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

Ashley I Naimi

June 2022

## Contents

## 1 Introduction to Stacking

In the early 1990s, Wolpert developed an approach to combine several "lower-level" machine learning methods into a "higher-level" model with the goal of increasing predictive accuracy (Wolpert, 1992). He termed the approach "stacked generalization," or "stacking." Later, Breiman demonstrated how stacking can be used to improve the predictive accuracy in a regression context, and showed that imposing certain constraints on the higher-level model improved predictive performance (Breiman, 1996). More recently, van der Laan and colleagues proved that stacking possesses certain ideal theoretical properties (van der Laan and S, 2003, van der Laan et al. (24), van der Laan et al. (2007)). In particular, their oracle inequality guarantees that in large samples the algorithm will perform at least as well as the best individual predictor included in the ensemble. Therefore, choosing a large library of diverse algorithms will enhance performance, and creating the best weighted combination of candidate algorithms will further improve performance. Here, "best" is defined in terms of a bounded loss function, and over-fitting is avoided with $V$-fold cross-validation. In this context, the term "Super Learner" was coined.[1]

[1] In other words, super learner is stacking or a stacked generalization.

Super Learner has tremendous potential for improving the quality of prediction algorithms in applied health sciences, and minimizing the extent to which empirical findings rely on parametric modeling assumptions. It can also serve to be a tremendously important tool when seeking to estimate causal effects using machine learning algorithms. Indeed, as we will see there are several software implementations of the super learner that are meant to be merged with various double robust estimators, including AIPW and TMLE.

## 2 Estimating a Dose-Response Curve with Super Learner

For 1,000 observations, we generate a continuous exposure $X$ by drawing from a uniform distribution with a minimum of zero and a maximum of eight and then generate a continuous outcome $Y$ as

$$Y = 5 + 4 \times \sqrt{9X} \times \mathbb{I}(X < 2) + \mathbb{I}(X \geq 2) \times (|x - 6|^2) + \epsilon, \qquad (1)$$

where $\mathbb{I}()$ denotes the indicator function evaluating to 1 if the argument is true (zero otherwise), and $\epsilon$ was drawn from a doubly-exponential distribution with mean zero and scale parameter one. The true dose-response curve is depicted by the black line in Figure 1.

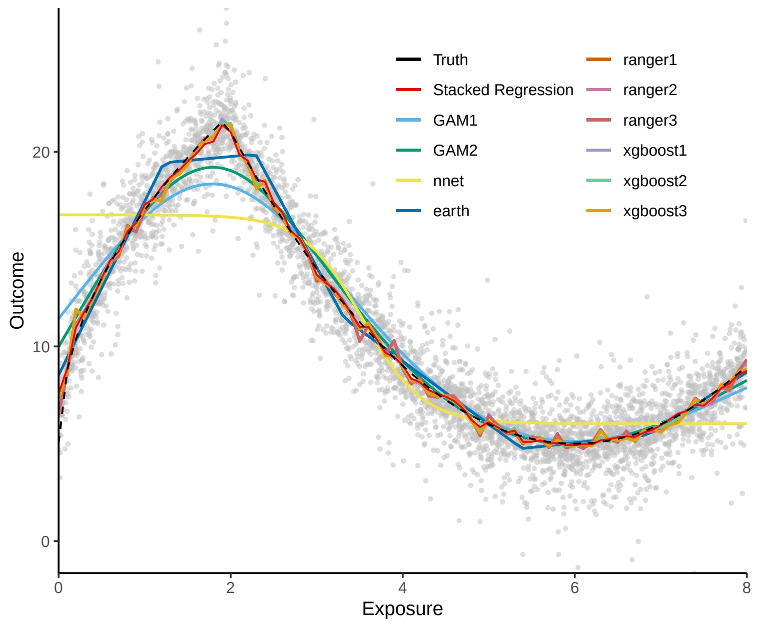

Figure 1: Example dose-response relation between a continuous variable X and an outcome variable Y. The black dashed line represents the true dose-response relation, with all additional lines representing various machine learning estimators of the fit, including the Super Learner (Stacked Regression).

