Below we set up a python virtual environment and install the *dowhy* dependencies:

# create a new environment

virtualenv\_create("r-reticulate")

# install python packages into virtualenv

virtualenv\_install("r-reticulate", c("numpy", "scipy", "scikit-learn", "pandas", "dowhy"))

Be sure to use python 3+. If you’re using windows you are out of luck as these support conda environments only and I’m not sure conda support *dowhy*.

Next, we import the dowhy module right into our R session!

dowhy <- import("dowhy")

For simplicity, we simulate a dataset with linear relationships between common causes and treatment, and common causes and outcome.

Beta is the true causal effect and is equal to 10. Below we generate the linear dataset:

data <- dowhy$datasets$linear\_dataset(

beta = 10L,

num\_common\_causes = 5L,

num\_instruments = 2L,

num\_effect\_modifiers = 1L,

num\_samples = 10000L,

treatment\_is\_binary = T

)

df\_r <- py\_to\_r(data[["df"]]) # to be used later

Below we can see the first few rows:

data["df"]$df$head() %>% pandoc.table(split.tables = Inf)

| **X0** | **Z0** | **Z1** | **W0** | **W1** | **W2** | **W3** | **W4** | **v0** | **y** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0.1236 | 0 | 0.741 | 1.659 | 0.2614 | -0.2472 | 0.1899 | -0.6062 | TRUE | 18.24 |
| 0.3468 | 0 | 0.5111 | 0.2102 | -0.9326 | 0.3437 | -0.9736 | -0.8639 | TRUE | 6.882 |
| 0.8701 | 0 | 0.3066 | 1.31 | 1.965 | 0.2338 | -0.1431 | -0.9677 | TRUE | 22.94 |
| -0.2349 | 0 | 0.2999 | 0.06766 | 0.7691 | -1.175 | -1.573 | -1.966 | FALSE | -4.887 |
| -0.045 | 0 | 0.4588 | 0.9863 | -0.0372 | -0.9543 | -0.5231 | -1.025 | FALSE | 1.24 |

We are interested with estimating the causal effect of \(v0\) (a binary treatment) on \(y\) (10 in this case). The *dowhy* library streamlines the process of estimating and validating the causal estimate by introducing a flow consisting of 4 key steps. The first is enumerating our assumed causal model, as encoded by a DAG.

**1) Enumerate the assumed causal model**

The above data object contains the underlying DAG representation. I’ve found that converting dot graphs to dagitty format is pretty straight forward and we’ll use that to plot the graph:

dagitty\_graph <- data[["dot\_graph"]]

dagitty\_graph <- gsub("digraph", "dag", dagitty\_graph)

dagitty\_graph <- dagitty(dagitty\_graph)

# let's arrange the nodes to conform with what is shown in the dowhy example

coordinates(dagitty\_graph) <- list(

x = c(

U = 0, W0 = 1, W1 = 2, W2 = 3, W3 = 6, W4 = 7,

X0 = 6.5, Z0 = 4, Z1 = 5, v0 = 3.5, y = 3

),

y = c(

U = 0, W0 = 0, W1 = 0, W2 = 0, W3 = 0, W4 = 0,

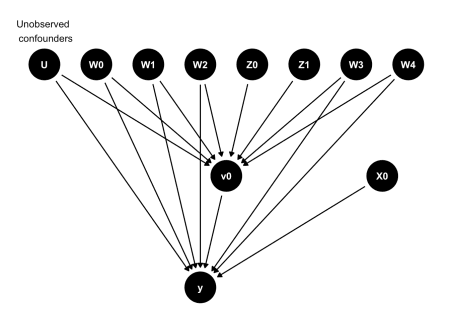
X0 = -1, Z0 = 0, Z1 = 0, v0 = -1, y = -2

)

)

ggdag(tidy\_dagitty(dagitty\_graph)) + theme\_dag\_blank() +

annotate(geom = "text", x = 0, y = 0.3, label = "Unobserved \n confounders")



We enumerate the causal model using the following code:

model <- dowhy$CausalModel(

data = data[["df"]],

treatment = data[["treatment\_name"]],

outcome = data[["outcome\_name"]],

graph = data[["gml\_graph"]]

