Assignment 3

Submission for Aline Abayo, Eleanor Adachi, Anna Cheyette, and Karla Neri

Our team's code can be found here: https://github.com/eleanor-adachi/ARE212_Materials/assignment3

1. Exercises (GMM)

When we approach a new estimation problem from a GMM perspective there’s a simple set of steps we can follow.

- Describe the parameter space B;
- Describe a function $g_j(b)$ such that $\mathbb{E}g_j(\beta) = 0$;
- Describe an estimator for the covariance matrix $\mathbb{E}g_j(\beta)g_j(\beta)^\top$.

```
In [ ]: # import GMM_class for later

        import sys

        # setting path
        sys.path.append('../../')

        from GMM_class_EA import GMM
```

(1) Explain how the steps outlined above can be used to construct an optimally weighted GMM estimator.

The optimally weighted (efficient) GMM estimator is constructed as follows:

1. **Describe the Parameter Space B**: This step involves defining the set of parameters, denoted as β, that you want to estimate. The parameter space B represents all possible values for the parameters of interest, i.e., $\beta \in B$.

2. **Describe a Function $g_j(\beta)$**: For each moment condition, represented by the function $g_j(\beta)$, you define a function that depends on the parameters of interest (denoted as β). The key aspect here is that the expected value of each moment condition should be equal to zero. Mathematically, it's represented as $\mathbb{E}g_j(\beta) = 0$, where β represents the true parameter values.

3. **Describe an Estimator for the Covariance Matrix**: To construct an optimally weighted GMM estimator b_{GMM}, you need to estimate the covariance matrix of the moment conditions, $\Omega = \mathbb{E}g_j(\beta)g_j(\beta)^T$. In practice, since $\mathbb{E}g_j(b) \neq 0$ if the model is overidentified ($\ell > k$), then $\hat{\Omega} = \mathbb{E}g_j(b)g_j(b)^T - \mathbb{E}g_j(b)\mathbb{E}g_j(b)^T$

4. **Optimally Weighted GMM Estimator**: With the moment conditions and their covariance matrix defined, you can construct the GMM estimator. The GMM estimator is then calculated as:

$$b_N = \underset{b \in B}{\operatorname{argmin}}\, g_N(b)^T A g_N(b)$$

where b represents the initial guess for the parameter vector, A is the optimal weighting matrix such that $A = \Omega^{-1}$, and $g_j(b)$ are the moment conditions evaluated at the estimated parameters and $g_N(b) = \frac{1}{N}\sum_{j=1}^N g_j(b)$.

The goal is to find the weighting matrix A that minimizes the variance of b_N. By choosing the weighting matrix optimally, the GMM estimator can achieve efficiency and robustness in estimating the parameters of interest. The weights are chosen such that more reliable moment conditions contribute more to the estimation process, leading to improved parameter estimates.

The asymptotic variance of the optimally weighted GMM estimator will be

$$V_b = \left(Q^\top \Omega^{-1} Q\right)^{-1}.$$

where $Q = \mathbb{E}\frac{\partial g_j}{\partial b^\top}(b)$

(2) Consider the following models. For each, provide a causal diagram; construct the optimally weighted GMM estimator of the unknown parameters (various Greek letters); and give an estimator for the covariance matrix of your estimates. If any additional assumptions are required for your estimator to be identified please provide these.

(a) $\mathbb{E}y = \mu; \mathbb{E}(y-\mu)^2 = \sigma^2; \mathbb{E}(y-\mu)^3 = 0$. *Note: y is a random variable.*

See https://math.stackexchange.com/questions/92648/calculation-of-the-n-th-central-moment-of-the-normal-distribution-mathcaln

See causal diagram below.

First, write the equations for g_j and solve for $\hat{\Omega}$ and Q. Let b represent a vector of parameter estimates, $(\hat{\mu}, \hat{\sigma}^2)$.

$$g_j(\mu, \sigma^2) = \left(\begin{array}{c} y_j - \mu \\ (y_j-\mu)^2 - \sigma^2 \\ (y_j - \mu)^3 \end{array} \right.$$

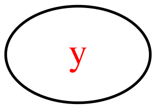
$$\hat{\Omega} = \mathbb{E}g_j(b)g_j(b)^T - \mathbb{E}g_j(b)\mathbb{E}g_j(b)^T$$

$$Q = \mathbb{E}\frac{\partial g_j}{\partial b^\top}(b) = \begin{bmatrix} -1 & 0 \\ \mathbb{E}(-2(y-\mu)) & -1 \\ \mathbb{E}(-3(y-\mu)^2) & -1 \end{bmatrix}$$
$$= \begin{bmatrix} -1 & 0 \\ 0 & -1 \\ -3\sigma^2 & -1 \end{bmatrix}$$

Plug into GMM estimator and asymptotic variance of GMM estimator:

$$b = \operatorname*{argmin}_{b \in B} g_N(b)^T \Omega^{-1} g_N(b)$$
$$V_b = \left(Q^\top \Omega^{-1} Q\right)^{-1}.$$

In [ ]:
```python
import graphviz as gr

g = gr.Digraph()
g.node('y', fontcolor='red')

g
```

Out[ ]:


(b) $y = \alpha + X\beta + u$; with $\mathbb{E}(X^T u) = \mathbb{E}u = 0$. *Note: y, X, and u are random variables.*

See causal diagram below.

First, write the equations for g_j and solve for $\hat{\Omega}$ and Q. Let b represent a vector of parameter estimates, $(\hat{\alpha}, \hat{\beta})$.

Assume that $u \sim N[0, \Omega]$.

$$g_j(\alpha, \beta) = \left(\begin{array}{c} y_j - \alpha - X_j\beta \\ X_j^T(y_j - \alpha - X_j\beta) \end{array} \right.$$

$$\hat{\Omega} = \mathbb{E}g_j(b)g_j(b)^T - \mathbb{E}g_j(b)\mathbb{E}g_j(b)^T$$

$$Q = \mathbb{E}\frac{\partial g_j}{\partial b^\top}(b) = \begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix}$$

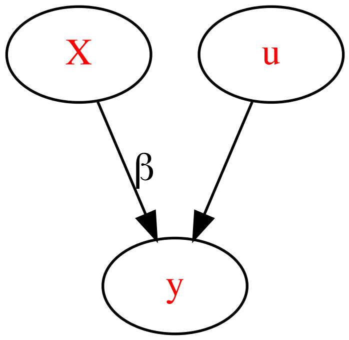
Plug into GMM estimator and asymptotic variance of GMM estimator:

$$b = \underset{b \in B}{\operatorname{argmin}}\, g_N(b)^T \Omega^{-1} g_N(b)$$

$$V_b = \left(Q^\top \Omega^{-1} Q\right)^{-1}.$$

In [ ]:
```python
import graphviz as gr

g = gr.Digraph()
g.node('X', fontcolor='red')
g.node('u', fontcolor='red')
g.node('y', fontcolor='red')
g.edge('u', 'y')
g.edge('X', 'y', label='β')

g
```

Out[ ]:



(c) $y = \alpha + X\beta + u$; with $\mathbb{E}(X^T u) = \mathbb{E}u = 0$, and $\mathbb{E}(u^2) = \sigma^2$. *Note: y, X, and u are random variables.*

Same causal diagram as (b).

First, write the equations for g_j and solve for $\hat{\Omega}$ and Q. Let b represent a vector of parameter estimates, $(\hat{\alpha}, \hat{\beta})$.

$$g_j(\alpha,\beta) = \left(\begin{array}{c} y_j - \alpha - X_j\beta \\ X_j^T(y_j - \alpha - X_j\beta) \\ (y_j - \alpha - X_j\beta)^2 - \sigma^2 \end{array}\right.$$

$$\hat{\Omega} = \mathbb{E}g_j(b)g_j(b)^T - \mathbb{E}g_j(b)\mathbb{E}g_j(b)^T$$

$$Q = \mathbb{E}\frac{\partial g_j}{\partial b^\top}(b) = \begin{bmatrix} -1 & -1 \\ -1 & -1 \\ 2\alpha - 2y_j + X_j\beta & 2X_j^T X_j\beta - 2y_jX_j + 2\alpha X_j \end{bmatrix}$$

Plug into GMM estimator and asymptotic variance of GMM estimator:

$$b = \underset{b\in B}{\operatorname{argmin}}\, g_N(b)^T\Omega^{-1}g_N(b)$$

$$V_b = \left(Q^\top\Omega^{-1}Q\right)^{-1}.$$

(d) $y = \alpha + X\beta + u$; with $\mathbb{E}(X^Tu) = \mathbb{E}u = 0$, and $\mathbb{E}(u^2) = e^{X\sigma}$. *Note: y, X, and u are random variables.*

See causal diagram below.

First, write the equations for g_j and solve for $\hat{\Omega}$ and Q. Let b represent a vector of parameter estimates, $(\hat{\alpha}, \hat{\beta})$.

$$g_j(\alpha,\beta) = \left(\begin{array}{c} y_j - \alpha - X_j\beta \\ X_j^T(y_j - \alpha - X_j\beta) \\ (y_j - \alpha - X_j\beta)^2 - e^{X\sigma} \end{array}\right.$$

$$\hat{\Omega} = \mathbb{E}g_j(b)g_j(b)^T - \mathbb{E}g_j(b)\mathbb{E}g_j(b)^T$$

$$Q = \mathbb{E}\frac{\partial g_j}{\partial b^\top}(b)$$

Plug into GMM estimator and asymptotic variance of GMM estimator:

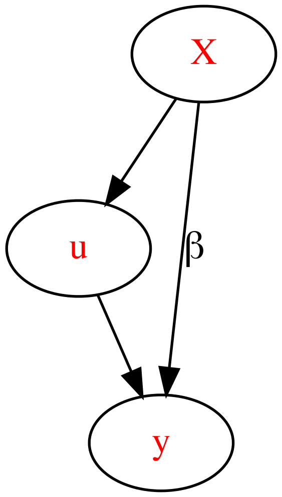
$$b = \underset{b\in B}{\operatorname{argmin}}\, g_N(b)^T\Omega^{-1}g_N(b)$$

$$V_b = \left(Q^\top\Omega^{-1}Q\right)^{-1}.$$

In [ ]:
```python
import graphviz as gr

g = gr.Digraph()
```

```
g.node('X', fontcolor='red')
g.node('u', fontcolor='red')
g.node('y', fontcolor='red')
g.edge('u', 'y')
g.edge('X', 'u')
g.edge('X', 'y', label='β')

g
```

Out[ ]:



(e) $y = \alpha + X\beta + u$; with $\mathbb{E}(Z^\top u) = \mathbb{E}u = 0$ and $\mathbb{E}Z^\top X = Q$. *Note: y, X, Z, and u are random variables.*

See causal diagram below.

Assume that Q is a positive definite matrix.

First, write the equations for g_j and solve for $\hat{\Omega}$ and Q. Let b represent a vector of parameter estimates, $(\hat{\alpha}, \hat{\beta})$.

$$g_j(\alpha, \beta) = Z_j^T \left(y_j - \alpha - X_j\beta\right)$$

Note that g_j is $\ell \times 2$, where ℓ is the number of instruments in Z.

$$\hat{\Omega} = \mathbb{E}g_j(b)g_j(b)^T - \mathbb{E}g_j(b)\mathbb{E}g_j(b)^T$$

$$Q = \mathbb{E}\frac{\partial g_j}{\partial b^\top}(b)$$

Plug into GMM estimator and asymptotic variance of GMM estimator:

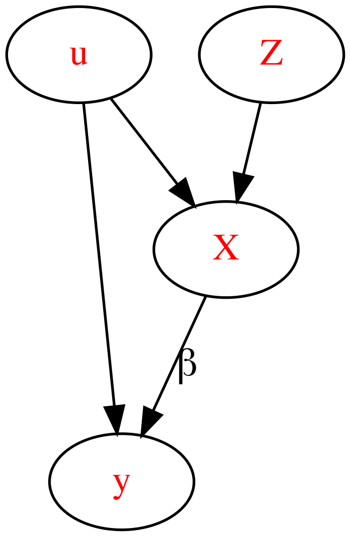
$$b = \underset{b \in B}{\operatorname{argmin}}\, g_N(b)^T \Omega^{-1} g_N(b)$$

$$V_b = \left(Q^\top \Omega^{-1} Q\right)^{-1}.$$

In [ ]:
```
import graphviz as gr
```

```
g = gr.Digraph()
g.node('X', fontcolor='red')
g.node('Z', fontcolor='red')
g.node('u', fontcolor='red')
g.node('y', fontcolor='red')
g.edge('u', 'y')
g.edge('u', 'X')
g.edge('Z', 'X')
g.edge('X', 'y', label='β')

g
```

Out[ ]:



(f) $y = f(X\beta) + u$; with f a known scalar function and with $\mathbb{E}(Z^\top u) = \mathbb{E}u = 0$ and $\mathbb{E}Z^\top X f'(X\beta) = Q(\beta)$. (Bonus question: where does this last restriction come from, and what role does it play?) *Note: y, X, Z, and u are random variables.*

See causal diagram below.

Assume that Q is a positive definite matrix.

First, write the equations for g_j and solve for $\hat{\Omega}$ and Q.

$$g_j(\beta) = Z_j^T\left(y_j - f(X_j\beta)\right)$$

$$\hat{\Omega} = \mathbb{E}g_j(b)g_j(b)^T - \mathbb{E}g_j(b)\mathbb{E}g_j(b)^T$$

$$Q = \mathbb{E}\frac{\partial g_j}{\partial b^\top}(b)$$

Plug into GMM estimator and asymptotic variance of GMM estimator:

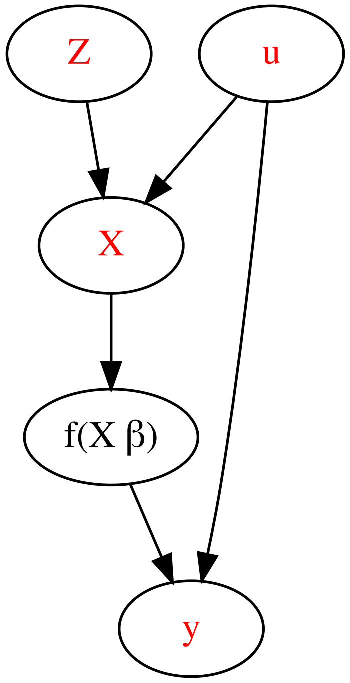
$$b = \operatorname*{argmin}_{b\in B} g_N(b)^T\Omega^{-1}g_N(b)$$

$$V_b = \left(Q^\top\Omega^{-1}Q\right)^{-1}.$$

In the context of IV, the restriction of $\mathbb{E}Z^\top X f'(X\beta) = Q(\beta)$ is used to ensure that the instruments in Z are valid and strong in the sense that they are sufficiently correlated with the transformed predictors $Xf'(X\beta)$, making the instruments relevant and providing a basis for consistent estimation of β.

In [ ]:
```python
import graphviz as gr

g = gr.Digraph()
g.node('X', fontcolor='red')
g.node('Z', fontcolor='red')
g.node('f(X β)')
g.node('u', fontcolor='red')
g.node('y', fontcolor='red')
g.edge('X', 'f(X β)')
g.edge('u', 'y')
g.edge('u', 'X')
g.edge('Z', 'X')
g.edge('f(X β)', 'y')

g
```

Out[ ]:



(g) $y = f(X, \beta) + u$; with f a known function and with $\mathbb{E}(Z^\top u) = \mathbb{E}u = 0$ and $\mathbb{E}Z^\top \frac{\partial f}{\partial \beta^T}(X, \beta) = Q(\beta)$. *Note: y, X, Z, and u are random variables.*

Same causal diagram as (f).

Assume that Q is a positive definite matrix.

First, write the equations for g_j and solve for $\hat{\Omega}$ and Q.

$$g_j(\beta) = Z_j^T\left(y_j - f(X_j, \beta)\right)$$

$$\hat{\Omega} = \mathbb{E}g_j(b)g_j(b)^T - \mathbb{E}g_j(b)\mathbb{E}g_j(b)^T$$

$$Q = \mathbb{E}\frac{\partial g_j}{\partial b^\top}(b)$$

Plug into GMM estimator and asymptotic variance of GMM estimator:

$$b = \underset{b \in B}{\operatorname{argmin}}\, g_N(b)^T \Omega^{-1} g_N(b)$$

$$V_b = \left(Q^\top \Omega^{-1} Q\right)^{-1}.$$

(h) $y^\gamma = \alpha + u$, with $y > 0$ and γ a scalar, and $\mathbb{E}(Z^\top u) = \mathbb{E}u = 0$ and $\mathbb{E}Z^\top \begin{bmatrix}\gamma y^{\gamma-1} \\ -1\end{bmatrix} = Q(\gamma)$. *Note: y, Z, and u are random variables.*

See causal diagram below.

Assume that Q is a positive definite matrix.

First, write the equations for g_j and solve for $\hat{\Omega}$ and Q.

$$g_j(\alpha) = Z_j^T\left(y^\gamma - \alpha\right)$$

$$\hat{\Omega} = \mathbb{E}g_j(b)g_j(b)^T - \mathbb{E}g_j(b)\mathbb{E}g_j(b)^T$$

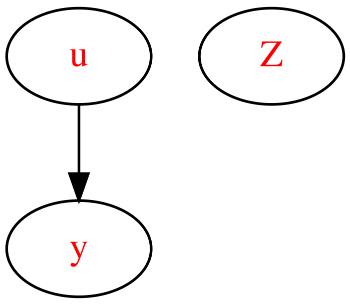
$$Q = \mathbb{E}\frac{\partial g_j}{\partial b^\top}(b)$$

Plug into GMM estimator and asymptotic variance of GMM estimator:

$$b = \underset{b \in B}{\operatorname{argmin}}\, g_N(b)^T \Omega^{-1} g_N(b)$$

$$V_b = \left(Q^\top \Omega^{-1} Q\right)^{-1}.$$

In [ ]:
```
import graphviz as gr

g = gr.Digraph()
g.node('u', fontcolor='red')
g.node('Z', fontcolor='red')
g.node('y', fontcolor='red')
g.edge('u', 'y')

g
```

Out[ ]: 

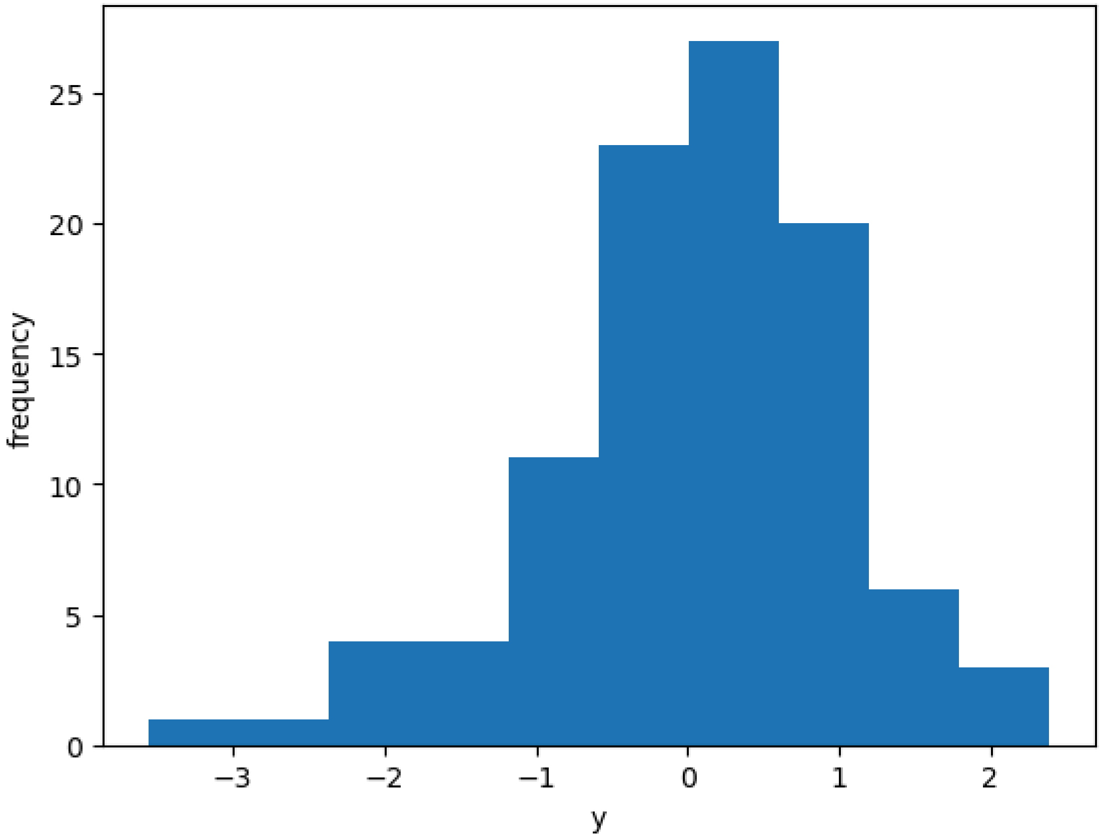
(3) For each of the models above write a data-generating process in python. Your function `dgp` should take as arguments a sample size N and a vector of "true" parameters `b0`, and return a dataset (y, X).