The code to generate these data is as follows:

```
library(rmutil)
library(here)
library(SuperLearner)

# set the seed for reproducibility
set.seed(123)

# generate the observed data
n = 1000
x = runif(n, 0, 8)
y = 5 + 4 * sqrt(9 * x) * as.numeric(x <
    2) + as.numeric(x >= 2) * (abs(x - 6)^(2)) +
    rlaplace(n)

# to plot the true dose-response curve,
# generate sequence of 'doses' from 0
# to 8 at every 0.1, \tthen generate
# the true outcome
xl <- seq(0, 8, 0.1)
yl <- 5 + 4 * sqrt(9 * xl) * as.numeric(xl <
    2) + as.numeric(xl >= 2) * (abs(xl -
    6)^(2))

D <- data.frame(x, y)  # observed data
Dl <- data.frame(xl, yl)  # for plotting the true dose-response curve

head(D)
```

```
##          x         y
## 1 2.300620 18.082553
## 2 6.306441  5.301833
## 3 3.271815 11.304711
## 4 7.064139  7.359499
## 5 7.523738  8.510791
## 6 0.364452 12.199149
```

We now manually demonstrate how the Super Learner can be used to flexibly model this relation without making parametric assumptions.

To simplify our illustration, we consider only two “level-zero” algorithms as candidates to the Super Learner library: generalized additive models with 5-knot natural cubic splines (`gam`) (Hastie and Tibshirani, 1990) and multivariate adaptive regression splines implemented via the `earth` package. In practice, a large and diverse library of candidate estimators is recommended. Our specific interest is in quantifying the mean of $Y$ as a (flexible) function of $X$. To measure the performance of candidate algorithms and to construct the weighted combination of algorithms, we select the $L$-2 squared error loss function $(Y - \hat{Y})^2$ where $\hat{Y}$ denotes our predictions. Minimizing the expectation of the $L$-2 loss function is equivalent to minimizing mean squared error, which is the same objective function used in ordinary least squares regression (Hastie et al., 2009)$^{(\text{section 2.4})}$. To estimate this expected loss, called the “risk”, we use $V$-fold cross-validation with $V = 5$ folds.

```
# Specify the number of folds for
# V-fold cross-validation
folds = 5
## split data into 5 groups for 5-fold
## cross-validation we do this here so
## that the exact same folds will be
## used in both the SL fit with the R
## package, and the hand coded SL
index <- split(1:1000, 1:folds)
splt <- lapply(1:folds, function(ind) D[index[[ind]],
    ])
# view the first 6 observations in the
# first [[1]] and second [[2]] folds
head(splt[[1]])
```

```
##            x         y
## 1  2.300620 18.082553
## 6  0.364452 12.199149
## 11 7.654667  7.659093
## 16 7.198600  7.229326
## 21 7.116315  6.165418
## 26 5.668244  4.659275
```

```
head(splt[[2]])
```

```
##            x         y
## 2  6.306441  5.301833
## 7  4.224844  8.943891
## 12 3.626673 10.249762
## 17 1.968702 21.964325
## 22 5.542427  6.089999
## 27 4.352528  7.891969
```

```r
#-------------------------------------------------------------------------------
# Fit using the SuperLearner Package
#-------------------------------------------------------------------------------
# Create the 5 df GAMs using functions
# called from programs 'sourced' above
SL.gam.5 <- create.Learner("SL.gam", params = list(deg.gam = 5))

# Specifying the SuperLearner library
# of candidate algorithms
sl.lib <- c(SL.gam.5$names, "SL.earth")

# Fit using the SuperLearner package,
# specify \t\toutcome-for-prediction
# (y), the predictors (x), the loss
# function (L2), \t\tthe library
# (sl.lib), and number of folds
fitY <- SuperLearner(Y = y, X = data.frame(x),
    method = "method.NNLS", SL.library = sl.lib,
    cvControl = list(V = folds, validRows = index))

# View the output: 'Risk' column
# returns the CV-MSE estimates
# \t\t'Coef' column gives the weights
# for the final SuperLearner
# (meta-learner)
fitY
```

```
##
## Call:
## SuperLearner(Y = y, X = data.frame(x), SL.library = sl.lib, method = "method.NNLS",
##     cvControl = list(V = folds, validRows = index))
##
##
##                   Risk      Coef
## SL.gam_1_All 2.446902 0.3379264
## SL.earth_All 2.309112 0.6620736
```

```
# Now predict the outcome for all
# possible x 'doses'
yS <- predict(fitY, newdata = data.frame(x = xl),
    onlySL = T)$pred

# Create a dataframe of all x 'doses'
# and predicted SL responses
Dl1 <- data.frame(xl, yS)
```

*Step 1.* Split the observed ”level-zero” data into $5$ mutually exclusive and exhaustive groups of $n/V = 1000/5 = 200$ observations. These groups are called ”folds.”

