Become a SuperLearner!

An Introduction to Ensemble Learning in `R`

Kat Hoffman

WCM Biostatistics Computing Club

January 19, 2021

Happy First Computing Club of 2021!

Today's presentation has two parts:

(I) A motivation and **overview** of **ensemble learning** methods for prediction

(II) A **step-by-step walkthrough** of one method of ensemble learning: **superlearning/stacking**

If you have to leave early... 😿

- Part II is covered in my blog post on superlearning
- The recording of this talk will be on the WCM Computing Club website

PART I:

An overview of Ensemble Learning

Ensemble Learning

- Used when the goal is optimizing **prediction** of an outcome
- The process of **combining multiple statistical learning models** with the goal of creating a final model that is **better than any individual model** would be by itself

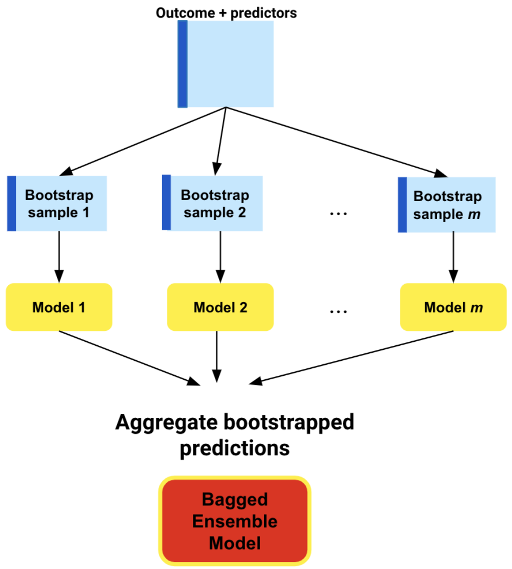
Jerusalem String Quartet. Image source: LiveAbout.com

- Three general categories:
  1. Bagging
  2. Boosting
  3. Stacking/Superlearning

Ensemble Learning Method #1: Bagging

Bootstrap Aggregating

1. Bootstrap the data (sample with replacement)
2. Fit a model on every bootstrapped data set
3. Aggregate (combine) the predictions



Bagging: A very simple example

- First, bootstrap the data (sample with replacement)

```r
library(dplyr)
# sample with replacement from the mtcars data set
boot_sample_1 <- sample_n(mtcars, size = nrow(mtcars), replace = T)
boot_sample_2 <- sample_n(mtcars, size = nrow(mtcars), replace = T)
boot_sample_3 <- sample_n(mtcars, size = nrow(mtcars), replace = T)
```

- Then, fit a model on every bootstrapped data set

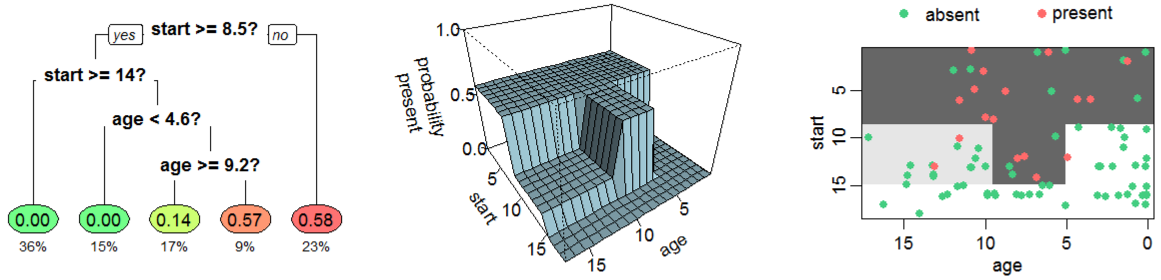
```r
# simplest model for the outcome `mpg`: mean model
model_1 <- mean(boot_sample_1$mpg)
model_2 <- mean(boot_sample_2$mpg)
model_3 <- mean(boot_sample_3$mpg)
```

- Finally, aggregate (combine) the predictions

```r
final_prediction <- mean(c(model_1, model_2, model_3))
```

Bagging in Practice

- Bagging is usually used to stabilize models with **high variance**, like **decision trees**
- **Decision tree**: simple, interpretable algorithm to sequentially split observations with the most homogenous responses. Each split is a "branch" and the terminal nodes, or "leaves", denote the outcome predictions



Decision tree to estimate the probability of kyphosis after surgery, given the age of the patient and the vertebra at which surgery was started.
Image source: Wikipedia

