Advanced Adjustment - An Introduction to Doubly Robust Methods

UCLA Stats 256 Spring 2022

Jun 10, 2022

CONTENTS

An Introduction to Doubly Robust Methods

By Group 3: Sunay Bhat, Ayush Chatterjee, Laxman Dahal, Nathan Hoffmann, Arya Nanda

Abstract

In the following tutorial, we cover doubly robust estimation and key methods. We begin by detailing a brief history on the originals of doubly robust estimation and precursor methods. We then detail a precise formulation and demonstrate the mechanism by which doubly robust estimation is achieved. We further cover a few methods of interest including Augmented Inverse propensity Weighting (AIPW), Targeted maximum Likelihood Estimation (TMLE), and Double Machine Learning (DML). We use coding examples to illustrate the implementation and impacts of each of these methods in order to provide a clear and working understanding on doubly robust estimation methods.

CHAPTER

ONE

WHAT DOES "DOUBLY ROBUST" MEAN?

Doubly robust methods estimate two models:

- an *outcome model* $\mu_d(X_i) = E(Y_i \mid D_i = d, X_i)$
- and a *exposure model* (or treament model or propensity score): $\pi(X_i) = E(D_i \mid X_i)$

where $\mu_d(\cdot)$ is the model of control or treatment $D_i = d = \{0, 1\}$, X_i is a vector of covariates for unit $i = 1, ..., N$ for treatment (1) and control (0), Y_i is the outcome, and $\pi(\cdot)$ is the exposure model. The covariates included in X_i can be different for the two models.

An estimator is called "doubly robust" if it achieves consistent estimation of the ATE (or whatever estimand we're interested in) as long as *at least one* of these two models is consistently estimated. This means that the outcome model can be completely misspecified, but as long as the exposure model is correct, our estimation of the ATE will be consistent. This also means that the exposure model can be completely wrong, as along as the outcome model is correct.

1.1 Origins of Doubly Robust Methods

According to Bang and Robins (2005), doubly robust methods have their origins in missing data models. Robins, Rotnitzky, and Zhao (1994) and Rotnitzky, Robins, and Scharfstein (1998) developed augmented orthogonal inverse probability-weighted (AIPW) estimators in missing data models, and Scharfstein, Rotnitzky, and Robins (1999) showed that AIPW was doubly robust and extended to causal inference.

But Kang and Schafer (2007) argue that doubly robust methods are older. They cite work by Cassel, Särndal, and Wretman (1976), who proposed "generalized regression estimators" for population means from surveys where sampling weights must be estimated.

Arguably, doubly robust methods go back even further than this. The form of doubly robust methods is similar to residual-on-residual regression, which dates back to Frisch, Waugh, and Lovell (1933) famous FWL theorem: $\beta_D = \frac{\text{Cov}(\tilde{Y}_i, \tilde{D}_i)}{\text{Var}(\tilde{D}_i)} where$\tilde D_i$ is the residual part of D_i after regressing it on X_i, and $\tilde Y_i$ is the residual part of Y_i after regressing it on X_i. This formulation writes the regression coefficient as composed of an outcome model ($\tilde Y_i$) and exposure model ($\tilde D_i$), the two models used in doubly robust estimators.

There are also links between doubly robust methods and matching with regression adjustment. This work goes back to at least Rubin (1973), who suggested that regression adjustment in matched data produces less biased estimates that either matching (exposure adjustment) or regression (outcome adjustment) do by themselves.

1.2 Assumptions

Most doubly robust methods require almost all of the standard assumptions necessary formost methods that depend on selection on observables. Although some doubly robust methods relax one or two of these, the six standard assumptions are:

1. Consistency
2. Positivity/overlap
3. One version of treatment
4. No interference
5. IID observations
6. Conditional ignorability: $\{Y_{i0}, Y_{i1}\} \perp\!\!\!\perp D_i \mid X_i$

Special attention should be paid to Assumption 6: doubly robust methods will not work if we do not measure an important confounder that affects both treatment and exposure. But notably, the doubly robust methods covered in this tutorial make no functional form assumptions. Most use flexible machine learning algorithms to estimate both the outcome and exposure models, with regularization (often through cross-fitting) to avoid overfitting.

If these six assumptions are met, and we use the right estimator, we get double robustness: consistent estimation if either treatment or outcome model correct.

1.3 A simple demonstration

To demonstrate double robustness, this section presents one of the simpler doubly robust estimators: augmented inverse probability weights (AIPW). The following is adapted from Chapter 12 of Matheus Facure Alves's (2021) *Causal Inference for the Brave and True*.

We can write this estimator as follows: $

$$\begin{aligned}\widehat{ATE} = &\frac{1}{N}\sum_{i=1}^{N}\left(\frac{D_i(Y_i - \hat{\mu}_1(X_i))}{\hat{\pi}(X_i)} + \hat{\mu}_1(X_i)\right) \\ &- \frac{1}{N}\sum_{i=1}^{N}\left(\frac{(1-D_i)(Y_i - \hat{\mu}_0(X_i))}{1-\hat{\pi}(X_i)} + \hat{\mu}_0(X_i)\right)\end{aligned}$$

$

For each individual in the sample, this estimator calculates two quantities:

- The treated potential outcome $\hat{Y}_{1i} = \frac{D_i(Y_i - \hat{\mu}_1(X_i))}{\hat{\pi}(X_i)} + \hat{\mu}_1(X_i)$
- The control potential outcome $\hat{Y}_{0i} = \frac{(1-D_i)(Y_i - \hat{\mu}_0(X_i))}{1-\hat{\pi}(X_i)} + \hat{\mu}_0(X_i)$

Let's focus on the treated model: $\hat{Y}_{1i} = \frac{D_i(Y_i - \hat{\mu}_1(X_i))}{\hat{\pi}(X_i)} + \hat{\mu}_1(X_i)$

First, assume that the outcome model $\mu_1(X_i)$ is *correctly* specified and the exposure model $\pi(X_i)$ is *incorreclty* specified. Let's also assume (for now) that we're dealing with a treated unit, i.e. $D_i = 1$. Then $\hat{\mu}_1(X_i) = Y_i and hence \hat{Y}_{1i} = \frac{D_i(0)}{\hat{\pi}(X_i)} + \hat{\mu}_1(X_i) = \hat{\mu}_1(X_i). So the model relies * only * on the outcome model! The incorrectly specified exposure model completely disappears from the equation. If we're dealing with a c \frac{0(Y_i - \hat{\mu}_1(X_i))}{\hat{\pi}(X_i)} + \hat{\mu}_1(X_i) = \hat{\mu}_1(X_i).$

Now, what if the *exposure* model $\pi(X_i)$ is correctly specified and the outcome model $\mu_1(X)$ is incorrect? First, we

rewrite the estimator for the treated outcome:

$$\begin{aligned}\hat{Y}_{1i} &= \frac{D_i(Y_i - \hat{\mu}_1(X_i))}{\hat{\pi}(X_i)} + \hat{\mu}_1(X_i) \\ &= \frac{D_i Y_i}{\hat{\pi}(X_i)} - \frac{D_i \hat{\mu}_1(X_i)}{\hat{\pi}(X_i)} + \frac{\hat{\pi}(X_i)\hat{\mu}_1(X_i)}{\hat{\pi}(X_i)} \\ &= \frac{D_i Y_i}{\hat{\pi}(X_i)} - \left(\frac{D_i - \hat{\pi}(X_i)}{\hat{\pi}(X_i)}\right)\hat{\mu}_1(X_i). \qquad (*)\end{aligned}$$

Since the exposure model is correclty specified, we have $D_i = \hat{\pi}(X_i)$ on average, so $$E[D_i - \hat{\pi}(X_i)] = 0. This means that the second term in equation (*) is 0, so $E[\hat{Y}_{1i}] = E\left[\frac{D_i Y_i}{\hat{\pi}(X_i)}\right]$.$

This shows that when the exposure model is correct, then the estimator depends *only* on the exposure model. We can make similar arguments for the control model $\hat{Y}_{0i}$.

This demonstration shows that this estimator achieves double robustness: the estimator is robust to misspecification of either the exposure or the outcome model (but not both!).

1.3.1 References

Bang, H., & Robins, J. M. (2005). Doubly Robust Estimation in Missing Data and Causal Inference Models. *Biometrics*, 61(4), 962–973. https://doi.org/10.1111/j.1541-0420.2005.00377.x

CASSEL, C. M., SÄRNDAL, C. E., & WRETMAN, J. H. (1976). Some results on generalized difference estimation and generalized regression estimation for finite populations. *Biometrika*, 63(3), 615–620. https://doi.org/10.1093/biomet/63.3.615

Frisch, R., & Waugh, F. V. (1933). Partial Time Regressions as Compared with Individual Trends. *Econometrica*, 1(4), 387–401. https://doi.org/10.2307/1907330

Kang, J. D. Y., & Schafer, J. L. (2007). Demystifying Double Robustness: A Comparison of Alternative Strategies for Estimating a Population Mean from Incomplete Data. *Statistical Science*, 22(4), 523–539. https://doi.org/10.1214/07-STS227

Robins, J. M., Rotnitzky, A., & Zhao, L. P. (1994). Estimation of Regression Coefficients When Some Regressors are not Always Observed. *Journal of the American Statistical Association*, 89(427), 846–866. https://doi.org/10.1080/01621459.1994.10476818

Rotnitzky, A., Robins, J. M., & Scharfstein, D. O. (1998). Semiparametric Regression for Repeated Outcomes with Nonignorable Nonresponse. *Journal of the American Statistical Association*, 93(444), 1321–1339. https://doi.org/10.2307/2670049

Rubin, D. B. (1973). The Use of Matched Sampling and Regression Adjustment to Remove Bias in Observational Studies. *Biometrics*, 29(1), 185–203. https://doi.org/10.2307/2529685

Scharfstein, D. O., Rotnitzky, A., & Robins, J. M. (1999). Adjusting for Nonignorable Drop-Out Using Semiparametric Nonresponse Models. *Journal of the American Statistical Association*, 94(448), 1096–1120. https://doi.org/10.1080/01621459.1999.10473862

CHAPTER

TWO

DOUBLY ROBUST METHODS

2.1 AIPW

2.1.1 Background

Augmented Inverse Propensity Weighting (AIPW) is a modification of standard Inverse Propensity Weighting to achieve double robustness. We first consider basic IPW, which considers a sample weight, or propensity score $\hat{\pi}(X_i)$, in the model.

$$\widehat{ATE}_{IPW} = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{D_i Y_i}{\hat{\pi}(X_i)} - \frac{(1-D_i)Y_i}{1-\hat{\pi}(X_i)}\right]$$

The augmenteed IPW, AIPW, as presetned by Glynn and Quinn, 2009 includes the outcome model in such a way that ensures doubly-robust estimation. This equations below reqrites the AIPW formulation such the basic IPW is seen clearly first along with the ourcome model augmentation.

$$\widehat{ATE}_{AIPW} = \frac{1}{N}\sum_{i=1}^{N}\left(\left[\frac{D_i Y_i}{\hat{\pi}(X_i)} - \frac{(1-D_i)Y_i}{1-\hat{\pi}(X_i)}\right] - \frac{(X_i - \hat{\pi}(X_i))Y_i}{\pi(X_i)(1-\pi(X_i))}\right)[(1-\hat{\pi}(X_i))\hat{\mathbb{E}}(Y_i|D_i=1,X_i) + +\hat{\pi}(X_i)\hat{\mathbb{E}}(Y_i|D_i=0,X_i)]$$

