Causal Inference Algorithms Evaluation Report

Group 3: Catherine Gao, Eve Washington, Siyuan Sang, Zi Fang

April 7, 2021

Project Overview

In this project, group 3 evaluates three causal inference algorithms to compute the average treatment effect (ATE) on two distict datasets and compare their computational efficiency and performance.

One dataset contains high dimensional data and another contains low dimensional data. We will use L1 penalized logistic regression to estimate the propensity scores for these two datasets, and apply the following three methods to calcualte ATE for each dataset:

Algorithm	Propensity Score Estimation
Propensity Scores Matching (full)	L1 penalized logistic regression
Doubly Robust Estimations	L1 penalized logistic regression
Stratification	L1 penalized logistic regression

This report includes a description of each algorithm, code to reproduce the results, and a comparison of the models.

2 Data Preparation

2.1 Load Required Packages

In both datasets, variable "Y" indicates the outcome variable and variable "A" indicates the treatment group assighnment. The remaining variables are covariates for consideration.

In [1]:

```
import numpy as np
import pandas as pd
import time
import copy
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings('ignore')
```

```
# setting graph styles
sns.set(rc={'figure.figsize':(10,8)})
sns.set_theme(style='ticks')

# set seed
random_state = 2021
```

2.2 Load Data

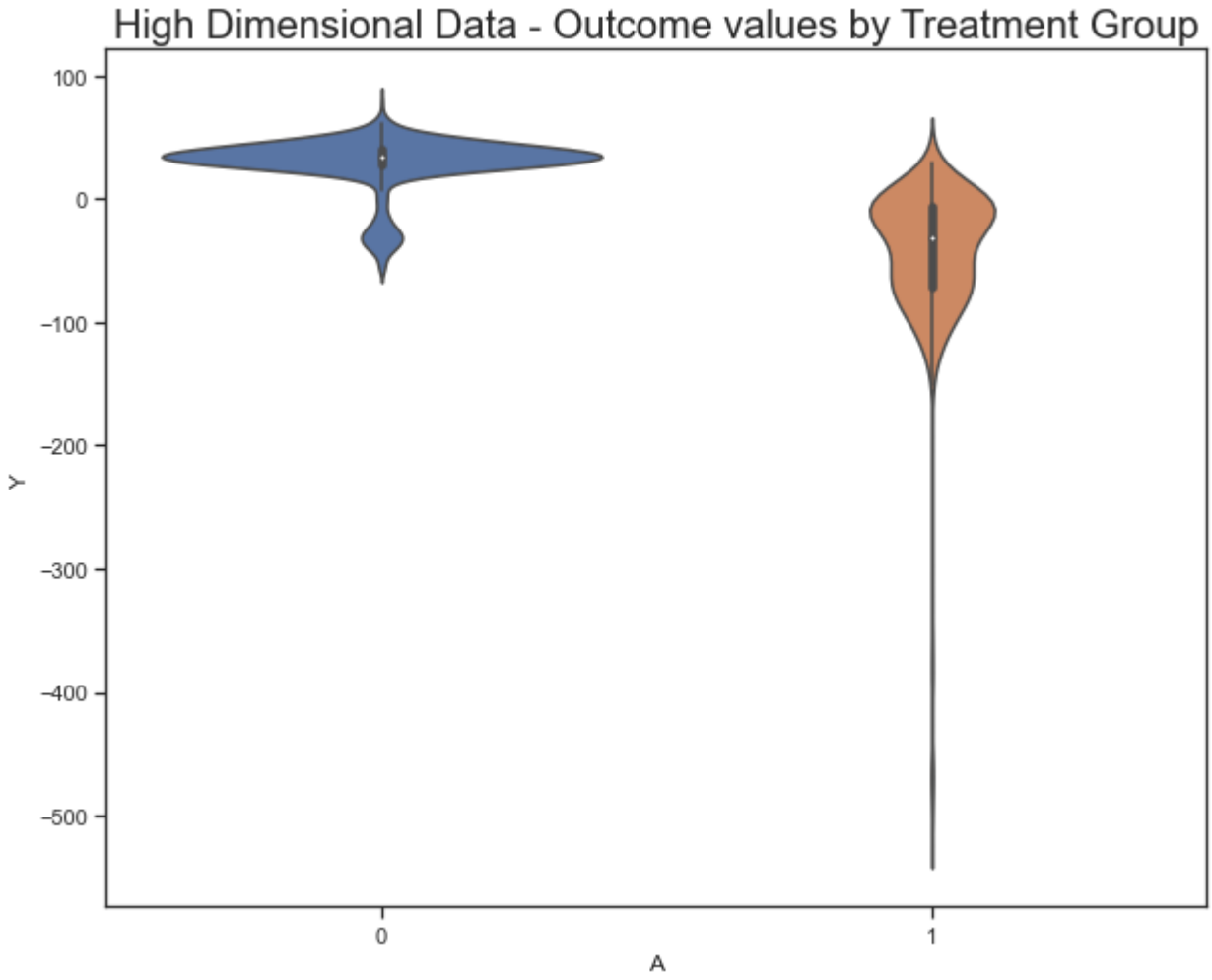
In [2]:
```
# load high dimensional data
highdim_data = pd.read_csv('../data/highDim_dataset.csv')

# load low dimensional data
lowdim_data = pd.read_csv('../data/lowDim_dataset.csv')
```