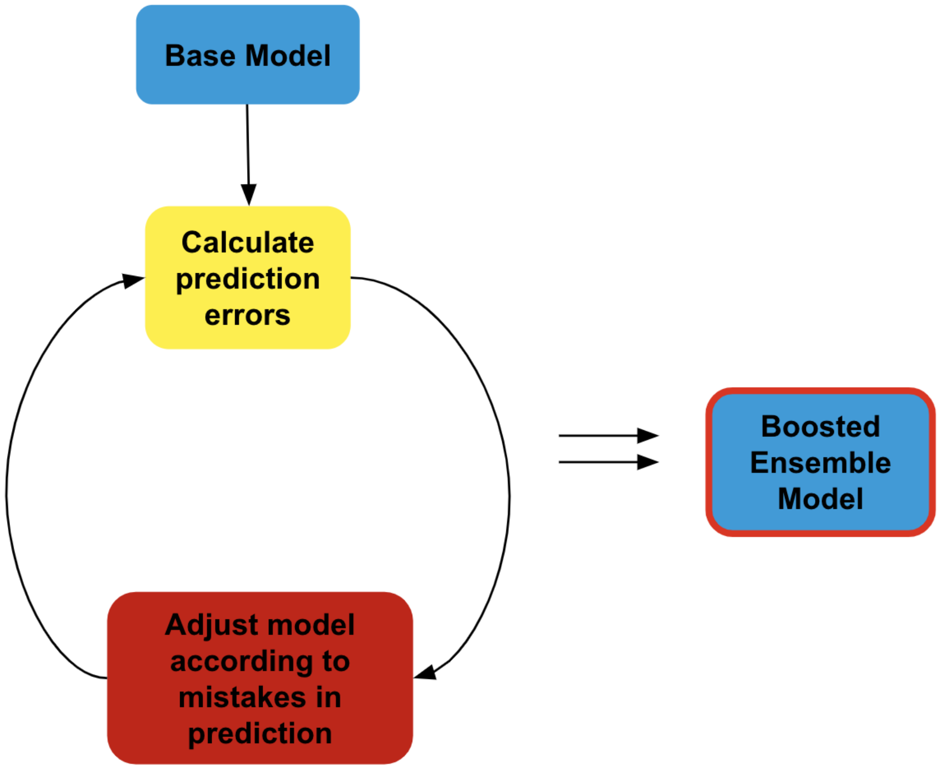
- One of the earliest proposals for ensemble learning was to **fit decision trees on bootstrapped data sets**, then:
    - Average the predictions (continuous outcomes)
    - Choose the majority prediction (categorical outcomes)

Random Forests

- A common variation of bagging is a **Random Forest**, where **slightly different decision trees** are fit to bootstrapped data
    - Different because *hyperparameters* such as maximum number of predictors, number of branches, depth of trees, or minimum number of observations in each leaf are **intentionally varied**
    - **Weakly correlated** trees allowing bagging to yield **robust predictions**

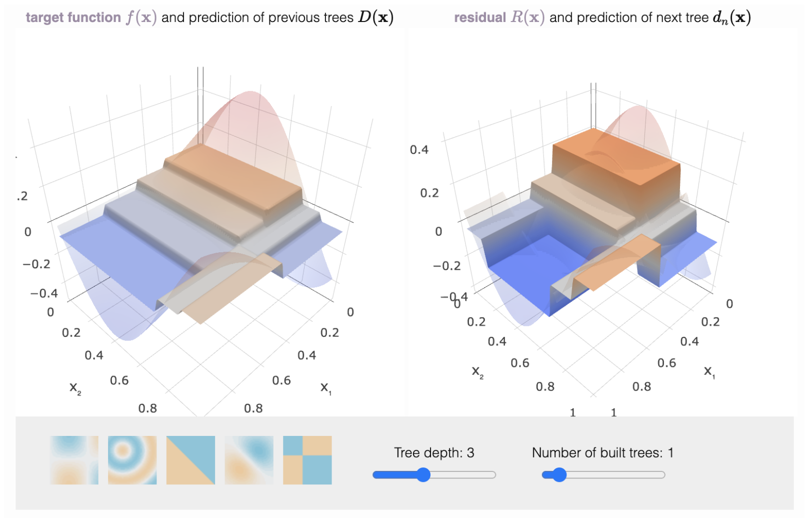
Ensemble Learning Method #2: Boosting

- During bagging, models are fit in **parallel**, but in boosting, models are **fit sequentially** with the **goal to learn from past mistakes**


Base Model
Calculate prediction errors
Adjust model according to mistakes in prediction
Boosted Ensemble Model

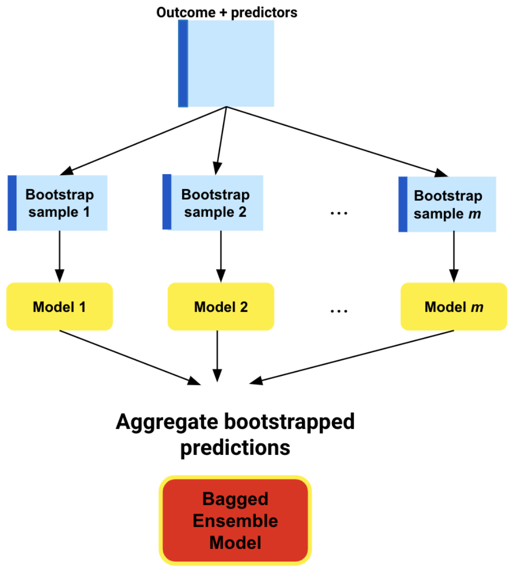
Boosting

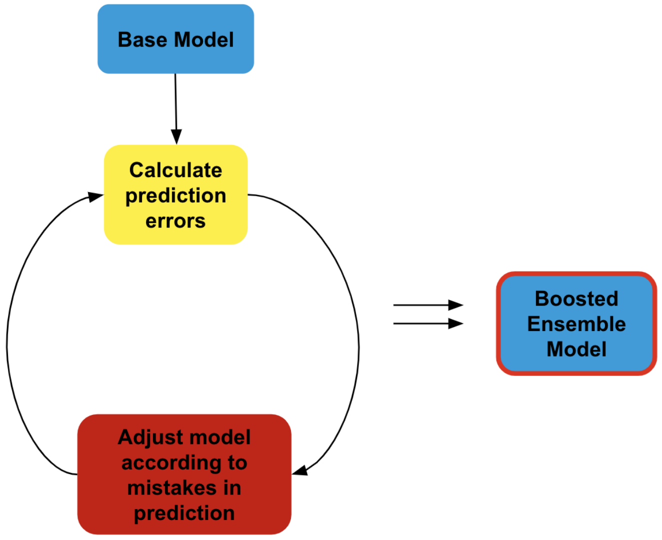
- Two main ways to do "fix the mistakes":
    - **Adaptive boosting:** Adjust model by **assigning a higher weight** to the predictions the previous model got wrong
    - **Gradient boosting:** Adjust model by making a **new model to predict the errors of the previous model** and adding that error prediction to the previous model
- I recommend this interactive tutorial to learn more:


target function $f(\mathbf{x})$ and prediction of previous trees $D(\mathbf{x})$
residual $R(\mathbf{x})$ and prediction of next tree $d_n(\mathbf{x})$
x₂
x₁
Tree depth: 3
Number of built trees: 1

Bagging vs. Boosting

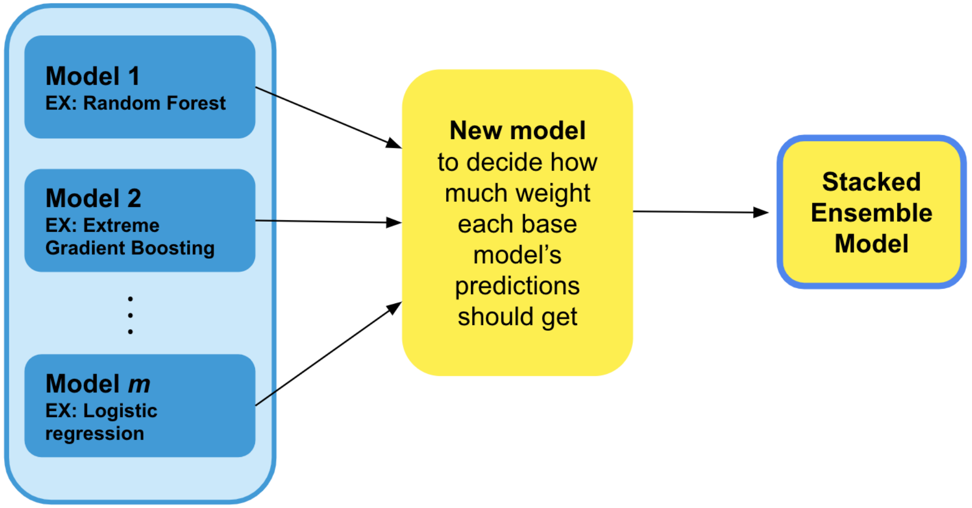
- **Bagging**: fitting the **same class of models** *in parallel*
- **Boosting**: fitting the **same class of models** *sequentially*




Base Model
Calculate prediction errors
Adjust model according to mistakes in prediction
Boosted Ensemble Model

- In contrast, **superlearning** (AKA stacking), combines **different classes of models** through **weighting**

The big picture of superlearning


Model 1
EX: Random Forest
Model 2
EX: Extreme Gradient Boosting
Model *m*
EX: Logistic regression
New model to decide how much weight each base model's predictions should get
Stacked Ensemble Model

- The left-most candidate models are often called **base learners**, and collectively referred to as a **stack**
- The model which assigns weights to the base learners is often called a **metalearner**
- Statistical theory described by van der Laan et. al (2007) tells us that the final stacked model should perform as good or better than any of the individual base learners

Stacking vs. superlearning?

- Similar algorithms proposed under many different names: "model-mix" (Stone 1974), "predictive sample reuse method" (Geisser 1975), "stacking" (and "stacked generalizations") (Wolpert 1992), "weighted ensembling"
- van der Laan et. al proved the theoretical properties, named it "superlearning," which gained popularity in statistics literature, but "stacking" caught on in data science/machine learning community

Ensemble Learning Method #3: Superlearning

- Superlearning combines many **different statistical learning algorithms** through **weighting**
  - Weights assigned according to how well the base learners **minimizes a specified loss function**, e.g. mean squared error (MSE) or area under the curve (AUC)
  - Employ **resampling** to avoid overfitting (we will look at K-fold cross-validation, although other resampling methods can be used)
- *Motivation:* a mixture of **multiple different algorithms may be more optimal** for a given data set than any single algorithm
  - Ex: a tree based model averaged with a linear model (e.g. random forests and LASSO) can smooth some of the model's edges to improve predictive performance

Part II:

The Superlearner algorithm, step by step

Initial set up: generate or obtain data

```r
library(tidyverse)
set.seed(7)

n <- 1000
obs <- tibble(
  id = 1:n,
  x1 = rnorm(n),
  x2 = rbinom(n, 1, plogis(10*x1)),
  x3 = rbinom(n, 1, plogis(x1*x2 + .5*x2)),
  x4 = rnorm(n, mean=x1*x2, sd=.5*x3),
  y = x1 + x2 + x2*x3 + sin(x4)
)
```

Simulated data set

id	x1	x2	x3	x4	y
1	2.29	1	1	2.99	4.43
2	−1.20	0	1	−1.05	−2.07
3	−0.69	0	0	−0.00	−0.69
4	−0.41	0	0	−0.00	−0.41
5	−0.97	0	0	−0.00	−0.97
6	−0.95	0	0	−0.00	−0.95

