Modules

In [72]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller

from scipy import stats
from datetime import timedelta, date
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX

from sklearn.metrics import mean_squared_error
from math import sqrt
```

Read the Dataset

In [73]:
```
df = pd.read_csv('/content/Electric_Production.csv', parse_dates=['DATE'])
df.set_index('DATE', inplace=True) # Make the DATE value as a index, lag, or t
df
```

Out[73]:

DATE	Value
1985-01-01	72.5052
1985-02-01	70.6720
1985-03-01	62.4502
1985-04-01	57.4714
1985-05-01	55.3151
...	...
2017-09-01	98.6154
2017-10-01	93.6137
2017-11-01	97.3359
2017-12-01	114.7212
2018-01-01	129.4048

397 rows × 1 columns

In [74]:
```
df.isnull().any()
```

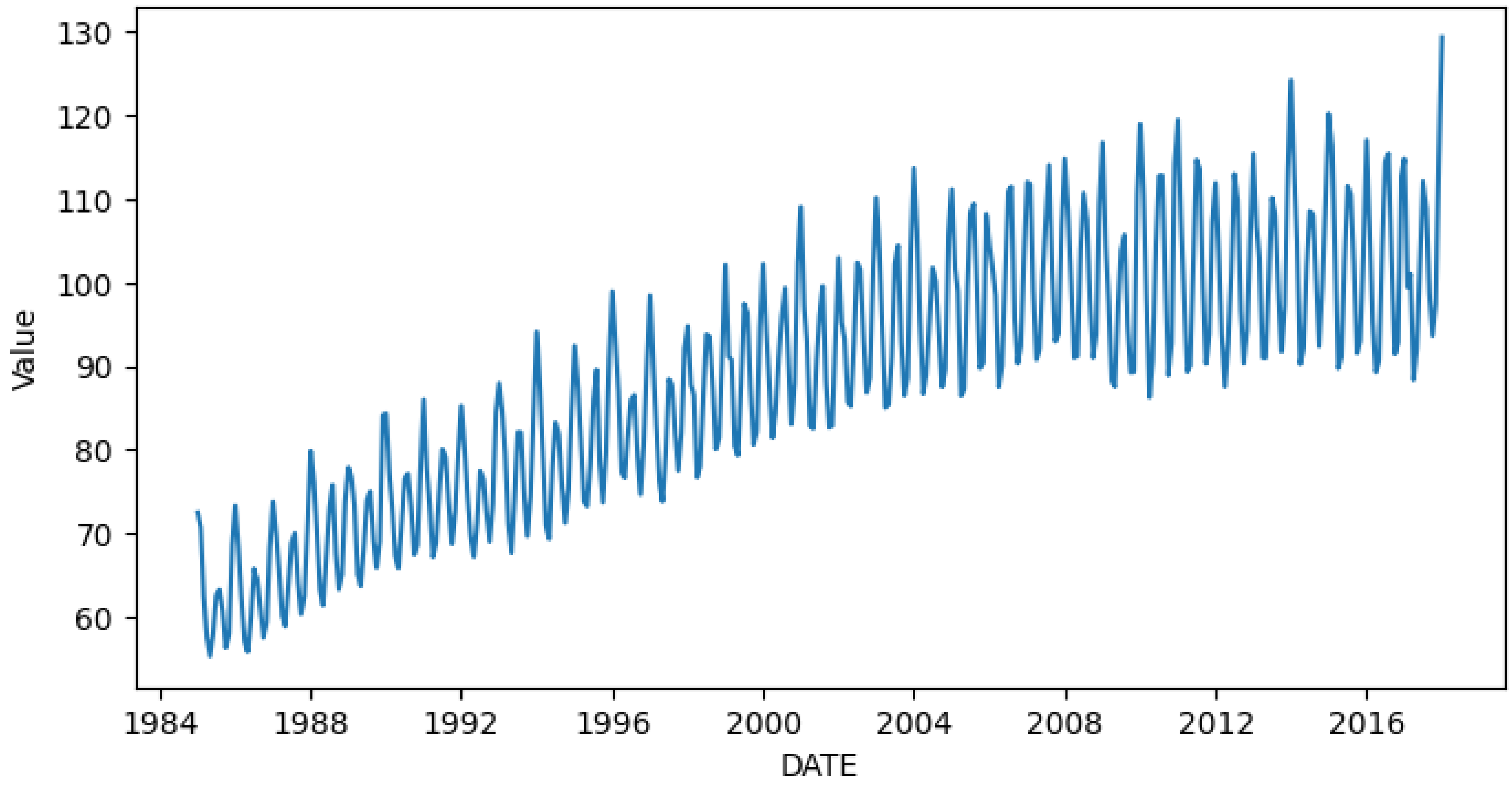
Out[74]:
```
Value    False
dtype: bool
```

In [75]:
```
df.index.min(), df.index.max()
```

Out[75]:
```
(Timestamp('1985-01-01 00:00:00'), Timestamp('2018-01-01 00:00:00'))
```

Plot the Orignial Dataset

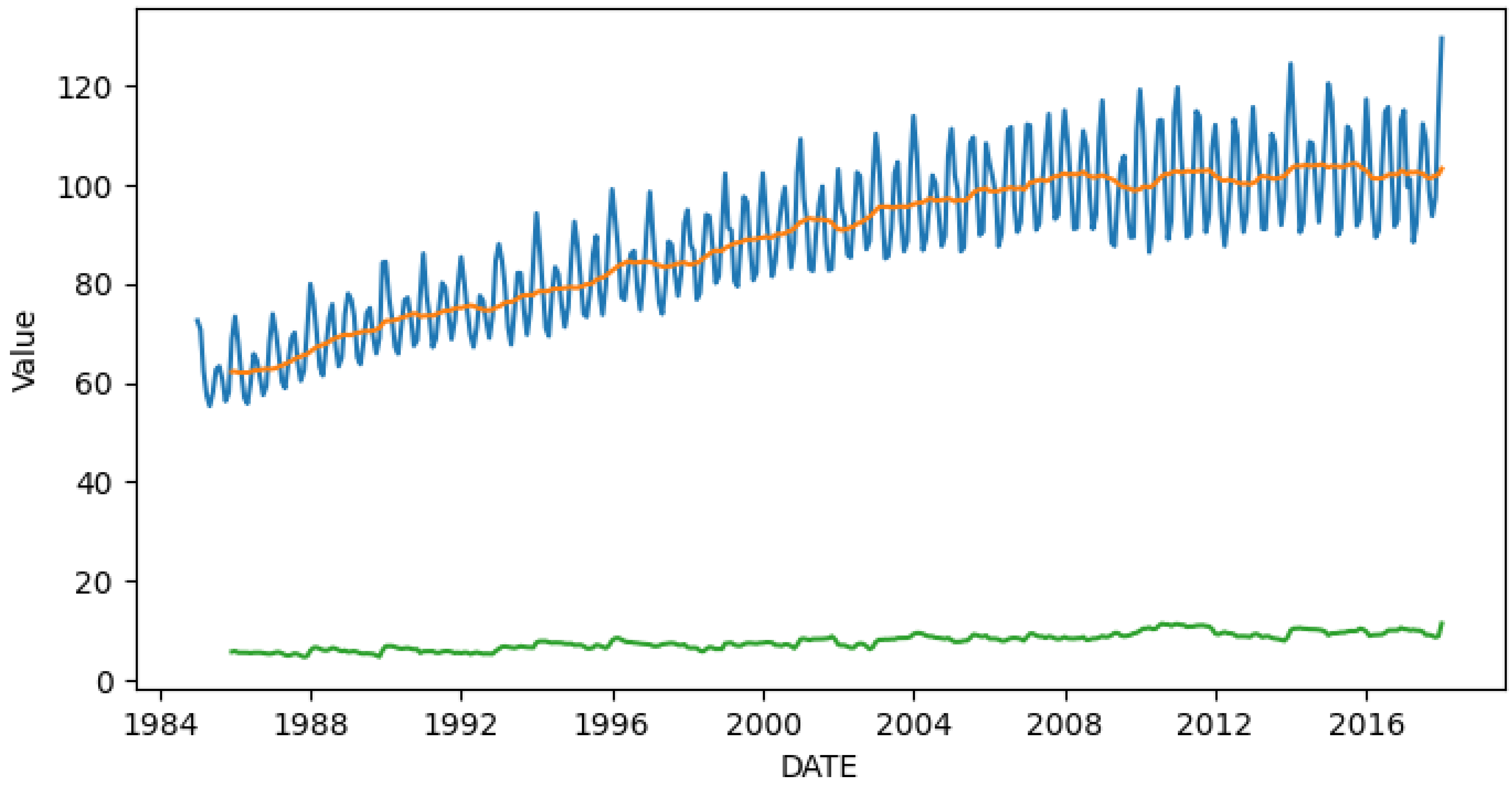
