negative_controls_high_dimensionality

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 for Figure 2b from the second notebook (negative_controls_extended_graphs.ipynb) by allowing for higher dimensional representations of $\vec{U}*$, $\vec{U}$, and $\vec{X}$. We fix $|\vec{U}^*| + |\vec{U}| = 10$ and vary $|\vec{X}| \leq 10$ and $1 \leq |\vec{U}| \leq 9$. We examine the bias of negative control under these scenarios for the first component of $\vec{X}$.

0.2 Figure 2b, with high dimensionality X

[3]:
```
# For a given set of parameters that specifies the DGP, return a dataframe of n␣
↪draws.
def simulate_UWXYZ_2b_high_dim_X(deltas, phi=0.5, n=int(1e6) ):
```

```
    dim_X = len(deltas['XY'])

    ## 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 = np.ones((n,dim_X))*phi+np.outer((df['u']-1.0/2), deltas['UX'])+np.outer((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
    x_vec = np.random.binomial(n=1, p=probs)
    df = pd.concat([df,pd.DataFrame(x_vec, columns=['x_%d'%i for i in range(dim_X)])], axis=1)
    df['x_state'] = np.dot(x_vec, [2**i for i in range(dim_X)])

    ## 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']+np.dot((x_vec-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
dim_X = 5
deltas = {'U*U':base_diff,
          'U*W':base_diff,
          'U*Y':base_diff,
          'UW':base_diff*2,
          'UX':np.ones(dim_X)*base_diff/dim_X,
          'UY':base_diff,
          'UZ':base_diff*2,
          'WY':base_diff,
          'XY':np.ones(dim_X)*base_diff/dim_X,
          'ZX':np.ones(dim_X)*base_diff/dim_X }
df_sim_2b = simulate_UWXYZ_2b_high_dim_X(deltas = deltas)
print(df_sim_2b.groupby(['u*','u','w','z','x_0']).mean()['y'].apply(lambda x:␣
 ↪np.round(x,3)).head(n=8))
print(df_sim_2b.head())
```

```
u*  u  w  z  x_0
0   0  0  0  0      0.337
             1      0.358
          1  0      0.342
             1      0.366
       1  0  0      0.442
             1      0.456
          1  0      0.448
             1      0.452
Name: y, dtype: float64
   u*  u  w  z  x_0  x_1  x_2  x_3  x_4  x_state  y
0   0  1  1  1    1    0    0    1    1       25  0
1   0  1  0  1    1    1    1    1    1       31  1
2   0  0  1  0    0    1    1    1    1       30  0
3   0  0  0  0    0    1    0    1    0       10  0
4   1  0  1  0    1    0    1    0    0        5  1
```

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

    X = 'x_state'
    # 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))

    Xs = df[X].unique()
    condition_numbers = [calculate_condition_number_given_x(df, X_val) for␣
 ↪X_val in Xs]

    #return(max(condition_numbers) )
    return(np.median(condition_numbers) )

print('Median condition number over x_states = %.
 ↪2f'%calculate_condition_number(df_sim_2b) )
```

```
Median condition number over x_states = 22.79
```

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

def calculate_true_ate(df, x_treatment = 'x_0'):

    def delta_p_cond(group, n_obs):
        group_frac = np.sum(group['count'])*1.0/n_obs
        if group[x_treatment].unique().size < 2:
            delta_p = 0.0
        else:
            delta_p = group.loc[group[x_treatment]==1,'mean'].iloc[0]-group.
 ↪loc[group[x_treatment]==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 not in ['y','x_state']]
    exog_cols_minus_x = [i for i in exog_cols if i!=x_treatment]

    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_2b)*100, deltas['XY'][0]*100) )
```

```
True ATEs:
Empirical is 1.92 p.p. Intended is 2.00 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_0'
    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()

        # 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_2b)

print('Method: relative bias (condition number), true ATE')
true_ate = calculate_true_ate(df_sim_2b)
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: 0.0% (23), 1.9%
```

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

    if 'x' not in df.columns:
        formula = formula.replace('x', ' + '.join(['x_%d'%i for i in␣
 ↪range(dim_X)]))
        x_treatment = 'x_0'

    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_treatment
    zeros_vector = model_result.params.index==x_treatment
```

```
    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_treatment] = 0
        p0 = model.predict(params_i, patsy_df, linear=False).mean()
        patsy_df[x_treatment] = 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_2b)
regression_comparison_results['LR, one treatment'] =␣
 ↪calculate_ate_regression(df_sim_2b, formula = 'y ~ 1 + w + x + z', dim_X = 1)
#regression_comparison_results['OLS'] = calculate_ate_regression(df_sim_2b,␣
 ↪family=sm.families.Gaussian() )
regression_comparison_results['LR, with U'] =␣
 ↪calculate_ate_regression(df_sim_2b, formula='y ~ 1 + u + w + x + z')

print('Method: relative bias (LB, UB)')
true_ate = calculate_true_ate(df_sim_2b)
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: 9.9% (-10.1%, 10.1%)
LR, one treatment: 10.0% (-10.1%, 10.1%)
LR, with U: -0.7% (-10.1%, 10.1%)
```

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

    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'][0]*100) )
```

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

    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 2b (high dim X) simulation results comparison:')
run_comparison(df_sim_2b, deltas, ['w','x','z'], ['u*','u','w','x','z'], dim_X␣
↪= dim_X)
```

```
Figure 2b (high dim X) simulation results comparison:
True empirical ATE is 1.92 p.p. Intended ATE is 2.00 p.p.
Method: relative bias (LB, UB)
LR, with obs. nodes: 9.9% (-10.1%, 10.1%)
LR, with obs. nodes + U: -0.7% (-10.1%, 10.1%)
LR, with all nodes: -0.7% (-10.1%, 10.1%)
Method: relative bias (condition number), true ATE
Negative controls: 0.0% (23), 1.92 p.p.
```

[9]:
```
## 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)
    dim_X = deltas['XY'].size
    regression_comparison_results = calculate_ate_regression(df, formula = 'y ~␣
↪1 + %s' % ' + '.join(obs_nodes),
                                                        dim_X = dim_X  )
    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_2b_high_dim_X, deltas,␣
↪['w','x','z'],
                                         n_comparison, 'Figure 2b (high dim X)')
```

```
Figure 2b (high dim X) simulation results over 100 comparisons:
LR, with obs. nodes: 10.0% +/- 1.8%
Negative controls:  -0.8% +/- 5.9%
```

0.3 Figure 2b, with high dimensionality U_* and X (Setup #7)

[10]:
```
# For a given set of parameters that specifies the DGP, return a dataframe of n␣
↪draws.
def simulate_UWXYZ_2b_high_dim_Ustar_X(deltas, phi=0.5, n=int(1e6) ):

    dim_X = len(deltas['XY'])
    dim_Ustar = len(deltas['U*U'])
```

```
    ## Draw U*
    if phi<0.0 or phi>1.0:
        print('phi is out of bounds.')
        return
    ustar_vec = np.random.binomial(n=1, p=phi, size=(n,dim_Ustar))
    df = pd.DataFrame(ustar_vec, columns=['ustar_%d'%i for i in␣
↪range(dim_Ustar)])

    ## Draw U, which is dependent on U*.
    probs = phi+np.dot((ustar_vec-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+np.dot((ustar_vec-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 = np.ones((n,dim_X))*phi+np.outer((df['u']-1.0/2), deltas['UX'])+np.
↪outer((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
    x_vec = np.random.binomial(n=1, p=probs)
    df = pd.concat([df,pd.DataFrame(x_vec, columns=['x_%d'%i for i in␣
↪range(dim_X)])], axis=1)
    df['x_state'] = np.dot(x_vec, [2**i for i in range(dim_X)])

    ## Draw Y, which is dependent on U*, U, W, and X
    probs = phi+np.dot((ustar_vec-1.0/2), deltas['U*Y'])+(df['u']-1.0/
↪2)*deltas['UY']+(df['w']-1.0/2)*deltas['WY']+np.dot((x_vec-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
dim_X = 1
dim_Ustar = 5
deltas = {'U*U':np.ones(dim_Ustar)*base_diff/dim_Ustar,
          'U*W':np.ones(dim_Ustar)*base_diff/dim_Ustar,
          'U*Y':np.ones(dim_Ustar)*base_diff/dim_Ustar,
          'UW':base_diff*2,
          'UX':np.ones(dim_X)*base_diff/dim_X,
          'UY':base_diff,
          'UZ':base_diff*2,
          'WY':base_diff,
          'XY':np.ones(dim_X)*base_diff/dim_X,
          'ZX':np.ones(dim_X)*base_diff/dim_X }
df_sim_2b = simulate_UWXYZ_2b_high_dim_Ustar_X(deltas = deltas)
print(df_sim_2b.groupby(['ustar_0','u','w','z','x_0']).mean()['y'].apply(lambda␣
 ↪x: np.round(x,3)).head(n=8))
print(df_sim_2b.head())
```

