Homework 2: ATE Estimation with the voter turnout data

Introduction

In this assignment, we'll be working with some of the data described in the paper "Social Pressure and Voter Turnout: Evidence from a Large-Scale Field Experiment" by Gerber et al. (2008). We attained the data from this Github repo, specifically this file. It's also included in the assignment zip file. In this assignment we'll build several ATE estimators by reduction to two estimations from incomplete data, as discussed in lecture. Parts of this notebook are based on a notebook from a tutorial/course on causal inference at Stanford GSB. Although not necessary, you may find it interesting to refer to, as they give more details about the covariates, and they they cover some methods that we don't get into (and vice versa).

Data prep

In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [2]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from scipy.stats import norm, sem
from scipy.interpolate import UnivariateSpline
from sklearn.calibration import calibration_curve
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats.stats import pearsonr
from numpy.random import default_rng
from collections import defaultdict
from scipy.stats import norm
```

In [3]:
```
## Read the data, select some columns, and lightly process
df = pd.read_csv('sp.csv.xz')
cts_variables_names = ["yob", "hh_size", "totalpopulation_estimate",
                         "percent_male", "median_age",
                         "percent_62yearsandover",
                         "percent_white", "percent_black",
                         "percent_asian", "median_income",
                         "employ_20to64", "highschool", "bach_orhigher",
                         "percent_hispanicorlatino"]
binary_variables_names = ["sex","g2000", "g2002", "p2000", "p2002", "p2004"]
scaled_cts_covariates = StandardScaler().fit_transform(df[cts_variables_names])
binary_covariates = df[binary_variables_names]
d = pd.DataFrame(np.concatenate((scaled_cts_covariates, binary_covariates), axis=1),
                        columns=cts_variables_names+binary_variables_names, index=df.in
d["W"] = df["treat_neighbors"]
d["Y"] = df["outcome_voted"]
```

Problem 1: ATE for RCT

All individuals in this experiment had an equal probability of being assigned to the treatment group, so the difference-of-means will be a reasonable estimator for the average treatment effect (ATE). Write a function that computes the difference-of-means estimator for the treatment effect, along with an approximate 95% confidence interval. Apply it to the dataset d computed above and report the results. Save the estimate of the ATE and the radius of the confidence interval for later use.

In [4]:
```
def get_diff_of_means(d, alpha=0.05):

    control = d.loc[d.W == 0]
    treatment = d.loc[d.W == 1]
    y1 = sum(treatment.W * treatment.Y)/sum(treatment.W)
    y0 = sum((1-control.W) * control.Y)/sum(1-control.W)
    ate = y1-y0

    std = []
    for b in range(500):
        sample = d.sample(n=len(d), replace=True)
        control = sample.loc[d.W == 0]
        treatment = sample.loc[d.W == 1]
        y1 = sum(treatment.W * treatment.Y)/sum(treatment.W)
        y0 = sum((1-control.W) * control.Y)/sum(1-control.W)
        std.append(y1-y0)

    std = np.std(std)
    return ate, norm.ppf(1 - alpha/2) * std


ate, ate_CI_radius = get_diff_of_means(d, alpha=0.05)
```

In [5]:
```
print(ate, ate_CI_radius)
```

```
0.08639696096960969 0.007326609603947526
```

Problem 2: ATE estimation with known confounders

In this problem we're going to take a relatively small and biased subsample of our full dataset and try to use that to estimate the ATE. Our approach is to reduce ATE estimation to estimating a mean in the MAR setting.

Below we give a function that takes a biased sample using an approach similar to this one. The details aren't important for what follows.

In [6]:
```
likely_voters =  (((d['g2002'] == 1) & (d['sex'] == 0)) | (d['p2004'] == 1))
unlikely_voters_control = (~likely_voters) & (d["W"] == 0)
likely_voters_treatment = likely_voters & (d["W"] == 1)

def get_biased_sample(d, overall_subsample_rate=.03, bias_rate=.4, rng=default_rng(0)):
    keep_prob = overall_subsample_rate * np.ones(len(d))
    keep_prob[unlikely_voters_control] *= bias_rate
    keep_prob[likely_voters_treatment] *= bias_rate
    keep = rng.random(len(d)) <= keep_prob
    d_bias = d[keep]
    return d_bias
```

```
d_bias = get_biased_sample(d)
print(f"We've sampled {len(d_bias)} instances out of {len(d)}.")
```

```
We've sampled 2740 instances out of 119999.
```

Part A

Below we provide the function get_basic_MAR_estimators, which computes the complete-case mean and the IPW mean in the MAR setting with known propensity scores. You are to complete the get_ATE_estimators function below so that it *uses the get_basic_MAR_estimators function* and computes one ATE estimate for each type of estimator produced by get_basic_MAR_estimators (i.e. the complete-case mean and the IPW mean). Use logistic regression to estimate the propensity scores from the data provided to the function. Run the existing code block below to display the results.

In [7]:
```
def get_basic_MAR_estimators(Y, X, pi, R):
    """
    
    Args:
        Y (pd.Series): the measurement of the outcome Y_i
        X (pd.DataFrame): covariates, rows correspond to entries of Y
        R (pd.Series): boolean series indicating whether Y was observed
        pi (pd.Series): propensity scores corresponding to observations
        
    Returns:
        dict of estimator names and estimates for EY
    """
    est = {}
    n = len(Y)
    ## All the estimators below assume we know the pi (i.e. "missing by design")
    est["mean"] = np.mean(Y[R])
    est["ipw_mean"] = np.sum(Y[R] / pi[R]) / n
    return est
```

In [8]:
```
def get_ATE_estimators(Y, X, W, get_MAR_estimators=get_basic_MAR_estimators):
    """
    
    Args:
        Y (pd.Series): the measurement of the outcome Y_i
        X (pd.DataFrame): covariates, rows correspond to entries or Y
        W (pd.Series): 0/1 series indicating control (0) or treatment (1) assignment
        get_MAR_estimators: function behaving like get_basic_MAR_estimators above
        
    Returns:
        dict of ATE estimator names and estimates, same format as for get_MAR_estimator
    """
    ate_est = {}
    model = LogisticRegression()
    model.fit(X, W)
    pi = model.predict_proba(X)
    mar_control = get_MAR_estimators(Y, X, pi[:, 0], W==0)
    mar_treatment = get_MAR_estimators(Y, X, pi[:, 1], W==1)
    names = list(mar_control.keys())
    for est in names:
        ate_est[est] = mar_treatment[est] - mar_control[est]
        
    return ate_est
```

In [9]:
```
## Run the following and report
d_bias = get_biased_sample(d, overall_subsample_rate=.03, bias_rate=.4, rng=default_rng
X = d_bias.drop(columns=['W','Y'])
ate_est = get_ATE_estimators(Y=d_bias["Y"], X=X, W=d_bias["W"], get_MAR_estimators=get_
print(ate_est)
```

```
{'mean': 0.03636661211129294, 'ipw_mean': 0.14150098309966663}
```

Part B

In this part we significantly expand our ATE estimators and see how they perform over repeated trials. Complete get_MAR_estimators below to include

- Self-normalized IPW mean
- Linear regression imputation
- IPW linear regression imputation (as defined in lecture)
- IW linear regression imputation (as defined in lecture)
- Augmented IPW using linear regression
- [Optional (not for credit): nonlinear regression, or any other variations you'd like to try]

Run the code below to assess performance of these estimators.

In [10]:
```
def get_MAR_estimators(Y, X, pi, R):
    """

    Args:
        Y (pd.Series): the measurement of the outcome Y_i
        X (pd.DataFrame): covariates, rows correspond to entries or Y
        R (pd.Series): boolean series indicating whether Y was observed
        pi (pd.Series): propensity scores corresponding to observations

    Returns:
        dict of estimator names and estimates for EY
    """
    est = {}
    n = len(Y)
    ## All the estimators below assume we know the pi (i.e. "missing by design")
    est["mean"] = np.mean(Y[R])
    est["ipw_mean"] = np.sum(Y[R] / pi[R]) / n


    est["sn_ipw_mean"] = np.sum(Y[R]/pi[R]) / np.sum(1/pi[R])


    lr = LinearRegression()
    lr.fit(X[R], Y[R])
    est["linear_regression"] = (sum(lr.predict(X[~R])) + sum(Y[R])) / n


    ipw_lr = LinearRegression()
    weights = 1 / pi[R]
    ipw_lr.fit(X[R], Y[R], sample_weight=weights)


    est["ipw_linear"] = (sum(ipw_lr.predict(X[~R])) + sum(Y[R])) / n

    iw_lr = LinearRegression()
    weights = (1-pi[R]) / pi[R]
    iw_lr.fit(X[R], Y[R], sample_weight=weights)
    est["iw_linear"] = (sum(iw_lr.predict(X[~R])) + sum(Y[R])) / n


    y_term = np.zeros(n)
    y_term[R] = Y[R]/pi[R]
    control = np.zeros(n)
    control[R] = lr.predict(X[R])/pi[R]
    est["aipw"] = sum(y_term - control + lr.predict(X)) / n

    return est
```

In [11]:

```
def get_estimator_stats(estimates, true_parameter_value=None):
    """


