ASHLEY I. NAIMI, PHD

PREDICTIVE ANALYTICS

Outline

- Predictive Analytics: An Overview
- Basic Concepts
- Prediction Algorithms
  - Classification and Regression Trees
  - Random Forest
  - Stacking

Predictive Analytics: An Overview

Thus far in the course, we've been introduced to the idea of causal inference and some of its nuances, as well as some of the core ideas in machine learning and artificial intelligence. I say *some* because, while ML/AI methods are a very versatile set of tools, they are often conflated for tools used exclusively in the context of predictive modeling. That is, the approach known as "machine learning" is often conflated with a particular purpose (i.e., predictive modeling).

In this section of the course, we will be introduced to the idea of predictive analytics or predictive modeling. Predictive modeling involves the use of data and algorithms[1] to create a tool that both accurately and precisely predicts the future.

[1] N.B., I am using the word algorithm here in a comprehensive sense that includes methods that fall in both Breiman's algorithmic and data modeling approaches.

Kuhn and Johnson Kuhn and Johnson (2018) define predictive modeling as "the process of developing a mathematical tool or model that generates an accurate prediction" (page 1). Predictive modeling has been around for a very very long time, and much of the research on the role of predictive modeling versus clinical expertise has shown the benefits that the former can bring to the latter. Consider, for instance, the initial work in this area by Paul Meehl who, in the 1950s, conducted several studies on the performance of very simple algorithms compared to clinical experts in predicting the longevity of cancer patients, length of hospital stays, diagnosis of CVD, or susceptibility to sudden infant death syndrome. In a commentary written after the publication of his very controversial book (P. Meehl 2013; P. E. Meehl 1986):

> When you are pushing 90 investigations, predicting everything from the outcome of football games to the diagnosis of liver disease and when you can hardly come up with half dozen studies showing even a weak tendency in favor of the clinician, it is time to draw a practical conclusion, whatever theoretical differences may still be disputed.

Similarly, in Ayres (2008; cited in Kuhn and Johnson 2018 p 5-6) study of the relation between expert opinion and predictive modeling, he concludes that

> "Traditional experts make better decisions when they are provided with the results of statistical prediction. Those who cling to the au-

thority of traditional experts tend to embrace the idea of combining the two forms of ’knowledge’ by giving the experts ’statistical support’ . . . Humans usually make better predictions when they are provided with the results of statistical prediction."

Basic Concepts

When generating a prediction algorithm, it’s important to understand a host of concepts central to the practice, as well as potential issues that might arise. In this section, we cover these basic concepts.

To motivate our discussion, we will assume interest in developing a prediction algorithm for preterm birth. Roughly speaking, preterm birth occurs when a fetus is born prior to 37 weeks of gestation. It can be divided into spontaneous and induced preterm birth.[2] Preterm birth is one of the most vexing clinical and population health challenges in North America. Roughly 11% of infants born between 20 and 32 weeks gestation in the U.S. do not survive past one year. Those that do face the prospects of life-long complications (Moster, Lie, and Markestad 2008; Trønnes et al. 2014). Preterm birth is among the most disparate health outcomes in United States (Green et al. 2005), with an estimated 60% higher risk of preterm birth for non-Hispanic black compared to non-Hispanic white women in 2013 (Hamilton et al. 2014)–a gap that has persisted for decades (Culhane and Goldenberg 2011). As a result of its longstanding and deleterious presence, interest in tools to identify women at high-risk of preterm birth dates back to at least the mid 1970s (Fedrick 1976).

[2] For our purposes, this distinction will not be important and thus not commented upon.

While the examples we will be using will relate specifically to preterm birth, it’s important to note that the general concepts will apply to the creation of any prediction algorithm in any research setting.

Practice 1: Identify the Purpose of a Given Prediction Algorithm

The first thing to do when constructing a prediction algorithm is to identify the reason for its use. How will a given preterm birth prediction algorithm be used? For example, different sets of information should be considered if a prediction algorithm will be used to identify cases that will undergo cervical cerclage[3] versus some less invasive approach. For a less invasive management approach, one

[3] A painful and invasive procedure in which the cervix is stiched up to prevent birth from occurring too soon

may seek to develop an algorithm that yields the largest sensitivities and specificities, which would be equivalent to maximizing the area under the ROC curve (LeDell, Laan, and Petersen 2016).

Alternatively, cervical cerclage is an invasive procedure that may result in premature contractions, membrane rupture, infection, or premature contractions. Thus, to identify cases that will undergo cerclage, one may prefer an algorithm that maximizes the number of women who will deliver preterm that get classified as preterm births (high sensitivity), but minimizes the number of women who will not deliver preterm and get classified as preterm births (fewest false positives). This would lead to the most optimal allocation of the drug, in terms of both effective use and cost.

Maximizing sensitivity and minimizing false positives is a strategy that was employed for the cost effective allocation of pre-exposure prophylactics among potential HIV seroconverters (Zheng et al. 2018). These researchers used custom optimization programs with the SuperLearner package in R (Laan, Polley, and Hubbard 2007) to maximize sensitivity and minimize false positives.

Practice 2: Identify the Context of a Given Prediction Algorithm

For a particular algorithm to be beneficial, one must articulate the context in which it can later be used. Will the algorithm be used once before conception, once early on in the gestational period, at the 36th week among high risk women, or many times throughout gestation? Clarifying such decisions is critical. Not only do these decisions dictate which data will be needed to properly develop and train the algorithm, but they inform future users of the gestational periods in which the performance of the algorithm is supported by data, and where one must rely on extrapolation to warrant the algorithm's use.

For example, fetal breathing movements, cervical length and funneling, amniotic fluid IL-6, and serum C-reactive protein have been identified as promising candidates for predicting spontaneous preterm birth among symptomatic women within 2 to 7 days of testing (Honest, Hyde, and Khan 2012). However, these studies generally did not provide information on the gestational age(s) in which the tests were applied. It is therefore not possible to determine whether these tests will perform well generally across gestation, or specifically within the gestational window(s) in which they were administered in these studies.

Practice 3: Consider the Range of Analytic Methods Available

The last two decades have seen an explosion in the development of prediction methods, including regression based and more complex machine learning techniques (Hastie, Tibshirani, and Friedman 2009; Berk 2008) Generally, methods available to researchers for predicting preterm birth fall into two classes: solitary and ensemble methods. A solitary learner consists of a single method, such as logistic regression, support vector machine, or classification and regression tree, fit to the data to generate a predictive algorithm. In contrast, ensemble learners consist of a grouping of solitary methods. The simplest example of an ensemble learner is a random forest, in which several classification and regression trees are fit to a single dataset, and combined into a single algorithm. A more complex example is a stacked generalization (or stacking, or the Super Learner).[24, 25] We have already seen SVMs. We will cover several other methods in the next section.

Practice 4: Honest Training and Validation

One critically important area that researchers often fail to consider is the evaluation of a given algorithm. Ultimately, the usefulness of any prediction algorithm depends on how well it performs. Unfortunately, it is relatively easy to develop an algorithm that performs exceedingly well in a study (e.g., high sensitivity and specificity), but may subject to serious issues in practice. Proper steps must be taken to protect the assessment of an algorithm from yielding misleading performance results.

A key principle to consider in the evaluation of any algorithm is the **firewall principle** (Mullainathan and Spiess 2017):

> no data used to estimate the model that will generate the predictions should be used to evaluate the predictive performance of that model.

To properly evaluate any algorithm, one must first create a hold-out sample (a sample removed from model estimation) before any analysis or algorithm development begins. Most often, the hold-out sample is a randomly chosen subset (often between 10% - 30% of the original data) (Kuhn and Johnson 2018) However, it is not always possible to set aside such a proportion of data, particularly when the sample size is small. In such a setting, an alternative approach is to use cross-validation or bootstrapping.

Cross-validation proceeds by splitting the data into *k* different partitions. While *k* is usually taken to be between 5 and 10, the smaller the dataset, the larger k should be (Naimi and Balzer 2018). One then proceeds to fit the prediction algorithm to the data *k* times. For each of the *k* analyses, one of the partitions is removed. This partition is the testing sample. The remaining $k\check{}1$ partitions, referred to as the training sample, are used to estimate (or train) the models. Once the models are fit in the training sample, the testing sample is used to generate the performance measure (e.g., sensitivity, specificity, or positive and negative predictive values). The chosen measure obtained from each of the *k* analyses is then averaged together to obtain an overall measure of the performance of the prediction algorithm. This procedure is depicted in Figure 1, which shows the process by which cross-validation can be used to generate average performance measures.

