6-8: SuperLearner and LTMLE - Solutions

```r
# Install packages
# ----------
if (!require("pacman")) install.packages("pacman")

pacman::p_load(# Tidyverse packages including dplyr and ggplot2
               tidyverse,
               ggthemes,
               ltmle,   # Longitudinal Targeted Maximum Likelihood Estimation
               tmle,    # Targeted Maximum Likelihood Estimation
               SuperLearner,
               caret,
               e1071,    # svm
               rpart,    # decision trees
               furrr,    # parallel processing
               parallel, # parallel processing
               ranger,
               tidymodels)   # fast implementation of random forests

# set seed
# ----------
set.seed(44)
```

Introduction

For our final lab, we will be looking at the SuperLearner library, as well as the Targeted Maximum Likelihood Estimation (TMLE) framework, with an extension to longitudinal data structures. This lab brings together a lot of what we learned about both machine learning and causal inference this year, and serves as an introduction to this intersection!

Data

For this lab, we will use the Boston dataset from the `MASS` library. This dataset is a frequently used toy dataset for machine learning. The main variables we are going to predict is `medv` which is the median home value for a house in Boston.

```r
# Load Boston dataset from MASS package
# ----------
data(Boston, package = "MASS")
glimpse(Boston)
```

```
## Rows: 506
## Columns: 14
## $ crim    <dbl> 0.00632, 0.02731, 0.02729, 0.03237, 0.06905, 0.02985, 0.08829,~
## $ zn      <dbl> 18.0, 0.0, 0.0, 0.0, 0.0, 0.0, 12.5, 12.5, 12.5, 12.5, 12.5, 1~
## $ indus   <dbl> 2.31, 7.07, 7.07, 2.18, 2.18, 2.18, 7.87, 7.87, 7.87, 7.87, 7.~
## $ chas    <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ nox     <dbl> 0.538, 0.469, 0.469, 0.458, 0.458, 0.458, 0.524, 0.524, 0.524,~
## $ rm      <dbl> 6.575, 6.421, 7.185, 6.998, 7.147, 6.430, 6.012, 6.172, 5.631,~
## $ age     <dbl> 65.2, 78.9, 61.1, 45.8, 54.2, 58.7, 66.6, 96.1, 100.0, 85.9, 9~
## $ dis     <dbl> 4.0900, 4.9671, 4.9671, 6.0622, 6.0622, 6.0622, 5.5605, 5.9505~
## $ rad     <int> 1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4,~
## $ tax     <dbl> 296, 242, 242, 222, 222, 222, 311, 311, 311, 311, 311, 311, 31~
## $ ptratio <dbl> 15.3, 17.8, 17.8, 18.7, 18.7, 18.7, 15.2, 15.2, 15.2, 15.2, 15~
## $ black   <dbl> 396.90, 396.90, 392.83, 394.63, 396.90, 394.12, 395.60, 396.90~
## $ lstat   <dbl> 4.98, 9.14, 4.03, 2.94, 5.33, 5.21, 12.43, 19.15, 29.93, 17.10~
## $ medv    <dbl> 24.0, 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9, 15~
```

SuperLearner

First, we are going to introduce the SuperLearner package for machine learning in R. SuperLearner was developed here at Berkeley, and we are going to follow the guide put together by Chris Kennedy. There are some differences, mostly because of the introduction of new machine learning tools like caret and tidymodels.

The basic idea underlying SuperLearner is that it combines cross-validation with ensemble learning to create a "meta-estimator" that is a weighted combination of constituent algorithms. This idea should sound familiar from when we explored ensemble methods in `sklearn` in Python.

Train/Test Split

Let's start with our typical train/test split. There are several options for doing this, but we will use the `tidymodels` method. Let's take a look:

```r
# initial split
# ----------
boston_split <-
  initial_split(Boston, prop = 3/4) # create initial split (tidymodels)


# Training
# ----------
train <-
  # Declare the training set with rsample::training()
  training(boston_split)

# y_train
y_train <-
  train %>%
  # is medv where medv > 22 is a 1, 0 otherwise
  mutate(medv = ifelse(medv > 22,
                       1,
                       0)) %>%
  # pull and save as vector
  pull(medv)

# x_train
x_train <-
  train %>%
  # drop the target variable
  select(-medv)

# Testing
# ----------
```

```r
test <-
  # Declare the training set with rsample::training()
  testing(boston_split)

# y test
y_test <-
  test %>%
  mutate(medv = ifelse(medv > 22,
                       1,
                       0)) %>%
  pull(medv)

# x test
x_test <-
  test %>%
  select(-medv)
```