Initial set up: choose base learners

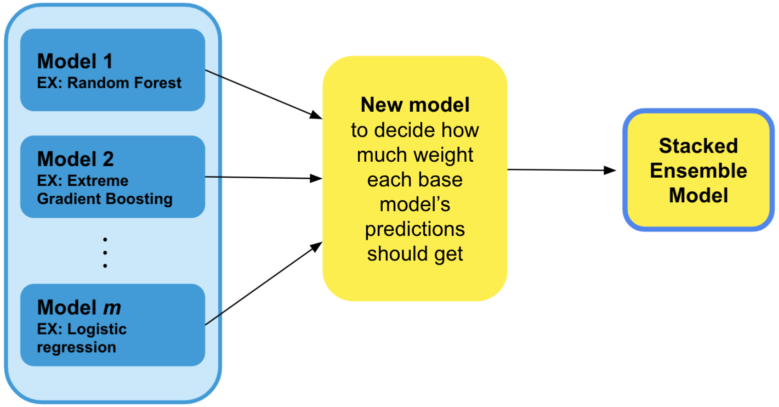
- Can be any **parametric or non-parametric supervised learning algorithm**
  - Best to choose a **diverse range** of learners
  - An example of a base learner library for a continuous outcome might be linear regression, LASSO, random forests, and multivariate adaptive regression splines (MARS)
- Here we will use three different linear regression models *only so that code for more complicated models is not distracting*

1. **Learner A**: $Y = \beta_0 + \beta_1 X_2 + \beta_2 X_4 + \epsilon$
2. **Learner B**: $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_3 + \beta_4 sin(X_4) + \epsilon$
3. **Learner C**:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_1 X_2 + \beta_5 X_1 X_3 + \beta_6 X_2 X_3 + \beta_7 X_1 X_2 X_3$$

Initial set up: choose a metalearner

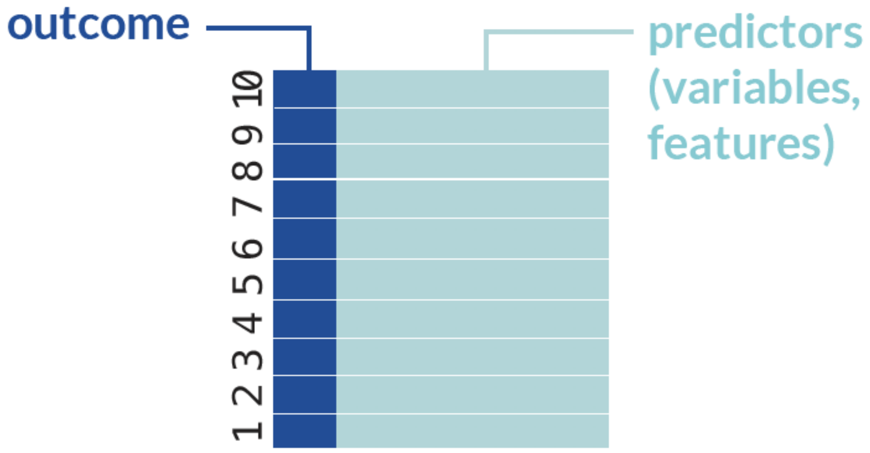
- Recall that the metalearner takes the base learner as inputs to produce a final superlearning algorithm


Model 1
EX: Random Forest
Model 2
EX: Extreme Gradient Boosting
Model *m*
EX: Logistic regression
New model to decide how much weight each base model's predictions should get
Stacked Ensemble Model

- The metalearner represents the choice of loss function
  - Common loss functions: Mean Squared Error (MSE) or Area Under the Curve (AUC)
- We will use a linear regression to minimize the MSE, again *only so that code for more complicated models is not distracting*

Step 1: Split data into K folds

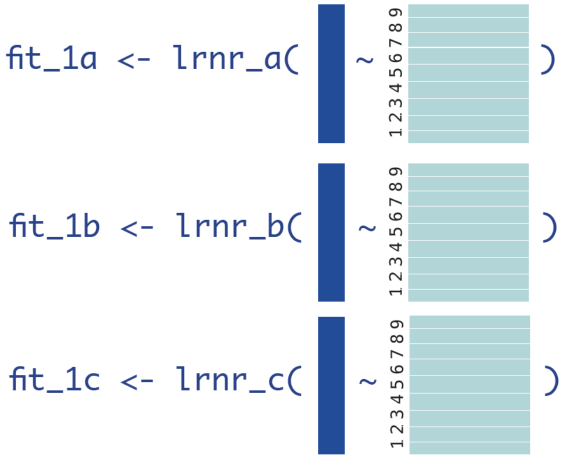
- The superlearner algorithm relies on **K-fold cross-validation** (CV) to avoid overfitting, so we will split our data into K=10 folds



```r
k <- 10 # 10 fold cv
cv_index <- rep(1:k, each = n/k) # create indices for each CV fold
```

