negative_controls_extended_graphs

February 12, 2020

[1]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import patsy

import statsmodels.formula.api as smf
import statsmodels.api as sm

from joblib import Parallel, delayed

%matplotlib inline
```

[2]:
```
plt.style.use('seaborn-whitegrid')
font = {#'family' : 'normal',
        'weight' : 'bold',
        'size'   : 16}
plt.rc('font', **font)
plt.rc('lines', linewidth=4)
plt.rc('xtick',labelsize=16)
plt.rc('ytick',labelsize=16)
plt.rc('legend', fontsize=8)
```

0.1 Entended graphs of $UWXYZ$

We extend the simulation from the first notebook (negative_controls_base_graph.ipynb) by adding parents to the observables $V \in W, X, Y, Z$ from the original graph, and adding both parents and children to the unobservable U. The illustration of these figure are provided in Figure 2b, 2c, and 2d of the paper. We will estimate the bias of ATE from both regression and negative controls.

0.2 Figure 2a

[3]:
```
# For a given set of parameters that specifies the DGP, return a dataframe of n␣
↪draws.
def simulate_UWXYZ_2a(deltas, phi=0.5, n=int(1e6) ):
```

```python
    ## Draw U
    if phi<0.0 or phi>1.0:
        print('phi is out of bounds.')
        return
    df = pd.DataFrame({'u':np.random.binomial(n=1, p=phi, size=n)})

    ## Draw W, which is dependent on U
    probs = phi+(df['u']-1.0/2)*deltas['UW']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for W are out of bounds.')
        return
    df['w'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw X, which is dependent on U
    probs = phi+(df['u']-1.0/2)*deltas['UX']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for X are out of bounds.')
        return
    df['x'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw Z, which is dependent on U and X
    probs = phi+(df['u']-1.0/2)*deltas['UZ']+(df['x']-1.0/2)*deltas['XZ']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for Z are out of bounds.')
        return
    df['z'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw Y, which is dependent on U, W, and X
    probs = phi+(df['u']-1.0/2)*deltas['UY']+(df['w']-1.0/
 ↪2)*deltas['WY']+(df['x']-1.0/2)*deltas['XY']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for Y are out of bounds.')
        return
    df['y'] = np.random.binomial(n=1, p=probs, size=n)

    return(df)

base_diff = 0.10
deltas = {'UW':base_diff*2,
          'UX':base_diff,
          'UY':base_diff,
          'UZ':base_diff*2,
          'WY':base_diff,
          'XY':base_diff,
          'XZ':base_diff }
df_sim_2a = simulate_UWXYZ_2a(deltas = deltas)
```

```
df_sim_2a.groupby(['u','w','z','x']).mean().apply(lambda x: np.round(x,3)).
↪head(n=8)
```

[3]:
```
             y
u w z x
0 0 0 0  0.347
      1  0.451
    1 0  0.353
      1  0.454
  1 0 0  0.452
      1  0.552
    1 0  0.449
      1  0.554
```

[4]:
```
def calculate_condition_number(df):

    X = 'x'
    # proxies
    W = 'w'; W_val = 1
    Z = 'z'; Z_val = 1

    # P(W | Z, x) represents two matrices, one for each value of x
    def calculate_condition_number_given_x(df, X_val):
        # p(W | X, Z=0)
        pWgXZ0 = np.bincount(df[(df[X]==X_val) & (df[Z]!=Z_val)][W]==W_val)
        pWgXZ0 = pWgXZ0 / pWgXZ0.sum()

        # p(W | X, Z=1)
        pWgXZ1 = np.bincount(df[(df[X]==X_val) & (df[Z]==Z_val)][W]==W_val)
        pWgXZ1 = pWgXZ1 / pWgXZ1.sum()

        pWZx = np.stack((pWgXZ0, pWgXZ1), axis=-1)
        return(np.linalg.cond(pWZx))

    condition_numbers = [calculate_condition_number_given_x(df, X_val) for␣
↪X_val in [0,1]]

    return(max(condition_numbers) )

print('%.2f'%calculate_condition_number(df_sim_2a) )
```

```
25.08
```

[5]:
```
## Let's estimate the true ATE from the simulation itself with the following␣
↪function.

def calculate_true_ate(df):
```

```
    def delta_p_cond(group, n_obs):
        group_frac = np.sum(group['count'])*1.0/n_obs
        if group['x'].unique().size < 2:
            delta_p = 0.0
        else:
            delta_p = group.loc[group['x']==1,'mean'].iloc[0]-group.
 ↪loc[group['x']==0,'mean'].iloc[0]
        return( pd.Series([group_frac,delta_p], index=['group_frac','delta_p'])␣
 ↪ )

    exog_cols = [i for i in df.columns.to_list() if i!='y']
    exog_cols_minus_x = [i for i in exog_cols if i!='x']

    true_ate = \
        df.groupby(exog_cols)\
            .agg(['count','mean'])['y']\
            .reset_index()\
            .groupby(exog_cols_minus_x)\
            .apply(delta_p_cond, df.shape[0])\
            .reset_index()\
            .apply(lambda x: x['group_frac']*x['delta_p'], axis=1)\
            .sum()
    return(true_ate)

print('True ATEs:\nEmpirical is %.2f p.p. Intended is %.2f p.p.' %
      (calculate_true_ate(df_sim_2a)*100, deltas['XY']*100) )
```

```
True ATEs:
Empirical is 10.17 p.p. Intended is 10.00 p.p.
```

[6]:
```
## Now, let's deploy the negative controls estimator and see how it recovers␣
 ↪the true ATE.

def calculate_ate_negative_controls(df):

    X = 'x'
    Y = 'y'
    # proxies
    W = 'w'; W_val = 1
    Z = 'z'; Z_val = 1

    def calculate_pYdoX(df, X_val):
        # p(Y | X, Z=0)
        pYgXZ0 = np.bincount(df[(df[X]==X_val) & (df[Z]!=Z_val)][Y])
        pYgXZ0 = pYgXZ0 / pYgXZ0.sum()
```

```python
        # p(Y | X, Z=1)
        pYgXZ1 = np.bincount(df[(df[X]==X_val) & (df[Z]==Z_val)][Y])
        pYgXZ1 = pYgXZ1 / pYgXZ1.sum()

        # p(W)
        pW = np.bincount(df[W]==W_val)
        pW = pW / pW.sum()

        # p(W | X, Z=0)
        pWgXZ0 = np.bincount(df[(df[X]==X_val) & (df[Z]!=Z_val)][W]==W_val)
        pWgXZ0 = pWgXZ0 / pWgXZ0.sum()

        # p(W | X, Z=1)
        pWgXZ1 = np.bincount(df[(df[X]==X_val) & (df[Z]==Z_val)][W]==W_val)
        pWgXZ1 = pWgXZ1 / pWgXZ1.sum()

        # Miao et al. adjustment (see paper)
        denom = pWgXZ0[0] - pWgXZ1[0]
        weight_0 = (pW[0] - pWgXZ1[0]) / denom
        weight_1 = (pWgXZ0[0] - pW[0]) / denom

        pYdoXmiao = pYgXZ0 * weight_0 + pYgXZ1 * weight_1

        # formula (5) using matrix inversion
        pWZx = np.stack((pWgXZ0, pWgXZ1), axis=-1)
        condition_number = np.linalg.cond(pWZx)
        weights = np.dot(np.linalg.pinv(pWZx), pW)

        pYdoXmiao_pinv = pYgXZ0 * weights[0] + pYgXZ1 * weights[1]

        return(pYdoXmiao_pinv[1], condition_number)

    pYdoX_results = [calculate_pYdoX(df, X_val) for X_val in [0,1]]

    condition_number = max([i[1] for i in pYdoX_results])
    negative_controls_ate = pYdoX_results[1][0] - pYdoX_results[0][0]
    return(negative_controls_ate, condition_number)

negative_controls_result = calculate_ate_negative_controls(df_sim_2a)

print('Method: relative bias (condition number), true ATE')
true_ate = calculate_true_ate(df_sim_2a)
print('Negative controls: %.1f%% (%.0f), %.1f%% ' %␣
 ↪((negative_controls_result[0]-true_ate)/true_ate*100,
                                                                ␣
 ↪negative_controls_result[1], true_ate*100 ) )
```

```
Method: relative bias (condition number), true ATE
Negative controls: -0.1% (25), 10.2%
```

[7]:
```python
def calculate_ate_regression(df, formula='y ~ 1 + w + x + z', family=sm.
 ↪families.Binomial()):

    model = smf.glm(formula=formula, data=df, family=family )
    model_result = model.fit(use_t=1)
    #print(model_result.summary())

    ##calculate ATE with 95% CI
    ones_vector  = model_result.params.index!='x'
    zeros_vector = model_result.params.index=='x'

    params_mid = model_result.params
    params_lower = model_result.params * ones_vector + model_result.
 ↪conf_int()[0] * zeros_vector
    params_upper = model_result.params * ones_vector + model_result.
 ↪conf_int()[1] * zeros_vector

    result_list = []
    patsy_df = patsy.dmatrices(model.formula, df, return_type='dataframe')[1]
    for params_i in [params_mid, params_lower, params_upper]:
        patsy_df['x'] = 0
        p0 = model.predict(params_i, patsy_df, linear=False).mean()
        patsy_df['x'] = 1
        p1 = model.predict(params_i, patsy_df, linear=False).mean()
        result_list.append(p1-p0)

    return(result_list[0], result_list[0]-result_list[1],␣
 ↪result_list[2]-result_list[0])

regression_comparison_results = {}
regression_comparison_results['LR'] = calculate_ate_regression(df_sim_2a)
regression_comparison_results['OLS'] = calculate_ate_regression(df_sim_2a,␣
 ↪family=sm.families.Gaussian() )
regression_comparison_results['LR, with U'] =␣
 ↪calculate_ate_regression(df_sim_2a, formula='y ~ 1 + u + w + x + z')

print('Method: relative bias (LB, UB)')
true_ate = calculate_true_ate(df_sim_2a)
for key, value in regression_comparison_results.items():
    print('%s: %.1f%% (-%.1f%%, %.1f%%)' % (key, (value[0]-true_ate)/
 ↪true_ate*100,
                                            value[1]/true_ate*100, value[2]/
 ↪true_ate*100))
```

```
Method: relative bias (LB, UB)
LR: 7.0% (-1.9%, 1.9%)
OLS: 7.0% (-1.9%, 1.9%)
LR, with U: 0.0% (-1.9%, 1.9%)
```

[8]:
```python
## Comparison of the two methods
def run_comparison(df, deltas, obs_nodes, all_nodes):

    true_ate = calculate_true_ate(df)
    print('True empirical ATE is %.2f p.p. Intended ATE is %.2f p.p.' %␣
↪(true_ate*100, deltas['XY']*100) )

    regression_comparison_results = {}
    regression_comparison_results['LR, with obs. nodes'] = \
        calculate_ate_regression(df, formula = 'y ~ 1 + %s' % ' + '.
↪join(obs_nodes))
    regression_comparison_results['LR, with obs. nodes + U'] = \
        calculate_ate_regression(df, formula = 'y ~ 1 + %s' % ' + '.
↪join(obs_nodes+['u']))
    regression_comparison_results['LR, with all nodes'] = \
        calculate_ate_regression(df, formula = 'y ~ 1 + %s' % ' + '.
↪join(all_nodes))

    print('Method: relative bias (LB, UB)')
    for key, value in regression_comparison_results.items():
        print('%s: %.1f%% (-%.1f%%, %.1f%%)' % (key, (value[0]-true_ate)/
↪true_ate*100,
                                                value[1]/true_ate*100, value[2]/
↪true_ate*100))

    negative_controls_result = calculate_ate_negative_controls(df)

    print('Method: relative bias (condition number), true ATE')
    print('Negative controls: %.1f%% (%.0f), %.2f p.p. ' %␣
↪((negative_controls_result[0]-true_ate)/true_ate*100,
                                                                           ␣
↪negative_controls_result[1], true_ate*100 ) )

    return

print('Figure 2a simulation results comparison:')
run_comparison(df_sim_2a, deltas, ['w','x','z'], ['u','w','x','z'])
```

```
Figure 2a simulation results comparison:
True empirical ATE is 10.17 p.p. Intended ATE is 10.00 p.p.
Method: relative bias (LB, UB)
LR, with obs. nodes: 7.0% (-1.9%, 1.9%)
```

```
LR, with obs. nodes + U: 0.0% (-1.9%, 1.9%)
LR, with all nodes: 0.0% (-1.9%, 1.9%)
Method: relative bias (condition number), true ATE
Negative controls: -0.1% (25), 10.17 p.p.
```

[9]:
```python
## Run many comparisons to get variance of negative controls
n_comparison = 100

def run_generic_comparison(func_gen_df, deltas, obs_nodes):

    df = func_gen_df(deltas)
    true_ate = calculate_true_ate(df)
    regression_comparison_results = calculate_ate_regression(df, formula = 'y ~␣
↪1 + %s' % ' + '.join(obs_nodes))
    negative_controls_result = calculate_ate_negative_controls(df)

    return((regression_comparison_results[0]-true_ate)/true_ate*100,
           (negative_controls_result[0]-true_ate)/true_ate*100 )

def run_generic_comparison_n_times_and_print(func_gen_df, deltas, obs_nodes,␣
↪n_comparison, desc):
    exp_list = Parallel(n_jobs=-1, max_nbytes=None)\
        (delayed(run_generic_comparison)(func_gen_df, deltas, obs_nodes)\
         for i in range(n_comparison) )

    print("%s simulation results over %d comparisons:" % (desc,len(exp_list)) )
    print("LR, with obs. nodes: %.1f%% +/- %.1f%%" % (np.mean([i[0] for i in␣
↪exp_list]),
                                                        2*np.std([i[0] for i in␣
↪exp_list])  ))
    print("Negative controls:  %.1f%% +/- %.1f%%" % (np.mean([i[1] for i in␣
↪exp_list]),
                                                       2*np.std([i[1] for i in␣
↪exp_list])  ))
    return

run_generic_comparison_n_times_and_print(simulate_UWXYZ_2a, deltas,␣
↪['w','x','z'], n_comparison, 'Figure 2a')
```

```
Figure 2a simulation results over 100 comparisons:
LR, with obs. nodes: 7.3% +/- 0.3%
Negative controls:  -0.1% +/- 1.2%
```

0.3 Figure 2b

[10]:
```
# For a given set of parameters that specifies the DGP, return a dataframe of n␣
↪draws.
def simulate_UWXYZ_2b(deltas, phi=0.5, n=int(1e6) ):

    ## Draw U*
    if phi<0.0 or phi>1.0:
        print('phi is out of bounds.')
        return
    df = pd.DataFrame({'u*':np.random.binomial(n=1, p=phi, size=n)})

    ## Draw U, which is dependent on U*.
    probs = phi+(df['u*']-1.0/2)*deltas['U*U']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for U are out of bounds.')
        return
    df['u'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw W, which is dependent on U and U*.
    probs = phi+(df['u*']-1.0/2)*deltas['U*W']+(df['u']-1.0/2)*deltas['UW']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for W are out of bounds.')
        return
    df['w'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw Z, which is dependent on U.
    probs = phi+(df['u']-1.0/2)*deltas['UZ']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for Z are out of bounds.')
        return
    df['z'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw X, which is dependent on U and Z
    probs = phi+(df['u']-1.0/2)*deltas['UX']+(df['z']-1.0/2)*deltas['ZX']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for X are out of bounds.')
        return
    df['x'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw Y, which is dependent on U*, U, W, and X
    probs = phi+(df['u*']-1.0/2)*deltas['U*Y']+(df['u']-1.0/
↪2)*deltas['UY']+(df['w']-1.0/2)*deltas['WY']+(df['x']-1.0/2)*deltas['XY']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for Y are out of bounds.')
        return
    df['y'] = np.random.binomial(n=1, p=probs, size=n)
```

```
    return(df)

base_diff = 0.10
deltas = {'U*U':base_diff,
          'U*W':base_diff,
          'U*Y':base_diff,
          'UW':base_diff*2,
          'UX':base_diff,
          'UY':base_diff,
          'UZ':base_diff*2,
          'WY':base_diff,
          'XY':base_diff,
          'ZX':base_diff }
df_sim_2b = simulate_UWXYZ_2b(deltas = deltas)
df_sim_2b.groupby(['u*','u','w','z','x']).mean().apply(lambda x: np.round(x,3)).
 ↪head(n=8)
```

[10]:

```
                y
u* u w z x
0  0 0 0 0  0.300
         1  0.399
       1 0  0.299
         1  0.396
     1 0 0  0.404
         1  0.496
       1 0  0.400
         1  0.495
```

[11]:

```
print('Figure 2b simulation results comparison:')
run_comparison(df_sim_2b, deltas, ['w','x','z'], ['u*','u','w','x','z'])
```

```
Figure 2b simulation results comparison:
True empirical ATE is 9.96 p.p. Intended ATE is 10.00 p.p.
Method: relative bias (LB, UB)
LR, with obs. nodes: 10.2% (-2.0%, 2.0%)
LR, with obs. nodes + U: -0.1% (-2.0%, 2.0%)
LR, with all nodes: -0.1% (-2.0%, 2.0%)
Method: relative bias (condition number), true ATE
Negative controls: 1.0% (25), 9.96 p.p.
```

[12]:

```
run_generic_comparison_n_times_and_print(simulate_UWXYZ_2b, deltas,␣
 ↪['w','x','z'], n_comparison, 'Figure 2b')
```

```
Figure 2b simulation results over 100 comparisons:
LR, with obs. nodes: 10.1% +/- 0.4%
Negative controls:  -0.0% +/- 1.6%
```

0.4 Figure 2c

[13]:
```python
# For a given set of parameters that specifies the DGP, return a dataframe of n␣
↪draws.
def simulate_UWXYZ_2c(deltas, phi=0.5, n=int(1e6) ):

    ## Draw U1
    if phi<0.0 or phi>1.0:
        print('phi is out of bounds.')
        return
    df = pd.DataFrame({'u1':np.random.binomial(n=1, p=phi, size=n)})

    ## Draw U, which is dependent on U1.
    probs = phi+(df['u1']-1.0/2)*deltas['U1U']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for U are out of bounds.')
        return
    df['u'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw U2, which is dependent on U.
    probs = phi+(df['u']-1.0/2)*deltas['UU2']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for U2 are out of bounds.')
        return
    df['u2'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw W, which is dependent on U and U2.
    probs = phi+(df['u']-1.0/2)*deltas['UW']+(df['u2']-1.0/2)*deltas['U2W']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for W are out of bounds.')
        return
    df['w'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw Z, which is dependent on U and U1.
    probs = phi+(df['u']-1.0/2)*deltas['UZ']+(df['u1']-1.0/2)*deltas['U1Z']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for Z are out of bounds.')
        return
    df['z'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw X, which is dependent on U, U1, and Z
    probs = phi+(df['u']-1.0/2)*deltas['UX']+(df['u1']-1.0/
↪2)*deltas['U1X']+(df['z']-1.0/2)*deltas['ZX']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for X are out of bounds.')
        return
    df['x'] = np.random.binomial(n=1, p=probs, size=n)
```

```
    ## Draw Y, which is dependent on U, U2, W, and X
    probs = phi+(df['u']-1.0/2)*deltas['UY']+(df['u2']-1.0/
 ↪2)*deltas['U2Y']+(df['w']-1.0/2)*deltas['WY']+(df['x']-1.0/2)*deltas['XY']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for Y are out of bounds.')
        return
    df['y'] = np.random.binomial(n=1, p=probs, size=n)

    return(df)

deltas = {'U1U':base_diff,
          'U1X':base_diff,
          'U1Z':base_diff,
          'UU2':base_diff,
          'UW':base_diff*2,
          'UX':base_diff,
          'UY':base_diff,
          'UZ':base_diff*2,
          'U2W':base_diff,
          'U2Y':base_diff,
          'WY':base_diff,
          'XY':base_diff,
          'ZX':base_diff }
df_sim_2c = simulate_UWXYZ_2c(deltas = deltas)
df_sim_2c.groupby(['u','u1','u2','w','z','x']).mean().apply(lambda x: np.
 ↪round(x,3)).head(n=8)
```

[13]:

```
                    y
u u1 u2 w z x
0 0  0  0 0 0   0.303
            1   0.400
          1 0   0.299
            1   0.400
        1 0 0   0.399
            1   0.499
          1 0   0.400
            1   0.490
```

[14]:

```
print('Figure 2c simulation results comparison:')
run_comparison(df_sim_2c, deltas, ['w','x','z'], ['u','u1','u2','w','x','z'])
```

```
Figure 2c simulation results comparison:
True empirical ATE is 10.00 p.p. Intended ATE is 10.00 p.p.
Method: relative bias (LB, UB)
LR, with obs. nodes: 10.9% (-1.9%, 1.9%)
LR, with obs. nodes + U: -0.1% (-1.9%, 1.9%)
```

```
LR, with all nodes: -0.0% (-1.9%, 1.9%)
Method: relative bias (condition number), true ATE
Negative controls: 0.7% (25), 10.00 p.p.
```

[15]:
```
run_generic_comparison_n_times_and_print(simulate_UWXYZ_2c, deltas,␣
 ↪['w','x','z'], n_comparison, 'Figure 2c')
```

```
Figure 2c simulation results over 100 comparisons:
LR, with obs. nodes: 10.8% +/- 0.5%
Negative controls:  -0.0% +/- 1.5%
```

0.5 Figure 2d

[16]:
```
# For a given set of parameters that specifies the DGP, return a dataframe of n␣
 ↪draws.
def simulate_UWXYZ_2d(deltas, phi=0.5, n=int(1e6) ):

    ## Draw U1
    if phi<0.0 or phi>1.0:
        print('phi is out of bounds.')
        return
    df = pd.DataFrame({'u1':np.random.binomial(n=1, p=phi, size=n)})

    ## Draw U, which is dependent on U1.
    probs = phi+(df['u1']-1.0/2)*deltas['U1U']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for U are out of bounds.')
        return
    df['u'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw U2, which is dependent on U.
    probs = phi+(df['u']-1.0/2)*deltas['UU2']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for U2 are out of bounds.')
        return
    df['u2'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw W, which is dependent on U2.
    probs = phi+(df['u2']-1.0/2)*deltas['U2W']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for W are out of bounds.')
        return
    df['w'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw Z, which is dependent on U1.
    probs = phi+(df['u1']-1.0/2)*deltas['U1Z']
```

```
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for Z are out of bounds.')
        return
    df['z'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw X, which is dependent on Z
    probs = phi+(df['z']-1.0/2)*deltas['ZX']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for X are out of bounds.')
        return
    df['x'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw Y, which is dependent on U2, W, and X
    probs = phi+(df['u2']-1.0/2)*deltas['U2Y']+(df['w']-1.0/
↪2)*deltas['WY']+(df['x']-1.0/2)*deltas['XY']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for Y are out of bounds.')
        return
    df['y'] = np.random.binomial(n=1, p=probs, size=n)

    ## Draw M, which is dependent on U1 and U2
    probs = phi+(df['u1']-1.0/2)*deltas['U1M']+(df['u2']-1.0/2)*deltas['U2M']
    if probs.min()<0.0 or probs.max()>1.0:
        print('probs for M are out of bounds.')
        return
    df['m'] = np.random.binomial(n=1, p=probs, size=n)

    return(df)

deltas = {'U1U':0.50,
          'U1M':0.20,
          'U1Z':0.40,
          'UU2':0.50,
          'U2M':0.20,
          'U2W':0.40,
          'U2Y':0.10,
          'WY':0.10,
          'XY':0.10,
          'ZX':0.10 }
df_sim_2d = simulate_UWXYZ_2d(deltas = deltas)
df_sim_2d.groupby(['u','u1','u2','m','w','z','x']).mean().apply(lambda x: np.
↪round(x,3)).head(n=8)
```

[16]:

```
                      y
u u1 u2 m w z x
0 0  0  0 0 0 0   0.350
              1   0.449
```

```
      1 0   0.351
        1   0.447
    1 0 0   0.450
        1   0.555
      1 0   0.447
        1   0.548
```

[17]:
```
print('Figure 2d simulation results comparison:')
run_comparison(df_sim_2d, deltas, ['m','w','x','z'],␣
 ↪['u','u1','u2','m','w','x','z'])
```

```
Figure 2d simulation results comparison:
True empirical ATE is 9.78 p.p. Intended ATE is 10.00 p.p.
Method: relative bias (LB, UB)
LR, with obs. nodes: 0.1% (-2.0%, 2.0%)
LR, with obs. nodes + U: 0.1% (-2.0%, 2.0%)
LR, with all nodes: -0.0% (-2.0%, 2.0%)
Method: relative bias (condition number), true ATE
Negative controls: 0.4% (25), 9.78 p.p.
```

[18]:
```
run_generic_comparison_n_times_and_print(simulate_UWXYZ_2d, deltas,␣
 ↪['w','x','z'], n_comparison, 'Figure 2d')
```

```
Figure 2d simulation results over 100 comparisons:
LR, with obs. nodes: 0.0% +/- 0.2%
Negative controls:  -0.0% +/- 0.5%
```