consolidated_models

December 2, 2020

This Notebook contains the code of all of the methods we used for this project.

The models are ordered in the following way:

1. Regression Estimate
2. Doubly Robust Estimation
3. Propensity Matching with Linear Propensity Score

Each model contains two analysis for both High and Low Dimension data.

At the end of this Notebook the reader will find a comparison table for the methods.

[26]:
```
# importing packages used in this Notebook

import pandas as pd
import numpy as np
import time
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
import statsmodels.api as sm
```

[9]:
```
# real ATE are given:

real_low = 2.5
real_high = -3
```

[117]:
```
# loading data

low_dim = pd.read_csv('data/lowDim_dataset.csv')
high_dim = pd.read_csv('data/highDim_dataset.csv')

# inspecting data

low_dim.isna().sum().sum(),high_dim.isna().sum().sum(),low_dim.shape, high_dim.
 ↪shape
```

1 Regression Estimate

1.1 Low Dimension

[43]:
```
# starting to measure run time for low dimension

start_time_low = time.time()

# deviding the data into treated and control groups

low_dim_treated = low_dim[low_dim['A'] == 1]
low_dim_treated = low_dim_treated.reset_index(drop = True)

low_dim_control = low_dim[low_dim['A'] == 0]
low_dim_control = low_dim_control.reset_index(drop = True)
```

[44]:
```
# running a regression for the treated group:

lr = LinearRegression()
X, y = low_dim_treated.iloc[:,2:], low_dim_treated.iloc[:,0]
lr.fit(X, y)
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
coef_treated_low = lr.coef_
intercept_treated_low =lr.intercept_
```

[45]:
```
# running a regression for the control group:

lr = LinearRegression()
X, y = low_dim_control.iloc[:,2:], low_dim_control.iloc[:,0]
lr.fit(X, y)
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
coef_control_low = lr.coef_
intercept_control_low =lr.intercept_
```

[46]:
```
# calculating fitted y's for both treated and control groups

fitted_y_treated_low = low_dim.iloc[:,2:].transpose().
↪multiply(coef_treated_low, axis =0).sum() + intercept_treated_low
fitted_y_control_low = low_dim.iloc[:,2:].transpose().
↪multiply(coef_control_low, axis =0).sum() + intercept_control_low
```

[47]:
```
# calculating the difference b.w the treatment and control group

ate_low = (fitted_y_treated_low - fitted_y_control_low).mean()
# measuring accuracy:
```

```
accuracy_low = (ate_low - real_low)/real_low
# stopping the clock:

run_time_low = time.time() - start_time_low
```

1.2 High Dimension

[48]:
```
# starting to measure run time for low dimension

start_time_high = time.time()

# deviding the data into treated and control groups

high_dim_treated = high_dim[high_dim['A'] == 1]
high_dim_treated = high_dim_treated.reset_index(drop = True)

high_dim_control = high_dim[high_dim['A'] == 0]
high_dim_control = high_dim_control.reset_index(drop = True)
```

[49]:
```
# running a regression for the treated group:

lr = LinearRegression()
X, y = high_dim_treated.iloc[:,2:], high_dim_treated.iloc[:,0]
lr.fit(X, y)
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
coef_treated_high = lr.coef_
intercept_treated_high =lr.intercept_
```

[50]:
```
# running a regression for the control group:

lr = LinearRegression()
X, y = high_dim_control.iloc[:,2:], high_dim_control.iloc[:,0]
lr.fit(X, y)
LinearRegression(copy_X=True, fit_intercept=True, normalize=False)
coef_control_high = lr.coef_
intercept_control_high =lr.intercept_
```

[51]:
```
# calculating fitted y's for both treated and control groups

fitted_y_treated_high = high_dim.iloc[:,2:].transpose().
 ↪multiply(coef_treated_high, axis =0).sum() + intercept_treated_high
fitted_y_control_high = high_dim.iloc[:,2:].transpose().
 ↪multiply(coef_control_high, axis =0).sum() + intercept_control_high
```

[52]:
```python
# calculating the difference b.w the treatment and control group

ate_high = (fitted_y_treated_high - fitted_y_control_high).mean()
# measuring accuracy

accuracy_high = (ate_high - real_high)/real_high
# stopping clock

run_time_high = time.time() - start_time_high
```

1.3 Results

[53]:
```python
# building a df for the results:

regression_estimate_low = pd.Series(data = [run_time_low, ate_low,␣
 ↪accuracy_low],
                                   index = ['run_time','ate','accuracy']).
 ↪rename('low')
regression_estimate_high = pd.Series(data = [run_time_high, ate_high,␣
 ↪accuracy_high],
                                    index = ['run_time','ate', 'accuracy']).
 ↪rename('high')
results_regression_estimate = pd.DataFrame([regression_estimate_low,␣
 ↪regression_estimate_high]).round(3)
results_regression_estimate = pd.concat({'regression_estimate':␣
 ↪results_regression_estimate}, names=['method'])
results_regression_estimate
```