2.1.2 Dataset

We will use a simulated dataset based on The National Study of Learning Mindsets. This was a randomized study conducted in U.S. public high schools, the purpose of which was to evaluate the impact of a nudge-like, optional intervention designed to instill students with a growth mindset. The study includes measured outcomes via an achievement score, a binary treatment of a growth mindset educational intervention, and 11 other potential confounding factors that could be parents of both the treatment and outcome. The first 5 rows of the dataset are shows in the table below.

```
df_mindset = pd.read_csv('learning_mindset.csv')
# print(df_mindset.info())
df_mindset.head()
```

```
   schoolid  intervention  achievement_score  success_expect  ethnicity  \
0        76             1              0.277               6          4
1        76             1             -0.450               4         12
2        76             1              0.770               6          4
3        76             1             -0.122               6          4
4        76             1              1.526               6          4
```

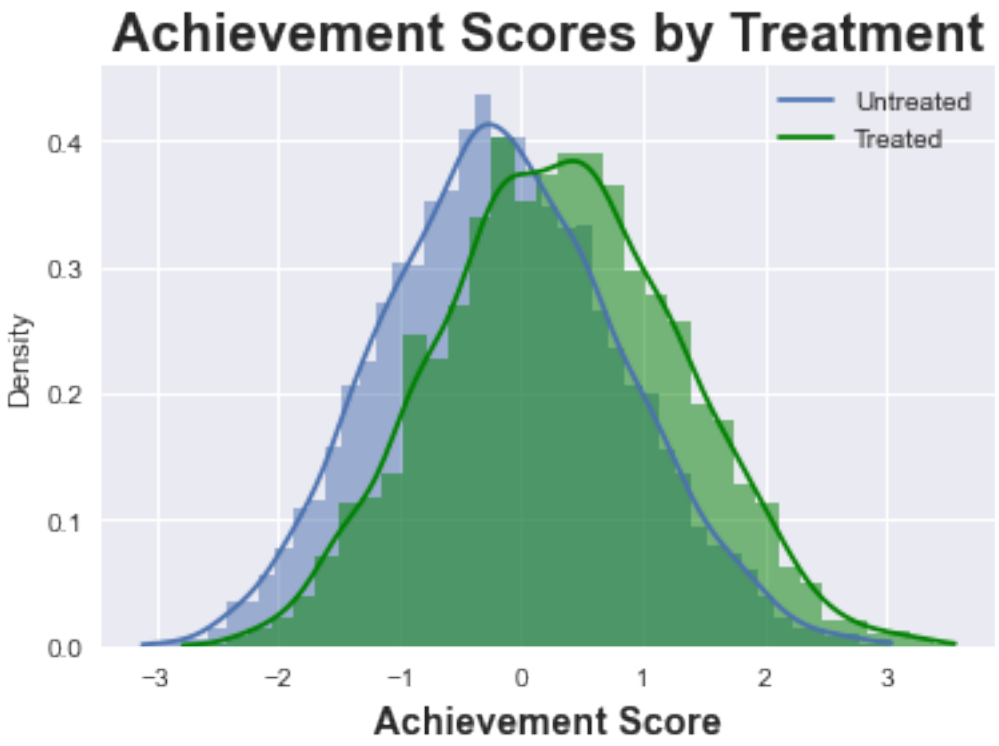
(continues on next page)

(continued from previous page)

```
   gender  frst_in_family  school_urbanicity  school_mindset  \
0       2               1                  4           0.335
1       2               1                  4           0.335
2       2               0                  4           0.335
3       2               0                  4           0.335
4       1               0                  4           0.335

   school_achievement  school_ethnic_minority  school_poverty  school_size
0               0.649                  -1.311           0.224       -0.427
1               0.649                  -1.311           0.224       -0.427
2               0.649                  -1.311           0.224       -0.427
3               0.649                  -1.311           0.224       -0.427
4               0.649                  -1.311           0.224       -0.427
```

2.1.3 Understanding the data and potential bias

We begin by visualizing the achievement scores of treated and untreated cohorts with no control or consideration for the other variables. It is clear form the plot below there is an impact of the treatment as the average of the treated group's achievement scores is clearly higher. But we can intuit a positive bias in this measurement. We should note again the intervention was an option to take a growth mindset course. So although the option was offered in a random fashion, *it is highly likely students who opt-in to the treatment are likely to have the features to provide higher achievement scores regardless*. Thus, we might hypothesize controlling for this bias would decrease the ATE from the *naive ATE* (meaning no adjustment or simple difference of means of the treated and untreated groups).



2.1.4 Method Implementations

The following code block implements the naive ATE, the standard IPW, and finally the AIPW methods as python functions. Note that propensity score, or the exposure model, is constructed as a *Logistic Regression problem*, and the outcome model is generated as a *Linear Regression problem*.

We do this to allow us to readily run many iterations of each method. We will use a bootstrap subsample method, where we will sample 1% of the original data (~100 data points), 100 times. This will allow us to generate a distribution of ATEs with an empirical standard deviation. Thus we can report our results comparing each of the three methods using various exposure and outcome models with 95% confidence intervals as well.

```
#### Define Estimation methods ####

### Linear Regression T on Y
def naive_ATE(df, T, Y):
    return df[Y][df_categ[T] == 1].mean() - df_categ[Y][df_categ[T] == 0].mean()

### IPW
def IPW(df, X, T, Y,true_ps = True):

    if true_ps:
        p_scores = LogisticRegression(C=1e6, max_iter=1000).fit(df[X], df[T]).predict_
↪proba(df[X])[:, 1]
    else:
        p_scores = np.random.uniform(0.1, 0.9, df.shape[0])

    df_ps = df.assign(propensity_score=p_scores)

    weight = ((df_ps["intervention"]-df_ps["propensity_score"]) / (df_ps["propensity_
↪score"]*(1-df_ps["propensity_score"])))

    weight_t = 1/df_ps.query("intervention==1")["propensity_score"]
    weight_nt = 1/(1-df_ps.query("intervention==0")["propensity_score"])

    y1 = sum(df_ps.query("intervention==1")["achievement_score"]*weight) / len(df_ps)
    y0 = sum(df_ps.query("intervention==0")["achievement_score"]*weight_nt) / len(df_
↪ps)

    return np.mean(weight * df_ps["achievement_score"]), p_scores, df_ps

### AIPW
def AIPW(df, X, T, Y,true_ps = True,true_mus = True):
    if true_ps:
        p_scores = LogisticRegression(C=1e6,max_iter=500).fit(df[X], df[T]).predict_
↪proba(df[X])[:, 1]
    else:
        p_scores = np.random.uniform(0.1, 0.9, df.shape[0])

    if true_mus:
        mu0 = LinearRegression().fit(df.query(f"{T}==0")[X], df.query(f"{T}==0")[Y]).
↪predict(df[X])
        mu1 = LinearRegression().fit(df.query(f"{T}==1")[X], df.query(f"{T}==1")[Y]).
↪predict(df[X])
    else:
        mu0 = np.random.uniform(0, 1, df.shape[0])
        mu1 = np.random.uniform(0, 1, df.shape[0])
```

(continues on next page)

(continued from previous page)

```
    return (
        np.mean(df[T]*(df[Y] - mu1)/p_scores + mu1) -
        np.mean((1-df[T])*(df[Y] - mu0)/(1-p_scores) + mu0)
    ), p_scores, mu0, mu1
```

2.1.5 Experiments and Results

The following block shows our bootstrap sampling method results displayed (100 iterations for ~100 samples of 1% of dataset). In this initial experiment, we correctly specify both the exposure and outcome models. The results are displayed in the plot and table below.

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Input In [6], in <cell line: 6>()
      6 for iSample in range(bootstrap_sample):
      7     df_bootstrap = df_categ.sample(frac=1)
----> 8     ate, ps, mu0, mu1 = AIPW(df_bootstrap, X, T, Y)
      9     AIPW_ates.append(ate)
     10     ate, ps,_ = IPW(df_bootstrap, X, T, Y,true_ps=False)

Input In [5], in AIPW(df, X, T, Y, true_ps, true_mus)
     34 if true_mus:
     35     mu0 = LinearRegression().fit(df.query(f"{T}==0")[X], df.query(f"{T}==0
 ↪")[Y]).predict(df[X])
---> 36     mu1 = LinearRegression().fit(df.query(f"{T}==1")[X], df.query(f"{T}==1
 ↪")[Y]).predict(df[X])
     37 else:
     38     mu0 = np.random.uniform(0, 1, df.shape[0])

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/linear_model/_
 ↪base.py:362, in LinearModel.predict(self, X)
    348 def predict(self, X):
    349     """
    350     Predict using the linear model.
    351
   (...)
    360         Returns predicted values.
    361     """
--> 362     return self._decision_function(X)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/linear_model/_
 ↪base.py:345, in LinearModel._decision_function(self, X)
    342 def _decision_function(self, X):
    343     check_is_fitted(self)
--> 345     X = self._validate_data(X, accept_sparse=["csr", "csc", "coo"],␣
 ↪reset=False)
    346     return safe_sparse_dot(X, self.coef_.T, dense_output=True) + self.
 ↪intercept_

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/base.py:566,␣
 ↪in BaseEstimator._validate_data(self, X, y, reset, validate_separately, **check_
 ↪params)
    564     raise ValueError("Validation should be done on X, y or both.")
    565 elif not no_val_X and no_val_y:
```

(continues on next page)

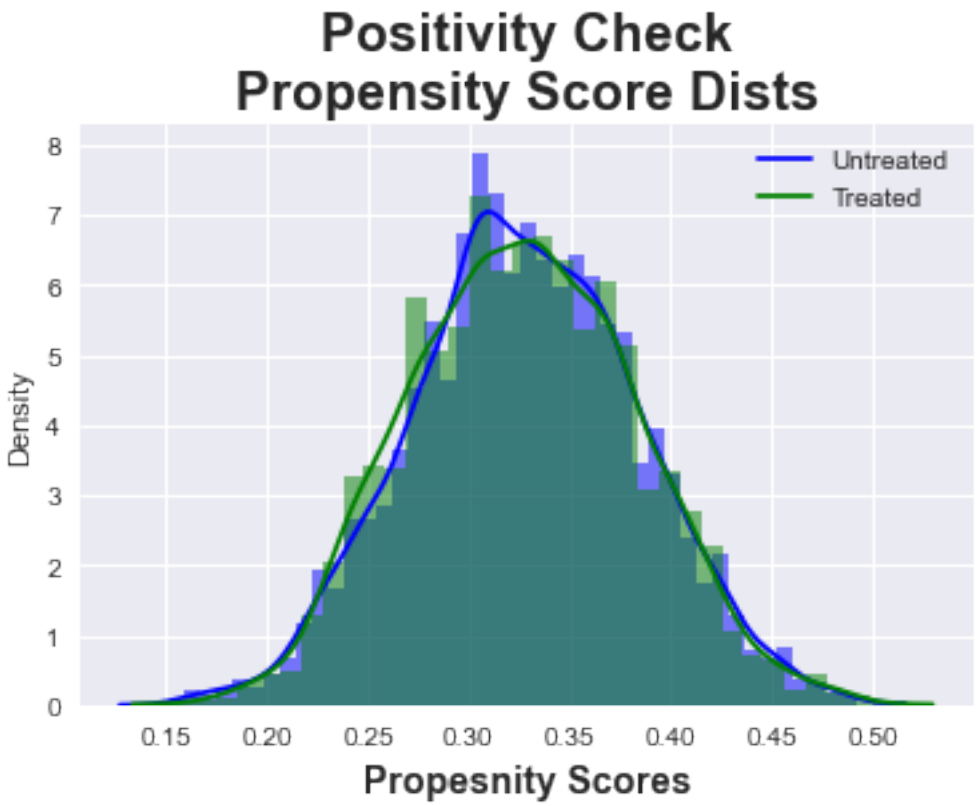
(continued from previous page)