SuperLearner Models

Now that we have our train and test partitions set up, let's see what machine learning models we have available to us. We can use the `listWrappers()` function from SuperLearner:

```r
# Superlearner:  list all the machine learning algorithms
# ----------
listWrappers()
```

```
##  [1] "SL.bartMachine"      "SL.bayesglm"         "SL.biglasso"
##  [4] "SL.caret"            "SL.caret.rpart"      "SL.cforest"
##  [7] "SL.earth"            "SL.gam"              "SL.gbm"
## [10] "SL.glm"              "SL.glm.interaction"  "SL.glmnet"
## [13] "SL.ipredbagg"        "SL.kernelKnn"        "SL.knn"
## [16] "SL.ksvm"             "SL.lda"              "SL.leekasso"
## [19] "SL.lm"               "SL.loess"            "SL.logreg"
## [22] "SL.mean"             "SL.nnet"             "SL.nnls"
## [25] "SL.polymars"         "SL.qda"              "SL.randomForest"
## [28] "SL.ranger"           "SL.ridge"            "SL.rpart"
## [31] "SL.rpartPrune"       "SL.speedglm"         "SL.speedlm"
## [34] "SL.step"             "SL.step.forward"     "SL.step.interaction"
## [37] "SL.stepAIC"          "SL.svm"              "SL.template"
## [40] "SL.xgboost"
## [1] "All"
## [1] "screen.corP"           "screen.corRank"        "screen.glmnet"
## [4] "screen.randomForest"   "screen.SIS"            "screen.template"
## [7] "screen.ttest"          "write.screen.template"
```

Notice how we have both prediction algorithms for supervised learning, and screening algorithms for feature selection (some may be both).

Let's go ahead and fit a model. We'll start with a LASSO which we can call via `glmnet`:

```r
# set seed
set.seed(12345)

# LASSO
# ----------
sl_lasso <- SuperLearner(Y = y_train,              # target
                          X = x_train,              # features
                          family = binomial(),      # binomial : 1,0s
                          SL.library = "SL.glmnet") # find the glmnet algo from SL

# view
sl_lasso
```

```
##
## Call:
## SuperLearner(Y = y_train, X = x_train, family = binomial(), SL.library = "SL.glmnet")
##
##
##
##                    Risk Coef
## SL.glmnet_All 0.0928939    1
```

Notice how it spits out a "Risk" and a "Coef". The "Coef" here is 1 because this is the only model in our ensemble right now. Risk is essentially a measure of accuracy, in this case something like mean squared error. We can see the model in our ensemble that had the lowest risk like this:

```
# Here is the risk of the best model (discrete SuperLearner winner).
# Use which.min boolean to find minimum cvRisk in list
sl_lasso$cvRisk[which.min(sl_lasso$cvRisk)]
```

```
## SL.glmnet_All
##     0.0928939
```

Multiple Models

Now let's extend this framework to multiple models. Within SuperLearner, all we need to do is add models to the `SL.library` argument:

```
# set seed
set.seed(987)

# multiple models
# ----------
sl = SuperLearner(Y = y_train,
                  X = x_train,
                  family = binomial(),
                  # notice these models are concatenated
                  SL.library = c('SL.mean',    # if you just guessed the average - serves as a baseline
                                 'SL.glmnet',
                                 'SL.ranger'))
sl
```

```
##
## Call:
## SuperLearner(Y = y_train, X = x_train, family = binomial(), SL.library = c("SL.mean",
##     "SL.glmnet", "SL.ranger"))
##
##
##                     Risk      Coef
## SL.mean_All   0.24842495 0.0000000
## SL.glmnet_All 0.09076349 0.0974408
## SL.ranger_All 0.07543247 0.9025592
```

Now let's move to our validation step. We'll use the `predict()` function to take our SuperLearner model (only keeping models that had weights) and generate predictions. We can then compare our predictions against our true observations.

```r
# predictions
# ----------
preds <-
  predict(sl,              # use the superlearner not individual models
          x_test,          # prediction on test set
          onlySL = TRUE)   # use only models that were found to be useful (had weights)


# start with y_test
validation <-
  y_test %>%
  # add our predictions - first column of predictions
  bind_cols(preds$pred[,1]) %>%
  # rename columns
  rename(obs = `...1`,       # actual observations
         pred = `...2`) %>% # predicted prob
  # change pred column so that obs above .5 are 1, otherwise 0
  mutate(pred = ifelse(pred >= .5,
                           1,
                           0))

# view
head(validation)
```

```
## # A tibble: 6 x 2
##     obs  pred
##   <dbl> <dbl>
## 1     1     1
## 2     1     1
## 3     1     1
## 4     0     0
## 5     0     0
## 6     0     0
```

