Statistics for Data Science

Suto Robert-Lucian (411)

1 Exercises

1.1 Exercise 1

The time series was taken from kaggle (https://www.kaggle.com/datasets/kandij/electric-production).
I began by plotting my time series next to the ones presented in the Laboratory to see if they are similar(the chosen one on the right).

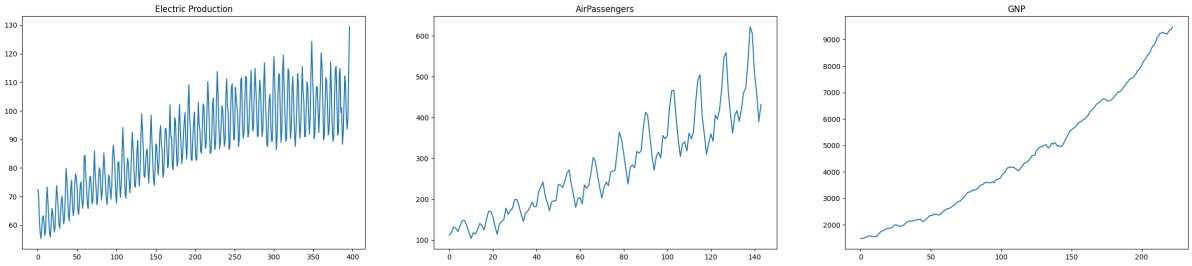

Figure 1: Data plots

We can see that besides the fact that they are increasing, they are not really similar. After this, I've renamed the columns, dropped null values, and set the date as an ID to make the work easier. To better view the data, I've plotted it again with markers, next to it's distribution.

Our time series does not meet the criteria for stationarity. It should have a stable mean, variance, and covariance, but this is not the case, because the mean keeps increasing over time. The fact that the series has a near-normal distribution (bell-shaped curve) is a positive sign. Another way to see this is by plotting the trend and the autocorrelation of the series.