```
--> 566     X = check_array(X, **check_params)
    567     out = X
    568 elif no_val_X and not no_val_y:

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/utils/
↪validation.py:617, in check_array(array, accept_sparse, accept_large_sparse,␣
↪dtype, order, copy, force_all_finite, ensure_2d, allow_nd, ensure_min_samples,␣
↪ensure_min_features, estimator)
    615 dtypes_orig = None
    616 has_pd_integer_array = False
--> 617 if hasattr(array, "dtypes") and hasattr(array.dtypes, "__array__"):
    618     # throw warning if columns are sparse. If all columns are sparse, then
    619     # array.sparse exists and sparsity will be preserved (later).
    620     with suppress(ImportError):
    621         from pandas.api.types import is_sparse

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/pandas/core/generic.
↪py:5747, in NDFrame.dtypes(self)
   5720 """
   5721 Return the dtypes in the DataFrame.
   5722
   (...)
   5744 dtype: object
   5745 """
   5746 data = self._mgr.get_dtypes()
-> 5747 return self._constructor_sliced(data, index=self._info_axis, dtype=np.
↪object_)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/pandas/core/series.
↪py:459, in Series.__init__(self, data, index, dtype, name, copy, fastpath)
    456         elif manager == "array":
    457             data = SingleArrayManager.from_array(data, index)
--> 459 NDFrame.__init__(self, data)
    460 self.name = name
    461 self._set_axis(0, index, fastpath=True)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/pandas/core/generic.
↪py:255, in NDFrame.__init__(self, data, copy, attrs)
    253     attrs = dict(attrs)
    254 object.__setattr__(self, "_attrs", attrs)
--> 255 object.__setattr__(self, "_flags", Flags(self, allows_duplicate_
↪labels=True))

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/pandas/core/flags.
↪py:49, in Flags.__init__(self, obj, allows_duplicate_labels)
     47 def __init__(self, obj, *, allows_duplicate_labels):
     48     self._allows_duplicate_labels = allows_duplicate_labels
---> 49     self._obj = weakref.ref(obj)

KeyboardInterrupt:
```

From the results it is clear both IPW and AIPW account for a positive bias we hypothesized. They estimate the ATE at ~0.39, up from the naive ATE estimate of ~0.47. We also note the IPW and AIPW methods agree with very close estimates and with very similar 95% confidence intervals. This is unsurprising considering the exposure model is correctly specified using logistic regression for both methods.

Now that we have propensity scores, we can also perform a quick positivity check visualizing the distribution of our

propensity scores to ensure we meet the positivity/overlap assumption which the plot below demonstrates.

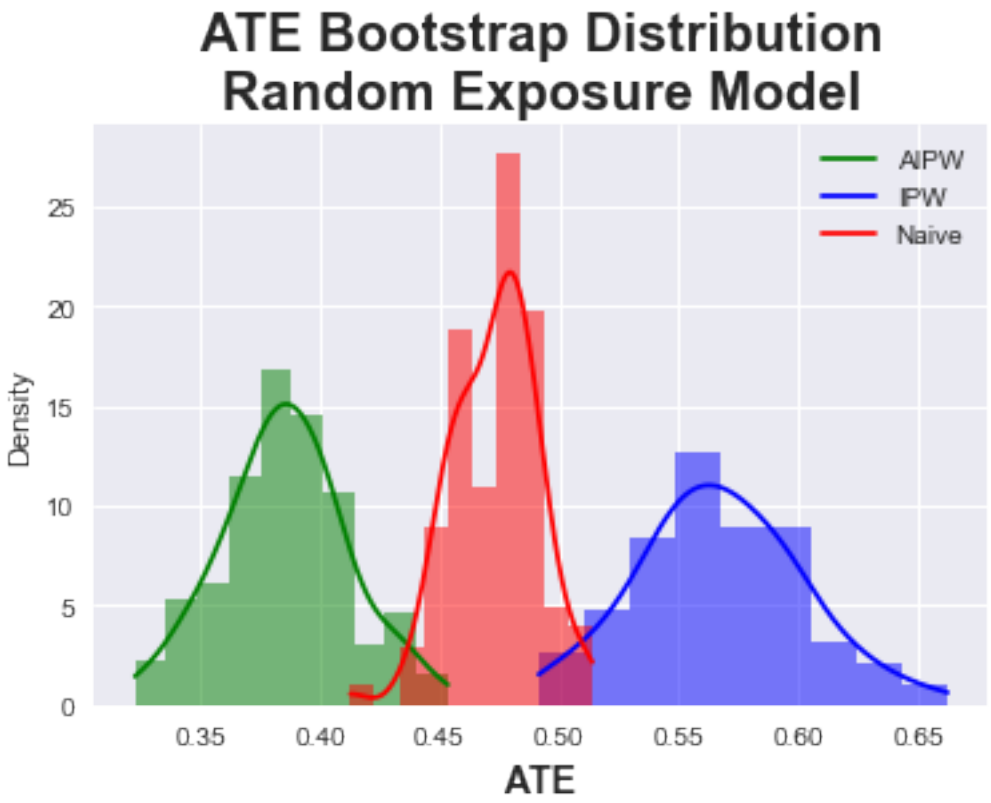


In the second experiment, we specify a bad exposure model. Instead of using logistic regression, we simply sample a uniform random distribution:

$$\hat{\pi}(X_i) \sim U(0.1, 0.9)$$

As we can see from the results below, the AIPE method is effectively stable, estimating a slightly lower ATE of about ~0.38. The standard deviation also increases slightly. On the other hand, the IPW method does far worse than the naive method, which again makes sense as we are feeding it random noise for the propensity scores. This is the first example of a doubly robust method showing how, since the outcome model is correctly specified, the estimation is still robust even to random noise for the exposure model.

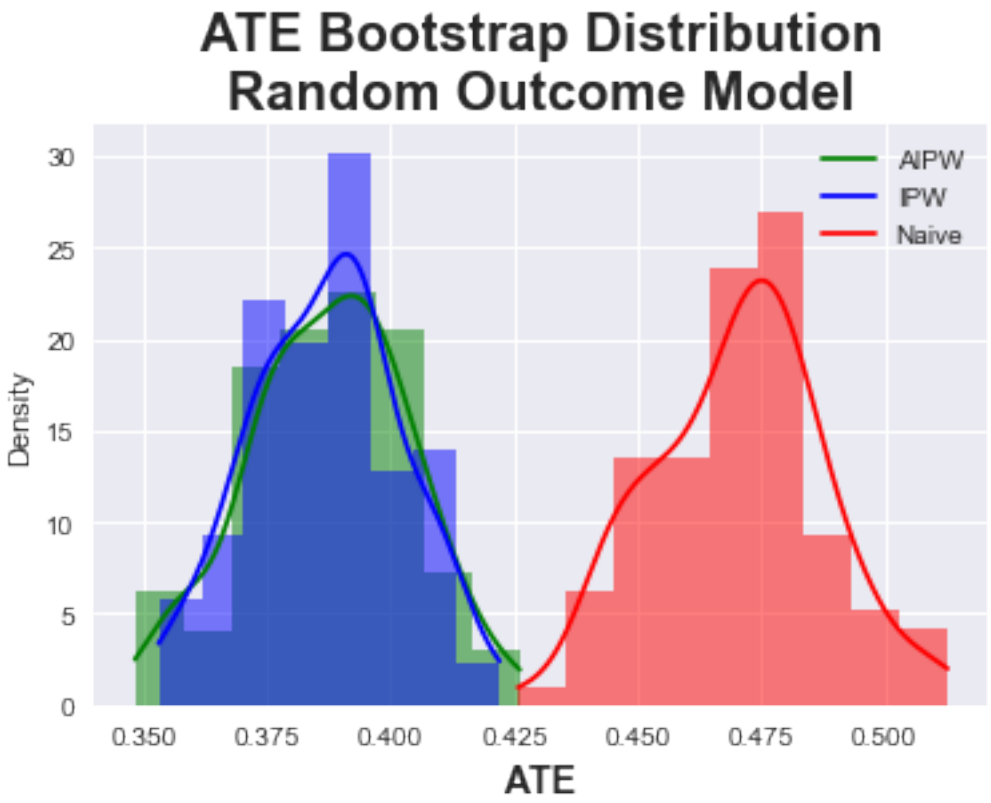
```
       Mean ATE  Std Dev  [.025  .975]
AIPW      0.385    0.026  0.334  0.435
IPW       0.567    0.035  0.498  0.638
Naive     0.473    0.018  0.440  0.507
```



In the third experiment, we now investigate the impact of a bad outcome model. We again sample from a uniform distribution to obtain the incorrect outcome data: $\$\mu_d(X_i) \sim U(0,1)\$$

Here once again see the AIPW and IPW methods both agree and estimate ~0.39. AIPW again shows the doubly robust property against the completely random outcome model, while IPW is unimpacted since the exposure model is correct. Both hence perform similarly to the original experiment.

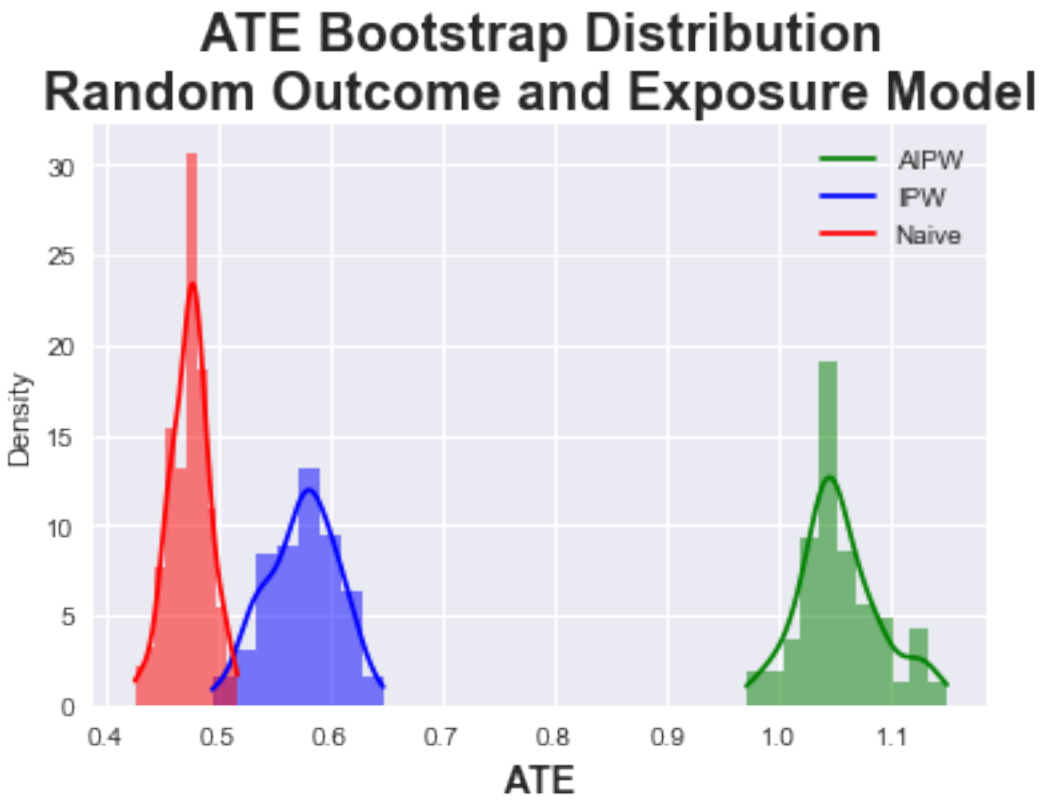
```
        Mean ATE  Std Dev  [.025  .975]
AIPW       0.388    0.016  0.356  0.417
IPW        0.387    0.015  0.357  0.413
Naive      0.470    0.017  0.441  0.506
```



In the final experiment, we show the impact of a bad outcome and exposure model: $\mu_d(X_i) \sim U(0,1), \hat{\pi}(X_i) \sim U(0.1, 0.9)$

In this experiment, we see that AIPW performs very poorly, vastly over-estimating the ATE. In this instance, either naive or IPW would perform better, although the naive without any consideration for random models does best.