Notice that the predictions are not binary 1/0s, but rather probabilities. So, we recode these so that if estimate >= .5 it becomes a 1, and 0 otherwise. We can then use our standard classification metrics and create a confusion matrix using `caret`:

TP: True Positives, predicted a 1 and observed a 1 FP: False Positives, predicted a 1 and observed a 0 TN: True Negatives, predicted a 0 and observed a 0 FN: False Negatives, predicted a 0 and observed a 1

```r
# confusion matrix
# ----------
caret::confusionMatrix(as.factor(validation$pred),
                       as.factor(validation$obs))
```

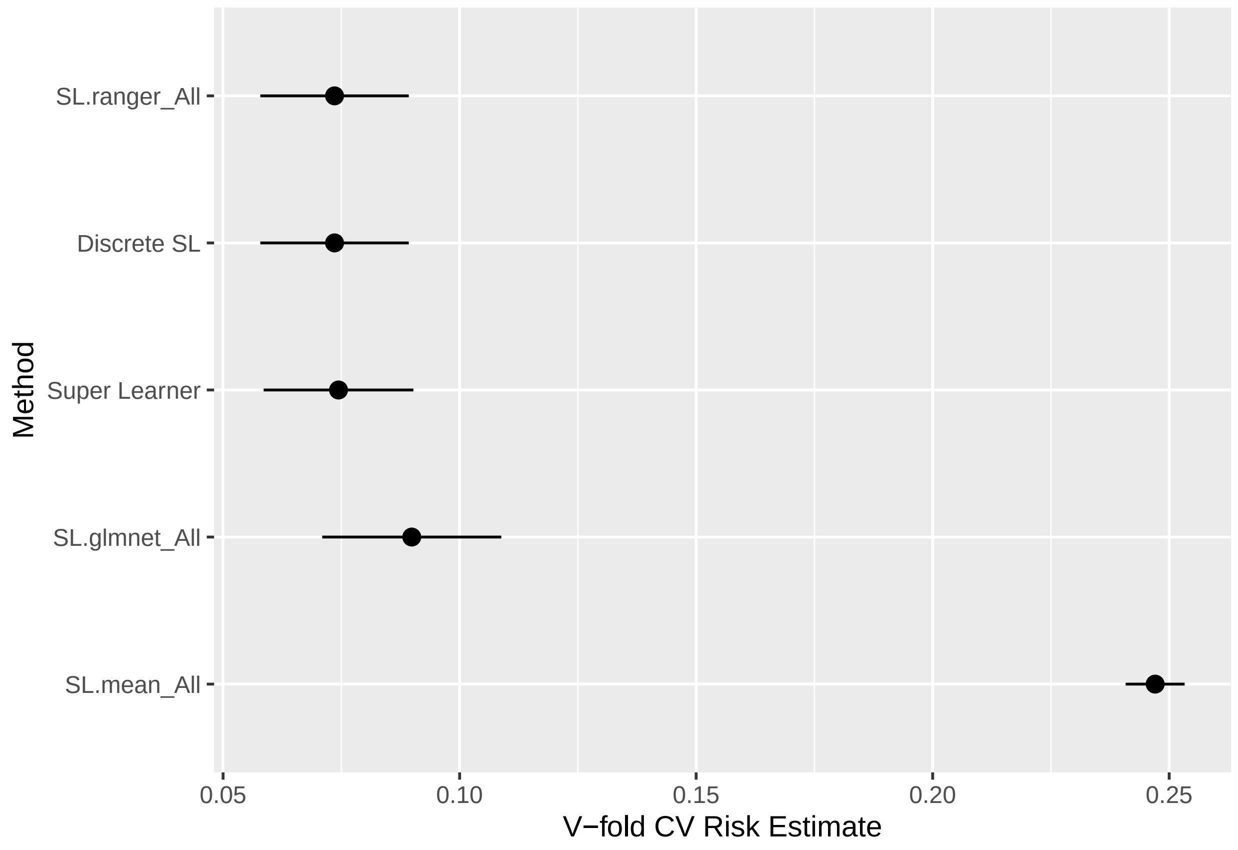
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 64 11
##          1  7 45
##
```

```
##                Accuracy : 0.8583
##                  95% CI : (0.7853, 0.9138)
##     No Information Rate : 0.5591
##     P-Value [Acc > NIR] : 4.827e-13
##
##                   Kappa : 0.7103
##
##  Mcnemar's Test P-Value : 0.4795
##
##             Sensitivity : 0.9014
##             Specificity : 0.8036
##          Pos Pred Value : 0.8533
##          Neg Pred Value : 0.8654
##              Prevalence : 0.5591
##          Detection Rate : 0.5039
##    Detection Prevalence : 0.5906
##       Balanced Accuracy : 0.8525
##
##        'Positive' Class : 0
##
```

Ensemble Learning and Parallel Processing

SuperLearner can take a long time to run, but we can also speed this up with parallelization. Unfortunately this works slightly differently in Windows and Mac/Linux (R doesn't seem to recognize Windows Subsystem for Linux).