In [3]:
```
print("The high dimensional data has",highdim_data.shape[0],"observations and", highdim
print("The low dimensional data has",lowdim_data.shape[0],"observations and", lowdim_da
```

```
The high dimensional data has 2000 observations and 187 variables.
The low dimensional data has 500 observations and 24 variables.
```

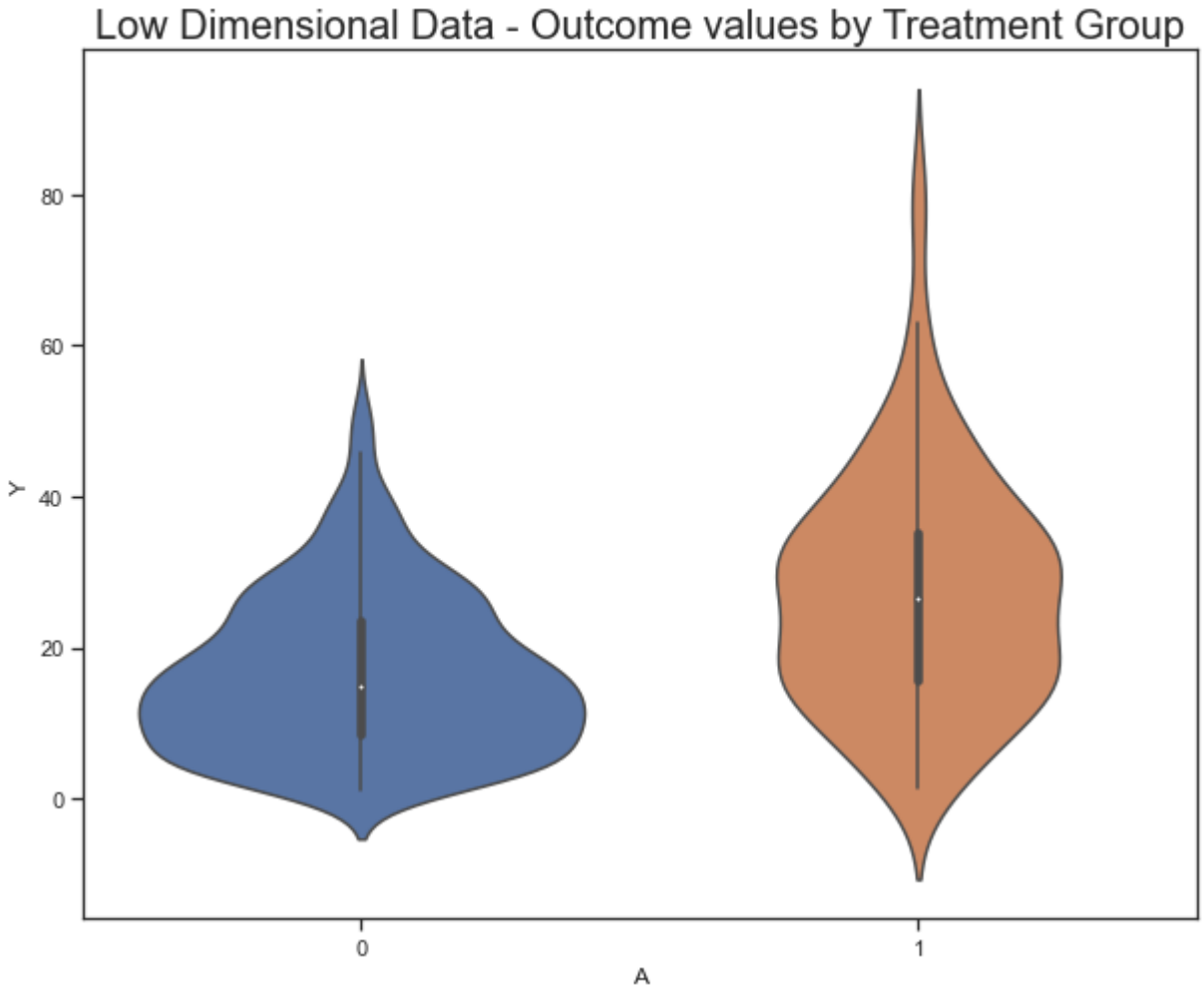
We use violin plots to visualize outcome values by treatment groups in both dataset to better understand the data. For the high dimensional data, we see that the outcome values of non-treated group mainly cluster between 0 to 50, but those of treated group below 0 and even have extreme values. For the low dimensional data, the outcome values are more evenly distributed between the treatment groups.

