[47] "SL.ridge"             "SL.rpart"
[49] "SL.spls"              "SL.step"
[51] "SL.step.forward"      "SL.step.interaction"
[53] "SL.step.plr"          "SL.stepAIC"
[55] "SL.svm"               "SL.svm.eps"
[57] "SL.template"
```

Table 1: Details of prediction algorithm wrappers

Function	**Package**	**Tuning Parameters**	**Description**

`bagging`	**ipred**	`nbagg`
		`minsplit`
		`cp`
		`maxdepth`
			Bagging CART trees

`bagTree`	**SuperLearner** & **rpart**	`cp`
		`minsplit`
		`maxdepth`
		`ntree`
		`weights`
			Bagging CART trees

(Continued on next page)

Table 1: Details of prediction algorithm wrappers

Function	**Package**	**Tuning Parameters**	**Description**
`bart`	**BayesTree**	`ntree`	Bayesian Regression Trees
		`sigdf`	
		`sigquant`	
		`k`	
		`power`	
		`base`	
		`ndpost`	
		`nskip`	
`bayesglm`	**arm**	`prior.mean`	Bayesian glm
		`prior.scale`	
		`prior.df`	
`cforest`	**party**	`ntree`	Conditional Tree Forest
		`mtry`	
		`mincriterion`	
		`teststat`	
		`testtype`	
		`replace`	
		`fraction`	
`cv.spls`	**spls**	`K`	Sparse partial least squares
		`eta`	
`DSA`	**DSA**	`maxsize`	Deletion\Substitution\Addition
		`maxorderint`	
		`maxsumofpow`	
		`Dmove`	
		`Smove`	
		`vfold`	
`earth`	**earth**	`degree`	Adaptive Regression Splines
		`penalty`	
		`nk`	
		`thresh`	
		`minspan`	
		`newvar.penalty`	
		`fast.k`	
		`fast.beta`	
		`nfold`	
		`pmethod`	
`gam`	**gam**	`deg.gam`	Generalized additive model

(Continued on next page)

Table 1: Details of prediction algorithm wrappers

Function	**Package**	**Tuning Parameters**	**Description**

`gbm`	**gbm**	`gbm.trees`
		`interaction.depth`
		`cv.folds`
		`shrinkage`
		`n.minobsinnode`
		`bag.fraction`
		`train.fraction`
			Gradient boosting

`glm`	**stats**	–	Generalized linear model

`glmnet`	**glmnet**	`alpha`
		`lambda`
		`nlambda`
		`lambda.min`
		`dfmax`
		`type`
			Elastic Net

`knn`	**class**	`k` `use.all`	k-Nearest neighbors

`loess`	**stats**	`span`
		`family`
		`degree`
			Local polynomial regression

`logreg`	**LogicReg**	`ntrees`
		`nleaves`
		`select`
		`penalty`
		`kfold`
		`control`
			Logic regression

`mars`	**mda**	`degree`
		`nk`
		`penalty`
		`thresh`
		`prune`
		`forward.step`
			Adaptive Regression Splines

`nnet`	**nnet**	`size`
		`decay`
		`rang`
			Neural network

`polymars`	**polspline**	`maxsize`
		`gcv`
		`additive`
		`knots`
		`know.space`
			Adaptive polynomial splines

`polyclass`	**polspline**	`maxdim`
		`cv`
		`additive`
			Polychotomous regression

(Continued on next page)

Table 1: Details of prediction algorithm wrappers

Function	Package	Tuning Parameters	Description
`randomForest`	**randomForest**	`ntree`	Random Forest
		`mtry`	
		`nodesizes`	
		`sampsize`	
		`replace`	
		`maxnodes`	
`Ridge`	**MASS**	`lambda`	Ridge regression
`rpart`	**rpart**	`cp`	Regression tree
		`minsplit`	
		`xval`	
		`maxdepth`	
		`minbucket`	
`step`	**stats**	`scope`	Stepwise regression
		`scale`	
		`direction`	
		`steps`	
		`k`	
`step.plr`	**stepPlr**	`type`	Stepwise penalized logistic
		`lambda`	
		`cp`	
		`max.terms`	
`svm`	**e1071**	`type`	Support vector machine
		`kernel`	
		`nu`	
		`degree`	
		`gamma`	
		`coef0`	
		`cost`	
		`cachesize`	
		`tolerance`	
		`epsilon`	
		`cross`	

The package contains a template function for prediction algorithms:

```
> SL.template

function (Y.temp, X.temp, newX.temp, family, obsWeights, id,
    ...)
{
    if (family$family == "gaussian") {
    }
    if (family$family == "binomial") {
    }
    out <- numeric()
    fit <- vector("list", length = 0)
    foo <- list(out = out, fit = fit)
    class(foo$fit) <- c("SL.template")
    return(foo)
}
<environment: namespace:SuperLearner>
```

Argument	Description
`Y.temp`	the outcome variable
`X.temp`	the training data set (the observations used to fit the model)
`newX.temp`	the validation data set (the observations to return predictions for)
`family`	a description of the error distribution
`id`	a cluster identification
`obsWeights`	observation weights

Table 2: The allowed arguments for a prediction algorithm in `SuperLearner`

Each prediction algorithm needs to return a list with 2 elements. The first element, `out`, contains the predicted values for the observations in `newX.temp` based on the model estimated with `X.temp`. The values of `out` should be on the scale of the outcome `Y.temp`. The second element, `fit`, is a list containing all the elements of the fitted model required to predict a new observation. The names in the `fit` list should match the arguments for the predict method associated with the prediction algorithm.

3.2. Example: creating prediction wrapper

Consider the `polymars` algorithm in the **polspline** package and the general data setting:

- continuous outcome Y
- data.frame of covariates X
- data.frame of covariates newX

The first step in creating a prediction algorithm wrapper is to know how to use the function to fit a model and predict values on new observations. Reading the help documentation for `polymars` we find the functions are:

```
> fit.mars <- polymars(Y, X)
> out <- predict.polymars(fit.mars, x = as.matrix(newX))
```

These functions fit the model using a numeric Y and a data.frame X and call that object `fit.mars`. The second line returns the predicted values from the `fit.mars` object on the observations in newX that needs to be converted to a matrix before evaluation. The `out` object is a numeric vector with the predicted observations with length the number of rows in newX. The next step is to take the template code and insert the `polymars` code:

```
> My.SL.polymars <- function(Y.temp, X.temp, newX.temp, family, ...) {
+   if(family$family=="gaussian"){
+     fit.mars <- polymars(Y.temp, X.temp)
+     out <- predict.polymars(fit.mars, x = as.matrix(newX.temp))
+   }
+   if(family$family=="binomial"){
+     # insert estimation function
+   }
+   ... # details below
+ }
```

Insert the `polymars` code into the gaussian family block. Also note that the name of the function needs to be unique. The wrapper name must not conflict with the name of a wrapper contained in the **SuperLearner** package otherwise the function in the package will take precedence over the new version. If only writing the wrapper for a specific problem the block for the binomial family could be ignored (or add the error `stop("only gaussian allowed")`), but if you want to save the new wrapper for future project that may include binomial outcomes we could look up the `polymars` code for a binomial family and the same data structure,

```
> fit.mars <- polyclass(Y, X, cv = 10)
> out <- ppolyclass(cov = newX, fit = fit.mars)[, 2]
```

and add these to the wrapper.

```
> My.SL.polymars <- function(Y.temp, X.temp,
+   newX.temp, family, ...) {
+   if(family$family=="gaussian"){
+     fit.mars <- polymars(Y.temp, X.temp)
+     out <- predict.polymars(fit.mars, x = as.matrix(newX.temp))
+   }
+   if(family$family=="binomial"){
+     fit.mars <- polyclass(Y.temp, X.temp, cv = 5)
+     out <- ppolyclass(cov = newX.temp, fit = fit.mars)[, 2]
+   }
+   ... # details below
+ }
```

All wrappers need to return two values:

1. `out`: predicted `Y` values for rows in `newX`
2. `fit`: a list with everything needed to use `predict` method

The out object is already in the wrapper, but need to figure out the `fit` object. In the `polymars` example

1. For the gaussian case, `predict()` needs: `object = fit.mars`
2. For the binomial case, `predict()` needs: `fit = fit.mars`

Note that SuperLearner does not use the fit list. If you do not plan to use the function predict.SuperLearner you can leave the fit object as: fit <- vector("list", length = 0). This object has been written to be very flexible since various prediction algorithms require a diverse set of objects for predicting with new observations. Now we add the fit code to the wrapper:

```
> My.SL.polymars  <- function(Y.temp, X.temp, newX.temp, family, ...) {
+   if(family$family=="gaussian"){
+     fit.mars <- polymars(Y.temp, X.temp)
+     out <- predict.polymars(fit.mars, x = as.matrix(newX.temp))
+     fit <- list(object = fit.mars)
+   }
+   if(family$family=="binomial"){
+     fit.mars <- polyclass(Y.temp, X.temp, cv = 5)
+     out <- ppolyclass(cov = newX.temp, fit = fit.mars)[, 2]
+     fit <- list(fit = fit.mars)
+   }
+   ... # details below
+ }
```

The final step is putting everything together into a list object. The list must have 2 elements and the names *must* be out and fit. One can also assign a class to the fit list. This will be used to look up the correct predict method. The SuperLearner is using S3 methods here. The class assignment is only important if using predict.SuperLearner afterwards. The final wrapper is:

```
> My.SL.polymars  <- function(Y.temp, X.temp, newX.temp, family, ...) {
+   if(family$family=="gaussian"){
+     fit.mars <- polymars(Y.temp, X.temp)
+     out <- predict.polymars(fit.mars, x = as.matrix(newX.temp))
+     fit <- list(object = fit.mars)
+   }
+   if(family$family=="binomial"){
+     fit.mars <- polyclass(Y.temp, X.temp, cv = 5)
+     out <- ppolyclass(cov = newX.temp, fit = fit.mars)[, 2]
+     fit <- list(fit = fit.mars)
+   }
+   foo <- list(out = out, fit = fit)
+   class(foo$fit) <- c("SL.polymars")
+   return(foo)
+ }
```

This wrapper should match the SL.polymars wrapper in the package. To complete the example, the correct predict method for this new wrapper is:

```
> predict.SL.polymars <- function (object, newdata, family, ...) {
+   if (family$family == "gaussian") {
+     out <- predict.polymars(object = object$object, x = as.matrix(newdata))
+   }
+   if (family$family == "binomial") {
+     out <- ppolyclass(cov = newdata, fit = object$fit)[, 2]
+   }
+   return(out)
+ }
```

4. Creating screening algorithm wrappers

```
> listWrappers(what = "screen")

All screening algorithm wrappers in SuperLearner:
[1] "All"
[1] "screen.SIS"          "screen.corP"
[3] "screen.corRank"      "screen.glmP"
[5] "screen.glmRank"      "screen.glmnet"
[7] "screen.randomForest" "screen.template"

> screen.template

function (Y.temp, X.temp, family, obsWeights, id, ...)
{
    if (family$family == "gaussian") {
    }
    if (family$family == "binomial") {
    }
    whichVariable <- rep(TRUE, ncol(X.temp))
    return(whichVariable)
}
<environment: namespace:SuperLearner>
```

Argument	Description
`Y.temp`	the outcome variable
`X.temp`	the training data set (the observations used to fit the model)
`family`	a description of the error distribution
`id`	a cluster identification
`obsWeights`	observation weights

Table 3: The allowed arguments for a screening algorithm in `SuperLearner`

Each screening algorithm needs to return a logical vector of length equal to the number of variables in `X.temp`. The value `TRUE` indicates the variable should be included in the prediction algorithms for that screening, while `FALSE` indicates the variable should not be included (screened out) of the prediction algorithm.

5. How to set up the SuperLearner library

Before running the `SuperLearner`, the library of prediction algorithms needs to be set up. A collection of predefined algorithms and screening algorithms are included in the package. The previous sections discussed how to create your own algorithm functions for the `SuperLearner`. With all the functions ready, there are two ways to specify the library

1. A character vector
2. A list of character vectors

If no screening algorithms are to be used, the character vector approach is easiest. The function names are specified by strings separated by commas. For example,

```
> library = c("SL.glm", "SL.glmnet", "SL.randomForest")
```

When there are screening algorithms, the users needs to match the prediction algorithms with the screening algorithms. The library will be a list, and each element a character vector. The first element in the vector is the prediction algorithm, followed by all the screening algorithms matched with that prediction algorithm. For example,

