Economics 691-06: Assignment 2

Nihar Shah

Solutions are marked in blue.

Problem 1: Heterogeneous Treatment Effects

The dataset for this homework comes from an actual survey in the US, where participants were randomly asked one of two questions with similar wording.[1] Assume the randomization scheme was as follows: a coin was flipped prior to each participant being surveyed to determine treatment. The wording is as follows:

> We are faced with many problems in this country, none of which can be solved easily or inexpensively. I'm going to name some of these problems, and for each one I'd like you to tell me whether you think we're spending too much money or too little money on it. Are we spending too much, too little, or about the right amount on (assistance to the poor/welfare)?

The treatment is choosing the wording – "assistance to the poor" (1) versus "welfare" (0). The response is that the person surveyed thinks the government spends too much (1) versus too little (0). This data, along with two covariates (age and party preference, for hypothetical parties A - I), can be found in the file data_assignment2_1.csv.

```
# Load libraries: numpy, pandas, and linear regression
import numpy as np
import pandas as pd
import statsmodels.formula.api as sm

# Load in data
from google.colab import files
import io
uploaded = files.upload()
data = pd.read_csv(io.BytesIO(uploaded['data_assignment2_1.csv']))

# Note that there is some missing data, so we have to correct that
# This is not the only way to correct it, but it is one approach,
# which is simply to drop all rows with a missing value anywhere
data = data.loc[(data.notnull()).all(axis = 1)]
```

[1] Note that the data has been modified to better suit this problem set.

```
# Define a custom function to perform the regression with right SEs
# Takes in a formula, and an index for a specific sample of the data
# This will be useful for all subsequent questions
def regression(formula, index):

  # Get data subset based on index
  sample = data.iloc[index]
  n = len(index)

  # Set some fixed seed
  np.random.seed(0)

  # Compute point estimate
  estimate = sm.ols(formula=formula, data = sample).fit()

  # Bootstrap to get accurate standard errors, since sample sizes aren't
  # fixed (as linear regression assumes); but are random

  # Since the data is unevenly distributed but should be equally split on
  # average, we must create a probability vector to sample from
  weight_control = sum(sample['treatment'] == 1)/len(index)
  weight_treatment = 1 - weight_control
  probability = ((sample['treatment'] == 1) *  weight_treatment +
                 (sample['treatment'] == 0) * weight_control)
  probability = probability/sum(probability)

  # Iterate through 2000 iterations
  samples = []
  for j in range(2000):

    # Sample from data
    b_index = np.random.choice(range(n), n, replace = True, p = probability)

    # Run regression and save results
    result = sm.ols(formula = formula, data = sample.iloc[b_index]).fit()
    samples.append(result.params)

  # Turn samples into dataframe and estimate standard deviation
  samples = pd.DataFrame(samples, columns = result.params.index)
  se = samples.std()

  # Return output
  print("The point estimates are: ")
  print(estimate.params)

  print("The t-statistics are: ")
  print(estimate.params/se)
```

1. **(10 points)** We suspect there is treatment heterogeneity in age, with older participants (those 50 or older) differing from younger participants (49 and below). Prove or disprove this hypothesis; and explain in words what this means about support for assistance to the poor / welfare for the elderly versus young. You may ignore the political party covariate for this and the next problem.

```
# Turn the age variable into binary
data['age_binary'] = (data['age'] >= 50) * 1

# Run regression, including interaction terms
formula = "response ~ treatment + age_binary + age_binary * treatment"
regression(formula, range(len(data['response'])))

The point estimates are:
Intercept               0.437261
treatment              -0.351515
age_binary             -0.020797
age_binary:treatment    0.035132
dtype: float64

The t-statistics are:
Intercept               96.532201
treatment              -67.037524
age_binary              -2.772582
age_binary:treatment     4.010960
dtype: float64
```

As the t-statistics show, all coefficients are significant. The interpretation is that the effect of treatment is negative for the young: wording the question as "assistance to the poor" makes young participants more likely to believe the government spends too little. However, this effect is statistically different and in fact smaller for the elderly. The elderly still receive a negative treatment effect, but a smaller treatment effect than for the young.

