Probit

April 23, 2020

1 The Informativeness of Estimation Moments

This notebook illustrates the ideas in "*The Informativeness of Estimation Moments*" to appear in *Journal of Applied Econometrics* by Bo Honoré, Thomas H. Jørgensen and Áureo de Paula.

The code replicates the Probit example in that paper. Exact numbers differ due to the original implementation being in Matlab.

[1]:
```
# import packages
import numpy as np
import scipy.stats as sci
```

1.1 Define moment function used in estimation

[2]:
```
# moment function
def mom_funci(beta,y,x):

    residual = y-sci.norm.cdf(x @ beta);

    # allocate memory to store moments
    n,k = x.shape
    momi = np.nan + np.zeros((n,k*(k+1)//2))

    # loop through all elements in x and calcilate moments on individual level
    ii=0
    for i1 in range(k):
        for i2 in range(i1,k):
            momi[:,ii]=residual*x[:,i1]*x[:,i2]
            ii=ii+1

    return momi

def mom_func(beta,y,x):

    # return average moment
    momi = mom_funci(beta,y,x)
    return np.mean(momi,axis=0)
```

1.2 Simulate synthetic discrete choice data

[3]:
```
# number of simulations and seed
n = 10_000_000
np.random.seed(2020)

#setup the beta-parameters with desired varaince and covariance structure
rho = 0.5 # must be positive in program
r = np.sqrt(rho/(1.0-rho))
k = 3
beta = np.ones(k)/np.sqrt(2+2*rho)

# generate explanatory varaibles, x
x = np.random.normal(size=(n,k))
a = np.random.normal(size=n)

x[:,0]=np.ones(n)
x[:,1]=(x[:,1]+a*r)/np.sqrt(1+r*r)
x[:,2]=(x[:,2]+a*r)/np.sqrt(1+r*r)

# generate binary outcome
y = (x @ beta + np.random.normal(size=n)) > 0
```

1.3 Calculate required objects (S, G) at β

[4]:
```
# calcualte covaraince matrix of estimation moments
momi = mom_funci(beta,y,x)
S = np.cov(momi,rowvar=False)

# Calculate the numerical gradient of the objective function at beta
def num_grad(fun,theta,num_mom,step=1.0e-4,**kargs):
    # Calculate numerical gradient for all parameters
    num_par   = len(theta)
    grad = np.nan + np.zeros((num_mom,num_par))

    for i in range(num_par):
        var_now       = np.zeros(num_par)
        var_now[i]    = 1

        forward   = fun(theta+step*var_now,**kargs);
        backward = fun(theta-step*var_now,**kargs);

        grad[:,i]    = (forward-backward)/(2*step)

    return grad

grad = num_grad(mom_func,beta,len(S),step=1.0e-4,y=y,x=x)
```