     Args:
        estimates (pd.DataFrame): each row corresponds to collection of estimates for a
            each column corresponds to an estimator
        true_parameter_value (float): the true parameter value that we will be comparin

    Returns:
        pd.Dataframe where each row represents data about a single estimator
    """


    est_stat = []
    for est in estimates.columns:
        pred_means = estimates[est]
        stat = {}
        stat['stat'] = est
        stat['mean'] = np.mean(pred_means)
        stat['SD'] = np.std(pred_means)
        stat['SE'] = np.std(pred_means) / np.sqrt(len(pred_means))
        if true_parameter_value:
            stat['bias'] = stat['mean'] - true_parameter_value
            stat['RMSE'] = np.sqrt(np.mean((pred_means - true_parameter_value) ** 2))
        est_stat.append(stat)

    return pd.DataFrame(est_stat)
```

In [12]:

```
def run_experiments(sampler, num_repeats=10,rng=default_rng(0)):
    data_list = []
    num_obs_list = []
    for i in range(num_repeats):
        d = sampler(rng)
        X = d.drop(columns=['W','Y'])
        ate_est = get_ATE_estimators(Y=d["Y"], X=X, W=d["W"], get_MAR_estimators=get_MA
        data_list.append(ate_est)
    results = pd.DataFrame(data_list)
    return results
```

In [13]:

```
def sampler(rng):
    return get_biased_sample(d, overall_subsample_rate=.06, bias_rate=.4, rng=rng)
rng = default_rng(0)
results = run_experiments(sampler, num_repeats=500, rng=rng)
```

In [14]:

```
results_eval = get_estimator_stats(results, true_parameter_value=ate)
results_eval
```

Out[14]:

	stat	mean	SD	SE	bias	RMSE
0	mean	0.033087	0.018351	0.000821	-0.053310	0.056380
1	ipw_mean	0.134785	0.028370	0.001269	0.048388	0.056091
2	sn_ipw_mean	0.101437	0.025028	0.001119	0.015040	0.029199
3	linear_regression	0.087300	0.022025	0.000985	0.000903	0.022043
4	ipw_linear	0.086882	0.022321	0.000998	0.000485	0.022327
5	iw_linear	0.086868	0.022254	0.000995	0.000471	0.022259
6	aipw	0.087040	0.022588	0.001010	0.000643	0.022597

Part C

You should see that the ATE estimate based on the IPW mean is significantly biased, and the bias seems to be driving most of the RMSE. In class we showed that the IPW mean is an unbiased estimator for EY in the MAR setting, and the corresponding ATE estimator is also unbiased. Why are we seeing bias in this experiment?

Problem 3: Bootstrap confidence intervals for ATE

Let's now consider a more realistic scenario, in which we only have a single sample to work with. In this case it's very important to be able to include some form of uncertainty measure with our estimates. In this problem we'll do this using the normal approximated bootstrap confidence intervals described in our module on CATEs. However, in this setting, rather than estimating the CATE, we're just estimating the ATE.

Part A

Complete the function get_stratified_bootstrap_CI to generate 95% normal approximated bootstrap confidence intervals for each of the ATE estimators we've developed above. Execute the code below to test the function and produce the results as a table and in a plot.

In [15]:
```
def get_bootstrap_stats(boot_estimates, full_estimates, alpha=0.05):
    est_stat = []
    signif_level = -norm.ppf(alpha/2)
    for est in full_estimates:
        est_boot = np.array(boot_estimates[est])
        stat = {}
        stat['estimator'] = est
        stat['estimate'] = full_estimates[est]
        #stat['mean_boot'] = np.mean(est_boot)
        stat['SD'] = np.std(est_boot)
        stat['CI_radius'] = signif_level * stat['SD']
        stat['lower_ci'] = stat['estimate'] - stat['CI_radius']
        stat['upper_ci'] = stat['estimate'] + stat['CI_radius']
        est_stat.append(stat)

    return pd.DataFrame(est_stat)


def get_stratified_bootstrap_CI(Y, X, W, estimators, num_bootstrap=10, alpha=0.05):
    """
    Returns:
        pd.Dataframe with
            column "estimator" with the name of the estimator,
            column "estimate" which is the estimate based on (Y,X,W)
                which serves as the center of our bootstrap confidence intervals
            column "CI_radius" which is the radius of the confidence interval
            (additional columns may be included as you wish)
    """

    n = len(Y)
    df = pd.concat([X, W, Y], axis=1)
    treatment = df.loc[df.W == 1]
    control = df.loc[df.W == 0]
    d = defaultdict(list)
    data = []
    mean_prediction = estimators(Y, X, W)

    for b in range(num_bootstrap):
        treatment_sample = treatment.sample(n=len(treatment), replace=True)
        control_sample = control.sample(n=len(control), replace=True)
        bootstrap = pd.concat([control_sample, treatment_sample], ignore_index=True)
        ate_est = estimators(bootstrap.Y, bootstrap.drop(columns=["W", "Y"]), bootstrap
        for est, prediction in ate_est.items():
            d[est].append(prediction)
    for est, lst in d.items():
        std = np.std(lst)
        radius = norm.ppf(1 - alpha/2) * std
        data.append({"estimator": est, "estimate": mean_prediction[est], "CI_radius": r

    return pd.DataFrame.from_dict(data)
```

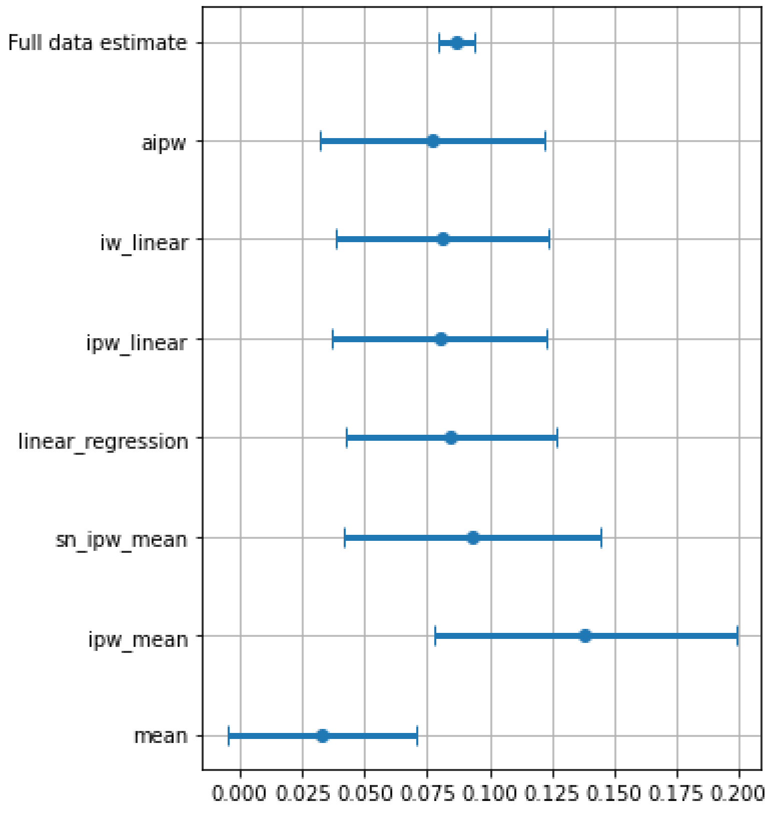
In [16]:

```
rng = default_rng(8)
d_samp = sampler(rng) # our single biased sample
X = d_samp.drop(columns=['W','Y'])
estimators = lambda Y, X, W: get_ATE_estimators(Y, X, W, get_MAR_estimators=get_MAR_est
ci = get_stratified_bootstrap_CI(Y=d_samp["Y"], X=X, W=d_samp["W"], estimators=estimato
print(f"A 95% confidence interval for the ATE based on the full dataset is {ate}+/-{ate
print(ci)
to_plot = ci[['estimator','estimate','CI_radius']]
to_plot = to_plot.append({'estimator':'Full data estimate','estimate':ate,'CI_radius':a
fig, ax =plt.subplots(1,1, figsize=(5,7))
ax.errorbar(to_plot['estimate'], np.arange(len(to_plot)), \
            xerr=to_plot['CI_radius'],
            fmt='o', elinewidth=3, capsize=5)
ax.grid('on')
ax.set_yticks(np.arange(len(to_plot)))
ax.set_yticklabels(to_plot["estimator"])
```

```
A 95% confidence interval for the ATE based on the full dataset is 0.08639696096960969+/
-0.007326609603947526
           estimator  estimate  CI_radius
0               mean  0.032720   0.037843
1           ipw_mean  0.138417   0.060523
2        sn_ipw_mean  0.093061   0.051309
3  linear_regression  0.084381   0.042323
4         ipw_linear  0.079859   0.043054
5          iw_linear  0.081245   0.042762
6               aipw  0.076795   0.045203
```

Out[16]:

```
[Text(0, 0, 'mean'),
 Text(0, 1, 'ipw_mean'),
 Text(0, 2, 'sn_ipw_mean'),
 Text(0, 3, 'linear_regression'),
 Text(0, 4, 'ipw_linear'),
 Text(0, 5, 'iw_linear'),
 Text(0, 6, 'aipw'),
 Text(0, 7, 'Full data estimate')]
```



```
In [ ]:
```