In [4]:
```
# visualize outcome values on high dimensional data by treatment group
sns.violinplot(x="A", y="Y", data=highdim_data)
plt.title('High Dimensional Data - Outcome values by Treatment Group', size=20)
plt.show()
```



In [5]:

```
# visualize outcome values on low dimensional data by treatment group
sns.violinplot(x="A", y="Y", data=lowdim_data)
plt.title('Low Dimensional Data - Outcome values by Treatment Group', size=20)
plt.show()
```



2.3 Scale data for regression model

We will scale the features in the original dataset and combine with the outcome variables to create scaled datasets for regression models. Feature scaling is crucial because it helps to normalize the range of all features for distance calculation.

In [6]:
```
# function to scale the datasets
def scaled_data(data):
    x = data.drop(['A','Y'], axis = 1)
    y = data[["A"]]

    data_columns = data.columns.drop(['Y','A'])

    x_scaled = StandardScaler().fit_transform(x)

    data_scaled = pd.DataFrame(x_scaled, index = data.index, columns = data_columns)

    data_scaled['A'] = data['A']
    data_scaled['Y'] = data['Y']

    display(data_scaled.head())

    return data_scaled
```

In [7]:
```
# scale the high dimentional dataset
```

```
highdim_scale_data = scaled_data(highdim_data)
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	
0	-1.015114	0.482748	-1.161393	0.303352	1.487812	-1.171070	-1.423520	1.686961	1.203321	0.38
1	-1.015114	-2.071474	-1.650640	-1.477143	-0.512424	-1.171070	0.204290	0.524392	1.203321	0.80
2	-1.015114	-2.071474	0.795598	-1.922267	-0.876103	-0.415004	-0.880917	0.669713	1.203321	-0.03
3	0.985111	-2.071474	-1.324475	-1.477143	-1.239782	-1.171070	-0.700049	0.698777	-0.831034	-0.45
4	0.985111	0.482748	-0.019815	0.971038	0.214934	0.492274	2.012968	0.756906	1.203321	0.38

5 rows × 187 columns

In [8]:
```
# scale the low dimentional dataset
lowdim_scale_data = scaled_data(lowdim_data)
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V
0	-0.502205	-0.352816	-0.257883	-0.266592	-0.34195	-0.465776	-0.266412	-0.649809	-0.35092	-0.159
1	-0.502205	-0.352816	-0.257883	-0.266592	-0.34195	-0.465776	-0.266412	1.034631	-0.35092	-0.159
2	-0.502205	-0.352816	-0.257883	-0.266592	-0.34195	-0.465776	-0.266412	-0.649809	-0.35092	-0.159
3	3.441468	-0.352816	-0.257883	-0.266592	-0.34195	-0.465776	-0.266412	-0.649809	-0.35092	-0.159
4	-0.253654	0.209949	-0.150877	-0.081459	-0.34195	0.445723	0.162041	0.493204	1.15826	-0.071

5 rows × 24 columns

3 Propensity Scores Estimation and Evaluation

Propensity score is the probability of assignment to the treatment group based on observed characteristics. It reduces each sample's set of covariates into a single score.

We use L1 penalized logistic regression to estimate propensity scores for both data sets. To ensure more accurate results, we first tune the optimal hyperparameters for logistic regression, then use the best parameter to estimate propensity scores.

We repeat the above steps for both the high dimensional and low dimensional datasets.

3.1 Create Propensity Score Estimation Functions

In [9]:
```
def best_param(data, random_state, param_grid, cv=10):
    '''
    Purpose: to find the best parameter "C" (coefficient of regularization strength) fo

    Parameters:
    data - dataset to best tested on
    random_state - set seed
```

```
    param_grid - set of parameter values to test on
    cv - number of folds for cross-validation

    '''

    x = data.drop(['A','Y'], axis = 1)
    y = data[['A']].values.ravel()

    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_st


    model_cv = GridSearchCV(LogisticRegression(penalty='l1',solver = 'liblinear'), para
    model_cv.fit(x_train, y_train)

    print("The best tuned coefficient of regularization strength is",model_cv.best_para
          "with a testing accuracy of", model_cv.score(x_test, y_test))

    return model_cv.best_params_.get('C')
```