)

**2) Enumerate identification strategies**

The next step is enumerating the identification strategies available given the above graph:

identified\_estimand <- model$identify\_effect(proceed\_when\_unidentifiable = T)

print(identified\_estimand)

## Estimand type: nonparametric-ate

## ### Estimand : 1

## Estimand name: backdoor

## Estimand expression:

## d

## ─────(Expectation(y|W1,W0,W2,W4,W3))

## d[v₀]

## Estimand assumption 1, Unconfoundedness: If U→{v0} and U→y then P(y|v0,W1,W0,W2,W4,W3,U) = P(y|v0,W1,W0,W2,W4,W3)

## ### Estimand : 2

## Estimand name: iv

## Estimand expression:

## Expectation(Derivative(y, [Z0, Z1])\*Derivative([v0], [Z0, Z1])\*\*(-1))

## Estimand assumption 1, As-if-random: If U→→y then ¬(U →→{Z0,Z1})

## Estimand assumption 2, Exclusion: If we remove {Z0,Z1}→{v0}, then ¬({Z0,Z1}→y)

We can see that there are 2 possible ways of estimating the causal effect: Either using the backdoor criteria or with instrument variables (IV).

We can really see how *dowhy* how does a great job at making causal discovery transparent and explicit in each step of the process.

**3) Estimate the causal effect**

Next up is doing the actual estimation.

We’ll first try the IV estimand:

causal\_estimate <- model$estimate\_effect(

identified\_estimand = identified\_estimand,

method\_name = "iv.instrumental\_variable"

)

print(paste0("Causal Estimate is ", causal\_estimate$value))

## [1] "Causal Estimate is 14.5722987087289"

Next, let’s try the backdoor estimand, using propensity score stratification:

causal\_estimate <- model$estimate\_effect(

identified\_estimand = identified\_estimand,

method\_name = "backdoor.propensity\_score\_stratification"

)

print(paste0("Causal Estimate is ", causal\_estimate$value))

## [1] "Causal Estimate is 10.2974994551257"

**4) Validate estimated causal effect**

The last step consists of analyzing the level of confidence we have in the estimated causal effect. We’ll proceed with the backdoor estimate.

Let’s start by adding a random (observed) confounder:

res\_random <- model$refute\_estimate(

estimand = identified\_estimand,

estimate = causal\_estimate,

method\_name = "random\_common\_cause"

)

print(res\_random)

## Refute: Add a Random Common Cause

## Estimated effect:(10.297499455125735,)

## New effect:(10.329135890941698,)

We can see the result is pretty stable. This means our sample size (10,000) is probably large enough to accommodate further confounders without loss of accuracy.

Next, let’s see how adding an unobserved confounder can change our estimate. In the code below the effect\_strength\_on\_treatment argument denotes the probability of the confounder flipping the treatment from 0 to 1 (or vice verca). The effect\_strength\_on\_outcome denotes the linear coefficient of the confounder on the outcome.

res\_unobserved <- model$refute\_estimate(

estimand = identified\_estimand,

estimate = causal\_estimate,

method\_name = "add\_unobserved\_common\_cause",

confounders\_effect\_on\_treatment = "binary\_flip",

confounders\_effect\_on\_outcome = "linear",

effect\_strength\_on\_treatment = 0.01,

effect\_strength\_on\_outcome = 0.02

)

print(res\_unobserved)

## Refute: Add an Unobserved Common Cause

## Estimated effect:(10.297499455125735,)

## New effect:(9.992195972192327,)

When using the effect strength values given in the example the effect on our estimate seems modest. Let’s try adding a stronger confounder:

res\_unobserved <- model$refute\_estimate(

estimand = identified\_estimand,

estimate = causal\_estimate,

method\_name = "add\_unobserved\_common\_cause",

confounders\_effect\_on\_treatment = "binary\_flip",

confounders\_effect\_on\_outcome = "linear",

effect\_strength\_on\_treatment = 0.05,

effect\_strength\_on\_outcome = 1

)

print(res\_unobserved)

## Refute: Add an Unobserved Common Cause

## Estimated effect:(10.297499455125735,)

## New effect:(7.065846279246412,)

It’s pretty surprising to see how bad our estimate would be given a single un-observed confounder with what I’d consider mild confounding effect.

