Chapter 14 Coding Homework

Matt Malis

Updated 2024-06-10

Follow the below instructions and turn in both your code and results:

1. Load the `nsw_mixtape` data that can be found in the **causaldata** package associated with the book, or download it fromLoad the `dengue.csv` file provided to you, or from this site. Documentation on the variables is available through the package, or here.

Then, drop the `data_id` variable from the data.

```r
d = read.csv(
  url('https://vincentarelbundock.github.io/Rdatasets/csv/causaldata/nsw_mixtape.csv')
  ) |> tibble()
d = d |> dplyr::select(-data_id,-rownames)

# data is sorted with all treated obs first, followed by all control
#  changing this just to see if the matching functions require it
set.seed(123)
d = d |> slice_sample(n=nrow(d),replace=F)
```

Language-specific instructions:

- In R or Python, store the data set as `nsw`.
- In R, after loading the **tidyverse** you can drop using `select(-droppedvariable)`
- In Stata, it's `drop droppedvariable`
- In Python, after loading **pandas** it's 'nsw.drop('droppedvariable', axis = 1)

2. Let's see where we're at before we do any matching at all. `nsw_mixtape` is from an experiment (read that documentation!) so that should already put us in a pretty good place.

First, create a variable called `weight` in your data equal to 1 for all observations (weights that are all 1 will have no effect, but this will give us a clue as to how we could incorporate matching weights easily).

```r
d = d |> mutate(
  weight = 1
) |>
  rename(y=re78)
```

Second, write code that uses a set of given weights to estimate the effect of `treat` on `re78`, using `weight` as weights, and prints out a summary of the regression results. The easiest way to do this is probably weighted regression; see The Effect Section 13.4.1, but without any controls or predictors other than `treat`. **Keep in mind the standard errors on the estimate won't be quite right, since they won't account for the uncertainty in creating the weights.**

Table 1: Summary Statistics

treat	0			1		
Variable	N	Wt. Mean	Wt. SD	N	Wt. Mean	Wt. SD
age	260	25	7.1	185	26	7.2
educ	260	10	1.6	185	10	2
black	260	0.83	0.38	185	0.84	0.36
hisp	260	0.11	0.31	185	0.059	0.24
marr	260	0.15	0.36	185	0.19	0.39
nodegree	260	0.83	0.37	185	0.71	0.46
re74	260	2107	5688	185	2096	4887
re75	260	1267	3103	185	1532	3219
y	260	4555	5484	185	6349	7867

```r
lm(y ~ treat, weights=weight, data=d) |> coeftest()
```

```
##
## t test of coefficients:
##
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept)  4554.80     408.05 11.1625 < 2.2e-16 ***
## treat        1794.34     632.85  2.8353  0.004788 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Third, write code that creates and prints out a weighted balance table for all variables across values of `treat`, using `weight` as weighted. See The Effect Section 14.6.3. Don't worry about getting a table with tests of significant differences for now; just the means.

```r
library(vtable)
sumtable(d , group='treat', group.weight = 'weight')
```

Language-specific instructions:

- One easy way to get the balance table in R is with `sumtable()` in **vtable** by setting the `group` and `group.weight` options (and possibly `group.test`).
- One easy way to get the table in Stata is with `tabstat` using the `by()` option.
- One easy way to get the table in Python is with `.groupby.aggregate()`.

2b. Is there anything potentially concerning about the balance table, given that this is a randomized experiment where `treat` was randomly assigned?

3. Using all of the variables in the data except `treat` and `re78` as matching variables, perform 3-nearest-neighbor Mahalanobis distance matching with replacement and no caliper (The Effect 14.4.1) and calculate the post-matching average treatment on the treated effect of `treat` on `re78`.

Check the documentation of the function you use to be sure you're matching with replacement.

