Super Learner for Prediction (Part 1)

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

8 Feb 2022

Introduction

In this practical, you will apply the SuperLearner. We will use part of the National Health and Nutrition Examination Survey (NHANES) dataset. We want to write the raw dataset to a data frame in our environment, and can do this with the following code:

```r
nhanes_installed <- require(NHANES)
if(!nhanes_installed) install.packages("NHANES", repos = "http://cran.us.r-project.org")
library(NHANES)

data("NHANES")
df <- NHANESraw
```

Data Processing

Inspect the data

```r
dim(df)
names(df)
table(df$SmokeNow, df$Smoke100, useNA='always')
```

We will be interested in smoking as an exposure. The dataset contains two smoking variables `Smoke100` which is a binary indicator of whether a person has smoked at least 100 cigarettes in their lifetime (but is not a current smoker), and `SmokeNow` which indicates if the person is a current smoker.

Combine these into a single factor variable that indicates whether the individual is an ex, current or never smoker.

```r
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))
```

A systolic blood pressure reading (`BPSysAve`), a continuous outcome, will be a primary outcome. If there is time, you can also consider diabetes status (`Diabetes`), a binary outcome.

Now we trim the variables in the data set, keeping only the ones relevant to the practical:

```r
df <- df %>% dplyr::select(one_of(
  "BPSysAve","BMI", "Age", "SleepHrsNight", "PhysActive","Smoke",
  "Gender", "Race1", "Poverty", "Diabetes", "TotChol"))
```

Inspect our outcome variables, and do the same for key covariates `BMI` and `Age`.

```
table(df$Diabetes,useNA = 'always')
summary(df$BPSysAve)


table(df$Smoke, useNA = 'always')
summary(df$BMI)
summary(df$Age)
```

For this illustration, we want to keep only the complete cases in the dataset. We do this using a single line of code.

```
df <- df[complete.cases(df),]
```

Now split the data into training and test sets. The split should be 50/50. Call the sets `df_train` and `df_holdout`.

```
set.seed(101)
train_obs <- sample(nrow(df), size = nrow(df)*0.5)

df_train <- df[train_obs, ]

# Create a holdout set for evaluating model performance.
# Note: cross-validation is even better than a single holdout sample.
df_holdout <- df[-train_obs, ]
```

We want the outcome (`BPSysAve`) in vector form (also split into train and holdout) and kept separate from the main data frame. Do this now.

```
# The continuous outcome will be the average of the 3 systolic blood pressure measurrement
# BPSysAve
y_train <- df$BPSysAve[train_obs]
y_holdout <- df$BPSysAve[-train_obs]
```

Fitting Individual Algorithms

Linear Regression

We begin by trying different learners individually. For example: a linear regression, lasso (glmnet), randomForest, XGBoost. These should ideally be tested with multiple hyperparameter settings for each algorithm. (You will get to do this later!)

Carry out a simple linear regression for the systolic blood pressure measurement, using `BMI`, `Age`, `SleepHrsNight`, `PhysActive`, `Smoke`, `Gender`, `Race1`, `Poverty`, `Diabetes` and `TotChol` as covariates.

Using the generic `predict()` function to make predictions, manually calculate the sum of square errors for the test set.

```
form <- "BPSysAve ~ BMI + Age + SleepHrsNight + PhysActive +
                     Smoke + Gender + Race1 + Poverty + Diabetes +TotChol"
mod.reg <- glm(form, data=df_train, family=gaussian)
Yhat.reg <- predict(mod.reg, newdata=df_holdout, type='response')
```

```
# sum squares error
SSE.reg <- sum((y_holdout - Yhat.reg)^2)
```

Boosting

Now we run a gradient boosting algorithm on the data. We specify the formula with only the main terms, but recall that trees automatically include interactions up to the specified depth.

Use the `gbm()` function to fit a gradient boosting model to the data.

```
#' Set the seed for reproducibility.
set.seed(1)

boost <- gbm( formula = as.formula(form),
              data = df_train,
              distribution = "gaussian",
              shrinkage = 0.001,
              n.trees = 5000,
              cv.folds = 5,
              interaction.depth=3 )
```

We can use the `gbm.perf()` function to check the performance of this model, and by specifying the option `method`, we can do this in multiple ways. Check the performance using out-of-bag validation (`method = "OOB"`) and cross-validation (`method = "CV"`).