In [10]:

```
def propensity_score(data, C=0.1, plot = True):
    '''
    Purpose: to estimate propensity score with L1 penalized logistic regression

    Parameters:
    data - dataset to estimate on
    C - coeficient of regularization strength
    plot - print out visualization to show distribution of propensity scores

    Returns:
    1. ps for Propensity Score
    2. Visualization plot to show distribution of propensity scores

    '''


    T = 'A'
    Y = 'Y'
    X = data.columns.drop([T,Y])

    ps_model = LogisticRegression(random_state=random_state, penalty='l1',
                                  solver='liblinear').fit(data[X], data[T])

    ps = ps_model.predict_proba(data[X])[:,1] # we are interested in the probability of

    if plot:
        df_plot = pd.DataFrame({'Treatment':data[T], 'Propensity Score':ps})

        sns.histplot(data=df_plot, x = "Propensity Score", hue = "Treatment", element =
        plt.title("Distribution of Propensity Score by Treatment Group", size=20)
        plt.show()

    return ps
```

In [11]:

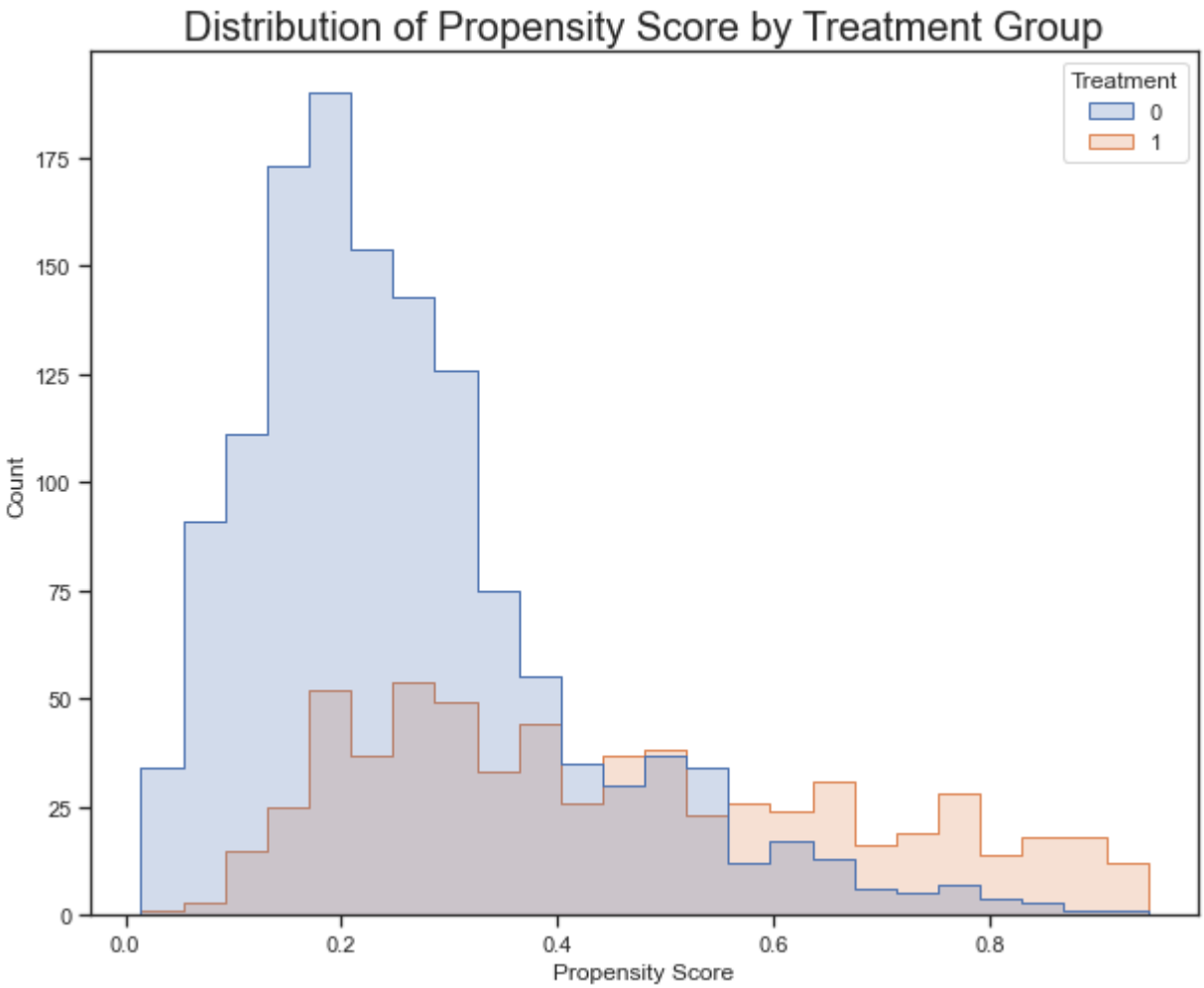
```
# setting parameters
param_grid = {"C":[0.01,0.05,0.1,0.3,0.5,0.7,1]}
```

3.2 Evaluate Propensity Scores for High Dimensional Data

In [12]:
```
# use 10-fold cross-validation to tune for the best parameter for logistic regression
c_high = best_param(highdim_scale_data, random_state=random_state, param_grid=param_gri
```

```
The best tuned coefficient of regularization strength is 0.05 with a testing accuracy of
0.716
```

