Super Learner for Prediction (Part 2)

David Whitney (based on materials by Karla Diaz-Ordaz and Jamie Burns)

8 Feb 2022

Introduction

In this practical, we'll evaluate the performance of the Super Learner using cross-validation (CV SL). If you have extra time, there is an optional example using a binary outcome. You can also experiment with different libraries, but will find that training times for CV SL are rather long.

We continue to use the NHANES data, which we process exactly as in the first practical:

```r
require(NHANES)
data("NHANES")
df <- force(NHANESraw)
df <- df %>% mutate(Smoke = ifelse(SmokeNow == "Yes", "Current",
                                   ifelse(Smoke100 == "Yes", "Ex", "Never")))%>%
  mutate(Smoke = ifelse(is.na(Smoke), "Never", Smoke)) %>%
  mutate(Smoke = factor(Smoke))

df <- df %>% dplyr::select(one_of(
  "BPSysAve","BMI", "Age", "SleepHrsNight", "PhysActive","Smoke",
  "Gender", "Race1", "Poverty", "Diabetes", "TotChol"))
df <- df[complete.cases(df),]
set.seed(101)
train_obs <- sample(nrow(df), size = nrow(df)*0.5)
df_train <- df[train_obs, ]
df_holdout <- df[-train_obs, ]
y_train <- df$BPSysAve[train_obs]
y_holdout <- df$BPSysAve[-train_obs]
X <- df %>%dplyr::select(-one_of("BPSysAve") )

# Also divide our design matrix into training and testing sets
x_train <- X[train_obs, ]
x_holdout <- X[-train_obs, ]
```

Fit ensemble with external cross-validation

We want an estimate of the performance of the ensemble itself to check the ensemble weights are successful in improving over the best single algorithm.

For this, we use an "external" layer of cross-validation, also called nested cross-validation. We generate a separate holdout sample that we don't use to fit the SuperLearner, which allows it to be a good estimate of the SuperLearner's performance on unseen data.

Typically we would run 10-fold external cross-validation, but even 5-fold is reasonable.

This also allows us to calculate standard errors on the performance of the individual algorithms and can compare them to the SuperLearner.

We will also want to keep track of the running time. Timings are not pre-programmed for the `CV.SuperLearner` function. We wrap the call to `CV.SuperLearner()` inside `system.time()`.

```r
set.seed(101)
my.library.2 <- c("SL.glm","SL.glm.interaction","SL.glmnet", "SL.ranger")
# my.library.2 <- c("SL.glm","SL.glm.interaction","SL.glmnet", "SL.ranger","SL.gbm")

system.time({
  cv_sl <- CV.SuperLearner(Y = y_train, X = x_train, family = gaussian(),
                          # For a real analysis we would use V = 10.
                          V = 5,
                          SL.library = my.library.2)
})
```

```
##    user  system elapsed
##  85.501   1.435  83.952
```

The second library can be uncommented instead, but note that the run time will be somewhat longer. The SL using smaller library (without `SL.gbm`) took about 85 seconds on my computer. This time grows to

We run summary on the `cv_sl` object rather than simply printing the object.

```r
summary(cv_sl)
```

```
##
## Call:
## CV.SuperLearner(Y = y_train, X = x_train, V = 5, family = gaussian(),
##     SL.library = my.library.2)
##
## Risk is based on: Mean Squared Error
##
## All risk estimates are based on V =  5
##
##               Algorithm    Ave     se    Min    Max
##           Super Learner 231.41 6.5045 215.41 251.83
##             Discrete SL 234.70 6.5344 219.51 254.37
##              SL.glm_All 237.06 6.6151 221.80 257.03
##  SL.glm.interaction_All 234.70 6.5344 219.51 254.37
##           SL.glmnet_All 236.86 6.6250 222.04 256.81
##           SL.ranger_All 237.72 6.6229 220.14 260.83
```