How do you interpret the plot that results from running `gbm.perf()`?

```
#' Checks performance using the out-of-bag (OOB) error
best.iter <- gbm.perf(boost, method = "OOB")
print(best.iter)

#' Checks performance using 5-fold cross-validation
best.iter2 <- gbm.perf(boost, method = "cv")
print(best.iter2)
```

The OOB error typically underestimates the optimal number of iterations.

Using the `summary()` function with the `n.trees` options, generate plots that show the relative influence of each variable after (a) a single tree and (b) with the optimal number of iterations as determined in the last question.

```
#' Plots relative influence of each variable
par(mfrow = c(1, 2))
summary(boost, n.trees = 1)          # using first tree
summary(boost, n.trees = best.iter)  # using estimated best number of trees
```

We see that in both (first and best tree) `Age` has the largest influence. `Gender` and `BMI` have second and third largest influence, using the first tree, while the best tree has `BMI` as higher than `Gender`.

Now using the generic `predict()` function, make predictions for the holdout set. Don't forget to stipulate how many iterations of the gradient boosting model should be used. This can be done using the `n.trees` option.

Also generate the sum of square errors that results from applying this model to the holdout set.

```
#' predictions will be on the link scale
Yhat.boost <- predict(boost, newdata = df_holdout, n.trees = best.iter, type = "link")
```

```r
#' Sum of Squared Errors SSE
SSE.boost <- sum((y_holdout - Yhat.boost)^2)
```

LASSO via SuperLearner

We will continue to fit single learners, but in order to get familiar it we do so using the `SuperLearner` package

First, check which learners have been integrated into the `SuperLearner` package. We can use any of these when we run the SuperLearner:

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

`SuperLearner` (SL) likes the outcome and matrix of covariates to be kept separate. Do this now.

```r
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, ]
```

Now let's fit penalised regression LASSO, but using SL. For now using all the defaults which we will explain later:

```r
# Fit lasso model
sl_lasso <- SuperLearner(Y = y_train,
                        X = x_train,
                        family = gaussian(),
                        SL.library = "SL.glmnet",
                        cvControl = list(V=5L))

# Review the elements in the SuperLearner object.
names(sl_lasso)
```

Again using the generic `predict()` function, calculate the sum of square errors on the test set.

```r
# Predict outcome in the holdout using lasso
Yhat.lasso <-  predict(sl_lasso, x_holdout, onlySL = TRUE)$pred

# calculate sum of square errors
SSE.lasso <- sum((y_holdout- Yhat.lasso)^2)
```

The SSE corresponding to LASSO is larger than the one obtained from boosting.

Random Forest via SuperLearner

Fit a random forest model using the wrapper `SL.ranger`. All other options as before. Also calculate the sum of square errors on the test set.

```r
# Fit random forest using the wrapper function SL.ranger, with all the defaults
# You can find these by typing ?SL.ranger

sl_rf <- SuperLearner(Y = y_train,
                     X = x_train,
                     family = gaussian(),
                      SL.library = "SL.ranger",
                      cvControl = list(V=5L))

# predict Y in the holdout
Yhat.rf <- predict(sl_rf, x_holdout, onlySL = TRUE)$pred

# root least squares error
SSE.ranger <- sum((y_holdout - Yhat.rf)^2)
```

The `SL.ranger` wrapper fits a random forest with 500 trees, minimum node size 5, and the number of variables used to split the squared root of the number of available predictors.

With these defaults, the SSE is a bit larger than the one we obtained with boosting.

Multiple Learners Stacked in SuperLearner

Instead of fitting the models separately and looking at the performance using sum of square errors as we have been doing, we now fit them simultaneously by including them all in the SuperLearner library.