*Step 2.* For each fold $v = \{1, \ldots, 5\}$,

a. Define the observations in fold $v$ as the validation set, and all remaining observations (80% of the data) as the training set.

b. Fit each algorithm on the training set.

c. For each algorithm, use its estimated fit to predict the outcome for each observation in the validation set. Recall the observations in the validation set are not used train each candidate algorithm.

d. For each algorithm, estimate the risk. For the $L$-2 loss, we average the squared difference between the outcome $Y$ and its prediction $\hat{Y}$ for all observations in the validation set $v$. In other words, we calculate the mean squared error (MSE) between the observed outcomes in the validation set and the predicted outcomes based on the algorithms fit on the training set.

*Step 3.* Average the estimated risks across the folds to obtain one measure of performance for each algorithm. In our simple example, the cross-validated estimates of the squared prediction error are 2.45 for `gam` and 2.31 for `earth`.

At this point, we could simply select the algorithm with smallest cross-validated risk estimate (here, `earth`). This approach is sometimes called the Discrete Super Learner. Instead, we combine the cross-validated predictions, which are referred to as the ”level-one” data, to improve performance and build the ”level-one” learner.

*Step 4.* Let $\hat{Y}_{\text{gam-cv}}$ and $\hat{Y}_{\texttt{earth-cv}}$ denote the cross-validated predicted outcomes from `gam` and `earth`, respectively. Recall the observed outcome is denoted $Y$. To calculate the contribution of each candidate algorithm to the final Super Learner prediction, we use non-negative least squares to regress the actual outcome against the predictions, while suppressing the intercept and constraining the coefficients to be non-negative and sum to 1:

$$\mathbb{E}(Y|\hat{Y}_{\text{gam-cv}}, \hat{Y}_{\text{earth-cv}}) = \alpha_1 \hat{Y}_{\text{gam-cv}} + \alpha_2 \hat{Y}_{\text{earth-cv}}, \tag{2}$$

such that $\alpha_1 \geq 0$; $\alpha_2 \geq 0$, and $\sum_{k=1}^{2} \alpha_k = 1$. Combining the $\hat{Y}_{\text{gam-cv}}$ and $\hat{Y}_{\text{earth-cv}}$ under these constraints (non-negative estimates that sum to 1) is referred to as a ”convex combination,” and is motivated by both theoretical results and improved stability in practice [@Breiman1996, @vanderLaan2007]. Non-negative least squares corresponds to minimizing the mean squared error, which is our chosen loss function (and thus, fulfills our objective). We then normalize the coefficients from this regression to sum to 1. In our simple example, the normalized coefficient values are $\hat{\alpha}_1 =$0.34 for `gam` and $\hat{\alpha}_2 =$0.66 for `earth`. Therefore, both generalized additive models and regression splines each contribute approximately 35% and 65% of the weight in the optimal predictor.

*Step 5.* The final step is to use the above weights to generate the Super Learner, which can then be applied to new data $(X)$ to predict the continuous outcome. To do so, re-fit `gam` and `earth` to the entire sample and denote the predicted outcomes as $\hat{Y}_{\text{gam}}$ and $\hat{Y}_{\text{earth}}$, respectively. Then combine these predictions with the estimated weights from Step 4:

$$\hat{Y}_{\text{SL}} = 0.34\hat{Y}_{\text{gam}} + 0.66\hat{Y}_{\text{earth}} \tag{3}$$

where $\hat{Y}_{\text{SL}}$ denotes our final Super Learner predicted outcome.

Figure 1 demonstrates the fit from several different lower-level algorithms (GAM, nnet, earth, ranger, xgboost), and the combined super learner algorithm (Stacked Regression).

To gain some intuition on the process described above, let's evaluate how we might get these answers without relying on the SuperLearner function, but by hand coding the process.[2]

[2] This example is taken from a paper by Laura Balzer and I, published in the European Journal of Epidemiology Naimi and Balzer (2018). In that work, we also demonstrate how the same super learner process works when the loss function is the rank loss (which corresponds to maximizing area under AUC). Code for this example is available here: https://github.com/ainaimi/SuperLearnerIntro

```r
library(nnls)
library(earth)
library(gam)

#-------------------------------------------------------------------------------
# Hand-coding Super Learner
#-------------------------------------------------------------------------------

## 2: the lapply() function is an
## efficient way to rotate through the
## folds to execute the following:
## \t(a) set the ii-th fold to be the
## validation set; (b) fit each
## algorithm on the training set, which
## is what's left after taking the
## ii-th fold out (i.e., splt[-ii]);
## (c) obtain the predicted outcomes
## for observations in the validation
## set; (d) estimate the estimated risk
## (CV-MSE) for each fold 2b: fit each
## algorithm on the training set (but
## not the ii-th validation set)
m1 <- lapply(1:folds, function(ii) gam(y ~
    s(x, 5), family = "gaussian", data = rbindlist(splt[-ii])))
m2 <- lapply(1:folds, function(ii) earth(y ~
    x, data = rbindlist(splt[-ii]), degree = 2,
    penalty = 3, nk = 21, pmethod = "backward",
    nfold = 0, ncross = 1, minspan = 0, endspan = 0))

## 2c: predict the outcomes for
## observation in the ii-th validation
## set
p1 <- lapply(1:folds, function(ii) predict(m1[[ii]],
    newdata = rbindlist(splt[ii]), type = "response"))
p2 <- lapply(1:folds, function(ii) predict(m2[[ii]],
    newdata = rbindlist(splt[ii]), type = "response"))

# add the predictions to grouped
# dataset 'splt'
for (i in 1:folds) {
    splt[[i]] <- cbind(splt[[i]], p1[[i]],
        p2[[i]])
}
# view the first 6 observations in the
# first fold column2 (y) is the
# observed outcome; column3 is the
# CV-predictions from gam column4 is
# the CV-predictions from earth;
head(splt[[1]])
```

```
##            x         y   p1[[i]]         y
## 1  2.300620 18.082553 18.315169 19.529790
## 6  0.364452 12.199149 12.705405 11.722377
## 11 7.654667  7.659093  7.570780  7.866627
## 16 7.198600  7.229326  6.527277  6.541223
## 21 7.116315  6.165418  6.355935  6.302089
## 26 5.668244  4.659275  4.993494  4.934016
```

```
## 2d: calculate CV risk for each
## method for with ii-th validation set
## our loss function is L2-squared
## error; so our risk is mean squared
## error
risk1 <- lapply(1:folds, function(ii) mean((splt[[ii]][,
    2] - splt[[ii]][, 3])^2))
```

```
risk2 <- lapply(1:folds, function(ii) mean((splt[[ii]][,
    2] - splt[[ii]][, 4])^2))

## 3: average the estimated risks
## across the 5 folds to obtain 1
## measure of performance for each
## algorithm
a <- rbind(cbind("gam", mean(do.call(rbind,
    risk1), na.rm = T)), cbind("earth", mean(do.call(rbind,
    risk2), na.rm = T)))

# checking to see match with SL output
fitY
```

```
##
## Call:
## SuperLearner(Y = y, X = data.frame(x), SL.library = sl.lib, method = "method.NNLS",
##     cvControl = list(V = folds, validRows = index))
##
##
##                  Risk      Coef
## SL.gam_1_All 2.446902 0.3379264
## SL.earth_All 2.309112 0.6620736
```

```
a
```

```
##      [,1]    [,2]
## [1,] "gam"   "2.4469015494462"
## [2,] "earth" "2.30911178845963"
```

```
#------------
## 4: estimate SL weights using nnls
## (for convex combination) and
## normalize
```

```
# create a new datafame with the
# observed outcome (y) and
# CV-predictions from the 3 algorithms
X <- data.frame(do.call(rbind, splt))[, -1]
names(X) <- c("y", "gam", "earth")
head(X)
```

```
##             y       gam     earth
## 1  18.082553 18.315169 19.529790
## 6  12.199149 12.705405 11.722377
## 11  7.659093  7.570780  7.866627
## 16  7.229326  6.527277  6.541223
## 21  6.165418  6.355935  6.302089
## 26  4.659275  4.993494  4.934016
```

```
SL.r <- nnls(cbind(X[, 2], X[, 3]), X[, 1])$x
alpha <- as.matrix(SL.r/sum(SL.r))
round(alpha, 3)
```

```
##       [,1]
## [1,] 0.338
## [2,] 0.662
```

```
# compare to the package's coefficients
fitY
```

```
##
## Call:
## SuperLearner(Y = y, X = data.frame(x), SL.library = sl.lib, method = "method.NNLS",
##     cvControl = list(V = folds, validRows = index))
##
##
##                   Risk      Coef
## SL.gam_1_All 2.446902 0.3379264
## SL.earth_All 2.309112 0.6620736
```

```
#-----------------
## 5a: fit all algorithms to original
## data and generate predictions
m1 <- gam(y ~ s(x, 5), family = "gaussian",
    data = D)
m2 <- earth(y ~ x, data = D, degree = 2,
    penalty = 3, nk = 21, pmethod = "backward",
    nfold = 0, ncross = 1, minspan = 0, endspan = 0)

## 5b: predict from each fit using all
## data
p1 <- predict(m1, newdata = D, type = "response")
p2 <- predict(m2, newdata = D, type = "response")

predictions <- cbind(p1, p2)

## 5c: for the observed data take a
## weighted combination of predictions
## using nnls coeficients as weights
y_pred <- predictions %*% alpha


#-------------------------------------------
## now apply to new dataset (all doses
## (xl)) to verify that our work
## predicts similar results as actual
## SL function
p1 <- predict(m1, newdata = data.frame(x = xl),
    type = "response")
p2 <- predict(m2, newdata = data.frame(x = xl),
    type = "response")

predictions <- cbind(p1, p2)
yS2 <- predictions %*% alpha
```

```r
# now we have a new dataframe of doses
# (xl) and SL manual predicted outcome
D12 <- data.frame(xl, yS2)

head(D12)
```

```
##    xl       yS2
## 1 0.0  8.972475
## 2 0.1  9.823825
## 3 0.2 10.674687
## 4 0.3 11.523978
## 5 0.4 12.370240
## 6 0.5 13.211791
```

```r
# plotting these data can give us a
# visual of our estimate of the
# dose-response function
```