2. **(5 points)** Repeat this same process, but using age as a continuous rather than discrete variable. Do your conclusions change?

```
formula = "response ~ treatment + age + age * treatment"
regression(formula, range(len(data['response'])))

The point estimates are:
Intercept        0.441273
treatment       -0.377397
age             -0.000261
age:treatment    0.000855
dtype: float64

The t-statistics are:
Intercept        43.584068
treatment       -32.112610
```

```
age              -1.245894
age:treatment     3.490110
dtype: float64
```

No, the conclusions do not change. Interestingly, age becomes insignificant – but do not get confused! That does not reflect changes in the *treatment effect*. The treatment effect remains negative (and statistically so); and the interaction term remains positive (and statistically so). The treatment effect is strongest for the young, and most muted for the elderly.

3. **(10 points)** We similarly suspect there is treatment heterogeneity in party preference. Prove or disprove this hypothesis; but be mindful of the higher dimensionality of the problem and implement a solution accordingly. You may ignore the age covariate for this problem.

   Below, we implement the causal forest methodology (with linear regression, rather than random forests). We split the sample into two, and use the first half to identify plausible heterogeneous treatment effects; and the second half to estimate the size of those effects.

```
# Split the sample into two
np.random.seed(0)
n = len(data['treatment'])
index1 = np.random.choice(n, round(n/2), replace = False)
index2 = list(set(range(n)) - set(index1))

# Run two regressions: to identify and estimate effects
formula = 'response ~ treatment + party + treatment * party'
regression(formula, index1)

# Note that coefficient sizes are also reported, but I suppress them
The t-statistics are:
Intercept               24.710501
party[T.b]               3.953281
party[T.c]               2.418468
party[T.d]               5.489515
party[T.e]              10.734353
party[T.f]              11.172727
party[T.g]              15.452105
party[T.h]               4.077905
party[T.i]              -0.311213
treatment              -20.371435
treatment:party[T.b]    -2.317925
treatment:party[T.c]    -1.384361
treatment:party[T.d]    -3.213990
treatment:party[T.e]    -6.368674
treatment:party[T.f]    -5.073996
treatment:party[T.g]    -6.571278
treatment:party[T.h]    -0.415128
treatment:party[T.i]     0.040992
```

```
regression(formula, index2)

# Note that t-statistics are also reported, but I suppress them
The point estimates are:
Intercept               0.295525
party[T.b]              0.068967
party[T.c]              0.061852
party[T.d]              0.084070
party[T.e]              0.218346
party[T.f]              0.240264
party[T.g]              0.299160
party[T.h]              0.253656
party[T.i]              0.021548
treatment              -0.247231
treatment:party[T.b]   -0.058375
treatment:party[T.c]   -0.042239
treatment:party[T.d]   -0.049516
treatment:party[T.e]   -0.125449
treatment:party[T.f]   -0.168608
treatment:party[T.g]   -0.164340
treatment:party[T.h]   -0.127707
treatment:party[T.i]    0.055158
dtype: float64
```

The first regression shows that the forms of heterogeneity to focus in on are: the interactions between treatment and affiliation with political parties B, D, E, F, and G. The second regression estimates the sizes, and so we can identify the sizes of those effects as such. Note that they are all negative. As it is, the baseline effect of treatment is negative and statistically so; but affiliation with parties B, D, E, F, and G makes the treatment effect statistically even more negative. (Note further that B often drops out in other sample splits, so your solution may not identify that as a valid form of heterogeneity.)

Problem 2: Non-Compliance

Suppose you are a data scientist at DoorDash, and you implement a new feature – linking of Venmo accounts to pay for meals – to increase revenue. You run a large experiment to test its impact in a given city, with an equal-sized treatment and control group. You want to understand the effect of linking the account on revenue. Of course, you cannot force users to link their accounts, and so that form of non-compliance will have to be handled.