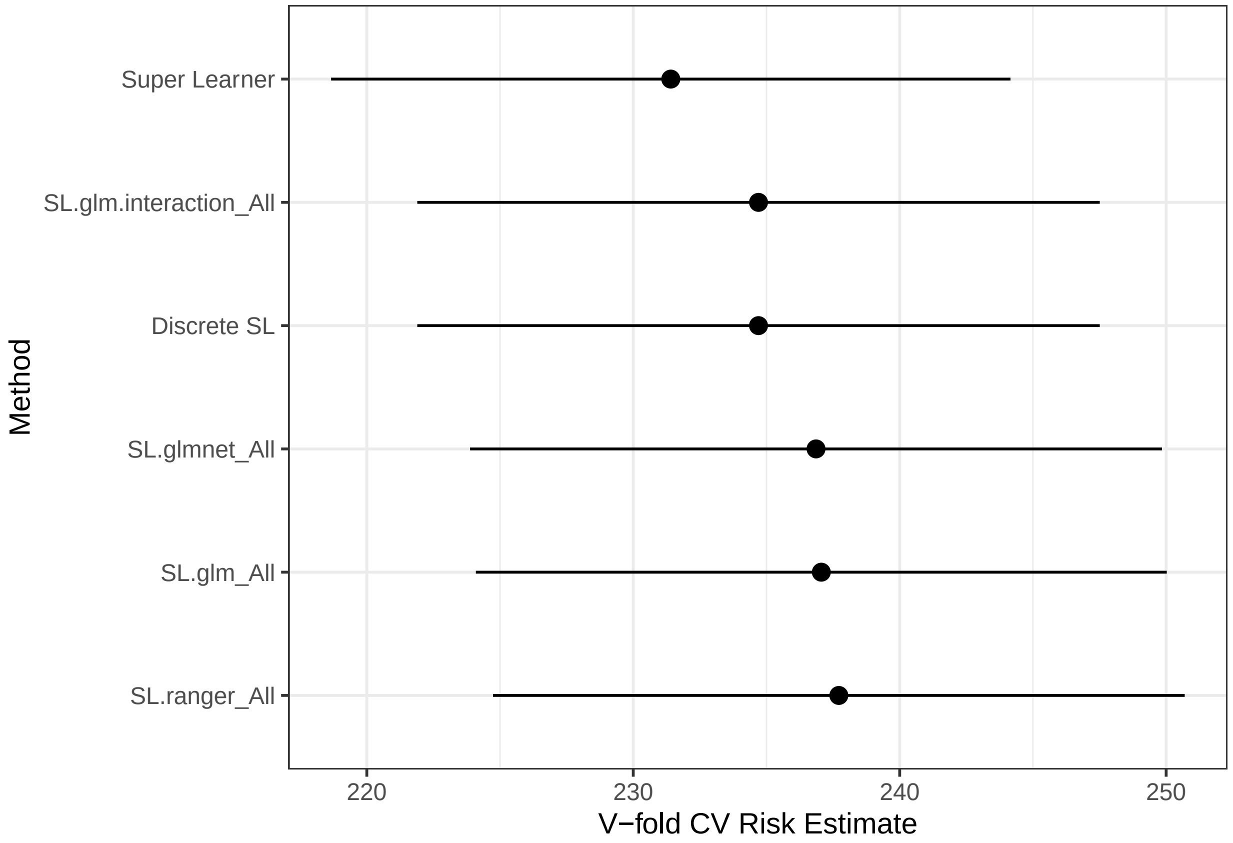
Note that the Risk is based on Mean Squared Error.

We review the distribution of the best single learner (Discrete SL) as external CV folds. Plot the performance with 95% CIs.

```r
table(simplify2array(cv_sl$whichDiscreteSL))
```

```
##
## SL.glm.interaction_All
##                      5
```

```r
plot(cv_sl) + theme_bw()
```



```r
# Save plot to a file.
ggsave("CV_SuperLearner.png")
```

We see two SuperLearner results: "Super Learner" and "Discrete SL".

- "Discrete SL" chooses the best single learner.
- "Super Learner" takes a weighted average of the learners.

Based on the outer cross-validation, we see that the `SuperLearner` is statistically tying with the best algorithm (which is `gbm`, if included). We know that asymptotically, the SL will always perform as good as the best algorithm included in the library due to the oracle property, but it is good to see it empirically in a finite sample.

Tuning hyperparameters

So far, we have been using the default hyperparameters that are pre-programmed in the SL Customizing a hyperparameter is equivalent to defining a new learner, which uses the same algorithm, but with different tuning parameters.

There are two ways of make a new learner in the SuperLearner syntax:

- Write your own function, or
- Use `create.Learner()`.