```
#
# Macs/Linux
# --------------------------------------------------------

# identify number of cores to use
n_cores <- availableCores() - 1


# plan separate session
plan(multisession,
     workers = n_cores)

# set seed - option will set same seed across multiple cores
set.seed(44, "L'Ecuyer-CMRG")

# cross-validation across cores
# ----------
cv_sl = CV.SuperLearner(Y = y_train,
                        X = x_train,
                        family = binomial(),      #
                        V = 20,                   # default fold recommendation for Superlearner
                        parallel = 'multicore',   # macs linux parallel specification, note its a strin
                        SL.library = c("SL.mean",
                                         "SL.glmnet",
                                         "SL.ranger"))

# plot
plot(cv_sl)
```



```
#
# Windows
# ----------------------------------------------------------

# Windows (for SL only, the above should work for tidymodels)
cluster = parallel::makeCluster(availableCores() - 1)

# Load SuperLearner onto all clusters
parallel::clusterEvalQ(cluster, library(SuperLearner))
# set seed using parallel library
parallel::clusterSetRNGStream(cluster, 1)

# cross-validation across cores
# ----------
cv_sl = CV.SuperLearner(Y = y_train,
                        X = x_train,
                        family = binomial(),
                        V = 20,              # folds
                        parallel = cluster,  # note "cluster" is not a string
                        SL.library = c("SL.mean",
                                       "SL.glmnet",
                                       "SL.ranger"))


parallel::stopCluster(cluster)
```

```r
# plot
plot(cv_sl)
```

Targeted Maximum Likelihood Estimation (TMLE)

We're now ready to move to the TMLE framework. TMLE combines machine learning and causal inference by using machine learning (i.e. data-adaptive models) to estimate some quantity of interest (so making inference). The key is that even though machine learning models oftentimes do not have outputs like coefficients, they can still be used to target the estimator (i.e. a regression) to that quantity of interest. This has a few different benefits, the primary one being that this framework creates a **doubly robust** estimator. Double robustness means that if we either:

1. Fit the right model to estimate the expected outcome correctly

OR

2. Fit the model to estimate the probability of treatment correctly

Then the final TMLE estimator will be **consistent**. Consistent means that as the sample size grows to infinity, the bias will drop to 0. We're going to explore this idea in depth. The example we will go through here is drawn from Katherine Hoffman's blog. She also provides visual guides to SuperLearner and TMLE. I recommend consulting the TMLE visual guide in particular as you work through these steps.

The basic procedure we will go through is:

1. Estimate the outcome model, $\bar{Q}_n^0(A, W)$
2. Estimate the probability of treatment model so we can "target" the initial estimate, $g(W) = P(A = 1|W)$
3. Extract the "clever covariate" or fluctuation parameter that tells us how to update our initial estimate
4. Update the initial estimate to get $\bar{Q}_n^1(A, W)$
5. Calculate ATE
6. Calculate confidence intervals

Step 1: Initial Estimate of the Outcome The first step in TMLE is to estimate the **outcome model**, or the prediction of our target,Y, conditional on the treatment status, A and covariates/features, W. We could do this via a classic regression model, but we could also flexibly fit the model using machine learning. In this case, we'll create a SuperLearner ensemble. This Q step gives us our initial estimate. Do the following:

1. Create a SuperLearner library called `sl_libs`
2. Prepare an outcome vector, Y and treatment/covariate matrix, W_A
3. Fit the SuperLearner model for the Q step to obtain an initial estimate of the outcome

```r
# specify which SuperLearner algorithms we want to use
# ----------
sl_libs <- c('SL.glmnet', 'SL.ranger', 'SL.glm')

# Prepare data for SuperLearner/TMLE
# ----------
# Mutate Y, A for outcome and treatment, use tax, age, and crim as covariates
data_obs <-
  Boston %>%
  mutate(Y = ifelse(medv > 22,
                    1,
                    0)) %>%
  rename(A = chas) %>%
  select(Y, A, tax, age, crim)  # A = borders the charles river

# Outcome
# ----------
Y <-
  data_obs %>%
  pull(Y)

# Covariates
# ----------
W_A <-
  data_obs %>%
  select(-Y)

# Fit SL for Q step, initial estimate of the outcome
# ----------
Q <- SuperLearner(Y = Y,                 # outcome
                  X = W_A,               # covariates + treatment
                  family = binomial(),   # binominal bc outcome is binary
                  SL.library = sl_libs) # ML algorithms
```

Now that we have trained our model, we need to create predictions for three different scenarios:

1. Predictions assuming every unit had its observed treatment status.
2. Predictions assuming every unit was treated.
3. Predictions assuming every unit was control.

Fill in the code to obtain these predictions.