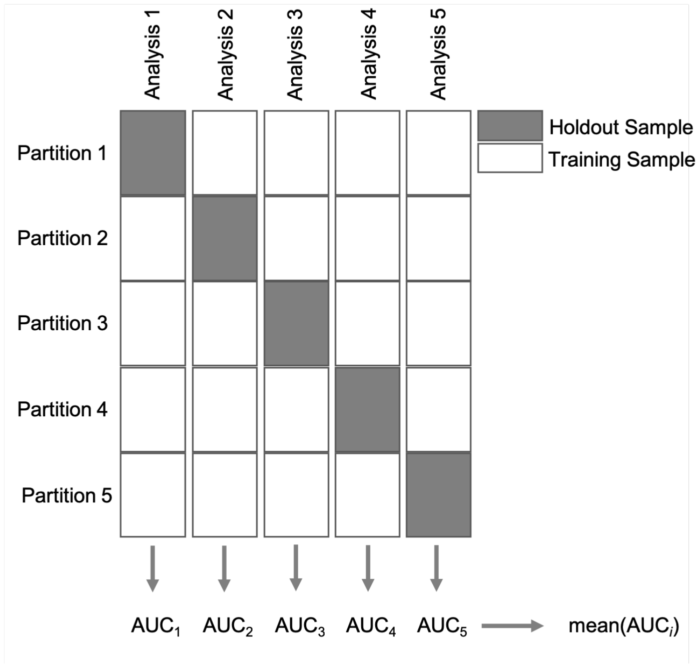


Figure 1: With five-fold cross validation, data are split into five partitions and five analyses are conducted. In the first analysis, the first partition is left out of the analysis. Algorithms are fit to partitions 2-4, and predictions are obtained in partition 1. These predictions are then compared to the observed data to obtain an algorithmic performance measure (in this example, AUC). This process is repeated four more times, and performance measures are averaged across all five runs.

Alternatively, one may use bootstrap aggregation (Figure 2), in which all data are resampled with replacement R times. Larger R's are better, with the smallest number of resamples being between 100 to 500. Bootstrap aggregation proceeds by first resampling the data. Any time a bootstrap resample is taken, there are individuals in the original data that are not in the resample. With bagging, these individuals are referred to as the "out-of-bag" sample, and this constitutes the holdout sample in a bagged estimator.

The algorithm is then fit bootstrap resample, and performance measures are obtained with the out-of-bag sample. This process is repeated in all R resamples, and performance measures are averaged across all R resamples.

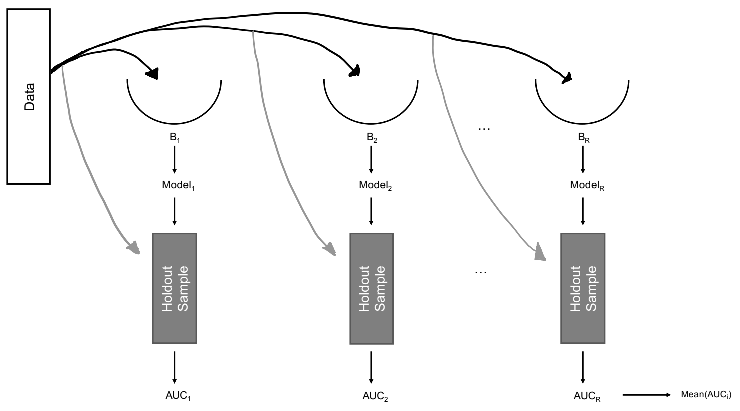


Figure 2: With bootstrap aggregation (bagging), the training data are resampled (with replacement) R times, and R analyses are conducted. For each resample, a model is created. This model is then used to in the holdout data to obtain an algorithmic performance measure (in this example, AUC). The performance measures from the holdout sample obtained from each of the R models are averaged.

Without some form of hold-out sample, either by setting aside a portion of the data, or (in smaller samples) relying on cross-validation or bagging, it is not possible quality of a given predictive algorithm, since the firewall principle has been broken. Even if the study suggests high sensitivity, specificity, AUC, or other performance measures, in actual fact, this may be the result of the fact that the same observations were used to both fit and validate the model. In such settings, great in-sample performance may actually indicate poor out-of-sample performance.

CART

Now that we've covered some general aspects of predictive modeling, let's get into some specific algorithms. Classification and regression trees are one approach, and were first introduced by Brieman, Friedman, and Stone (L. Breiman et al. 1983). They are (to my knowledge) the simplest tree-based algorithms available. Trees are classification based when the outcome being quantified is a binary/categorical variable and we want to predict that category. In such a case, the prediction model is a probability model. On the other hand, they are regression-based when the outcome is continuous and the conditional expectation is being predicted.

The basic idea behind classification and regression trees is to split the outcome data into groups on the basis of the predictors. The algorithm proceeds as follows: first, a single variable is found which **best** splits the data into two groups. The data are separated, and the splitting occurs again, separately in each group. The process is then again repeated recursively until the subgroups either reach a minimum user-defined size (e.g., no less than 5 in each group) or until no **improvement** can be made.

Here, the words "best" and "improvement" are mathematically defined through an **impurity** measure. The split that maximizes impurity is defined as the "best." Change in impurity prior to and proceeding any given split is what is used to define whether a split improves the fit. For example, for a binary outcome, several impurity measures can be defined, among them being the Gini-based impurity measure:

$$\text{Gini}(t) = 1 - \sum_{y=0}^{1} [p(y \mid t)]^2$$

Any split that leads to a decrease in Gini(t) is deemed to improve the fit.

The primary package that enables us to implement CART in R is the `rpart` package[4]:

[4] The natural function would have been `cart()`. However, early authors trademarked the name, forcing those who developed software to seek alternative names. The `rpart` package stands for recursive partitioning, which is what classification and regression trees do.

```
install.packages("rpart", repos = "http://lib.stat.cmu.edu/R/CRAN/")
```

```
##
## The downloaded binary packages are in
##  /var/folders/3s/wfps7lfd62x48nbyfs_v_yh80000gp/T//RtmpDEmkCF/downloaded_packages
```

```
library(rpart)
packageVersion("rpart")
```

```
## [1] '4.1.15'
```

```r
install.packages("rpart.plot", repos = "http://lib.stat.cmu.edu/R/CRAN/")
```

```
##
## The downloaded binary packages are in
##  /var/folders/3s/wfps7lfd62x48nbyfs_v_yh80000gp/T//RtmpDEmkCF/downloaded_packages
```

```r
library(rpart.plot)
```

Along with the rpart package, there is a "kyphosis" dataset used to illustrate the function. These data come from a study of children who have had spinal surgery to correct a curvature known as kyphosis:

```r
## load the data
data("kyphosis")


## dimensions of data
dim(kyphosis)
```

```
## [1] 81  4
```

```r
## first six observations
head(kyphosis)
```

```
##   Kyphosis Age Number Start
## 1   absent  71      3     5
## 2   absent 158      3    14
## 3  present 128      4     5
## 4   absent   2      5     1
## 5   absent   1      4    15
## 6   absent   1      2    16
```

```r
## structure of data
str(kyphosis)
```

```
## 'data.frame':    81 obs. of  4 variables:
##  $ Kyphosis: Factor w/ 2 levels "absent","present": 1 1 2 1 1 1 1 1 1 2 ...
##  $ Age     : int  71 158 128 2 1 1 61 37 113 59 ...
##  $ Number  : int  3 3 4 5 4 2 2 3 2 6 ...
##  $ Start   : int  5 14 5 1 15 16 17 16 16 12 ...
```

In these data, `kyphosis` is a factor indicator of whether kyphosis was present or absent; `age` represents age in months; `number` represents the number of vertebrae that were operated on; and `start` represents the number of the topmost vertebra operated on. To predict kyphosis as a function of these variables, we can use a classification and regression tree:

In the first model, we use the default tuning parameter values:

```
fit <- rpart(Kyphosis ~ Age + Number + Start, data = kyphosis,
    method = "class")
```

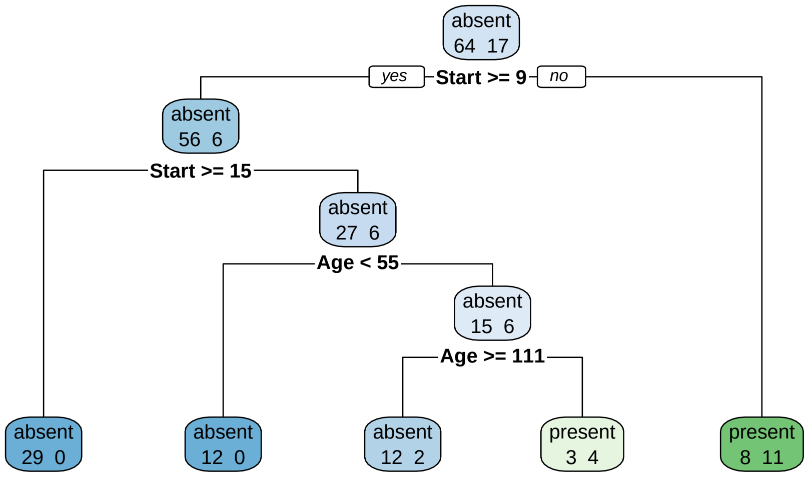


Figure 3: Classification and regression tree fit from the kyphosis data using default tuning parameters

This figure shows that there are two variables that predict kyphosis: which vertebra is the topmost vertebra (start) and age.[5] The most predictive is start: having surgery on a vertebra at or greater than the 9th from the top. If the child's surgery included vertebrae at or greater than the 9th from the top, the next most predictive is whether the surgery involved vertebrae at or greater than the 15th from the top. Among those whose surgery invovled a vertebrae at or greater than the 15th from the top, the next most predictive is age (<55 months) and age again (< 111 months).

[5] The figure was obtained using rpart.plot via `rpart.plot(fit, extra=1)`

This figure is effectively what is returned by classification and regression tree. However, this is not the **only** implementation. There are several tuning parameters one can change to slightly alter the tree. For the `rpart` function, the tuning parameters of importance are encoded in the `rpart.control` function, and include:

Parameter	Default	Interpretation
minsplit	20	minimum number of observations in a node for a split to be attempted
minbucket	round(minsplit/3)	minimum permitted number of observations in any terminal node
cp	0.01	complexity parameter, which is a function of the chosen impurity index. A threshold factor above which a split is deemed "unimportant".
xval	10	number of cross validations.
maxdepth	30	maximum depth of any node in the final tree.

Table 1: Select tuning parameters for the 'rpart' function in R. Source: R help.

We can explore what happens to the fit of the model when these parameters are changed. For example, we can change the complexity parameter from its default of 0.01 to 0.05, thus favoring a simpler model:

```r
fit <- rpart(Kyphosis ~ Age + Number + Start, data = kyphosis,
    method = "class", control = rpart.control(cp = 0.05))
```

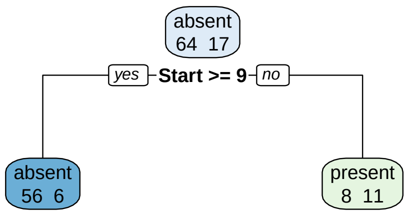


Figure 4: Classification and regression tree fit from the kyphosis data using a complexity parameter of 0.05

On the other hand, we can get extremely flexible, and fit a model with tuning parameters that are highly flexible:

```r
fit <- rpart(Kyphosis ~ Age + Number + Start, data = kyphosis,
    method = "class", control = rpart.control(cp = 0.001,
        minsplit = 5, minbucket = 4))
```

This much more flexible model leads (as expected) to a much richer tree. But this is not necessarily better. In fact, we are more than likely overfitting the data here.

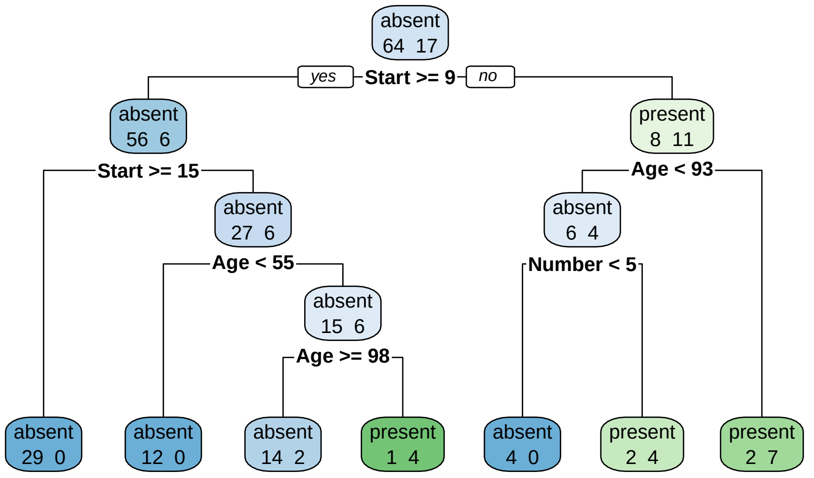


Figure 5: Classification and regression tree fit from the kyphosis data using tuning parameters that yield a highly flexible model

The number of options that we can modify can keep us very busy. But a more important question is which specification of the tuning parameters is better? There is actually no way to tell. However, we will see how stacking (Super Learner) can help us answer this question.

CART Performance

When they were introduced, it was quickly realized that classification and regression trees were a powerful tool for exploring data and making predictions. Relative to other prediction algorithms (e.g., principal component analysis, partial least squares), trees are not affected by transformations (e.g., centering and scaling) of the predictors. They are immune to the effects of predictor outliers. Unlike logistic regression, they do not require complex link functions for binary outcomes. Unlike neural networks, the relation between the predictors and the outcome is relatively easy to understand. Finally, trees are easy to use because they can be directly applied to data without requiring complex transformations to optimize predictive validity (Hastie, Tibshirani, and Friedman 2009, (p352)).

However, it was also soon realized that trees suffered from an important problem: relatively low accuracy.[6] The problem is that trees tend to grow too deep and learn highly irregular patterns: they overfit (i.e., undersmooth) the data. In other words, CART tends to have low bias, but very high variance.

[6] For classification trees, accuracy is captured via a confusion matrix. For regression, accuracy is defined as mean squared error.

This problem results from the fact that trees tend to encode highly irregular features of the data under a wide range of tuning parameters. We can visualize this effect by fitting a CART model to the NHEFS data under the default tuning parameters and a moderate decrease in the complexity parameter. In this classification tree, we are trying to predict high blood pressure (`high_BP`) as a function of `qsmk`, `sex`, `age`, `race`, `income`, `marital`, `school`, `asthma`, `bronch`, `diabetes`.

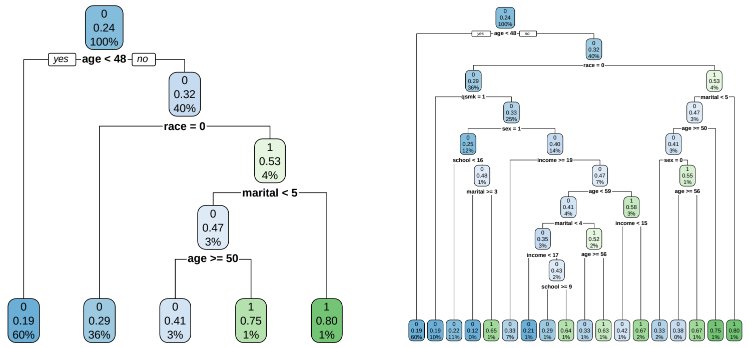


Figure 6: Classification and Regression Tree Algorithms fit to the NHEFS Data with a complexity parameter of 0.01 (left panel) and 0.0075 (right panel).

As displayed in Figure 1, as soon as we lower the complexity parameter from its default by even a little amount, the number of "important" features captured by the CART tree jumps from 3 to 14. Indeed, it is very easy to create highly complex trees that fit the data at hand nearly perfectly. But in doing so, we lose the ability to make scientific generalizations from our models, which is primarily why we would do the analysis.