We begin by making a random forest "wrapper" function that fits more trees, i.e. the tuning parameter controlling the number of trees is different than that from the standard implementation. Increasing the number of trees may reduce error and can't hurt over fitting. Let us recall the default arguments for the wrapper function for random forests using the ranger package. We type

```
SL.ranger
```

```
## function (Y, X, newX, family, obsWeights, num.trees = 500, mtry = floor(sqrt(ncol(X))),
##     write.forest = TRUE, probability = family$family == "binomial",
##     min.node.size = ifelse(family$family == "gaussian", 5, 1),
##     replace = TRUE, sample.fraction = ifelse(replace, 1, 0.632),
##     num.threads = 1, verbose = T, ...)
## {
##     .SL.require("ranger")
##     if (family$family == "binomial") {
##         Y = as.factor(Y)
##     }
##     if (is.matrix(X)) {
##         X = data.frame(X)
##     }
##     fit <- ranger::ranger(`_Y` ~ ., data = cbind(`_Y` = Y, X),
##         num.trees = num.trees, mtry = mtry, min.node.size = min.node.size,
##         replace = replace, sample.fraction = sample.fraction,
##         case.weights = obsWeights, write.forest = write.forest,
##         probability = probability, num.threads = num.threads,
##         verbose = verbose)
##     pred <- predict(fit, data = newX)$predictions
##     if (family$family == "binomial") {
##         pred = pred[, "1"]
##     }
##     fit <- list(object = fit, verbose = verbose)
##     class(fit) <- c("SL.ranger")
##     out <- list(pred = pred, fit = fit)
##     return(out)
## }
## <bytecode: 0x7f9c1ad862c8>
## <environment: namespace:SuperLearner>
```

Now, we customize the hyper parameters using `create.Learner`. First, we create a list of tuning parameters, over which we want to create a set of new RF learners.

```
tune.ranger = list(num.trees = c(200,1000),
                   mtry = c(3,6)) #number of variables to split at in each node
```

We now create new ranger learners, with each of these combinations. We also shorten the name prefix (so that the plots later on are more readable).

```
ranger.learners = create.Learner("SL.ranger",
                                 tune = tune.ranger,
                                 detailed_names = T,
                                 name_prefix = "Rg")
```

Let's check how many learners we added (number of possible configurations)

```
length(ranger.learners$names)
```

```
## [1] 4
```

We now repeat the process for boosting. We use two different packages, `gbm` and `xgboost` (which runs much faster).

Recall that you can use type `SL.xgboost` to inspect the defaults. For `xgboost` the SL implementation has `ntrees = 1000`, interaction depth `max_depth = 4`, and learning rate `shrinkage = 0.1`.

We let the hyper-parameters take 2 values

```
#### tuning for gbm ####
tune = list(ntrees = c(100,500),
            interaction.depth = c(2,4),
            shrinkage = c(0.05))

gb.learners = create.Learner("SL.gbm",tune = tune, detailed_names = T, name_prefix = "gb")

xgb.learners = create.Learner("SL.xgboost",tune = tune, detailed_names = T, name_prefix = "gb")

length(xgb.learners$names)
```

```
## [1] 4
```

For the sake of finishing in good time, we selected only 4 configurations, though you may want to also change the learning rate.

Begin by adding to the previous library the xgboost learners with diff tuning parameters. Then we fit the `CV.SuperLearner` with this enlarged library.

We use `V = 3` to save computation time; for a real analysis use `V = 10` at least.

Note: when using `xgboost` you may get lots of warnings, ignore these.

The code using `xgb` should take around 30 min. If you want to test the library with gbm learners, or indeed to add more hyperparameters, I suggest you do this after learning how to run `CV.SuperLearner` in parallel (see the extra material provided).

```
my.library.3 <- c(ranger.learners$names,my.library.2)
# my.library.3 <- c(xgb.learners$names,ranger.learners$names,my.library.2)

set.seed(101)
system.time({
  cv_sl1 <- CV.SuperLearner(Y = y_train, X = x_train,
                            family = gaussian(),
                            V = 3,
                            SL.library = my.library.3)
})
```

```
##    user  system elapsed
## 269.120   2.985 261.155
```