1.4 Calculate Sensitivity Measures

[5]:
```
# sensitivity measures
def sensitivity(grad,S,W):

    sens = dict()

    # calculate objects re-used below
    GW        = grad.T @ W
    GWG       = GW @ grad
    GWG_inv   = np.linalg.inv(GWG)

    GSi  = grad.T @ np.linalg.inv(S)
    GSiG = GSi @ grad

    Avar    = GWG_inv @ (GW @ S @ GW.T) @ GWG_inv
    AvarOpt = np.linalg.inv(GSiG)

    # sensitivity measures
    sens['M1'] = - GWG_inv @ GW

    num_mom = len(S)
    num_par = len(grad[0])
    shape = (num_par,num_mom)
    sens['M2'] = np.nan + np.zeros(shape)
    sens['M3'] = np.nan + np.zeros(shape)
    sens['M4'] = np.nan + np.zeros(shape)
    sens['M5'] = np.nan + np.zeros(shape)
    sens['M6'] = np.nan + np.zeros(shape)

    sens['M2e'] = np.nan + np.zeros(shape)
    sens['M3e'] = np.nan + np.zeros(shape)
    sens['M4e'] = np.nan + np.zeros(shape)
    sens['M5e'] = np.nan + np.zeros(shape)
    sens['M6e'] = np.nan + np.zeros(shape)

    for k in range(num_mom):
        # pick out the kk'th element: Okk
        O      = np.zeros((num_mom,num_mom))
        O[k,k] = 1

        M2kk     = (np.linalg.inv(GSiG) @ (GSi @ O @ GSi.T)) @ np.linalg.
 ↪inv(GSiG)          # num_par-by-num_par
        M3kk     = GWG_inv @ (GW @ O @ GW.T) @ GWG_inv
        M6kk     =  - GWG_inv @ (grad.T@ O @ grad) @ Avar \
                    + GWG_inv @ (grad.T @ O @ S @ W @ grad) @ GWG_inv \
                    + GWG_inv @ (grad.T @ W @ S @ O @ grad) @ GWG_inv \
```

```
                    - Avar @ (grad.T @ O @ grad) @ GWG_inv   # NumPar-by-NumPar

        sens['M2'][:,k]  = np.diag(M2kk) # store only the diagonal: the effect␣
↪on the variance of a given parameter from a slight change in the variance of␣
↪the kth moment
        sens['M3'][:,k]  = np.diag(M3kk) # store only the diagonal: the effect␣
↪on the variance of a given parameter from a slight change in the variance of␣
↪the kth moment
        sens['M6'][:,k]  = np.diag(M6kk) # store only the diagonal: the effect␣
↪on the variance of a given parameter from a slight change in the variance of␣
↪the kth moment

        sens['M2e'][:,k]  = sens['M2'][:,k]/np.diag(AvarOpt) * S[k,k] # store␣
↪only the diagonal: the effect on the variance of a given parameter from a␣
↪slight change in the variance of the kth moment
        sens['M3e'][:,k]  = sens['M3'][:,k]/np.diag(Avar) * S[k,k]    # store␣
↪only the diagonal: the effect on the variance of a given parameter from a␣
↪slight change in the variance of the kth moment
        sens['M6e'][:,k]  = sens['M6'][:,k]/np.diag(Avar) * W[k,k]    #  store␣
↪only the diagonal: the effect on the variance of a given parameter from a␣
↪slight change in the variance of the kth moment

        # remove the kth moment from the weight matrix and
        # calculate the asymptotic variance without this moment
        W_now      = W.copy()
        W_now[k,:] = 0
        W_now[:,k] = 0

        GW_now   = grad.T@W_now
        GWG_now  = GW_now@grad
        Avar_now = (np.linalg.inv(GWG_now) @ (GW_now@S@GW_now.T)) @ np.linalg.
↪inv(GWG_now)

        sens['M4'][:,k]  = np.diag(Avar_now) - np.diag(Avar)
        sens['M4e'][:,k] = sens['M4'][:,k] / np.diag(Avar)

        # optimal version
        S_now = np.delete(S,k,axis=0)
        S_now = np.delete(S_now,k,axis=1)
        grad_now = np.delete(grad,k,axis=0)
        AvarOpt_now = np.linalg.inv((grad_now.T @ np.linalg.inv(S_now)) @␣
↪grad_now)
        sens['M5'][:,k]  = np.diag(AvarOpt_now) - np.diag(AvarOpt)
        sens['M5e'][:,k] = sens['M5'][:,k] / np.diag(AvarOpt)

    return sens
```

[6]:
```
# optimal weighting matrix
W_opt = np.linalg.inv(S)
sens_opt = sensitivity(grad,S,W_opt)

# alternative diagonal weigting matrix
W = np.linalg.inv(np.diag(np.diag(S)));
sens = sensitivity(grad,S,W)
```

[7]:
```
sens['M2e']
```

[7]:
```
array([[1.10315270e+00, 8.78792726e-02, 8.74665631e-02, 2.98777341e-03,
        4.27540262e-03, 2.72210840e-03],
       [5.88726908e-02, 1.20963174e+00, 4.59410115e-02, 3.04011423e-03,
        4.54749958e-04, 2.67789913e-04],
       [5.94956727e-02, 4.57366611e-02, 1.20702823e+00, 2.69201749e-04,
        4.83812415e-04, 2.72471938e-03]])
```

[8]:
```
sens['M3e']
```

[8]:
```
array([[0.651282  , 0.10085313, 0.1008842 , 0.08049781, 0.012906  ,
        0.08013231],
       [0.07040271, 0.81716034, 0.03589065, 0.03634127, 0.03443829,
        0.03898091],
       [0.07031687, 0.03581351, 0.81727943, 0.03892429, 0.03438723,
        0.036492  ]])
```

[9]:
```
sens['M6e']
```

[9]:
```
array([[-0.10116606,  0.00179543,  0.00209301,  0.03977895,  0.01711848,
         0.04038018],
       [ 0.01176726, -0.14319952,  0.00182328,  0.04395316,  0.04846761,
         0.03718821],
       [ 0.01160534,  0.00187896, -0.14255851,  0.03694306,  0.04837379,
         0.04375736]])
```

[10]:
```
sens['M4e']
```

[10]:
```
array([[ 1.0767247 ,  0.34324854,  0.34147116, -0.00994256, -0.01067718,
        -0.01077989],
       [ 0.0406604 ,  3.80560198,  0.11447516, -0.03860138, -0.03170452,
        -0.02825486],
       [ 0.0411031 ,  0.11382766,  3.80250227, -0.02804604, -0.03163221,
        -0.03818067]])
```

[11]:
```
sens['M5e']
```

[11]:
```
array([[1.20357111e+00, 2.93734864e-01, 2.92638645e-01, 1.51146490e-03,
        2.88056253e-03, 1.38124222e-03],
       [6.42317874e-02, 4.04317201e+00, 1.53705769e-01, 1.53794325e-03,
        3.06388849e-04, 1.35880972e-04],
       [6.49114785e-02, 1.52873955e-01, 4.03837870e+00, 1.36184689e-04,
        3.25969748e-04, 1.38256708e-03]])
```