Step 2: Fit base learners on first CV-fold

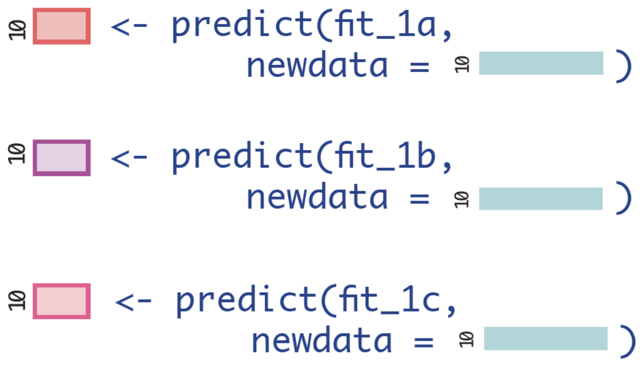
- To begin cross-validation, we take 9 of the 10 folds of data and use those to train each of our base learners:


fit_1a <- lrnr_a(~ 1 2 3 4 5 6 7 8 9)
fit_1b <- lrnr_b(~ 1 2 3 4 5 6 7 8 9)
fit_1c <- lrnr_c(~ 1 2 3 4 5 6 7 8 9)

```
cv_train_1 <- obs[-which(cv_index == 10),]
fit_1a <- glm(y ~ x2 + x4, data=cv_train_1)
fit_1b <- glm(y ~ x1 + x2 + x1*x3 + sin(x4), data=cv_train_1)
fit_1c <- glm(y ~ x1*x2*x3, data=cv_train_1)
```

Step 3: Obtain predictions for first CV-fold

- Then, we "test" or "validate" the fits for each of our base learners using the 10th fold of data:



```r
cv_valid_1 <- obs[which(cv_index == 10),]
pred_1a <- predict(fit_1a, newdata = cv_valid_1)
pred_1b <- predict(fit_1b, newdata = cv_valid_1)
pred_1c <- predict(fit_1c, newdata = cv_valid_1)
```

Step 3: Obtain predictions for first CV-fold

- We now have three vectors that are the length of one fold and contain the validation set's predictions:

First CV round of predictions		
pred_1a	pred_1b	pred_1c
−0.89	−1.07	−0.99
1.83	1.40	2.07
−0.77	−1.36	−1.45
1.90	1.68	2.34

- Now we need to get these CV-predictions for the entire data set

Step 4: CV predictions for entire data

Initial set up for mapping function:

```
cv_folds <- as.list(1:k)
names(cv_folds) <- paste0("fold",1:k)
```

A function to run steps 2 and 3 for any CV `fold`:

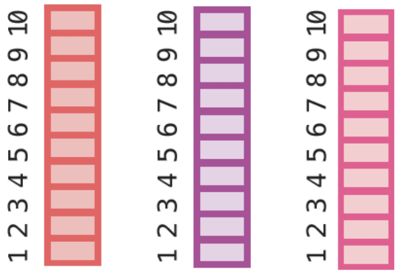
```
get_preds <- function(fold){
  cv_train <- obs[-which(cv_index == fold),]  # make a training data set that contains all data exce
  fit_a <- glm(y ~ x2 + x4, data=cv_train)  # fit all the base learners to that data
  fit_b <- glm(y ~ x1 + x2 + x1*x3 + sin(x4), data=cv_train)
  fit_c <- glm(y ~ x1*x2*x3, data=cv_train)
  cv_valid <- obs[which(cv_index == fold),]  # make a validation data set that only contains data fr
  pred_a <- predict(fit_a, newdata = cv_valid)  # obtain predictions from all the base learners for
  pred_b <- predict(fit_b, newdata = cv_valid)
  pred_c <- predict(fit_c, newdata = cv_valid)
  return(data.frame("obs_id" = cv_valid$id, "cv_fold" = fold, pred_a, pred_b, pred_c))  # save the p
}
```

Loop through every `fold` (`1:k`) and binds the rows of results together:

```
cv_preds <- purrr::map_dfr(cv_folds, ~get_preds(fold = .x))
```

Step 4: CV predictions for entire data

- Each observation has now participated in one validation set, so we have three vectors of CV-predictions that are the same length as the full data

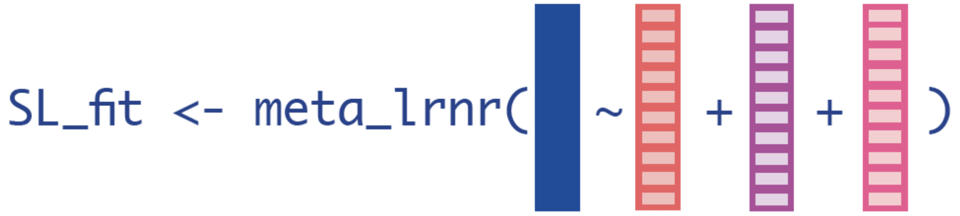


Cross-validated predictions from each base learner for entire data

obs_id	cv_fold	pred_a	pred_b	pred_c
1	1	5.54	4.59	5.37
2	1	−1.93	−2.03	−1.18
3	1	−0.77	−0.80	−0.69
4	1	−0.77	−0.59	−0.41

Step 5: Fit metalearner

- This is the key step of the superlearner algorithm: we will use a new learner, a metalearner, to take information from all of the base learners and create that new algorithm.



```
obs_preds <- full_join(obs, cv_preds, by=c("id" = "obs_id"))
sl_fit <- glm(y ~ pred_a + pred_b + pred_c, data = obs_preds)
```

- This gives us the coefficients, or weights, to apply to our base learners to minimize our loss function of interest