```
# observed treatment
# ----------
Q_A <- as.vector(predict(Q)$pred)

# if every unit was treated (pretending every unit was treated)
# ----------
W_A1 <- W_A %>% mutate(A = 1)
Q_1 <- as.vector(predict(Q, newdata = W_A1)$pred)

# if everyone was control (pretending every unit was not treated)
# ----------
W_A0 <- W_A %>% mutate(A = 0)
Q_0 <- as.vector(predict(Q, newdata = W_A0)$pred)
```

We can take our predictions and put them all into one dataframe:

```
# combine all predictions into one dataframe
# ----------
dat_tmle <- tibble(Y = Y,
                   A = W_A$A,
                   Q_A,
                   Q_0,
                   Q_1)
# view
head(dat_tmle)
```

```
## # A tibble: 6 x 5
##       Y     A   Q_A   Q_0   Q_1
##   <dbl> <int> <dbl> <dbl> <dbl>
## 1     1     0 0.848 0.848 0.872
## 2     0     0 0.190 0.190 0.239
## 3     1     0 0.699 0.699 0.728
## 4     1     0 0.860 0.860 0.884
## 5     1     0 0.929 0.929 0.933
## 6     1     0 0.907 0.907 0.913
```

We *could* go ahead and grab our ATE now using G-computation, which is the difference in expected outcomes under treatment and control conditional on covariates. However, the estimate we get below is targeted (optimized the bias-variance tradeoff) at the predictions of the outcome, not the ATE.

```
# G-computation
# ----------
ate_gcomp <- mean(dat_tmle$Q_1 - dat_tmle$Q_0)
ate_gcomp
```

```
## [1] 0.118331
```

Step 2: Probability of Treatment Now that we have an initial estimate, we want to "target" it. To do this, we first need to estimate the probability of every unit receiving treatment. This step should look similar to how we estimated the propensity score during matching. Fit a g model using SuperLearner. **Hint**: Think carefully about what should be supplied to the Y and X arguments here.

```
# prepare data for analysis
# ----------
A <- W_A$A
W <- Boston %>% select(tax, age, crim)  # select jsut a few covariates

# model probability of treatment (similar to matching)
# ----------
g <- SuperLearner(Y = A,              # outcome is treatment
                  X = W,              # vector of covariates (predictors)
                  family=binomial(),  # binary outcome
                  SL.library=sl_libs) # models
```

Now that we have a model for our propensity score, we can go ahead and calculate both the inverse probability of receiving treatment and the negative inverse probability of not receiving treatment (basically the probability of not receiving treatment). Fill in the code below to obtain both of these quantities. **Hint**: The name "negative inverse probability of not receiving treatment" is a mouthful but should give you a hint about how to set up the calculation.

```
# Prediction for probability of treatment
# ----------
g_w <- as.vector(predict(g)$pred) # Pr(A=1|W)

# probability of treatment: iptw : (inverse probability weight of receiving treatment)
# ----------
H_1 <- 1/g_w

# probability of control: niptw : (negative inverse probability weight of not receiving treatment)
# ----------
H_0 <- -1/(1-g_w)
```

We can then create the "clever covariate" which assigns the probability of treatment to treated variables, and the probability of control to control variables.

```
# create a clever covariate
# ----------
```

```r
dat_tmle <- # add clever covariate data to dat_tmle
  dat_tmle %>%
  bind_cols(
         H_1 = H_1,
         H_0 = H_0) %>%
  # create clever covariate whereby case takes iptw or niptw based on treatment status
  mutate(H_A = case_when(A == 1 ~ H_1,
                         A == 0 ~ H_0))
```

We now have the initial estimate of the outcome, Q_A, and the estimates for the probability of treatment, H_A.

Step 3: Fluctuation Parameter We are now ready to update our initial estimate, Q_A with the information from our propensity score estimates, H_A. We perform this update by fitting a logistic regression where we predict our outcome after passing our initial estimate, Q_A through a logistic transformation and then fitting a logistic regression (the `-1 + offset()` here is necessary because our intercept term is not constant).