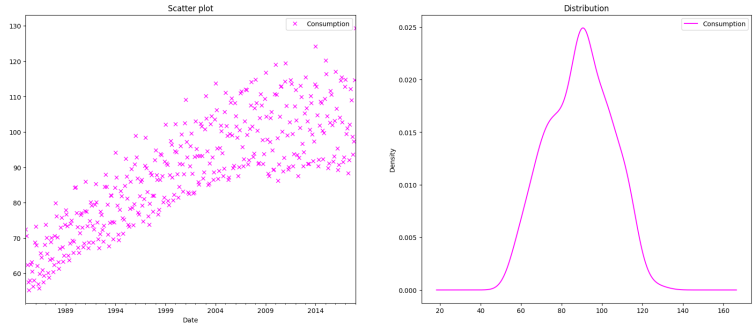

Figure 2: Data plot and it's distribution

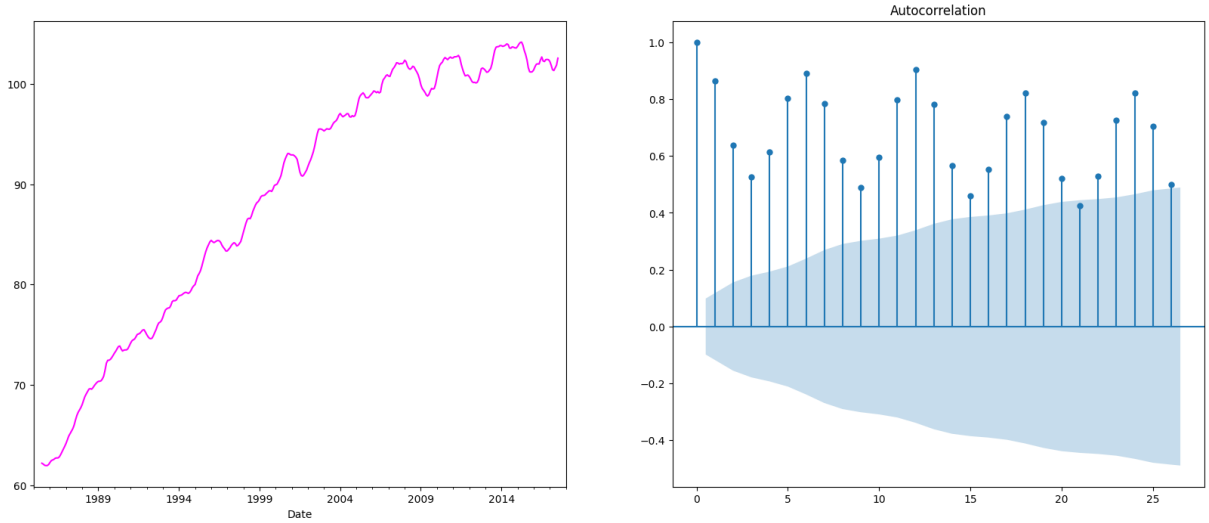

Figure 3: Trend plot and ACF plot

By looking at this two plots, and the seasonality of the data, we can get more insight into our data. The plots show an increasing trend and a seasonal pattern of electricity consumption reaching its peak every year. To test the stationarity of the time series, I used the Augmented Dickey-Fuller (ADF) test, which gives a set of values that indicate how stationary the series is. The most important values are

- **ADF statistic** is a negative number and is used to determine the stationarity of the time series. More negative values indicate stronger evidence against the null hypothesis of non-stationarity
- **p-value** is the chance of getting a test result like the one we got if the time

series is not stationary. A smaller p-value means more evidence that the time series is stationary.

- **critical values** are used to compare with the ADF statistic to determine the stationarity of the series. The dictionary keys correspond to different significance levels (e.g., 1%, 5%, 10%), and the values are the corresponding critical values.

We can conclude that the series is not stationary if we do not reject the null hypothesis.

The series becomes stationary if the mean and standard deviation are constant (flat lines). To check these, I created a function that plots the series and performs the ADF test on it.

```
adf   -2.25699035004725
p-value   0.18621469116586592
usedlag   15
nobs   381
critical values   {'1%': -3.4476305904172904, '5%': -2.869155980820355, '10%': -2.570827146203181}
```

