Estimation with Potential Outcomes and Instrumental Variables

Caleb Leedy

March 29, 2023

Goals

- To understand how to make causal simulation studies,
- To see algorithms that utilize the potential outcomes framework,
- To use an instrumental variable to solve the unconfoundedness problem, and
- To identify weaknesses of existing approaches.

Generating the Data

```
# For data cleaning and manipulation
library(dplyr)
library(rlang)
library(stringr)
library(tidyr)
library(purrr)
library(knitr)

# For parallelization
library(doParallel)
library(doRNG)
library(parallelly)

# For additional aipw and iv estimators
library(grf)
library(ivreg)

expit <- function(x) {1 / (1 + exp(-x))}

set.seed(1)
```

Data Generating Models

First, we create a data generating function for later simulations of the potential outcomes framework. This function is designed to be sufficiently general in that a user can modify the function inputs to easily test different simulation settings.

```
#' The create_data function generates different kinds of data for the following
#' tests.
#'
#' @param n An integer for the number of samples
#' @param outcome_expr A string
```

```r
#' @param response_str A string
#' @param tau_str A string
#'
#' @details outcome_str, response_str, and tau_str can only display
#' functions of x1, x2, x3, x4, x5, x6, x7, x8, x9, x10.
create_data <- function(n, outcome_str, response_str, tau_str,
                        show_pi_quantiles = FALSE) {

  # Steps to generate data:
  # 1. Generate covariates
  # 2. Generate potential outcomes
  # 3. Assign treatment

  # 1. Generate covariates
  # All distributions have mean 0, variance 1.
  x1 <- rnorm(n)
  x2 <- rnorm(n)
  x3 <- rnorm(n)
  x4 <- runif(n, -1, 1) / sqrt(1 / 3)
  x5 <- (rchisq(n, df = 1) - 1) / sqrt(2)
  x6 <- (rchisq(n, df = 3) - 3) / sqrt(6)
  x7 <- (rchisq(n, df = 10) - 10)  / sqrt(20)
  x8 <- (2 * rbinom(n, 1, prob = 0.5) - 1)
  x9 <- rpois(n, 1) - 1
  x10 <- (rpois(n, 5) - 5) / sqrt(5)

  # 2. Generate potential outcomes
  # NOTE: We are assuming an additive treatment effect.
  # NOTE: We also have normal noise in our potential outcomes.
  tau_vec <- eval(rlang::parse_expr(tau_str))
  out_vec <- eval(rlang::parse_expr(outcome_str))
  y0_vec <- out_vec + rnorm(n)
  y1_vec <- tau_vec + out_vec + rnorm(n)

  # 3. Assign treatments
  # NOTE: Since response_str is user determined, we do not assume MCAR, MAR, or
  # NMAR a priori.
  pi_vec <- expit(eval(rlang::parse_expr(response_str)))
  a_vec <- rbinom(n, 1, pi_vec)

  if (show_pi_quantiles) {
    print(paste0("The 1% quantile of pi_vec is: ",
                 round(quantile(pi_vec, 0.01), 4)))
    print(paste0("The 50% quantile of pi_vec is: ",
                 round(quantile(pi_vec, 0.5), 4)))
    print(paste0("The 99% quantile of pi_vec is: ",
                 round(quantile(pi_vec, 0.99), 4)))

  }

  # From SUTVA: Y = Y(1) * A + Y(0) * (1 - A)
  y_obs <- y1_vec * a_vec + y0_vec * (1 - a_vec)
```

```
  return(tibble(Y = y_obs,
                X1 = x1, X2 = x2, X3 = x3, X4 = x4, X5 = x5,
                X6 = x6, X7 = x7, X8 = x8, X9 = x9, X10 = x10,
                A = a_vec,
                pi_vec = pi_vec,
                tau = tau_vec))

}
```

