Assignment 2

April 20, 2024

This document describes our process to code the conditional and unconditional likelihood functions for the AR(7) process based on the starter code provided in the assignment. Furthermore, it outlines our approach to estimate the AR(7) paramteres for the `INDPRO` variable from the FRED-MD dataset and our forecast.

Contents

1 Conditional Likelihood

1.1 Theoretical Outline

The AR(7) Model is defined as:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \phi_3 y_{t-3} + \phi_4 y_{t-4} + \phi_5 y_{t-5} + \phi_6 y_{t-6} + \phi_7 y_{t-7} + u_t \tag{1}$$

where $u_t \sim N(0, \sigma^2)$.

We can then define the conditional likelihood function as:

$$L(y, \phi) = f(y_1, y_2, y_3, y_4, y_5, y_6, y_7; \phi) \prod_{t=8}^{T} f(y_t | y_{t-1}, ..., y_{t-7}; \phi) \tag{2}$$

The conditional log-likelihood function is:

$$\ell(y; \phi) = \log(f(y_1, ..., y_7; \phi)) + \sum_{t=8}^{T} \log(f(y_t | y_{t-1}, ..., y_{t-7}; \phi)) \tag{3}$$

We regard the first 7 observation or their joint distribution as a determenistic quantity and thus:

$$\ell(y; \phi) \propto \sum_{t=8}^{T} \log(f(y_t | y_{t-1}, ..., y_{t-7}; \phi)) \tag{4}$$

Also, since the error terms are normally distributed with mean 0 and varieance σ^2, we can determine the conditional distribution of y_t as:

$$y_t | y_{t-1}, y_{t-2}, ..., y_{t-7} \sim N(c + \phi_1 y_{t-1} + ... + \phi_7 y_{t-7}, \sigma^2) \tag{5}$$

Therefore, to implement the conditional log-likelihood function we can code it as the sum of normally distributed variables.

$$\ell_C(y; \phi) \propto \sum_{t=8}^{T} \log \left(\frac{1}{\sigma\sqrt{2\pi}} \exp \left\{ - \frac{(y_t - (c + \phi_1 y_{t-1} + ... + \phi_7 y_{t-7}))^2}{2\sigma^2} \right\} \right) \tag{6}$$

1.2 Implementation in Python

```
from scipy.stats import norm
from scipy.stats import multivariate_normal
import numpy as np

def lagged_matrix(Y, max_lag=7):
    n = len(Y)
```

```
    lagged_matrix = np.full((n, max_lag), np.nan)
    # Fill each column with the appropriately lagged data
    for lag in range(1, max_lag + 1):
        lagged_matrix[lag:, lag - 1] = Y[:-lag]
    return lagged_matrix


def cond_loglikelihood_ar7(params, y):
    c = params[0]
    phi = params[1:8]
    sigma2 = params[8]
    mu, Sigma, stationary = unconditional_ar_mean_variance(c, phi, sigma2)
    ## We could check that at phis the process is stationary and return -Inf if it is not
    if not(stationary):
        return -np.inf
    ## The distribution of
    # y_t|y_{t-1}, ..., y_{t-7} ~ N(c+\phi_{1}*y_{t-1}+...+\phi_{7}y_{t-7}, sigma2)
    ## Create lagged matrix
    X = lagged_matrix(y, 7)
    yf = y[7:]
    Xf = X[7:,:]
    loglik = np.sum(norm.logpdf(yf, loc=(c + Xf@phi), scale=np.sqrt(sigma2)))
    return loglik
```

2 Unconditional Likelihood

2.1 Theoretical Outline

The unconditional likelihood function starts from a similar approach. However, instead of regarding the first 7 observations as determenistic it regards them as multivariate normally distributed with mean μ and covariance Σ.

$$\ell_U(y;\phi) = \log(f(y_1,...,y_7;\phi,\mu,\Sigma)) + \sum_{t=8}^{T} \log(f(y_t|y_{t-1},...,y_{t-7};\phi)) \tag{7}$$

The second term on the right hand side is the conditional log-likelihood, the first term is the multivariate normal distribution of the first seven observations.