(a) $\mathbb{E}y = \mu; \mathbb{E}(y-\mu)^2 = \sigma^2; \mathbb{E}(y-\mu)^3 = 0$. *Note: y is a random variable.*

In [ ]:
```python
import numpy as np
import pandas as pd
from scipy.stats import distributions as iid

def dgp_a(N, b0):
    '''
    Generate dataset for GMM model A

    Takes as inputs sample size N and vector of true parameters (mu and sigm

    Returns a tuple with numpy array y, all of length N
    '''
    # Retrieve true parameters
    mu = b0[0]
    sigma = b0[1]

    # Construct u
    u = iid.norm(loc=mu, scale=sigma)

    # Construct y
    y = u.rvs(N)

    # Store in DataFrame
    df = pd.DataFrame(columns=['y'])
    df['y'] = y
    return df[['y']]
```

In [ ]:
```python
np.random.seed(1234)

y = dgp_a(100, [0, 1])
```

In [ ]:
```python
import matplotlib.pyplot as plt

plt.hist(y)
plt.xlabel('y')
plt.ylabel('frequency')
plt.show()
```



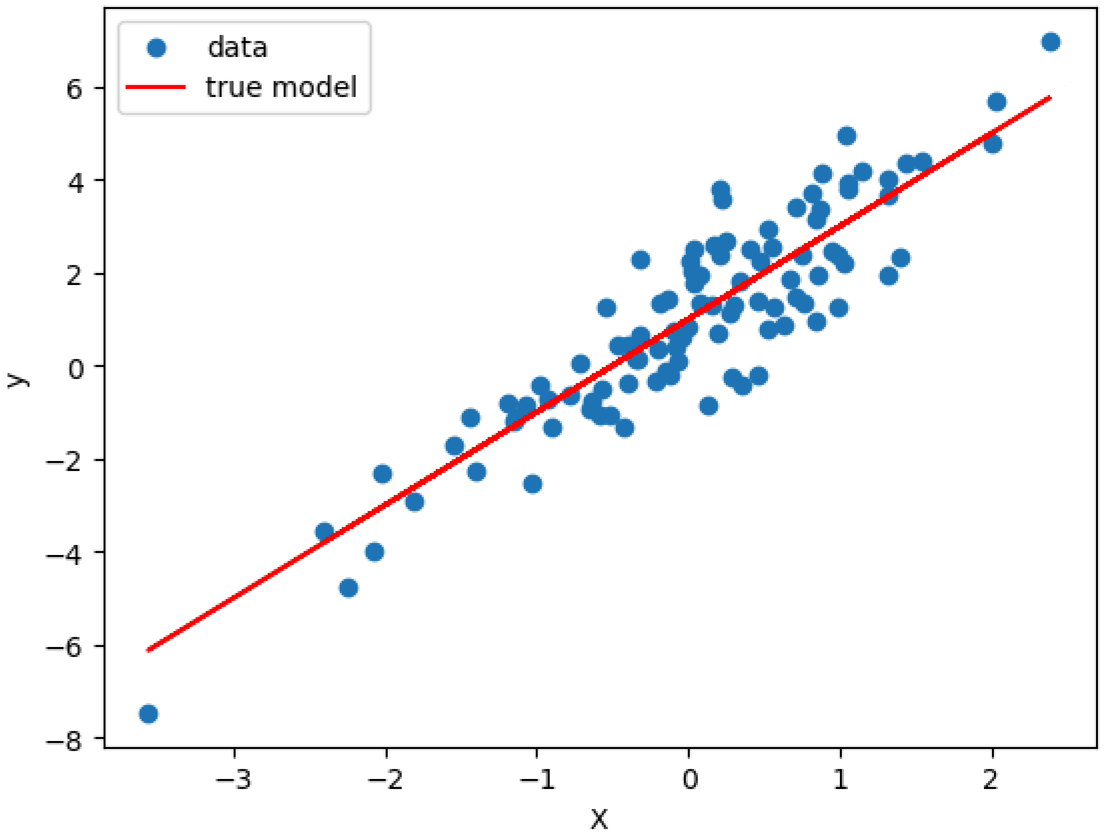
(b) $y = \alpha + X\beta + u$; with $\mathbb{E}(X^T u) = \mathbb{E}u = 0$. *Note: y, X, and u are random variables.*

```
In [ ]: import numpy as np
        import pandas as pd
        from scipy.stats import distributions as iid

        # Structural parameters;
        mu = 0
        sigma = 1

        # Construct u
        u = iid.norm(loc=mu, scale=sigma)

        def dgp_b(N, b0):
            '''
            Generate dataset for GMM model B

            Takes as inputs sample size N and vector of true parameters (alpha and b

            Returns a tuple with numpy arrays y and X, all of length N
            '''
            # Retrieve true parameters
            alpha = b0[0]
            beta = b0[1]
```

```
    # Construct X
    X = np.random.normal(size=N)

    # Construct y
    y = alpha + X*beta + u.rvs(N)

    # Store in DataFrame
    df = pd.DataFrame(columns=['y', 'X'])
    df['y'] = y
    df['X'] = X
    return df[['y']], df[['X']]
```

In [ ]:
```
np.random.seed(1234)

alpha = 1
beta = 2
N = 100

y, X = dgp_b(N, [alpha, beta])
```

In [ ]:
```
import matplotlib.pyplot as plt

y_true = alpha + X*beta

plt.scatter(X, y, label='data')
plt.plot(X, y_true, label='true model', color='r')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```



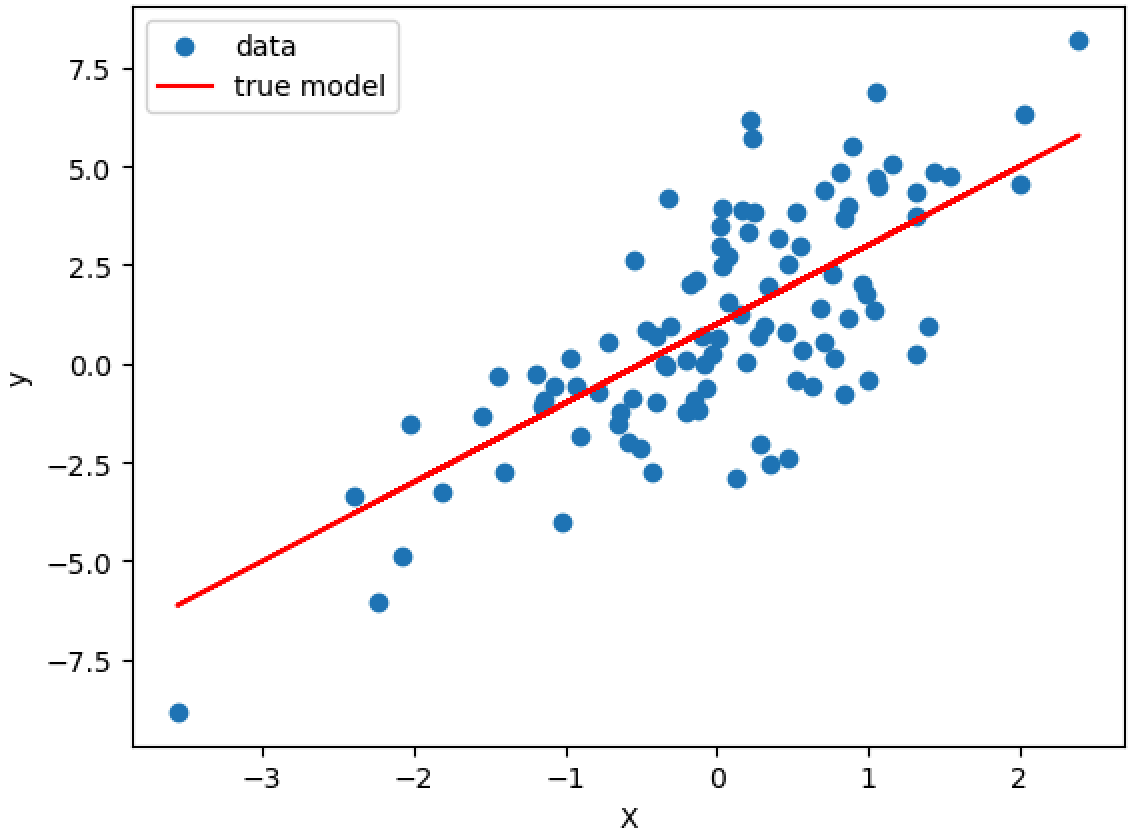
(c) $y = \alpha + X\beta + u$; with $\mathbb{E}(X^T u) = \mathbb{E}u = 0$, and $\mathbb{E}(u^2) = \sigma^2$. *Note: y, X, and u are random variables.*

```
In [ ]: import numpy as np
        import pandas as pd
        from scipy.stats import distributions as iid

        def dgp_c(N, b0):
            '''
            Generate dataset for GMM model C

            Takes as inputs sample size N and vector of true parameters (alpha, beta

            Returns a tuple with numpy arrays y and X, all of length N
            '''
            # Retrieve true parameters
            alpha = b0[0]
            beta = b0[1]
            sigma = b0[2]

            # Construct u
            u = iid.norm(loc=0, scale=sigma)

            # Construct X
            X = np.random.normal(size=N)

            # Construct y
            y = alpha + X*beta + u.rvs(N)
```

```
    # Store in DataFrame
    df = pd.DataFrame(columns=['y', 'X'])
    df['y'] = y
    df['X'] = X
    return df[['y']], df[['X']]
```

In [ ]:
```
np.random.seed(1234)

alpha = 1
beta = 2
sigma = 2
N = 100

y, X = dgp_c(N, [alpha, beta, sigma])
```

In [ ]:
```
import matplotlib.pyplot as plt

y_true = alpha + X*beta

plt.scatter(X, y, label='data')
plt.plot(X, y_true, label='true model', color='r')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```

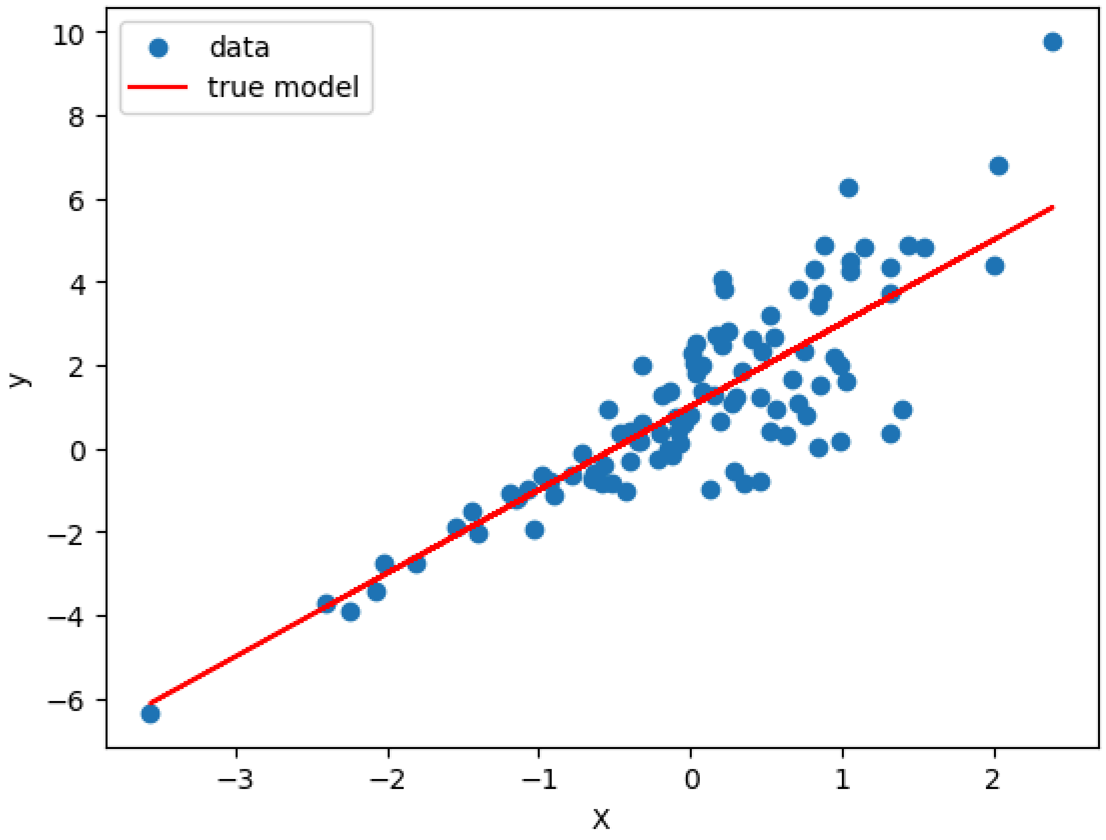


(d) $y = \alpha + X\beta + u$; with $\mathbb{E}(X^T u) = \mathbb{E}u = 0$, and $\mathbb{E}(u^2) = e^{X\sigma}$. *Note: y, X, and u are random variables.*

In [ ]:
```
import numpy as np
import pandas as pd
from scipy.stats import distributions as iid

def dgp_d(N, b0, sigma=1):
    '''
    Generate dataset for GMM model D

    Takes as inputs sample size N and vector of true parameters (alpha and b

    Returns a tuple with numpy arrays y and X, all of length N
    '''
    # Retrieve true parameters
    alpha = b0[0]
    beta = b0[1]

    # Construct X
    X = np.random.normal(size=N)

    # Construct u
    u = np.random.normal(size=N, loc=0, scale=np.sqrt(np.exp(X**sigma)))

    # Construct y
    y = alpha + X*beta + u

    # Construct df
    combined_dict = {'y':y, 'x':X, 'Constant':1}
    df = pd.DataFrame(combined_dict)

    return df['y'], df[['Constant','x']]
```

In [ ]:
```
np.random.seed(1234)

alpha = 1
beta = 2
sigma = 1
N = 100

y, X = dgp_d(N, [alpha, beta, sigma])
```

In [ ]:
```
import matplotlib.pyplot as plt

y_true = X@[alpha, beta]

plt.scatter(X['x'], y, label='data')
plt.plot(X['x'], y_true, label='true model', color='r')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```



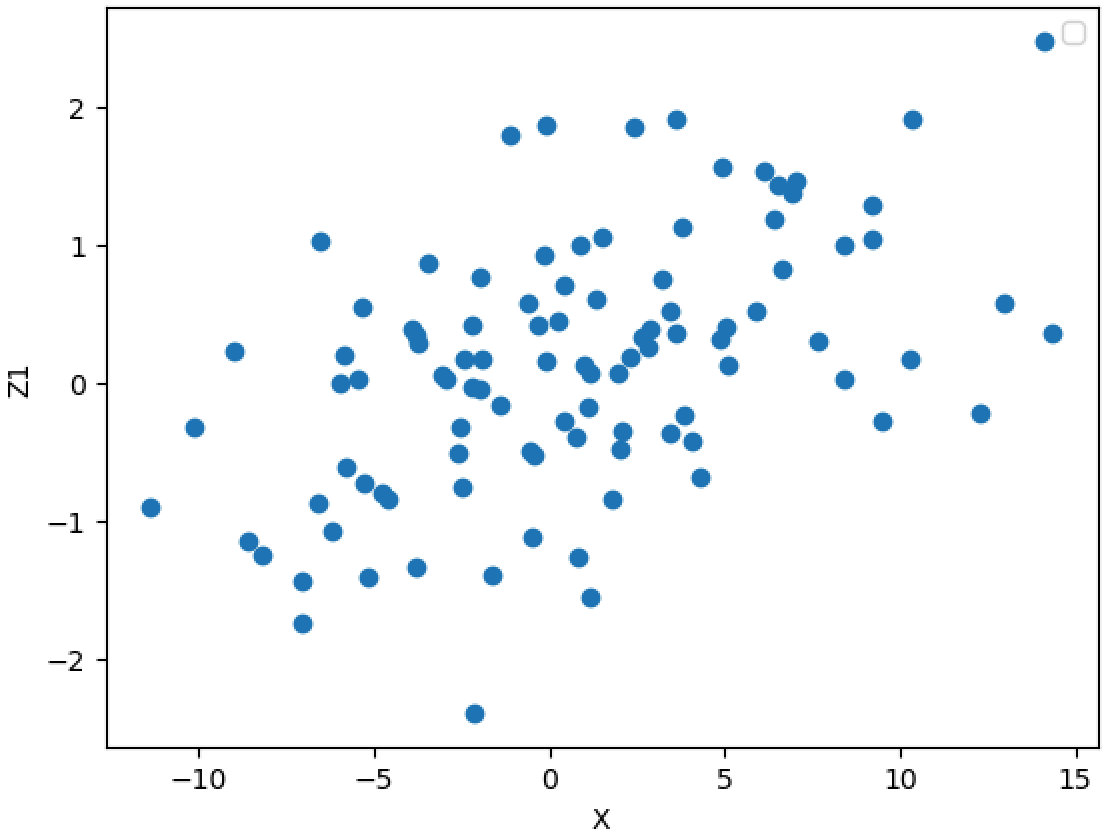
(e) $y = \alpha + X\beta + u$; with $\mathbb{E}(Z^\top u) = \mathbb{E}u = 0$ and $\mathbb{E}Z^\top X = Q$. *Note: y, X, Z, and u are random variables.*

In [ ]:
```
import numpy as np
import pandas as pd
from scipy.stats import distributions as iid

# 2 instruments
def dgp_e(N, b0, pi=[3,4], sigma_u=2, sigma_v=3):
    '''
    Generate dataset for GMM model E

    Takes as inputs sample size N and vector of true parameters (alpha and b

    Returns a tuple with numpy arrays y, X and Z, all of length N
    '''
    # Retrieve true parameters
    alpha = b0[0]
    beta = b0[1]

    # Construct random variables
    u = iid.norm(scale=sigma_u).rvs(N)
    v = iid.norm(scale=sigma_v).rvs(N)
    Z = iid.norm().rvs(size=(N,len(pi)))

    # Construct X and y
    X = Z@pi + v
    y = X*beta + u
    # Construct z_cols and z_dict
    z_cols = []
    z_dict = {}
    for i in range(len(pi)):
        z_col = 'z'+str(i+1)
        z_cols.append(z_col)
        z_dict[z_col] = Z[:, i]

    # Construct df
    combined_dict = {'y':y, 'x':X, 'Constant':1}
    combined_dict.update(z_dict)
    df = pd.DataFrame(combined_dict)

    return df['y'], df[['Constant','x']], df[['Constant']+z_cols] # y as ser
```

In [ ]:
```
np.random.seed(1234)

alpha = 1
beta = 2
N = 100

y, X, Z = dgp_e(N, [alpha, beta], pi=[3,4])
```

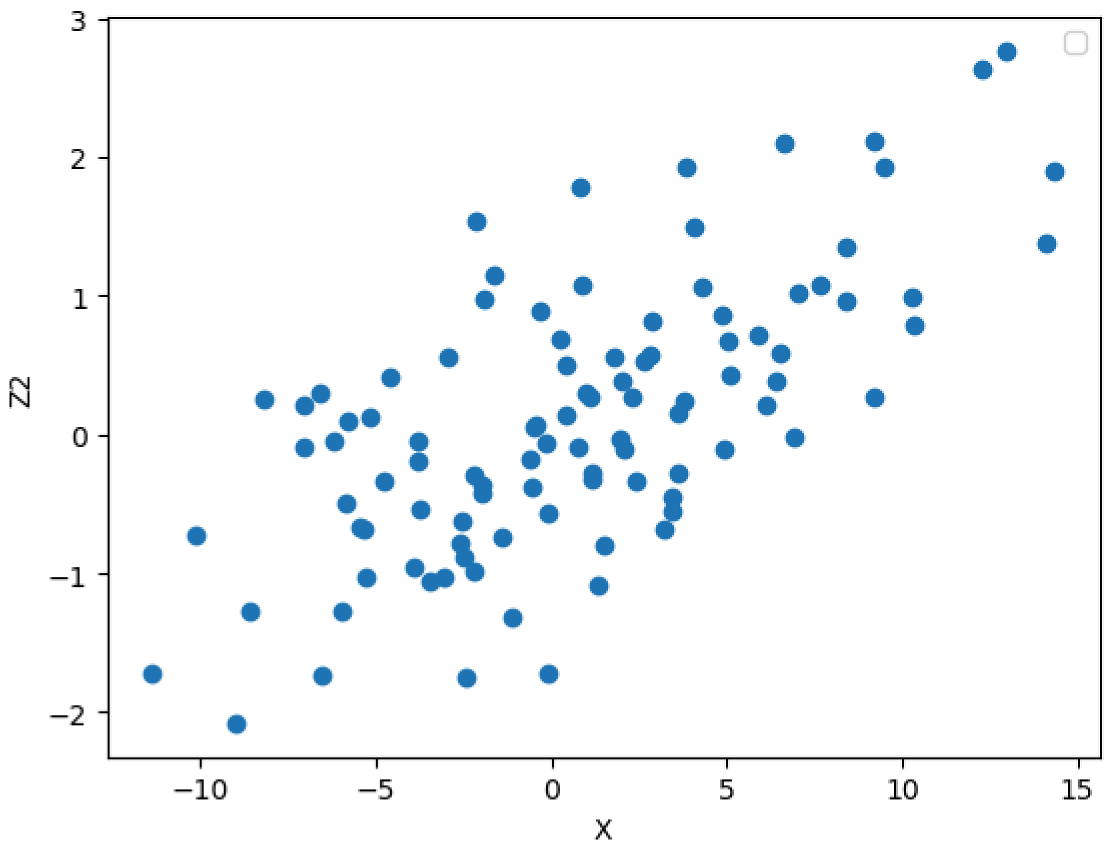
In [ ]:
```
import matplotlib.pyplot as plt

plt.scatter(X['x'], Z['z1'])
plt.xlabel('X')
plt.ylabel('Z1')
plt.legend()
plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose labe
l start with an underscore are ignored when legend() is called with no argum
ent.
```

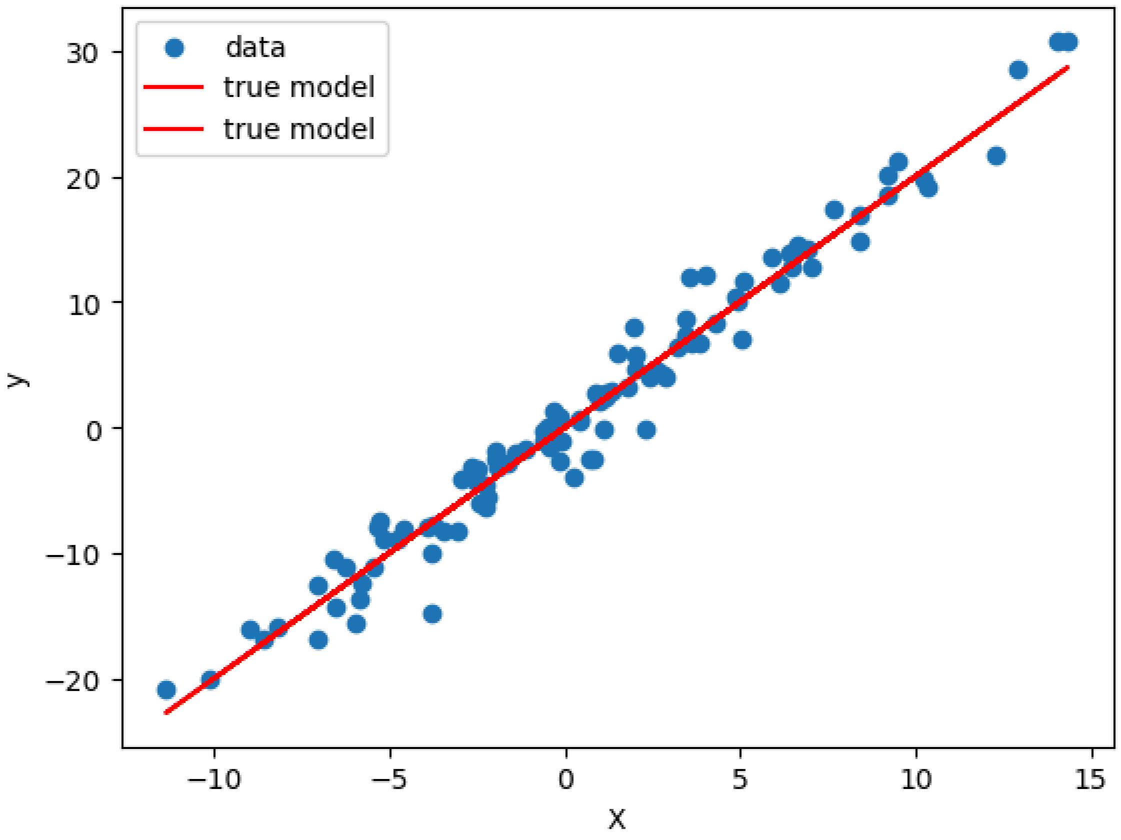