In [13]:
```
# estimate propsensity scores
ps_high = propensity_score(highdim_scale_data, C = c_high)
```

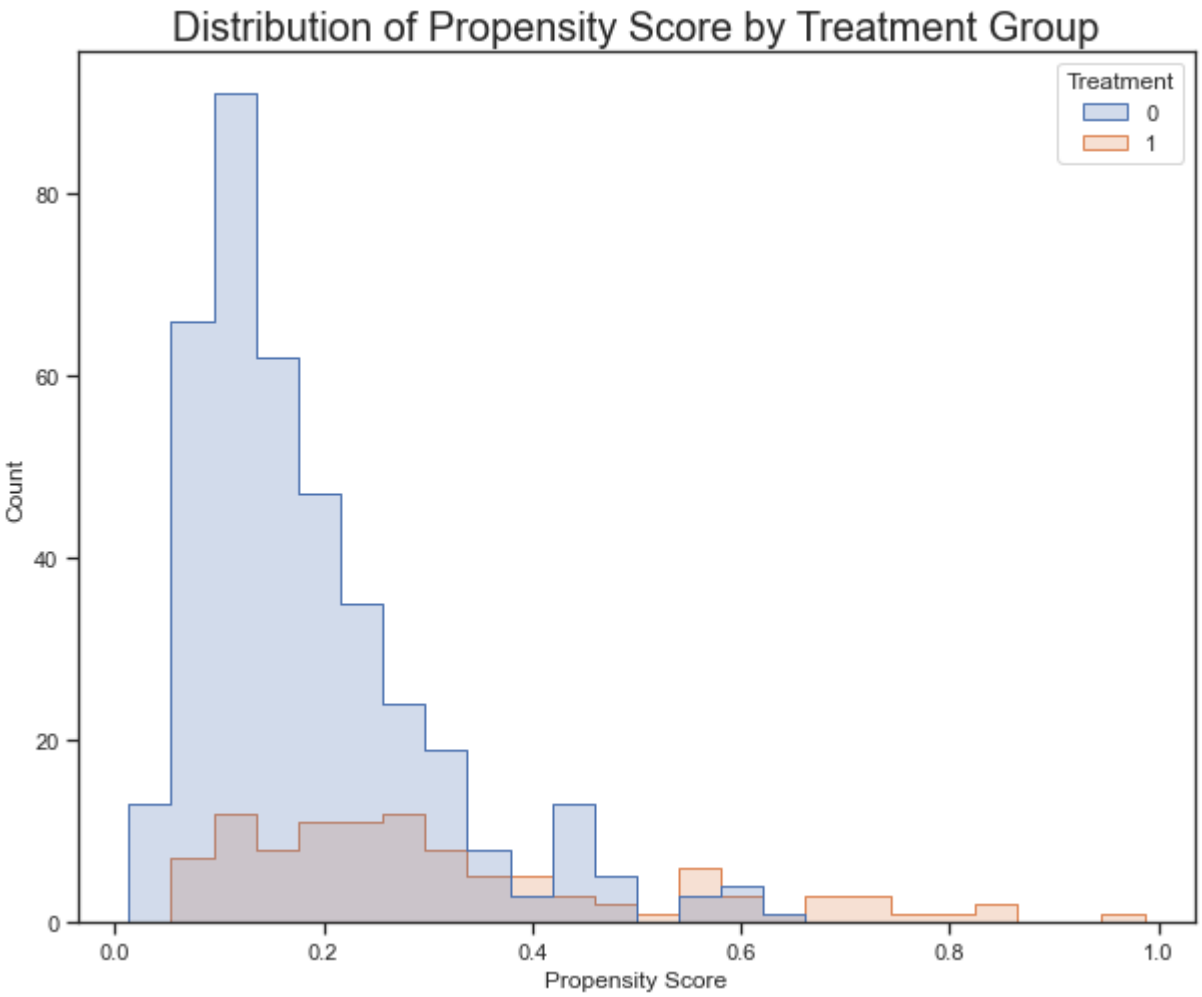


3.3 Evaluate Propensity Scores for Low Dimensional Data

In [14]:
```
# use 10-fold cross-validation to tune for the best parameter for logistic regression
c_low = best_param(lowdim_scale_data, random_state=random_state, param_grid=param_grid)
```

```
The best tuned coefficient of regularization strength is 0.3 with a testing accuracy of
0.792
```

In [15]:
```
ps_low = propensity_score(lowdim_scale_data, C = c_low)
```



The estimated propensity scores for both the high and low dimensional datasets are skewed to the right for the non-treatment group and spread out for the treatment group. This will create some challenges for propensity score matching.

4. Algorithms and Evaluation

In [16]:
```
# set true ATE
true_low = 2.0901
true_high = -54.8558
```

4.1 Propensity Scores Matching (Full)

Full matching creates a series of matched sets in an optimal way so that each matched set contains at least one treated individual and at least one control individual.

In [17]:
```
# append propensity score to original data
highdim_data['PS']=ps_high
lowdim_data['PS']=ps_low
```

In [18]:
```
# create function to calculate PSM
def PSM(treated_df, control_df):
```

```
    #Get Distances
    treated_df.loc[:,'group']=None
    treated_df.loc[:,'control_Y']=None
    treated_df.loc[:,'D']=None
    for i in range(len(treated_df)):
        temp_d=[]
        for j in range(len(control_df)):
            temp_d.append(abs(treated_df.loc[i,'PS']-control_df.loc[j,'PS']))
        index=temp_d.index(np.min(temp_d))
        treated_df.loc[i,'control_Y']=control_df.loc[index,'Y']
        treated_df.loc[i,'D']=np.min(temp_d)

    #Split class
    #Here, we deide to create 5 subclasses
    r=(max(treated_df.loc[:,'D'])-min(treated_df.loc[:,'D']))/5
    for i in range(len(treated_df)):
        if treated_df.loc[i,'D'] <= min(treated_df.loc[:,'D'])+r:
            treated_df.loc[i,'group']=0
        elif treated_df.loc[i,'D'] > min(treated_df.loc[:,'D'])+r and treated_df.loc[i,
            treated_df.loc[i,'group']=1
        elif treated_df.loc[i,'D'] > min(treated_df.loc[:,'D'])+2*r and treated_df.loc[
            treated_df.loc[i,'group']=2
        elif treated_df.loc[i,'D'] > min(treated_df.loc[:,'D'])+3*r and treated_df.loc[
            treated_df.loc[i,'group']=3
        else:
            treated_df.loc[i,'group']=4

    #Calculate ATE
    TE=[]
    for i in range(5):
        temp=treated_df[treated_df.loc[:,'group']==i]
        a=(temp.loc[:,'Y']-temp.loc[:,'control_Y']).mean()*len(temp)/len(treated_df)
        TE.append(a)
    ATE=np.nanmean(TE)

    return ATE
```

Low dimensional data

In [19]:

```
PS_low_start = time.time()
treated_low=lowdim_data[lowdim_data.A==1]
treated_low.reset_index(drop=True, inplace=True)
control_low=lowdim_data[lowdim_data.A==0]
control_low.reset_index(drop=True, inplace=True)
```

In [20]:

```
PS_low_ATE = PSM(treated_low, control_low)
PS_low_end = time.time()
PS_low_accu = (1 - (abs(PS_low_ATE - true_low)/abs(true_low)))*100
PS_low_time = PS_low_end - PS_low_start
```

In [21]:

```
# display results
PS_low_result = pd.Series(data = ['PSM', 'Low', PS_low_time, PS_low_ATE, PS_low_accu],
          index = ['Method','Data Type','Run Time','ATE','Accuracy'])

print(f'PSM method for low dimensional dataset:\n ATE = {PS_low_ATE:0.2f}\n Accuracy =
```

```
PSM method for low dimensional dataset:
 ATE = 0.36
 Accuracy = 17.43
 PSM running time = 1.06
```

High dimensional data

In [22]:
```
PS_high_start = time.time()
treated_high=highdim_data[highdim_data.A==1]
treated_high.reset_index(drop=True, inplace=True)
control_high=highdim_data[highdim_data.A==0]
control_high.reset_index(drop=True, inplace=True)
```

In [23]:
```
PS_high_ATE = PSM(treated_high, control_high)
PS_high_end = time.time()
PS_high_accu = (1 - (abs(PS_high_ATE - true_high)/abs(true_high)))*100
PS_high_time = PS_high_end - PS_high_start
```

In [24]:
```
# display results
PS_high_result = pd.Series(data = ['PSM', 'High', PS_high_time, PS_high_ATE, PS_high_ac
          index = ['Method','Data Type','Run Time','ATE', 'Accuracy'])

print(f'PSM method for high dimensional dataset:\n ATE = {PS_high_ATE:0.2f}\n Accuracy
```

```
PSM method for high dimensional dataset:
 ATE = -11.71
 Accuracy = 21.35
 PSM running time = 18.57
```

4.2 Doubly Robust Estimations

Doubly Robust estimation combines outcome regression model with weighting by propensity score model. Doubly robust estimation remains consistent even if either the outcome model or the propensity model is incorrect.