The data following the experiment is in the file data_assignment2_2.csv. The file has 200,000 rows. Each row corresponds to a separate user, for which there are seven columns. Six of the columns are collected at the time of the experiment: revenue brought in by the user, an indicator for whether the user is in the treatment group (where 1 indicates treatment), an indicator for whether the user actually complied with the treatment (i.e. linked their Venmo account, where 1 indicates compliance), the age of the user, the gender of the user (male, female, or other), and the geographic zone (numbered one through eight) of the user in the city. The seventh column ("signal") should be put aside for now, and will be incorporated into the question later.

Note that the larger size of the dataset, when coupled with many thousands of Bootstrap iterations, may occasionally cause out-of-memory errors. Note that 2,000 iterations is often considered enough; and you can go even lower if facing severe memory limitations.

```
# Load libraries: numpy, pandas, and linear regression
import numpy as np
import pandas as pd
import statsmodels.formula.api as sm

# Load in data
from google.colab import files
import io
uploaded = files.upload()
data = pd.read_csv(io.BytesIO(uploaded['data_assignment2_2.csv']))
```

1. **(5 points)** Compute the estimate as given by the as-treated and per-protocol estimator. Explain in one sentence the assumption that both embed, and show using the data that this assumption is likely invalid.

```
effect_at = (np.average(data.loc[data['compliance'] == 1,'revenue']) -
          np.average(data.loc[data['compliance'] == 0,'revenue']))
print("The as-treated estimator is " + str(round(effect_at, 2)))

effect_pp = (np.average(data.loc[(data['compliance'] == 1) &
                                 (data['treatment'] == 1),'revenue']) -
          np.average(data.loc[(data['compliance'] == 0) &
                              (data['treatment'] == 0),'revenue']))
print("The per-protocol estimator is " + str(round(effect_pp, 2)))
```

```
The as-treated estimator is 5.52
The per-protocol estimator is 5.48
```

The assumption for these estimators to be valid is that *compliance with treatment* (rather than treatment) is random. This is almost certainly violated, and we can show that it is not random by showing covariate imbalances. As the next few lines show, the group opting into treatment is nine years younger than their counterparts!

```
age_gap = (np.average(data.loc[data['compliance'] == 1,'age']) -
          np.average(data.loc[data['compliance'] == 0,'age']))
print("The age gap for the as-treated estimator is " + str(round(age_gap, 2)))

age_gap = (np.average(data.loc[(data['compliance'] == 1) &
                               (data['treatment'] == 1),'age']) -
          np.average(data.loc[(data['compliance'] == 0) &
                              (data['treatment'] == 0),'age']))
print("The age gap for the per-protocol estimator is " + str(round(age_gap, 2)))
```

```
The age gap for the as-treated estimator is -9.24
The age gap for the per-protocol estimator is -8.91
```

2. **(10 points)** Compute the estimate as given by the intent-to-treat estimator and Wald estimator. In one sentence each, explain exactly what each number measures. Explain in one further sentence why they differ from each other.

```
effect_it = (np.average(data.loc[data['treatment'] == 1,'revenue']) -
          np.average(data.loc[data['treatment'] == 0,'revenue']))
print("The intent-to-treat estimator is " + str(round(effect_it, 2)))

effect_wd = effect_it/np.average(data.loc[data['treatment'] == 1,'compliance'])
print("The Wald estimator is " + str(round(effect_wd, 2)))
```

```
The intent-to-treat estimator is 0.28
The Wald estimator is 4.31
```

The intent-to-treat estimator measures the effect of being placed in the treatment group. The Wald estimator measures the effect of actually taking treatment. While they are closely linked mathematically, the former measures the effect of being assigned (independent of compliance) treatment while the latter measures the effect of being assigned and being compliant with treatment.