In [ ]:
```
plt.figure(figsize=(8, 4))
sns.lineplot(data=df, x=df.index,y=df.Value)
plt.show()
```



In [76]:
```
df['rollMean'] = df.Value.rolling(window=12).mean()
df['rollStd'] = df.Value.rolling(window=12).std()
```

In [77]:
```
plt.figure(figsize = (8,4))
sns.lineplot(data=df, x=df.index, y=df.Value)
sns.lineplot(data=df, x=df.index, y=df.rollMean)
sns.lineplot(data=df, x=df.index, y=df.rollStd)
```

Out[77]:
```
<Axes: xlabel='DATE', ylabel='Value'>
```



Dickey-Fuller Test

In [78]:
```
adfTest = adfuller(df['Value'],autolag = "AIC",)
adfTest
```

Out[78]:
```
(-2.256990350047235,
 0.1862146911658712,
 15,
 381,
 {'1%': -3.4476305904172904,
  '5%': -2.869155980820355,
  '10%': -2.570827146203181},
 1840.8474501627156)
```

In [79]:
```
stats = pd.Series(adfTest[0:4],index=['Test Statistic', 'p-value','#lags used','number of observations used'])
stats # it's not stationary since the p-value is bigger than 0.05
```

Out[79]:
```
Test Statistic                  -2.256990
p-value                          0.186215