Low Dimensional Data

In [25]:
```
# reload data, add propensity score column and divide data into treat and control group
lowdim_data = pd.read_csv('../data/lowDim_dataset.csv')
lowdim_data_new = pd.read_csv('../data/lowDim_dataset.csv')
lowdim_data_new['PS_low'] = pd.Series(ps_low, index=lowdim_data_new.index)
lowdim_treat = lowdim_data[lowdim_data['A'] == 1].reset_index(drop = True)
lowdim_control = lowdim_data[lowdim_data['A'] == 0].reset_index(drop = True)
```

In [26]:
```
# fit regression model to treat and control group
xlow_treat = lowdim_treat.drop(['A','Y'],axis=1)
ylow_treat = lowdim_treat['Y']
lr_low_treat = LinearRegression().fit(xlow_treat, ylow_treat)

xlow_control = lowdim_control.drop(['A','Y'],axis=1)
ylow_control = lowdim_control['Y']
lr_low_control = LinearRegression().fit(xlow_control, ylow_control)
```

In [27]:
```
# make prediction based on trained models and construct a full dataset
xlow = lowdim_data_new.drop(['A','Y','PS_low'],axis=1)
lowdim_data_new['mtreat'] = lr_low_treat.predict(xlow)
lowdim_data_new['mcontrol'] = lr_low_control.predict(xlow)
```

In [28]:
```
# perform Doubly Robust Estimation algorithm
DR_low_start = time.time()

DR_low_1 = 0
DR_low_0 = 0

for i in range(len(lowdim_data_new)):
    DR_low_1 = DR_low_1 + (lowdim_data_new['A'][i] * lowdim_data_new['Y'][i] - (lowdim_
    DR_low_0 = DR_low_0 + ((1-lowdim_data_new['A'][i])* lowdim_data_new['Y'][i] + (lowd

DR_low_ATE = (DR_low_1 - DR_low_0)/len(lowdim_data_new)
DR_low_accu = (1 - abs((DR_low_ATE -true_low)/true_low))*100
DR_low_end = time.time()
DR_low_time = DR_low_end - DR_low_start
```

In [29]:
```
# print the ATE, accuracy and algorithm running time result
DR_low_result = pd.Series(data = ['Doubly Robust', 'Low', DR_low_time, DR_low_ATE, DR_l
          index = ['Method','Data Type','Run Time','ATE', 'Accuracy'])

print(f'Doubly robust estimation method for low dimensional dataset:\n ATE = {DR_low_AT
```

```
Doubly robust estimation method for low dimensional dataset:
 ATE = 2.09
 Accuracy = 99.76
 DR running time = 0.08
```

High dimensional data

In [30]:
```
# reload data, add propensity score column and divide data into treat and control group
highdim_data = pd.read_csv('../data/highDim_dataset.csv')
highdim_data_new = pd.read_csv('../data/highDim_dataset.csv')
highdim_data_new['PS_high'] = pd.Series(ps_high, index=highdim_data.index)
highdim_treat = highdim_data[highdim_data.A == 1].reset_index(drop = True)
highdim_control = highdim_data[highdim_data.A == 0].reset_index(drop = True)
```

In [31]:
```
# fit regression model to treat and control group
xhigh_treat = highdim_treat.drop(['A','Y'],axis=1)
yhigh_treat = highdim_treat['Y']
lr_high_treat = LinearRegression().fit(xhigh_treat, yhigh_treat)

xhigh_control = highdim_control.drop(['A','Y'],axis=1)
yhigh_control = highdim_control['Y']
lr_high_control = LinearRegression().fit(xhigh_control, yhigh_control)
```

In [32]:
```
# make prediction based on trained models and construct a full dataset
xhigh = highdim_data_new.drop(['A','Y','PS_high'],axis=1)
highdim_data_new['mtreat'] = lr_high_treat.predict(xhigh)
highdim_data_new['mcontrol'] = lr_high_control.predict(xhigh)
```

In [33]:
```
# perform Doubly Robust Estimation algorithm
DR_high_start = time.time()

DR_high_1 = 0
DR_high_0 = 0

for i in range(len(highdim_data_new)):
    DR_high_1 = DR_high_1 + (highdim_data_new['A'][i] * highdim_data_new['Y'][i] - (hig
    DR_high_0 = DR_high_0 + ((1-highdim_data_new['A'][i])* highdim_data_new['Y'][i] + (

DR_high_ATE = (DR_high_1 - DR_high_0)/len(highdim_data_new)
DR_high_accu = (1 - abs((DR_high_ATE - true_high)/true_high))*100
DR_high_end = time.time()
DR_high_time = DR_high_end - DR_high_start
```

In [34]:
```
# save results and print the ATE, accuracy and algorithm running time result

DR_high_result = pd.Series(data = ['Doubly Robust', 'High', DR_high_time, DR_high_ATE,
          index = ['Method','Data Type','Run Time','ATE', 'Accuracy'])

print(f'Doubly robust estimation method for high dimensional dataset:\n ATE = {DR_high_
```

```
Doubly robust estimation method for high dimensional dataset:
 ATE = -57.04
 Accuracy = 96.02
 DR running time = 0.24
```

4.3 Stratification

We will rank and stratify 5 mutually exclusive subsets based on the propensity scores. Within each stratum, subjects have roughly similar values of the propensity scores.