Soon after CART was introduced, several researchers began to make modifications to the algorithm to make it more robust to overfitting. The two that were shown to have the most important impacts on the performance of a tree were bootstrap aggregating (bagging) and the random subspace method.

Bootstrap Aggregation

Bootstrap aggregation, or bagging, is a simple technique meant to reduce the variability of a flexibly specified classification and regression tree. The basic premise starts with a bootstrap sample of the original data. For example, suppose we let X represent all (outcome and covariates) of the NHEFS data that we are using to fit a CART model. Suppose further that we let $\text{Tree}(X)$ represent the tree fit to these data (i.e., as in Figure 1, left panel). We can take B bootstrap samples from X, which we denote $X_1, X_2, \ldots X_B$, which gives us a total of B trees. This multiplicity of trees is why we call the approach random forests.

Going back to the NHEFS data example, we can take nine bootstrap resamples and fit a flexible CART model to each resample, giving us a total of nine trees:

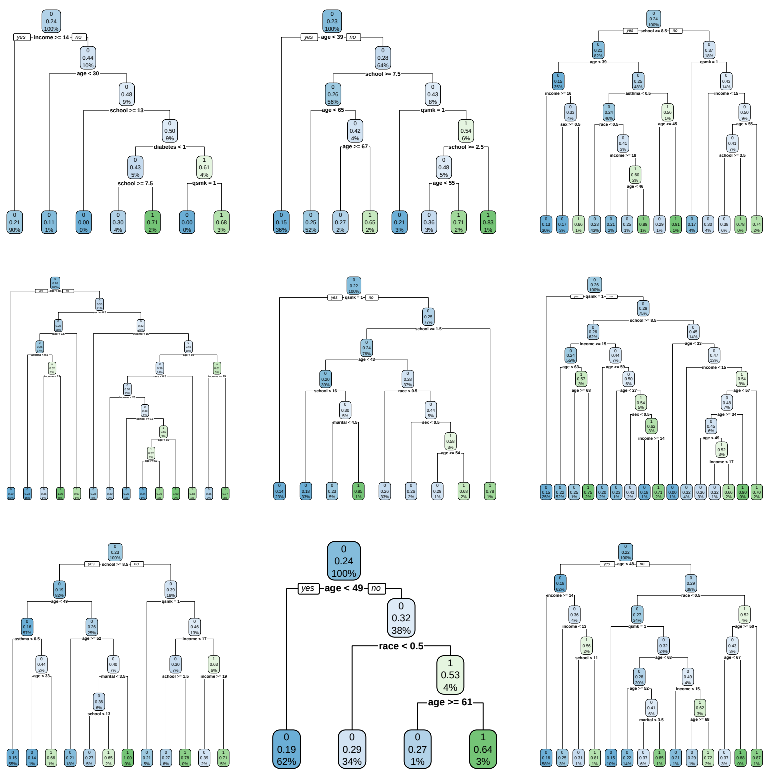


Figure 7: Nine Classification and Regression Tree Algorithms fit to bootstrap resamples of the NHEFS Data under default tuning parameters.

Note the variation in the structure of these trees, which is an indication of how sensitive CART is to (here relatively minor) pertubartions in the data. This variation is not of direct interest, in the sense that we do not use the bootstrap to quantify confidence intervals. Rather, random forests rely on the bootsrtap to obtain numerous trees, which are then averaged over to get a single prediction. By taking the average of several trees fit to the same data, bagging is better able to deal with the volatility inherent in CART to yield a better performing algorithm.

Mathematically, we can write that a random forest is simply an aggregation of bootstrapped trees:

$$\text{RF} = \frac{1}{B}\sum_{b=1}^{B}\text{Tree}(X_b)$$

It is the averaging of numerous trees that results in a better performance of the random forest over CART. To give a more precise illustration of the mechanism here, consider the trees in Figure 2. Consider further that we might be interested in predicting the probability of high blood pressure if everyone quit smoking. From each tree, we would obtain the predicted probability of high blood pressure if `qsmk=1`, giving us a total of nine predictions, from which we would compute the average.[7] For a classifiation algorithm, the final prediction (i.e., whether a person has high BP or not) will be based on the "majority vote" of all Another benefit of using the bootstrap in this context is that we have a way of estimating the error in the algorithm. This error is calculated as follows:

[7] Note that for some trees, the prediction doesn't change since `qsmk` is not in the tree.

Step 1: Compare the observations in each of the B bootstrapped trees to the observations in the original sample. From all B trees, select only those that do not have the first observation in the original sample. These trees are referred to as "out of bag" trees.

Step 2: Predict the event using only the out of bag trees.

Step 3: Take the difference of the out-of-bag prediction and the actual outcome for the first observation.

Step 4: Repeat for all N.

Step 5: Take the mean of all N errors.

The end result from this process is referred to as the out-of-bag error. But why is it important? First, the OOB error gives an idea of how "close" the forest predictions are to the actual predictions. Second, Brieman (1983) showed that the out-of-bag error is as accurate as doing a formal analysis with a testing and validation set (e.g., cross-validation, which we will discuss later). As a result, one need not actually use formal testing/validation with random forests.

Random Subspace Selection

A second feature of random forests is the use of random subspace selection, which works as follows: Instead of using **all available covariates** when fitting each tree to the bootstrapped data, the algorithm randomly chooses only a subset of the predictors. What this does is it reduces the correlation between each of the bootstrapped trees, thus allowing us to compute error rates without accounting for violations of the IID assumption.[8] It also tends to reduce the impact of a variable that is so highly predictive that it would force the tree to overfit the data whenever it is included in the tree. Thus, by repeated the fitting process a large number of times the overfitting induced by these highly predictive variables is minimized.

[8] Recall, the independent and identically distributed (IID) assumption is required for the validity of most estimation methods, including the mean estimator, which is used to compute the OOB error.

Random Forests via 'ranger'

In R, there are several implementations of the random forest algorithm for a wide range of data. These include `RandomForest` and `ranger`, which implement more "classical" random forests (i.e., for regression and classification), as well as `quantregForest` and `randomForestSRC`, which implement versions that predict quantiles of a distribution (e.g, survival curves), and not just the expected value.

Arguably, (IMO) the `ranger` package is the best of these given its speed and simplicity. To install `ranger`, we use the standard approach:

```
install.packages("ranger", repos = "http://lib.stat.cmu.edu/R/CRAN/")

##
## The downloaded binary packages are in
##  /var/folders/3s/wfps7lfd62x48nbyfs_v_yh80000gp/T//RtmpDEmkCF/downloaded_packages
```

```
library(ranger)
```

We will use this function to predict whether an individual case will have high blood pressure in the NHEFS data:[9]

[9] These are the same data we used for the g Computation example

```
a <- fread("../data/gComp_example.csv")
names(a)
```

```
##  [1] "qsmk"               "smkintensity82_71" "smokeintensity"
##  [4] "active"             "exercise"          "wt82_71"
##  [7] "sbp"                "dbp"               "hbp"
## [10] "ht"                 "hbpmed"            "sex"
## [13] "age"                "hf"                "race"
## [16] "income"             "marital"           "school"
## [19] "asthma"             "bronch"            "diabetes"
## [22] "id"                 "map"               "high_BP"
```

```
a <- a %>%
    select(high_BP, qsmk, sex, age, race, income, marital,
        school, asthma, bronch, diabetes) %>%
    na.omit()
a$qsmk <- as.factor(a$qsmk)
dim(a)
```

```
## [1] 2000   11
```

In the `ranger` function, there are several options that we should be aware of. These options are effectively tuning parameters that will change the behaviour of the algorithm. Among them include:

To implement `ranger`, we can use the following code:

```
rF <- ranger(as.factor(high_BP) ~ ., data = a)
```

Note that the `y ~ .` notation in the above code is telling `R` to regress `y` against all variables in the dataset.[10] Note also that for this to work, **we had to code our outcome as a factor.** If we don't the `ranger` function will run a regression, and not classification model. If we call the ranger object, we get important information on the algorithm:

[10] note also that we can do this for all functions in R that take a formula, including `lm()` and `glm()`.