For now, we include those algorithms we tried up to now:

```
# Select candidate algorithms
my.library <-c("SL.glm","SL.glmnet", "SL.ranger")

# Set seed
set.seed(101)

# Execute the call to SuperLearner
sl0 <- SuperLearner(Y = y_train,  # Y is the outcome variable
                    X = x_train,  # X is a dataframe of predictor variables, in this case
                                  # everything except for outcome
                    family = gaussian(),  # family will be discussed in more detail when
                                          # we see how wrappers are written.
                                          # for now gaussian (outcome is coninuous)
                                          # binomial() for 0/1 outcome
                    method = "method.NNLS",
                    # method specifies how the ensembling is done (i.e. how the optimal
                    # combination is chose)
                    # for now we will use the \sum_{k=1}^K \alpha_k f_{k,n} method by deafault
                    SL.library = my.library,
                    cvControl = list(shuffle = F, V = 5)
                    # cvControl specifies parameters related to cross validation,
                    # used to estimate the risk on future data
                    # the default is for V = 10-fold
)
```

Now check the output by simply calling the `SuperLearner` object.

```
sl0
```

The output has two main components:

Firstly, the risk is a measure of model accuracy or performance, and we want our models to minimize the estimated risk (according to the specified loss function). Because we did not change it by an option, our SL has used the default mean square error, but this can be altered (see the help files for `SuperLearner`). In this case, the risks for each algorithm in the library are all broadly similar, with the random forest slightly outperforming the other two algorithms in the library.

The `coef` column tells us the importance of each algorithm in the final ensemble. By default (because we use NNLS) the weights are always greater than or equal to 0 and sum to 1 (a 'convex combination'). If a coefficient is 0, it means that the algorithm is not being used in the SuperLearner ensemble. Here we see that `glm` has been given no weight in the final (ensemble) predictor and so is not used.

Now see which has the lowest risk, and how long it took to run, using the following code:

```r
# Let's see which is the discrete SL (i.e min risk)
sl0$cvRisk[which.min(sl0$cvRisk)]

# Review how long it took to run the SuperLearner:
sl0$times$everything
```

Now that we have a SL ensemble predictor, make predictions on the holdout data set and review the results. This can be done using the generic `predict()` function, but we stipulate the option `onlySL = TRUE` so we do not fit algorithms in the library that had zero weight, saving computation.

```r
pred.sl0 <- predict(sl0, x_holdout, onlySL = TRUE)
```

Check the structure of this prediction object using `str(pred.sl0)`. You will see that the prediction object is a list with two objects. The first is a vector of SL predictions. The second is a matrix of predictions; these are the predictions for each of the individual learners in the SL library.

We want the first object in the list to serve as our predictions. Use this to calculate the sum of square errors resulting from applying the SL predictor to the test set.

```r
# Pick out SL predictions
Yhat.sl0 <- pred.sl0[[1]]

# Calculate the sum of square errors
SSE.sl0 <- sum((y_holdout - Yhat.sl0)^2)
```

The SSE is the lower than those SSEs corresponding to LASSO and RF (which were included in the SL library).

Extend the Ensemble

We will now add other base learners to the library You can check which ones are implemented by running `listWrappers()`. (You can also write your own wrapper functions, we will see later how to do this).

For now, add `gbm()` and `glm.interaction`. [Note: xgboost has deprecated a function used in the SL wrapper so warnings "reg:linear is now deprecated in favor of reg:squarederror" will be displayed, so I won't use for this exercise, though ordinarily I do!]

Run another SL predictor using the same three algorithms as before alongside the two new ones. Review the output and the time it took to run as before.

```r
# Define a new, larger library
my.library.2 <- c("SL.glm","SL.glm.interaction","SL.glmnet", "SL.ranger","SL.gbm")

# Set seed
set.seed(101)

# Run SL with new library
```

```r
sl1 <- SuperLearner( Y = y_train,
                     X = x_train,
                     family = gaussian(),
                     SL.library = my.library.2,
                     cvControl = list(shuffle = F, V = 5) )

# Review the SL object
sl1

# Review times
sl1$times$everything
```

You should see that the gradient boosted predictor (`gbm()`) has the most weight. The new `glm` with interactions is the second most important, and the others are either zero or make only small contributions to the ensemble.

As before, use the SL predictor to make predictions on the holdout set.

```r
Yhat.sl1 <- predict(sl1, x_holdout, onlySL = TRUE)[[1]]

# SSE
SSE.sl1 <- sum((y_holdout - Yhat.sl1)^2)
```

We see that adding more learners slightly improves the SSE. Adding more (without removing) will result in further improvements.