3. **(5 points)** Compute the standard deviation of the Wald estimate using the bootstrap. As always, be very clear about precisely how you performed the bootstrap, and why you chose to do it that way.

```
# First identify the treated and control units
samples = []
i = np.where(data['treatment'] == 1)[0]
j = np.where(data['treatment'] == 0)[0]

# Set some fixed seed
np.random.seed(0)

# Run 2000 iterations of the bootstrap
for counter in range(2000):

  # Implement re-sampling; notice we respect 100k treated and 100k control
  # units at all times, to mimic the original experiment
  i_sample = np.random.choice(i, len(i), replace = True)
  j_sample = np.random.choice(j, len(j), replace = True)

  # Compute difference in means in resampled data
  effect = (np.average(data.iloc[i_sample]['revenue']) -
            np.average(data.iloc[j_sample]['revenue']))
  effect = effect/(np.average(data.iloc[i_sample]['compliance']))
  samples.append(effect)

print('The bootstrapped SE is ' + str(round(np.std(samples), 2)))
```

```
The bootstrapped SE is 0.39
```

```
# Note that we will save the samples for a later question
wald_samples = samples
```

4. Using the variables given (excluding "signal"), construct the estimate as given by the weighted Wald estimator. There are three key steps:

   Before diving into the question, we will construct a useful function that implements all steps: creates models to predict compliance, estimates the weighted Wald estimator, and generates bootstrapped samples.

```
# Remember that zone -- while encoded as a number -- should be encoded as
# a categorical variable!
data['zone'] = data['zone'].astype(str)

# Define a custom-built function to compute the weighted Wald estimator
# This will make future analysis far easier
# The user enters in prediction formula for the linear model; and a flag
# indicating whether they want the parameter estimate (0), the
# bootstrapped standard error (1), or the raw bootstrap samples (2); and
# a final rsquared_flag if they want model diagnostics

def weighted_wald(formula, flag = 0, rsquared_flag = False):

  # Set some fixed seed
  np.random.seed(0)

  # First step: train a model to predict compliance

  # This requires splitting the treatment (and control) set into two
  i = np.where(data['treatment'] == 1)[0]
  j = np.where(data['treatment'] == 0)[0]
  i_half1 = np.random.choice(i, round(len(i)/2), replace = False)
  i_half2 = list(set(i) - set(i_half1))
  j_half1 = np.random.choice(j, round(len(j)/2), replace = False)
  j_half2 = list(set(j) - set(j_half1))

  # Now generate two models, each trained on half of the treatment set
  model1 = sm.ols(formula=formula, data = data.iloc[i_half1]).fit()
  model2 = sm.ols(formula=formula, data = data.iloc[i_half2]).fit()
  if rsquared_flag:
    print("The average R^2 of the model in question is: " +
          str(round(0.5 * (model1.rsquared + model2.rsquared), 2)))

  # Finally, use each model to predict on the other half of the treatment
  # set and half of the control set
  data['p'] = -1
  col_index = data.columns.get_loc('p')
  data.iloc[i_half2, col_index] = model1.predict(data.iloc[i_half2])
```

```
data.iloc[j_half2, col_index] = model1.predict(data.iloc[j_half2])
data.iloc[i_half1, col_index] = model2.predict(data.iloc[i_half1])
data.iloc[j_half1, col_index] = model2.predict(data.iloc[j_half1])

# Second step: for a given set of indices, compute weighted Wald estimator
# Nest this in a function-within-function to make it easier
def estimate(i_sample, j_sample):
  effect = (np.average(data.iloc[i_sample]['revenue'] *
                       data.iloc[i_sample]['p']) -
            np.average(data.iloc[j_sample]['revenue'] *
                       data.iloc[j_sample]['p']))
  effect = effect/(np.average(data.iloc[i_sample]['compliance']  *
                              data.iloc[i_sample]['p']))
  return(effect)

# Third step: compute the point estimate
if flag == 0:
  return(estimate(i, j))

# Fourth step: Bootstrap!
samples = []

# Run 2000 iterations of the bootstrap
for counter in range(2000):

  # Implement re-sampling; notice we respect 100k treated and 100k control
  # units at all times, to mimic the original experiment
  i_sample = np.random.choice(i, len(i), replace = True)
  j_sample = np.random.choice(j, len(j), replace = True)
  samples.append(estimate(i_sample, j_sample))

# Either return the standard error or the actual bootstrap estimates
if flag == 1:
  return(np.std(samples))
return(samples)
```