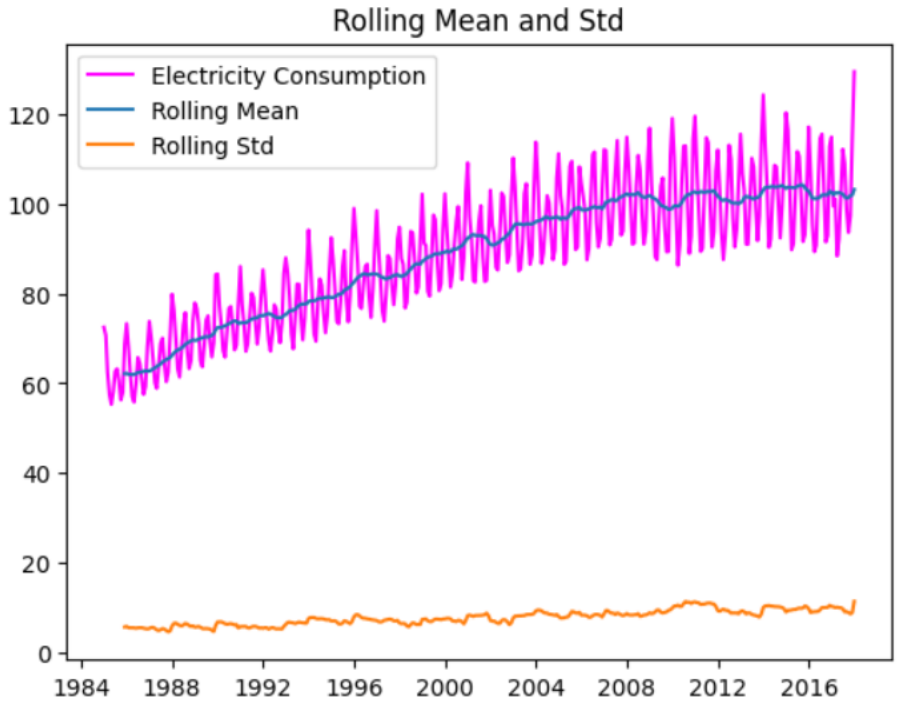

Figure 4: ADF test and mean + mean & standard deviation plot

The p-value is above 0.05, so we keep the null hypothesis. The test statistic is also much higher than the critical values, so the series can still be considered to be non-stationary. I applied the log transformation to reduce the size and trend of the series.

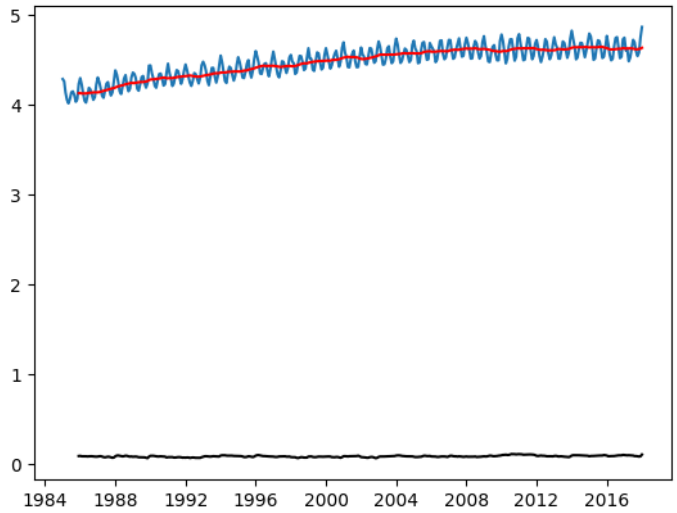

Figure 5: mean + standard deviation plot of log(series)

However, the series is still not quite stationary, so I used differencing to remove the time dependence of the series (which affects the trend and seasonality).

Differencing is simply subtracting the previous value from the current value.

After applying this, we can now observe in Figure 6 that the p-value is very low (approx 0.000000002) and that the ADF statistic is much lower than the critical values (almost two times lower). In addition to this, the rolling mean and standard deviation seem to be flat as well, so I think we can safely assume that now, the time series is stationary.

```
adf   -6.748333370019161
p-value   2.995161498115556e-09
usedlag   14
nobs   381
critical values   {'1%': -3.4476305904172904, '5%': -2.869155980820355, '10%': -2.570827146203181}
```

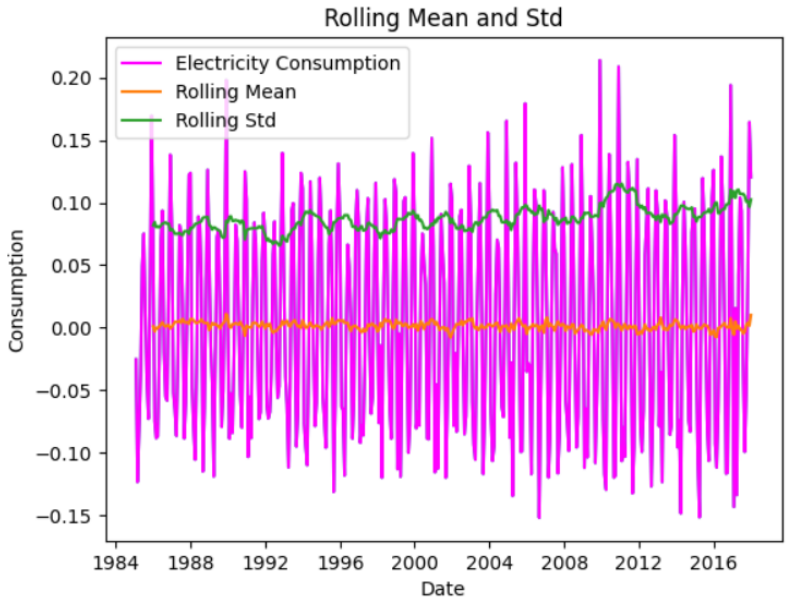

Figure 6: ADF test and + mean & standard deviation plot

Another way to check the stationarity of the series is to plot the rolling mean and standard deviation of the residual values (Figure 7). This should confirm our result. Now that we have a stationary series, we can proceed to find the best parameters for our model. I have chosen the ARIMA model presented in the laboratory, which is called by **ARIMA(p,d,q)**, where:

- p is how many previous values affect the current value,
- d is how many times we need to difference the series to make it stationary
- q is how many past errors influence the current prediction.

The values of p and q can be found by looking at the ACF and PACF plots (Figure 8). Because the graphs intersect with the origin at about the value of 3, we can now use this value and others close to it, as the p & q parameters.

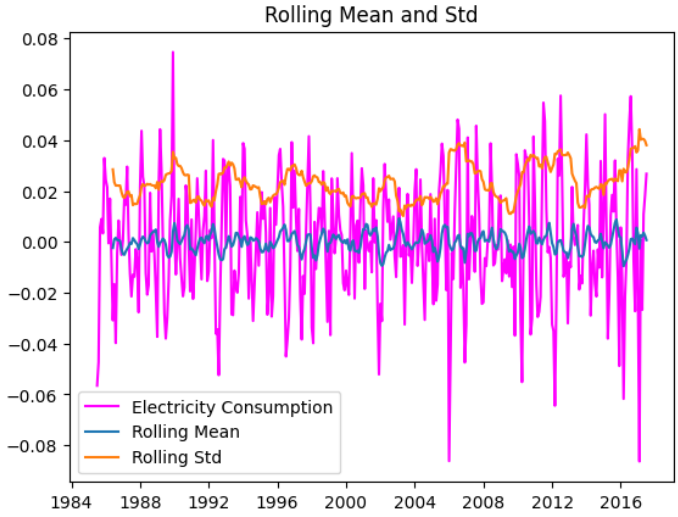

Figure 7: mean & standard deviation plot of residuals

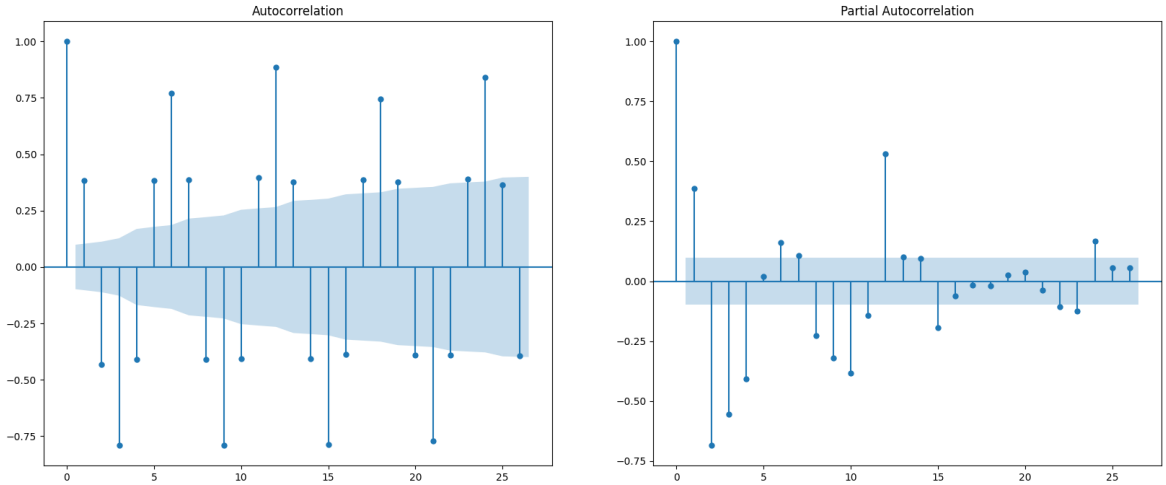

Figure 8: ACF & PACF plots

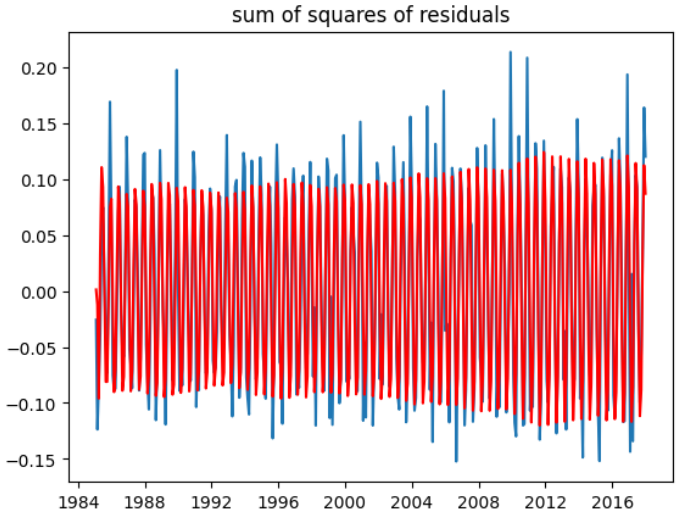

Figure 9: Actual series and fitted values

After applying the model, We can see that the fitted values(red) are well contained between the initial values(blue) and do not seem out of place. Also, the