```
In [ ]: import matplotlib.pyplot as plt

        plt.scatter(X['x'], Z['z2'])
        plt.xlabel('X')
        plt.ylabel('Z2')
        plt.legend()
        plt.show()
```

```
No artists with labels found to put in legend.  Note that artists whose labe
l start with an underscore are ignored when legend() is called with no argum
ent.
```



```
In [ ]: import matplotlib.pyplot as plt

        y_true = X*beta

        plt.scatter(X['x'], y, label='data')
        plt.plot(X, y_true, label='true model', color='r')
        plt.xlabel('X')
        plt.ylabel('y')
        plt.legend()
        plt.show()
```



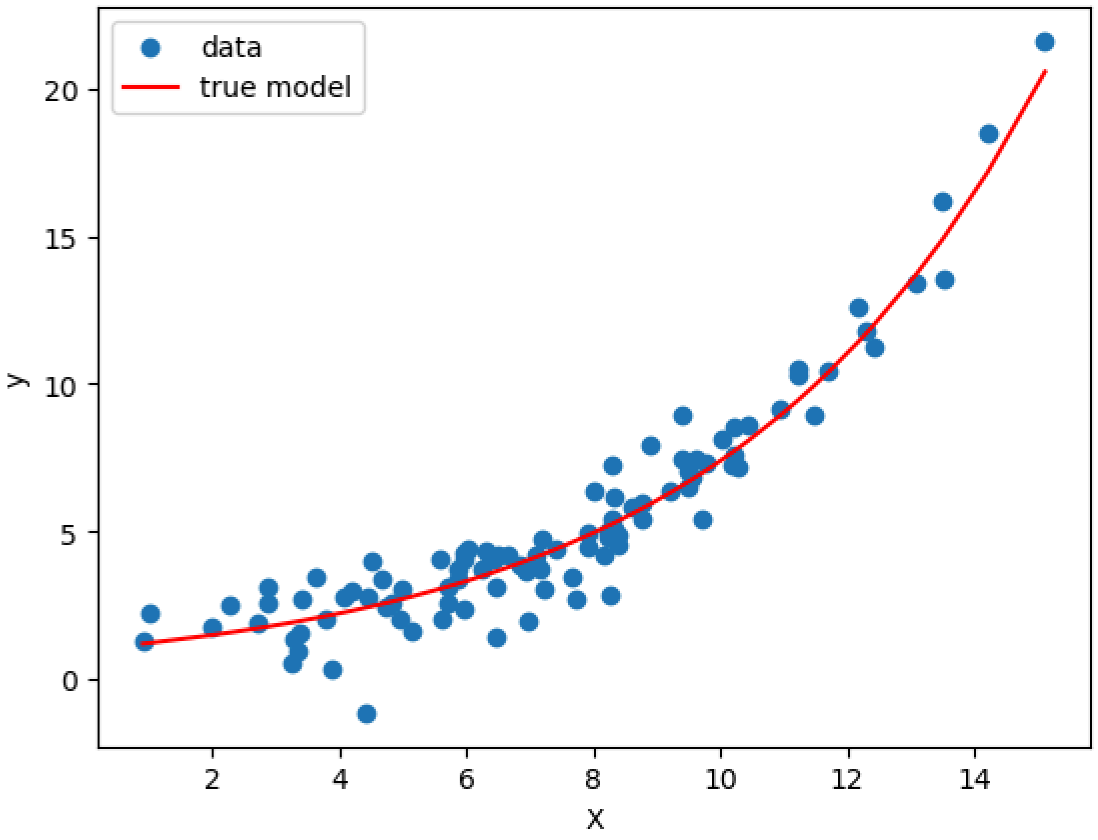
(f) $y = f(X\beta) + u$; with f a known scalar function and with $\mathbb{E}(Z^\top u) = \mathbb{E}u = 0$ and $\mathbb{E}Z^\top X f'(X\beta) = Q(\beta)$. *Note: y, X, Z, and u are random variables.*

```
In [ ]: def dgp_f(N, b0, pi=[3,4], sigma_u=1, sigma_v=2):
            '''
            Generate dataset for GMM model F

            Takes as inputs sample size N and vector of true parameters (alpha and b

            Returns a tuple with numpy arrays y, X and Z, all of length N
            '''
            # Retrieve true parameters
            alpha = b0[0]
            beta = b0[1]

            # Construct random variables
            u = iid.norm(scale=sigma_u).rvs(N)
            v = iid.norm(scale=sigma_v).rvs(N)
            Z = np.random.normal(size=(N,len(pi)), loc=1, scale=0.5)

            # Construct X and y
            X = Z@pi + v
            y = alpha*np.exp(0.1*beta*X) + u

            # Construct z_cols and z_dict
            z_cols = []
            z_dict = {}
```

```
    for i in range(len(pi)):
        z_col = 'z'+str(i+1)
        z_cols.append(z_col)
        z_dict[z_col] = Z[:, i]

    # Construct df
    combined_dict = {'y':y, 'x':X, 'Constant':1}
    combined_dict.update(z_dict)
    df = pd.DataFrame(combined_dict)

    return df['y'], df[['Constant','x']], df[['Constant']+z_cols] # y as ser
```

In [ ]:
```
np.random.seed(1234)

alpha = 1
beta = 2
N = 100

y, X, Z = dgp_f(N, [alpha, beta], pi=[3,4])
```

In [ ]:
```
import matplotlib.pyplot as plt

y_true = alpha*np.exp(0.1*beta*X['x'])
true_xy = pd.DataFrame(columns=['x', 'y'])
true_xy['x'] = X['x']
true_xy['y'] = y_true
true_xy = true_xy.sort_values('x')

plt.scatter(X['x'], y, label='data')
plt.plot(true_xy['x'], true_xy['y'], label='true model', color='r')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```



(g) $y = f(X, \beta) + u$; with f a known function and with $\mathbb{E}(Z^\top u) = \mathbb{E}u = 0$ and $\mathbb{E}Z^\top \frac{\partial f}{\partial \beta^T}(X, \beta) = Q(\beta)$. *Note: y, X, Z, and u are random variables.*

Let $f(X) = \alpha + \beta x_1 + \gamma x_2^2$

```
In [ ]: def dgp_g(N, b0, pi=[1,2,3,4], sigma_u=1, sigma_v=2):
            '''
            Generate dataset for GMM model G

            Takes as inputs sample size N and vector of true parameters (alpha, beta

            Returns a tuple with numpy arrays y, X and Z, all of length N
            '''
            # Retrieve true parameters
            alpha = b0[0]
            beta = b0[1]
            gamma = b0[2]

            # Construct random variables
            u = iid.norm(scale=sigma_u).rvs(N)
            v = iid.norm(scale=sigma_v).rvs(N)
            Z = np.random.normal(size=(N,len(pi)), loc=1, scale=0.5)

            # Construct X
            X1 = 5 + v
            X2 = Z@pi + v
```

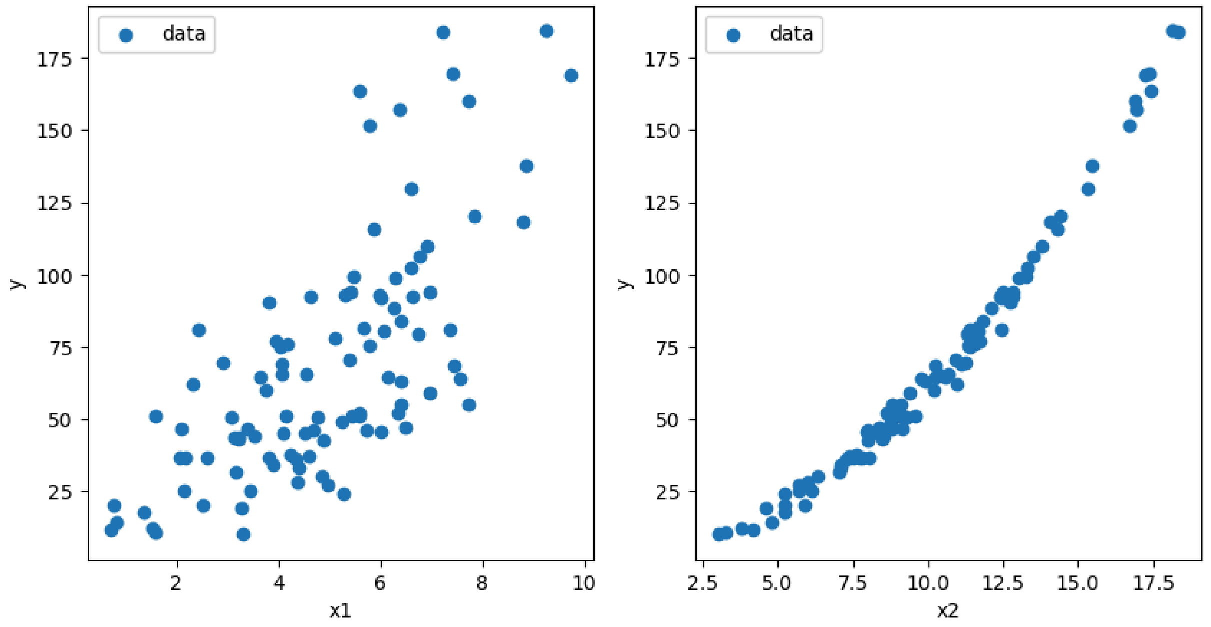
```
    Xdf = pd.DataFrame()
    Xdf['Constant'] = np.ones(N)
    Xdf['x1'] = X1
    Xdf['x2'] = X2

    # Create df and add y
    df = Xdf.copy()
    y = alpha + beta*X1 + gamma*X2**2 + u
    df['y'] = y

    # Add Z to df
    z_cols = []
    for i in range(len(pi)):
        z_col = 'z'+str(i+1)
        z_cols.append(z_col)
        df[z_col] = Z[:, i]

    return df['y'], Xdf, df[['Constant']+z_cols] # y as series
```

In [ ]:

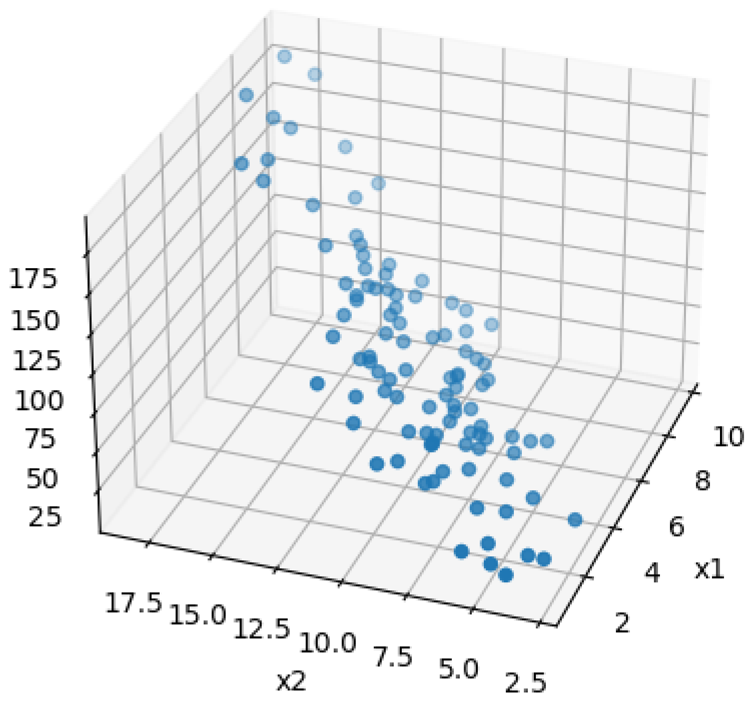
```
np.random.seed(1234)

alpha = 1
beta = 2
gamma = 0.5
N = 100

y, X, Z = dgp_g(N, [alpha, beta, gamma], pi=[1,2,3,4])
```

In [ ]:

```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))

ax1.scatter(X['x1'], y, label='data')
ax1.set_xlabel('x1')
ax1.set_ylabel('y')
ax1.legend()

ax2.scatter(X['x2'], y, label='data')
ax2.set_xlabel('x2')
ax2.set_ylabel('y')
ax2.legend()

plt.show()
```



In [ ]:
```python
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X['x1'], X['x2'], y)

ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')

# ax.view_init(10, 250)
ax.view_init(30, 200)

plt.show()
```



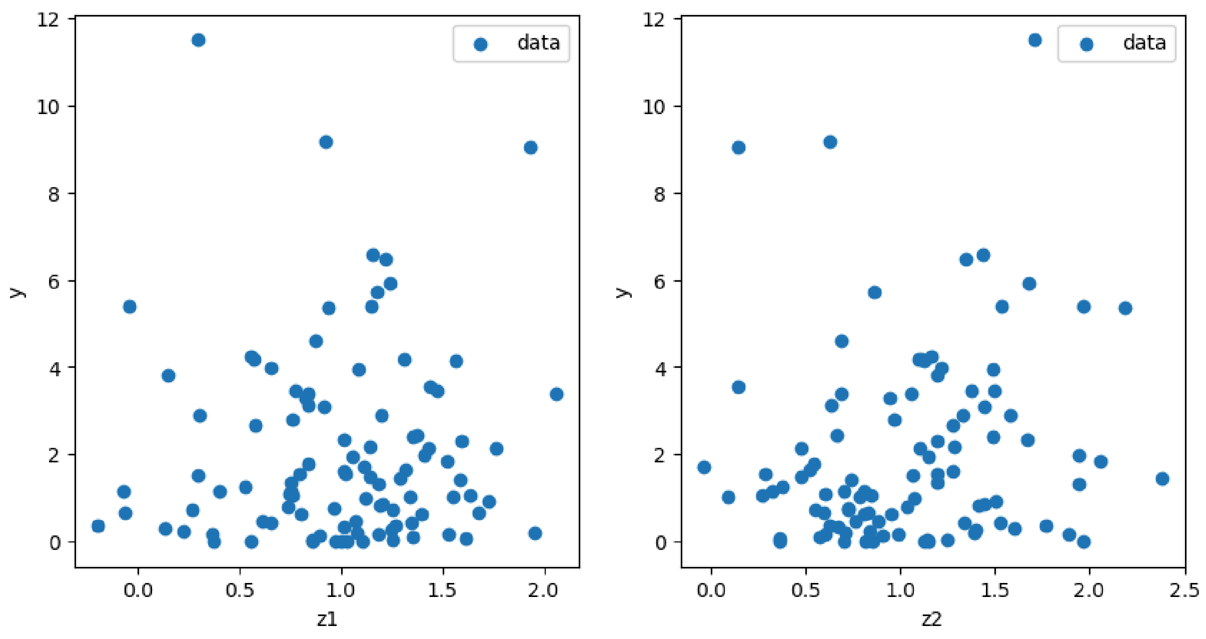
(h) $y^\gamma = \alpha + u$, with $y > 0$ and γ a scalar, and $\mathbb{E}(Z^\top u) = \mathbb{E}u = 0$ and $\mathbb{E}Z^\top \begin{bmatrix} \gamma y^{\gamma-1} \\ -1 \end{bmatrix} = Q(\gamma)$. *Note: y, Z, and u are random variables.*

```
In [ ]: def dgp_h(N, b0, pi=[3,4], sigma_u=1):
            '''
            Generate dataset for GMM model G

            Takes as inputs sample size N and vector of true parameters (alpha and g

            Returns a tuple with numpy arrays y and Z, all of length N
            '''
            # Retrieve true parameters
            alpha = b0[0]
            gamma = b0[1]

            # Construct random variables
            u = iid.norm(scale=sigma_u).rvs(N)
            Z = np.random.normal(size=(N,len(pi)), loc=1, scale=0.5)

            # Create df and add y
            df = pd.DataFrame()
            y = (alpha + u)**(1/gamma)
            df['y'] = y
            df['Constant'] = 1

            # Add Z to df
            z_cols = []
            for i in range(len(pi)):
```

```
        z_col = 'z'+str(i+1)
        z_cols.append(z_col)
        df[z_col] = Z[:, i]

    return df['y'], df[['Constant']+z_cols] # y as series
```

In [ ]:
```
np.random.seed(1234)

alpha = 1
gamma = 0.5
N = 100

y, Z = dgp_h(N, [alpha, gamma], pi=[3,4])
```

In [ ]:
```
import matplotlib.pyplot as plt

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10,5))

ax1.scatter(Z['z1'], y, label='data')
ax1.set_xlabel('z1')
ax1.set_ylabel('y')
ax1.legend()

ax2.scatter(Z['z2'], y, label='data')
ax2.set_xlabel('z2')
ax2.set_ylabel('y')
ax2.legend()

plt.show()
```



(4) Select the most interesting of the data generating processes you developed, and using the code in `gmm.py` or `GMM_class.py` (see https://github.com/ligonteaching/ARE212_Materials/) use data

from your `dgp` to analyze the finite sample performance of the corresponding GMM estimator you've constructed. Of particular interest is the distribution of your estimator using a sample size N and how this distribution compares with the limiting distribution as $N \to \infty$.

In [ ]:
```
import numpy as np
from numpy.linalg import inv
from scipy.optimize import minimize_scalar

def gj(b,y,X,Z):
    """Observations of g_j(b).

    This defines the deviations from the predictions of our model; i.e.,
    e_j = Z_ju_j, where EZ_ju_j=0.

    Can replace this function to testimate a different model.
    """
    return Z*(y - X*b)

def gN(b,data):
    """Averages of g_j(b).

    This is generic for data, to be passed to gj.
    """
    e = gj(b,*data) # is this actually a residual? Why do we call it e inste

    # Check to see more obs. than moments.
    assert e.shape[0] > e.shape[1], "Need more observations than moments!"

    return e.mean(axis=0)

def Omegahat(b,data):
    """Calculate estimate of Omega, inverse of weighting matrix"""

    e = gj(b,*data)

    # IMPORTANT: Recenter! We have Eu=0 under null.
    # Important to use this information.
    e = e - e.mean(axis=0)

    # TODO: Where does 1/N come from?
    return e.T@e/e.shape[0] # Omega = (u u^T)/N??

def J(b,W,data):
    """Define GMM criterion function"""

    m = gN(b,data) # Sample moments @ b
    N = data[0].shape[0]

    return (N*m.T@W@m) # Scale by sample size

def two_step_gmm(data):
    # First step uses identity weighting matrix
    # data is a tuple
    W1 = np.eye(gj(1,*data).shape[1]) # *tuple means treat elements of this

    # b1 is value of b that minimizes J given initial estimate of weighting
    b1 = minimize_scalar(lambda b: J(b,W1,data)).x # minimize_scalar().x ret
    # remember that b1 is consistent

    # Construct 2nd step weighting matrix using
    # first step estimate of beta
    W2 = inv(Omegahat(b1,data))

    return minimize_scalar(lambda b: J(b,W2,data))
```

In [ ]:

```
# using pre-built methods
def linear_iv(b,data):
    """
    Return matrix Z.T*e (ell x N)
    """
    y,X,Z=data
    e = y - X@b

    return (Z.T*e).T

# define true parameters
alpha = 1
beta = 2

# generate data
y,X,Z = dgp_e(N, [alpha, beta], pi=[3,4])

# create GMM estimator
est = GMM(linear_iv, (y,X,Z), 2)

# run GMM estimator
b, fun = est.two_step_gmm()

print(b)

print(fun)
```

```
[0.20951407 2.06501508]
0.0598679685048684
```

In [ ]:

```
est.Omegahat(b)
```

Out[ ]:

	Constant	z1	z2
Constant	3.537147	0.182723	0.836641
z1	0.182723	3.385555	-0.663815
z2	0.836641	-0.663815	3.255536

In [ ]:

```
# using pre-built methods
```

```
N = 1000 # Sample size

D = 1000 # Monte Carlo draws

b_draws_1000 = pd.DataFrame(columns=['b0', 'b1'])
J_draws_1000 = []

# true parameters
alpha = 1
beta = 2

for d in range(D):
    est = GMM(linear_iv, dgp_e(N, [alpha, beta], pi=[3,4]), 2)
    b, fun = est.two_step_gmm()
    b_df = pd.DataFrame(b, index=b_draws_1000.columns).T
    b_draws_1000 = pd.concat([b_draws_1000, b_df], ignore_index=True)
    J_draws_1000.append(fun)
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/1182248769.
py:18: FutureWarning: The behavior of DataFrame concatenation with empty or
all-NA entries is deprecated. In a future version, this will no longer exclu
de empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  b_draws_1000 = pd.concat([b_draws_1000, b_df], ignore_index=True)
```

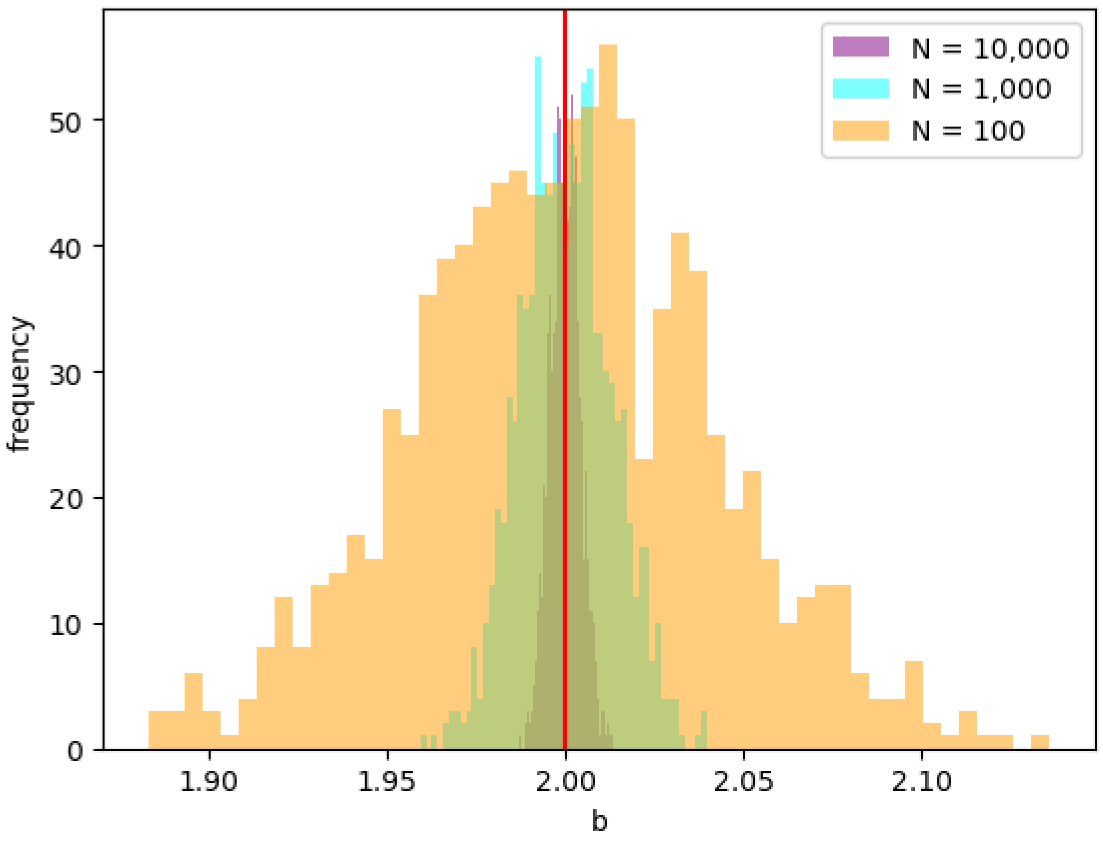
In [ ]:
```
# What if reduce sample size?
N = 100

b_draws_100 = []
J_draws_100 = []

b_draws_100 = pd.DataFrame(columns=['b0', 'b1'])
J_draws_100 = []
for d in range(D):
    est = GMM(linear_iv, dgp_e(N, [alpha, beta], pi=[3,4]), 2)
    b, fun = est.two_step_gmm()
    b_df = pd.DataFrame(b, index=b_draws_100.columns).T
    b_draws_100 = pd.concat([b_draws_100, b_df], ignore_index=True)
    J_draws_100.append(fun)
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/3091103090.
py:13: FutureWarning: The behavior of DataFrame concatenation with empty or
all-NA entries is deprecated. In a future version, this will no longer exclu
de empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  b_draws_100 = pd.concat([b_draws_100, b_df], ignore_index=True)
```

In [ ]:
```
# What if increase sample size?
N = 10000

b_draws_10000 = pd.DataFrame(columns=['b0', 'b1'])
J_draws_10000 = []
for d in range(D):
    est = GMM(linear_iv, dgp_e(N, [alpha, beta], pi=[3,4]), 2)
    b, fun = est.two_step_gmm()
    b_df = pd.DataFrame(b, index=b_draws_10000.columns).T
    b_draws_10000 = pd.concat([b_draws_10000, b_df], ignore_index=True)
    J_draws_10000.append(fun)
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/581988622.p
y:10: FutureWarning: The behavior of DataFrame concatenation with empty or a
ll-NA entries is deprecated. In a future version, this will no longer exclud
e empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  b_draws_10000 = pd.concat([b_draws_10000, b_df], ignore_index=True)
```

In [ ]:
```
import matplotlib.pyplot as plt

_ = plt.hist(b_draws_10000['b1'], bins=50, color='purple', alpha=0.5, label=
_ = plt.hist(b_draws_1000['b1'], bins=50, color='cyan', alpha=0.5, label='N 
_ = plt.hist(b_draws_100['b1'], bins=50, color='orange', alpha=0.5, label='N
_ = plt.xlabel('b')
_ = plt.ylabel('frequency')
_ = plt.legend()
_ = plt.axvline(beta, color='r')
```



In [ ]:
```
Q = np.array(Z.T@X)
Winv = est.Omegahat(b)
```

In [ ]:
```
# Limiting distribution of criterion (under null)
limiting_J = iid.chi2(2-1)

# Limiting SE of b
# note, avar is embedded: avar(b) = inv(Q.T@inv(Winv)@Q)
sigma_0 = lambda N: np.sqrt(inv(Q.T@inv(Winv)@Q)/N)[0][0]
```

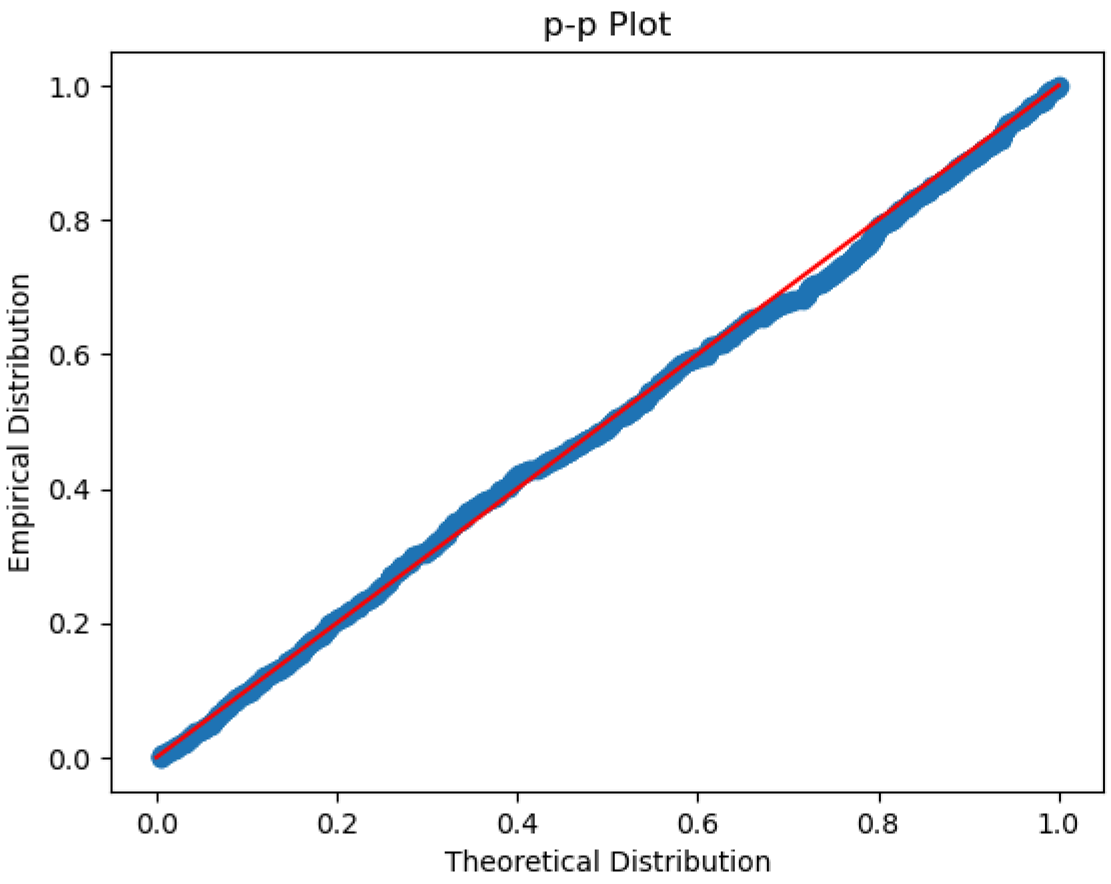
```
# Limiting distribution of estimator
limiting_b = iid.norm(loc=beta,scale=sigma_0(N)) # from central limit theore

print("Monte Carlo standard errors (N = 10000): %g" % np.std(b_draws_10000['
print("Asymptotic approximation: %g" % sigma_0(N)) # asymptotic distributior
print("Critical value for J statistic: %g (5%%)" % limiting_J.isf(.05))
```

```
Monte Carlo standard errors (N = 10000): 0.00414828
Asymptotic approximation: 0.000197577
Critical value for J statistic: 3.84146 (5%)
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/378196487.p
y:6: RuntimeWarning: invalid value encountered in sqrt
  sigma_0 = lambda N: np.sqrt(inv(Q.T@inv(Winv)@Q)/N)[0][0]
```

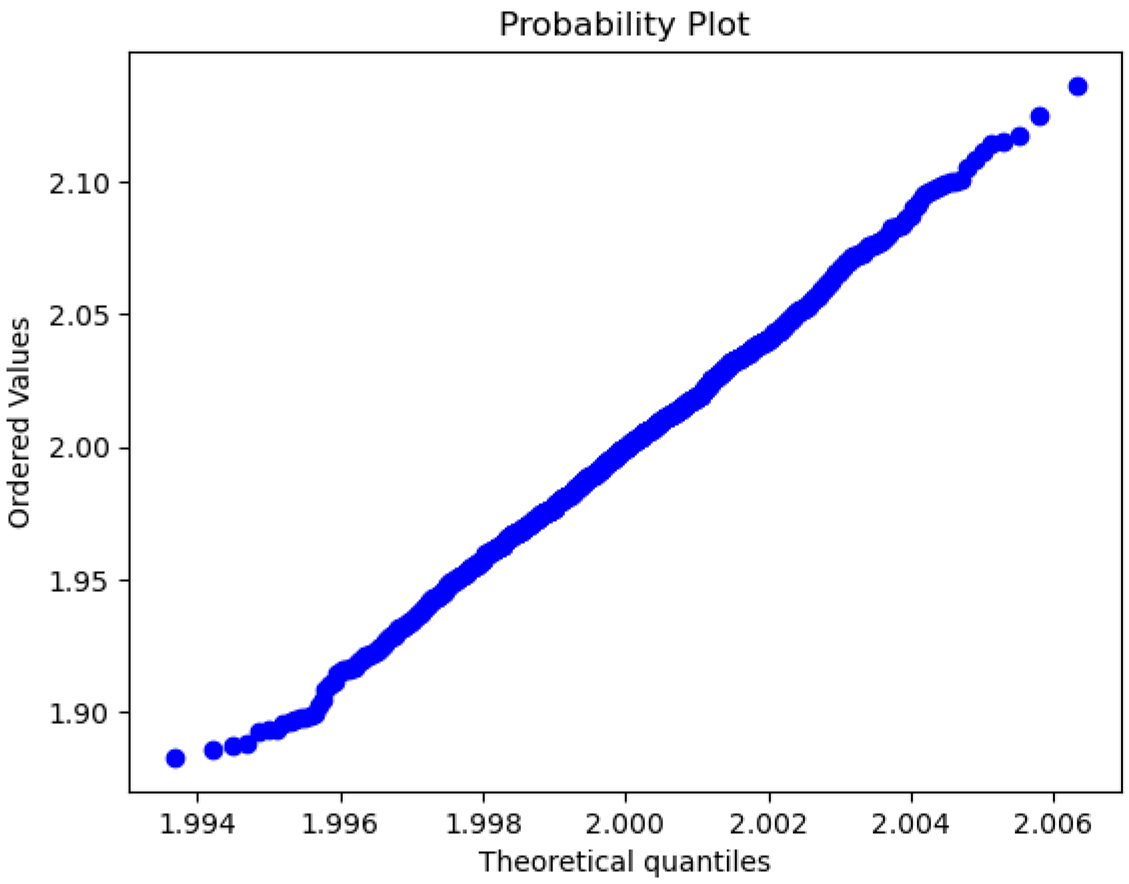
In [ ]:
```
import matplotlib.pyplot as plt

def ppplot(data,dist):
    data = np.array(data)

    # Theoretical CDF, evaluated at points of data
    P = [dist.cdf(x) for x in data.tolist()]

    # Empirical CDF, evaluated at points of data
    Phat = [(data<x).mean() for x in data.tolist()]

    fig, ax = plt.subplots()

    ax.scatter(P,Phat)
    ax.plot([0,1],[0,1],color='r') # Plot 45
    ax.set_xlabel('Theoretical Distribution')
    ax.set_ylabel('Empirical Distribution')
    ax.set_title('p-p Plot')

    return ax

_ = ppplot(J_draws_100, limiting_J)
print(np.mean(np.array(J_draws_100) < limiting_J.isf(.05)))
```

```
0.948
```



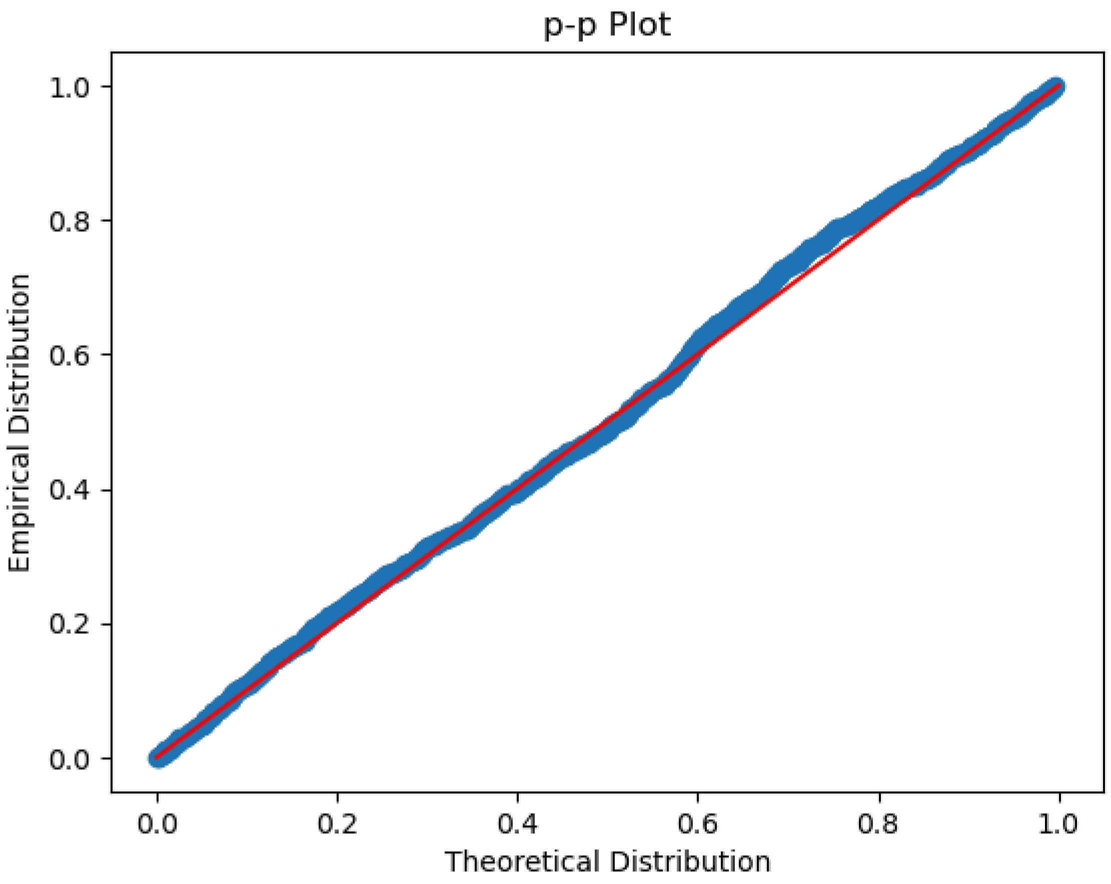
In [ ]:
```
from scipy.stats import probplot

limiting_b = iid.norm(loc=beta, scale=sigma_0(100)) # from central limit the
_ = probplot(b_draws_100['b1'], dist=limiting_b, fit=False, plot=plt) # look
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/378196487.p
y:6: RuntimeWarning: invalid value encountered in sqrt
  sigma_0 = lambda N: np.sqrt(inv(Q.T@inv(Winv)@Q)/N)[0][0]
```



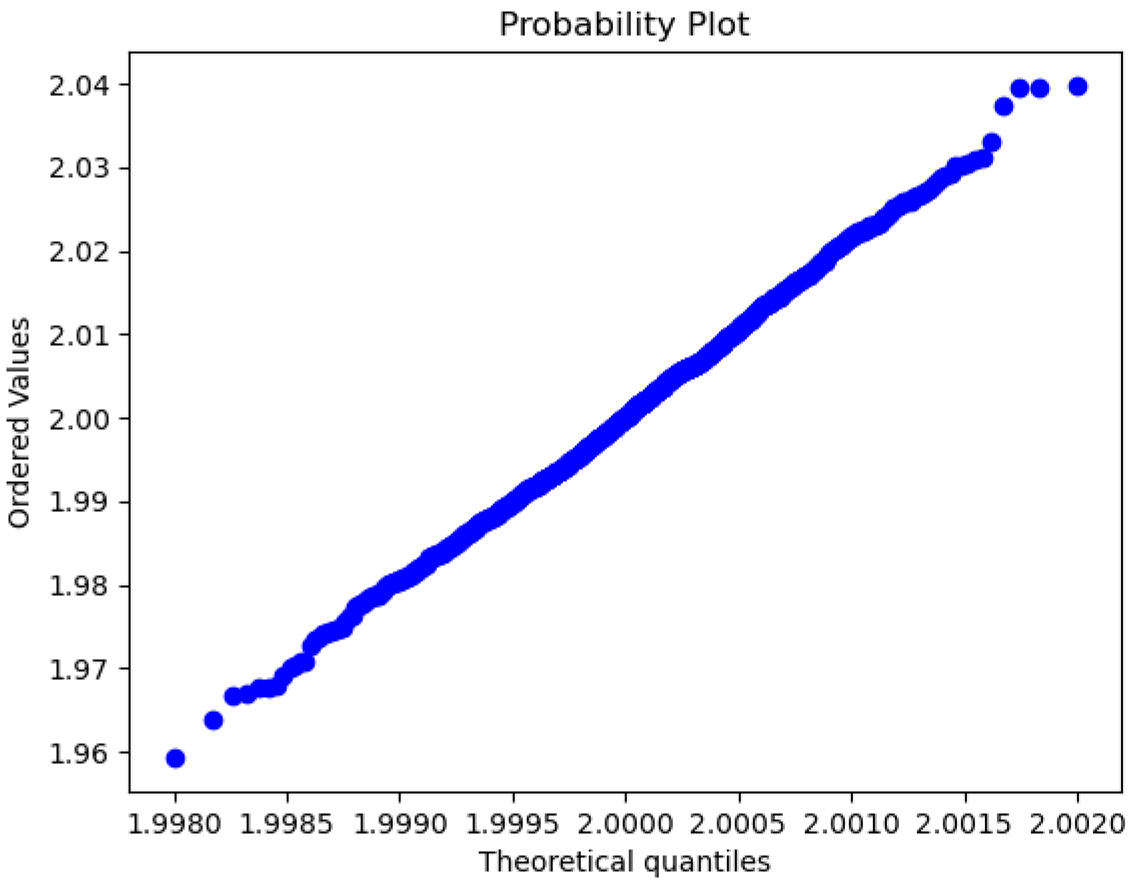
```
In [ ]: _ = ppplot(J_draws_1000, limiting_J)
        print(np.mean(np.array(J_draws_1000) < limiting_J.isf(.05)))
```

```
0.953
```