[53]:
```
                          run_time    ate  accuracy
method
regression_estimate low      6.618  2.532     0.013
                    high     4.319 -2.951    -0.016
```

2 Doubly Robust Estimation

2.1 Low Dimension

[118]:
```python
# Create X and Y variables from origianal datasets for L1 Logistic Regression
def logit_dataset (df):
    #all the covariates as X
    X = df.drop(['A','Y'], axis = 1)
    y = df[['A']]
    return X, y
```

```
#Scaler training features for regression model
def std_feature (x_train, x_test):
    sc = StandardScaler()
    x_train_scaled = sc.fit_transform(x_train)
    x_test_scaled = sc.transform(x_test)
    return x_train_scaled,  x_test_scaled
```

[119]:
```
X_low = logit_dataset(low_dim)[0]
y_low = logit_dataset(low_dim)[1]

# Split data into training and testing sets
X_train_low, X_test_low, y_train_low, y_test_low = train_test_split(X_low,␣
↪y_low, test_size=0.25, random_state=0)

# Standardize features
X_train_low_std =std_feature(X_train_low,X_test_low)[0]
X_test_low_std = std_feature(X_train_low,X_test_low)[1]
```

[120]:
```
param_grid = {'C': [1, 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.5, 0.4, 0.3, 0.2, 0.
↪25,0.1]}
clf = GridSearchCV(LogisticRegression(penalty='l1'), param_grid)
clf=LogisticRegression(penalty='l1',solver = 'liblinear')
clf_cv=GridSearchCV(clf,param_grid,cv=5)
clf_cv.fit(X_train_low_std, y_train_low.values.ravel())
```

[120]:
```
GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                          fit_intercept=True,
                                          intercept_scaling=1, l1_ratio=None,
                                          max_iter=100, multi_class='warn',
                                          n_jobs=None, penalty='l1',
                                          random_state=None, solver='liblinear',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='warn', n_jobs=None,
             param_grid={'C': [1, 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.5, 0.4,
                               0.3, 0.2, 0.25, 0.1]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

[121]:
```
start_time_l1_low = time.time()
# Best: C = 0.1
clf_low = LogisticRegression(penalty='l1', C = 0.1, solver = 'liblinear')
#Calculate propensity scores
clf_low.fit(X_low, y_low.values.ravel())
ps_low=clf_low.predict_proba(X_low)[:, 1]
```

[122]:
```
full_low_dim = low_dim.copy()
full_low_dim['PS']=pd.Series(ps_low, index=full_low_dim.index)
```

[123]:
```
# deviding the low dimensional data into treated and control groups

lowDim_treated = low_dim[low_dim['A'] == 1]
lowDim_treated = lowDim_treated.reset_index(drop = True)

lowDim_control = low_dim[low_dim['A'] == 0]
lowDim_control = lowDim_control.reset_index(drop = True)

#Fit a regression model to get the estimation of y given T=1 and X
X1_low_treated = lowDim_treated.drop(['Y'], axis = 1)
y_low_treated = lowDim_treated['Y']
lr_low_treated = LinearRegression().fit(X1_low_treated, y_low_treated)

# Fit a regression model to get the estimation of y given T=0and X
X1_low_control = lowDim_control.drop(['Y'], axis = 1)
y_low_control = lowDim_control['Y']
lr_low_control = LinearRegression().fit(X1_low_control, y_low_control)


# Select all covariates and 'A' columns from full dataset
X_low_new = full_low_dim.drop(['Y','PS'], axis = 1)
m1_low= lr_low_treated.predict(X_low_new)
m0_low= lr_low_control.predict(X_low_new)
# join m1 and m0 to full_low_dim
full_low_dim['m1'] = pd.Series(m1_low, index = full_low_dim.index)
full_low_dim['m0'] = pd.Series(m0_low, index = full_low_dim.index)
```

[124]:
```
def DRE(full_data):

    n = len(full_data.index)
    result1 = 0
    result2 = 0

    for i in range(n):
        result1 = result1 + (full_data['A'][i] * full_data['Y'][i] -␣
↪(full_data['A'][i] - full_data['PS'][i])*full_data['m1'][i])/
↪full_data['PS'][i]
        result2 = result2 + ((1-full_data['A'][i])* full_data['Y'][i] -␣
↪(full_data['A'][i] - full_data['PS'][i])*full_data['m0'][i])/
↪(1-full_data['PS'][i])