In [35]:
```
## Stratification

def stratification(data, prop):
    start = time.time()
    K = 5 # k, quintiles is reccomended
   # N = len(df.index)
    strata = [1,2,3,4,5]
    ATE = 0

    #split propensity scores into thier respective quintiles
    prop_split = pd.qcut(prop, K)
    prop_split.categories = strata


    #label the dataset and group accordingly
    quintiles = copy.copy(data)
    quintiles["strata"] = prop_split
    quintiles = quintiles[["A", "strata", "Y"]]

    #calucate the average Y
    quintiles = quintiles.groupby(["A", "strata"]).mean()

    for num in strata:
        ATE += quintiles.loc[pd.IndexSlice[(1, num)]] - quintiles.loc[pd.IndexSlice[(0,

    #Divide by k
```

```
    ATE = ATE/K

    end = time.time()

    print("Estamated ATE: ", round(ATE.values[0], 2))
    print("Runtime: ", end-start)

    return(ATE, end-start)
```

Low Dimensional Data

In [36]:
```
print("low")

S_low = stratification(lowdim_data, ps_low)

S_low_ATE = S_low[0]
S_low_ATE = S_low_ATE.values[0]
S_low_time = S_low[1]
S_low_accu = (1 - (abs(S_low_ATE - true_low)/abs(true_low)))*100
```

```
low
Estamated ATE:  2.38
Runtime:  0.0349123477935791
```

In [37]:
```
# save results and print the ATE, accuracy and algorithm running time result

S_low_result = pd.Series(data = ['Stratification', 'Low', S_low_time, S_low_ATE, S_low_
          index = ['Method','Data Type','Run Time','ATE', 'Accuracy'])

print(f'Stratification estimation method for low dimensional dataset:\n ATE = {S_low_AT
```

```
Stratification estimation method for low dimensional dataset:
 ATE = 2.38
 Accuracy = 85.97
 DR running time = 0.03
```

High Dimensional Data

In [38]:
```
print("high")

S_high = stratification(highdim_data, ps_high)

S_high_ATE = S_high[0]
S_high_ATE = S_high_ATE.values[0]
S_high_time = S_high[1]
S_high_accu = (1 - (abs(S_high_ATE - true_high)/abs(true_high)))*100
```

```
high
Estamated ATE:  -59.83
Runtime:  0.021941423416137695
```

In [39]:
```
# save results and print the ATE, accuracy and algorithm running time result

S_high_result = pd.Series(data = ['Stratification', 'High', S_high_time, S_high_ATE, S_
          index = ['Method','Data Type','Run Time','ATE', 'Accuracy'])

print(f'Stratification estimation method for high dimensional dataset:\n ATE = {S_high_
```

```
Stratification estimation method for high dimensional dataset:
 ATE = -59.83
 Accuracy = 90.94
 DR running time = 0.02
```

5. Model Comparison and Conclusion

Propensity scores have been estimated using L1 penalized logistic regression. For algorithm testing, use ps_low for the low dimensional data and ps_high for the high dimensional data.

In [40]:
```
# store all final results into dataframe
result_table = pd.DataFrame([PS_low_result, DR_low_result, S_low_result, PS_high_result
result_table = result_table.round(2)
```

In [41]:
```
# display results
result_table
```

Out[41]:

	Method	Data Type	Run Time	ATE	Accuracy
0	PSM	Low	1.06	0.36	17.43
1	Doubly Robust	Low	0.08	2.09	99.76
2	Stratification	Low	0.03	2.38	85.97
3	PSM	High	18.57	-11.71	21.35
4	Doubly Robust	High	0.24	-57.04	96.02
5	Stratification	High	0.02	-59.83	90.94

In summary, Propensity Score Matching has the longest run time and lowest accuracy of all three methods. This is due to the fact that the method goes through each sample and match the treated and non-treated unit based on propensity score, and yet the distributions of propensity scores for the non-treated group are skewed, resulting in some unmatched samples from the treated group.

On the other hand, stratification provides the least run time on all three models with a relatively high accuracy. The stratification method groups subjects with similar propensity scores into mutually exclusive stratum. This method is powerful in our scenario because it reduces the impact of the extremely unbalanced distribution of propensity scores between the treated and non-treated group by creating relatively more balanced subgroups.

Our best model is Doubly Robust Estimation, which returns an almost 100% accuracy on the low dimensional dataset and 96% accuracy on the high dimensional dataset. Doubly Robust estimation provides a simple way of combining linear regression with the propensity score to produce a doubly robust estimator, requiring only one of the models to be correct to identify the causal effect. Proven by our results, this algorithm outperforms the other models with its strong consistency and accuracy in prediction.