What’s weirder is that the confounder can affect only the treatment (setting effect\_strength\_on\_outcome = 0, essentially making it a non confounder) and still destabilize the estimate in pretty much the same way:

res\_unobserved <- model$refute\_estimate(

estimand = identified\_estimand,

estimate = causal\_estimate,

method\_name = "add\_unobserved\_common\_cause",

confounders\_effect\_on\_treatment = "binary\_flip",

confounders\_effect\_on\_outcome = "linear",

effect\_strength\_on\_treatment = 0.05,

effect\_strength\_on\_outcome = 0

)

print(res\_unobserved)

## Refute: Add an Unobserved Common Cause

## Estimated effect:(10.297499455125735,)

## New effect:(6.821708758664067,)

I’ll probably need to research the internal mechanics a bit more to understand how that’s possible.

Let’s try permuting the treatment (making it effectively a placebo) and see how our estimate changes:

res\_placebo <- model$refute\_estimate(

estimand = identified\_estimand,

estimate = causal\_estimate,

method\_name = "placebo\_treatment\_refuter",

placebo\_type = "permute"

)

print(res\_placebo)

## Refute: Use a Placebo Treatment

## Estimated effect:(10.297499455125735,)

## New effect:(0.1271016806333491,)

We can see that the estimated effect is essentially 0, which is what we’d like to see (this means our estimator doesn’t catch random noise as treatment effect).

Finally, let’s see how sensitive is our estimate to removal of 10% of the observations:

res\_subset <- model$refute\_estimate(

estimand = identified\_estimand,

estimate = causal\_estimate,

method\_name = "data\_subset\_refuter",

subset\_fraction = 0.9,

random\_seed = 1L

)

print(res\_subset)

## Refute: Use a subset of data

## Estimated effect:(10.297499455125735,)

## New effect:(10.333160141396403,)

Pretty stable. Thinking about this a bit more I believe this is very similar to using bootstrap sampling to evaluate the estimate variability. Let’s implement bootstrap sampling for variability estimation just for fun:

if (!"res.rds" %in% list.files("../../../")) {

set.seed(1)

M <- 100

res <- vector(length = M)

for (m in 1:M) {

bootstarpped\_data <- df\_r[[sample.int](http://sample.int)(

n = nrow(df\_r),

nrow(df\_r), replace = T

), ]

dat\_m <- r\_to\_py(bootstarpped\_data)

model\_m <- dowhy$CausalModel(

data = dat\_m,

treatment = data[["treatment\_name"]],

outcome = data[["outcome\_name"]],

graph = data[["gml\_graph"]]

)

identified\_estimand <- model\_m$identify\_effect(

proceed\_when\_unidentifiable = T

)

causal\_estimate <- model\_m$estimate\_effect(

identified\_estimand = identified\_estimand,

method\_name = "backdoor.propensity\_score\_stratification"

)

res[m] <- causal\_estimate$value

}

saveRDS(res, "../../../res.rds")

} else {

res <- readRDS("../../../res.rds")

}

res\_sd <- round(sd(res), 4)

data.frame(causal\_estimate = res) %>%

ggplot(aes(causal\_estimate)) + geom\_histogram(bins = 10) +

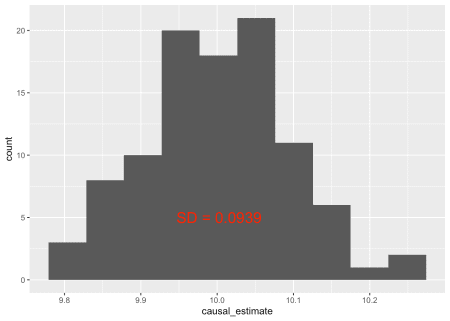
annotate(

geom = "text", x = mean(res), y = 5,

label = paste0("SD = ", res\_sd),

color = "red", size = 6

)



Oddly enough it would seem the estimate is pretty far off from 10.

The code above demonstrates how *reticulate* enables seamless simultaneous coding in Python and R.