#lags used                      15.000000
number of observations used    381.000000
dtype: float64
```

Dickey-Fuller Test for Time Shift Differencing and Logarithm

In [80]:
```
from typing import ValuesView
def test_stationarity(df, Value):
  df['rollMean'] = df[Value].rolling(window=12).mean()
  df['rollStd'] = df[Value].rolling(window=12).std()

  adfTest = adfuller(df[Value], autolag='AIC')
  stats = pd.Series(adfTest[0:4], index = ['Test Statistic','p-value','#lags used','number of observationis used'])
  print(stats)
```

```
for key, values in adfTest[4].items():
  print('criticality', key,':', values)

sns.lineplot(data=df, x=df.index,y=Value)
sns.lineplot(data=df, x=df.index,y= 'rollMean')
sns.lineplot(data=df, x=df.index,y= 'rollStd')
```

Time Shift Differencing

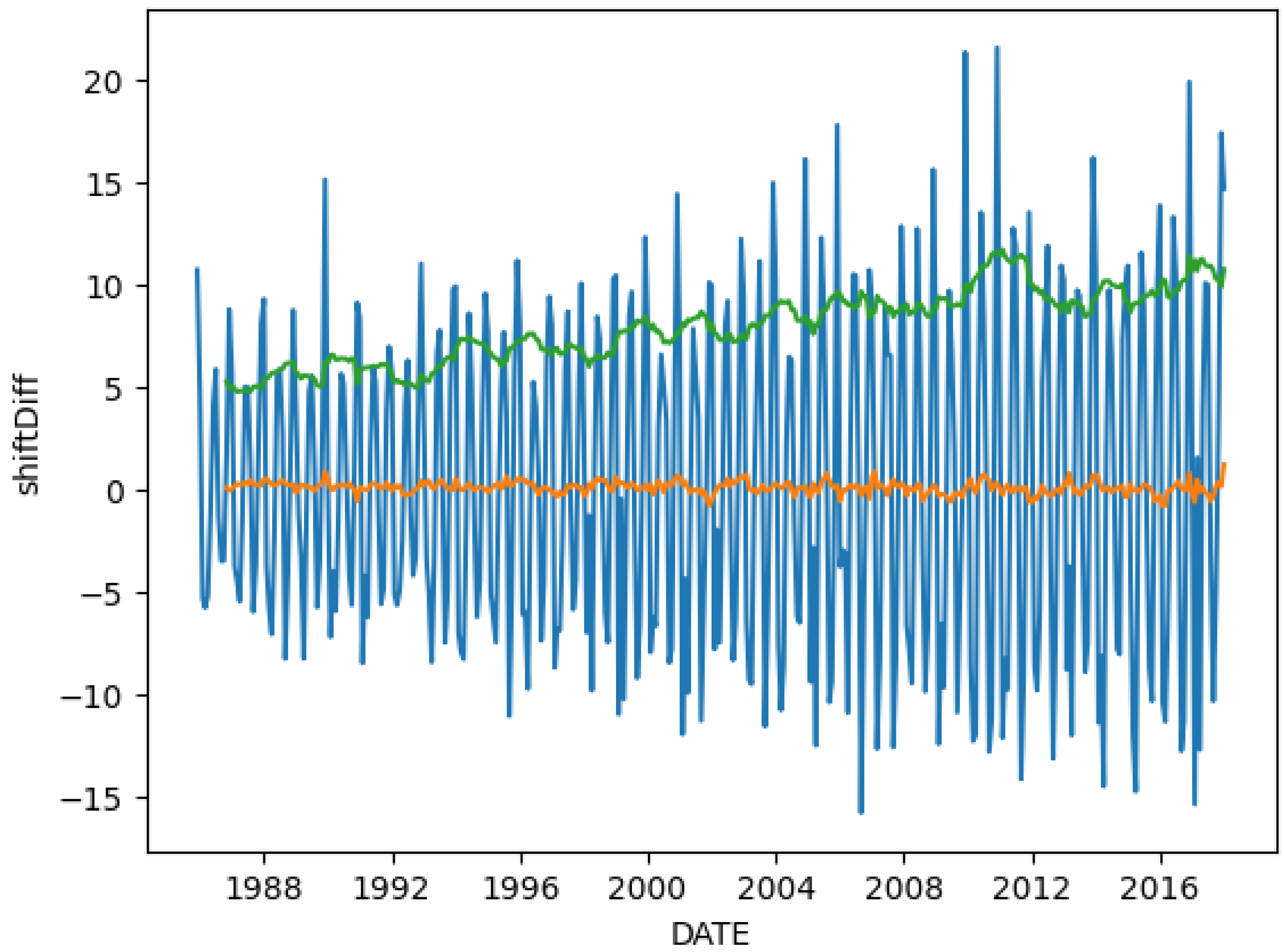
In [81]:
```
# Using Time shift Differencing
df['Shift'] = df.Value.shift()
df['shiftDiff'] = df['Value'] - df['Shift']
df.head()
```

Out[81]:

DATE	Value	rollMean	rollStd	Shift	shiftDiff