```
        Mean ATE  Std Dev  [.025  .975]
AIPW       1.054    0.037  0.986  1.132
IPW        0.575    0.032  0.515  0.630
Naive      0.475    0.017  0.439  0.508
```



2.1.6 Concluding Thoughts for AIPE

We clearly demonstrate AIPW's doubly robust properties using the simulated National Mindset dataset. But it is important to note, our 'incorrect' models were uniform random which would be about as poor as one could imagine. In reality, misspecified models contain more subtle biases or noise, and thus there is a whole host of literature investigating the sensitivity of doubly robust methods to various types and degrees of misspecification. For instance in the example where both models were incorrect, one could imagine scenarios where model misspecifications cancel out, and actually produce a relatively accurate ATE estimate. It is an area of active research on when doubly robust methods should be used when there might be uncertainty on both models.

2.2 TMLE

2.2.1 Background and Setup

Targeted Maximum Likelihood Estimation (TMLE) is a semi-parametric method with minimal assumptions on the underlying data distribution demonstrated by Van der Laan & Rubin in 2006. We will briefly walk through the steps of a TMLE estimation algorithm on the same data without diving too deep into the formulation.

The TMLE algorithm begins by first estimating a model by training and predicting a super learning ensemble of algorithms. In the hidden code block below, we do so using 9 models from pre-built libraries. We report the root mean squared errors for all algorithms, demonstrating the super learner ensemble performs best.

```
Train (5195, 31) (5195,) Test (5196, 31) (5196,)
Meta  (5195, 9) (5195,)
```

(continues on next page)

(continued from previous page)

```
LinearRegression: RMSE 0.828
ElasticNet: RMSE 0.990
SVR: RMSE 0.849
DecisionTreeRegressor: RMSE 1.023
KNeighborsRegressor: RMSE 0.887
AdaBoostRegressor: RMSE 0.827
BaggingRegressor: RMSE 0.900
RandomForestRegressor: RMSE 0.898
ExtraTreesRegressor: RMSE 0.954
Super Learner: RMSE 0.808
```

In the second step we use the super learner algorithm to estimate the expected value of the outcome using the treatment and confounders as predictors. Within this, there are three steps:

1. predict with the interventions
2. predict with every sample receiving no treatment
3. predict with every sample receiving treatment.

We can take the difference of the last two as an ATE estimate, which is effectively the g-estimation approach. We see below this provides a decent 1st estimate.

```
df_predict = df_categ.copy()
X = df_predict[df_predict.columns.drop([Y])].to_numpy()
Q_a = super_learner_predictions(X, models, meta_model)
df_predict['intervention'] = 0
X = df_predict[df_predict.columns.drop([Y])].to_numpy()
Q_0 = super_learner_predictions(X, models, meta_model)
df_predict['intervention'] = 1
X = df_predict[df_predict.columns.drop([Y])].to_numpy()
Q_1 = super_learner_predictions(X, models, meta_model)

df_tmle = pd.DataFrame([df_categ[Y].to_numpy(),df_categ[T].to_numpy(), Q_a,Q_0,Q_1]).T
df_tmle.columns = ['Y','D','Q_a','Q_0','Q_1']

df_tmle['Q_1'].mean() - df_tmle['Q_0'].mean()
```

```
0.4044480650676532
```

In the third step we obtain propensity scores (ps) and form a "clever covariate" from these values which will be used to refine our model. the inverse ps with indicator is added with the negative inverse of not being treated (1-ps) also multiplied with indicator if not being treated:

$$H(D,X) = \frac{I(D=1)}{\hat{\pi}(X_i)} - \frac{I(D=0)}{1-\hat{\pi}(X_i)}$$

```
T = 'intervention'
Y = 'achievement_score'
X = df_categ.columns.drop([T, Y])
ate, ps,_ = IPW(df_categ, X, T, Y)

df_tmle['H_1'] = 1/ps
df_tmle['H_0'] = -1/(1-ps)
df_tmle['H_a'] = df_tmle['D'] * df_tmle['H_1'] + (1-df_tmle['D']) * df_tmle['H_0']
```

In the fourth and fifth steps, we estimate the fluctuation parameter using the logit function:

$$logit(\mathbb{E}[Y|D,X]) = logit(\hat{\mathbb{E}}[Y|D,X]) = \epsilon H(D,X)$$

We then update out initial estimates with the fluctuation parameter adjustment.

```
eps_fit = np.polyfit(df_tmle['H_a'], df_tmle['Y'] - df_tmle['Q_a'], 1)[0]
df_tmle['Q_0_hat'] = df_tmle['Q_0'] + eps_fit * df_tmle['H_0']
df_tmle['Q_1_hat'] = df_tmle['Q_1'] + eps_fit * df_tmle['H_1']
df_tmle['Q_a_hat'] = df_tmle['Q_a'] + eps_fit * df_tmle['H_a']
```

```
TMLE_ate = df_tmle['Q_1_hat'].mean() - df_tmle['Q_0_hat'].mean()
print('TMLE TAE estimate: {:.4f}'.format(TMLE_ate))
```

```
TMLE TAE estimate: 0.3864
```

We see how the fluctuation adjusted outcomes estimates vastly improves the ATE to be more in line with AIPW. One major benefit of TMLE is a whole set of nice statistical and convergence properties. In this case, we can use something called the influence function to calculate a closed form standard error, unlike the empirical error estimates we gained by bootstrapping in the AIPW case.

$$\hat{IF} = (Y - \hat{\mathbb{E}} * [Y|D,X])H(D,X) + \hat{\mathbb{E}} * [Y|D=1,X] - \hat{\mathbb{E}} * [Y|D=0,X] - ATE_{TMLE}$$

$$SE = \sqrt{var(IF)/N}$$

Using the above method, we see we get a SE very similar to our AIPW empirical methods.

```
IF = (df_tmle['Y'] - df_tmle['Q_a_hat']) * df_tmle['H_a'] + df_tmle['Q_1_hat'] - df_
↪tmle['Q_0_hat'] - TMLE_ate
print('SE calculated from influcence function: {:.4f}'.format(np.sqrt(IF.var()/df_
↪tmle.shape[0])))
```

```
SE calculated from influcence function: 0.0166
```

2.2.2 References

1. Glynn, A. N., & Quinn, K. M. (2010). An introduction to the augmented inverse propensity weighted estimator. Political analysis, 18(1), 36-56.

2. https://matheusfacure.github.io/python-causality-handbook/12-Doubly-Robust-Estimation.html

3. Gruber S, van der Laan MJ. Targeted minimum loss based estimator that outperforms a given estimator. Int J Biostat. 2012 May 18;8(1):Article 11. doi: 10.1515/1557-4679.1332. PMID: 22628356; PMCID: PMC6052865.

CHAPTER

THREE

FWL THEOREM AND DOUBLE MACHINE LEARNING

Following up on the previous notebook where we covered several doubly robust methods (e.g., AIPW and TMLE), we will go through double machine learning (DML) in detail in this notebook. But before diving into theory let us understand why we need DML in the first place, shall we?

Augmented inverse propensity weighting (AIPW) is a modification of the inverse propensity weighting (IPW) that guarantees double roubustness and consistent average treatment effect(ATE) estimate even if 1) treatment/exposure model ($\hat{\pi}(x)$) or 2) outcome model ($\hat{\mu}(x)$) is misspecified GLynn and Quinn, 2009. Although AIPW provides a nice flexibility in estimating a consistent ATE, it does necessitate at least one model to be correctly specified. If both the models are incorrectly specified, the naive IPW outperforms AIPW. Similarly, targeted maximum likelihood estimation (TMLE) is a semiparametric estimation framework. TMLE tends to work well when the treatment is not a weak predictor of the outcome. If that's not the case, the estimation tends to be biased toward zero which obviously might not be the baseline truth.

The main objective of DML is to provide a general framework to estimating and performing inference on low-dimensional parameter (θ_0) in presence of high-dimensional nuisance parameter utilizing nonparametric machine learning methods. DML works for both binary and continuous treatment variables which is not the case for some of the doubly robust methods. As the name suggests, DML leverages "double" or two high-performance ML methods to estimate a high-quality θ_0. Specifically, the first ML algorithm is used for treatment model while the second algorithm is used for the outcome model. Finally, Frisch-Waugh-Lovell (FWL)-type residuals-on-residuals regressioin is utilized to get a de-biased estimate of θ_0. DML is also know as "debiased-ML" or "orthogonalized ML."

3.1 FWL Theorem/Orthogonalization

Orthogonalization (or equivalently FWL Theorem (Frisch and Waugh, 1933; Lovell, 1963) is the backbone of DML. Its principled approach guarantees an unbiased estimate. Since it is a key to understanding why DML works, we will first prove the FWL Theorem and implement it in an example to demonstrate how it debiases the data before moving on to the details of DML.

Let us take a multivariate linear regression \$Y $= D_1\beta_1 + X\beta_2 + \epsilon where Y is$ n \times 1 $outcome$,D isn \times p_1 $treatment variables, and X is$n \times p_2\$ covariates or nuisance parameters.

Multiply the equation with residual maker function (G) of the treatment parameters D. The residual maker is defined by $Gy = y - D(D'D)^{-1}D'y \equiv y - D\beta \equiv \epsilon_D$

$$GY = GD_1\beta_1 + GX\beta_2 + G\epsilon$$

Since $GD_1\beta_1 \equiv 0$, the equation above simplifies to

$$GY = GX\beta_2 + G\epsilon$$

Now, the final equation becomes

$$GY = GX\beta_2 + \epsilon$$

Taking a closer look at the equation, we can see that we are regressing residuals on residuals. This shows that results obtained from multivariate linear regression is the same as the residuals-on-residuals regression.

Now that we have seen a proof of why residuals-on-residuals regression work, let us go through an example implementation to see orthogonalization in action.