Step 5: Fit metalearner

Metalearner regression coefficients

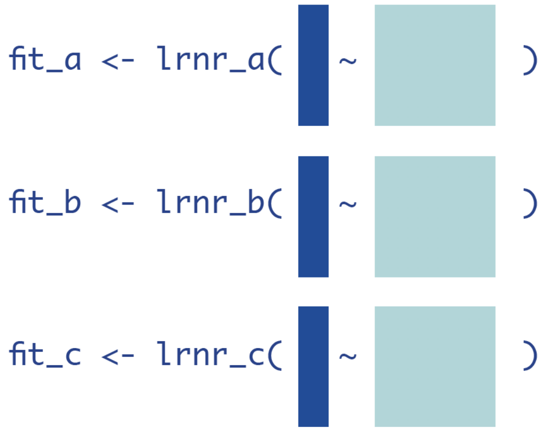
term	estimate	std.error	statistic	p.value
(Intercept)	−0.00	0.00	−0.70	0.48
pred_a	−0.02	0.01	−2.16	0.03
pred_b	0.86	0.01	59.31	0.00
pred_c	0.16	0.01	13.36	0.00

- After our metalearning step, we conclude that given a set of predictions from Learner A, B, and C, we obtain our best possible predictions by adding -0.02 $\times$ predictions from Learner A, 0.86 $\times$ predictions from Learner B, and 0.16 $\times$ predictions from Learner C

- Note that Learner B gets the highest coefficient! We will see later this is because it has the lowest CV-MSE.

Step 6: Fit base learners on entire data

- Now fit the base learners to the *entire* data set:

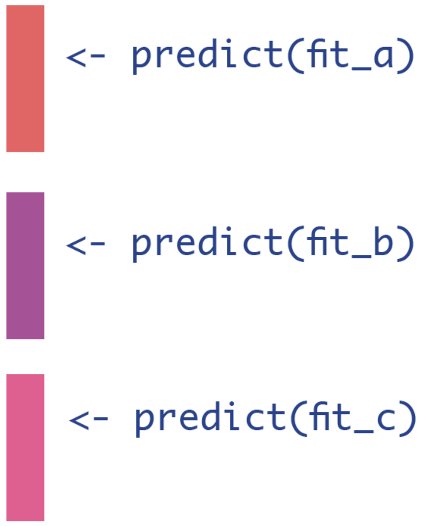


```
fit_a <- glm(y ~ x2 + x4, data=obs)
fit_b <- glm(y ~ x1 + x2 + x1*x3 + sin(x4), data=obs)
fit_c <- glm(y ~ x1*x2*x3, data=obs)
```

- Recall that when we previously fit these base learners, it was only on 9/10 of the data

Step 7: Obtain predictions from each base learner on entire data set

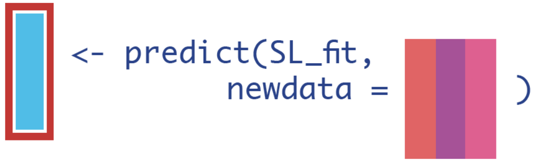
- Obtain the base learner predictions for the *entire* data set



```r
pred_a <- predict(fit_a)
pred_b <- predict(fit_b)
pred_c <- predict(fit_c)
full_data_preds <- tibble(pred_a, pred_b, pred_c)
```

Step 8: Weight base learners from entire data

- Use the base learner predictions (from the full data set) as inputs to the metalearner fit to apply the appropriate weight to each base learner prediction.



```r
sl_predictions <- predict(sl_fit, newdata = full_data_preds)
sl_predictions[[1]] # first observation's prediction
```

```
## [1] 4.698735
```

Step 9: Obtain predictions on new data

- To predict on entirely new data, use the fits from each base learner (fit on entire data) to obtain base learner predictions for the new observations, then plug those base learner predictions into the metalearner fit

Example row of new data:

```r
new_obs <- tibble(x1 = .5, x2 = 0, x3 = 0, x4 = -3)
```

Predict using base learners:

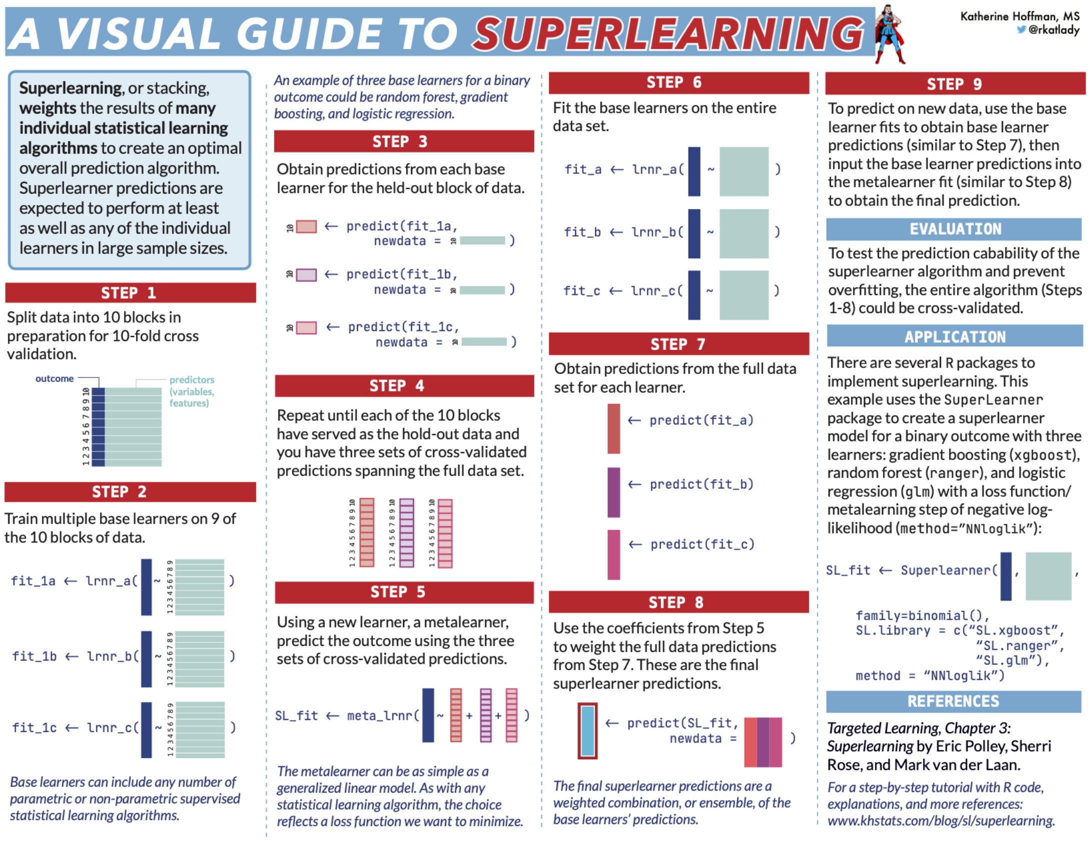
```r
new_pred_a <- predict(fit_a, newdata = new_obs)
new_pred_b <- predict(fit_b, newdata = new_obs)
new_pred_c <- predict(fit_c, newdata = new_obs)
new_pred_df <- tibble("pred_a" = new_pred_a, "pred_b" = new_pred_b, "pred_c" = new_pred_c)
```