```
> library = list(c("SL.glm", "screen.glmP"), c("SL.glmnet"),
+   c("SL.randomForest", "All", "screen.glmP", "screen.glmRank"))
```

One special included screening algorithms is the "All" function. "All" is a function to fit the matching prediction algorithm on the entire dataset. When no screening algorithm is specified, the "All" function is attached to the prediction algorithm. In the example above, the "SL.glmnet" algorithm is a singleton, but the `SuperLearner` will interpret this as `c("SL.glmnet", "All")`. If at least one screening algorithm is provided, then the "All" is not attached to the prediction algorithm. In the example above, "SL.glm" will only be run on the "screen.glm" subset of the variables and not on the entire set of variables. If the users wants to run the prediction algorithm on the entire dataset and on a screened subset, then the user needs to add the special function "All" to the corresponding vector. In the example above, "SL.randomForest" will be run on the entire dataset ("All") and on the screened subsets from the screening algorithms "screen.glmP" and "screen.glmRank".

6. Example: Boston Housing Data

The Boston Housing Data in the `mlbench` package contains information on housing prices from 506 census tracts around Boston from the 1970 census. The original data description can be found in Harrison and Rubinfeld (1979) a corrections in Gilley and Pace (1996). We will build a model to predict the median value of owner-occupied homes based on the covariates supplied (for example, per capita crime rate, average number of rooms per dwelling and average age of homes). First we load the data and examine the variables included:

```
> library(mlbench)
> data(BostonHousing2)
```

With the data loaded, we need to change the `chas` variable from a factor to a numeric and then remove the variables we will not be using in the analysis.

```
> BostonHousing2$chas <- as.numeric(BostonHousing2$chas=="1")
> DATA <- BostonHousing2[, c("cmedv", "crim", "zn", "indus", "chas", "nox",
+   "rm", "age", "dis", "rad", "tax", "ptratio", "b", "lstat")]
```

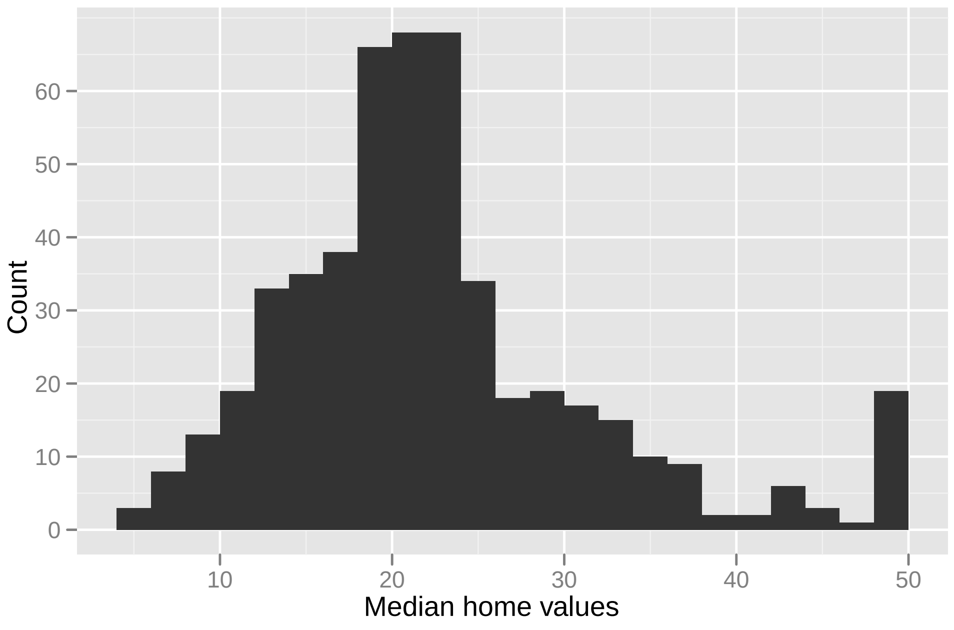

Figure 2: Histogram of median home values (in 1000 USD)

7. SuperLearner

```
> library(SuperLearner)
```

Before running the Super Learner we first need to set up the library of prediction algorithms. As we saw above, the **SuperLearner** package includes many built-in prediction algorithms. But the list of algorithms may not be sufficient for a given problem. Expert contextual knowledge may recommend new algorithms to consider or different values for the tuning parameters in the existing algorithms.

For example, the `SL.gam` function has the tuning parameter `deg.gam = 2`. If we wanted to also consider the `gam` model with degree 3 we do not need to write a new wrapper function but can easily use the current function and change the argument. For example:

```
> SL.gam.3 <- function(..., deg.gam = 3){
+         SL.gam(..., deg.gam = deg.gam)
+ }
```

`SL.gam.3` is a simple function calling the `SL.gam` function but changing the argument `deg.gam` to the desired value for the tuning parameter. The "…" will simply pass all other arguments from `SL.gam.3` into `SL.gam` since these should be the same. This mechanism of writing functions of the wrapper functions allows for the easy adaptation of the currently available prediction algorithms to the current prediction problem and the desired range for tuning parameters.

```
> SL.library <- c("SL.gam",
+          "SL.gam.3",
+          "SL.gam.4",
+          "SL.gam.5",
+          "SL.gbm.1",
+          "SL.gbm.2",
+          "SL.glm",
+          "SL.glmnet",
+          "SL.glmnet.alpha25",
+          "SL.glmnet.alpha50",
+          "SL.glmnet.alpha75",
+          "SL.polymars",
+          "SL.randomForest",
+          "SL.ridge",
+          "SL.svm",
+          "SL.bayesglm",
+          "SL.step",
+          "SL.step.interaction",
+          "SL.bart")
```

With the library setup, we can now run the super learner. We need to specify the data frame of covariates X and the outcome Y. We can set the number of folds for the cross-validation step to estimate the weights for each algorithm. The default value is 20, but we will use 10 here for computational reason. The verbose statement specifies if you want a detailed report of the progress while the super learner runs. Set verbose to `TRUE` for a detailed output. Specify the library of prediction algorithms with the `SL.library` argument. The `shuffle` argument while randomize the rows in the data frame before splitting the data into V folds.

```
> fitSL <- SuperLearner(Y=log(DATA$cmedv),
+          X=subset(DATA, select = -c(cmedv)),
+          V=10,
+          verbose=FALSE,
+          SL.library=SL.library,
+          shuffle = TRUE,
+          family = gaussian()
+ )

> fitSL

Call:
SuperLearner(Y = log(DATA$cmedv), X = subset(DATA, select = -c(cmedv)),
    SL.library = SL.library, V = 10, shuffle = TRUE, verbose = FALSE,
    family = gaussian())
```

```
                              Risk       Coef
SL.gam_All              0.03513343 0.00000000
SL.gam.3_All            0.03364400 0.00000000
SL.gam.4_All            0.03262047 0.00000000
SL.gam.5_All            0.03181313 0.00000000
SL.gbm.1_All            0.02918967 0.00000000
SL.gbm.2_All            0.02445375 0.00000000
SL.glm_All              0.03662357 0.00000000
SL.glmnet_All           0.03669622 0.00000000
SL.glmnet.alpha25_All   0.03676771 0.00000000
SL.glmnet.alpha50_All   0.03688589 0.00000000
SL.glmnet.alpha75_All   0.03661601 0.00000000
SL.polymars_All         0.03393655 0.00000000
SL.randomForest_All     0.02152117 0.23500781
SL.ridge_All            0.03663368 0.00000000
SL.svm_All              0.02660668 0.01244019
SL.bayesglm_All         0.03662259 0.00000000
SL.step_All             0.03649106 0.00000000
SL.step.interaction_All 0.02373183 0.28336464
SL.bart_All             0.01927439 0.46918736
```

8. Example: ALL Data

The super learner can also be used with a binary outcome.

```
> # install Bioconductor default packages and ALL package
> # source("http://bioconductor.org/biocLite.R")
> # biocLite()
> # biocLite("ALL")
> library(ALL)
> library(genefilter)
> data(ALL)
```

With the data from the **ALL** package, some filtering is still required before running the analysis. The steps below follow The chapter on supervised learning in **?**.

```
> # restrict to only the NEG and BCR/ABL outcomes
> bcell <- grep("^B", as.character(ALL$BT))
> moltyp <- which(as.character(ALL$mol.biol) %in% c("NEG", "BCR/ABL"))
> ALL_bcrneg <- ALL[, intersect(bcell, moltyp)]
> ALL_bcrneg$mol.biol <- factor(ALL_bcrneg$mol.biol)  #drops unused levels
> #
> # filter features
> ALLfilt_bcrneg <- nsFilter(ALL_bcrneg, var.cutoff = 0.75)$eset
> #
> # standardize the features
> rowIQRs <- function(eSet) {
+         numSamp <- ncol(eSet)
+         lowQ <- rowQ(eSet, floor(0.25 * numSamp))
+         upQ <- rowQ(eSet, ceiling(0.75 * numSamp))
+         upQ - lowQ
+ }
> standardize <- function(x) {
+         (x - rowMedians(x)) / rowIQRs(x)
+ }
> exprs(ALLfilt_bcrneg) <- standardize(exprs(ALLfilt_bcrneg))
> #
> # convert to numeric matrix for the SuperLearner
> Y <- as.numeric(ALLfilt_bcrneg$mol.biol == "BCR/ABL")
> X <- t(exprs(ALLfilt_bcrneg))
```

With the data ready, the next step is to put together the library. We consider the k-nearest neighbors, elastic net, and random forest algorithms with various levels of the tuning parameters. `knn` has a few built-in functions for various levels of k and the elastic net function, `glmnet`, and includes a few values for α. For `randomForest`, we consider a few values for `mtry` and `nodesize`. Since we are adjusting two tuning parameters, a grid is created with all possible pairwise combinations of the values for `mtry` and `nodesize`. The code below is similar to the code used above to create the new `gam` models with high degrees of freedom.

```
> tuneGrid <- expand.grid(mtry = c(500, 1000, 2200), nodesize = c(1, 5, 10))
> for(mm in seq(nrow(tuneGrid))) {
+         eval(parse(file = "", text = paste("SL.randomForest.", mm,
+         "<- function(..., mtry = ", tuneGrid[mm, 1], ", nodesize = ",
+         tuneGrid[mm, 2], ") { SL.randomForest(..., mtry = mtry,
+         nodesize = nodesize) }", sep = "")))
+ }
> SL.library <- c("SL.knn",
+                 "SL.knn20",
+                 "SL.knn30",
+                 "SL.knn40",
+                 "SL.knn50",
+                 "SL.randomForest",
+                 "SL.glmnet",
+                 "SL.glmnet.alpha25",
+                 "SL.glmnet.alpha50",
+                 "SL.glmnet.alpha75",
+                 "SL.mean",
+                 paste("SL.randomForest.", seq(nrow(tuneGrid)), sep = ""))
```

Since the outcome is binary, the family argument must be set to `binomial()` in the `SuperLearner`.

```
> fitSL <- SuperLearner(Y = Y, X = X, V = 10, SL.library = SL.library,
+          family = binomial(), verbose = FALSE,
+          method = "NNLS", stratifyCV = TRUE)
> fitSL

Call:
SuperLearner(Y = Y, X = X, SL.library = SL.library, V = 10,
    verbose = FALSE, family = binomial(), method = "NNLS",
    stratifyCV = TRUE)


                            Risk       Coef
SL.knn_All           0.20430380 0.00000000
SL.knn20_All         0.22313291 0.00000000
SL.knn30_All         0.22277887 0.00000000
SL.knn40_All         0.23432812 0.00000000
SL.knn50_All         0.23915886 0.00000000
SL.randomForest_All  0.12618567 0.00000000
SL.glmnet_All        0.08414461 0.82257005
SL.glmnet.alpha25_All 0.09834590 0.00000000
SL.glmnet.alpha50_All 0.09359018 0.00000000
SL.glmnet.alpha75_All 0.08521177 0.13392874
SL.mean_All          0.24924069 0.00000000
SL.randomForest.1_All 0.12792559 0.00000000
SL.randomForest.2_All 0.12235342 0.00000000
SL.randomForest.3_All 0.11827363 0.00000000
SL.randomForest.4_All 0.12984818 0.00000000
SL.randomForest.5_All 0.12063995 0.00000000
SL.randomForest.6_All 0.11875061 0.00000000
SL.randomForest.7_All 0.12984432 0.00000000
SL.randomForest.8_All 0.12098649 0.00000000
SL.randomForest.9_All 0.11613709 0.04350121
```

Honest V-fold cross-validated risk estimates can be obtained using the `CV.SuperLearner`. The results can be found in table **??**.

9. Computing Environment

- R version 2.11.0 Patched (2010-04-27 r51837), `x86_64-apple-darwin9.8.0`
- Base packages: base, datasets, grDevices, graphics, grid, methods, splines, stats, utils
- Other packages: ALL 1.4.7, AnnotationDbi 1.10.0, BayesTree 0.3-1, Biobase 2.8.0, DBI 0.2-5, MASS 7.3-5, Matrix 0.999375-38, R2WinBUGS 2.1-16, RSQLite 0.8-4, SuperLearner 1.1-18, abind 1.1-0, akima 0.5-4, arm 1.3-02, car 1.2-16, class 7.3-2, coda 0.13-5, digest 0.4.2, e1071 1.5-24, foreign 0.8-40, gam 1.03, gbm 1.6-3,

Table 4: 20-fold cross-validated risk estimates for the super learner, the discrete super learner and each algorithm in the library

Algorithm	subset	Risk	SE	Min	Max
SuperLearner	–	0.194	0.017	0.103	0.309
Discrete SL	–	0.238	0.024	0.127	0.415
SL.knn(10)	All	0.249	0.020	0.144	0.532
SL.knn(10)	clinical	0.239	0.019	0.138	0.496
SL.knn(10)	cor ($p < 0.1$)	0.262	0.023	0.095	0.443
SL.knn(10)	cor ($p < 0.01$)	0.224	0.020	0.088	0.365
SL.knn(10)	glmnet	0.219	0.028	0.007	0.465
SL.knn(20)	All	0.242	0.013	0.171	0.397
SL.knn(20)	clinical	0.236	0.012	0.154	0.382
SL.knn(20)	cor ($p < 0.1$)	0.233	0.017	0.108	0.342
SL.knn(20)	cor ($p < 0.01$)	0.206	0.018	0.121	0.321
SL.knn(20)	glmnet	0.217	0.026	0.018	0.405
SL.knn(30)	All	0.239	0.013	0.171	0.396
SL.knn(30)	clinical	0.236	0.012	0.169	0.386
SL.knn(30)	cor ($p < 0.1$)	0.232	0.014	0.143	0.319
SL.knn(30)	cor ($p < 0.01$)	0.215	0.017	0.136	0.346
SL.knn(30)	glmnet	0.210	0.023	0.039	0.402
SL.knn(40)	All	0.240	0.011	0.182	0.331
SL.knn(40)	clinical	0.238	0.010	0.179	0.319
SL.knn(40)	cor ($p < 0.1$)	0.236	0.012	0.166	0.316
SL.knn(40)	cor ($p < 0.01$)	0.219	0.015	0.154	0.309
SL.knn(40)	glmnet	0.211	0.021	0.060	0.346
SL.glmnet($\alpha = 1.0$)	corRank.50	0.229	0.029	0.078	0.445
SL.glmnet($\alpha = 1.0$)	corRank.20	0.208	0.026	0.048	0.424
SL.glmnet($\alpha = 0.75$)	corRank.50	0.221	0.027	0.077	0.420
SL.glmnet($\alpha = 0.75$)	corRank.20	0.209	0.026	0.046	0.421
SL.glmnet($\alpha = 0.50$)	corRank.50	0.226	0.027	0.077	0.426
SL.glmnet($\alpha = 0.50$)	corRank.20	0.211	0.026	0.059	0.419
SL.glmnet($\alpha = 0.25$)	corRank.50	0.229	0.027	0.084	0.424
SL.glmnet($\alpha = 0.25$)	corRank.20	0.216	0.025	0.072	0.406
SL.randomForest	clinical	0.198	0.019	0.098	0.391
SL.randomForest	cor ($p < 0.01$)	0.204	0.018	0.101	0.341
SL.randomForest	glmnet	0.220	0.025	0.072	0.378
SL.bagging	clinical	0.207	0.016	0.108	0.408
SL.bagging	cor ($p < 0.01$)	0.205	0.018	0.107	0.353
SL.bagging	glmnet	0.206	0.022	0.077	0.388
SL.bart	clinical	0.202	0.018	0.109	0.365
SL.bart	cor ($p < 0.01$)	0.210	0.021	0.092	0.376
SL.bart	glmnet	0.220	0.028	0.043	0.423
SL.mean	All	0.250	0.003	0.246	0.251

genefilter 1.30.0, ggplot2 0.8.7, glmnet 1.3, hgu95av2.db 2.4.1, lattice 0.18-5, lme4 0.999375-33, mlbench 2.0-0, nnet 7.3-1, nnls 1.3, org.Hs.eg.db 2.4.1, plyr 0.1.9, polspline 1.1.4, proto 0.3-8, quadprog 1.5-3, randomForest 4.5-34, reshape 0.8.3, survival 2.35-8

- Loaded via a namespace (and not attached): annotate 1.26.0, nlme 3.1-96, tools 2.11.0, xtable 1.5-6

Affiliation:

Eric Polley, Mark van der Laan
Division of Biostatistics
University of California, Berkeley
E-mail: `ecpolley@berkeley.edu`, `laan@berkeley.edu`
URL: `http://www.stat.berkeley.edu/~ecpolley/`
`http://www.stat.berkeley.edu/~laan/`