```
/Users/sunaybhat/miniconda3/envs/stats256/lib/python3.10/site-packages/xgboost/
↪compat.py:36: FutureWarning: pandas.Int64Index is deprecated and will be removed␣
↪from pandas in a future version. Use pandas.Index with the appropriate dtype␣
↪instead.
  from pandas import MultiIndex, Int64Index
```

3.1.1 Orthogonalization: Example

To demonstrate how orthogonalization debiases the data, we will use a simulated data on ice cream sales. The outcome (Y) is the number of sales, the treatment is price, and the covariates (X) are temperature, weekday (categorical variable) and cost of the ice cream.

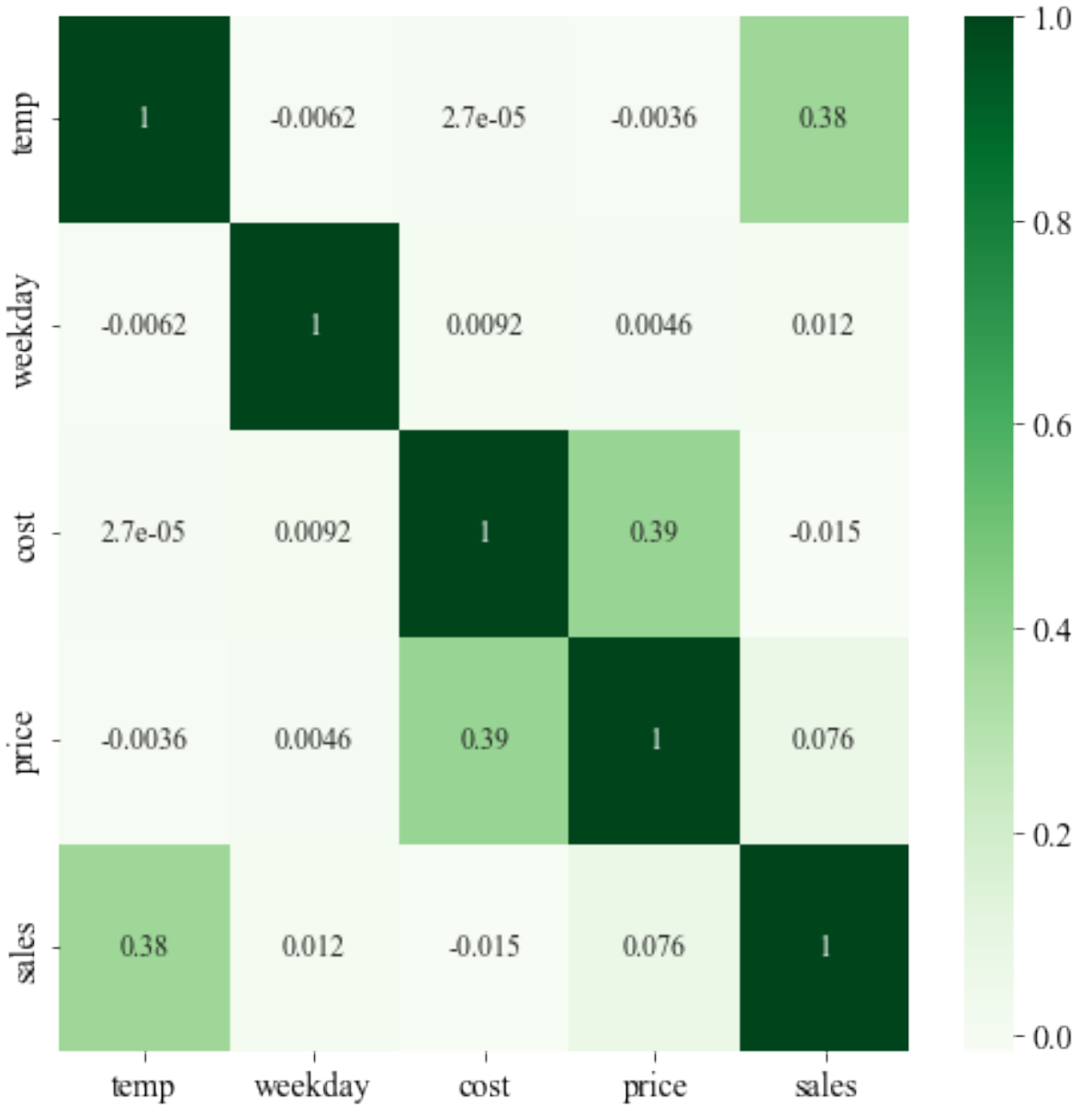
```
df_ortho = pd.read_csv('ice_cream_sales.csv')
df_ortho.head()
```

	temp	weekday	cost	price	sales
0	17.3	6	1.5	5.6	173
1	25.4	3	0.3	4.9	196
2	23.3	5	1.5	7.6	207
3	26.9	1	0.3	5.3	241
4	20.2	1	1.0	7.2	227

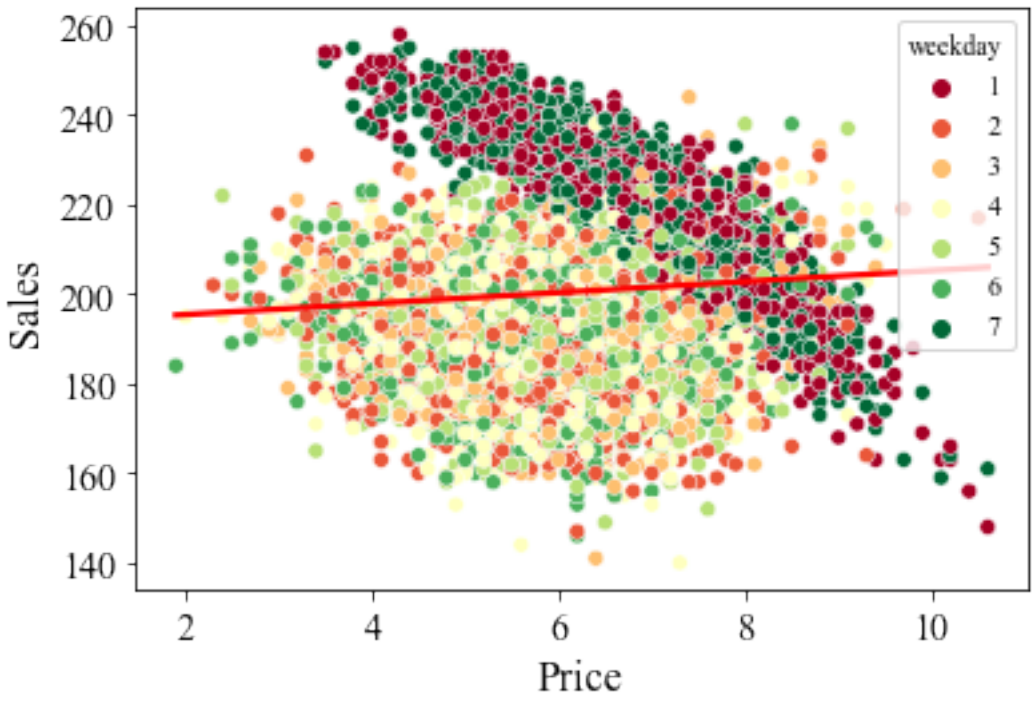
There are no missing data as we can see below:

```
temp       0
weekday    0
cost       0
price      0
sales      0
dtype: int64
```

The figure below shows a heatmap of the Pearsons' correlation plot of the dataset. The correlation plot shows positive linear relationship between three pairs of variables (sales-temp, cost-price, and price-sales)- two of which makes sense, one does not. As the temperature increases, we often expect ice cream sales to increase because people buy more ice cream if it is hot. Similarly, the price of the ice crease will increase if the purchase cost for the vendor is high. However, the third positive correlation is between price and sales which necessarily doesn't make sense because if the price is high, people tend to buy less so if anything, there should be a negative correlation. The positive correlation could potentially because of bias.



Looking at the scatter plot between sales and price, we can see that the data clearly is biased. First, we can see the two distinct cluster. On weekends, the sales is high because more people go outside which increases the demand. The vendors likely take an advantage of the increased demands and hike up the prices which ultimately reduces the sales. However, the sales appear to be roughly uniform regardless of the price during weekdays. The higher sales on weekends and the consistent sales during weekdays gives a positive relationship between sales and price as shows by a linear fit line (red line) in the figure below.

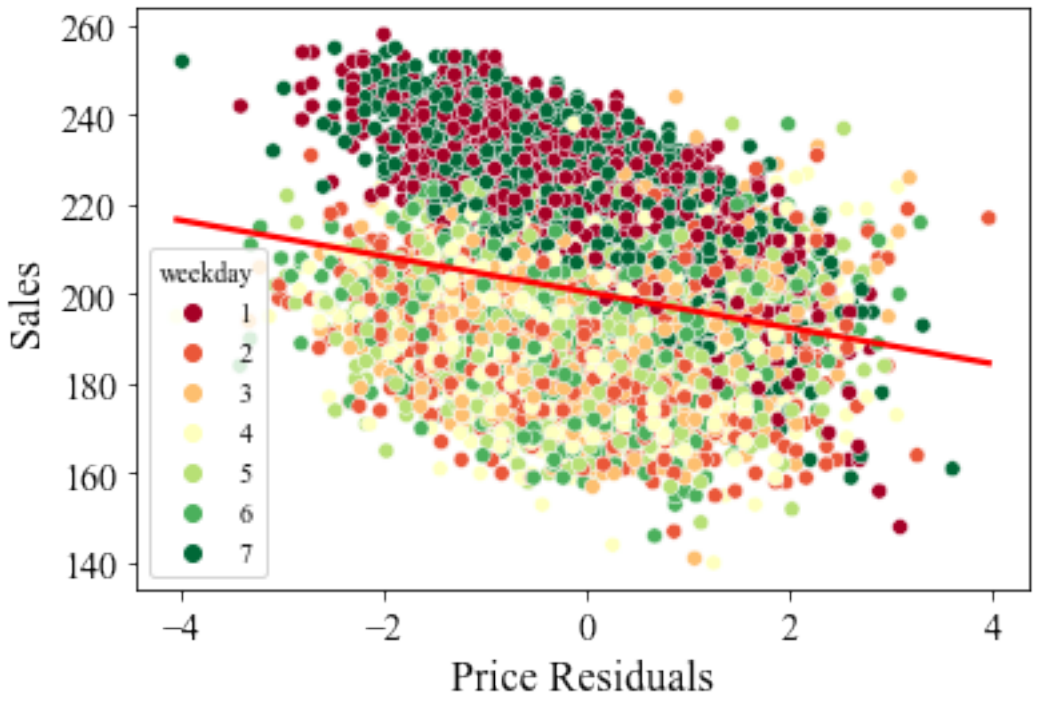


To debiase the data, we need two models- treatment and outcome model. The treatment model debiases the bias induced in price using all the other confounders, while the outcome model debiases the bias in sales introduced by the same covariates. Consistent with FWL Theorem, we used OLS to create the treatment and outcome models as shown below:

```
#create a treatment model
model_treatment = smf.ols("price ~ cost + C(weekday) + temp", data=df_ortho).fit()
#create an outcome model
model_outcome = smf.ols("sales ~ cost + C(weekday) + temp", data=df_ortho).fit()

debiased_df_ortho = df_ortho.assign(**{"resid_output_sales":model_outcome.resid,
                                       'resid_treatment_price':model_treatment.resid})
```

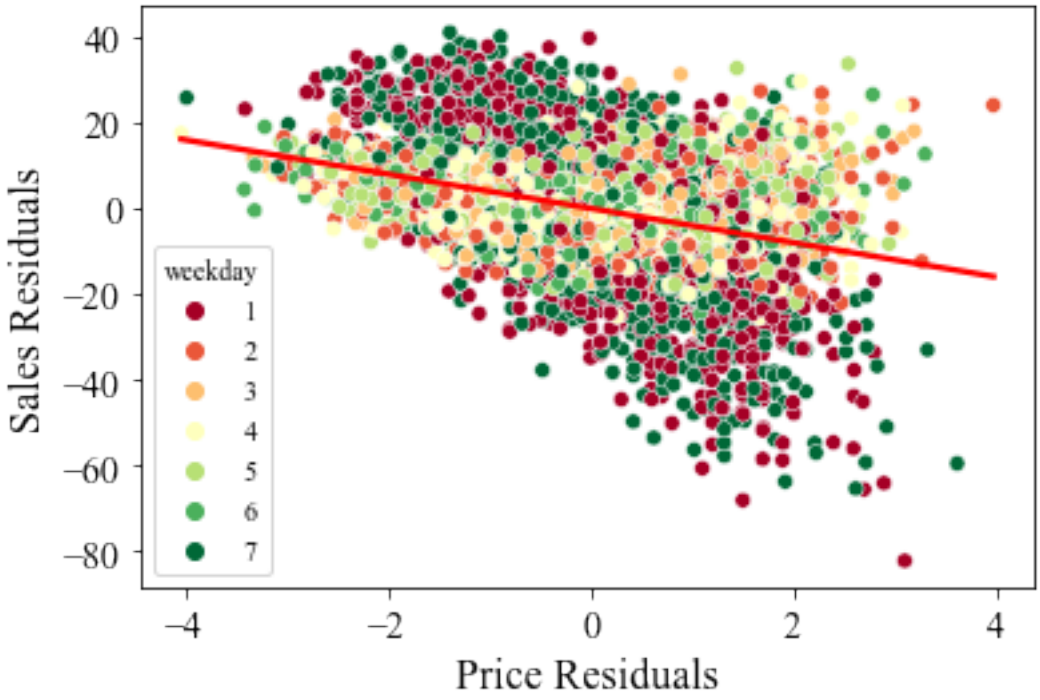
If we plot price-residuals against sales, we can see that we have debiased the bias in price. First, we have uncovered the negative relationship between the price and sales as expected. Most importantly, we can see that decline in sales during the weekend is consistent and not necessarily depending on the price. In the raw data above, we saw that as the price increased, the sales decreased drastically, thus inducing price bias. But in this case, the number of sales on the left and the right side of price-residual roughly appears to be the same.

Even though we have debiased the price, we can still see that the data has two distinct clusters as a function of the sale day. During weekends, the sales seems to be higher as compared to the weekdays.



Finally, lets see if we can debiase the bias in sales amount. The figure below plots sales-residuals against price-residuals. We can see that the we no longer have distinct clusters of data neither do we see a dramatic decline in sales as the price increase. The slope of the linear fit line (red line) is the debiased estimated of ATE that is obtained by regressing price-residuals on sales-residuals.

```
plt.figure()
sns.scatterplot(data=debiased_df_ortho, x="resid_treatment_price", y="resid_output_
↪sales",
                hue="weekday", palette='RdYlGn', legend='full')
sns.regplot(x='resid_treatment_price', y='resid_output_sales', data=debiased_df_ortho,
            scatter=False, ci=False, color ='red')
plt.xlabel('Price Residuals', fontsize=16)
plt.ylabel('Sales Residuals', fontsize=16)
# plt.savefig('scatter_doubleDebiased_iceCream.png', bbox_inches='tight')
plt.show()
```



This example illustrates the role orthogonalization playing in debiasing the data and more importantly estimating a debiased ATE estimate.

3.2 Debiased/Double Machine Learning

Now that we understand orthogonalization, we will dig a little deeper into DML formulations. Good news is that if you understand the intuition behind orthogonalization, you already understand DML. To simplify, DML is fundamentally the same as orthogonalization except that advanced ML algorithms are used to model treatment and outcome instead of OLS.

One may wonder, ML algorithms are widely used in anything and everything, haven't researchers used to estimate treatment effect? Yes, they have and that's when naive approach comes in. In this section, naive approach refers to methods involving ML methods with no modifications. Since DML is a slighly different class of roubly-robust method in a sense that it utilized ML algorith, we introduce Naive or prediction-based ML approach to make a direct comparison against DML and show why DML is better than the naive approach.

To be consistent with the Double/Debiased Machine Learning Paper(Chernozhukov et al., 2018), we will use the same notations.

Let us take a partial linear regression, $$Y = D\theta_0 + g_0(X) + U, \quad E[U|X, D] = 0$$

$$D = m_0(X) + V, \quad E[V|X] = 0$$

where Y is the outcome, D is the treatment, X is the covariates/confounders, and U and V are the noise. The quantity of interest is the regression coefficient, θ_0.

Under naive approach, the following steps are undertaken to estimate the θ_0

1. Predict Y using D and X. This gives you $\hat{Y}$ in the form of $D\hat{\theta}_0 + \hat{g}_0(X)$
2. Run advanced ML algorithm (e.g., Random Forest) of $Y - D\hat{\theta}_0$ on X to fit $\hat{g}_0(X)$
3. Run OLS of $Y - \hat{g}_0(X)$ on D to fit $\hat{\theta}_0$. In other words, $\hat{\theta}_0$ is given by:

$$\hat{\theta}_0 = \left(\frac{1}{n}\sum_{i\in I} D_i^2\right)^{-1} \frac{1}{n}\sum_{i\in I} D_i(Y_i - \hat{g}(X_i))$$

The naive approach displays excellent predictive performance but introduces a regularization bias in learning g_0. Lets take a closer look at the decomposition of the estimation error in $\hat{\theta}_0$ to islolate the regularization bias,

$$\sqrt{n}(\hat{\theta}_0 - \theta_0) = \underbrace{\left(\frac{1}{n}\sum_{i\in I} D_i^2\right)^{-1} \frac{1}{\sqrt{n}}\sum_{i\in I} D_i U_i}_{:a} + \underbrace{\left(\frac{1}{n}\sum_{i\in I} D_i^2\right)^{-1} \frac{1}{\sqrt{n}}\sum_{i\in I} D_i(g_0(X_i) - \hat{g}_0(X_i))}_{:b}$$

The first term a is well-behaved under mild conditions and has zero mean $a \rightsquigarrow N(0, \bar{\Sigma})$ for some $\bar{\Sigma}$. However, the regularization term (b) does not center around 0, and in fact diverges for the majority of the ML algorithms. The regularization bias is addressed using orthogonalization. How exactly does DML do it? Using the following three steps:

1. Predict Y and D using X using the advanced ML methods to obtain $\widehat{E[Y|X]}$ and $\widehat{E[D|X]}$
2. Obtain residuals from the two models i.e. $\widehat{W} = Y - \widehat{E[Y|X]}$ and $\widehat{V} = D - \widehat{E[D|X]}$
3. Use orthogonalization, i.e. regress $\widehat{W}$ on $\widehat{V}$ to get $\hat{\theta}_0$

In DML, the estimation error in $\hat{\theta}_0$ can be decomposed into

$$\sqrt{n}(\hat{\theta}_0 - \theta_0) = a^* + b^* + c^*$$

where, $$a^* = (EV^2)^{-1}\frac{1}{\sqrt{n}}\sum_{i\in I} V_i U_i \rightsquigarrow N(0, \Sigma),$$

$$b^* = (EV^2)^{-1}\frac{1}{\sqrt{n}}\sum_{i\in I}(\hat{m}_0(X_i) - m_0(X_i))(\hat{g}_0(X_i) - g_0(X_i))$$

and,

$$c^* = o_P(1)$$

Similar to naive approach, a^* has a zero mean. The second term, b^* also nearly as zero mean because the the high predictive performance of advanced ML algorithms ensure that the product of the estimation error in $\hat{m}_0$ and $\hat{g}_0$ nearly vanishes to zero. The c^* term represents bias induced due to overfitting, which is sufficiently well-behaved and vanishes in probability under sample splitting. For DML to be doubly robust, it is paramount to split the data into multiple folds while estimating $\hat{\theta}_0$. For a detailed proof on why c^* vanishes in probability in presence of sample splitting, we invite readers to read the Double/Debiased Machine Learning Paper.

3.2.1 DML: Example Implementation

In this example, we will use the data on 401(k) eligibility (treatment variable) on the total accumulated net financial assess (net_tfa). This dataset was assembled from the 1991 Survey of Income and Porgram Participation. Since the assignment was not random, the DML is implemented to negate bias due to non-randomness assignment.

```
df_401k = fetch_401K(return_type='DataFrame')
df_401k.sample(n=5)
```

```
         nifa  net_tfa        tw  age      inc  fsize  educ  db  marr  \
7831  15750.0   38650.0   78650.0   32  47628.0      2    12   1     1
9229  82202.0   63176.0  114176.0   37  69264.0      4    16   0     1
3051    550.0 -17550.0   -9550.0   45  48000.0      4    14   0     1
```

(continues on next page)

(continued from previous page)

```
8457   2050.0  11750.0  111750.0   39  71829.0      4    12   0      1
7057    480.0 -21130.0  -19538.0   33  45300.0      3    12   1      1

      twoearn  e401  p401  pira  hown
7831        1     1     1     0     0
9229        0     1     1     0     1
3051        1     0     0     0     0
8457        0     1     1     0     1
7057        1     1     0     0     0
```

The description of the features are highlighted below:

1. **age:** age of the employee
2. **inc:** income amount
3. **fsize:** family size
4. **educ:** years of education
5. **marr:** marriage indicator (1: married, 0: otherwise)
6. **twoearn:** two-earner status indicator in the family
7. **db:** a defined benefit pension status indicator
8. **pira:** an IRA participation indicator
9. **hown:** a home ownership indicator
10. **net_tfa:** net total financial assets. Defined as the sum fo IRA balances, 401(k) balances, checking accounts, U.S. saving bonds, stocks, mutual funds, etc.

As discussed below, we will not use all of the features in this example.

```
df_401k.shape
```

```
(9915, 14)
```

Difference in mean between the employees who were eligible vs not eligible.

```
df_401k.groupby('e401')['net_tfa'].mean()
```

```
e401
0    10788.044922
1    30347.388672
Name: net_tfa, dtype: float32
```

Difference in mean between the employees who opted to participate in 401(k) vs those that did not participate.

```
df_401k.groupby('p401')['net_tfa'].mean()
```

```
p401
0    10890.477539
1    38262.058594
Name: net_tfa, dtype: float32
```

For consistency, we will use the same covariates used in the Double/Debiased Machine Learning Paper.

```
features_cherno = [ 'age', 'inc', 'fsize', 'educ', 'marr', 'twoearn', 'db', 'pira',
 ↪'hown']
```

```
#outcome model
my = smf.ols('net_tfa ~ ' + '+'.join(features_cherno), data = df_401k).fit()

#treatment model
mt = smf.ols('e401 ~  ' + '+'.join(features_cherno), data = df_401k).fit()
```

One of the limitations we have not mentioned before is that the orthogonalization is limited to linear relationship between the covariates, treatment, and outcome models. Of course, the linear regression can be extended to a polynomial regression to capture nonlinear relationship but having to specify the nonlinear functional form is not that flexible and desirable. We implement linear and polynomial regression to shed light on the high-level predictive performance of commonly used ML algorithms.