```r
# fluctuation parameter
# ----------
glm_fit <- glm(Y ~ -1 + offset(qlogis(Q_A)) + H_A,
               data=dat_tmle,
               family=binomial)
```

We can then grab the coefficient from our model:

```r
# information we need to summarize probability of treatment step above -- how much to update
# ----------
eps <- coef(glm_fit)
```

eps (or epsilon/$\hat{\epsilon}$) is the **fluctuation parameter**. This is the coefficient on our "clever covariate", and basically tells us how much to update our intial model, Q by.

Step 4: Update Initial Estimates We can now update the expected outcome for all of observations, conditional on their observed treatment status and their covariates:

```r
# add H_A variable
# ----------
H_A <- dat_tmle$H_A

# predictions for treatment status
# ----------
Q_A_update <- plogis(qlogis(Q_A) + eps*H_A) # updating by adding eps * H_A
```

Do the same for outcome under treatment:

```r
# predictions assuming everyone under treatment
# ----------
Q_1_update <- plogis(qlogis(Q_1) + eps*H_1) # updating by adding eps * H_1
```

Do the same for outcome under control:

```r
# predictions assuming everyone under  control
# ----------
Q_0_update <- plogis(qlogis(Q_0) + eps*H_0) # updating by adding eps * H_0
```

We now have updated expected outcomes for each unit for their actual treatment status, as well as simulated if they were all in treatment and all in control.

Step 5: Compute the Statistical Estimand of Interest Fill in the code to calculate the ATE using our updated information for Q:

```
# calculate ATE
# ----------
tmle_ate <- mean(Q_1_update - Q_0_update)
tmle_ate
```

```
## [1] 0.2338486
```

Step 6: Calculate Standard Errors for CIs and p-values And finally calculate the standard errors and p-values:

```
# calculate standard errors and p-values
# ----------
infl_fn <- (Y - Q_A_update) * H_A + Q_1_update - Q_0_update - tmle_ate
```

```
# calculate ATE
# ----------
tmle_se <- sqrt(var(infl_fn)/nrow(data_obs))

# confidence intervals
conf_low <- tmle_ate - 1.96*tmle_se
conf_high <- tmle_ate + 1.96*tmle_se

# p-values
pval <- 2 * (1 - pnorm(abs(tmle_ate / tmle_se)))

# view
tmle_ate
```

```
## [1] 0.2338486
```

```
conf_low
```

```
## [1] 0.1974691
```

```
conf_high
```

```
## [1] 0.2702281
```

```
pval
```

```
## [1] 0
```

Via TMLE Package

Thankfully, we do not need to take all of these steps every time we use the TMLE framework. The `tmle()` function handles all of this for us! Notice how we get a similar ATE from running this function:

```
# set seed for reproducibility
set.seed(1000)

# implement above all in one step using tmle
# ----------
tmle_fit <-
  tmle::tmle(Y = Y,                    # outcome
             A = A,                    # treatment
             W = W,                    # baseline covariates
             Q.SL.library = sl_libs, # libraries for initial estimate
             g.SL.library = sl_libs) # libraries for prob to be in treatment

# view results
tmle_fit
```

```
##  Additive Effect
##    Parameter Estimate:  0.19172
##    Estimated Variance:  0.00069196
##               p-value:  3.1378e-13
##     95% Conf Interval:  (0.14016, 0.24328)
##
##  Additive Effect among the Treated
##    Parameter Estimate:  0.12664
##    Estimated Variance:  0.0074648
##               p-value:  0.14273
##     95% Conf Interval:  (-0.042702, 0.29598)
##
##  Additive Effect among the Controls
##    Parameter Estimate:  0.41387
##    Estimated Variance:  0.00064296
##               p-value:  <2e-16
##     95% Conf Interval:  (0.36417, 0.46357)
```

LTMLE

One of the main shortcomings of the TMLE framework is that it can only handle baseline covariates (covariates measured before treatment). It also requires that intervention happens at a single time point. The `ltmle` library allows us to relax some of these restrictions so that we can handle time-dependent confounders. The basic setup for the LTMLE function is that we need ot set up our W, A, and Y, and then pass it to the `ltmle()` function:

```
# process data
# ----------
data_obs_ltmle <-
  data_obs %>%
  # need to specify W1, W2, etc
  rename(W1 = tax, W2 = age, W3 = crim) %>%
  select(W1, W2, W3, A, Y)

# ltmle implementation
# ----------
result <- ltmle(data_obs_ltmle, # dataset
                Anodes = "A",    # vector that shows treatment
                Ynodes = "Y",    # vector that shows outcome
                abar = 1)
# view
result
```

```
## Call:
## ltmle(data = data_obs_ltmle, Anodes = "A", Ynodes = "Y", abar = 1)
##
## TMLE Estimate:  0.6658262
```

Single Time Point

Imagine if there was a time ordering to our data, in particular a model like:

$$W1->W2->W3->A->Y$$

Let's simulate some data that captures this relationship. Notice how we added two arguments here, `Lnodes` and `Sl.library`. `SL.library` should look familiar now, and `Lnodes` refers to a "time varying covariate". For now we're going to leave this as a NULL, but we'll see how to use it soon.