As described in the assignment the parameters of the multivariate normal distribution can be obtained through:

$$\mu = (I - A)^{-1}b \tag{8}$$

to compute the mean and the Lyapunov equation

$$\Sigma = A\Sigma A^T + Q \tag{9}$$

to compute the covariance matrix of the autoregressive process.

2.2 Implementation in Python

First, the matrix A is constructed, which is a square matrix where the first row is made up of $\phi_1, \phi_2, ..., \phi_p$ of the autoregressive process of order p and the other rows are in reduced row echelon form.

For this the command `np.zeros((p,p))` is used, which creates an array of dimension $p \times p$ filled with zeros. Next, we replace the first row with $\phi_1, \phi_2, ..., \phi_p$ by subsetting A with the command `A[0, :] = phis`. Next, to fill the remaining rows except the last row with leading ones the comman np.eye is used.

Thus for given p and vector ϕ, this code creates the matrix A and then calculates the mean and the covariance. The code also checks whether the AR(7) process is stationary with respect to the given parameters.

```
def unconditional_ar_mean_variance(c, phis, sigma2):
    ## The length of phis is p
    p = len(phis)
    A = np.zeros((p, p))
    A[0, :] = phis
    A[1:, 0:(p-1)] = np.eye(p-1)
    ## Check for stationarity
    eigA = np.linalg.eig(A)
    if all(np.abs(eigA.eigenvalues)<1):
        stationary = True
    else:
        stationary = False
    # Create the vector b
    b = np.zeros((p, 1))
    b[0, 0] = c

    # Compute the mean using matrix algebra
    I = np.eye(p)
    mu = np.linalg.inv(I - A) @ b

    # Solve the discrete Lyapunov equation
    Q = np.zeros((p, p))
    Q[0, 0] = sigma2
    #Sigma = np.linalg.solve(I - np.kron(A, A), Q.flatten()).reshape(7, 7)
    Sigma = scipy.linalg.solve_discrete_lyapunov(A, Q)

    return mu.ravel(), Sigma, stationary
```

This function is used to build the unconditional likelihood function:

```
def uncond_loglikelihood_ar7(params, y):
    ## The unconditional loglikelihood
    ## is the unconditional "plus" the density of the
    ## first p (7 in our case) observations
    cloglik = cond_loglikelihood_ar7(params, y)

    ## Calculate initial
    # y_1, ..., y_7 ~ N(mu, sigma_y)
    c = params[0]
    phi = params[1:8]
    sigma2 = params[8]
    mu, Sigma, stationary = unconditional_ar_mean_variance(c, phi, sigma2)
    if not(stationary):
        return -np.inf
    mvn = multivariate_normal(mean=mu, cov=Sigma, allow_singular=True)
    uloglik = cloglik + mvn.logpdf(y[0:7])
    return uloglik
```

3 Maximum Likelihood Estimation for INDPRO

As in assignment 1, we import the FRED-MD dataset using `pandas`. Since we are only interested in the `INDPRO` variable, we select it and transform it using log differences:

```
import pandas as pd
import numpy as np
#Read Data
df = df = pd.read_csv('~/Downloads/current.csv')
```

```
#Select INDPRO
INDPRO = df['INDPRO']
#Drop first Row
INDPRO = INDPRO.drop(index=0)
#transform INDPRO using log differences
INDPRO = np.log(INDPRO).diff()

#implement Starter Code from the assignment
## Lagged Matrix Function
def lagged_matrix(Y, max_lag=7):
    n = len(Y)
    lagged_matrix = np.full((n, max_lag), np.nan)
    # Fill each column with the appropriately lagged data
    for lag in range(1, max_lag + 1):
        lagged_matrix[lag:, lag - 1] = Y[:-lag]
    return lagged_matrix

## Mean- Variance - Stationarity Function
def unconditional_ar_mean_variance(c, phis, sigma2):
    ## The length of phis is p
    p = len(phis)
    A = np.zeros((p, p))
    A[0, :] = phis
    A[1:, 0:(p-1)] = np.eye(p-1)
    ## Check for stationarity
    eigA = np.linalg.eig(A)
    if all(np.abs(eigA.eigenvalues)<1):
        stationary = True
    else:
        stationary = False
    # Create the vector b
    b = np.zeros((p, 1))
    b[0, 0] = c

    # Compute the mean using matrix algebra
    I = np.eye(p)
    mu = np.linalg.inv(I - A) @ b

    # Solve the discrete Lyapunov equation
    Q = np.zeros((p, p))
    Q[0, 0] = sigma2
    #Sigma = np.linalg.solve(I - np.kron(A, A), Q.flatten()).reshape(7, 7)
    Sigma = scipy.linalg.solve_discrete_lyapunov(A, Q)

    return mu.ravel(), Sigma, stationary

## Conditional Likelihood
def cond_loglikelihood_ar7(params, y):
    c = params[0]
    phi = params[1:8]
    sigma2 = params[8]
    mu, Sigma, stationary = unconditional_ar_mean_variance(c, phi, sigma2)
    ## We could check that at phis the process is stationary and return -Inf if it is not
    if not(stationary):
        return -np.inf
    ## The distribution of
    # y_t|y_{t-1}, ..., y_{t-7} ~ N(c+\phi_{1}*y_{t-1}+...+\phi_{7}y_{t-7}, sigma2)
    ## Create lagged matrix
```

```
    X = lagged_matrix(y, 7)
    yf = y[7:]
    Xf = X[7:,:]
    loglik = np.sum(norm.logpdf(yf, loc=(c + Xf@phi), scale=np.sqrt(sigma2)))
    return loglik

## Unconditional Likelihood
def uncond_loglikelihood_ar7(params, y):
    ## The unconditional loglikelihood
    ## is the unconditional "plus" the density of the
    ## first p (7 in our case) observations
    cloglik = cond_loglikelihood_ar7(params, y)

    ## Calculate initial
    # y_1, ..., y_7 ~ N(mu, sigma_y)
    c = params[0]
    phi = params[1:8]
    sigma2 = params[8]
    mu, Sigma, stationary = unconditional_ar_mean_variance(c, phi, sigma2)
    if not(stationary):
        return -np.inf
    mvn = multivariate_normal(mean=mu, cov=Sigma, allow_singular=True)
    uloglik = cloglik + mvn.logpdf(y[0:7])
    return uloglik

# Using INDPRO as the target variable.
## Computing OLS
X = lagged_matrix(INDPRO, 7)
yf = INDPRO[7:]
Xf = np.hstack((np.ones((len(INDPRO)-7,1)), X[7:,:]))
beta = np.linalg.solve(Xf.T@Xf, Xf.T@yf)
sigma2_hat = np.mean((yf - Xf@beta)**2)
params= np.hstack((beta, sigma2_hat))
print("The parameters of the OLS model are", params)

# to maximize likelihood a function of the negative likelihood is defined to be minimized
params = np.array([
    0.0012, ## c
    0.0291, 0.07, 0.059, 0.04, 0.04, 0.02, 0.06, ## phi
    0.008 ## sigma2
    ])

def cobj(params, y):
    return - cond_loglikelihood_ar7(params,y)

#Same Procedure for unconditional likelihood
params= np.hstack((beta, sigma2_hat))
```

As the next step we construct the matrix containing `INDPRO` and its lagged values .

3.1 Results: Conditional Likelihood

3.1.1 Unconstrained Maximization

We estimate four different models for the conditional likelihood:

1. `Model 1`: We use the OLS coefficients as the initial guess and minimize the conditional likelihood function using the 'L-BFGS-B' method.
2. `Model 2`: We use the OLS coefficients as the initial guess and minimize the conditional likelihood function using the 'Neler-Mead' method.