Plug those base learner predictions into the metalearner fit.

```r
predict(sl_fit, newdata = new_pred_df)
```

```
##          1
## 0.09909956
```

If you missed anything...



A VISUAL GUIDE TO SUPERLEARNING

Katherine Hoffman, MS
@rkatlady

Superlearning, or stacking, **weights** the results of **many individual statistical learning algorithms** to create an optimal overall prediction algorithm. Superlearner predictions are expected to perform at least as well as any of the individual learners in large sample sizes.

STEP 1

Split data into 10 blocks in preparation for 10-fold cross validation.

STEP 2

Train multiple base learners on 9 of the 10 blocks of data.

```
fit_1a ← lrnr_a(  ~  )
fit_1b ← lrnr_b(  ~  )
fit_1c ← lrnr_c(  ~  )
```

Base learners can include any number of parametric or non-parametric supervised statistical learning algorithms.

An example of three base learners for a binary outcome could be random forest, gradient boosting, and logistic regression.

STEP 3

Obtain predictions from each base learner for the held-out block of data.

```
10 ← predict(fit_1a,
        newdata = 10 )
10 ← predict(fit_1b,
        newdata = 10 )
10 ← predict(fit_1c,
        newdata = 10 )
```

STEP 4

Repeat until each of the 10 blocks have served as the hold-out data and you have three sets of cross-validated predictions spanning the full data set.

STEP 5

Using a new learner, a metalearner, predict the outcome using the three sets of cross-validated predictions.

```
SL_fit ← meta_lrnr(  ~  +  +  )
```

The metalearner can be as simple as a generalized linear model. As with any statistical learning algorithm, the choice reflects a loss function we want to minimize.

STEP 6

Fit the base learners on the entire data set.

```
fit_a ← lrnr_a(  ~  )
fit_b ← lrnr_b(  ~  )
fit_c ← lrnr_c(  ~  )
```

STEP 7

Obtain predictions from the full data set for each learner.

```
← predict(fit_a)
← predict(fit_b)
← predict(fit_c)
```

STEP 8

Use the coefficients from Step 5 to weight the full data predictions from Step 7. These are the final superlearner predictions.

```
← predict(SL_fit,
      newdata =  )
```

The final superlearner predictions are a weighted combination, or ensemble, of the base learners' predictions.

STEP 9

To predict on new data, use the base learner fits to obtain base learner predictions (similar to Step 7), then input the base learner predictions into the metalearner fit (similar to Step 8) to obtain the final prediction.

EVALUATION

To test the prediction cabability of the superlearner algorithm and prevent overfitting, the entire algorithm (Steps 1-8) could be cross-validated.

APPLICATION

There are several R packages to implement superlearning. This example uses the `SuperLearner` package to create a superlearner model for a binary outcome with three learners: gradient boosting (`xgboost`), random forest (`ranger`), and logistic regression (`glm`) with a loss function/ metalearning step of negative log-likelihood (`method="NNloglik"`):

```
SL_fit ← Superlearner(  ,  ,
    family=binomial(),
    SL.library = c("SL.xgboost",
                   "SL.ranger",
                   "SL.glm"),
    method = "NNloglik")
```

REFERENCES

Targeted Learning, Chapter 3: Superlearning by Eric Polley, Sherri Rose, and Mark van der Laan.

For a step-by-step tutorial with R code, explanations, and more references: www.khstats.com/blog/sl/superlearning.