Table 2: Select tuning parameters for the 'ranger' function in R. Source: R help.

Parameter	Default	Interpretation
num.trees	500	Number of bootstraps from the data; the size of the forest
mtry	round($\sqrt{p}$)	Number of variables randomly selected for subspace (where p is total number)
importance	none	Whether a variable importance measure will be computed.
splitrule	gini/variance	Impurity measure used to determine whether a split should occur
min.node.size	1	Smallest number of observations in each node.
probability	F	For binary outcomes, determines whether probability or classificaiton should be computed.
quantreg	F	For continuous outcomes, determines whether quantile forest should be fit.

```
rF
```

```
## Ranger result
##
## Call:
##  ranger(as.factor(high_BP) ~ ., data = a)
##
## Type:                             Classification
## Number of trees:                  500
## Sample size:                      2000
## Number of independent variables:  10
## Mtry:                             3
## Target node size:                 1
## Variable importance mode:         none
## Splitrule:                        gini
## OOB prediction error:             26.85 %
```

From this object, we can get predicted values and compare them to the actual values:

```
prediction <- predict(rF, data = a)
names(prediction)
```

```
## [1] "predictions"               "num.trees"
## [3] "num.independent.variables" "num.samples"
## [5] "treetype"
```

```
prediction <- prediction$predictions
table(a$high_BP, prediction)
```

```
##    prediction
##        0    1
##   0 1491   31
##   1  282  196
```

Stacking via Super Learner

All of the examples above represent implementations of single algorithms under a specific set of tuning parameters. However, selecting appropriate set of tuning parameters can make or break the performance of an algorithm. Furthermore, there will always be the question of which particular algorithm one should use?

In the early 1990s, Wolpert developed an approach to combine several `lower-level'' predictive algorithms into a`higher-level'' model with the goal of increasing predictive performance (Wolpert 1992). He termed the approach `stacked generalization'', which later became known as`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 (Leo Breiman 1996). More recently, van der Laan and colleagues proved that, in large samples, the algorithm will perform at least as well as the best individual predictor included in the ensemble [(**vanderLaan2003?**);vanderLaan2006a;vanderLaan2007]. Therefore, choosing a large library of diverse algorithms will enhance performance, while creating the best weighted combination of candidate algorithms will further improve performance. Here, best is defined in terms of a bounded loss function, which allows us to define and quantify how well a given algorithm (e.g., regression, machine learning) performs at predicting or explaining the data. Over-fitting is avoided with V-fold cross-validation. In this context, the term "Super Learner" was coined.

Super Learner has tremendous potential for improving the quality of predictions in applied health sciences, and minimizing the extent to which empirical findings rely on parametric modeling assumptions. The following introduction is taken from Naimi and Balzer (2018). Full `R` code for these examples is available at GitHub.

Example 1: Dose-Response Curve

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

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

where $\mathbb{I}()$ denotes the indicator function evaluating to 1 if the argument is true (zero otherwise), and ϵ 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 @ref{F1}. We now manually demonstrate how 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 (**Milborrow2017?**). 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

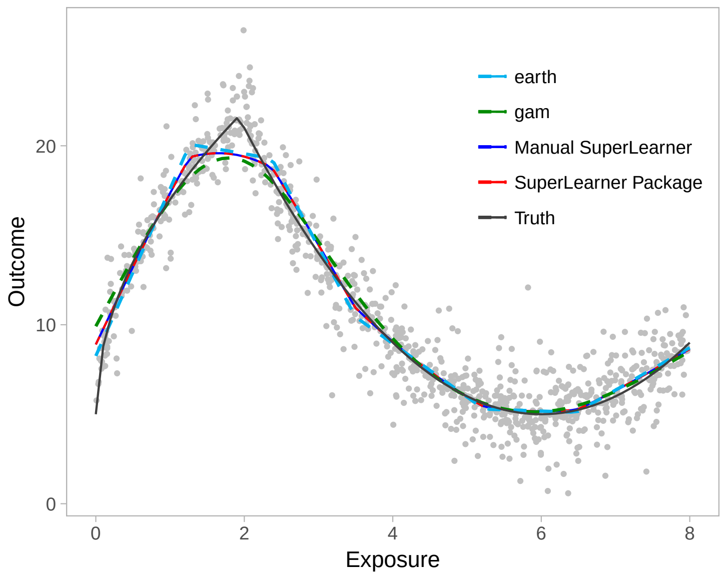


Figure 8: Dose-response curves for the relation between our simulated continuous exposure and continuous outcome in Example 1. The black line represents the true curve, while the red and blue lines represent curves estimated with the programmed Super Learner package in R, and the manually coded Super Learner. Light blue and green curves show the fits from the level-zero algorithms, exttteearth and extttgam respectively. Gray dots represent observed data-points.

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, Tibshirani, and Friedman 2009)(section 2.4). To estimate this expected loss, called the "risk'', we use V-fold cross-validation with $V = 5$ folds. Cross-validation is a sample splitting technique to assess estimator performance using data drawn from the same distribution. As detailed below, each candidate estimator is fit on a portion of the data and used to predict the outcomes for observations that were not used in the training process. As a result, cross-validation helps us avoid poor out-of-sample predictions and gives us a more honest measure of performance. Without this crucial step, we risk generating a predictive algorithm that performs flawlessly for the data at hand, but poorly for another sample. To implement Super Learner, we:

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 $v = \{1, \dots, 5\}$,

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 2.58 for `gam` and 2.48 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:

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

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 $\hat{\alpha}_1 = 0.387$ for `gam` and $\hat{\alpha}_2 = 0.613$ for `earth`. Therefore, both generalized additive models and regression splines each contribute approximately 40% and 60% 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` on 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:

$$\hat{Y}_{\text{SL}} = 0.387\hat{Y}_{\text{gam}} + 0.613\hat{Y}_{\text{earth}} \tag{3}$$

where $\hat{Y}_{\text{SL}}$ denotes our final Super Learner predicted outcome. The resulting predictions are shown in blue in Figure reffig:F1. For comparison, the predictions from the `R` package `SuperLearner`-v2.0-23-9000 are shown in red [@Polley2016] while the predictions from `gam` and `earth` are shown in light blue and green, respectively.

R Code for Example 1: Dose-Response Curve

```
library(SuperLearner)
library(data.table)
library(nnls)
library(rmutil)
library(ranger)
library(xgboost)
library(splines)
library(Matrix)
library(ggplot2)
library(xtable)
```

```
# EXAMPLE 1 set the seed for reproducibility
set.seed(12345)

# 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

# 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)

learners = create.Learner("SL.gam", params = list(deg.gam = 5))

learners

## $grid
## NULL
##
```

```
## $names
## [1] "SL.gam_1"
##
## $base_learner
## [1] "SL.gam"
##
## $params
## $params$deg.gam
## [1] 5
```

```
# create.SL.gam(deg.gam = 5)
# create.SL.nnet(size=4)

# Specifying the SuperLearner library of
# candidate algorithms
sl.lib <- c(learners$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 2.580372 0.3867099
## SL.earth_All 2.476462 0.6132901
```

Example 2: Binary Classification

Our second example is predicting the occurrence of a binary outcome with goal of maximizing the area under the receiver operating characteristic (ROC) curve, which shows the balance between sensitivity and specificity for varying discrimination thresholds. For 10,000 observations, we generate five covariates $\mathbf{X} = \{X_1, \ldots, X_5\}$ by drawing from a multivariate normal distribution and then generate the outcome Y by drawing from a Bernoulli distribution with probability

$$\mathbb{P}(Y = 1 \mid \mathbf{X}) = 1 - \text{expit}\,\{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 + \beta_5 X_5 \\ + \boldsymbol{\beta}_1 (X_1 : X_5)_1 + \boldsymbol{\beta}_2 (X_1 : X_5)_2\},$$

where $\text{expit}(\bullet) = (1 + \exp[-\bullet])^{-1}$; and $(X_1 : X_5)_1$ and $(X_1 : X_5)_2$ denote all first and second order interactions between $X_1, \ldots, X_5$, respectively; and $\boldsymbol{\beta}_1$ and $\boldsymbol{\beta}_2$ denote a set of parameters, one for each interaction. In total, there were 25 terms plus the intercept in this model. The intercept was set to $\beta_0 = 2$, while all other parameters were drawn from a uniform distribution bounded by 0 and 1.