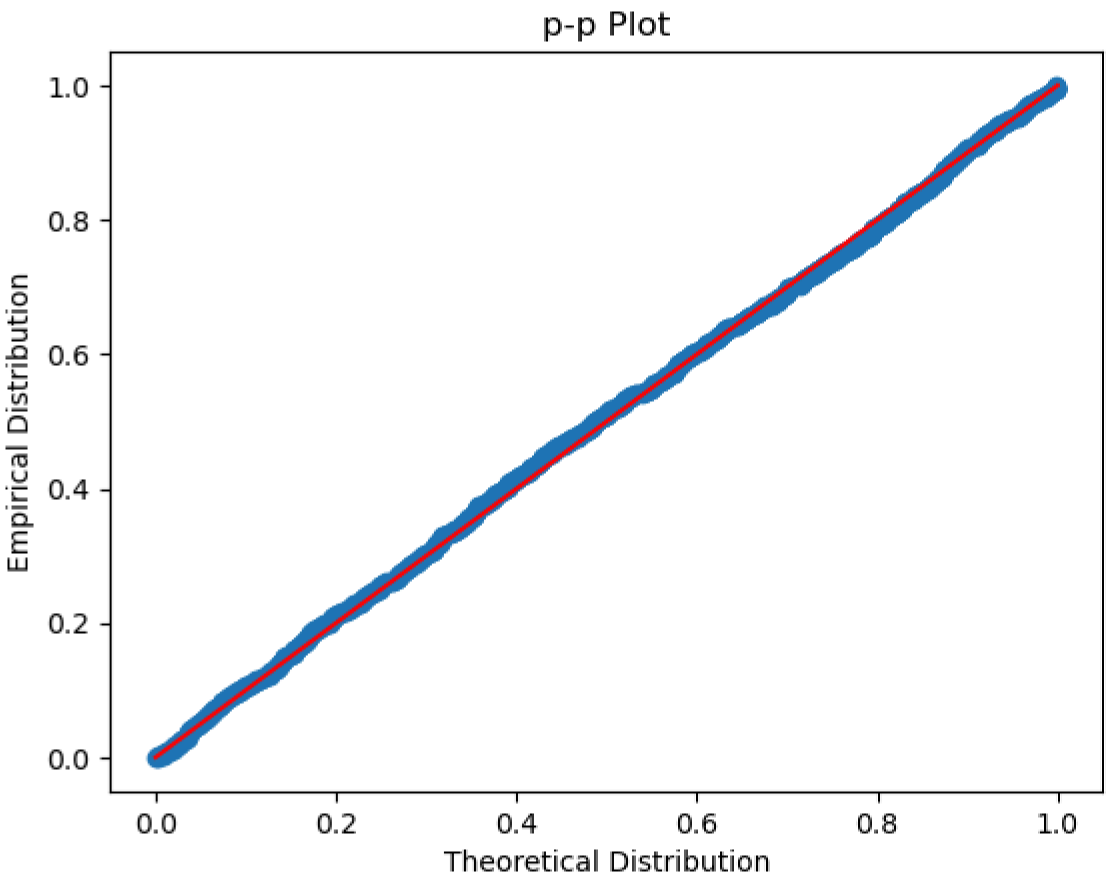
In [ ]:
```
from scipy.stats import probplot

limiting_b = iid.norm(loc=beta, scale=sigma_0(1000)) # from central limit th
_ = probplot(b_draws_1000['b1'], dist=limiting_b, fit=False, plot=plt) # loc
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/378196487.p
y:6: RuntimeWarning: invalid value encountered in sqrt
  sigma_0 = lambda N: np.sqrt(inv(Q.T@inv(Winv)@Q)/N)[0][0]
```



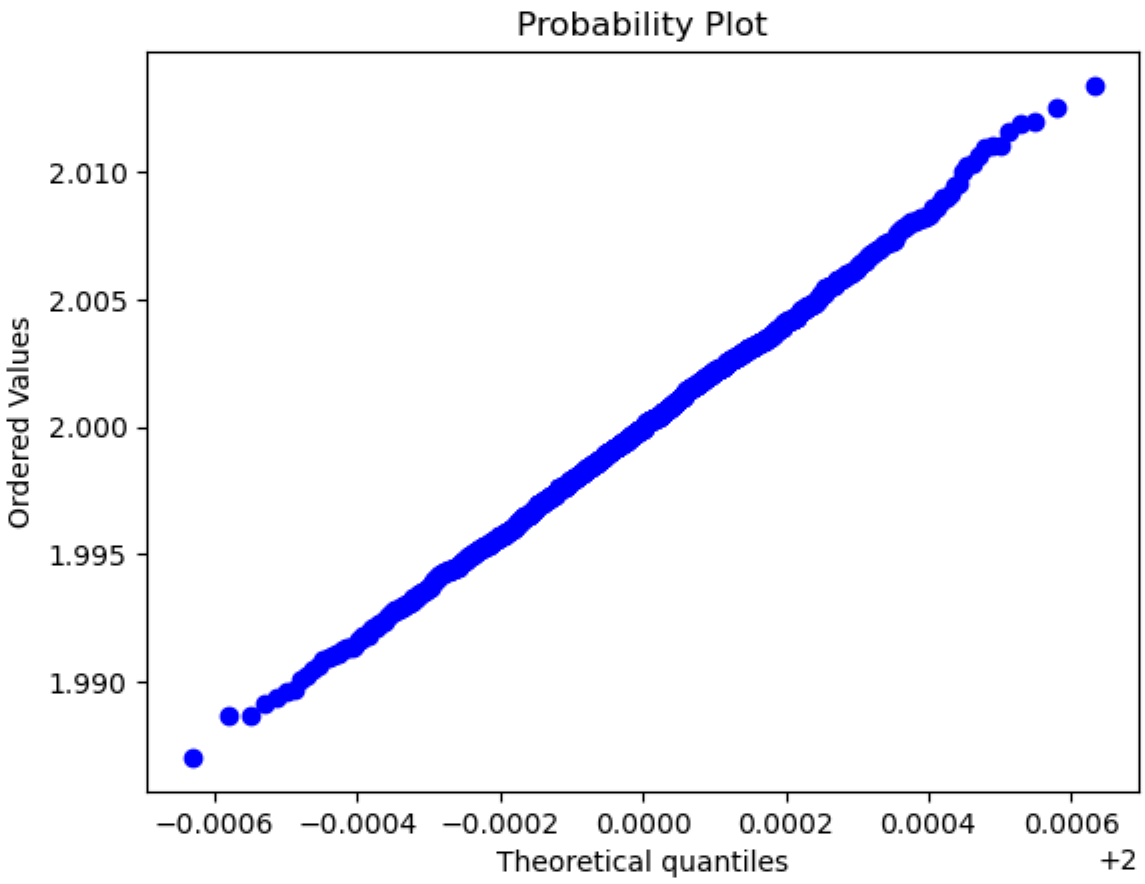
```
In [ ]: _ = ppplot(J_draws_10000, limiting_J)
        print(np.mean(np.array(J_draws_10000) < limiting_J.isf(.05)))
```

```
0.951
```



In [ ]:
```
from scipy.stats import probplot

limiting_b = iid.norm(loc=beta, scale=sigma_0(10000)) # from central limit t
_ = probplot(b_draws_10000['b1'], dist=limiting_b, fit=False, plot=plt) # lo
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/378196487.p
y:6: RuntimeWarning: invalid value encountered in sqrt
  sigma_0 = lambda N: np.sqrt(inv(Q.T@inv(Winv)@Q)/N)[0][0]
```



3. Breusch-Pagan Extended

Consider a linear regression of the form

$y = \alpha + \beta x + u$,

with (y, x) both scalar random variables, where it is assumed that (a.i) $\mathbb{E}(u \cdot x) = \mathbb{E}u = 0$ and (a.ii) $\mathbb{E}(u^2|x) = \sigma^2$.

(1) The condition a.i is essentially untestable; explain why.

Condition a.i makes the assumption that X is exogenous and has no correlation with the error u. This is essentially untestable because u is unobserved and can only be estimated. An assumption of exogeneity can only be justified on theoretic grounds, not observed data.

(2) Breusch and Pagan (1979) argue that one can test a.ii via an auxiliary regression $\hat{u}^2 = c + dx + e$, where the $\hat{u}$ are the residuals from the first regression, and the test of a.ii then becomes a test of $H_0 : d = 0$. Describe the logic of the test of a.ii.

If there is heteroskedasticity, then this means that there is some relationship between the variance u^2 and x, e.g., u^2 might increase or decrease as x increases. On the other hand, if the model is homoskedastic, then the coefficients of this auxiliary regression should be 0 because there is no relationship between u^2 and x.

(3) Use the two conditions a.i and a.ii to construct a GMM version of the Breusch-Pagan test.

NOTE: Redefining model to be $\hat{u}^2 = c + dx + w$ to avoid confusion between "e" and estimated error

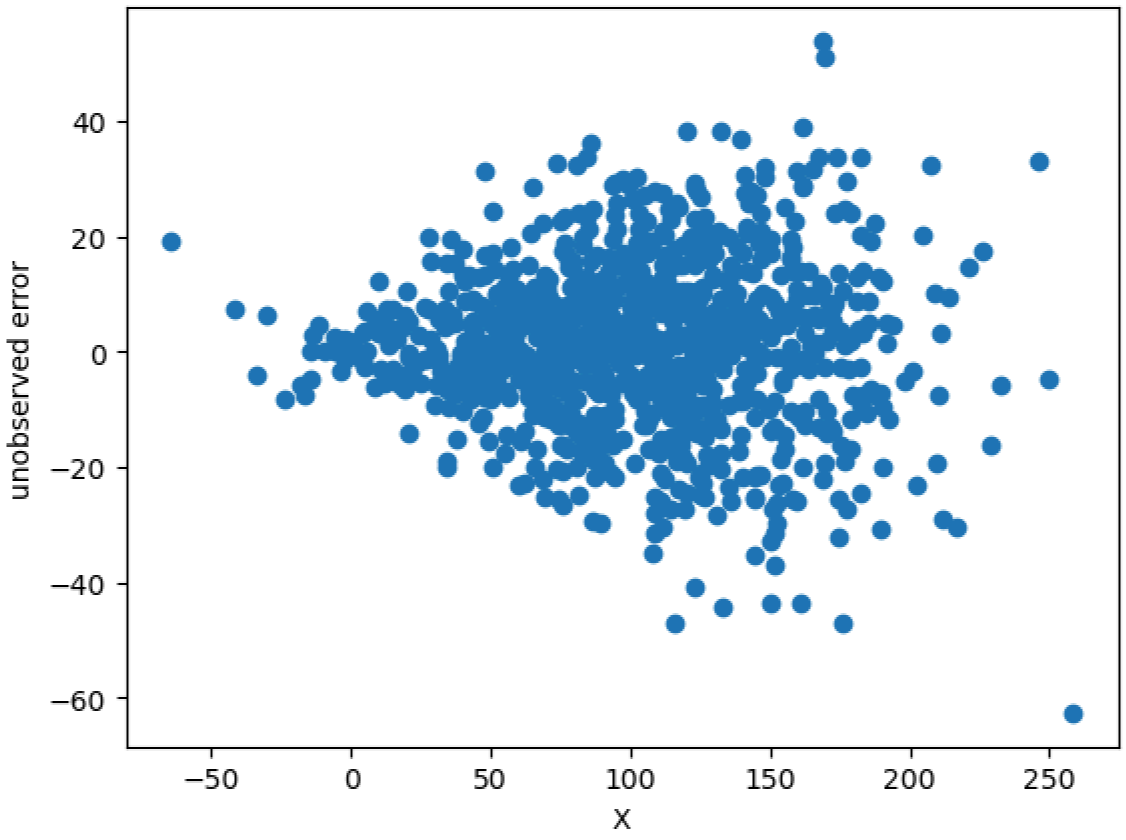
$y = \alpha + \beta x + u$

Step 1: Estimate α and β, calculate residuals $\hat{u}$

Step 2: Test $H_0 : d = 0$ for $\hat{u}^2 = c + dx + w$ assuming (a.i) $\mathbb{E}(u \cdot x) = \mathbb{E}u = 0$

$$g_j(d) = x(\hat{u}^2 - \overline{\hat{u}^2} - dx)$$

```
In [ ]: # Illustrate heteroskedasticity
        import matplotlib.pyplot as plt

        N = 1000
        true_c = 1
        true_d = 2
        X = np.random.normal(size=N, loc=100, scale=50)
        u = np.random.normal(size=N, loc=true_c, scale=np.sqrt(true_d*np.abs(X)))

        _ = plt.scatter(X,u)
        _ = plt.xlabel('X')
        _ = plt.ylabel('unobserved error')
```



In [ ]:
```
import numpy as np
import pandas as pd
from scipy.stats import distributions as iid

def bp_gdp(N, b0, true_c=0, true_d=0):
    '''
    Generate dataset for model with linear heteroskedasticity

    Takes as inputs sample size N and vector of true parameters (alpha and b

    Returns a tuple with numpy arrays y, X and Z, all of length N
    '''
    # Retrieve true parameters
    alpha = b0[0]
    beta = b0[1]

    # Construct X
    X = np.random.normal(size=N, loc=100, scale=50)

    # Construct u
    if true_d == 0:
        u = np.random.normal(size=N, loc=true_c, scale=1)
    else:
        u = np.random.normal(size=N, loc=true_c, scale=np.sqrt(true_d*np.abs

    # Construct y
    y = alpha + X*beta + u

    # Construct df
    combined_dict = {'y':y, 'x':X, 'Constant':1}
    df = pd.DataFrame(combined_dict)

    return df['y'], df[['Constant','x']]
```

In [ ]:
```
def linear_regression(b,data):
    """
    Return matrix X.T*e (ell x N)
    """
    y,X=data
    e = y - X@b

    return (X.T*e).T
```

In [ ]:
```
# generate data
N = 1000
alpha = 3
beta = 4
true_c = 1
true_d = 2
y,X = bp_gdp(N, [alpha, beta], true_c=true_c, true_d=true_d)

# create GMM estimator
est = GMM(linear_regression, (y,X), 2)

# run GMM estimator
b, fun = est.two_step_gmm()
print(b)

# get residuals
u_hat = y - X@b
u_hat2 = u_hat**2

# regress residuals
bp_est = GMM(linear_regression, (u_hat2,X), 2)
d, bp_fun = bp_est.two_step_gmm()
d
```

```
[3.2961958 4.0085438]
```

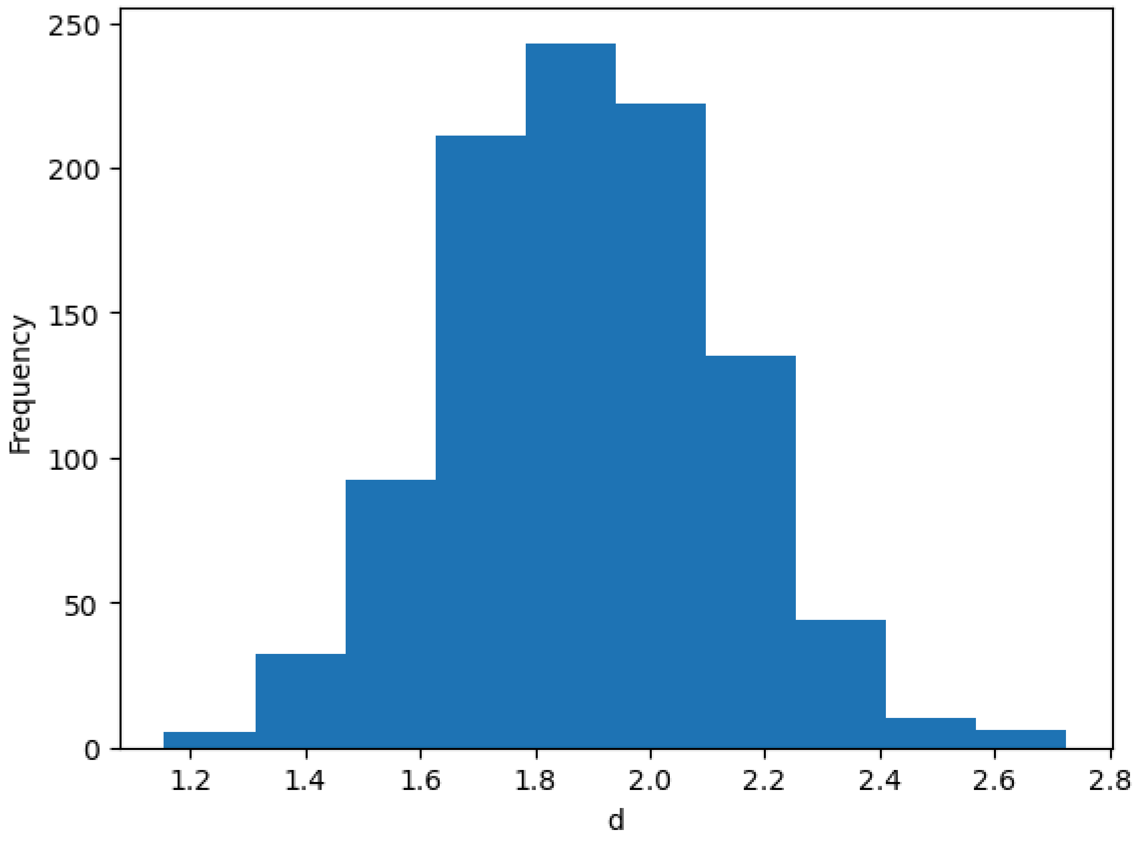
Out[ ]:
```
array([5.72519027, 1.99833698])
```

In [ ]:
```
# Monte Carlo simulation

N = 1000 # Sample size
D = 1000 # Monte Carlo draws

# Structural parameters
alpha = 3
beta = 4
true_c = 1
true_d = 2

d_draws = pd.DataFrame(columns=['c', 'd'])
for i in range(D):
    y,X = bp_gdp(N, [alpha, beta], true_c=true_c, true_d=true_d)
    est = GMM(linear_regression, (y,X), 2)
    b, fun = est.two_step_gmm()
    u_hat2 = (y - X@b)**2
    bp_est = GMM(linear_regression, (u_hat2,X), 2)
    d, bp_fun = bp_est.two_step_gmm()
    d_df = pd.DataFrame(d, index=d_draws.columns).T
    d_draws = pd.concat([d_draws, d_df], ignore_index=True)
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/3295730941.
py:21: FutureWarning: The behavior of DataFrame concatenation with empty or
all-NA entries is deprecated. In a future version, this will no longer exclu
de empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  d_draws = pd.concat([d_draws, d_df], ignore_index=True)
```

In [ ]:
```
import matplotlib.pyplot as plt

plt.hist(d_draws['d'])
plt.xlabel('d')
plt.ylabel('Frequency')
```

Out[ ]:
```
Text(0, 0.5, 'Frequency')
```



In [ ]:
```
from scipy.stats import t

d_avg = d_draws['d'].mean()
bias = d_avg - true_d
var = np.sum(np.square(d_draws['d'] - d_avg))
sd = np.sqrt(var)
se = sd/np.sqrt(len(d_draws))
```

```
print('Average estimate: ',d_avg)
print('Bias: ', bias)
print('Standard error (precision): ', se)

# Size of two-tailed t-test for d = 0
t_score = (d_avg - 0)/se
n = len(d_draws)
df = n - 1
pval = t.sf(np.abs(t_score), df) * 2
print('p-value: ', pval)
```

```
Average estimate:  1.8912832820729422
Bias:  -0.10871671792705784
Standard error (precision):  0.24109346100320092
p-value:  1.1119883494574107e-14
```

In [ ]:

```
# Structural parameters;
true_c = 0.2
true_d = 0.5

N = 1000 # Sample size
D = 1000 # Monte Carlo draws

d_draws = pd.DataFrame(columns=['c', 'd'])
for i in range(D):
    est = GMM(linear_regression, bp_gdp(N, [alpha, beta], true_c=true_c, tru
    b, fun = est.two_step_gmm()
    u_hat2 = (y - X@b)**2
    bp_est = GMM(linear_regression, (u_hat2,X), 2)
    d, bp_fun = bp_est.two_step_gmm()
    d_df = pd.DataFrame(d, index=d_draws.columns).T
    d_draws = pd.concat([d_draws, d_df], ignore_index=True)
```

