Modeling the Exposure and the Outcome: Introduction to the Super Learner

Ashley I Naimi

Oct 2022

Contents

1 An Example sl3 Implementation 2
2 Create Tuning Parameter Grids 7

1 An Example sl3 Implementation

Thus far in this workshop, we've seen that the performance of all of the algorithms explored (random forest, XGboost, neural networks, etc) depends on a set of tuning parameters. This leads to the important question: which tuning parameter values should I select. This can be a difficult question to address, since the answer will depend on the minutae of any given project.

Indeed, researchers often spend considerable time/resources on determining the optimal combination of tuning parameters. We will soon see how the superlearner can alleviate this process. But first, let's explore how we might run the super learner via `sl3`.

```r
library(sl3)

scale_ <- function(x) {
    (x - mean(x, na.rm = TRUE))/sd(x, na.rm = TRUE)
}
nhefs <- read_csv(here("data", "nhefs.csv")) %>%
    mutate(wt_delta = as.numeric(wt82_71 >
        median(wt82_71)), age = scale_(age),
        sbp = scale_(sbp), dbp = scale_(dbp),
        price71 = scale_(price71), tax71 = scale_(tax71)) %>%
    select(-wt82_71)

head(nhefs)
```

```
## # A tibble: 6 x 11
##    seqn  qsmk   sex     age income    sbp     dbp price71  tax71  race wt_delta
##   <dbl> <dbl> <dbl>   <dbl>  <dbl>  <dbl>   <dbl>   <dbl>  <dbl> <dbl>    <dbl>
## 1   233     0     0 -0.112      19  2.50   1.74     0.197  0.204     1        0
## 2   235     0     0 -0.614      18 -0.284  0.224    0.922  1.44      0        1
## 3   244     0     1  1.06       15 -0.712 -0.251   -2.53  -2.38      1        1
## 4   245     0     0  2.06       15  1.06   0.0342  -2.81  -2.51      1        1
## 5   252     0     0 -0.279      18 -0.551 -0.0609   0.922  1.44      0        1
## 6   257     0     1 -0.0286     11  0.680  0.509    0.314  0.450     1        1
```

```r
# Begin modeling nhefs with super
# learner create the prediction task
# (i.e., use nhefs to predict outcome)
task <- make_sl3_Task(data = nhefs, outcome = "wt_delta",
    covariates = c("qsmk", "age", "sbp",
        "dbp", "price71", "tax71", "sex",
        "income", "race"), folds = 5)

# let's look at the task
task
```

```
## An sl3 Task with 1394 obs and these nodes:
## $covariates
## [1] "qsmk"    "age"     "sbp"     "dbp"     "price71" "tax71"   "sex"
## [8] "income"  "race"
##
## $outcome
## [1] "wt_delta"
##
## $id
## NULL
##
## $weights
## NULL
##
## $offset
## NULL
##
## $time
## NULL
```

The `sl3_Task` function defines what we want to do with our data in the context of super learning. Many options about how the super learner prediction function will operate can be defined here. For example, we can change the number of cross-validation folds, implement stratified cross-validation, use clustered cross-validation (by adding an `id` argument), and many other things. Details on the arguments for the sl3 task can be obtained using the help functions (i.e., `?sl3_Task`)

The next step is to construct a library of learners that we will implement in the super learner. Before doing this, let's see what learners are available to us in the context of `sl3`. First, we get a list of sl3 task properties supported by at least one learner in the set of available learners:

```
sl3_list_properties()
```

```
##  [1] "binomial"      "categorical"   "continuous"    "cv"
##  [5] "density"       "h2o"           "ids"           "importance"
##  [9] "offset"        "preprocessing" "sampling"      "screener"
## [13] "timeseries"    "weights"       "wrapper"
```

As we can see, there are a number of options here, including the analysis of binary, categorical, and continuous outcome data, the creation of variable importance metrics, preprocessing (for, e.g., missing data), the use of screening algorithms, and the creation of custom wrappers for user-defined algorithms.