3. `Model 3`: We supply a different initial guess and minimize the conditional likelihood function using the 'L-BFGS-B' method.

4. `Model 4`: We supply a different initial guess and minimize the conditional likelihood function using the 'Neler-Mead' method.

```
# Now we estimate multiple models:
## OLS model
modOLS = np.hstack((beta, sigma2_hat))
## Conditional Likelihood
## 1. Using the OLS parameters as the initial guess
### L-BFGS-B
mod1 = scipy.optimize.minimize(fun = cobj, x0 =  modOLS, args = INDPRO, method='L-BFGS-B').x
### Nelder-Mead
mod2 = scipy.optimize.minimize(fun = cobj, x0 =  modOLS, args = INDPRO, method='Nelder-Mead').x
## 2. Using a slightly different initial guess
Initial_Guess = np.array([
    0.0012, ## c
    0.0291, 0.07, 0.059, 0.04, 0.04, 0.02, 0.06, ## phi
    0.009 ## sigma2
])
### L-BFGS-B
mod3 = scipy.optimize.minimize(fun = cobj, x0 =  Initial_Guess, args = INDPRO, method='L-BFGS-B').x
### Nelder-Mead
mod4 = scipy.optimize.minimize(fun = cobj, x0 =  Initial_Guess, args = INDPRO, method='Nelder-Mead').x

# Compiling models to print to LateX table
def as_latex_table(rows, rownames, colnames, caption):
    df = np.array(rows)
    df = pd.DataFrame(df)
    df = df.T
    df.insert(0, colnames[1], rownames)
    df.columns = column_names
    tabular = df.to_latex(index=False,
                          caption=caption)
    return tabular

column_names = ("Coefficients", "OLS", "Model 1", "Model 2", "Model 3", "Model 4")
rownames = ["$c$", "$\\phi_1$", "$\\phi_2$", "$\\phi_3$", "$\\phi_4$",
            "$\\phi_5$", "$\\phi_6$", "$\\phi_7$", "$\\sigma^2$"]
caption="Results of the unconstrained maximization of the conditional likelihood"
print(as_latex_table([modOLS, mod1, mod2, mod3, mod4],
        rownames=rownames, colnames=column_names, caption=caption))
```

The results are displayed in Table 1. The results show that if the initial guess are the OLS coefficients, the conditional MLE correctly finds identical coefficients. However, when we supply a different initial guess, the minimization procedure using 'L-BFGS-B' returns simply the same initial guess while using the 'Neler-Mead' method changes the estimated coefficients quite dramatically.

3.1.2 Constrained Maximization

As discussed in the previous section, the unconstrained likelihood maximization faced several issues. The reason for this, as error messages made clear, is that using unbounded optimization, leads to negative values in σ^2 which created errors in:

`scale=np.sqrt(sigma2)`

Using the constraint $\sigma^2 \geq 0$

Table 1: Results of the unconstrained maximization of the conditional likelihood

Coefficients	OLS	Model 1	Model 2	Model 3	Model 4
c	0.001263	0.001263	0.001263	0.001200	0.000874
ϕ_1	0.291263	0.291263	0.291263	0.029100	0.056374
ϕ_2	-0.074517	-0.074517	-0.074517	0.070000	0.017666
ϕ_3	0.046809	0.046809	0.046809	0.059000	0.060468
ϕ_4	0.047565	0.047565	0.047565	0.040000	0.009790
ϕ_5	-0.024910	-0.024910	-0.024910	0.040000	0.058602
ϕ_6	0.061501	0.061501	0.061501	0.020000	0.038055
ϕ_7	0.020106	0.020106	0.020106	0.060000	0.114138
σ^2	0.000089	0.000089	0.000089	0.009000	0.000095

```
bounds_sigma = tuple((0,np.inf) for _ in range(1))
```

similarly created error messages, since σ enters into the denominaor in the PDF of the normal distribution. Thus, we decided to use the constraint: $\sigma^2 > 0$, i.e. σ^2 must be strictly larger than zero, which yielded estimated parameters close to the OLS estimate even when using very different initial guesses. We reestimated Models 1-4 using constrained maximization:

```
## We then reestimate these models unsing constrained maximization:
bounds_constant = tuple((-np.inf, np.inf) for _ in range(1))
bounds_phi = tuple((-1, 1) for _ in range(7))
bounds_sigma = tuple((0.000001,np.inf) for _ in range(1))
bounds = bounds_constant + bounds_phi + bounds_sigma

## Conditional Likelihood
## 1. Using the OLS parameters as the initial guess
### L-BFGS-B
mod1 = scipy.optimize.minimize(fun = cobj, x0 =  modOLS, args = INDPRO, method='L-BFGS-B', bounds=bounds).x
### Nelder-Mead
mod2 = scipy.optimize.minimize(fun = cobj, x0 =  modOLS, args = INDPRO, method='Nelder-Mead', bounds=bounds)
## 2. Using a slightly different initial guess
Initial_Guess = np.array([
    0.0012, ## c
    0.0291, 0.07, 0.059, 0.04, 0.04, 0.02, 0.06, ## phi
    0.009 ## sigma2
])
### L-BFGS-B
mod3 = scipy.optimize.minimize(fun = cobj, x0 =  Initial_Guess, args = INDPRO, method='L-BFGS-B', bounds=bou
### Nelder-Mead
mod4 = scipy.optimize.minimize(fun = cobj, x0 =  Initial_Guess, args = INDPRO, method='Nelder-Mead', bounds=
```

Table 2: Results of the constrained maximization of the conditional likelihood

Coefficients	OLS	Model 1	Model 2	Model 3	Model 4
c	0.001263	0.001263	0.001263	0.001283	0.000874
ϕ_1	0.291263	0.291263	0.291263	0.283561	0.056374
ϕ_2	-0.074517	-0.074517	-0.074517	-0.018719	0.017666
ϕ_3	0.046809	0.046809	0.046809	0.003397	0.060468
ϕ_4	0.047565	0.047565	0.047565	0.031652	0.009790
ϕ_5	-0.024910	-0.024910	-0.024910	-0.010921	0.058602
ϕ_6	0.061501	0.061501	0.061501	0.029010	0.038055
ϕ_7	0.020106	0.020106	0.020106	0.039583	0.114138
σ^2	0.000089	0.000089	0.000089	0.000089	0.000095

As the table shows, even when choosing an initial guess different from the OLS parameters the bounded optimization yields largely accurate estimates when using the 'L-BFGS-B' method.

3.2 Results: Unconditional Likelihood

3.2.1 Unconstrained Maximization

We estimate the following models for the unconditional likelihood function:

- `Model 5`: We use the OLS coefficients as the initial guess and minimize the unconditional likelihood function using the 'L-BFGS-B' method.
- `Model 6`: We use the OLS coefficients as the initial guess and minimize the unconditional likelihood function using the 'Neler-Mead' method.

```
## Unbounded Unconditional Likelihood
## 1. Using the OLS parameters as the initial guess
### L-BFGS-B
mod5 = scipy.optimize.minimize(fun = uobj, x0 =  modOLS, args = INDPRO, method='L-BFGS-B').x
### Nelder-Mead
mod6 = scipy.optimize.minimize(fun = uobj, x0 =  modOLS, args = INDPRO, method='Nelder-Mead').x
## 2. Using a slightly different initial guess
Initial_Guess = np.array([
    0.0012, ## c
    0.00291, 0.007, 0.0509, 0.0024, 0.0409, 0.012, 0.0601, ## phi
    0.009 ## sigma2
])
### L-BFGS-B
#mod7 = scipy.optimize.minimize(fun = uobj, x0 =  Initial_Guess, args = INDPRO, method='L-BFGS-B').x
### Nelder-Mead
#mod8 = scipy.optimize.minimize(fun = uobj, x0 =  Initial_Guess, args = INDPRO, method='Nelder-Mead').x
```

The results are displayed in Table 2:

```
umods = np.array([modOLS, mod5, mod6])
umodsDF = pd.DataFrame(umods)
umodsDF = umodsDF.T
rownames = ["$c$", "$\phi_1$", "$\phi_2$", "$\phi_3$", "$\phi_4$",
            "$\phi_5$", "$\phi_6$", "$\phi_7$", "$\sigma^2$"]
umodsDF.insert(0, "Coefficients", rownames)
column_names = ("Coefficients", "OLS", "Model 5", "Model 6")
umodsDF.columns = column_names
print(umodsDF.to_latex())
```

Table 3: Results of the unconstrained maximization of the unconditional likelihood

Coefficients	OLS	Model 5	Model 6
c	0.001263	0.001263	0.001277
ϕ_1	0.291263	0.291263	0.309304
ϕ_2	-0.074517	-0.074517	-0.076766
ϕ_3	0.046809	0.046809	0.040008
ϕ_4	0.047565	0.047565	0.038904
ϕ_5	-0.024910	-0.024910	-0.031723
ϕ_6	0.061501	0.061501	0.056227
ϕ_7	0.020106	0.020106	0.020665
σ^2	0.000089	0.000091	0.000091

3.2.2 Constrained Maximization

We introduce two further models analagous to the earlier section to see whether using a different initial guess yields reliable results.

- `Model 7`: Bounded optimization using 'L-BFGS-B'
- `Model 8`: Bounded optimization using 'Nelder-Mead'

Table 4: Results of the constrained maximization of the unconditional likelihood

Coefficients	OLS	Model 5	Model 6	Model 7	Model 8
c	0.001263	0.001263	0.001277	0.001142	0.001462
ϕ_1	0.291263	0.291263	0.309304	0.287896	0.009534
ϕ_2	-0.074517	-0.074517	-0.076766	0.007889	-0.014135
ϕ_3	0.046809	0.046809	0.040008	0.030199	0.079556
ϕ_4	0.047565	0.047565	0.038904	0.024074	-0.005964
ϕ_5	-0.024910	-0.024910	-0.031723	0.001736	0.050275
ϕ_6	0.061501	0.061501	0.056227	0.025235	0.043152
ϕ_7	0.020106	0.020106	0.020665	0.049086	0.034731
σ^2	0.000089	0.000091	0.000091	0.000092	0.000100

4 Forecast

We use the estimated parameters to generate forecasts for `INDPRO` eight periods ahead. For this purpose we define a function, which takes the forecast horizon and the estimated model parameters as inputs, using the aforementioned eight periods and the OLS coefficients as defaults.

```
### Forecasting:
def forecastAR(h=8, model = mods[0]): # defaults: h is the forecasting horizon, mods[0] the OLS model
    forecastArray = np.empty(h) # Empty array to store forecasts
    lastrow = np.array(INDPRO.tail(7)) #use last 7 rows of INDPRO for the first forecast
    lastrow = np.flip(lastrow) # invert array to align with order of estimated parameters in the model
    for i in range(1, h+1):
        forecast = model[0] + model[1:8] @ lastrow # mods[0] uses the OLS estimates
        lastrow = np.insert(lastrow, 0, forecast)
        lastrow = np.delete(lastrow, -1)
        forecastArray[i-1] = forecast

    return forecastArray
```

4.1 Forecast: Conditional Likelihood

Again we display the results in a Table.

```
rownames = ("$y_{t+1}$", "$y_{t+2}$", "$y_{t+3}$", "$y_{t+4}$",
            "$y_{t+5}$","$y_{t+6}$", "$y_{t+7}$", "$y_{t+8}$")
results = {'Forecasts': rownames,'OLS': forecastAR(), 'Model 1': forecastAR(model = mods[1]),
           'Model 2': forecastAR(model = mods[2]),'Model 3': forecastAR(model = mods[3]),
           'Model 4': forecastAR(model = mods[4])}
results = pd.DataFrame(results)

caption="Forecasts using the estimated coefficients from OLS and the bounded conditional likelihood maximizat
print(results.to_latex(index=False, caption=caption))
```

4.2 Forecast: Unconditional Likelihood

Table 5: Forecasts using the estimated coefficients from OLS and the bounded conditional likelihood maximization.

Forecasts	OLS	Model 1	Model 2	Model 3	Model 4
y_{t+1}	0.000840	0.000840	0.000840	0.001076	0.001931
y_{t+2}	0.001962	0.001962	0.001962	0.001785	0.000468
y_{t+3}	0.001172	0.001172	0.001172	0.001534	0.000843
y_{t+4}	0.001467	0.001467	0.001467	0.001420	0.000218
y_{t+5}	0.001817	0.001817	0.001817	0.001823	0.001225
y_{t+6}	0.001751	0.001751	0.001751	0.001796	0.001080
y_{t+7}	0.001745	0.001745	0.001745	0.001785	0.000970
y_{t+8}	0.001904	0.001904	0.001904	0.001884	0.001311

```
rownames = ("$y_{t+1}$", "$y_{t+2}$", "$y_{t+3}$", "$y_{t+4}$",
            "$y_{t+5}$","$y_{t+6}$", "$y_{t+7}$", "$y_{t+8}$")
results = {'Forecasts': rownames,'OLS': forecastAR(), 'Model 5': forecastAR(model = mods[5]),
           'Model 6': forecastAR(model = mods[6]),'Model 7': forecastAR(model = mods[7]),
           'Model 8': forecastAR(model = mods[8])}
results = pd.DataFrame(results)

caption="Forecasts using the estimated coefficients from OLS and the bounded unconditional likelihood maximiz
print(results.to_latex(index=False, caption=caption))
```

Table 6: Forecasts using the estimated coefficients from OLS and the bounded unconditional likelihood maximization.

Forecasts	OLS	Model 5	Model 6	Model 7	Model 8
y_{t+1}	0.000840	0.000840	0.000890	0.001169	0.002049
y_{t+2}	0.001962	0.001962	0.002034	0.001515	0.001083
y_{t+3}	0.001172	0.001172	0.001268	0.001404	0.001195
y_{t+4}	0.001467	0.001467	0.001507	0.001245	0.001470
y_{t+5}	0.001817	0.001817	0.001852	0.001727	0.001587
y_{t+6}	0.001751	0.001751	0.001782	0.001707	0.001607
y_{t+7}	0.001745	0.001745	0.001761	0.001704	0.001675
y_{t+8}	0.001904	0.001904	0.001910	0.001827	0.001751

4.3 Comparing Forecasts

Since the models which use OLS as the initial guess yield forecasts equal to the OLS forecasts, we focus here on the models using bounded optimization and the 'L-BFGS-B' method. We compute their respective deviation from the OLS forecasts and the sum of the squared deviations. As the table shows, the deviations are quite small, but smaller for the conditional likelihood forecast.

4.4 Evaluating Forecasts

Here we choose a grahpical display. The idea is to choose a certain date, for example 01/01/2000 and train a model on all observations up to this date. Then, we estimate the next h periods and compare them to the actual values. This code produces such a plot displaying the actual values, the OLS forecasts and forecasts generated by models 3 and 7, that is bounded maximum likelihood estimation using 'L-BFGS-B'.

```
# Task 4: Comparing Forecasts by plotting them
# plotting forecasts:
# Using the data up to 2000 as training data and the rest of the data as testing data
# Creating a new dataframe containgin all the data up to a certain date, say for example 01/01/2020 and forec
# actual to forecasted data.
df = df.drop(index=0)
```

Table 7: Forecasts using OLS, Model 3 (Bounded Conditional Likelihood, Different Initial Guess), Model 7(Bounded Unconditional Likelihood, Different Initial Guess)

Forecasts	OLS	Model 3	Model 7	OLS - Model 3	OLS - Model 7
y_{t+1}	0.000839982	0.001076327	0.001168732	-0.000236345	-0.000328750
y_{t+2}	0.001962069	0.001784693	0.001514858	0.000177376	0.000447211
y_{t+3}	0.001172201	0.001534039	0.001404130	-0.000361838	-0.000231929
y_{t+4}	0.001467235	0.001419880	0.001244736	0.000047355	0.000222499
y_{t+5}	0.001817004	0.001822645	0.001727321	-0.000005641	0.000089683
y_{t+6}	0.001751388	0.001795515	0.001706722	-0.000044128	0.000044666
y_{t+7}	0.001745432	0.001785201	0.001704291	-0.000039769	0.000041141
y_{t+8}	0.001903688	0.001884076	0.001826758	0.000019612	0.000076929
$\sum \Delta^2$				0.000000224	0.000000429

```
df['sasdate'] = pd.to_datetime(df['sasdate'])
date = "01/01/2000"
lasttrain = df.index[df['sasdate']== date].to_list()
train = INDPRO[:lasttrain[0]]
test = INDPRO[lasttrain[0]-1:]

#First, OLS Model:
X = lagged_matrix(train, 7)
yf = train[7:]
Xf = np.hstack((np.ones((len(train)-7,1)), X[7:,:]))
beta = np.linalg.solve(Xf.T@Xf, Xf.T@yf)
sigma2_hat = np.mean((yf - Xf@beta)**2)
params= np.hstack((beta, sigma2_hat))
modOLS = params

#Second, conditional likelihood:
modCon = scipy.optimize.minimize(fun = cobj, x0 =  Initial_Guess, args = train, method='L-BFGS-B', bounds=bou

#Third, unconditional likelihood:
modUCon = scipy.optimize.minimize(fun = uobj, x0 =  Initial_Guess, args = train, method='L-BFGS-B', bounds=bo

mods = np.array([modOLS, modCon, modUCon])
def forecastAR(h=8, model = mods[0], data=train): # defaults: h is the forecasting horizon, mods[0] the OLS r
    forecastArray = np.empty(h) # Empty array to store forecasts
    lastrow = np.array(data.tail(7)) #use last 7 rows of test data for the first forecast
    lastrow = np.flip(lastrow) # invert array to align with order of estimated parameters in the model
    for i in range(1, h+1):
        forecast = model[0] + model[1:8] @ lastrow # mods[0] uses the OLS estimates
        lastrow = np.insert(lastrow, 0, forecast)
        lastrow = np.delete(lastrow, -1)
        forecastArray[i-1] = forecast

    return forecastArray


h = 8 # forecast for h periods
forecast={'INDPRO': test[:h],
          'Forecast OLS': forecastAR(h=h),
          'Forecast Conditional': forecastAR(h=h, model = mods[1]),
          'Forecast Unconditional': forecastAR(h=h, model = mods[2])}

forecast = pd.DataFrame(forecast)
```

```
plotdata = {'INDPRO': train,
            'Forecast OLS': train,
            'Forecast Conditional': train,
            'Forecast Unconditional': train}

plotdata = pd.DataFrame(plotdata)
plotdata =  [plotdata, forecast]
plotdata = pd.concat(plotdata)
plotdata.insert(0, "sasdate",df.iloc[1:len(plotdata)+1,0])
plotdata = plotdata.tail(2*h) # seletct the h forecasted observations and the h previous observations for plo
print(plotdata)

plt.plot(plotdata['sasdate'], plotdata['Forecast OLS'], color = "blue")
plt.plot(plotdata['sasdate'], plotdata['Forecast Conditional'], color = "green")
plt.plot(plotdata['sasdate'], plotdata['Forecast Unconditional'], color = "red")
plt.plot(plotdata['sasdate'], plotdata['INDPRO'],color="black")
plt.legend(['Forecast OLS', 'Forecast Conditional', 'Forecast Unconditional', 'INDPRO'])
plt.show()
```