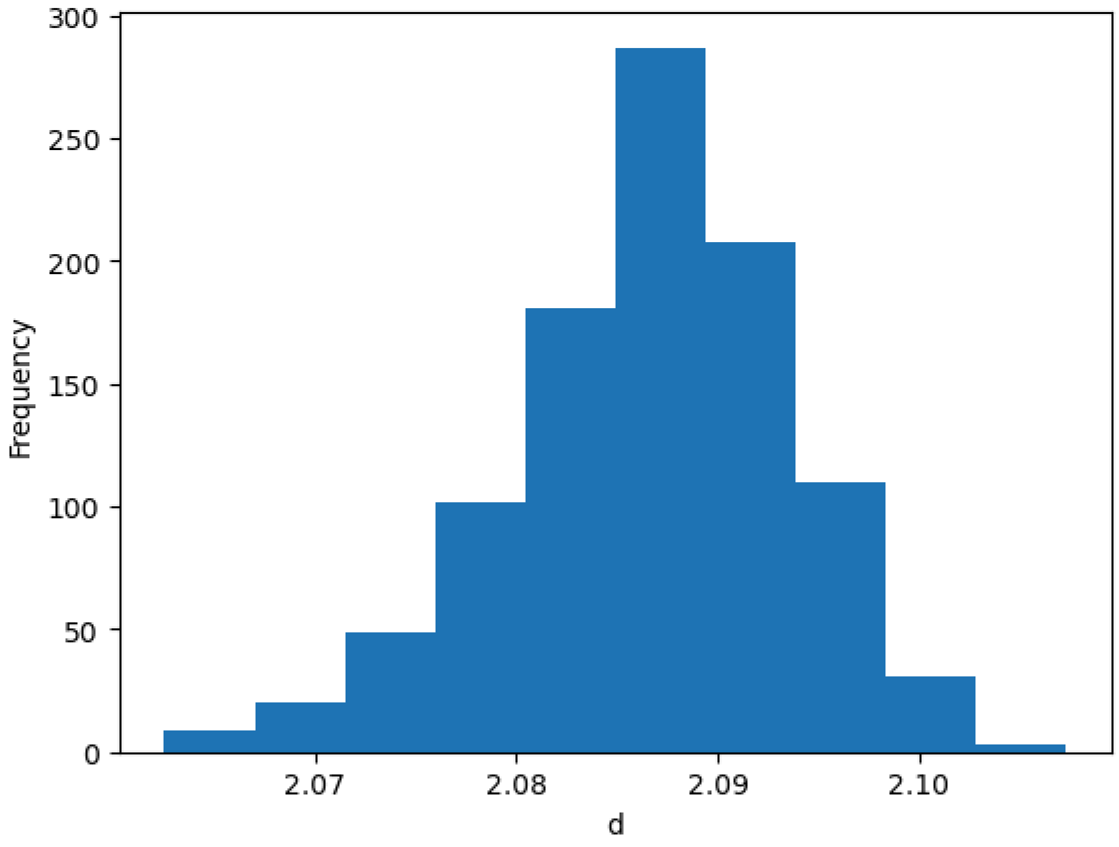
```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/4027050908.
py:16: FutureWarning: The behavior of DataFrame concatenation with empty or
all-NA entries is deprecated. In a future version, this will no longer exclu
de empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  d_draws = pd.concat([d_draws, d_df], ignore_index=True)
```

In [ ]:

```
plt.hist(d_draws['d'])
plt.xlabel('d')
plt.ylabel('Frequency')
```

Out[ ]:

```
Text(0, 0.5, 'Frequency')
```



In [ ]:
```
from scipy.stats import t

d_avg = d_draws['d'].mean()
bias = d_avg - true_d
var = np.sum(np.square(d_draws['d'] - d_avg))
sd = np.sqrt(var)
se = sd/np.sqrt(len(d_draws))

print('Average estimate: ',d_avg)
print('Bias: ', bias)
print('Standard error (precision): ', se)

# Size of two-tailed t-test for d = 0
t_score = (d_avg - 0)/se
n = len(d_draws)
df = n - 1
pval = t.sf(np.abs(t_score), df) * 2
print('p-value: ', pval)
```

```
Average estimate:  2.086593764793533
Bias:  1.586593764793533
Standard error (precision):  0.007054714944087051
p-value:  0.0
```

(4) What can you say about the performance or relative merits of the Breusch-Pagan test versus your GMM alternative?

In this case, the Breusch-Pagan test is not able to detect heteroskedasticity, while the GMM alternative is. In addition, for larger values of d, the GMM alternative yields a reasonable estimate of d. The B-P test does not tell us anything about which variable is correlated with u, only that there is heteroskedasticity present.

However, the drawback of this GMM estimator is that we have assumed the form that heteroskedasticity takes, i.e., $\hat{u}^2 = c + dx + w$. If there was heteroskedasticity of a different form, then the GMM estimator's performance would probably be poor. The B-P test is more flexible because it can detect heteroskedasticity regardless of its form.

In [ ]:
```
# c, d = 1, 2

import statsmodels.formula.api as smf
from statsmodels.compat import lzip
import statsmodels.stats.api as sms

N = 1000
alpha = 3
beta = 4
true_c = 1
true_d = 2
y,X = bp_gdp(N, [alpha, beta], true_c=true_c, true_d=true_d)

df = X.copy()
df['y'] = y

# canned OLS regression
fit = smf.ols('y ~ x', data=df).fit()
print(fit.summary())

# canned Breusch-Pagan test
names = ['Lagrange multiplier statistic', 'p-value',
        'f-value', 'f p-value']
test = sms.het_breuschpagan(fit.resid, fit.model.exog)

lzip(names, test)
```

```
                            OLS Regression Results                            
==============================================================================
==
Dep. Variable:                      y   R-squared:                       0.9
95
Model:                            OLS   Adj. R-squared:                  0.9
95
Method:                 Least Squares   F-statistic:                 1.903e+
05
Date:                Wed, 01 May 2024   Prob (F-statistic):               0.
00
Time:                        17:50:38   Log-Likelihood:                -410
2.7
No. Observations:                1000   AIC:                             820
9.
Df Residuals:                     998   BIC:                             821
9.
Df Model:                           1                                         
Covariance Type:            nonrobust                                         
==============================================================================
==
                 coef    std err          t      P>|t|      [0.025      0.97
5]
------------------------------------------------------------------------------
--
Intercept      3.3355      1.015      3.286      0.001       1.343       5.3
28
x              4.0122      0.009    436.219      0.000       3.994       4.0
30
==============================================================================
==
Omnibus:                       11.091   Durbin-Watson:                   1.9
16
Prob(Omnibus):                  0.004   Jarque-Bera (JB):               16.4
01
Skew:                           0.064   Prob(JB):                     0.0002
74
Kurtosis:                       3.614   Cond. No.                         24
2.
==============================================================================
==

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is corre
ctly specified.
```

Out[ ]:
```
[('Lagrange multiplier statistic', 87.3516483439737),
 ('p-value', 9.0844394903266e-21),
 ('f-value', 95.52084862598025),
 ('f p-value', 1.3141318131828075e-21)]
```

In [ ]:
```python
# c, d = 0.2, 0.5

true_c = 0.2
true_d = 0.5
y,X = bp_gdp(N, [alpha, beta], true_c=true_c, true_d=true_d)
```

```
df = X.copy()
df['y'] = y

# canned OLS regression
fit = smf.ols('y ~ x', data=df).fit()
print(fit.summary())

# canned Breusch-Pagan test
names = ['Lagrange multiplier statistic', 'p-value',
        'f-value', 'f p-value']
test = sms.het_breuschpagan(fit.resid, fit.model.exog)

lzip(names, test)
```

```
                            OLS Regression Results
==============================================================================
==
Dep. Variable:                      y   R-squared:                       0.9
99
Model:                            OLS   Adj. R-squared:                  0.9
99
Method:                 Least Squares   F-statistic:                 8.157e+
05
Date:                Wed, 01 May 2024   Prob (F-statistic):               0.
00
Time:                        17:50:38   Log-Likelihood:                -335
3.3
No. Observations:                1000   AIC:                             671
1.
Df Residuals:                     998   BIC:                             672
1.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
==
                 coef    std err          t      P>|t|      [0.025      0.97
5]
------------------------------------------------------------------------------
--
Intercept      2.9018      0.491      5.904      0.000       1.937       3.8
66
x              4.0052      0.004    903.134      0.000       3.997       4.0
14
==============================================================================
==
Omnibus:                        9.675   Durbin-Watson:                   2.0
22
Prob(Omnibus):                  0.008   Jarque-Bera (JB):               13.6
03
Skew:                           0.067   Prob(JB):                      0.001
11
Kurtosis:                       3.556   Cond. No.                         24
9.
==============================================================================
==

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is corre
ctly specified.
```

```
Out[ ]:  [('Lagrange multiplier statistic', 111.61681703198134),
          ('p-value', 4.3349487464815334e-26),
          ('f-value', 125.3891175942347),
          ('f p-value', 1.6849021218680938e-27)]
```

(5) Suppose that in fact that x is distributed uniformly over the interval $[0, 2\pi]$, and $\mathbb{E}(u^2|x) = \sigma^2(x) = \sigma^2 sin(2x)$, thus violating a.ii. What can you say about the performance of the

Breusch-Pagan test in this circumstance? Can you modify your GMM test to provide a superior alternative?

NOTE: $sin(2x)$ ***has negative values in the interval*** $[0, 2\pi]$. ***Assume that*** $\mathbb{E}(u^2|x) = \sigma^2(x) = \sigma^2|sin(2x)|$

Our original GMM estimator is no longer able to detect heteroskedasticity when it takes a sinusoidal form. If we use a 5% significance level, the *p*-value is greater than 0.05, so we are unable to reject the null hypothesis $H_0 : d = 0$.

However, we can modify the GMM estimator to test for error that varies sinusoidally with X. After making this modification, the estimator is able to detect heteroskedasticity. In practice, though, this type of estimator is most useful when we have some theoretical basis for expecting heteroskedasticity of a particular form. If we do not know the form of heteroskedasticity, it is better to use the B-P test.

In [ ]:
```python
import numpy as np
import pandas as pd
from scipy.stats import distributions as iid

def sin_gdp(N, b0, sigma2=1):
    '''
    Generate dataset for model with sinusoidal heteroskedasticity

    Takes as inputs sample size N and vector of true parameters (alpha and b

    Returns a tuple with numpy arrays y, X and Z, all of length N
    '''
    # Retrieve true parameters
    alpha = b0[0]
    beta = b0[1]

    # Construct X
    X = np.random.uniform(size=N, low=0, high=2*np.pi)

    # Construct u
    u = np.random.normal(size=N, loc=0, scale=np.sqrt(sigma2*np.abs(np.sin(2

    # Construct y
    y = alpha + X*beta + u

    # Construct df
    combined_dict = {'y':y, 'x':X, 'Constant':1}
    df = pd.DataFrame(combined_dict)

    return df['y'], df[['Constant','x']]
```

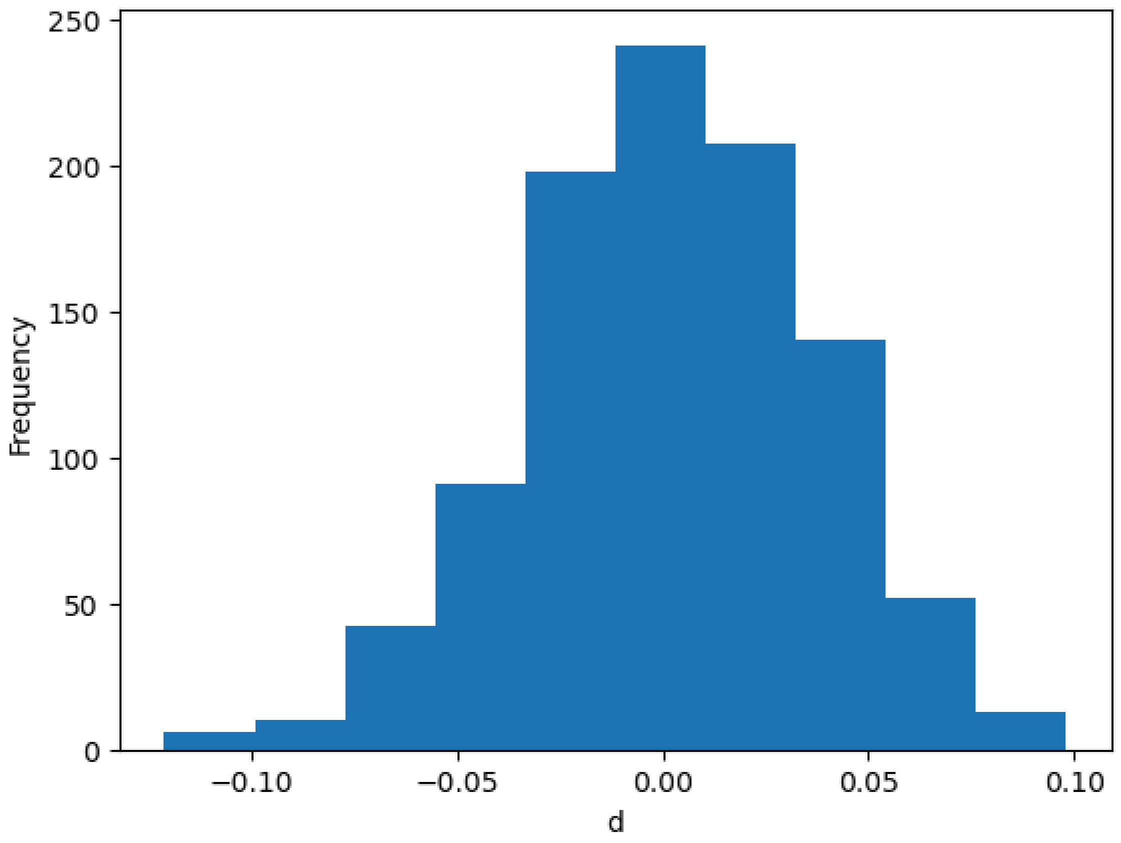
In [ ]:
```python
# Monte Carlo simulation
N = 1000 # Sample size
D = 1000 # Monte Carlo draws

# True parameters
```

```
alpha = 3
beta = 4
true_sigma2 = 2

d_draws_sin = pd.DataFrame(columns=['c', 'd'])
for i in range(D):
    y,X = sin_gdp(N, [alpha, beta], sigma2=true_sigma2)
    est = GMM(linear_regression, (y,X), 2)
    b, fun = est.two_step_gmm()
    u_hat2 = (y - X@b)**2
    bp_est = GMM(linear_regression, (u_hat2,X), 2)
    d, bp_fun = bp_est.two_step_gmm()
    d_df = pd.DataFrame(d, index=d_draws_sin.columns).T
    d_draws_sin = pd.concat([d_draws_sin, d_df], ignore_index=True)
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/1263452945.
py:19: FutureWarning: The behavior of DataFrame concatenation with empty or
all-NA entries is deprecated. In a future version, this will no longer exclu
de empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  d_draws_sin = pd.concat([d_draws_sin, d_df], ignore_index=True)
```

In [ ]:
```
plt.hist(d_draws_sin['d'])
plt.xlabel('d')
plt.ylabel('Frequency')
```

Out[ ]:
```
Text(0, 0.5, 'Frequency')
```



In [ ]:
```
from scipy.stats import t
```

```
d_avg = d_draws_sin['d'].mean()
var = np.sum(np.square(d_draws_sin['d'] - d_avg))
sd = np.sqrt(var)
se = sd/np.sqrt(len(d_draws_sin))

print('Average estimate: ',d_avg)
print('Standard error (precision): ', se)

# Size of two-tailed t-test for d = 0
t_score = (d_avg - 0)/se
n = len(d_draws_sin)
df = n - 1
pval = t.sf(np.abs(t_score), df) * 2
print('p-value: ', pval)
```

```
Average estimate:  0.002090164437928306
Standard error (precision):  0.03524100771100963
p-value:  0.9527166000294748
```

Alternatively, we can modify the GMM estimator applied to the auxiliary regression so that it has a sinusoidal functional form.

In [ ]:
```
def g_j_sin(b,data):
    """
    Return matrix u^2 - sigma2*sin(2x)

    sigma2 represents sigma^2
    """
    sigma2 = b[0]
    u2,X=data

    u2_df = pd.DataFrame(u2, columns=['x'])

    return u2_df - sigma2*np.abs(np.sin(2*X[['x']]))
```

In [ ]:
```
# Monte Carlo simulation
N = 1000 # Sample size
D = 1000 # Monte Carlo draws

# True parameters
alpha = 3
beta = 4
true_sigma2 = 2

sigma2_draws = pd.DataFrame(columns=['sigma2'])
for i in range(D):
    y,X = sin_gdp(N, [alpha, beta], sigma2=true_sigma2)
    est = GMM(linear_regression, (y,X), 2)
    b, fun = est.two_step_gmm()
    u_hat2 = (y - X@b)**2
    bp_est = GMM(g_j_sin, (u_hat2,X), 1)
    sigma2, bp_fun = bp_est.two_step_gmm()
    sigma2_df = pd.DataFrame(sigma2, index=sigma2_draws.columns).T
    sigma2_draws = pd.concat([sigma2_draws, sigma2_df], ignore_index=True)
```

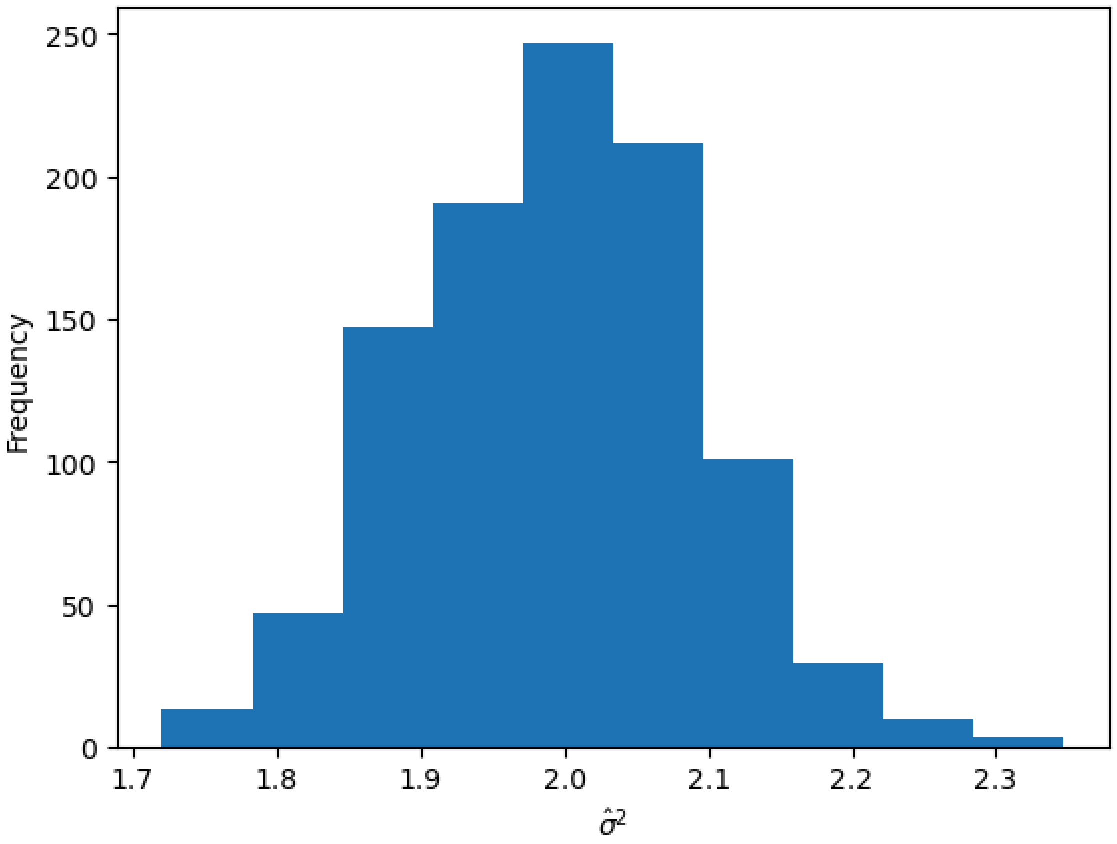
```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/1808578301.
py:19: FutureWarning: The behavior of DataFrame concatenation with empty or
all-NA entries is deprecated. In a future version, this will no longer exclu
de empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  sigma2_draws = pd.concat([sigma2_draws, sigma2_df], ignore_index=True)
```

In [ ]:

```
plt.hist(sigma2_draws['sigma2'])
plt.xlabel('$\hat{\sigma}^2$')
plt.ylabel('Frequency')
```

Out[ ]:

```
Text(0, 0.5, 'Frequency')
```



In [ ]:

```
from scipy.stats import t

sigma2_avg = sigma2_draws['sigma2'].mean()
var = np.sum(np.square(sigma2_draws['sigma2'] - sigma2_avg))
sd = np.sqrt(var)
se = sd/np.sqrt(len(sigma2_draws))

print('Average estimate: ',sigma2_avg)
print('Standard error (precision): ', se)

# Size of two-tailed t-test for sigma^2 = 0
t_score = (sigma2_avg - 0)/se
n = len(sigma2_draws)
df = n - 1
pval = t.sf(np.abs(t_score), df) * 2
print('p-value: ', pval)
```

```
Average estimate:  1.994537531037995
Standard error (precision):  0.09712492833721854
p-value:  1.8617232295101384e-78
```

(6) In the above, we've considered a test of a specific functional form for the variance of u. Suppose instead that we don't have any prior information regarding the form of $\mathbb{E}(u^2|x) = f(x)$. Discuss how you might go about constructing an extended version of the Breusch-Pagan test which tests for $f(x)$ non-constant.