Identifying Causal Effects

To estimate the average treatment effect, we implement three algorithms following the procedure in the Potential Outcomes slides. These algorithms use estimate a linear outcome model, a logistic response model, and an AIPW model that combines the linear outcome and logistic response model. Again, these functions are designed to be sufficiently general so that users can define the functional forms (coefficients) in these models via the parameters. To do this input a string with R formula syntax as a parameter to `mod_str` or `out_mod_str` or `resp_mod_str`. See the simulations below for more examples.

```
#' This function constructs an outcome model to estimate the ATE.
#'
#' @param df A data frame generated from the create_data function.
#' @param mod_str A string. This string determines the functional form used in
#' the linear outcome model. See the simulations below for more examples.
outcome_mod <- function(df, mod_str) {

  mod_str <- str_to_upper(mod_str)

  # We fix the model string if the user did not start it with "Y~".
  if (!str_detect(mod_str, "~")) {
    mod_str <- str_c("Y ~", mod_str)
  }

  # OLS models
  y1_mod <- lm(mod_str, data = dplyr::filter(df, A == 1))
  y0_mod <- lm(mod_str, data = dplyr::filter(df, A == 0))

  y1_pred <- predict(y1_mod, newdata = df)
  y0_pred <- predict(y0_mod, newdata = df)

  tau_hat <- mean(y1_pred) - mean(y0_pred)

  # TODO: Add variance
  # This is a temporary variance that is clearly incorrect.
  est_var <- -1

  return(list(tau = tau_hat, var = est_var))
}

#' This function constructs a response model to estimate the ATE.
#'
#' @param df A data frame generated from the create_data function.
#' @param mod_str A string. This string determines the functional form used in
#' the logistic response model. See the simulations below for more examples.
```

```r
response_mod <- function(df, mod_str) {

  mod_str <- str_to_upper(mod_str)

  # We fix the model string if the user did not start it with "A~".
  if (!str_detect(mod_str, "~")) {
    mod_str <- str_c("A ~", mod_str)
  }

  # Logistic Model
  pi_mod <- glm(mod_str, data = df, family = binomial(link = logit))
  pi_hat <- pi_mod$fitted.values

  tau_hat <- mean(df$Y * df$A / pi_hat) - mean(df$Y * (1 - df$A) / (1 - pi_hat))

  # TODO: Add variance
  # This is a temporary variance that is clearly incorrect.
  est_var <- -1

  return(list(tau = tau_hat, var = est_var))
}

#' This function constructs an AIPW model to estimate the ATE.
#'
#' @param df A data frame generated from the create_data function.
#' @param out_mod_str A string. This string determines the functional form used
#' in the linear outcome model.
#' @param resp_mod_str A string. This string determines the functional form used
#' in the logistic response model. See the simulations below for more examples.
custom_aipw <- function(df, out_mod_str, resp_mod_str) {

  # Outcome Model Estimation
  out_mod_str <- str_to_upper(out_mod_str)

  # We fix the model string if the user did not start it with "Y~".
  if (!str_detect(out_mod_str, "~")) {
    out_mod_str <- str_c("Y ~", out_mod_str)
  }

  # OLS models
  y1_mod <- lm(out_mod_str, data = dplyr::filter(df, A == 1))
  y0_mod <- lm(out_mod_str, data = dplyr::filter(df, A == 0))

  y1_pred <- predict(y1_mod, newdata = df)
  y0_pred <- predict(y0_mod, newdata = df)

  # Response Model Estimation
  resp_mod_str <- str_to_upper(resp_mod_str)

  # We fix the model string if the user did not start it with "A~".
  if (!str_detect(resp_mod_str, "~")) {
    resp_mod_str <- str_c("A ~", resp_mod_str)
  }
```

```
  # Logistic Model
  pi_mod <- glm(resp_mod_str, data = df, family = binomial(link = logit))
  pi_hat <- pi_mod$fitted.values

  # Combine for AIPW
  mu1_est <- mean(y1_pred + df$A / pi_hat * (df$Y - y1_pred))
  mu0_est <- mean(y0_pred + (1 - df$A) / (1 - pi_hat) * (df$Y - y0_pred))

  tau_hat <- mu1_est - mu0_est

  # TODO: Add variance
  # This is a temporary variance that is clearly incorrect.
  est_var <- -1

  return(list(tau = tau_hat, var = est_var))
}
```