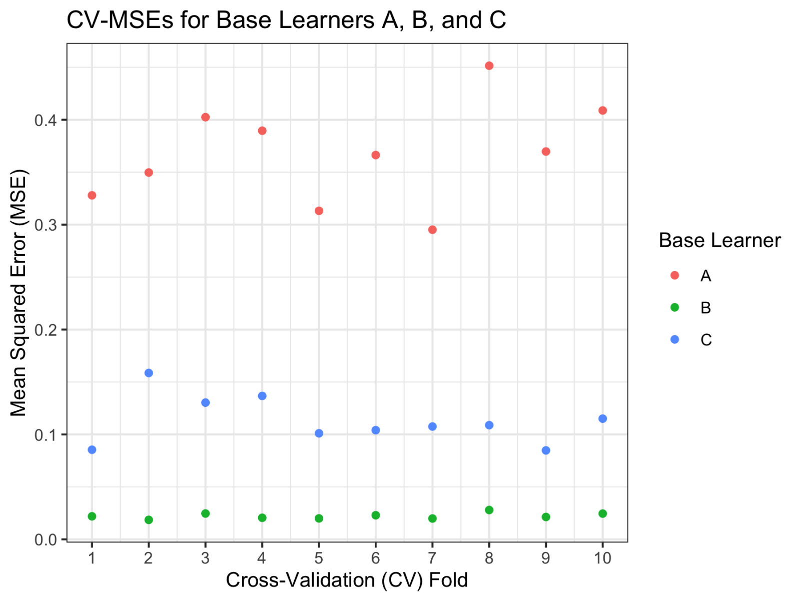
A printable pdf of this cheat sheet is on my Github.

Step 10 and beyond...

- We could **compute the MSE** of the final superlearner predictions:

```
sl_mse <- mean((obs$y - sl_predictions)^2)
```

- Or, we could look at the MSE for each CV-fold:



Step 10 and beyond...

- A minor note: if we had decided to use Learner B because it minimized our loss function of interest, that choice is often called the **discrete superlearner**

```
discrete_sl_predictions <- predict(glm(y ~ x1 + x2 + x1*x3 + sin(x4), data=obs))
```

- Could also **add more algorithms** to base learner stack (we definitely should, since we only used linear regressions!)
  - Write functions to **tune these algorithms' hyperparameters** over various grids
- May want to **cross-validate** the entire process to evaluate the predictive performance of our superlearner algorithm
  - Alternatively, we could leave a hold-out data set for testing

Using the SuperLearner package

- Building an ensemble superlearner for our data with the base learner stack of `ranger` (random forests), `glmnet` (LASSO, by default), and `earth` (MARS) using the `SuperLearner` package in `R`:

```r
library(SuperLearner)
x_df <- obs %>% select(x1:x4) %>% as.data.frame()
sl_fit <- SuperLearner(Y = obs$y,
                       X = x_df,
                       family = gaussian(),
                     SL.library = c("SL.ranger", "SL.glmnet", "SL.earth"))
```

- Detailed vignette
- Base learners are easily customizable
    - Add screens, tune over hyperparameters with `caret`
- Cross-validate entire algorithm with `CV.SuperLearner` function

Using the SuperLearner package

- View weights (`Coef`) by printing model fit:

```
sl_fit
```

```
##
## Call:
## SuperLearner(Y = obs$y, X = x_df, family = gaussian(), SL.library = c("SL.ranger",
##     "SL.glmnet", "SL.earth"))
##
##
##                        Risk     Coef
## SL.ranger_All 0.018386303 0.142705
## SL.glmnet_All 0.094759246 0.000000
## SL.earth_All  0.003855509 0.857295
```

- `Coef`s are inversely related to the CV-`Risk` (default is MSE)
  - Here, MARS gets most of the prediction weight and LASSO gets none
- Specify metalearner with `method` argument
  - Default is Non-Negative Least Squares (NNLS) using `nnls` package (linear regression with coefficients restrained to non-negative numbers)

A brief comparison of `R` Packages

- `SuperLearner` is the oldest `R` package
  - Pros: well-vetted and documented; Cons: very slow
- `h2o` is an AI company which offers scalable prediction tools. The `R` version of `h2o` is available on CRAN.
  - Pros: fast, well-vetted; Cons: steeper learning curve for one-off projects
- `sl3` is the updated version of `SuperLearner` and is the backend of several causal inference packages out of UC Berkeley which allow for ensemble learning
  - Pros: fast; Cons: not well-vetted yet (not on CRAN), uses `R6` interface which can be confusing for non-Python programmers
- New `stacks` package by `tidymodels` through Rstudio
  - Pros: unified framework with other `tidymodels` packages; Cons: commitment to `tidymodels` framework

Questions?

Slides: {xaringan}

Email: kah2797@med.cornell.edu

References

MJ Van der Laan, EC Polley, AE Hubbard, Super Learner, Statistical applications in genetics and molecular, 2007

Polley, Eric. “Chapter 3: Superlearning.” Targeted Learning: Causal Inference for Observational and Experimental Data, by M. J. van der. Laan and Sherri Rose, Springer, 2011.

Polley E, LeDell E, Kennedy C, van der Laan M. Super Learner: Super Learner Prediction. R package version 2.0-22.

Naimi AI, Balzer LB. Stacked generalization: an introduction to super learning. *Eur J Epidemiol.* 2018;33(5):459-464. doi:10.1007/s10654-018-0390-z

LeDell, E. (2015). Scalable Ensemble Learning and Computationally Efficient Variance Estimation. UC Berkeley. ProQuest ID: LeDell_berkeley_0028E_15235. Merritt ID: ark:/13030/m5wt1xp7.

M. Petersen and L. Balzer. Introduction to Causal Inference. UC Berkeley, August 2014.