1985-01-01	72.5052	NaN	NaN	NaN	NaN
1985-02-01	70.6720	NaN	NaN	72.5052	-1.8332
1985-03-01	62.4502	NaN	NaN	70.6720	-8.2218
1985-04-01	57.4714	NaN	NaN	62.4502	-4.9788
1985-05-01	55.3151	NaN	NaN	57.4714	-2.1563

In [82]:
```
test_stationarity(df.dropna(),'shiftDiff')
```

```
Test Statistic                  -6.998284e+00
p-value                          7.435144e-10
#lags used                       1.400000e+01
number of observationis used     3.710000e+02
dtype: float64
criticality 1% : -3.4480996560263386
criticality 5% : -2.8693621113224137
criticality 10% : -2.570937038891028
```



Log

In [83]:
```
#Log
log_df = df[['Value']]
log_df['log'] = np.log(log_df['Value'])
log_df
```

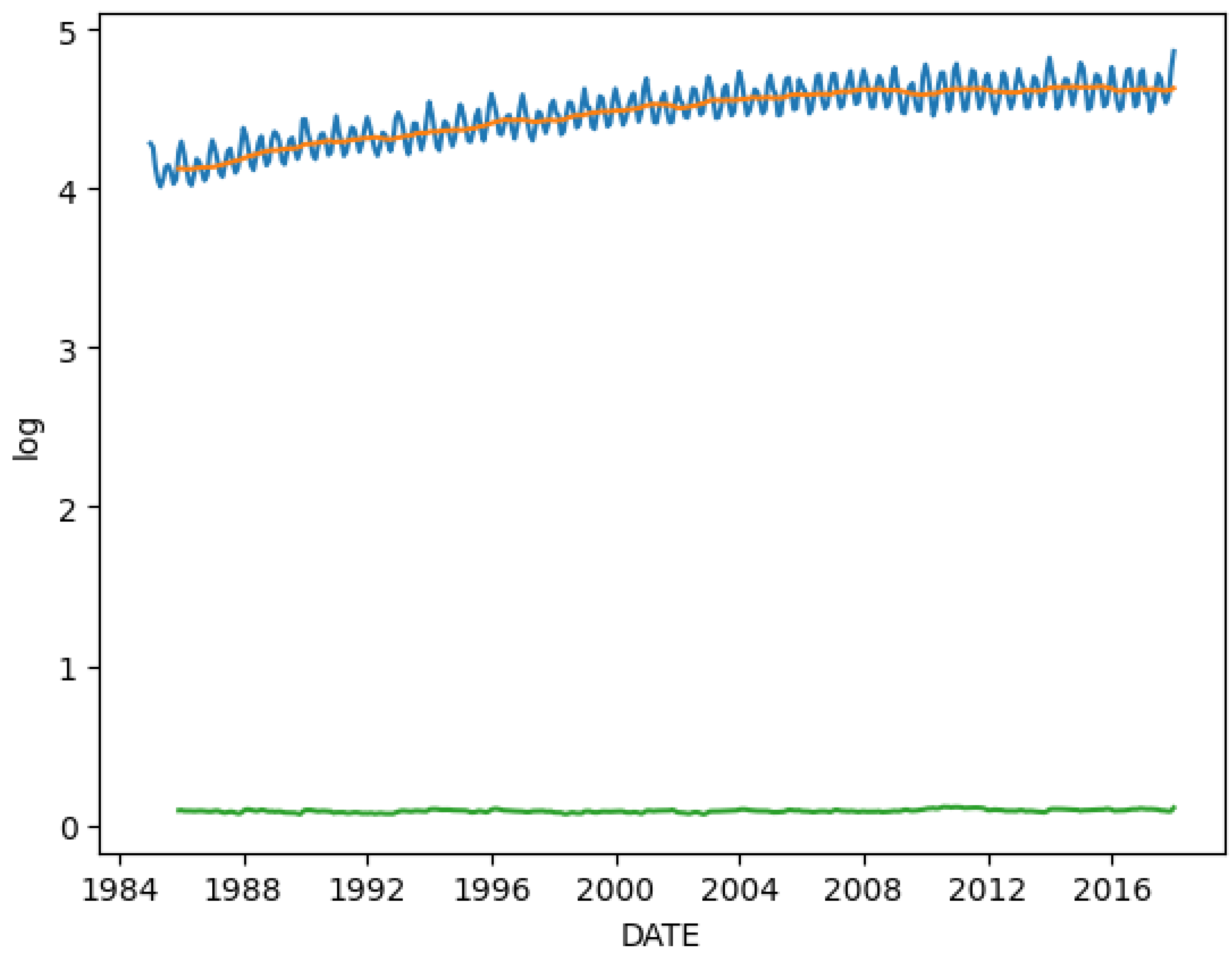
Out[83]:

	Value	log
DATE		
1985-01-01	72.5052	4.283658
1985-02-01	70.6720	4.258049
1985-03-01	62.4502	4.134369
1985-04-01	57.4714	4.051287
1985-05-01	55.3151	4.013046
...	...	...
2017-09-01	98.6154	4.591227
2017-10-01	93.6137	4.539177
2017-11-01	97.3359	4.578168
2017-12-01	114.7212	4.742505