Running the Simulations

The following code simulates the estimation of the average treatment effect using different algorithms that use the potential outcomes framework. I encourage readers to change some of the parameters to explore the weaknesses of each algorithm. The parameters are the following:

- `true_tau`: This is the magnitude of the true average treatment effect. The way that the data is generated it could be a function of another variable but this has not been tested *and* the subsequent t-test will only work with a constant tau.
- `y_mod_str`: This stringdefines the model '$Y = X\beta$'. As currently written, the different distributions of x_i are only available up from '$x_1, \ldots, x_{10}$' with different distributions seen in the code for the function `create_data`.
- `response_model`: This string determines the probability that each element is selected for treatment. Most algorithms do better with balanced data. Also, we have assumed that the probability of being in a treatment group is bounded away from zero. If this does not occurs, the algorithms will probably perform worse.
- `out_mod_form`: This string determines the functional form of any outcome model that is being used.
- `resp_mod_form`: This string determines the functional form of any response model that is being used.

If you have more computing power, one can increase the number of observations `n_obs` or the number of Monte Carlo simulations `B`. However, when running in local interactive use, I recommend commenting out the causal forest because it is much longer then the other algorithms.

Additional scenarios I recommend trying include: unbalanced data, NMAR, model misspecification in the outcome and response models, asymmetric covariates, or any combination of the above. All of the covariates have mean zero and variance one because I thought that initially this would be important for understanding the true value; however, I don't think that this matters. If desired, feel free to change some (or add additional) parameters to the `create_data` function to use.

```
clust <- makeCluster(availableCores() - 2)
registerDoParallel(clust)

B <- 1000

ptoc_res <-
  foreach(iter = 1:B, .packages = c("stringr", "dplyr", "grf")) %dorng% {

    if (iter %% 100 == 0) {
```

```
    print(paste0("Iter: ", iter))
  }

  # Parameters
  true_tau <- 2
  n_obs <- 1000

  # These are true values
  y_mod_str <- "3 + x1 - 0.5 * x2 + 1.2 * x3"
  response_model <- "x1 + 1.2 * x2"

  # These are functional forms of estimated models
  # Use formula notation.
  out_mod_form <- "x1 + x2 + x3"
  resp_mod_form <- "x1 + x2"

  # Run models
  df <- create_data(n = n_obs,
                    outcome_str = y_mod_str,
                    response_str = response_model,
                    tau_str = as.character(true_tau))

  out_res <- outcome_mod(df, mod_str = out_mod_form)   # Correctly specified
  res_res <- response_mod(df, mod_str = resp_mod_form) # Correctly specified
  cust_aipw_res <-
    custom_aipw(df,
                out_mod_str = out_mod_form,            # Correctly specified
                resp_mod_str = resp_mod_form)          # Correctly specified

  # Causal forest
  # NOTE: For more interactive local use, I recommend not running the causal
  # forest. It takes significantly longer than the other algorithms.
  cf_res <- -1
  x_mat <-
    model.matrix(as.formula(paste0("~", str_to_upper(out_mod_form))),
                 data = df)
  cf_mod <- causal_forest(X = x_mat, Y = df$Y, W = df$A)
  cf_res <- average_treatment_effect(cf_mod, target.sample = "all")

  tibble(true = true_tau,
         out = out_res$tau,
         resp = res_res$tau,
         aipw = cust_aipw_res$tau,
         cf = cf_res[1])

} %>% bind_rows()

stopCluster(clust)
```

After running the simulations we test for unbiasedness in the mean.

```
# NOTE: The t-test only works for a constant tau.
ptoc_res %>%
  summarize(
```

```
    mean_out = mean(out),
    mean_resp = mean(resp),
    mean_aipw = mean(aipw),
    mean_cf = mean(cf),
    var_out = var(out),
    var_resp = var(resp),
    var_aipw = var(aipw),
    var_cf = var(cf),
    tstat_out = (mean(out) - mean(true)) / (sqrt(var(out) / nrow(ptoc_res))),
    tstat_resp = (mean(resp) - mean(true)) / (sqrt(var(resp) / nrow(ptoc_res))),
    tstat_cf = (mean(cf) - mean(true)) / (sqrt(var(cf) / nrow(ptoc_res))),
    tstat_aipw = (mean(aipw) - mean(true)) / (sqrt(var(aipw) / nrow(ptoc_res)))
  ) %>%
  pivot_longer(cols = everything(),
               names_to = c(".value", "algorithm"),
               names_pattern = "(.*)_(.*)") %>%
  mutate(pval = ifelse(tstat < 0,
                       pt(tstat, df = nrow(ptoc_res)),
                       pt(-tstat, df = nrow(ptoc_res)))) %>%
  knitr::kable(digits = 3)
```