We can adapt GMM to simulate feasible generalized least squares, which can handle many different forms of heteroskedasticity.

Specifically, we can use this form of heteroskedasticity, which can be used to fit many forms of heteroskedasticity:

$\sigma_i^2 = e^{\alpha_0+\alpha_1 Z_{1,i}+\ldots+\alpha_m Z_{m,i}}$

which can be rewritten as

$$log(\hat{\epsilon}_i^2) = \alpha_0 + \alpha_1 Z_{1,i} + \ldots + \alpha_m Z_{m,i} + log\left(\frac{\epsilon_i^2}{\sigma_i^2}\right)$$

where $\hat{\epsilon}_i^2$ is the OLS residuals from the first regression, $y = X\beta + \epsilon$

Alternative idea, more related to GLS: https://web.vu.lt/mif/a.buteikis/wp-content/uploads/2019/11/MultivariableRegression_4.pdf

(7) Show that you can use your ideas about estimating $f(x)$ to construct a more efficient estimator of β if $f(x)$ isn't constant. Relate your estimator to the optimal generalized least squares (GLS) estimator.

This estimator is less efficient than the optimal GLS estimator but its advantage is that it works with many different forms of heteroskedasticity.

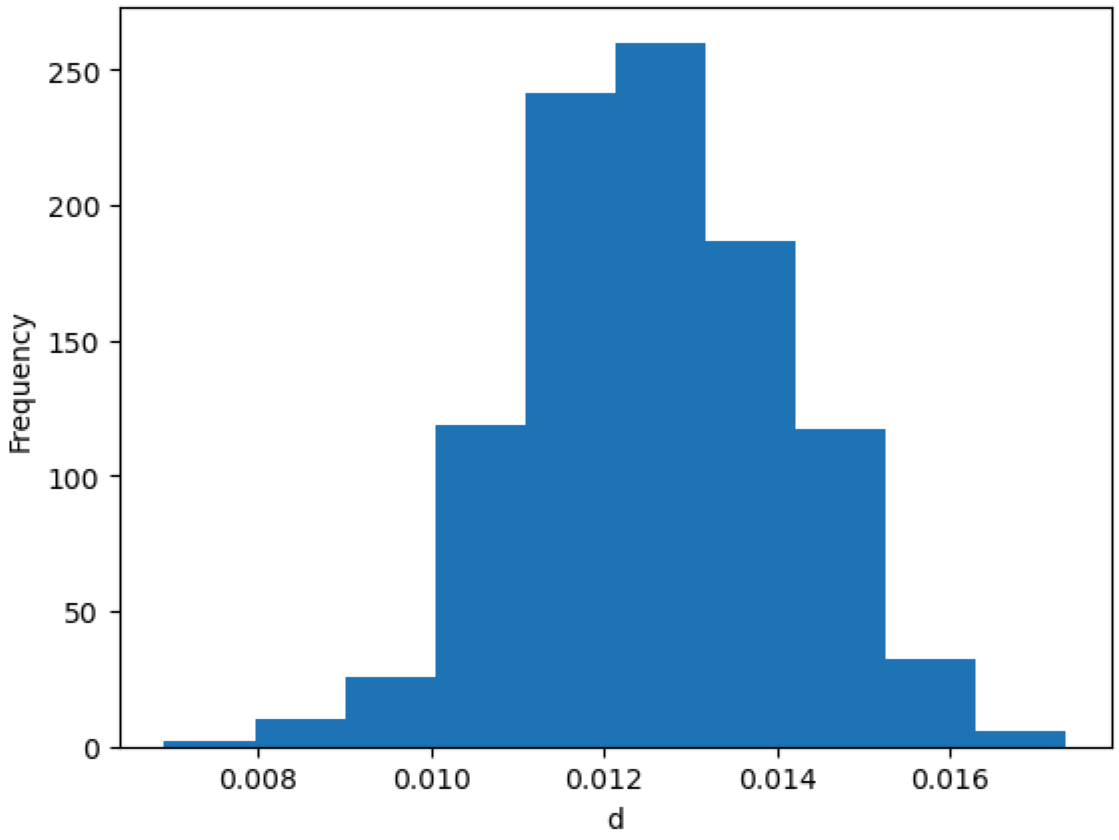
```
In [ ]: # Structural parameters
        N = 1000
        alpha = 3
        beta = 4
        true_c = 1
        true_d = 2

        D = 1000 # Monte Carlo draws
        b_draws = pd.DataFrame(columns=['a', 'b'])
        d_draws = pd.DataFrame(columns=['c', 'd'])
        for i in range(D):
```

```
    y,X = bp_gdp(N, [alpha, beta], true_c=true_c, true_d=true_d)
    est = GMM(linear_regression, (y,X), 2)
    b0, fun0 = est.two_step_gmm()
    log_e2 = np.log((y - X@b0)**2)
    bp_est = GMM(linear_regression, (log_e2,X), 2)
    d, bp_fun = bp_est.two_step_gmm()
    d_df = pd.DataFrame(d, index=d_draws.columns).T
    d_draws = pd.concat([d_draws, d_df], ignore_index=True)
    log_e2_fit = X@d
    h_fit = np.exp(log_e2_fit)
    bp_est2 = GMM(linear_regression, (y/np.sqrt(h_fit), (X.T/np.sqrt(h_fit))
    b, fun = bp_est2.two_step_gmm()
    b_df = pd.DataFrame(b, index=b_draws.columns).T
    b_draws = pd.concat([b_draws, b_df], ignore_index=True)
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/977415252.p
y:19: FutureWarning: The behavior of DataFrame concatenation with empty or a
ll-NA entries is deprecated. In a future version, this will no longer exclud
e empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  d_draws = pd.concat([d_draws, d_df], ignore_index=True)
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/977415252.p
y:25: FutureWarning: The behavior of DataFrame concatenation with empty or a
ll-NA entries is deprecated. In a future version, this will no longer exclud
e empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  b_draws = pd.concat([b_draws, b_df], ignore_index=True)
```

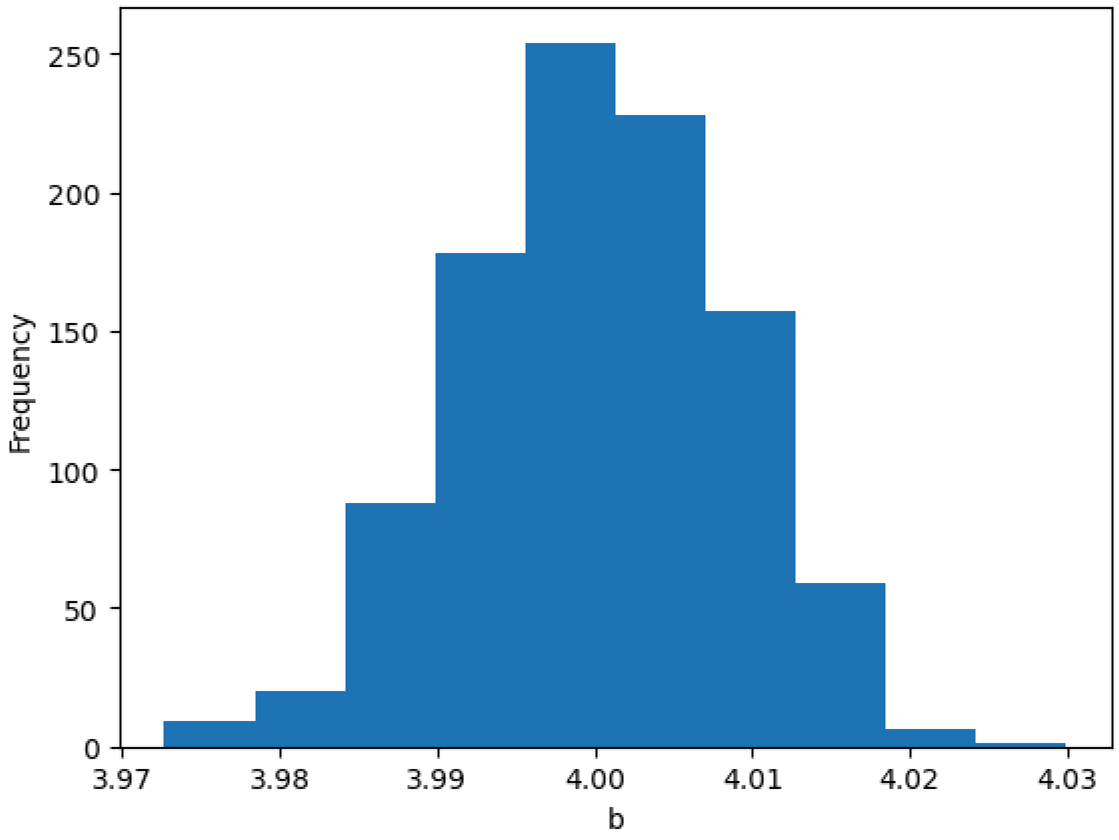
In [ ]:
```
import matplotlib.pyplot as plt

plt.hist(d_draws['d'])
plt.xlabel('d')
plt.ylabel('Frequency')
```

Out[ ]:
```
Text(0, 0.5, 'Frequency')
```



In [ ]:
```
import matplotlib.pyplot as plt

plt.hist(b_draws['b'])
plt.xlabel('b')
plt.ylabel('Frequency')
```

Out[ ]:
```
Text(0, 0.5, 'Frequency')
```



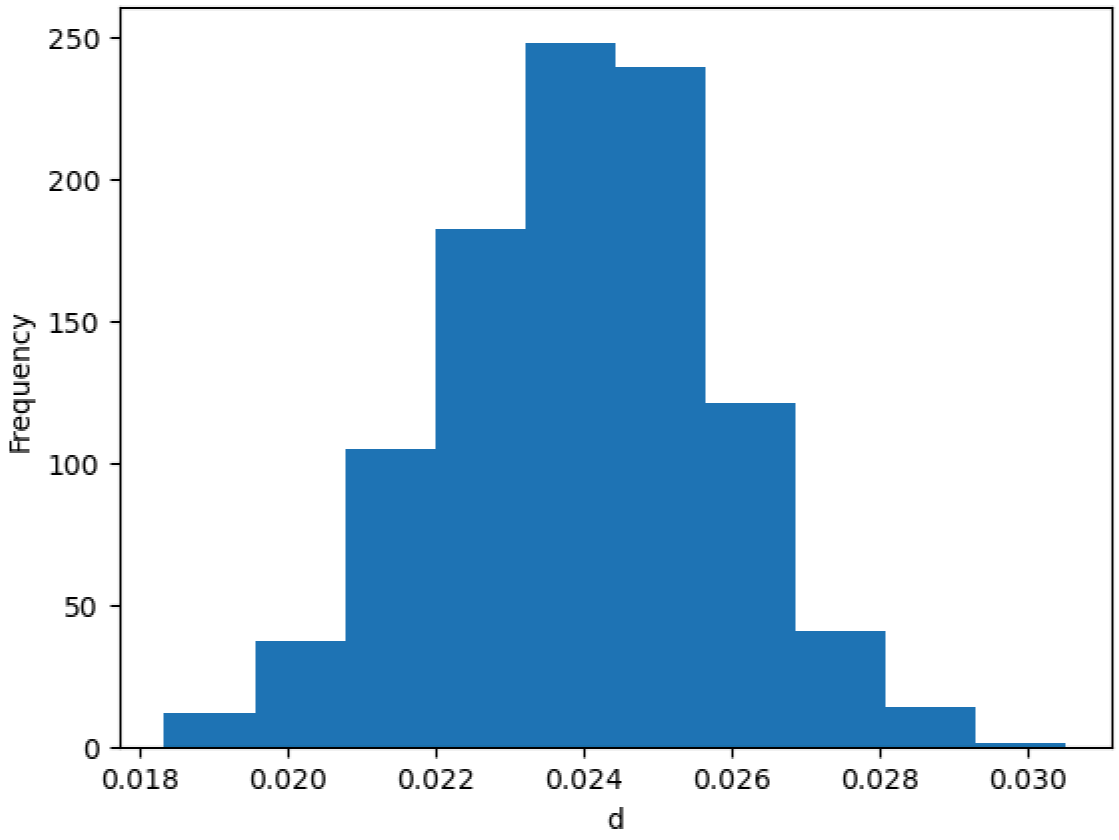
```
In [ ]: def quad_gdp(N, b0, true_c=0, true_d=0):
            '''
            Generate dataset for model with quadratic heteroskedasticity

            Takes as inputs sample size N and vector of true parameters (alpha and b

            Returns a tuple with numpy arrays y, X and Z, all of length N
            '''
            # Retrieve true parameters
            alpha = b0[0]
            beta = b0[1]

            # Construct X
            X = np.random.normal(size=N, loc=100, scale=50)

            # Construct u
            if true_d == 0:
                u = np.random.normal(size=N, loc=true_c, scale=1)
            else:
                u = np.random.normal(size=N, loc=true_c, scale=np.sqrt(true_d*X**2))

            # Construct y
            y = alpha + X*beta + u

            # Construct df
            combined_dict = {'y':y, 'x':X, 'Constant':1}
            df = pd.DataFrame(combined_dict)

            return df['y'], df[['Constant','x']]
```

In [ ]:
```
# Structural parameters
N = 1000
alpha = 3
beta = 4
true_c = 1
true_d = 2

# Generalized least squares method
D = 1000 # Monte Carlo draws
b_draws = pd.DataFrame(columns=['a', 'b'])
d_draws = pd.DataFrame(columns=['c', 'd'])
for i in range(D):
    y,X = quad_gdp(N, [alpha, beta], true_c=true_c, true_d=true_d)
    est = GMM(linear_regression, (y,X), 2)
    b0, fun0 = est.two_step_gmm()
    log_e2 = np.log((y - X@b0)**2)
    bp_est = GMM(linear_regression, (log_e2,X), 2)
    d, bp_fun = bp_est.two_step_gmm()
    d_df = pd.DataFrame(d, index=d_draws.columns).T
    d_draws = pd.concat([d_draws, d_df], ignore_index=True)
    log_e2_fit = X@d
    h_fit = np.exp(log_e2_fit)
    bp_est2 = GMM(linear_regression, (y/np.sqrt(h_fit), (X.T/np.sqrt(h_fit))
    b, fun = bp_est2.two_step_gmm()
    b_df = pd.DataFrame(b, index=b_draws.columns).T
    b_draws = pd.concat([b_draws, b_df], ignore_index=True)
```

```
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/1700492155.
py:20: FutureWarning: The behavior of DataFrame concatenation with empty or
all-NA entries is deprecated. In a future version, this will no longer exclu
de empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  d_draws = pd.concat([d_draws, d_df], ignore_index=True)
/var/folders/_1/m0sq7yvd6lx8srw1yzkxmlk40000gn/T/ipykernel_77082/1700492155.
py:26: FutureWarning: The behavior of DataFrame concatenation with empty or
all-NA entries is deprecated. In a future version, this will no longer exclu
de empty or all-NA columns when determining the result dtypes. To retain the
old behavior, exclude the relevant entries before the concat operation.
  b_draws = pd.concat([b_draws, b_df], ignore_index=True)
```

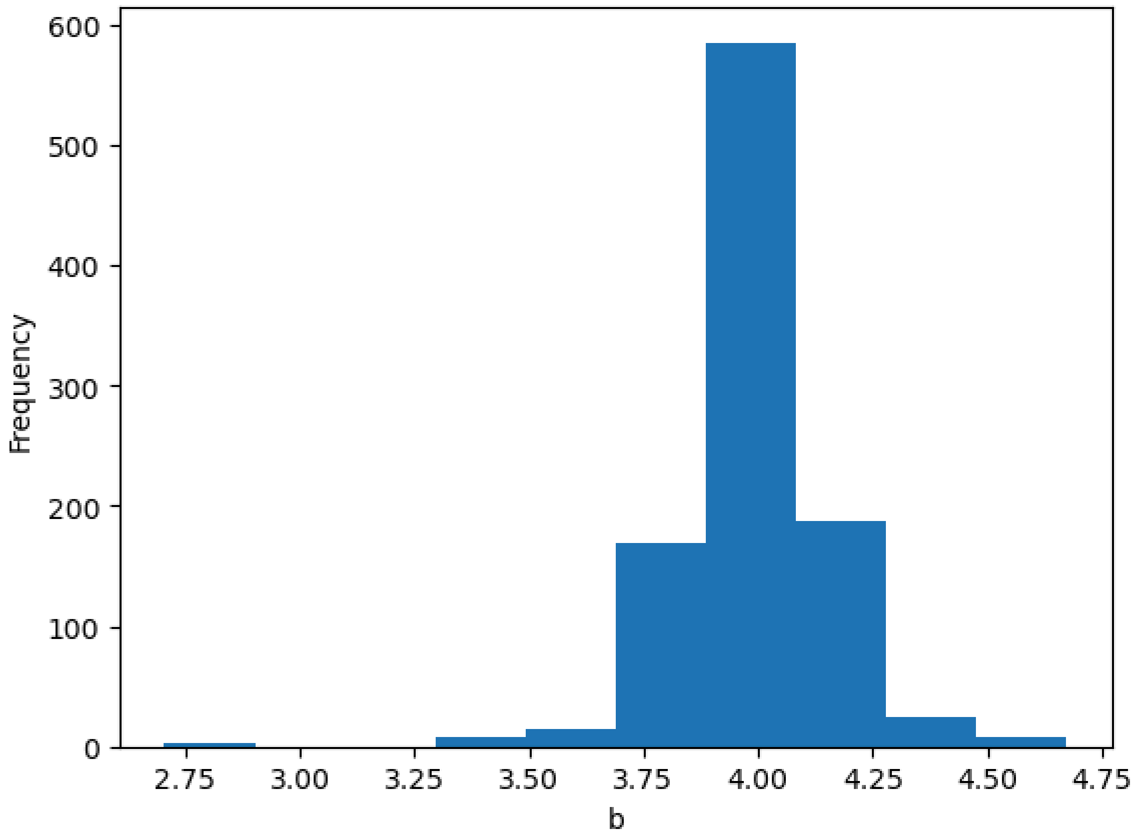
In [ ]:
```
import matplotlib.pyplot as plt

plt.hist(d_draws['d'])
plt.xlabel('d')
plt.ylabel('Frequency')
```

Out[ ]:
```
Text(0, 0.5, 'Frequency')
```



```
In [ ]: import matplotlib.pyplot as plt

        plt.hist(b_draws['b'])
        plt.xlabel('b')
        plt.ylabel('Frequency')
```

```
Out[ ]: Text(0, 0.5, 'Frequency')
```



4. Tests of Normality

Suppose we have a sample of iid observations $x_1, x_2, \ldots, x_N$; we want to test whether these are drawn from a normal distribution. Note the fact that the integer central moments of the normal distribution satisfy

$Ex = \mu$

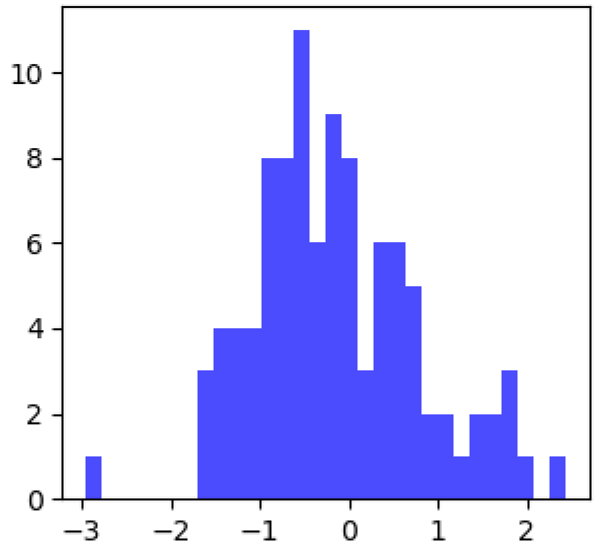
$E(x-\mu)^m = 0, \quad if m$ odd

$E(x-\mu)^m = \sigma^m(m-1)!! \quad if m$ even

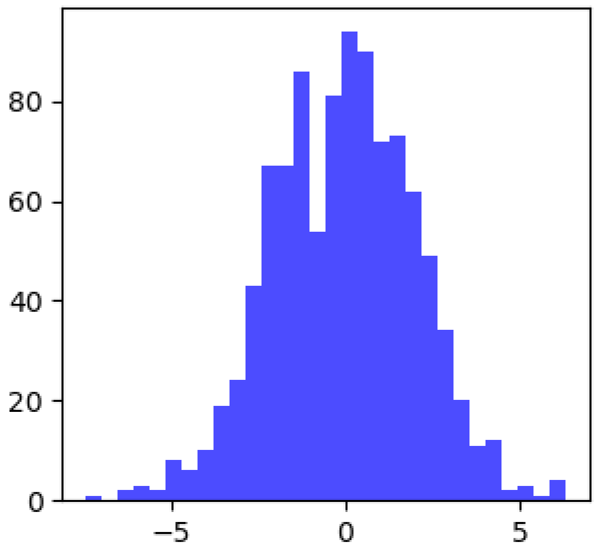
where n!! is the double factorial, i.e., $n!! = n(n-2)(n-4)\ldots$

1. Using the analogy principle, construct an estimator for the first k moments of the distribution of x. Use this to define a k-vector of moment restrictions $g_N(\mu,\sigma)$ satisfying $E_{gN}(\mu,\sigma) = 0$ under the null hypothesis of normality.