```
orthogonal = smf.ols("tfa_res~e401_res",
        data=df_401k.assign(tfa_res=my.resid, # sales residuals
                            e401_res=mt.resid) # price residuals
       ).fit()
orthogonal.summary().tables[1]
```

```
<class 'statsmodels.iolib.table.SimpleTable'>
```

	coef	std err	2.5 %	97.5 %
0	5896.198421	1249.446	3447.029	8345.367

Creating a polynomial regression

```
my_poly = smf.ols('net_tfa ~ age + inc + educ + fsize + marr + twoearn + db + pira +␣
 ↪hown + age*fsize + educ*age + fsize**2',
                  data = df_401k).fit()
mt_poly = smf.ols('e401 ~ age + inc + educ + fsize + marr + twoearn + db + pira + hown
 ↪', data = df_401k).fit()
```

```
polynomial = smf.ols("tfa_res~e401_res",
        data=df_401k.assign(tfa_res=my_poly.resid, # sales residuals
                            e401_res=mt_poly.resid) # price residuals
       ).fit()
```

As we can see, for the polynomial regression, the estimated ATE is slighly higher by about $200 as compared to OLS. This indicates that the data is highly nonlinear and we can leverage the ML algorithms to capture the nonlinear relationship.

	coef	std err	2.5 %	97.5 %
0	6084.770503	1247.310	3639.789	8529.752

```
# Initialize DoubleMLData (data-backend of DoubleML)
data_dml_base = dml.DoubleMLData(df_401k,
                                 y_col='net_tfa',
                                 d_cols='e401',
                                 x_cols=features_cherno)
```

In the following section, Random Forest, Xtreme Gradient Boosting (XGBoost), and Regression trees are implemented to estimate the ATE in terms of the total financial asset. Two variations of implementation are provided (3 folds and 5 folds data splits) to highlight the role sample spliting plays in reducing the bias.

```
# Random Forest with 3 folds split
randomForest = RandomForestRegressor(
    n_estimators=500, max_depth=7, max_features=3, min_samples_leaf=3)
randomForest_class = RandomForestClassifier(
    n_estimators=500, max_depth=5, max_features=4, min_samples_leaf=7)

np.random.seed(123)
dml_plr_forest = dml.DoubleMLPLR(data_dml_base,
                                 ml_g = randomForest,
                                 ml_m = randomForest_class,
                                 n_folds = 3,
                                n_rep=10)
dml_plr_forest.fit(store_predictions=True)
forest_summary3 = dml_plr_forest.summary

forest_summary3
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Input In [22], in <cell line: 13>()
      7 np.random.seed(123)
      8 dml_plr_forest = dml.DoubleMLPLR(data_dml_base,
      9                                  ml_g = randomForest,
     10                                  ml_m = randomForest_class,
     11                                  n_folds = 3,
     12                                 n_rep=10)
---> 13 dml_plr_forest.fit(store_predictions=True)
     14 forest_summary3 = dml_plr_forest.summary
     16 forest_summary3

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/doubleml/double_ml.
↪py:477, in DoubleML.fit(self, n_jobs_cv, keep_scores, store_predictions)
    473     self._dml_data.set_x_d(self._dml_data.d_cols[i_d])
    475 # ml estimation of nuisance models and computation of score elements
    476 self._psi_a[:, self._i_rep, self._i_treat], self._psi_b[:, self._i_rep,␣
↪self._i_treat], preds =\
--> 477     self._nuisance_est(self.__smpls, n_jobs_cv)
    479 if store_predictions:
    480     self._store_predictions(preds)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/doubleml/double_ml_
↪plr.py:148, in DoubleMLPLR._nuisance_est(self, smpls, n_jobs_cv)
    144 x, d = check_X_y(x, self._dml_data.d,
    145                  force_all_finite=False)
    147 # nuisance g
--> 148 g_hat = _dml_cv_predict(self._learner['ml_g'], x, y, smpls=smpls, n_jobs=n_
↪jobs_cv,
    149                         est_params=self._get_params('ml_g'), method=self._
↪predict_method['ml_g'])
    150 _check_finite_predictions(g_hat, self._learner['ml_g'], 'ml_g', smpls)
    152 # nuisance m

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/doubleml/_utils.
↪py:100, in _dml_cv_predict(estimator, x, y, smpls, n_jobs, est_params, method,
↪return_train_preds)
```

(continues on next page)

(continued from previous page)

```
     97 if not manual_cv_predict:
     98     if est_params is None:
     99         # if there are no parameters set we redirect to the standard method
--> 100         preds = cross_val_predict(clone(estimator), x, y, cv=smpls, n_
 ↪jobs=n_jobs, method=method)
    101     else:
    102         assert isinstance(est_params, dict)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/model_
 ↪selection/_validation.py:962, in cross_val_predict(estimator, X, y, groups, cv,␣
 ↪n_jobs, verbose, fit_params, pre_dispatch, method)
    959 # We clone the estimator to make sure that all the folds are
    960 # independent, and that it is pickle-able.
    961 parallel = Parallel(n_jobs=n_jobs, verbose=verbose, pre_dispatch=pre_
 ↪dispatch)
--> 962 predictions = parallel(
    963     delayed(_fit_and_predict)(
    964         clone(estimator), X, y, train, test, verbose, fit_params, method
    965     )
    966     for train, test in splits
    967 )
    969 inv_test_indices = np.empty(len(test_indices), dtype=int)
    970 inv_test_indices[test_indices] = np.arange(len(test_indices))

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
 ↪py:1046, in Parallel.__call__(self, iterable)
   1043 if self.dispatch_one_batch(iterator):
   1044     self._iterating = self._original_iterator is not None
-> 1046 while self.dispatch_one_batch(iterator):
   1047     pass
   1049 if pre_dispatch == "all" or n_jobs == 1:
   1050     # The iterable was consumed all at once by the above for loop.
   1051     # No need to wait for async callbacks to trigger to
   1052     # consumption.

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
 ↪py:861, in Parallel.dispatch_one_batch(self, iterator)
    859     return False
    860 else:
--> 861     self._dispatch(tasks)
    862     return True

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
 ↪py:779, in Parallel._dispatch(self, batch)
    777 with self._lock:
    778     job_idx = len(self._jobs)
--> 779     job = self._backend.apply_async(batch, callback=cb)
    780     # A job can complete so quickly than its callback is
    781     # called before we get here, causing self._jobs to
    782     # grow. To ensure correct results ordering, .insert is
    783     # used (rather than .append) in the following line
    784     self._jobs.insert(job_idx, job)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/_parallel_
 ↪backends.py:208, in SequentialBackend.apply_async(self, func, callback)
    206 def apply_async(self, func, callback=None):
```

(continues on next page)

(continued from previous page)

```
    207     """Schedule a func to be run"""
--> 208     result = ImmediateResult(func)
    209     if callback:
    210         callback(result)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/_parallel_
↪backends.py:572, in ImmediateResult.__init__(self, batch)
    569 def __init__(self, batch):
    570     # Don't delay the application, to avoid keeping the input
    571     # arguments in memory
--> 572     self.results = batch()

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
↪py:262, in BatchedCalls.__call__(self)
    258 def __call__(self):
    259     # Set the default nested backend to self._backend but do not set the
    260     # change the default number of processes to -1
    261     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 262         return [func(*args, **kwargs)
    263                 for func, args, kwargs in self.items]

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
↪py:262, in <listcomp>(.0)
    258 def __call__(self):
    259     # Set the default nested backend to self._backend but do not set the
    260     # change the default number of processes to -1
    261     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 262         return [func(*args, **kwargs)
    263                 for func, args, kwargs in self.items]

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/utils/fixes.
↪py:216, in _FuncWrapper.__call__(self, *args, **kwargs)
    214 def __call__(self, *args, **kwargs):
    215     with config_context(**self.config):
--> 216         return self.function(*args, **kwargs)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/model_
↪selection/_validation.py:1044, in _fit_and_predict(estimator, X, y, train, test,␣
↪verbose, fit_params, method)
  1042     estimator.fit(X_train, **fit_params)
  1043 else:
-> 1044     estimator.fit(X_train, y_train, **fit_params)
  1045 func = getattr(estimator, method)
  1046 predictions = func(X_test)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/ensemble/_
↪forest.py:450, in BaseForest.fit(self, X, y, sample_weight)
    439 trees = [
    440     self._make_estimator(append=False, random_state=random_state)
    441     for i in range(n_more_estimators)
    442 ]
    444 # Parallel loop: we prefer the threading backend as the Cython code
    445 # for fitting the trees is internally releasing the Python GIL
    446 # making threading more efficient than multiprocessing in
    447 # that case. However, for joblib 0.12+ we respect any
    448 # parallel_backend contexts set at a higher level,
```

(continues on next page)

(continued from previous page)

```
    449 # since correctness does not rely on using threads.
--> 450 trees = Parallel(
    451     n_jobs=self.n_jobs,
    452     verbose=self.verbose,
    453     **_joblib_parallel_args(prefer="threads"),
    454 )(
    455     delayed(_parallel_build_trees)(
    456         t,
    457         self,
    458         X,
    459         y,
    460         sample_weight,
    461         i,
    462         len(trees),
    463         verbose=self.verbose,
    464         class_weight=self.class_weight,
    465         n_samples_bootstrap=n_samples_bootstrap,
    466     )
    467     for i, t in enumerate(trees)
    468 )
    470 # Collect newly grown trees
    471 self.estimators_.extend(trees)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
↪py:1046, in Parallel.__call__(self, iterable)
   1043 if self.dispatch_one_batch(iterator):
   1044     self._iterating = self._original_iterator is not None
-> 1046 while self.dispatch_one_batch(iterator):
   1047     pass
   1049 if pre_dispatch == "all" or n_jobs == 1:
   1050     # The iterable was consumed all at once by the above for loop.
   1051     # No need to wait for async callbacks to trigger to
   1052     # consumption.

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
↪py:861, in Parallel.dispatch_one_batch(self, iterator)
    859     return False
    860 else:
--> 861     self._dispatch(tasks)
    862     return True

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
↪py:779, in Parallel._dispatch(self, batch)
    777 with self._lock:
    778     job_idx = len(self._jobs)
--> 779     job = self._backend.apply_async(batch, callback=cb)
    780     # A job can complete so quickly than its callback is
    781     # called before we get here, causing self._jobs to
    782     # grow. To ensure correct results ordering, .insert is
    783     # used (rather than .append) in the following line
    784     self._jobs.insert(job_idx, job)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/_parallel_
↪backends.py:208, in SequentialBackend.apply_async(self, func, callback)
    206 def apply_async(self, func, callback=None):
    207     """Schedule a func to be run"""
```

(continues on next page)

(continued from previous page)

```
--> 208     result = ImmediateResult(func)
    209     if callback:
    210         callback(result)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/_parallel_
↪backends.py:572, in ImmediateResult.__init__(self, batch)
    569 def __init__(self, batch):
    570     # Don't delay the application, to avoid keeping the input
    571     # arguments in memory
--> 572     self.results = batch()

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
↪py:262, in BatchedCalls.__call__(self)
    258 def __call__(self):
    259     # Set the default nested backend to self._backend but do not set the
    260     # change the default number of processes to -1
    261     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 262         return [func(*args, **kwargs)
    263                 for func, args, kwargs in self.items]

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/joblib/parallel.
↪py:262, in <listcomp>(.0)
    258 def __call__(self):
    259     # Set the default nested backend to self._backend but do not set the
    260     # change the default number of processes to -1
    261     with parallel_backend(self._backend, n_jobs=self._n_jobs):
--> 262         return [func(*args, **kwargs)
    263                 for func, args, kwargs in self.items]