```r
# create function
# ----------
rexpit <- function(x) rbinom(n=length(x), size=1, prob=plogis(x))

# specify model parameters
# ----------
n <- 1000
W1 <- rnorm(n)
W2 <- rbinom(n, size=1, prob=0.3)
W3 <- rnorm(n)
A <- rexpit(-1 + 2 * W1 + W3)
Y <- rexpit(-0.5 + 2 * W1^2 + 0.5 * W2 - 0.5 * A + 0.2 * W3 * A - 1.1 * W3)
data <- data.frame(W1, W2, W3, A, Y)

# implement above all in one step using tmle
# ----------
result <- ltmle(data,
                Anodes="A",
                Lnodes=NULL,      # no Lnodes
                Ynodes="Y",
                abar=1,
                SL.library=sl_libs)
# view
result
```

```
## Call:
## ltmle(data = data, Anodes = "A", Lnodes = NULL, Ynodes = "Y",
##     abar = 1, SL.library = sl_libs)
##
## TMLE Estimate:  0.5499798
```

Longitudinal Data Structure

Now imagine we instead of a time ordering like this:

$$W->A1->L->A2->Y$$

We can simulate some more data to reflect this structure, and then fit a `ltme()` function. Notice how now we have "L" nodes. In the data structure specified above, we have some baseline covariates that occur before the first treatment, some time dependent covariate that comes after the first treatment but before the second, then a second treatment and then outcome. Notice how we can now add `L` nodes that occur in between the two treatments so that we can deal with this "time-dependent confounding".

```r
# specify model parameters
# ----------
```

```r
n <- 1000
W <- rnorm(n)
A1 <- rexpit(W)
L <- 0.3 * W + 0.2 * A1 + rnorm(n)
A2 <- rexpit(W + A1 + L)
Y <- rexpit(W - 0.6 * A1 + L - 0.8 * A2)
data <- data.frame(W, A1, L, A2, Y)

# implement ltmle
# ----------
ltmle(data,
      Anodes=c("A1", "A2"),  # two treatment variables
      Lnodes="L",            # L indicator
      Ynodes="Y",            # outcome
      abar=c(1, 1),          # treatment indicator in Anodes vector
      SL.library = sl_libs)
```

```
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
```

```
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
## Error in lognet(xd, is.sparse, ix, jx, y, weights, offset, alpha, nobs,  :
##   one multinomial or binomial class has 1 or 0 observations; not allowed
## Error in pred[, "1"] : subscript out of bounds
```