residual sum of squares (RSS), is small, sitting at **0.5** . In the end, we can plot the predicted data, next to the origianl.

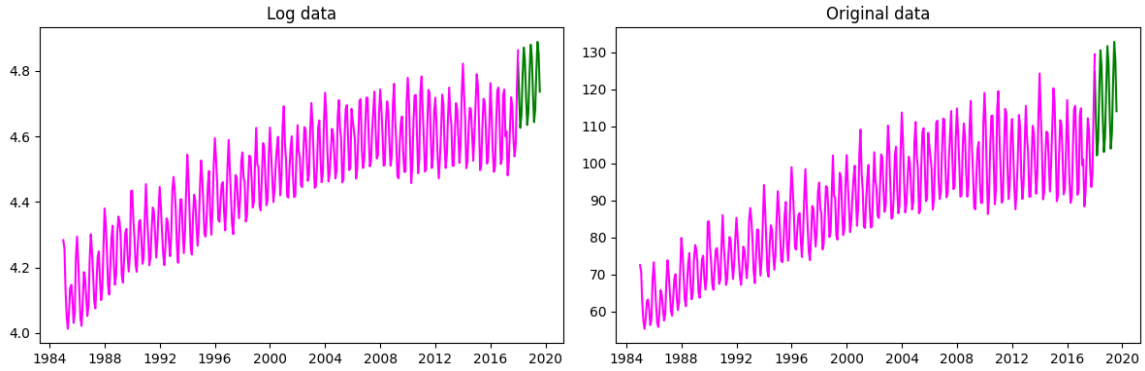

Figure 10: Predicted Data next to initial data

Again, the values don't seem to be out of place, and respect the seasonality and trend of the last values.

1.2 Exercise 2

The Durbin-Levinson algorithm helps us compute the coefficients of an autoregressive model of order p for a given time series.With the input of a time series and an integer p representing the order of the autoregressive model, the function should return a lower-triangular matrix containing which contain the coefficients of the autoregressive model. As mentioned in the requirement, the main diagonal of the matrix, should have similar values to those returned by the PACF function.

I started by generating my variables as suggested in the requirements. AR(2) with the parameters $\phi = (1, -.9))$ with 200 observations. I've saved them in a text file so I can work with them both in Python and R. The plots from both languages are as follows:

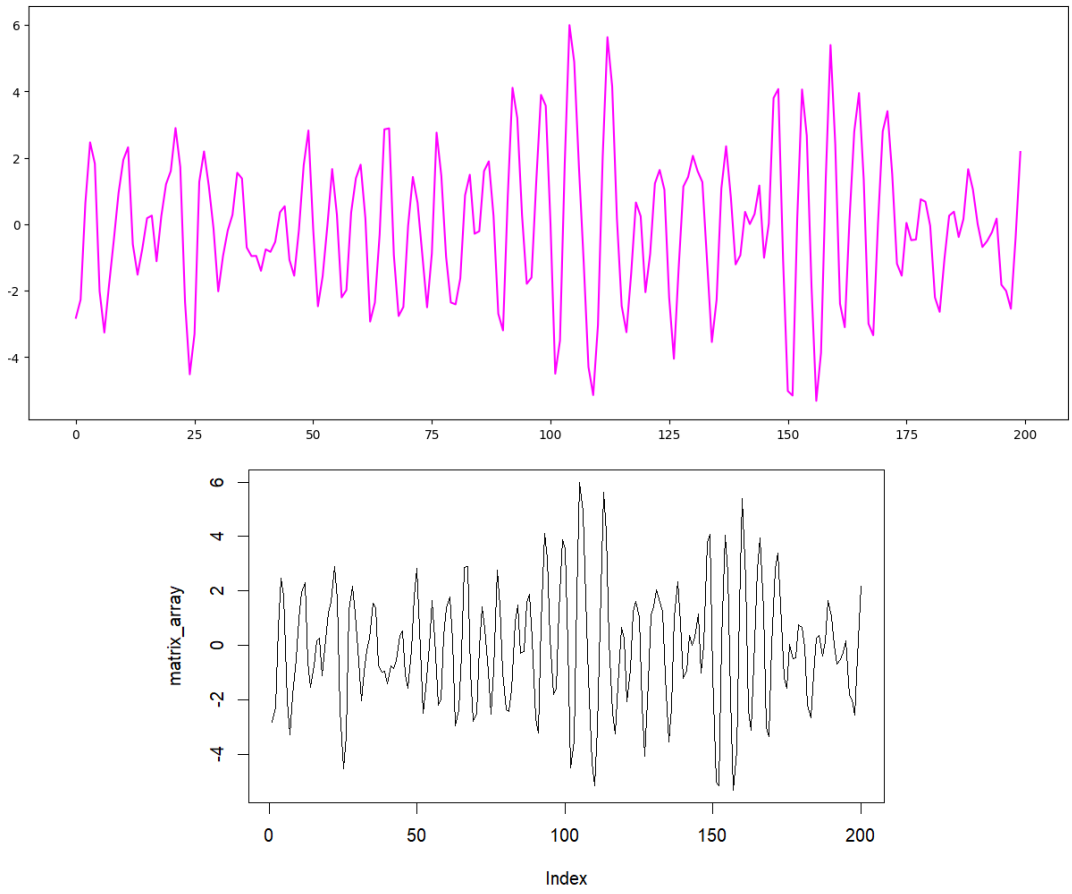

Figure 11: Plots of the AR(2) in Python and R

Previously, I've mentioned that I used both Python and R. This experiment was done purely because of the fact that I observed different results between the pacf function in R and the pacf function from the statsmodels.tsa.stattools python package. The implementation of the algorithm was done in Python, and only the pacf function was used from R. While comparing the results from my D-L algorithm implementation, the pacf values from R are almost the same, while the pacf values from python seem to be off by a couple of digits.

```
durbin_levinson(matrix_array, p=2)
```

```
array([[ 0.51660161,  0.        ],
       [ 0.9479924 , -0.83505505]])
```

```
from statsmodels.tsa.stattools import pacf
pacf(matrix_array,nlags=2)
```

```
array([ 1.        ,  0.5191976 , -0.84658575])
```

```
> pacf(matrix_array, lag.max = 2, plot = FALSE)$acf
, , 1

           [,1]
[1,]  0.5166016
[2,] -0.8350551
```

Figure 12: Durbin-Levinson output and the pacf values from both python & R

2 Code

2.1 Exercise 1

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
from statsmodels.tsa.stattools import acf,pacf
import numpy as np
import seaborn as sns
import astsadata
from statsmodels.tsa.stattools import adfuller