algorithm	mean	var	tstat	pval
out	1.999	0.005	-0.329	0.371
resp	2.004	0.071	0.436	0.331
aipw	2.000	0.008	0.072	0.471
cf	1.986	0.007	-5.381	0.000

Instrumental Variables

After simulating algorithms in the potential outcomes framework, we modify our approach so that we have an instrumental variable setup. Overall, the structure of the code is similar as above. We start with a function to create the data, then we move to estimation. Unlike the previous simulations, however, this simulation only uses algorithms from existing packages.

In this IV setup, we have the assigned treatment, `A`, which can differ from the observed treatment, `obs_treat`. Since some units always do the same thing no matter what, the observed treatment is correlated with the outcome error. So we use the assigned treatment `A` as an instrument. The bias that occurs is to be expected because IV methods are not unbiased.

Overall, I have less confidence in this simulation then the previous one. If you find any errors, please let me know and (ideally) submit a pull request to the GitHub repo.

In addition to some of the same kind of modifications one can make to the previous simulation, one can also run the simulation with the instrument being weaker, which makes the problem harder.

```
#' The create_iv function generates different kinds of data for estimating IV
#' problems.
#'
#' @param n An integer for the number of samples
#' @param outcome_expr A string
#' @param response_str A string
#' @param tau_str A string
#' @param always_frac A double
#' @param never_frac A double
```

```r
#' @param show_pi_quantiles A boolean
#'
#' @details outcome_str, response_str, and tau_str can only display
#' functions of x1, x2, x3, x4, x5, x6, x7, x8, x9, x10.
#' @note always_frac + never_frac <= 1 and both should be nonnegative.
create_iv <- function(n, outcome_str, response_str, tau_str,
                      always_frac = 0,
                      never_frac = 0.2,
                      show_pi_quantiles = FALSE) {

  # Steps to generate data:
  # 1. Generate covariates
  # 2. Generate potential outcomes
  # 3. Assign treatment
  # 4. Strata individual by group type (compliers, always/never takers)

  # 1. Generate covariates
  # All distributions have mean 0, variance 1.
  x1 <- rnorm(n)
  x2 <- rnorm(n)
  x3 <- rnorm(n)
  x4 <- runif(n, -1, 1) / sqrt(1 / 3)
  x5 <- (rchisq(n, df = 1) - 1) / sqrt(2)
  x6 <- (rchisq(n, df = 3) - 3) / sqrt(6)
  x7 <- (rchisq(n, df = 10) - 10)  / sqrt(20)
  x8 <- (2 * rbinom(n, 1, prob = 0.5) - 1)
  x9 <- rpois(n, 1) - 1
  x10 <- (rpois(n, 5) - 5) / sqrt(5)

  # 2. Generate potential outcomes
  # NOTE: We are assuming an additive treatment effect.
  # NOTE: We also have normal noise in our potential outcomes.
  tau_vec <- eval(rlang::parse_expr(tau_str))
  out_vec <- eval(rlang::parse_expr(outcome_str))
  y0_vec <- out_vec + rnorm(n)
  y1_vec <- tau_vec + out_vec + rnorm(n)

  # 3. Assign treatments
  # NOTE: Since response_str is user determined, we do not assume MCAR, MAR, or
  # NMAR a priori.
  pi_vec <- expit(eval(rlang::parse_expr(response_str)))
  a_vec <- rbinom(n, 1, pi_vec)

  if (show_pi_quantiles) {
    print(paste0("The 1% quantile of pi_vec is: ",
                 round(quantile(pi_vec, 0.01), 4)))
    print(paste0("The 50% quantile of pi_vec is: ",