In our NHEFS data, `wt_delta` (our outcome) is binary, so we can look at a list of algorithms available to analyze binary outcome data:

```
sl3_list_learners(properties = "binomial")
```

```
##  [1] "Lrnr_bartMachine"               "Lrnr_bayesglm"
##  [3] "Lrnr_bound"                     "Lrnr_caret"
##  [5] "Lrnr_cv_selector"               "Lrnr_dbarts"
##  [7] "Lrnr_earth"                     "Lrnr_ga"
##  [9] "Lrnr_gam"                       "Lrnr_gbm"
## [11] "Lrnr_glm"                       "Lrnr_glm_fast"
## [13] "Lrnr_glmnet"                    "Lrnr_grf"
## [15] "Lrnr_gru_keras"                 "Lrnr_h2o_glm"
## [17] "Lrnr_h2o_grid"                  "Lrnr_hal9001"
## [19] "Lrnr_lightgbm"                  "Lrnr_lstm_keras"
## [21] "Lrnr_mean"                      "Lrnr_nnet"
## [23] "Lrnr_nnls"                      "Lrnr_optim"
## [25] "Lrnr_pkg_SuperLearner"          "Lrnr_pkg_SuperLearner_method"
## [27] "Lrnr_pkg_SuperLearner_screener" "Lrnr_polspline"
## [29] "Lrnr_randomForest"              "Lrnr_ranger"
## [31] "Lrnr_rpart"                     "Lrnr_screener_correlation"
## [33] "Lrnr_solnp"                     "Lrnr_stratified"
## [35] "Lrnr_svm"                       "Lrnr_xgboost"
```

With this list of learners, we can select a few and begin the modeling process:

```
# create a simple glm learner and a
# simple mean learner note: no change
# in any tuning parameters
lrn_glm <- Lrnr_glm$new()
lrn_mean <- Lrnr_mean$new()

# create ridge and lasso regression:
# note: rigde and lasso are defined by
# setting alpha tuning parameter in the
# glmnet function
lrn_ridge <- Lrnr_glmnet$new(alpha = 0)
lrn_lasso <- Lrnr_glmnet$new(alpha = 1)

# create xgboost and ranger: note:
# using default tuning parameters
lrn_ranger <- Lrnr_ranger$new()
lrn_xgb <- Lrnr_xgboost$new()
```

With these learners, we can now combine them to create an `sl3` stack:

```
stack <- Stack$new(lrn_glm, lrn_mean, lrn_ridge,
    lrn_lasso, lrn_ranger, lrn_xgb)
stack
```

```
## [1] "Lrnr_glm_TRUE"
## [2] "Lrnr_mean"
## [3] "Lrnr_glmnet_NULL_deviance_10_0_100_TRUE_FALSE"
## [4] "Lrnr_glmnet_NULL_deviance_10_1_100_TRUE_FALSE"
## [5] "Lrnr_ranger_500_TRUE_none_1"
## [6] "Lrnr_xgboost_20_1"
```

Note above how each learner in the stack is named, particularly for those with tuning parameter options. For example, take the ridge regression learner we created using GLMNET:

```
lrn_ridge
```

```
## [1] "Lrnr_glmnet_NULL_deviance_10_0_100_TRUE_FALSE"
```

We can call the help function for the glmnet learner to confirm that the arguments appended to the end of the learner's name refers to the values of the tuning parameters that can be modified (e.g., type `?Lrnr_glmnet` in the R console).

Now that we have a stack of learners, we can create the function we will need to run the super learner:

```
sl <- Lrnr_sl$new(learners = stack, metalearner = Lrnr_nnls$new(convex = T))
```

In the above function, we are telling the super learner function that we want to use the algorithms in our `stack` library, and that we want to combine these algorithms into a single meta algorithm using the non-negative least squares meta-learner.

The NNLS meta-learner function is one of many options that can be chosen on the basis of the outcome.[1] Here, we will stick to NNLS.

[1] See `https://tlverse.org/sl3/reference/metalearners.html` for more information

We now have all the pieces in place to run the super learner function:

```
set.seed(123)
sl_fit <- sl$train(task = task)


sl_fit$fit_object$cv_meta_fit
```

```
## [1] "Lrnr_nnls_TRUE"
##                                                 lrnrs      weights
## 1:                                   Lrnr_glm_TRUE 0.000000000
## 2:                                       Lrnr_mean 0.000000000
## 3: Lrnr_glmnet_NULL_deviance_10_0_100_TRUE_FALSE 0.383085319
## 4: Lrnr_glmnet_NULL_deviance_10_1_100_TRUE_FALSE 0.607834436
## 5:                       Lrnr_ranger_500_TRUE_none_1 0.000000000
## 6:                                Lrnr_xgboost_20_1 0.009080245
```

In the above output, we see the weight that each algorithm contributes to the overall meta-algorithm when they are combined using NNLS in a convex combination. If we print the `sl_fit` object to the console, we can also see the mean squared error risks for each algorithm, and for the overall super learner.