```r
library(Matching)
X = d |> dplyr::select(-y,-treat) |> as.matrix()
M = Match(Y=d$y, Tr = d$treat, X=X
          ,Weight=2 # for Mahalanobis distance
          ,M=3 # number of matches
          ,BiasAdjust=F
          ,exact=F
          ,replace=T
          )
summary(M)
```

```
##
## Estimate...  1865.2
## AI SE......  797.63
## T-stat.....  2.3385
## p.val......  0.019361
##
## Original number of observations..............  445
## Original number of treated obs...............  185
## Matched number of observations...............  185
## Matched number of observations  (unweighted).  671
##
## Number of obs dropped by 'exact' or 'caliper'  0
```

Language-specific notes:

- Due to a namespace conflict (the same name used for two different functions in different packages), loading the **Matching** package may make the `select()` function not work. So be sure to re-load the **tidyverse** again afterwards, or use `dplyr::select()` instead of `select()` to let R know which version of `select()` to use.
- The documentation isn't super clear, but Stata's `teffects nnmatch` does match with replacement by default.

4. Create a post-matching balance table showing balance for all the matching variables (you'll probably want to use the balance function designed to follow the matching function you used, from the same package). Write a sentence commenting on whether the balance looks good. You may have to read the documentation for the function you use to figure out what the results mean.

```r
matched_inds_weights = tibble(
  ind_t = M$index.treated,
  ind_c = M$index.control,
  w = M$weights
)

# verifying that all treated inds from original data are included in M$index.treated
all_trt_inds = which(d$treat==1)
table(all_trt_inds %in% M$index.treated)
```

```
##
## TRUE
##  185
```

```r
# each treated ind has between 3 and 11 matches, 3 is by far most common
matched_inds_weights |> group_by(ind_t) |> summarise(n=n())  |>
  dplyr::select(n) |> table()
```

```
## n
##   3   4   5   6   7   8   9  10  11
## 141  19  11   4   1   2   2   3   2
```

```r
# weights sum to 1 for each treated ind
matched_inds_weights |> group_by(ind_t) |> summarise(w_sum = sum(w))  |>
  dplyr::select(w_sum) |> table()
```

```
## w_sum
##   1
## 185
```

```r
# each control ind (included in M) has between 1 and 9 matches, 1 or 2 are most common
matched_inds_weights |> group_by(ind_c) |> summarise(n=n()) |>
  dplyr::select(n) |> table()
```

```
## n
##  1  2  3  4  5  6  7  8  9
## 52 74 32 26 12 20  6  5  1
```

```r
# weights do not sum to 1 for each control ind
matched_inds_weights |> group_by(ind_c) |> summarise(w_sum = sum(w))  |>
  dplyr::select(w_sum) |> summary()
```

```
##      w_sum
##  Min.   :0.1250
##  1st Qu.:0.3583
##  Median :0.6667
##  Mean   :0.8114
##  3rd Qu.:1.0000
##  Max.   :2.6667
```

```r
# example: for ind_c==4, we can see it accounts for
#  1/4 of the matches for ind_t = 51,
#  and 1/6 of the matches for ind_t = 189
matched_inds_weights |> filter(ind_c==4)
```

```
## # A tibble: 2 x 3
##   ind_t ind_c     w
##   <dbl> <dbl> <dbl>
## 1    51     4 0.25
## 2   189     4 0.167
```

```r
matched_inds_weights |> filter(ind_c==4) |> pull(w) |> sum()
```

```
## [1] 0.4166667
```

```r
# these weights sum to 0.4167; so ind_c=4 will receive total weight 0.4167
#  (accounting for all the treated obs for which it is in the matching set)

# merge in the total weight for each ind_c
d_matched = d |> dplyr::select(-weight) |>
  mutate(id = row_number()) |>
  left_join(
    matched_inds_weights |> group_by(id=ind_c) |> summarise(weight = sum(w)),
    by='id'
  )

# set weight=1 for all treated obs, and weight=0 for all control obs with NA weight
d_matched = d_matched |> mutate(
  weight = case_when(
    treat==1 ~ 1,
    treat==0 & is.na(weight) ~ 0,
    T ~ weight
  )
)

# exactly the ATT from M$est, hooray!
lm(y~treat, weights = weight, data=d_matched) |> coef()
```

```
## (Intercept)       treat
##    4483.894    1865.250
```

```r
# SE is too small, because it doesn't account for variability in the matching
lm(y~treat, weights = weight, data=d_matched) |> summary() |> coef() %>% .[2,2]
```