                 round(quantile(pi_vec, 0.5), 4)))
    print(paste0("The 99% quantile of pi_vec is: ",
                 round(quantile(pi_vec, 0.99), 4)))

  }
```

```
  # From SUTVA: Y = Y(1) * A + Y(0) * (1 - A)
  y_obs <- y1_vec * a_vec + y0_vec * (1 - a_vec)

  res_df <- tibble(Y = y_obs,
                   Y0 = y0_vec,
                   Y1 = y1_vec,
                   X1 = x1, X2 = x2, X3 = x3, X4 = x4, X5 = x5,
                   X6 = x6, X7 = x7, X8 = x8, X9 = x9, X10 = x10,
                   A = a_vec,
                   pi_vec = pi_vec,
                   tau = tau_vec)

  # 4. Group based on strata
  complier_frac <- 1 - always_frac - never_frac
  group_cuts <- cumsum(c(complier_frac, always_frac, never_frac))
  group_id <- runif(n)

  # Compliers = 0, Always takers = 1, Never takers = 2
  group_id <- purrr::map_dbl(group_id, function(x) {sum(x > group_cuts)})

  res_df %>%
    mutate(group_id = group_id) %>%
    mutate(Y = case_when(group_id == 0 ~ Y,
                         group_id == 1 ~ Y1,
                         group_id == 2 ~ Y0)) %>%
    mutate(obs_treat = ifelse(group_id == 0, A,
                              ifelse(group_id == 1, 1, 0)))

}

# In this simulation the instrument is A, but we also observe obs_treat.
```

```
clust <- makeCluster(availableCores() - 2)
registerDoParallel(clust)

B <- 1000

iv_res <-
  foreach(iter = 1:B,
          .packages = c("stringr", "dplyr", "ivreg", "grf")) %dorng% {

    if (iter %% 100 == 0) {
      print(paste0("Iter: ", iter))
    }

    # Parameters
    true_tau <- 2
    n_obs <- 1000

    # These are true values
    y_mod_str <- "3 + x1 - 0.5 * x2 + 1.2 * x3"
    response_model <- "x1 + 1.2 * x2"

    # These are functional forms of estimated models
```

```
    # Use formula notation.
    out_mod_form <- "x1 + x2 + x3"
    resp_mod_form <- "x1 + x2"

    df <- create_iv(n = 1000,
                    outcome_str = y_mod_str,
                    response_str = response_model,
                    tau_str = as.character(true_tau))

    # 2SLS using ivreg package
    ivreg_form <-
      as.formula(paste0("Y ~",  str_to_upper(out_mod_form), " | obs_treat | A"))
    twosls_res <- ivreg(ivreg_form, data = df)

    # LATE using grf package
    x_mat <-
      model.matrix(as.formula(paste0("~", str_to_upper(out_mod_form))),
                   data = df)
    iv_for <-
      instrumental_forest(X = x_mat, Y = df$Y, W = df$obs_treat, Z = df$A)
    for_res <- average_treatment_effect(iv_for)

    tibble(true = true_tau,
           twosls = twosls_res$coefficients[2],
           cf = for_res[1])

  } %>%
  bind_rows()

stopCluster(clust)

iv_res %>%
  summarize(
    mean_twosls = mean(twosls),
    mean_cf = mean(cf),
    var_twosls = var(twosls),
    var_cf = var(cf),
    tstat_twosls = (mean(twosls) - mean(true)) / (sqrt(var(twosls) / nrow(iv_res))),
    tstat_cf = (mean(cf) - mean(true)) / (sqrt(var(cf) / nrow(iv_res))),
  ) %>%
  pivot_longer(cols = everything(),
               names_to = c(".value", "algorithm"),
               names_pattern = "(.*)_(.*)") %>%
  mutate(pval = ifelse(tstat < 0,
                       pt(tstat, df = nrow(iv_res)),
                       pt(-tstat, df = nrow(iv_res)))) %>%
  knitr::kable(digits = 3)
```

algorithm	mean	var	tstat	pval
twosls	2.003	0.009	1.141	0.127
cf	1.990	0.011	-3.163	0.001