```
## Call:
## ltmle(data = data, Anodes = c("A1", "A2"), Lnodes = "L", Ynodes = "Y",
##     abar = c(1, 1), SL.library = sl_libs)
##
## TMLE Estimate:  0.2885667
```

The `ltmle` vignette provides even more examples, including for censored data. The main concept here is that `ltmle` allows for causal estimates in observational data in complicated treatment regimes such as staggered adoption, subject dropout, multiple treatments, etc.

Concluding Thoughts

We have covered a lot of ground this year! To recap:

- Reproducible Data Science
  - GitHub/version control
  - Collaborative data science
- Machine Learning
  - Supervised Learning
    - Regression
    - Classification
  - Unsupervised Learning
    - Dimensionality Reduction
    - Clustering/grouping
- Text Analysis
  - Text preprocessing

    - Dictionary methods, word2vec, sentiment analysis
    - Prediction
- Causal Inference
    - Matching
    - Regression Discontinuity
    - Diff-in-Diffs/Synthetic Control
    - Sensitivity Analysis
    - SuperLearner/Targeted Maximum Likelihood Estimation

This covers a lot of computational social science! But there's still more. As you're thinking about where to go next, I recommend exploring these areas:

- Computational Tools/Data Acquisition
    - Web scraping (selenium)
    - APIs
    - Bash/CLI
    - XML/HTML parsing (BeautifulSoup)
- High Performance Computing
    - Parallel processing. We mostly covered "embarrassingly parallel" techniques, but there are other more advanced ones as well
    - Amazon Web Services (AWS)/Microsoft Azure cloud computing
    - Secure File Transfer Protocol (SFTP)
- Deep Learning
    - We covered some of the basics but more applications would be good
    - GPU acceleration
    - Deep Learning for text analysis (Bidirectional Encoder Representations from Transformers (BERT))
- Machine Learning and Causal Inference
    - Meta-estimators (like TMLE)
    - Heterogeneous Treatment Effects

Appendix for SL & LTMLE

a) We can now plot our AUC-ROC curve:

$$Sensitivity = TruePositiveRate = Recall = \frac{TP}{TP + FN}$$

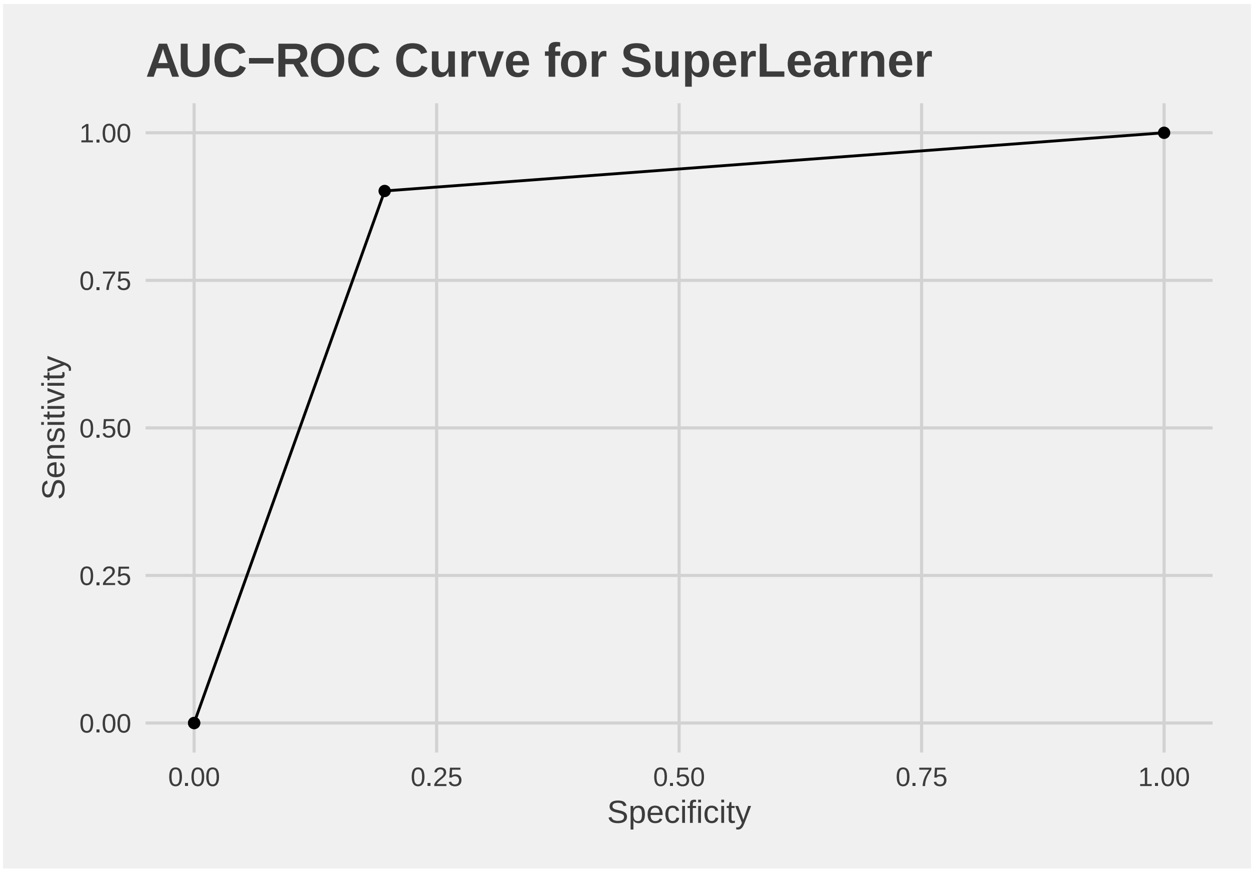
$$Specificity = TrueNegativeRate = \frac{TN}{TN + FP}$$

```
# process data
# ----------
validation_fct <-
  validation %>%
  # treat as factor
  mutate(obs2 = as_factor(obs)) %>%
  glimpse()
```

```
## Rows: 127
## Columns: 3
## $ obs  <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0,~
## $ pred <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0,~
## $ obs2 <fct> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0,~
```

```
# ROC curve
# ----------
```

```r
roc_df <- roc_curve(validation_fct,
                    pred,
                    truth = obs2)

# plot
# ----------
roc_df %>%
  ggplot() +
  geom_point(aes(x = specificity, y = 1 - sensitivity)) +
  geom_line(aes(x = specificity, y = 1 - sensitivity)) +
  theme_fivethirtyeight() +
  theme(axis.title = element_text()) +
  ggtitle('AUC-ROC Curve for SuperLearner') +
  xlab('Specificity') +
  ylab('Sensitivity')
```


AUC−ROC Curve for SuperLearner
Sensitivity
Specificity