```
## [1] 675.8459
```

```r
# bootstrapping the whole process:
match_boot_fun = function(){
# create bootstrap sample
  d_boot = d |> slice_sample(n=nrow(d),replace=T)
  X_boot = d_boot |> dplyr::select(-y,-treat) |> as.matrix()
  # run the matching on the bootstrap sample
  M_boot = Match(Y=d_boot$y, Tr = d_boot$treat, X=X_boot
                ,Weight=2 # for Mahalanobis distance
                ,M=3 # number of matches
                ,BiasAdjust=F
                ,exact=F
                ,replace=T
  )
  # store the weights
  mboot_inds_weights = tibble(
    ind_t = M_boot$index.treated,
    ind_c = M_boot$index.control,
    w = M_boot$weights
  )

  # merge in the total weight for each ind_c
  d_boot_matched = d_boot |> dplyr::select(-weight) |>
    mutate(id = row_number()) |>
    left_join(
      mboot_inds_weights |> group_by(id=ind_c) |> summarise(weight = sum(w)),
      by = 'id'
    )

  # set weight=1 for all treated obs, and weight=0 for all control obs with NA weight
  d_boot_matched = d_boot_matched |> mutate(
    weight = case_when(
      treat==1 ~ 1,
      treat==0 & is.na(weight) ~ 0,
      T ~ weight
    )
  )

  # store and return the ATT estimate from the weighted regression
  att_boot = lm(y~treat, weights = weight, data=d_boot_matched) |> coef() %>% .['treat']
  att_boot |> unname()
}

# run 1000 bootstrap replications
set.seed(123)
match_boot_out = replicate(n=1000, match_boot_fun())

# point estimate and 95% CI
quantile(match_boot_out, c(.025, .5, .975))
```

```
##      2.5%       50%     97.5%
##  385.9967 1943.8241 3487.9365
```

```r
sd(match_boot_out) # compare to to the AI SE of 797.63
```

```
## [1] 792.3987
```

5. Switching over to propensity score matching, use the same matching variables as in Question 3 to estimate the propensity to be treated (with a logit regression), and then add the treatment propensity to the data set as a new variable called `propensity`. Trim the propensity score, setting to missing any values from 0 to .05 or from .95 to 1 (this is a different method than done in the chapter).

Be careful to get the predicted *probability of treatment* and not the predicted *index function*. You can check this by making sure the values are all between 0 and 1.

(also, note, your estimation shouldn't produce any propensities that end up actually getting trimmed, but write the code to do so anyway, just in case)

Language-specific notes:

- You can set values to missing based on their value using `variable = ifelse(condition, NA_real_, variable)` or `variable = case_when(condition ~ NA_real_, TRUE ~ variable)` inside a `mutate()` in R, `replace variable = . if condition` in Stata, or using `.loc[condition, 'variable'] = pd.NA` in Python.

	(1)	(2)
(Intercept)	4625.657	4625.657
	(398.614)	(398.614)
treat	1565.426	1565.426
	(636.061)	(636.061)
Num.Obs.	445	445

- It's not necessary, but there are ways of avoiding having to type out every single variable name in each langauge in your regression:
- In R, if your data set contains only your outcome variable and the predictors, you can use the formula `y~.` and that will regress the variable `y` on all the other variables in the data
- In Stata, you can use `a-b` to refer to the full set of variables between `a` and `b`. If your variables are `order`ed properly, this can be the list of predictors
- In Python, you can use **statsmodels.api** instead of **statsmodels.formula.api** and use a matrix of your predictors (or just use the **CausalModel** approach with a matrix, which you're already using)

6. Create a new variable in the data called `ipw` with the inverse probability weight, and then estimate the treatment effect using those weights in a linear regression (keeping in mind the standard errors won't be quite right).

Note that the same tools you used to trim `propensity` conditional on its value can also be used to calculate `ipw` in one way for treated observations and in another way for untreated observations.