2 Create Tuning Parameter Grids

One of the great benefits of using the Super Learner is that it can alleviate the uncertainty that results from tuning parameter selection by enabling us to incorporate a wide variety of alternatively tuned models. For example, using the `glmnet` function, we may want to explore what happens when we change the `alpha` parmeter. In the code above, we did this by calling the `glmnet` learner twice:

```
lrn_ridge <- Lrnr_glmnet$new(alpha = 0)
lrn_lasso <- Lrnr_glmnet$new(alpha = 1)
```

Techncially, however, `alpha` can take on any range of values between [0, 1]. So what if the optimal value is somewhere in between. With the `sl3` package, we can easily create a wide range of learners to explore the impact of different tuning parameters. For example:

```
# glmnet learner
grid_params <- seq(0, 1, by = 0.1)
lrnr_glmnet <- vector("list", length = length(grid_params))
for (i in 1:length(grid_params)) {
    lrnr_glmnet[[i]] <- make_learner(Lrnr_glmnet,
        alpha = grid_params[i])
}
```

The code above creates an object (`lrnr_glmnet`) that contains a list of all of the different `glmnet` learners with an `alpha` ranging from 0 to 1 by increments of 0.1:

```
# first four elements of the new
# lrnr_glmnet object
lrnr_glmnet[1:4]
```

```
## [[1]]
## [1] "Lrnr_glmnet_NULL_deviance_10_0_100_TRUE_FALSE"
##
## [[2]]
## [1] "Lrnr_glmnet_NULL_deviance_10_0.1_100_TRUE_FALSE"
##
## [[3]]
## [1] "Lrnr_glmnet_NULL_deviance_10_0.2_100_TRUE_FALSE"
##
## [[4]]
## [1] "Lrnr_glmnet_NULL_deviance_10_0.3_100_TRUE_FALSE"
```

We can then run the superlearner function incorporating each version of this new `lrnr_glmnet`:

```
# create the learning stack with the
# new lrnr_glmnet library
stack <- Stack$new(lrnr_glmnet)

# define the metalearner
sl <- Lrnr_sl$new(learners = stack, metalearner = Lrnr_nnls$new(convex = T))

# run the super learner and print the
# results
set.seed(123)
sl_fit <- sl$train(task = task)

sl_fit$fit_object$cv_meta_fit
```

```
## [1] "Lrnr_nnls_TRUE"
##                                               lrnrs   weights
##  1:   Lrnr_glmnet_NULL_deviance_10_0_100_TRUE_FALSE 0.0000000
##  2: Lrnr_glmnet_NULL_deviance_10_0.1_100_TRUE_FALSE 0.0000000
##  3: Lrnr_glmnet_NULL_deviance_10_0.2_100_TRUE_FALSE 0.0000000
##  4: Lrnr_glmnet_NULL_deviance_10_0.3_100_TRUE_FALSE 0.0000000
##  5: Lrnr_glmnet_NULL_deviance_10_0.4_100_TRUE_FALSE 0.0000000
##  6: Lrnr_glmnet_NULL_deviance_10_0.5_100_TRUE_FALSE 0.0000000
##  7: Lrnr_glmnet_NULL_deviance_10_0.6_100_TRUE_FALSE 0.0000000
##  8: Lrnr_glmnet_NULL_deviance_10_0.7_100_TRUE_FALSE 0.0000000
##  9: Lrnr_glmnet_NULL_deviance_10_0.8_100_TRUE_FALSE 0.2458391
## 10: Lrnr_glmnet_NULL_deviance_10_0.9_100_TRUE_FALSE 0.0000000
## 11:   Lrnr_glmnet_NULL_deviance_10_1_100_TRUE_FALSE 0.7541609
```

As we can see in the above output, most learners contributed a weight of zero to the meta-learner. Only the versions with an alpha parameter of 0.8 (i.e., elastic net, contributing ~ 25%) and of 1 (i.e., the LASSO, contributing ~ 75%) contributed to the fit of the superlearner in the NHEFS data here.

Of course, in the above example, we only searched a grid for GLMNET, and the grid consisted of only a single parameter value. In more realistic settings, the procedure can accommodate much more complex searchers. Here is an example searching over three tuning parameters in the `ranger` function:

```
grid_params <- list(num.trees = c(250, 500,
    1000, 2000), mtry = c(2, 4, 6), min.node.size = c(50,
    100))
grid <- expand.grid(grid_params, KEEP.OUT.ATTRS = FALSE)
lrnr_ranger <- vector("list", length = nrow(grid))
for (i in 1:nrow(grid)) {
    lrnr_ranger[[i]] <- make_learner(Lrnr_ranger,
        num.trees = grid[i, ]$num.trees,
        mtry = grid[i, ]$mtry, min.node.size = grid[i,
            ]$min.node.size)
}

# create the learning stack with the
```

```r
# new lrnr_ranger library
stack <- Stack$new(lrnr_ranger)
stack
```

```
##  [1] "Lrnr_ranger_250_TRUE_none_1_2_50"   "Lrnr_ranger_500_TRUE_none_1_2_50"
##  [3] "Lrnr_ranger_1000_TRUE_none_1_2_50"  "Lrnr_ranger_2000_TRUE_none_1_2_50"
##  [5] "Lrnr_ranger_250_TRUE_none_1_4_50"   "Lrnr_ranger_500_TRUE_none_1_4_50"
##  [7] "Lrnr_ranger_1000_TRUE_none_1_4_50"  "Lrnr_ranger_2000_TRUE_none_1_4_50"
##  [9] "Lrnr_ranger_250_TRUE_none_1_6_50"   "Lrnr_ranger_500_TRUE_none_1_6_50"
## [11] "Lrnr_ranger_1000_TRUE_none_1_6_50"  "Lrnr_ranger_2000_TRUE_none_1_6_50"
## [13] "Lrnr_ranger_250_TRUE_none_1_2_100"  "Lrnr_ranger_500_TRUE_none_1_2_100"
## [15] "Lrnr_ranger_1000_TRUE_none_1_2_100" "Lrnr_ranger_2000_TRUE_none_1_2_100"
## [17] "Lrnr_ranger_250_TRUE_none_1_4_100"  "Lrnr_ranger_500_TRUE_none_1_4_100"
## [19] "Lrnr_ranger_1000_TRUE_none_1_4_100" "Lrnr_ranger_2000_TRUE_none_1_4_100"
## [21] "Lrnr_ranger_250_TRUE_none_1_6_100"  "Lrnr_ranger_500_TRUE_none_1_6_100"
## [23] "Lrnr_ranger_1000_TRUE_none_1_6_100" "Lrnr_ranger_2000_TRUE_none_1_6_100"
```

```r
# define the metalearner
sl <- Lrnr_sl$new(learners = stack, metalearner = Lrnr_nnls$new(convex = T))

# run the super learner and print the
# results
set.seed(123)
sl_fit <- sl$train(task = task)

sl_fit$fit_object$cv_meta_fit
```

```
## [1] "Lrnr_nnls_TRUE"
##                                  lrnrs     weights
##  1:   Lrnr_ranger_250_TRUE_none_1_2_50 0.000000000
##  2:   Lrnr_ranger_500_TRUE_none_1_2_50 0.000000000
##  3:  Lrnr_ranger_1000_TRUE_none_1_2_50 0.000000000
##  4:  Lrnr_ranger_2000_TRUE_none_1_2_50 0.000000000
##  5:   Lrnr_ranger_250_TRUE_none_1_4_50 0.000000000
##  6:   Lrnr_ranger_500_TRUE_none_1_4_50 0.000000000
##  7:  Lrnr_ranger_1000_TRUE_none_1_4_50 0.000000000
##  8:  Lrnr_ranger_2000_TRUE_none_1_4_50 0.000000000
##  9:   Lrnr_ranger_250_TRUE_none_1_6_50 0.000000000
## 10:   Lrnr_ranger_500_TRUE_none_1_6_50 0.000000000
## 11:  Lrnr_ranger_1000_TRUE_none_1_6_50 0.000000000
## 12:  Lrnr_ranger_2000_TRUE_none_1_6_50 0.000000000
## 13:  Lrnr_ranger_250_TRUE_none_1_2_100 0.006488196
## 14:  Lrnr_ranger_500_TRUE_none_1_2_100 0.632748465
## 15: Lrnr_ranger_1000_TRUE_none_1_2_100 0.000000000
## 16: Lrnr_ranger_2000_TRUE_none_1_2_100 0.000000000
## 17:  Lrnr_ranger_250_TRUE_none_1_4_100 0.000000000
## 18:  Lrnr_ranger_500_TRUE_none_1_4_100 0.000000000
## 19: Lrnr_ranger_1000_TRUE_none_1_4_100 0.360763340
## 20: Lrnr_ranger_2000_TRUE_none_1_4_100 0.000000000
## 21:  Lrnr_ranger_250_TRUE_none_1_6_100 0.000000000
## 22:  Lrnr_ranger_500_TRUE_none_1_6_100 0.000000000
## 23: Lrnr_ranger_1000_TRUE_none_1_6_100 0.000000000
## 24: Lrnr_ranger_2000_TRUE_none_1_6_100 0.000000000
##                                 lrnrs     weights
```

In practice, it is recommended that the tuning parameter grid search approach be combined for several algorithms.