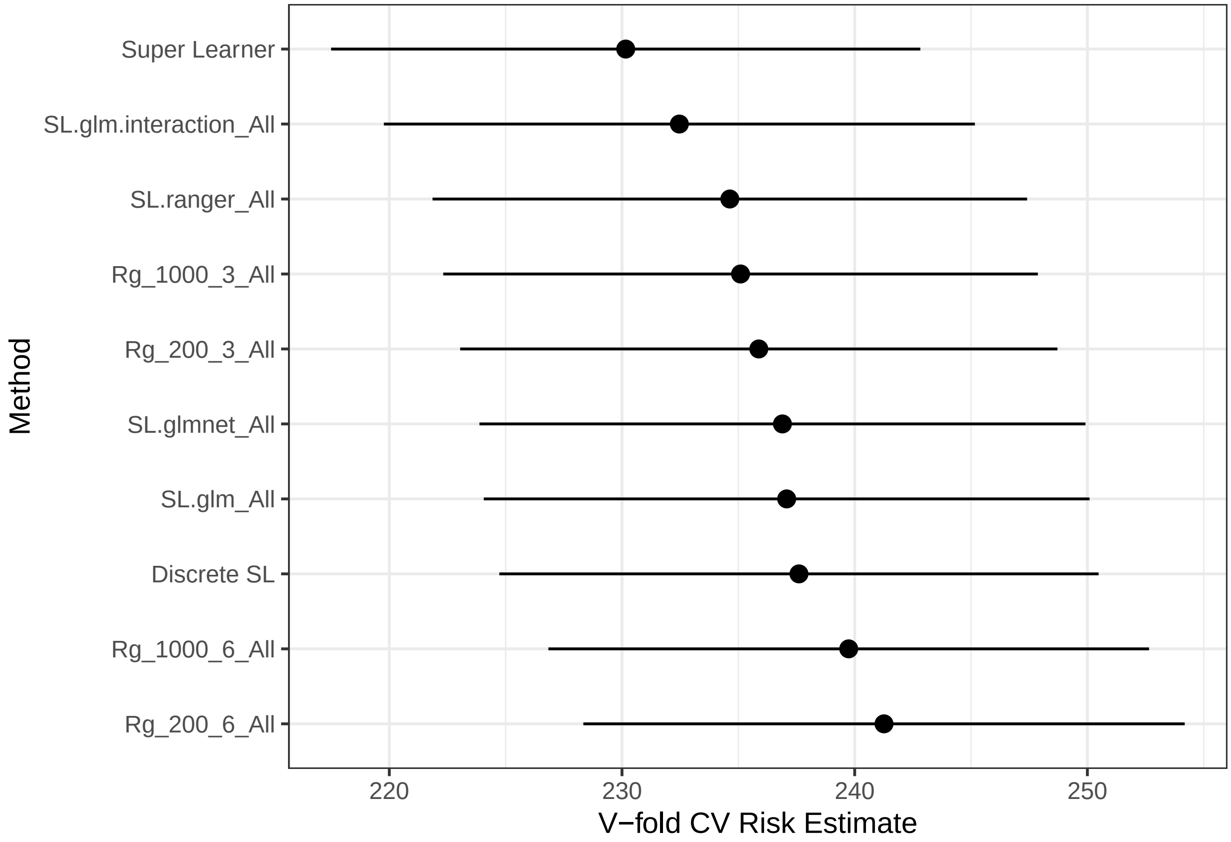
Review the results and plot the performance with 95% CIs.

```
summary(cv_sl1)
```

```
##
## Call:
## CV.SuperLearner(Y = y_train, X = x_train, V = 3, family = gaussian(),
##     SL.library = my.library.3)
##
## Risk is based on: Mean Squared Error
##
```

```
## All risk estimates are based on V =  3
##
##                Algorithm    Ave     se    Min    Max
##            Super Learner 230.16 6.4599 226.65 233.02
##              Discrete SL 237.61 6.5699 234.04 241.19
##             Rg_200_3_All 235.88 6.5478 233.81 238.39
##            Rg_1000_3_All 235.09 6.5192 233.69 237.58
##             Rg_200_6_All 241.26 6.5931 237.35 244.33
##            Rg_1000_6_All 239.74 6.5853 236.54 242.40
##               SL.glm_All 237.08 6.6414 231.39 241.39
##   SL.glm.interaction_All 232.47 6.4792 229.40 235.36
##            SL.glmnet_All 236.89 6.6437 231.27 241.19
##            SL.ranger_All 234.63 6.5186 233.72 236.14
```

```
plot(cv_sl1) + theme_bw()
```



We see that several of the gradient boosting are statistically similar to the Super Learner. In particular those using 100 iterations and the one with the defaults hyperparameter.