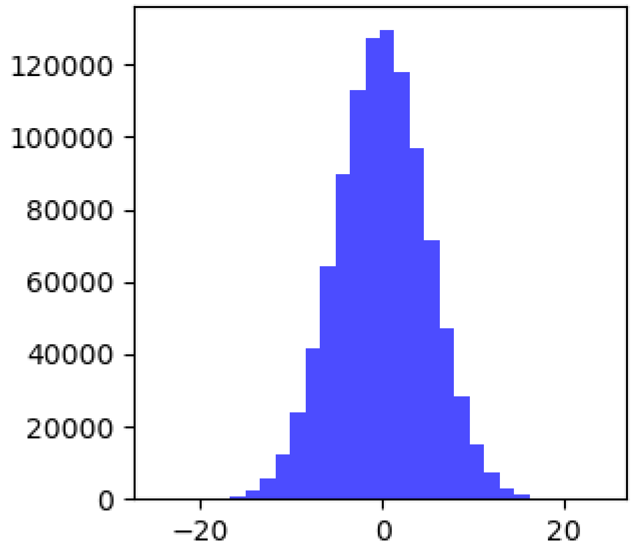
In [ ]:
```
import numpy as np
from scipy.special import factorial2
import matplotlib.pyplot as plt

def sample_moments(data, k):
    mu_hat = np.mean(data)  # First moment: Mean
    sigma2_hat = np.var(data)  # Second moment: Variance
    moments = [mu_hat, sigma2_hat]

    for m in range(3, k+1):  # Start calculating from the third moment
        central_moment = np.mean((data - mu_hat)**m)
        moments.append(central_moment)

    return moments

def moment_restrictions(data, k):
    moments = sample_moments(data, k)
    restrictions = [moments[0], moments[1] - 1]  # Check if mean is zero and

    for m in range(3, k+1):
        if m % 2 == 1:  # Odd moments should be close to zero for normal dis
            restrictions.append(moments[m-1])  # Higher odd moments
        else:  # Even moments
            sigma_hat = np.sqrt(moments[1])  # Standard deviation
            theoretical_moment = sigma_hat**m * factorial2(m-1)
            restrictions.append(moments[m-1] - theoretical_moment)
    return restrictions

def plot_diagnostics(data):
    plt.figure(figsize=(6, 3))
    plt.subplot(1, 2, 1)
    plt.hist(data, bins=30, alpha=0.7, color='blue')
    plt.tight_layout()
    plt.show()


# Example usage
data = np.random.normal(0, 1, 100)  # Sample data from a standard normal dis
plot_diagnostics(data)
k = 4  # Number of moments to consider
print("First example")
print(moment_restrictions(data, k))


np.random.seed(12)
data = np.random.normal(0, 2, 1000)  # Sample data from a standard normal di
plot_diagnostics(data)
k = 5  # Number of moments to consider
print("Second example")
print(moment_restrictions(data, k))


data = np.random.normal(0, 5, 1000000)  # Sample data from a standard normal
plot_diagnostics(data)
k = 5  # Number of moments to consider
print("Third example")
print(moment_restrictions(data, k))
```



```
First example
[-0.13468204713112125, -0.10766458317481398, 0.26595133317395364, 0.20477182
993080456]
```



```
Second example
[-0.023240453660223766, 3.211943079723471, -0.594954535613723, 0.84168783990
78414, -27.036894496511035]
```



```
Third example
[-0.012786586974319598, 23.984885763915596, 0.29740558508727755, -0.57020174
52156269, 29.244790909209676]
```

2. What is the covariance matrix of the sample moment restrictions (again under the null)? I.e., what can be said about $Eg_j(\mu,\sigma)g_j(\mu,\sigma)^T - Eg_j(\mu,\sigma)Eg_j(\mu,\sigma)^T$?

$Eg_j(\mu,\sigma)g_j(\mu,\sigma)^T$ --- This term represents the expected value of the outer product of the vector of moment restrictions with itself. This results in a matrix where each element is the expected product of two moment conditions.

$Eg_j(\mu,\sigma)Eg_j(\mu,\sigma)^T$ --- Since under the null hypothesis $Eg_j(\mu,\sigma) = 0$ (because each is g_N is constructed to be zero when the data follow a normal distribution), this term simplifies to a matrix of zeros.

Thus, the expression simplifies to: $Cov[g_j(\mu,\sigma)] = Eg_j(\mu,\sigma)g_j(\mu,\sigma)^T$ because $Eg_j(\mu,\sigma)Eg_j(\mu,\sigma)^T$ is a matrix of zeros.

3. Using your answers to the previous two questions, suggest a GMM-based test of the hypothesis of normality, taking $k > 2$.

The Generalized Method of Moments (GMM) is a statistical method used to estimate parameters of a statistical model. It uses sample moments (averages computed from data) that are known or hypothesized based on the model. In the context of testing normality, we are testing whether the sample data comes from a normal distribution.

Moment restrictions are functions of data that, under the null hypothesis (the data is normally distributed), are expected to equal zero. For a normal distribution, certain properties hold:

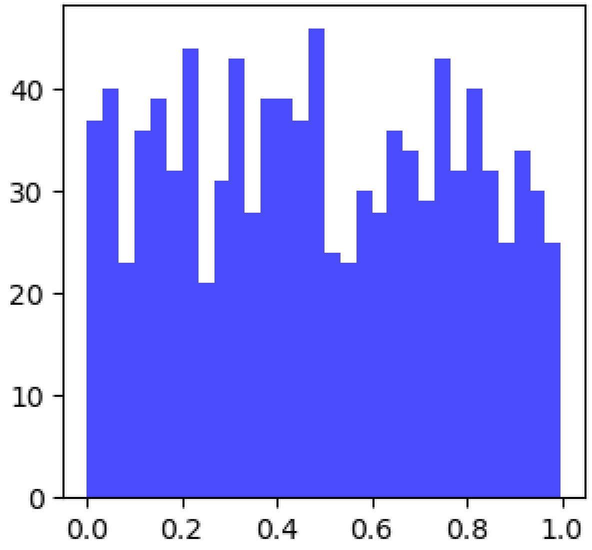
- Odd central moments are zero.

- Even central moments are defined by $\sigma^m(m-1)$ where σ is the standard deviation of the distribution and m is the moment order.

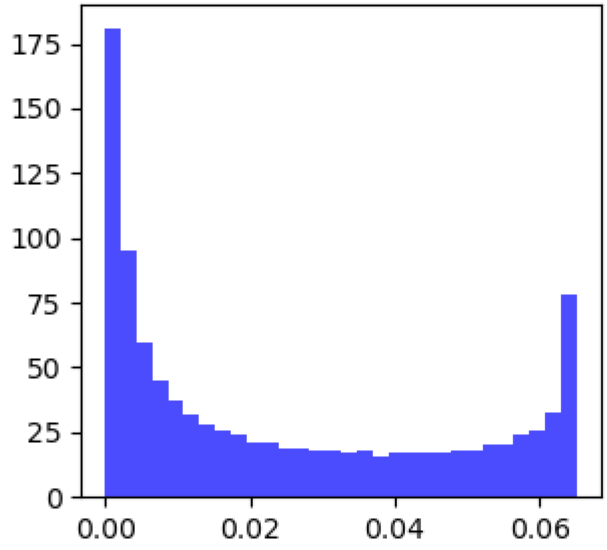
4. Implement the test you've devised using python. You may want to use scipy.stats.distributions.chi2.cdf and scipy.optimize.minimize.

In [ ]:
```
import numpy as np
from scipy.optimize import minimize
from scipy.stats import chi2
from scipy.special import factorial2
from scipy import stats

def moment_restrictions(data, params, k):
    mu, sigma = params
    moments = []
    for m in range(1, k+1):
        if m % 2 == 1:  # Odd moments should ideally be zero
            moment = np.mean((data - mu) ** m)
        else:  # Even moments, adjust by theoretical moments
            theoretical_moment = sigma ** m * factorial2(m - 1)
            moment = np.mean((data - mu) ** m) - theoretical_moment
        moments.append(moment)
    return np.array(moments)

def bootstrap_covariance(data, params, k, B=1000):
    bootstrap_samples = np.random.choice(data, (B, len(data)), replace=True)
    moment_samples = np.array([moment_restrictions(sample, params, k) for sa
    return np.cov(moment_samples, rowvar=False)

def gmm_objective(params, data, cov_inv, k):
    g = moment_restrictions(data, params, k)
    return g.T @ cov_inv @ g

def gmm_test_normality(data, k, initial_params):
    cov_matrix = bootstrap_covariance(data, initial_params, k)
    cov_inv = np.linalg.inv(cov_matrix)
    result = minimize(gmm_objective, initial_params, args=(data, cov_inv, k)

    test_statistic = result.fun
    p_value = 1 - chi2.cdf(test_statistic, df=k-2)

    print(f"Test statistic: {test_statistic}, p-value: {p_value}")
    if p_value < 0.05:
        print("Reject the null hypothesis of normality.")
    else:
        print("Do not reject the null hypothesis of normality.")

# Example usage:

data = np.random.uniform(0, 1, size=1000)
k = 4
initial_params = [np.mean(data), np.std(data)]
plot_diagnostics(data)
#plt.title('Random uniform distribution')
```

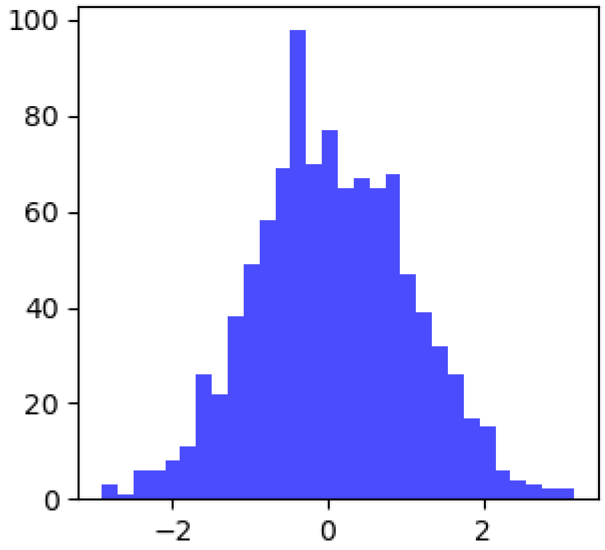
```python
gmm_test_normality(data, k, initial_params)


x = np.linspace (0, 40, 1000)
data = stats.gamma.pdf(x, a=5, scale=3)
k = 4
initial_params = [np.mean(data), np.std(data)]
plot_diagnostics(data)
#plt.title('Gamma distribution')
gmm_test_normality(data, k, initial_params)


data = np.random.normal(0, 1, size=1000)
k = 4
initial_params = [np.mean(data), np.std(data)]
plot_diagnostics(data)
#plt.title('Normal distribution')
gmm_test_normality(data, k, initial_params)
```



```
Test statistic: 364.94862811769866, p-value: 0.0
Reject the null hypothesis of normality.
```



```
Test statistic: 1243.1138411219276, p-value: 0.0
Reject the null hypothesis of normality.
```



```
Test statistic: 2.4969300623210136, p-value: 0.28694491048967563
Do not reject the null hypothesis of normality.
```

5. What can be said about the optimal choice of k?

The optimal choice of k, the number of moment conditions used in a GMM approach, is crucial and can significantly influence the test's power and size.

- When the number of moment conditions exceeds the number of parameters being estimated, the model becomes over-identified.
- Using more moment conditions can potentially make the estimator more efficient because it utilizes more information about the distribution. This is generally true up to a point where the added moments bring in relevant additional information.
- For a normality test, including higher than the second moments is important because the normal distribution is completely characterized by its first two moments (mean and variance). Moments higher than the second (skewness, kurtosis, etc.) can provide crucial information about departures from normality (e.g., asymmetry, tail heaviness).
- As k increases, the moments might become increasingly difficult to estimate accurately, especially for small sample sizes. Moments of higher order are also more sensitive to outliers and can become numerically unstable.

6. Compare the GMM estimates of (μ, σ) to the maximum likelihood estimates of these parameters. Do they differ? Why

- MLE directly aims to find the parameter values that maximize the likelihood function given the data. For the normal distribution, MLE for (μ, σ) are the sample mean and sample standard deviation, respectively, making it very efficient (in the statistical sense) and unbiased in large samples.

- MLE is known for being asymptotically efficient for many models, including the normal distribution. This means it achieves the lowest possible variance among all unbiased estimators as the sample size grows.
- GMM is based on the method of moments, where parameters are chosen so that the sample moments (often beyond the first two) match their theoretical counterparts as closely as possible. This can be particularly useful when multiple moment conditions are involved, or the distributional assumptions for MLE are not fully justified.
- GMM can be more robust to certain specification errors or when dealing with distributions where MLE is difficult to implement or unreliable due to model complexity or constraints.
- For the normal distribution, if the model is correctly specified and the sample size is large, GMM and MLE for (μ, σ) should be quite similar, especially if the GMM uses the first and second moments as part of its conditions.
- If there are deviations from normality or if additional moments (like skewness and kurtosis) are incorporated into the GMM, the estimates might differ. GMM might then provide an estimate that compensates for these higher-order characteristics, whereas MLE would strictly interpret parameters under the assumption of normality.
- In small samples, MLE can be biased (particularly the estimate for (σ), whereas GMM might also suffer from high variability depending on the moments used.

5. Logit

This problem is meant to help draw connections between GMM estimators and maximum likelihood estimators, with a particular focus on the 'logit' model.

The development of a maximum likelihood estimator typically begins with an assumption that some random variable has a (conditional) distribution which is made up of a k-vector of parameters β. Consider the case in which we observe N independent realizations of a Bernoulli random variable $\textbackslash\text{textcolor}redY$, with
$Pr(\textbackslash\text{textcolor}redY = 1|X) = \sigma(\beta^T X)$, and
$Pr(\textbackslash\text{textcolor}redY = 0|X) = 1 - \sigma(\beta^T X)$.

(1). Show that under this model $\mathbb{E}(\textbackslash\text{textcolor}redY - \sigma(X\beta)|X) = 0$. Assume that σ is a known function, and use this fact to develop a GMM estimator of β. Is your estimator just- or over-identified?

In the logit model, the function σ typically represents the logistic function, which for a given vector of parameters β and a matrix of predictors X is defined as:

$$\sigma(\beta^T X) = \frac{1}{1 + e^{-\beta^T X}}$$

Given that $\textbackslash\text{textcolor}redY$ is a Bernoulli random variable, its expected value given X is the probability that $\textbackslash\text{textcolor}redY = 1$:

$$\mathbb{E}(\backslash\mathrm{textcolor}redY|X) = Pr(\backslash\mathrm{textcolor}redY = 1|X) = \sigma(\beta^T X)$$

Hence,

$$\mathbb{E}(\backslash\mathrm{textcolor}redY - \sigma(X\beta)|X) = \mathbb{E}(\backslash\mathrm{textcolor}redY|X) - \sigma(X\beta) = \sigma(\beta^T X) - \sigma(X$$

This follows directly from the definition of the expected value of a Bernoulli random variable and the assumption that the probability of success is given by the logistic function $\sigma(X\beta)$.

Developing a GMM Estimator of β

The moment condition for the GMM estimator is given by the expectation we just calculated:

$$\mathbb{E}(\backslash\mathrm{textcolor}redY - \sigma(X\beta)|X) = 0$$

This can be turned into a set of sample moment conditions by using the sample analog:

$$\frac{1}{N}\sum_{i=1}^{N}(Y_i - \sigma(X_i\beta))X_i = 0$$

The GMM estimator for β can then be obtained by solving the following minimization problem:

$$\hat{\beta}_{GMM} = \arg\min_{\beta}\left[\frac{1}{N}\sum_{i=1}^{N}(Y_i - \sigma(X_i\beta))X_i\right]^T W\left[\frac{1}{N}\sum_{i=1}^{N}(Y_i - \sigma(X_i\beta))X_i\right]$$

Where W is a weight matrix, which is often chosen to be the inverse of the covariance matrix of the moment conditions for efficiency.

Identification of the Estimator

The model will be just-identified if the number of moment conditions equals the number of parameters to be estimated. In this case, since there is one moment condition for each parameter in β, the estimator is just-identified. If there were more moment conditions than parameters, it would be over-identified.

(2). Show that the likelihood conditional on realizations of data (y, X) can be written as

$$L(\beta|y,X) = \prod_{i=1}^{N}\sigma(\beta^T X_i)^{y_i}\left(1 - \sigma(\beta^T X_i)\right)^{1-y_i}.$$

In a logit model, the likelihood of observing the given data (y, X) is the product of the probabilities of each individual observation, where y_i is the realization of the Bernoulli random variable for the i-th observation, and X_i is the vector of predictors for that observation. The probability of y_i being 1 is $\sigma(\beta^T X_i)$, and the probability of y_i being 0 is $1-\sigma(\beta^T X_i)$.

The likelihood for a single observation is therefore $\sigma(\beta^T X_i)$ when $y_i = 1$ and $1-\sigma(\beta^T X_i)$ when $y_i = 0$. We can combine these two cases into one expression using the fact that y_i can only be 0 or 1:

$$\text{For a single observation, } i: L(\beta|y_i, X_i) = \sigma(\beta^T X_i)^{y_i}\left(1-\sigma(\beta^T X_i)\right)^{1-y_i}$$

Since the observations are independent, the likelihood for all observations is the product of the individual likelihoods:

$$L(\beta|y, X) = \prod_{i=1}^{N} L(\beta|y_i, X_i) = \prod_{i=1}^{N} \sigma(\beta^T X_i)^{y_i}\left(1-\sigma(\beta^T X_i)\right)^{1-y_i}$$

This likelihood function reflects the product of the probabilities across all observations, each raised to the power of the observed y_i value (which is either 0 or 1), thereby capturing the presence or absence of the event $y_i = 1$.

(3). To obtain the maximum likelihood estimator (MLE) one can chose b to maximize $\log L(b|y, X)$. When the likelihood is well-behaved, the MLE estimator satisfies the first order conditions (also called the "scores") from this maximization problem, in which case this is called a "type I" MLE. Let $\sigma(z) = \frac{1}{1+e^{-z}}$ (this is sometimes called the logistic function, or the sigmoid function), and obtain the scores $S_N(b)$ for this estimation problem. Show that $\mathbb{E} S_N(\beta) = 0$. Demonstrate that these moment conditions can serve as the basis for a GMM estimator of β, and compare this estimator to the GMM estimator you developed above. Which is more efficient, and why?

- Derivation of Scores for MLE

In the logistic regression context, the log likelihood function $logL(b|y, X)$ for parameters b given binary outcomes y and predictors X is the natural logarithm of the likelihood function $L(b|y, X)$. The likelihood function itself was established as:

$L(b|y, X) = \prod_{i=1}^{N} \sigma(b^T X_i)^{y_i}\left(1-\sigma(b^T X_i)\right)^{1-y_i}$

Thus, the log likelihood becomes:

$logL(b|y, X) = \sum_{i=1}^{N} \left[y_i log\sigma(b^T X_i) + (1-y_i) log(1-\sigma(b^T X_i))\right]$

- Score Function $S_N(b)$

To obtain the maximum likelihood estimator, we need the gradient (or score) of the log likelihood function with respect to b. The derivative of the log likelihood involves the derivative of the logistic function $\sigma(z)$, which is:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Thus, the derivative of the log likelihood with respect to b is:

$$\frac{\partial}{\partial b} logL(b|y, X) = \sum_{i=1}^{N} X_i(y_i - \sigma(b^T X_i))$$

This expression $\sum_{i=1}^{N} X_i(y_i - \sigma(b^T X_i))$ is the score function $S_N(b)$.

- Expectation $E[S_N(\beta)] = 0$

Under the true parameter β, the expected value of y_i given X_i is $\sigma(\beta^T X_i)$, and thus:

$$E[y_i - \sigma(\beta^T X_i)|X_i] = \sigma(\beta^T X_i) - \sigma(\beta^T X_i) = 0$$

Given the independence of y_i from y_j for $i \neq j$ given X:

$$E[S_N(\beta)] = E\left[\sum_{i=1}^{N} X_i(y_i - \sigma(\beta^T X_i))\right] = \sum_{i=1}^{N} E[X_i(y_i - \sigma(\beta^T X_i))] = 0$$

- Using $S_N(\beta)$ for GMM Estimation

The score function can serve as the moment conditions for a GMM estimator. The GMM objective would minimize the quadratic form:

$$\min_b \left(S_N(b)^T W S_N(b)\right)$$

where W is a weighting matrix.

Interpreting Efficiency

An efficient estimator is one that achieves the smallest variance among all unbiased estimators. In other words, it is the estimator that uses the information provided by the data most fully.

For the MLE, efficiency means that the estimator asymptotically achieves the Cramér-Rao lower bound (CRLB), indicating that it has the smallest variance that any unbiased estimator can have based on the information available in the data. When we say that the scores are efficient, it implies that the estimator uses all the information necessary to reach the CRLB.

The MLE is typically more efficient than the GMM estimator when it exploits the full distributional assumptions of the data. This is because MLE is specifically tailored to the likelihood function derived from the underlying data distribution, whereas GMM is a more general approach that does not require full specification of the distribution.

Therefore, under the conditions where the scores are efficient, and the MLE makes full use of the data distribution information, the MLE will be more efficient than the GMM estimator, which uses the scores as moment conditions. The MLE's efficiency comes from its exploitation of the entire data structure, as provided by the likelihood function, to achieve the lower bound on the variance of the estimate.