```
ustar_0  u  w  z  x_0
0        0  0  0  0      0.340
                  1      0.439
               1  0      0.338
                  1      0.438
            1  0  0      0.439
                  1      0.539
               1  0      0.434
                  1      0.540
Name: y, dtype: float64
   ustar_0  ustar_1  ustar_2  ustar_3  ustar_4  u  w  z  x_0  x_state  y
0        0        1        1        0        0  1  0  1    0        0  0
1        1        1        0        0        0  1  1  0    1        1  1
2        0        0        0        0        1  0  0  0    0        0  0
3        0        1        0        1        0  1  1  1    0        0  1
4        1        1        1        1        1  0  1  1    0        0  1
```

[11]:
```
print('Setup #7')
print('Figure 2b (high dim U*) simulation results comparison:')
run_comparison(df_sim_2b, deltas, ['w','x','z'], ['ustar_%d'%i for i in␣
 ↪range(dim_Ustar)]+['u','w','x','z'],
               dim_X = dim_X )
```

```
Setup #7
```

```
Figure 2b (high dim U*) simulation results comparison:
True empirical ATE is 10.04 p.p. Intended ATE is 10.00 p.p.
Method: relative bias (LB, UB)
LR, with obs. nodes: 9.4% (-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: 1.2% (26), 10.04 p.p.
```

[12]:
```
print('Setup #7')
run_generic_comparison_n_times_and_print(simulate_UWXYZ_2b_high_dim_Ustar_X,␣
↪deltas, ['w','x','z'],
                                          n_comparison, 'Figure 2b (high dim␣
↪U*)')
```

```
Setup #7
Figure 2b (high dim U*) simulation results over 100 comparisons:
LR, with obs. nodes: 9.5% +/- 0.4%
Negative controls:  -0.1% +/- 1.4%
```

0.4 Figure 2b, with high dimensionality U_*, X (Setup #8)

[13]:
```
base_diff = 0.10
dim_X = 5
dim_Ustar = 5
deltas = {'U*U':np.ones(dim_Ustar)*base_diff/dim_Ustar,
          'U*W':np.ones(dim_Ustar)*base_diff/dim_Ustar,
          'U*Y':np.ones(dim_Ustar)*base_diff/dim_Ustar,
          'UW':base_diff*2,
          'UX':np.ones(dim_X)*base_diff/dim_X,
          'UY':base_diff,
          'UZ':base_diff*2,
          'WY':base_diff,
          'XY':np.ones(dim_X)*base_diff/dim_X,
          'ZX':np.ones(dim_X)*base_diff/dim_X }
df_sim_2b = simulate_UWXYZ_2b_high_dim_Ustar_X(deltas = deltas)
print(df_sim_2b.groupby(['ustar_0','u','w','z','x_0']).mean()['y'].apply(lambda␣
↪x: np.round(x,3)).head(n=8))
print(df_sim_2b.head())
```

```
ustar_0  u  w  z  x_0
0        0  0  0  0      0.376
                  1      0.397
               1  0      0.379
                  1      0.395
            1  0  0      0.478
                  1      0.497
```

```
            1  0      0.484
               1      0.497
Name: y, dtype: float64
   ustar_0  ustar_1  ustar_2  ustar_3  ustar_4  u  w  z  x_0  x_1  x_2  x_3  \
0        1        1        1        0        1  0  1  1    0    1    0    0
1        0        0        0        1        0  0  0  1    1    1    0    0
2        0        0        0        1        1  0  0  0    0    0    0    0
3        1        1        1        1        0  1  1  0    0    1    0    1
4        1        0        0        1        0  1  1  1    0    0    0    1

   x_4  x_state  y
0    0        2  1
1    0        3  1
2    1       16  0
3    1       26  1
4    1       24  1
```

[14]:
```
print('Setup #8')
print('Figure 2b (high dim U*) simulation results comparison:')
run_comparison(df_sim_2b, deltas, ['w','x','z'], ['ustar_%d'%i for i in␣
 ↪range(dim_Ustar)]+['u','w','x','z'],
               dim_X = dim_X )
```

```
Setup #8
Figure 2b (high dim U*) simulation results comparison:
True empirical ATE is 1.97 p.p. Intended ATE is 2.00 p.p.
Method: relative bias (LB, UB)
LR, with obs. nodes: 10.3% (-9.9%, 9.9%)
LR, with obs. nodes + U: 0.0% (-9.8%, 9.8%)
LR, with all nodes: 0.4% (-9.8%, 9.8%)
Method: relative bias (condition number), true ATE
Negative controls: 0.7% (26), 1.97 p.p.
```

[15]:
```
print('Setup #8')
run_generic_comparison_n_times_and_print(simulate_UWXYZ_2b_high_dim_Ustar_X,␣
 ↪deltas, ['w','x','z'],
                                         n_comparison, 'Figure 2b (high dim␣
 ↪U*)')
```

```
Setup #8
Figure 2b (high dim U*) simulation results over 100 comparisons:
LR, with obs. nodes: 9.4% +/- 1.6%
Negative controls:  -0.7% +/- 6.0%
```