Weight distribution for Super Learner

The weights or coefficients of the Super Learner are stochastic - they will change as the data changes.

So we don't necessarily trust a given set of weights as being the "true" weights, by using `CV.SuperLearner` we effectively have multiple samples from the distribution of the "true" weights.

The following function cam extract the weights at each CV.SuperLearner iteration and summarize the distribution of those weights. (credit Chris Kennedy from UC Berkeley Data Lab).

```
#' Functon to Review meta-weights (coefficients) from a CV.SuperLearner object
review_weights = function(cv_sl) {
  meta_weights = coef(cv_sl)
  means = colMeans(meta_weights)
  sds = apply(meta_weights, MARGIN = 2,  FUN = sd)
  mins = apply(meta_weights, MARGIN = 2, FUN = min)
  maxs = apply(meta_weights, MARGIN = 2, FUN = max)
  # Combine the stats into a single matrix.
  sl_stats = cbind("mean(weight)" = means, "sd" = sds, "min" = mins, "max" = maxs)
  # Sort by decreasing mean weight.
  sl_stats[order(sl_stats[, 1], decreasing = TRUE), ]
}
```

It is recommended to review the weight distribution for any SuperLearner project to better understand which algorithms are chosen for the ensemble.

```
print(review_weights(cv_sl1), digits = 3)
```

```
##                          mean(weight)     sd   min   max
## SL.glm_All                     0.3290 0.1167 0.213 0.446
## SL.glm.interaction_All         0.3045 0.0377 0.270 0.345
## SL.ranger_All                  0.2325 0.2453 0.000 0.489
## Rg_200_3_All                   0.0695 0.1204 0.000 0.209
## Rg_1000_3_All                  0.0644 0.1115 0.000 0.193
## Rg_200_6_All                   0.0000 0.0000 0.000 0.000
## Rg_1000_6_All                  0.0000 0.0000 0.000 0.000
## SL.glmnet_All                  0.0000 0.0000 0.000 0.000
```

As you can see, there are many learners in our library that the ensemble never uses.

Optional: binary outcomes

If you have time, you can try running a SL with Diabetes as outcome. Since it's binary, you need to change the argument family to "binomial". You will also need to redefine the design matrix, so it excludes diabetes from the set of predictors. We create training and holdout datasets

```
y.bin <- as.numeric(df$Diabetes)-1
y.bin_train <- y.bin[train_obs]
y.bin_holdout <- y.bin[-train_obs]

X <- df %>% dplyr::select(-one_of("Diabetes") )
x_train <- X[train_obs, ]
x_holdout <- X[-train_obs, ]
```

By default, SL uses method NNLS. But for binary prediction, other loss functions are available. We can specify `method = "method.NNloglik"` or `"method.AUC"`. AUC may be preferred when our outcome is rare.

This appears to be the case here (type `table(df$Diabetes)` to see this), we will use AUC first. You can try other options to see if it makes a difference.

We begin by setting the seed and specifying learners for binary data. Then run the binary SL.

```
set.seed(1)
my.library.bin <- c("SL.glm", "SL.glm.interaction", "SL.ranger","SL.gbm")


sl.bin <- SuperLearner(Y = y.bin_train, X = x_train, family =  binomial(),
                       SL.library = my.library.bin,
                       method = "method.AUC",
                       cvControl = list(shuffle = F, V=10))
```

You can check the time it took by typing

```
sl.bin$times$everything
```

Check the SL object

```
sl.bin
```

All the learners should have non-zero weights.

We can now use the Super Learner to predict on new data.

```
Yhat.sl.bin <- predict(sl.bin, x_holdout, onlySL = TRUE)[[1]]
```

We can also use the method argument to change the loss function and the way we select convex combinations.

Run the following to obtain a super learner that uses Non-negative Log-likelihood. Check the SL object, and use the resulting SL to predict on new data

```
sl.bin.2 <- SuperLearner(Y = y.bin_train, X = x_train, family =  binomial(),
                         SL.library = my.library.bin,
                         method = "method.NNloglik",
                         cvControl = list(shuffle = F, V=10))

sl.bin.2
# all the learners have non-zero weights

Yhat.sl.bin.2 <- predict(sl.bin.2, x_holdout, onlySL = TRUE)[[1]]
```