As we can see from the above hand coding implementation, the mean squared errors and super learner weights are identical between the hand-coded and `SuperLearner` versions.

## 3 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`.

```
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
```

```
# 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
```

```r
# 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:

```r
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:

```r
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.[3] Here, we will stick to NNLS.

[3] 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.

## 4 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
# 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"
```

```
# 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.

## References

Leo Breiman. Stacked regressions. *Machine Learning*, 24(1):49–64, 1996.

Trevor Hastie and Robert Tibshirani. *Generalized Additive Models*. Chapman & Hall, London; New York, 1990.

Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY, 2009.

Ashley I Naimi and Laura B Balzer. Stacked generalization: an introduction to super learning. *Eur J Epidemiol*, 33(5):459–464, 2018.

Mark J van der Laan and Dudoit S. Unified cross-validation methodology for selection among estimators and a general cross-validated adaptive epsilon-net estimator: Finite sample oracle inequalities and examples. *U.C. Berkeley Division of Biostatistics Working Paper Series*, Working Paper 130 (https://biostats.bepress.com/ucbbiostat/paper130), 2003.

Mark J van der Laan, Eric C Polley, and Alan E Hubbard. Super learner. *Statistical Applications in Genetics and Molecular Biology*, 6(1):Article 25, 2007.

Mark J. van der Laan, Sandrine Dudoit, and Aad W. van der Vaart. The cross-validated adaptive epsilon-net estimator. *Statistics & Decisions*, 373-395, 24.

DH Wolpert. Stacked generalization. *Neural Netw*, 5(2):241–59, 1992.