# import platform
```

```
# print(platform.python_version())

airpassengers = pd.read_csv('AirPassengers.csv')
series = pd.read_csv('Electric_Production.csv'#, index_col="DATE",
    parse_dates=True
                     )

fig, axs = plt.subplots(2, 2, figsize=(30, 6))
plt.subplot(1,3, 1)
plt.title("Electric Production")
plt.plot(series["Value"])
plt.subplot(1, 3, 2)
plt.title("AirPassengers")
plt.plot(airpassengers["#Passengers"])
plt.subplot(1,3, 3)
gnp_data = astsadata.gnp
plt.title("GNP")
plt.plot(gnp_data["value"].to_numpy())

series['DATE'] = pd.to_datetime(series['DATE'])
series=series.dropna()
series.head()

series.columns=['Date', 'Consumption']
series.set_index('Date', inplace=True)
series.head()

fig, (ax1, ax2) = plt.subplots(1,2, figsize=(20,8))
series.plot(style='x',ax=ax1,title="Scatter plot", color = 'magenta')
series.plot(kind='kde',ax=ax2,title="Distribution",color = 'magenta')
plt.show()

from statsmodels.tsa.seasonal import seasonal_decompose
fig, (ax1, ax2) = plt.subplots(1,2, figsize=(20,8))
result = seasonal_decompose(series)
result.trend.plot(color = "magenta",ax=ax1)
plot_acf(series,ax=ax2);


def test_stationarity(series):
    rolmean = series.rolling(12).mean()
    rolstd = series.rolling(12).std()
    plt.plot(series, color='magenta', label='Electricity Consumption')
    plt.plot(rolmean, label='Rolling Mean')
```

```
    plt.plot(rolstd, label = 'Rolling Std')
    plt.title('Rolling Mean and Std')
    plt.legend()
    adft = adfuller(series['Consumption'],autolag='AIC')
    labels = ['adf','p-value','usedlag','nobs','critical values']
    for i in range(5):
        print(labels[i]," ",adft[i])

def test_stationarity_graph(series):
    rolmean = series.rolling(12).mean()
    rolstd = series.rolling(12).std()
    plt.plot(series, color='magenta', label='Electricity Consumption')
    plt.plot(rolmean, label='Rolling Mean')
    plt.plot(rolstd, label = 'Rolling Std')
    plt.title('Rolling Mean and Std')
    plt.legend()

test_stationarity(series)

series_log = np.log(series)
moving_avg = series_log.rolling(12).mean()
std_dev = series_log.rolling(12).std()
plt.plot(series_log)
plt.plot(moving_avg, color="red")
plt.plot(std_dev, color ="black")
plt.show()

series_log_diff = series_log - series_log.shift()

plt.xlabel("Date")
plt.ylabel("Consumption")
plt.plot(series_log_diff)
series_log_diff.dropna(inplace=True)
test_stationarity(series_log_diff)

result = seasonal_decompose(series_log, model='additive', period = 12)
residual = result.resid
residual.dropna(inplace=True)
test_stationarity_graph(residual)

fig, (ax1, ax2) = plt.subplots(1,2, figsize=(20,8))
plot_acf(series_log_diff,ax=ax1);
plot_pacf(series_log_diff,ax=ax2);

```

```
import warnings
warnings.filterwarnings('ignore', 'statsmodels.tsa.arima_model.ARMA',
                        FutureWarning)
warnings.filterwarnings('ignore', 'statsmodels.tsa.arima_model.ARIMA',
                        FutureWarning)