```r
# single run of the IPW estimation
#  (point estimate should be fine, SE will be wrong):
X = d |> dplyr::select(-y,-treat) |> as.matrix()
# fit the model predicting treatment assignment
pscore_logit_mod = glm(d$treat ~ X , family=binomial(link='logit'))
# store the estimated propensity scores
pscore_pred = predict(pscore_logit_mod,type='response')
d$pscore = pscore_pred
# trim the pscores, drop any < 0.05 or > 0.95
d = d |> mutate(
  pscore_trimmed = case_when(
    pscore < 0.05 | pscore > .95 ~ NA_real_,
    T ~ pscore
  )
)
# compare IPW with trimmed vs untrimmed pscores
ipw_mod_untrimmed = lm(y~treat, weights=1/pscore, data=d)
ipw_mod_trimmed = lm(y~treat, weights=1/pscore_trimmed, data=d)
modelsummary(models = list(ipw_mod_untrimmed, ipw_mod_trimmed),
             gof_map = c('nobs'))
```

```r
# in this case, there are no pscores outside of (0.05, 0.95), so trimming doesn't matter
# might matter in the bootstrapping
```

```r
# bootstrap function
ipw_boot_fun = function(){
  d_boot = d |> slice_sample(n=nrow(d),replace=T)
  X_boot = d_boot |> dplyr::select(-y,-treat) |> as.matrix()

  # fit the model predicting treatment assignment
  pscore_logit_mod_boot = glm(d_boot$treat ~ X_boot, family=binomial(link='logit'))
  # store the predicted pscores
  pscore_pred_boot = predict(pscore_logit_mod_boot,type='response')
  d_boot$pscore = pscore_pred_boot
  # trim the pscores, drop any < 0.05 or > 0.95
  d_boot = d_boot |> mutate(
    pscore_trimmed = case_when(
      pscore < 0.05 | pscore > .95 ~ NA_real_,
      T ~ pscore
    )
  )
  num_trummed = sum(is.na(d_boot$pscore_trimmed))

  # run the regression with 1/pscore_trimmed as weights
  ipw_mod_boot_untrimmed = lm(y~treat, weights=1/pscore, data=d_boot)
  ipw_mod_boot_trimmed = lm(y~treat, weights=1/pscore_trimmed, data=d_boot)

  # return both bhats (trimmed and untrimmed)
  att_boot_trimmed = ipw_mod_boot_trimmed |> coef() %>% .['treat']
  att_boot_untrimmed = ipw_mod_boot_untrimmed |> coef() %>% .['treat']
  c(att_boot_trimmed,att_boot_untrimmed,num_trummed)
}

# run 1000 boostrap replications
set.seed(123)
ipw_boot_out = replicate(1000, ipw_boot_fun()) |> t()

# point estimate and 95% CI for trimmed IPW
quantile(ipw_boot_out[,1], c(.025,.5,.975))
```

```
##      2.5%       50%     97.5%
##  170.3711 1451.5364 2966.8340
```

```r
sd(ipw_boot_out[,1]) # bootstrap SE
```

```
## [1] 698.1576
```

```r
# point estimate and 95% CI for untrimmed IPW
quantile(ipw_boot_out[,2], c(.025,.5,.975))
```

```
##      2.5%       50%     97.5%
## -1003.602  1322.892  2925.064
```

```r
sd(ipw_boot_out[,2]) # bootstrap SE
```

```
## [1] 946.119
```

Language-specific notes:

- In Python, the trimming process may have converted your `ipw` variable into an "object" instead of a numeric. You may need to use `pd.to_numeric()` to fix the variable before using it as a weight. May as well fix `propensity` while you're at it.

7. Make a common support graph, overlaying the density of the `propensity` variable for treated observations on top of the density of the `propensity` variable for untreated observations. You may want to refer to this guide if you are not familiar with your language's graphing capabilities.

Write a line commenting on how the common support looks.

8. Use the prepackaged command for inverse probability weighting used in the chapter for your language to estimate the treatment effect. Don't apply a trim (as previously established, for this particular problem it doesn't do much).