    ETA = 1/n*(result1-result2)

    return ETA
```

[125]:
```
start_time_dre_low = time.time()
ate_dre_low=DRE(full_low_dim)
accuracy_dre_low = (ate_dre_low - real_low)/real_low
run_time_l1_low = time.time() - start_time_l1_low
run_time_dre_low = time.time() - start_time_dre_low
```

2.2 High Dimension

[126]:
```
X_high = logit_dataset(high_dim)[0]
y_high = logit_dataset(high_dim)[1]

X_train_high, X_test_high, y_train_high, y_test_high = train_test_split(X_high,␣
↪y_high, test_size=0.25, random_state=0)

# Standardize features
X_train_high_std =std_feature(X_train_high,X_test_high)[0]
X_test_high_std = std_feature(X_train_high,X_test_high)[1]
```

[127]:
```
# L1 penalized logistic regression for propencity scores

param_grid = {'C': [.06, .05, .04, .03, .02, .01, 0.008, 0.005, 0.001]}
clf = GridSearchCV(LogisticRegression(penalty='l1'), param_grid)
clf=LogisticRegression(penalty='l1',solver = 'liblinear')
clf_cv=GridSearchCV(clf,param_grid,cv=5)
clf_cv.fit(X_train_high_std, y_train_high.values.ravel())

# Best: C = 0.04
start_time_l1_high = time.time()
clf_high = LogisticRegression(penalty='l1', C = 0.04, solver = 'liblinear')
#Calculate propensity scores
clf_high.fit(X_high, y_high.values.ravel())
ps_high=clf_high.predict_proba(X_high)[:, 1]
```

[128]:
```
# Doubly Robust Estimation Algorithm to calculate ATE

full_high_dim= high_dim.copy()
full_high_dim['PS']=pd.Series(ps_high, index=full_high_dim.index)

# deviding the high dimensional data into treated and control groups
highDim_treated = high_dim[high_dim['A'] == 1]
highDim_treated = highDim_treated.reset_index(drop = True)

highDim_control = high_dim[high_dim['A'] == 0]
highDim_control = highDim_control.reset_index(drop = True)
```

```python
# Fit a regression model to get the estimation of y given T=1 and X
X1_high_treated = highDim_treated.drop(['Y'], axis = 1)
y_high_treated = highDim_treated['Y']
lr_high_treated = LinearRegression().fit(X1_high_treated, y_high_treated)


# Fit a regression model to get the estimation of y given T=0 and X
X1_high_control = highDim_control.drop(['Y'], axis = 1)
y_high_control = highDim_control['Y']
lr_high_control = LinearRegression().fit(X1_high_control, y_high_control)

#Add m1 and m0 to dataset
X_high_new = full_high_dim.drop(['Y','PS'], axis = 1)
m1_high= lr_high_treated.predict(X_high_new)
m0_high= lr_high_control.predict(X_high_new)
full_high_dim['m1'] = pd.Series(m1_high, index = full_high_dim.index)
full_high_dim['m0'] = pd.Series(m0_high, index = full_high_dim.index)
```

[129]:
```python
start_time_dre_high = time.time()
ate_dre_high=DRE(full_high_dim)
accuracy_dre_high= (ate_dre_high - real_high)/real_high
run_time_l1_high = time.time() - start_time_l1_high
run_time_dre_high = time.time() - start_time_dre_high
```

2.3 Results

[130]:
```python
dre_low = pd.Series(data = [run_time_dre_low, ate_dre_low, accuracy_dre_low],
                                  index = ['run_time','ate','accuracy']).
 ↪rename('low')

dre_high = pd.Series(data = [run_time_dre_high, ate_dre_high,␣
 ↪accuracy_dre_high],
                                    index = ['run_time','ate', 'accuracy']).
 ↪rename('high')
results_dre = pd.DataFrame([dre_low, dre_high]).round(3)
results_dre = pd.concat({'DRE': results_dre}, names=['method'])
results_dre
```

[130]:

method		run_time	ate	accuracy
DRE	low	0.115	2.645	0.058
	high	0.438	-3.082	0.027

3 PSM

[34]:
```
X_high = high_dim.drop(['A','Y'], axis = 1)
X_low = low_dim.drop(['A','Y'], axis = 1)

y_high = high_dim[['A']]
y_low = low_dim[['A']]
```

[35]:
```
# Choosing the best parameter to calculate propensity score

def best_para(data, C):
    X=data.drop(['A','Y'], axis = 1)
    y=data[['A']]
    diff=[]
    for c in C:
        clf = LogisticRegression(penalty='l1', C = c, solver = 'liblinear')
        clf.fit(X, y.values.ravel())
        ps_logit=clf.predict_log_proba(X)[:, 1]
        data['log_ps']=ps_logit
        treated=data[data['A']==1]
        control=data[data['A']==0]
        di=max(treated['log_ps'])-min(treated['log_ps'])
        dj=max(control['log_ps'])-min(control['log_ps'])
        diff.append(abs(di-dj))
    best_ind=diff.index(min(diff))
    best_c=C[best_ind]
    best_diff=diff[best_ind]
    return best_c, best_diff
```

[36]:
```
# Propensity score matching model