(a) **(10 points)** First, predict compliance across the entire sample using all pre-experimental variables. To predict compliance, you'll first have to build a model, using your treatment group to train. Explain why we use only the treatment group as the training set, and further explain any other choices you make while constructing the model. Note that full credit will be awarded to linear regression models, but you are welcome to pick any more complicated class of models (e.g. logistic regression, random forest, etc) if curious.

This is implemented in the function weighted_wald (above). The treatment group is the only group that was given the option to take or reject treatment, and so they are the only observations we can use for training. Most other choices are standard, but I will mention three. First, notice that I have cast "zone" (encoded as a numeric variable) into a categorical one, as it reflects a neighborhood. Second, I split the training group into

two, and use a model trained on each half to estimate the other half. This ensures that no observation is ever incorporated both in training the model and being predicted on by the model, as a defense against overfitting. Third, I split the control group into two, and similarly use each of the two models to predict half. This ensures that the weights in the control group are generated the same way as the weights in the treatment group, and thus have the same distribution.

(b) **(5 points)** Second, using these predictions as the weights, compute estimate as given by the weighted Wald estimator. Remember that the weighted Wald estimator is just the Wald estimator, except each term in the numerator and denominator is multiplied by weights.

```
formula = 'compliance ~ age + gender + zone'
print('The estimate is: ' + str(round(weighted_wald(formula, 0), 2)))

The estimate is: 4.0
```

(c) **(10 points)** Third, the standard deviation of the weighted Wald estimator by bootstrapping. There are two approaches: you can construct a new model for each bootstrap iteration, or you can hold your original model fixed. Discuss the tradeoffs between these two approaches. You may implement either one for full credit.

```
formula = 'compliance ~ age + gender + zone'
print('The bootstrap SD is: ' + str(round(weighted_wald(formula, 1), 2)))

The bootstrap SD is: 0.53
```

The approach chosen here is to hold the model fixed over iterations. There are two considerations. On one hand, rebuilding the model across Bootstrap iterations better mimics the underlying experiment. On the other hand, rebuilding the model is extremely computationally costly. For fickle models that require lots of tuning (e.g. neural nets), the former effect is important; but for relatively stable models (e.g. linear regression), the former effect is negligible.

5. Now let's introduce the last variable, signal. A data scientist at DoorDash tells you that she has found a high-quality predictor of compliance, called signal.

(a) **(10 points)** First, using this new variable along with all original variables, build a new model to predict compliance, generate a new weighted Wald estimate, and generate new standard errors by bootstrap.

```
formula = 'compliance ~ age + gender + zone + signal'
print('The estimate is: ' + str(round(weighted_wald(formula, 0), 2)))
print('The bootstrap SD is: ' + str(round(weighted_wald(formula, 1), 2)))

The estimate is: 3.97
The bootstrap SD is: 0.37
```

(b) **(5 points)** Second, compare the model performance (e.g. R^2) and the standard errors for this versus your original weighted Wald estimator. Which estimate do you trust more, and why?

```
formula = 'compliance ~ age + gender + zone'
print('The simple model bootstrap SD is: ' +
      str(round(weighted_wald(formula, 1, True), 2)))
formula = 'compliance ~ age + gender + zone + signal'
print('The complex model bootstrap SD is: ' +
      str(round(weighted_wald(formula, 1, True), 2)))
```

```
The average R^2 of the model in question is: 0.21
The simple model bootstrap SD is: 0.53
The average R^2 of the model in question is: 0.75
The complex model bootstrap SD is: 0.37
```

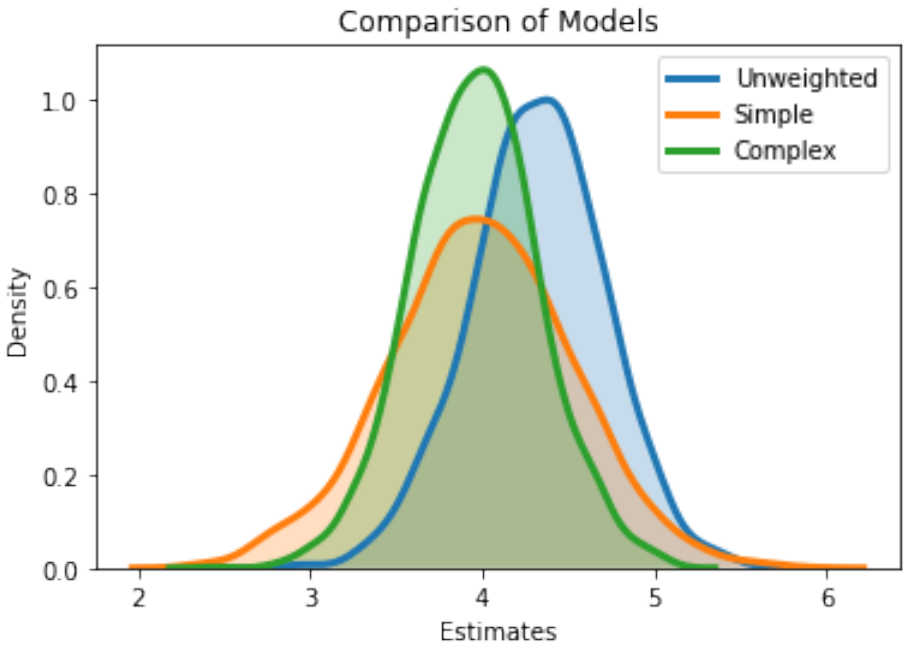
The R^2 of the complex model (that incorporates "signal") is much higher, and this is reflected in a lower standard error and thus a more trustworthy estimate.

6. **(5 points)** We now have collected three estimates with three standard errors: Wald, weighted Wald for a simple model (i.e. no "signal"), and weighted Wald for an enhanced model (i.e. with "signal"). Depict all three point estimates and variances graphically.

```
# We already collected the vanilla Wald samples as wald_samples
# Now collect the raw bootstrap iterations for the weighted samples
wald_simple_samples = weighted_wald('compliance ~ age + gender + zone', 2)
wald_complex_samples = weighted_wald(
    'compliance ~ age + gender + zone + signal', 2)

import matplotlib.pyplot as plt
import seaborn as sns
sns.distplot(wald_samples, hist = False, kde = True,
                 kde_kws = {'shade': True, 'linewidth': 3},
                 label = 'Wald')
sns.distplot(wald_simple_samples, hist = False, kde = True,
                 kde_kws = {'shade': True, 'linewidth': 3},
                 label = 'Weighted Wald (Simple)')
sns.distplot(wald_complex_samples, hist = False, kde = True,
                 kde_kws = {'shade': True, 'linewidth': 3},
                 label = 'Weighted Wald (Complex)')
plt.title('Comparison of Models')
plt.xlabel('Estimates')
plt.ylabel('Density')
```

In general, remember that high-quality predictions are important to offset the natural increase in variance that comes from an unequally-weighted estimator. This effect is clearly on display here, where a low-quality model leads to higher standard errors than the baseline Wald estimator. The high-quality model, however, performs similarly and even slightly better than the baseline Wald estimator.


Comparison of Models
Unweighted
Simple
Complex
Density
1.0
0.8
0.6
0.4
0.2
0.0
2
3
4
5
6
Estimates