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/utils/fixes.
↪py:216, in _FuncWrapper.__call__(self, *args, **kwargs)
    214 def __call__(self, *args, **kwargs):
    215     with config_context(**self.config):
--> 216         return self.function(*args, **kwargs)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/ensemble/_
↪forest.py:185, in _parallel_build_trees(tree, forest, X, y, sample_weight, tree_
↪idx, n_trees, verbose, class_weight, n_samples_bootstrap)
    182     elif class_weight == "balanced_subsample":
    183         curr_sample_weight *= compute_sample_weight("balanced", y,␣
↪indices=indices)
--> 185     tree.fit(X, y, sample_weight=curr_sample_weight, check_input=False)
    186 else:
    187     tree.fit(X, y, sample_weight=sample_weight, check_input=False)

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/tree/_classes.
↪py:1315, in DecisionTreeRegressor.fit(self, X, y, sample_weight, check_input, X_
↪idx_sorted)
  1278 def fit(
  1279     self, X, y, sample_weight=None, check_input=True, X_idx_sorted=
↪"deprecated"
  1280 ):
  1281     """Build a decision tree regressor from the training set (X, y).
  1282
  1283     Parameters
  (...)
```

(continues on next page)

(continued from previous page)

```
   1312         Fitted estimator.
   1313     """
-> 1315     super().fit(
   1316         X,
   1317         y,
   1318         sample_weight=sample_weight,
   1319         check_input=check_input,
   1320         X_idx_sorted=X_idx_sorted,
   1321     )
   1322     return self

File ~/miniconda3/envs/stats256/lib/python3.10/site-packages/sklearn/tree/_classes.
 ↪py:420, in BaseDecisionTree.fit(self, X, y, sample_weight, check_input, X_idx_
 ↪sorted)
    409 else:
    410     builder = BestFirstTreeBuilder(
    411         splitter,
    412         min_samples_split,
   (...)
    417         self.min_impurity_decrease,
    418     )
--> 420 builder.build(self.tree_, X, y, sample_weight)
    422 if self.n_outputs_ == 1 and is_classifier(self):
    423     self.n_classes_ = self.n_classes_[0]

KeyboardInterrupt:
```

```
# Random Forest with 5 folds split
randomForest = RandomForestRegressor(
    n_estimators=500, max_depth=7, max_features=3, min_samples_leaf=3)
randomForest_class = RandomForestClassifier(
    n_estimators=500, max_depth=5, max_features=4, min_samples_leaf=7)

np.random.seed(123)
dml_plr_forest = dml.DoubleMLPLR(data_dml_base,
                                 ml_g = randomForest,
                                 ml_m = randomForest_class,
                                 n_folds = 5,
                                n_rep=10)
dml_plr_forest.fit(store_predictions=True)
forest_summary5 = dml_plr_forest.summary

forest_summary5
```

```
              coef      std err         t         P>|t|        2.5 %  \
e401  8961.175025  1309.593551  6.842715  7.770632e-12  6394.418831

             97.5 %
e401  11527.931219
```

```
# Gradient Boosted Trees with 3 folds split
boost = XGBRegressor(n_jobs=1, objective = "reg:squarederror",
                     eta=0.1, n_estimators=35)
boost_class = XGBClassifier(use_label_encoder=False, n_jobs=1,
```

(continues on next page)

(continued from previous page)

```
                            objective = "binary:logistic", eval_metric = "logloss",
                            eta=0.1, n_estimators=34)

np.random.seed(123)
dml_plr_boost = dml.DoubleMLPLR(data_dml_base,
                                ml_g = boost,
                                ml_m = boost_class,
                                dml_procedure='dml2',
                                n_folds = 3,
                                n_rep = 10)
dml_plr_boost.fit(store_predictions=True)
boost_summary3 = dml_plr_boost.summary

boost_summary3
```

```
              coef      std err         t         P>|t|        2.5 %  \
e401  9002.744739  1399.883887  6.431065  1.267127e-10  6259.022737

            97.5 %
e401  11746.46674
```

```
# Gradient Boosted Trees with 5 folds split
boost = XGBRegressor(n_jobs=1, objective = "reg:squarederror",
                     eta=0.1, n_estimators=35)
boost_class = XGBClassifier(use_label_encoder=False, n_jobs=1,
                            objective = "binary:logistic", eval_metric = "logloss",
                            eta=0.1, n_estimators=34)

np.random.seed(123)
dml_plr_boost = dml.DoubleMLPLR(data_dml_base,
                                ml_g = boost,
                                ml_m = boost_class,
                                dml_procedure='dml2',
                                n_folds = 5,
                                n_rep = 10)
dml_plr_boost.fit(store_predictions=True)
boost_summary5 = dml_plr_boost.summary

boost_summary5
```

```
              coef      std err         t         P>|t|        2.5 %  \
e401  8852.014728  1383.993593  6.395994  1.595063e-10  6139.437131

            97.5 %
e401  11564.592325
```

```
# Regression Decision Trees with 3 folds split
trees = DecisionTreeRegressor(
    max_depth=30, ccp_alpha=0.0047, min_samples_split=203, min_samples_leaf=67)
trees_class = DecisionTreeClassifier(
    max_depth=30, ccp_alpha=0.0042, min_samples_split=104, min_samples_leaf=34)

np.random.seed(123)
```

(continues on next page)

(continued from previous page)

```
dml_plr_tree = dml.DoubleMLPLR(data_dml_base,
                               ml_g = trees,
                               ml_m = trees_class,
                               n_folds = 3,
                               n_rep = 10)
dml_plr_tree.fit(store_predictions=True)
tree_summary3 = dml_plr_tree.summary

tree_summary3
```

```
             coef      std err         t         P>|t|        2.5 %  \
e401  8494.390142  1332.352929  6.375481  1.823902e-10  5883.026386

            97.5 %
e401  11105.753898
```

```
# Regression Decision Trees with 3 folds split
trees = DecisionTreeRegressor(
    max_depth=30, ccp_alpha=0.0047, min_samples_split=203, min_samples_leaf=67)
trees_class = DecisionTreeClassifier(
    max_depth=30, ccp_alpha=0.0042, min_samples_split=104, min_samples_leaf=34)

np.random.seed(123)
dml_plr_tree = dml.DoubleMLPLR(data_dml_base,
                               ml_g = trees,
                               ml_m = trees_class,
                               n_folds = 5,
                               n_rep = 10)
dml_plr_tree.fit(store_predictions=True)
tree_summary5 = dml_plr_tree.summary

tree_summary5
```

```
             coef      std err         t         P>|t|        2.5 %  \
e401  8365.634772  1319.168039  6.341599  2.273925e-10  5780.112926

            97.5 %
e401  10951.156619
```

```
               coef      std err        2.5 %        97.5 %
forest   8961.175025  1309.593551  6394.418831  11527.931219
tree     8365.634772  1319.168039  5780.112926  10951.156619
xgboost  8852.014728  1383.993593  6139.437131  11564.592325
```

The summary of orthogonalization, DML with 3 folds, and DML with 5 folds sample splits are shown in the dataframe below. We can see that, as we increase the sample splits, the standard error decrease which gives us a tighter confidence bounds and a more robust ATE estimate

```
                                 coef      std err        2.5 %        97.5 %
Model         ML
Orthogonal    linear      5896.198421     1249.446     3447.029      8345.367
              polynomial  6084.770503     1247.310     3639.789      8529.752
PLR (3 folds) forest      9018.368261  1315.291812   6440.44368  11596.292842
```

(continues on next page)

(continued from previous page)

```
                tree        8494.390142  1332.352929  5883.026386  11105.753898
                xgboost     9002.744739  1399.883887  6259.022737   11746.46674
PLR (5 folds)   forest      8961.175025  1309.593551  6394.418831  11527.931219
                tree        8365.634772  1319.168039  5780.112926  10951.156619
                xgboost     8852.014728  1383.993593  6139.437131  11564.592325
```

3.2.2 Summary

Double Machine Learning (DML) leverages predictive power of advance Machine Learning (ML) algrotighms in estimating heterogeneous treatment effects when all potential confounders are observed and are also high-dimensional. At its core, DML utilizes orthogonalization to address the regularization bias induced by ML algorithm in estimating high-dimensional nuisance parameters. DML requires two ML methods to predict treatment and outcome using the observed covariates. The residuals from the treatment and outcome model is then used to estimate the causal parameter of interest, the treatment effect. The purpose of the treatment residuals is to represent the debiased version of the treatment model because, by definition, residuals are orthogonal to the features used to contruct the model. Similarly, the outcome residuals denoises the outcome model because the outcome residuals can essentially be viewed as a version of the treatment where all the variance from the features are explained. Thus, DML provides a general yet robust framework for estimating and performing inference on treatment/causal variables.

3.2.3 References

1. Chernozhukov, V., Chetverikov, D., Demirer, M., Duflo, E., Hansen, C., Newey, W., & Robins, J. (2018). Double/debiased machine learning for treatment and structural parameters.

2. Glynn, A. N., & Quinn, K. M. (2010). An introduction to the augmented inverse propensity weighted estimator. Political analysis, 18(1), 36-56.

3. https://matheusfacure.github.io/python-causality-handbook/22-Debiased-Orthogonal-Machine-Learning.html

4. https://www.youtube.com/watch?v=eHOjmyoPCFU&t=444s

5. Bach, P., Chernozhukov, V., Kurz, M. S., and Spindler, M. (2022), DoubleML - An Object-Oriented Implementation of Double Machine Learning in Python, Journal of Machine Learning Research, 23(53): 1-6, https://www.jmlr.org/papers/v23/21-0862.html.

6. Frisch, R., & Waugh, F. V. (1933). Partial time regressions as compared with individual trends. Econometrica: Journal of the Econometric Society, 387-401.

7. Lovell, M. C. (1963). Seasonal adjustment of economic time series and multiple regression analysis. Journal of the American Statistical Association, 58(304), 993-1010.

8. Lovell, M. C. (2008). A simple proof of the FWL theorem. The Journal of Economic Education, 39(1), 88-91.

CHAPTER

FOUR

CONCLUSION AND NEW DIRECTIONS

4.1 Summary of Estimators

- **AIPW** - It is a weighting based estimator that improves IPTW by fully utilizing information about both the treatment assignment and the outcome. It is a combination of IPTW and a weighted average of the outcome regression estimators.
- **TMLE** - It incorporates a targeting step that optimizes the bias-variance tradeoff for the targeted estimator, i.e., ATE. It obtains initial outcome estimates via outcome modeling and propensity scores via treatment modeling, respectively. These initial outcome estimates are then updated to reduce the bias of confounding, which generates the targeted predicted outcome values.
- **DML** - It utilizes predictive power of advanced ML algorithms in estimating heterogeneous treatment effects when all potential confounders are observed and are also high-dimensional.

4.2 Doubly Robust Methods: Applications

- Molecular Epidemiology - Meng et al. (2021) applied efficient estimators like AIPW and TMLE to estimate average treatment effects under various scenarios of mis-specification.
- Social Sciences - Knaus (2020) showed the efficiency of DML for the evaluation of programs of the Swiss Active Labour Market Policy.
- Medical Sciences - Rose et al. (2020) proposed the use of TMLE for the evaluation of the comparative effectiveness of drug-eluting coronary stents.

4.3 Potential Future Works (Tan et al. (2022))

- It is recommended to do a variable selection first, followed by using SuperLearner to model PS and outcomes. After that TMLE can be applied for estimating ATE.
- The use of ML algorithms like random forest and neural networks can be used to remove treatment predictors for variable selection.
- Soft variable selection strategies can be used where the variable selection is conducted without requiring any modeling on the outcome regression, and thus provides robustness against mis-specification.