# warnings.warn(ARIMA_DEPRECATION_WARN, FutureWarning)

from statsmodels.tsa.arima_model import ARIMA

model = ARIMA(series_log, order=(2,1,2))
result_AR = model.fit(disp = 0)
plt.plot(series_log_diff)
plt.plot(result_AR.fittedvalues, color='red')
plt.title("sum of squares of residuals")
print('RSS :', sum((result_AR.fittedvalues-series_log_diff["Consumption"
    ])**2))

series_log

from datetime import datetime
from dateutil.relativedelta import relativedelta

def generate_dates(n, start_date):
    dates = []
    current_date = datetime.strptime(start_date, '%Y-%m-%d')

    for _ in range(n):
        dates.append(current_date.date())
        current_date += relativedelta(months=1)

    return dates

result = generate_dates(20, '2018-01-01')
predicted_values = pd.DataFrame({'Date': result, 'Consumption': result_AR
    .forecast(steps=20)[0]})

predicted_values.set_index('Date', inplace=True)

fig, axs = plt.subplots(1, 2, figsize=(12, 4))

axs[0].plot(predicted_values, color='green')
axs[0].plot(series_log, color='magenta')
axs[0].set_title('Log data')
```

```
140
141 axs[1].plot(np.exp(predicted_values), color='green')
142 axs[1].plot(series, color='magenta')
143 axs[1].set_title('Original data')
144
145 # Adjust spacing between subplots
146 plt.tight_layout()
147
148 # Show the plot
149 plt.show()
```

2.2 Exercise 2

```
1  import statsmodels.api as sm
2  import numpy as np
3  import matplotlib.pyplot as plt
4  from matplotlib.pyplot import figure
5
6  # arparams = np.array([1, -0.9])
7  # maparams = np.array([])
8  # ar = np.r_[1, -arparams]
9  # ma = np.r_[1, maparams]
10 # y = sm.tsa.arma_generate_sample(ar, ma, 200)
11
12 # figure(figsize=(15, 6))
13 # plt.plot(y, color="magenta")
14
15 import numpy as np
16 from statsmodels.tsa.stattools import pacf
17
18 with open('C:/Users/sutob/Desktop/mno.txt', 'r') as file:
19     lines = file.readlines()
20
21 numeric_lines = [list(map(float, line.split())) for line in lines]
22 matrix_array = np.array(numeric_lines)
23
24 pacf(matrix_array,nlags=2)
25 figure(figsize=(15, 6))
26 plt.plot(matrix_array, color="magenta")
27
28 def durbin_levinson(time_series, order):
29     sample_autocov = compute_sample_autocovariance(time_series, order +
    1)
30     phi = np.zeros((order, order))
```

```
    variances = np.zeros(order)
    phi[0, 0] = sample_autocov[1] / sample_autocov[0]
    variances[0] = sample_autocov[0] * (1 - phi[0, 0]**2)

    for i in range(1, order):
        temp_var = sample_autocov[i+1] - sum([phi[i-1, j] *
    sample_autocov[i-j] for j in range(i)])
        phi[i, i] = temp_var / variances[i-1]
        for j in range(i):
            phi[i, j] = phi[i-1, j] - phi[i, i] * phi[i-1, i-j-1]
        variances[i] = variances[i-1] * (1 - phi[i, i]**2)

    return phi


def compute_sample_autocovariance(time_series, max_lag):
    n_samples = len(time_series)
    mean_val = np.mean(time_series)
    autocovs = [np.sum((time_series[:n_samples-lag] - mean_val) * (
    time_series[lag:] - mean_val)) / n_samples for lag in range(max_lag +
    1)]

    return autocovs

durbin_levinson(matrix_array, p=2)

from statsmodels.tsa.stattools import pacf
pacf(matrix_array,nlags=2)
```

3 Bibliography

1. Statistics for Data Science Lecture - Marina Anca Cidota. 2023 University of Bucharest, Faculty of Math & Computer Science
2. How to Create an ARIMA Model for Time Series Forecasting in Python - Jason Brownlee. (https://machinelearningmastery.com/arima-for-time-series-forecasting-with-pyt
3. Levinson algorithm in python - Simd. 2014 (https://stackoverflow.com/questions/21562651/levinson-algorithm-in-python)
4. Source code for statsmodels.tsa.stattools (https://www.statsmodels.org/dev/_modules/statsmodels/tsa/stattools.html)