def PSM(treated_df, control_df):

    treated_df.loc[:,'group']=None
    treated_df.loc[:,'control_Y']=None

    diff_i=[]
    for i in range(len(treated_df)):
        diff_j=[]
        for j in range(len(control_df)):
            diff_j.append(abs(control_df.loc[j,'log_ps']-treated_df.
 ↪loc[i,'log_ps']))

        ind_j=diff_j.index(min(diff_j))
        treated_df.loc[i,'control_Y']=control_df.loc[ind_j,'Y']
        diff_i.append(min(diff_j))
```

```
    r=(max(diff_i)-min(diff_i))/5
    for i in range(len(treated_df)):
        if diff_i[i] < min(diff_i)+r:
            treated_df.loc[i,'group']=1
        elif diff_i[i] >= min(diff_i)+r and diff_i[i] < min(diff_i)+r*2:
            treated_df.loc[i,'group']=2
        elif diff_i[i] >= min(diff_i)+r*2 and diff_i[i] < min(diff_i)+r*3:
            treated_df.loc[i,'group']=3
        elif diff_i[i] >= min(diff_i)+r*3 and diff_i[i] < min(diff_i)+r*4:
            treated_df.loc[i,'group']=4
        else:
            treated_df.loc[i,'group']=5

    ATE=0
    for k in range(treated_df.loc[:,'group'].max()):
        group=treated_df[treated_df.loc[:,'group']==k+1]
        if len(group)!=0:
            ATE=ATE+(group.loc[:,'Y']-group.loc[:,'control_Y']).
 ↪mean()*len(group)/len(treated_df)

    return ATE
```

3.1 Low Dimension

[37]:
```
start_low = time.time()
clf_low = LogisticRegression(penalty='l1', C = 0.2, solver = 'liblinear')
clf_low.fit(X_low, y_low.values.ravel())
ps_logit_low = clf_low.predict_log_proba(X_low)[:, 1]
low_dim['log_ps']=ps_logit_low
```

[38]:
```
treated_low = low_dim[low_dim['A']==1]
treated_low = treated_low.reset_index(drop = True)
control_low = low_dim[low_dim['A']==0]
control_low = control_low.reset_index(drop = True)

ate_psm_low = PSM(treated_low, control_low)
run_time_psm_low = time.time()-start_low
accuracy_psm_low = (ate_psm_low - real_low)/real_low
```

3.2 High Dimension

[39]:
```
start_high = time.time()
clf_high = LogisticRegression(penalty='l1', C = 0.5, solver = 'liblinear')
clf_high.fit(X_high, y_high.values.ravel())
```

```
ps_logit_high = clf_high.predict_log_proba(X_high)[:, 1]
high_dim['log_ps']=ps_logit_high
```

[40]:
```
treated_high = high_dim[high_dim['A']==1]
treated_high = treated_high.reset_index(drop = True)
control_high = high_dim[high_dim['A']==0]
control_high = control_high.reset_index(drop = True)

ate_psm_high = PSM(treated_high, control_high)
accuracy_psm_high = (ate_psm_high - real_high)/real_high
run_time_psm_high = time.time()-start_high
```

3.3 Results

[41]:
```
# building a df for psm results:

psm_low = pd.Series(data = [run_time_psm_low, ate_psm_low, accuracy_psm_low],
                                   index = ['run_time','ate','accuracy']).
 ↪rename('low')

psm_high = pd.Series(data = [run_time_psm_high, ate_psm_high,␣
 ↪accuracy_psm_high],
                                     index = ['run_time','ate', 'accuracy']).
 ↪rename('high')
results_psm = pd.DataFrame([psm_low, psm_high]).round(3)
results_psm = pd.concat({'psm': results_psm}, names=['method'])

results_psm
```

[41]:

method		run_time	ate	accuracy
psm	low	3.323	2.586	0.035
	high	23.656	-3.069	0.023

4 Methods Comparison

[131]:
```
results_final = pd.concat([results_regression_estimate, results_dre,␣
 ↪results_psm]).round(3)
results_final['accuracy'] = (1 - results_final['accuracy'].abs()) * 100
```

[132]:
```
results_final
```

[132]:

```
                              run_time    ate  accuracy
method
regression_estimate low      6.618  2.532      98.7
                    high     4.319 -2.951      98.4
DRE                 low      0.115  2.645      94.2
                    high     0.438 -3.082      97.3
psm                 low      3.323  2.586      96.5
                    high    23.656 -3.069      97.7
```

[ ]: