Report: Assignment 2 – Le Julie

Theory premise: Conditional Likelihood and Unconditional Likelihood

In general, the likelihood is the probability of observing the data we have observed. We have *f* which is the joint density function. If we knew the joint density of data, we would calculate the value of the density at the value of the data we have in the dataset. But there is a problem, we don't know what the density is. But we can overcome this issue by partitioning the joint density as the product of the conditional. Thus, the likelihood can be written as:

$$\prod_{t=2}^{T} f\left(y_{t-1},..\right) * f(y1)$$

Note: there is no y_0 to take the conditional.

We consider an autoregressive model of order 7, AR(7), which is:

$$x_t=c+\phi_1 xt_{-1}+\phi_2 x_{t-2}+\phi_3 x_{t-3}+\ldots+\phi_7 x_{t-7}+u_t$$

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

Thus, the *conditional likelihood* will be:

$$L(x;\phi) = \prod_{t=8}^{T} f(xt|x_{t-1}, ...x_{t-7}; \phi)$$

In order to find the ϕ vectors we don't use *L(X;ϕ)* but the log-likelihood function.

$$l(x;\phi)= \log log\, f\left(x_1, ..., x_7; \phi\right) + \sum_{t=8}^{T} \log log\, (f(x_t|x_{t-1}, ..., x_{t-7}; \phi))$$

We notice that the lag value of x are deterministic quantities not random variables. Therefore, we know that x_t is distributed as a normal, since the error terms (u_t) are normally distributed.

$$x_t|x_{t-1},.., xt_{-7} \sim N(c+\phi_1 x_{t-1}+\ldots+\phi 7 x_{t-7},\sigma^2)$$

The *unconditional likelihood* function refers to the probability of observing a complete sequence of data without conditions on past observations. In this case, previous observations are not considered as conditions, thus we assume that the time series observations are jointly distributed as a multivariate normal distribution and it depends only on the model parameters, ϕ vector and σ^2 without considering specific observations.

Coding the likelihood functions

The function `unconditional_ar_mean_variance` takes 3 parameters: `c` as a constant term, `phis` as coefficients of the autoregressive terms and `sigma2` as variance. The `phis` array has length `p` and through this we can construct the autoregressive matrix `A` with dimension `p` by `p`.

The eigenvalues of matrix `A` are calculated using `np.linalg.eig(A)`. If all the absolute values of the eigenvalues are less than 1, it indicates that the autoregressive process is stationary. Stationarity is a property that indicates that the statistical characteristics of the process remain constant over time (variance and mean are not dependent on time).

In the context of autoregressive models, the mean vector μ represents the expected value of the process observations, given the observed past. It indicates what the average value of the process should be in the next time period, considering the historical information available. μ is computed through matrix algebra:

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

Where I is the identity matrix, b is a vector representing the constant c of the autoregressive model. Using the inverse matrix (I-A) allows us to invert the effect of A on b.

The discrete Lyapunov equation allows us to compute the covariance matrix. This matrix gives us information on casual variables' variability over time and the relationships between them (covariance).

```
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
```

The `lagged_matrix` function creates the `max_lag` by `max_lag` matrix with appropriately lagged data necessary for the autoregression.

```
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
```

$$X_t = c + \phi_1 X_{t-1} + \phi_2 X_{t-2} + \ldots + \phi_7 X_{t-7} + \epsilon_t$$

The conditional likelihood function takes 2 parameters:

1. `params` is an array containing the parameters of the AR(7) process, described by the equation above: the constant c, the parameters φ_1, ..., φ_7, and the variance of the error ε_t, σ^2
2. y is the observed time series data

We then compute the unconditional mean and covariance of the AR(7) process through the `unconditional_ar_mean_variance`, assessing also if the process is stationary. If it so, the function proceeds to calculate the conditional log-likelihood of observing data `yf` given the parameters of the model. This log-likelihood is calculated assuming that the data follows a normal distribution with mean `c+Xf@phi` and standard deviation `sqrt(sigma2)`.

We follow the same process for the unconditional likelihood function.

```
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
```

```
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
```

Maximizing the likelihood and Parameter estimation

We import the data from the current.csv dataset, and use the `apply_transformation` function to apply the appropriate transformation to each column, in order to make each series stationary, transforming INDPRO into a series of its log differences

```
df = pd.read_csv("/Users/lavin/Downloads/current (1).csv")
df_cleaned = df.drop(index=0)
df_cleaned.reset_index(drop=True, inplace=True)
df_cleaned['sasdate'] = pd.to_datetime(df_cleaned['sasdate'], format='%m/%d/%Y')

transformation_codes = df.iloc[0, 1:].to_frame().reset_index()
transformation_codes.columns = ['Series', 'Transformation_Code']

def apply_transformation(series, code):
    if code == 1:
        # No transformation
        return series
    elif code == 2:
        # First difference
        return series.diff()
    elif code == 3:
        # Second difference
        return series.diff().diff()
    elif code == 4:
        # Log
        return np.log(series)
    elif code == 5:
        # First difference of log
        return np.log(series).diff()
    elif code == 6:
        # Second difference of log
```

```
        return np.log(series).diff().diff()
    elif code == 7:
        # Delta (x_t/x_{t-1} - 1)
        return series.pct_change()
    else:
        raise ValueError("Invalid transformation code")

for series_name, code in transformation_codes.values:
    df_cleaned[series_name] =
apply_transformation(df_cleaned[series_name].astype(float), float(code))

df_cleaned = df_cleaned[2:]
df_cleaned.reset_index(drop=True, inplace=True)
df_cleaned.head()
```

We will use this data to compute the OLS parameters of the INDPRO series, and the variance of the error σ^2:

```
X = lagged_matrix(df_cleaned['INDPRO'], 7)
yf = df_cleaned['INDPRO'][7:]
Xf = np.hstack((np.ones((772,1)), 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))

column_titles = [f'Phi_{i}' for i in range(8)] + ['sigma2']
params_df = pd.DataFrame([params], columns = column_titles)
print(f'the OLS parameters are:\n{params_df}')
```

The OLS parameters are:

φ_0[1]	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	σ^2
0.001257	0.293199	-0.073489	0.046731	0.047076	-0.025917	0.061311	0.019114	0.000089

Now, in order to compute the parameters $\varphi_0, \ldots, \varphi_7, \sigma^2$ through conditional likelihood maximization and unconditional likelihood maximization, we take the following steps:

1. We define the two functions `cobj` and `uobj` which return the negatives of the conditional and unconditional likelihoods, respectively. The likelihoods are computed through the previously defined `cond_loglikelihood_ar7` and `uncond_loglikelihood_ar7` functions.
2. We then use the `scipi.optimize.minimize` routine to minimize the negative of each likelihood, therefore effectively maximizing the likelihoods. We use the OLS parameters which we computed previously as starting values for these optimizations.
3. We stored the results of these minimizations in the objects results_cond and results_uncond:

[1] φ_0 is the constant, which was previously referred to as c in this report.

this way, through `results_cond.x` and `results_uncond.x`, we can access the $\varphi_0, \ldots, \varphi_7, \sigma^2$ parameters computed through conditional and unconditional likelihood maximization respectively.

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

def uobj(params, y):
    return - uncond_loglikelihood_ar7(params,y)

results_cond = scipy.optimize.minimize(cobj, params, args =
df_cleaned['INDPRO'], method='L-BFGS-B')

bounds_constant = tuple((-np.inf, np.inf) for _ in range(1))
bounds_phi = tuple((-1, 1) for _ in range(7))
bounds_sigma = tuple((0,np.inf) for _ in range(1))
bounds = bounds_constant + bounds_phi + bounds_sigma

## L-BFGS-B support bounds
results_uncond = scipy.optimize.minimize(uobj, results_cond.x, args =
df_cleaned['INDPRO'], method='L-BFGS-B', bounds = bounds)

newparams_df = params_df = pd.DataFrame([params, results_cond.x,
results_uncond.x], columns = column_titles, index=['OLS', 'cond. MLE', 'uncond.
MLE'])
print(f'The parameters computed with each approach are:\n{newparams_df}')
```

Method used:	φ_0	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	σ^2
OLS	0.001257	0.293199	-0.073489	0.046731	0.047076	-0.025917	0.061311	0.019114	0.000089
cond. MLE	0.001257	0.293199	-0.073489	0.046731	0.047076	-0.025917	0.061311	0.019114	0.000089
uncond. MLE	0.001236	0.293200	-0.073490	0.046729	0.047075	-0.025918	0.061311	0.019114	0.000091

We observe that the parameters computed through conditional likelihood maximization are equal to those computed through ordinary least squares estimation.

Forecasting

Now, in order to forecast the future values of the log differences of INDPRO for the next 8 months, we define the `ar_forecast` function, which takes the following arguments:

- `data`: the time series which is the object of the forecast,
- `phi`: the $\varphi_0, \ldots, \varphi_p$ parameters of the autoregression of order p,
- `p`: the order of the autoregression,
- `h`: the number of periods ahead which will be forecasted,

and returns the `h` values for the `h` steps ahead forecasted through the parameters `phi` applied to an autoregression of order `p`:

```
def ar_forecast (data, phi, p, h):
    forecasted_values = np.zeros(h)
    for i in range(h):
        # Construct the lagged values for the autoregressive model
        lagged_values = np.flip(data[-p:])
        # Calculate the forecasted value using the autoregressive model
        forecasted_values[i] = np.dot(phi, np.concatenate(([1], lagged_values)))
        # Append the forecasted value to the time series for the next forecast
iteration
        data = np.append(data, forecasted_values[i])

    return forecasted_values
```

Applying this function to the INDPRO series, with p = 7 and h = 8, we obtain the values forecasted for the next 8 months with the parameters estimated with each approach:

```
# Forecast future values

forecasted_values_cond = ar_forecast(df_cleaned['INDPRO'], phi =
results_cond.x[0:8], p = 7, h = 8)
forecasted_values_uncond = ar_forecast(df_cleaned['INDPRO'], phi =
results_uncond.x[0:8], p = 7, h = 8)
print(f'The values forecasted through conditional likelihood maximization are:
{forecasted_values_cond}\nThe values forecasted through unconditional likelihood
maximization are: {forecasted_values_uncond}')
```

This returns:

```
The values forecasted through conditional likelihood maximization are:
[0.00082654 0.00196014 0.00116907 0.00147207 0.0018135  0.00174744
 0.00174124 0.00189847]
The values forecasted through unconditional likelihood maximization are:
[0.0008056  0.00193306 0.00114173 0.00144412 0.00178412 0.00171793
 0.00171063 0.00186669]
```

We can evaluate the accuracy of these forecasts by using the `ar_forecast` function to conduct an out of sample evaluation: we first compute the parameters based on INDPRO data up until 2009-04-01, and then define the `evalforecast` function, which we will use to forecasts the values of INDPRO from this date to the present. `evalforecast` returns the actual value of INDPRO for each date and the relative forecast error, as well as the predicted value. The `phis` argument of the `evalforecast` function is to be set to the parameters φ_0 , ..., φ which we wish to use to compute the forecast: by setting phis equal to `results_cond.x[0:8]` and `results_uncond.x[0:8]` we will obtain the forecasts computed through each set of parameters.

```
# Comparing the accuracy of the forecasts

series = df_cleaned['INDPRO'][:600]
results_cond = scipy.optimize.minimize(cobj, params, args = series,
method='L-BFGS-B')
results_uncond = scipy.optimize.minimize(uobj, results_cond.x, args = series,
method='L-BFGS-B', bounds = bounds)

def evalforecast (phis, i, n):
    e = []
    Y_hat = []
    Y_actual = []
    for j in range(0, i):
        yhat = ar_forecast(data = df_cleaned['INDPRO'][:n+j], phi = phis, p = 7,
h = 1)
        Y_hat.append(yhat.flatten())
        yactual=df_cleaned['INDPRO'][n+1+j]
        Y_actual.append(yactual.flatten())
        ehat = yactual - yhat
        e.append(ehat.flatten())
    return (Y_actual, Y_hat, e)

condforecast = evalforecast(phis = results_cond.x[0:8], i = 178, n = 600)
uncondforecast = evalforecast(phis = results_uncond.x[0:8],i = 178, n = 600)
```

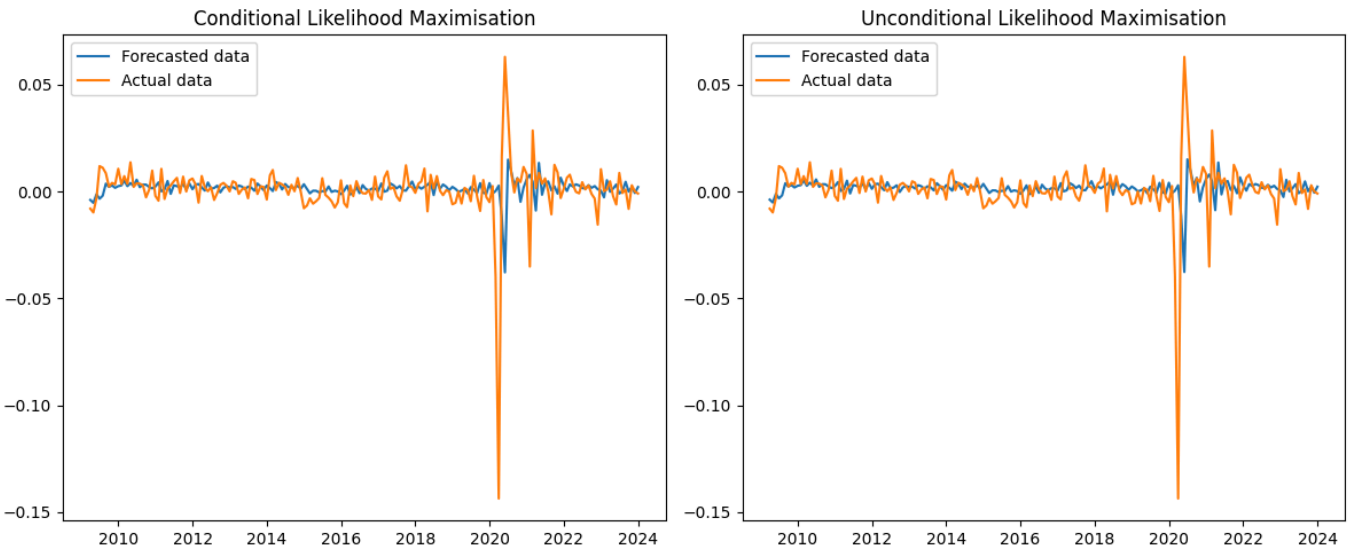
We can now plot the results:

```
# Plotting actual data vs data forecasted with conditional and unconditional MLE

import matplotlib.pyplot as plt
import matplotlib.dates as mdates

dates= pd.to_datetime(df_cleaned['sasdate'], format='%m/%d/%Y')
titles = ['Conditional Likelihood Maximisation', 'Unconditional Likelihood
Maximisation']
forecasts = [condforecast, uncondforecast]

fig, axs = plt.subplots(1, 2, figsize=(12, 5))
for i, ax in enumerate(axs):
    ax.plot(dates[601:], forecasts[i][1], label='Forecasted data')
    ax.plot(dates[601:], forecasts[i][0], label='Actual data')
    ax.set_title(titles[i])
    ax.legend()
plt.tight_layout()
plt.show()
```



We can also compute the MSFEs:

```
e_cond, e_uncond = pd.DataFrame(condforecast[2]),
pd.DataFrame(uncondforecast[2])
msfe_cond, msfe_uncond = (e_cond.apply(np.square).mean()),
(e_uncond.apply(np.square).mean())
msfe_cond, msfe_uncond = float(msfe_cond.iloc[0]), float(msfe_uncond.iloc[0])

print(f'The MSFE of the forecast obtained through conditional MLE is:
{msfe_cond:.10f}\nThe MSFE of the forecast obtained through unconditional MLE
is: {msfe_uncond:.10f}')
```

This returns:

```
The MSFE of the forecast obtained through conditional MLE is: 0.0002364001
The MSFE of the forecast obtained through unconditional MLE is: 0.0002366056
```

Observing the plotted data and the mean squared forecast errors, we can conclude that the conditional model is not significantly better than the unconditional model for forecasting the data. This indicates that the AR(7) process is a good fit for describing the patterns of the data in the INDPRO series.