Our library of candidate algorithms consists of Bayesian GLMs (`bayesglm`), implemented via the `arm` package (v 1.9-3) (**Gelman2016?**), and multivariate polynomial adaptive regression splines (`polymars`) via the `polspline` package (**Kooperberg2015?**). Our objective is to generate an algorithm that correctly classifies individuals given covariates. Correct classification is a function of both sensitivity and specificity. Thus, to measure the performance of these level-zero algorithms and build the meta-learner, we use the rank loss function. Instead of minimizing mean squared error, the rank loss function aims to maximize the area under the ROC curve (a function of both sensitivity and specificity), thus optimizing the algorithms ability to correctly classify observations (**Ledell2016?**). We again use 5-fold cross-validation to obtain an honest measure of performance and avoid over-fitting.

Steps 1-3. Implementation of Steps 1-3 are analogous to the previous example. In place of `gam` and `earth`, we use `bayesglm`, and `polymars`. In place of the L-2 loss function, we use the rank loss. Specifically, for each validation set and each algorithm, we estimate the sensitivity, specificity, and then compute the area under the ROC curve (AUC). The expected loss (i.e., "risk") can then be computed as $1-$AUC. Averaging the estimated risks across the folds yields one measure of performance for each algorithm. In our simple example, the cross-validated risk estimates are 0.122 for `bayesglm` and 0.114 for `polymars`.

As before, we could simply select the model with lowest cross-validated risk estimate (here, `polymars`). Instead in Steps 4-5, we combine the resulting cross-validated predictions to generate the level-one learner.

Step 4. Let $\hat{Y}_{\text{bglm-cv}}$ and $\hat{Y}_{\text{pm-cv}}$ denote the cross-validated predictions from `bayesglm` and `polymars`, respectively. To calculate the contribution of each candidate algorithm to the final Super Learner prediction, use the rank loss function to define "optimal" as the convex combination that maximizes the AUC. Then estimate the α parameters in the following constrained regression

$$\mathbb{P}(Y=1|\hat{Y}_{\text{bglm-cv}}, \hat{Y}_{\text{pm-cv}}) = \alpha_1\hat{Y}_{\text{bglm-cv}} + \alpha_2\hat{Y}_{\text{pm-cv}}, \text{ where } \alpha_1 \geq 0; \alpha_2 \geq 0; \text{ and } \sum_{k=1}^{2}\alpha_k = 1 \tag{4}$$

such that $(1-AUC)$ is minimized when comparing Super Learner predicted probabilities to the observed outcomes. These parameters can be estimated with a tailored optimization function, such as `optim` in `R` or `proc nlmixed` in SAS. In our simple example, the coefficient values are $\hat{\alpha}_1 = 0.223$ for `bayesglm` and $\hat{\alpha}_2 = 0.776$ for `polymars`.

Step 5. As before, the final step is to use the above coefficients to generate the Super Learner. To do so, refit `bayesglm` and `polymars` on the entire sample and denote the predicted outcomes as $\hat{Y}_{\texttt{bglm}}$ and

$\hat{Y}_{\texttt{pm}}$ Then combine these predictions with the estimated weights:

$$\mathbb{P}(Y = 1 \mid \hat{Y}_{\text{bglm}}, \hat{Y}_{\text{pm}}) = 0.223\hat{Y}_{\text{bglm}} + 0.776\hat{Y}_{\text{pm}} \tag{5}$$

These predictions can then be used to compute the ROC curve displayed in blue in Figure
refF2. For comparison the predictions from the `R` package `SuperLearner` are shown in red.

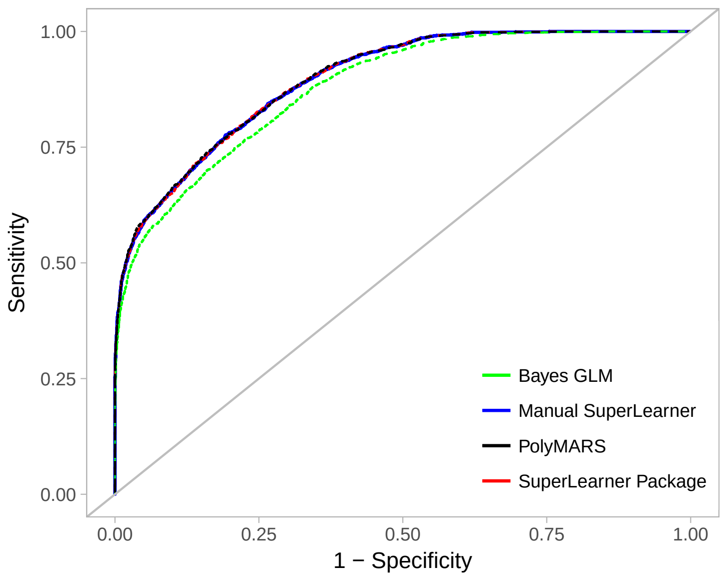


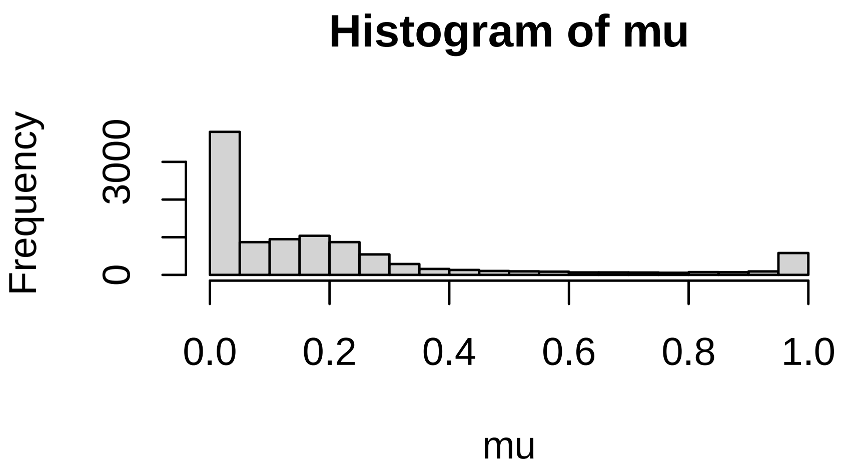
Figure 9: Receiver operating characteristic curves displaying the ability of 5 simulated exposures to predict the simulated outcome in Example 2. Blue line represents the curve obtained from the Super Learner package. Red dotted line represents curve obtained from manually coded Super Learner. The green line represents the curve from level-zero Bayes GLM algorithm, and the black line represents the curve from PolyMARS.

R Code for Example 2: Binary Classification

```
library(SuperLearner)
library(data.table)
library(nnls)
library(mvtnorm)
library(ranger)
library(xgboost)
library(splines)
library(Matrix)
library(ggplot2)
library(xtable)
library(pROC)
library(here)

# EXAMPLE 2
set.seed(123)
n = 10000
sigma <- abs(matrix(runif(25, 0, 1), ncol = 5))
sigma <- forceSymmetric(sigma)
sigma <- as.matrix(nearPD(sigma)$mat)
x <- rmvnorm(n, mean = c(0, 0.25, 0.15, 0, 0.1), sigma = sigma)
modelMat <- model.matrix(as.formula(~(x[, 1] + x[,
    2] + x[, 3] + x[, 4] + x[, 5])^3))
beta <- runif(ncol(modelMat) - 1, 0, 1)
beta <- c(2, beta)  # setting intercept
mu <- 1 - plogis(modelMat %*% beta)  # true underlying risk of the outcome
y <- rbinom(n, 1, mu)

hist(mu)
```



```
mean(y)

## [1] 0.2091

x <- data.frame(x)
D <- data.frame(x, y)

# 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]])
```

```
##            X1          X2         X3         X4         X5 y
## 1   0.6889718  0.56396790 -0.1772133  1.3860987  1.2951780 0
## 6  -0.4407854  0.52801706 -0.8487474  0.3946236  0.2617834 1
## 11  2.1482189 -0.30732404  1.3160864  1.7300411  1.6806760 0
## 16 -0.2006415  0.08236686  0.2913948 -0.4763249 -0.3242203 0
## 21 -0.3006780  0.07211962  0.7107014 -0.9971198 -0.6546590 1
## 26 -0.2246566 -0.08258837  0.5089715 -0.7729345 -0.5747198 0
```

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

```
##             X1         X2         X3         X4          X5 y
## 2  -0.54517808 -0.5654673 -0.6801941 -0.3488104 -0.71118281 0
## 7  -0.19328078  1.0828619 -0.2843008  0.4593875  0.66758582 0
## 12 -0.32548369  0.4190843  0.1314735 -0.3700632 -0.15779494 0
## 17  0.47131455  0.7012344  1.2437019 -0.1398840  0.43968892 0
## 22  0.49734548  1.2332820  0.9727911  0.3570545  0.97595847 0
## 27  0.07816149  0.1639731  0.5595036 -0.2939324 -0.06762245 0
```

```
#-------------------------------------------------------------------------------
# Fit using the SuperLearner Package
#-------------------------------------------------------------------------------
# Specify the outcome-for-prediction (y), the
# predictors (x), \t\tfamily (for a binary
# outcome), measure of performance (1-AUC),
# \t\tthe library (sl.lib), and number of folds
sl.lib <- c("SL.bayesglm", "SL.polymars")
fitY <- SuperLearner(Y = y, X = x, family = "binomial",
    method = "method.AUC", SL.library = sl.lib, cvControl = list(V = folds),
    verbose = F)
```

```
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## step half ouch...
## warning - model size was reduced
## step half ouch...
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## warning - model size was reduced
## step half ouch...
## step half ouch...
## step half ouch...
## step half ouch...
## warning - model size was reduced
## step half ouch...
## step half ouch...
## step half ouch...
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## warning - model size was reduced
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## step half ouch...
## step half ouch...
## step half ouch...
## warning - model size was reduced
## warning - model size was reduced
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## warning - model size was reduced
## step half ouch...
## step half ouch...
## warning - model size was reduced
## warning - model size was reduced
## step half ouch...
## step half ouch...
## warning - model size was reduced
## step half ouch...
## step half ouch...
## warning - model size was reduced
## step half ouch...
## step half ouch...
```

```
# Note: for rare binary outcomes, consider using
# the stratifyCV option to \t\tmaintain roughly
# the same # of outcomes per fold View the
# output: 'Risk' column returns the CV estimates
# of (1-AUC) \t\t'Coef' column gives the weights
# for the final SuperLearner (meta-learner)
fitY
```

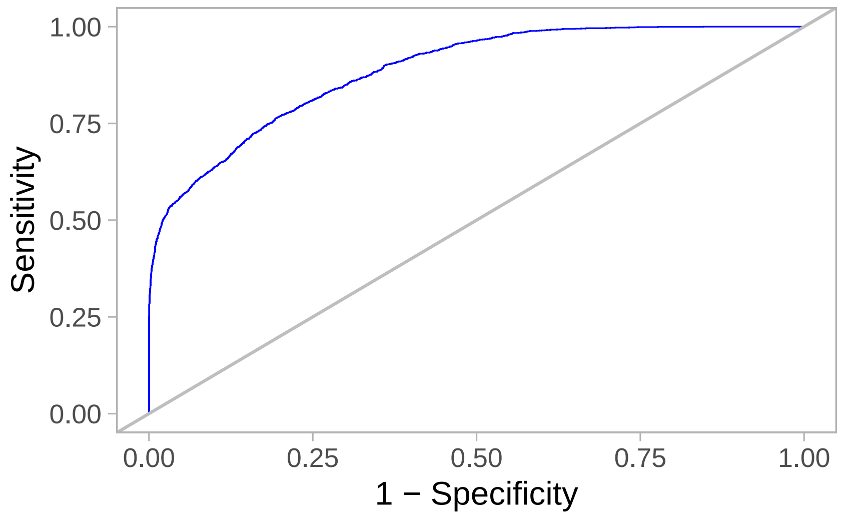
```
##
## Call:
## SuperLearner(Y = y, X = x, family = "binomial", SL.library = sl.lib, method = "method.AUC",
##     verbose = F, cvControl = list(V = folds))
##
##
##                       Risk      Coef
## SL.bayesglm_All 0.1219818 0.2839194
## SL.polymars_All 0.1138237 0.7160806
```

```
# Obtain the predicted probability of the outcome
# from SL
y_pred <- predict(fitY, onlySL = T)$pred
p <- data.frame(y = y, y_pred = y_pred)
head(p)
```

```
##   y     y_pred
## 1 0 0.02409601
## 2 0 0.27270746
## 3 0 0.26399186
## 4 0 0.56002787
## 5 1 0.87975844
## 6 1 0.29587210

# Use the roc() function to obtain measures of
# performance for binary classification
a <- roc(p$y, p$y_pred, direction = "auto")
# To plot the ROC curve, we need the sensitivity
# and specificity
C <- data.frame(sens = a$sensitivities, spec = a$specificities)

ggplot() + geom_step(data = C, aes(1 - spec, sens),
    color = "blue", size = 0.25) + theme_light() +
    theme(panel.grid.major = element_blank(), panel.grid.minor = element_blank()) +
    labs(x = "1 - Specificity", y = "Sensitivity") +
    geom_abline(intercept = 0, slope = 1, col = "gray")
```



Discussion

Stacked generalizations, notably Super Learner, are fast becoming an important part of the epidemiologic toolkit. There are several challenges in understanding precisely what stacking is, and how it is implemented. These challenges include the use of complex machine learning algorithms as candidate algorithms, and the actual process by which the resulting predictions are combined into the meta-learner. Here, we sought to clarify the latter aspect, namely, implementation of the Super Learner algorithm.

Several considerations (including strengths and limitations) merit attention. First, any number of loss functions could be chosen determine the optimal combination of algorithm-specific predictions. The choice should be based on the objective of the analysis. The target parameter depends on the loss function choice, but various loss functions can identify the same target parameter as minimizer of its risk. Our examples demonstrated the use of the L-2 loss to minimize prediction error when estimating a dose-response curve, and the rank loss to maximize AUC when developing a binary classifier. Other loss functions could be entertained. For example, Zheng *et al.* recently aimed to simultaneously maximize sensitivity and minimize rate of false positive predictions with application to identify high-risk individuals for pre-exposure prophylaxis (Zheng et al. 2018).

Second, a wide array of candidate algorithms can be included in the Super Learner library. We recommend including standard parametric modeling approaches (e.g., generalized linear models, simple mean, simple median) and more complex data-adaptive methods (e.g., penalized regression, and tree- or kernel-based methods). Often, the performance of data-adaptive methods depends on how tuning parameters are specified. Indeed, tuning a machine learning algorithm is a critical step in optimizing performance, and is straightforward with Super Learner. One need only include the same algorithm in the Super Learner library multiple times, with different tuning parameter values for each candidate entry (e.g., extreme gradient boosting with varying shrinkage rates). We refer the readers to Polley et al (Polley et al. 2016) and Kennedy (**Kennedy2018gh?**) for practical demonstrations of how to deal with tuning parameters.

Third, a critically important part of Super Learning is the use of V-fold cross-validation. However, the optimal choice of V is not always clear. At one end of the extreme, leave-one-out cross validation chooses $V = N$, but is subject to potentially high variance and low bias. On the other end, 2-fold cross validation is subject to potentially low variance and high bias. A general rule of thumb is to use

$V = 10$ (**Kohavi1995?**) Though common, this number will not optimize performance in all settings (Zhang et al. 2015) In general, we recommend increasing the number of folds (V) as sample size n decreases. We also note that other cross-validation schemes (beyond V-fold) are available, even in settings with dependent data.

While Super Learner with a rich set of candidates represents a versatile tool, important limitations should be noted.

First, no algorithm (Super Learner or any other machine learning method) should be used to replace careful thinking of the underlying causal structure. Super Learner cannot distinguish between confounders, instrumental variables, mediators, colliders, and the exposure. Instead, the goal of Super Learner is to do the best (as specified through the loss function) possible job predicting the outcome (or exposure) given the inputted variables. Super Learner can, however, minimize assumptions regarding the nature of the relation between the covariates, the exposure, and the outcome. These assumptions are often present in the form of selected link functions, the absence of interactions, and the shape of particular dose-response relations (e.g., linear, polynomial). Nonetheless, if such functional form information is available, this knowledge can readily be included in Super Learner. For example, physicians might prescribe treatments according to a specific algorithm, and this knowledge can be included as a candidate algorithm (e.g. a parametric regression) in Super Learner's library.

Secondly, Super Learner is a"black box' ' algorithm; so the exact contribution of each covariate to prediction is unclear. These contributions can be revealed by estimating variable importance measures, which quantify the marginal association between each predictor and the outcome after adjusting for the others. Nevertheless, a large predictor contribution may be the result of direct causation, unmeasured confounding, collider stratification, reverse causation, or some other mechanism.

The goal of prediction is distinct from causal effect estimation, but prediction is often an intermediate step in estimating causal effects (**vanderLaan2011?**). Indeed, some researchers have advocated for the use of data-adaptive methods, including Super Learner, for effect estimation via singly-robust methods, depending on estimation of either the conditional mean outcome or the propensity score (McCaffrey et al. 2013; Westreich, Lessler, and Funk 2010; Lee, Lessler, and Stuart 2010; Snowden, Rose, and Mortimer 2011; Westreich et al. 2015; Pirracchio and Carone 2016; Moodie and Stephens 2017). While flexible algorithms can reduce the risk of bias due to regression model misspecification, a serious concern is that the use of data-adaptive algorithms in this context can result in invalid statistical inference (i.e. misleading confidence intervals). Specifically, there is no theory to support the resulting estimator is asymptotically linear (i.e., consistent and asymptotically normal), which is required to obtain, for example, centered confidence intervals with nominal coverage properties.

References

Ayres, I. 2008. *Super Crunchers: Why Thinking-by-Numbers Is the New Way to Be Smart*. Bantam Books.

Berk, Richard A. 2008. *Statistical Learning from a Regression Perspective*. New York, NY: Springer.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J Stone. 1983. *Classification and Regression Trees.* Belmont, Ca: Wadsworth.

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

Brieman, L. 1983. "Out-of-Bag Estimation." ftp://ftp.stat.berkeley.edu/pub/users/breiman/OOBestimation: University of California, Berkeley.

Culhane, Jennifer F, and Robert L Goldenberg. 2011. "Racial Disparities in Preterm Birth." *Semin Perinatol* 35 (4): 234–39. `https://doi.org/10.1053/j.semperi.2011.02.020`.

Fedrick, J. 1976. "Antenatal Identification of Women at High Risk of Spontaneous Pre-Term Birth." *Br J Obstet Gynaecol* 83 (5): 351–54.

Green, Nancy S, Karla Damus, Joe Leigh Simpson, Jay Iams, E Albert Reece, Calvin J Hobel, Irwin R Merkatz, Michael F Greene, and Richard H Schwarz. 2005. "Research Agenda for Preterm Birth: Recommendations from the March of Dimes." *Am J Obstet Gynecol* 193 (3 Pt 1): 626–35. `https://doi.org/10.1016/j.ajog.2005.02.106`.

Hamilton, Brady E., Joyce A. Martin, Michelle J. K. Osterman, and Sally C. Curtin. 2014. "Births: Preliminary Data for 2013." *National Vital Statistics Reports* 63 (2).

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

Hastie, Trevor, Robert Tibshirani, and Jerome H. Friedman. 2009.

The Elements of Statistical Learning: Data Mining, Inference, and Prediction. New York, NY: Springer.

Honest, Honest, Chris J Hyde, and Khalid S Khan. 2012. "Prediction of Spontaneous Preterm Birth: No Good Test for Predicting a Spontaneous Preterm Birth." *Curr Opin Obstet Gynecol* 24 (6): 422–33.

Kuhn, M., and K. Johnson. 2018. *Applied Predictive Modeling*. Springer New York. `https://books.google.com/books?id=3ZqtzQEACAAJ`.

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

LeDell, Erin, Mark J van der Laan, and Maya Petersen. 2016. "AUC-Maximizing Ensembles Through Metalearning." *Int J Biostat* 12 (1): 203–18.

Lee, Brian K., Justin Lessler, and Elizabeth A. Stuart. 2010. "Improving Propensity Score Weighting Using Machine Learning." *Stat Med* 29 (3): 337–46.

McCaffrey, Daniel F, Beth Ann Griffin, Daniel Almirall, Mary Ellen Slaughter, Rajeev Ramchand, and Lane F Burgette. 2013. "A Tutorial on Propensity Score Estimation for Multiple Treatments Using Generalized Boosted Models." *Stat Med* 32 (19): 3388–3414. `https://doi.org/10.1002/sim.5753`.

Meehl, P. 2013. *Clinical Versus Statistical Prediction: A Theoretical Analysis and a Review of the Evidence*. Echo Point Books; Media. `https://books.google.com/books?id=Ix6HmAEACAAJ`.

Meehl, P E. 1986. "Causes and Effects of My Disturbing Little Book." *J Pers Assess* 50 (3): 370–75.

Moodie, Erica E. M., and David A. Stephens. 2017. "Treatment Prediction, Balance, and Propensity Score Adjustment." *Epidemiology* 28 (5). `http://journals.lww.com/epidem/Fulltext/2017/09000/Treatment_Prediction,_Balance,_and_Propensity.24.aspx`.

Moster, Dag, Rolv Terje Lie, and Trond Markestad. 2008. "Long-Term Medical and Social Consequences of Preterm Birth." *N Engl J Med* 359 (3): 262–73.

Mullainathan, Sendhil, and Jann Spiess. 2017. "Machine Learning:

An Applied Econometric Approach." *Journal of Economic Perspectives* 31 (2): 87–106.

Naimi, Ashley I, and Laura B Balzer. 2018. "Stacked Generalization: An Introduction to Super Learning." *Eur J Epidemiol* 33 (5): 459–64.

Pirracchio, Romain, and Marco Carone. 2016. "The Balance Super Learner: A Robust Adaptation of the Super Learner to Improve Estimation of the Average Treatment Effect in the Treated Based on Propensity Score Matching." *Stat Methods Med Res*, December, 962280216682055. `https://doi.org/10.1177/0962280216682055`.

Polley, Eric, Erin LeDell, Chris Kennedy, and Mark van der Laan. 2016. *SuperLearner: Super Learner Prediction*. `https://github.com/ecpolley/SuperLearner`.

Snowden, Jonathan M., Sherri Rose, and Kathleen M. Mortimer. 2011. "Implementation of g-Computation on a Simulated Data Set: Demonstration of a Causal Inference Technique." *Am J Epidemiol* 173 (7): 731–38.

Trønnes, Håvard, Allen J Wilcox, Rolv T Lie, Trond Markestad, and Dag Moster. 2014. "Risk of Cerebral Palsy in Relation to Pregnancy Disorders and Preterm Birth: A National Cohort Study." *Developmental Medicine & Child Neurology* 56 (8): 779–85.

Westreich, Daniel, Jessie K Edwards, Stephen R Cole, Robert W Platt, Sunni L Mumford, and Enrique F Schisterman. 2015. "Imputation Approaches for Potential Outcomes in Causal Inference." *International Journal of Epidemiology*, Published ahead of print July 25, 2015.

Westreich, Daniel, Justin Lessler, and Michele Jonsson Funk. 2010. "Propensity Score Estimation: Neural Networks, Support Vector Machines, Decision Trees (CART), and Meta-Classifiers as Alternatives to Logistic Regression." *J Clin Epidemiol* 63 (8): 826–33.

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

Zhang, Y Tara, Barbara A Laraia, Mahasin S Mujahid, Aracely Tamayo, Samuel D Blanchard, E Margaret Warton, N Maggi Kelly, et al. 2015. "Does Food Vendor Density Mediate the Association Between Neighborhood Deprivation and BMI?: A g-Computation Mediation Analysis." *Epidemiology* 26 (3): 344–52.

Zheng, Wenjing, Laura Balzer, Mark van der Laan, and Maya Petersen. 2018. "Constrained Binary Classification Using Ensemble Learning: An Application to Cost-Efficient Targeted PrEP Strategies." *Stat Med* 37 (2): 261–79.