2018-01-01	129.4048	4.862945

397 rows × 2 columns

In [84]:
```
log_df.isnull().any()
```
Out[84]:
```
Value    False
log      False
dtype: bool
```

In [85]:
```
test_stationarity(log_df,'log')
```
```
Test Statistic                       -3.145360
p-value                               0.023373
#lags used                           15.000000
number of observationis used        381.000000
dtype: float64
criticality 1% : -3.4476305904172904
criticality 5% : -2.869155980820355
criticality 10% : -2.570827146203181
```



In [86]:
```
log_df
```

Out[86]:

DATE	Value	log	rollMean	rollStd
1985-01-01	72.5052	4.283658	NaN	NaN
1985-02-01	70.6720	4.258049	NaN	NaN
1985-03-01	62.4502	4.134369	NaN	NaN
1985-04-01	57.4714	4.051287	NaN	NaN
1985-05-01	55.3151	4.013046	NaN	NaN
...	...	...	...	...
2017-09-01	98.6154	4.591227	4.613704	0.090198
2017-10-01	93.6137	4.539177	4.615619	0.088161
2017-11-01	97.3359	4.578168	4.619515	0.085080
2017-12-01	114.7212	4.742505	4.620945	0.087140
2018-01-01	129.4048	4.862945	4.630888	0.106964

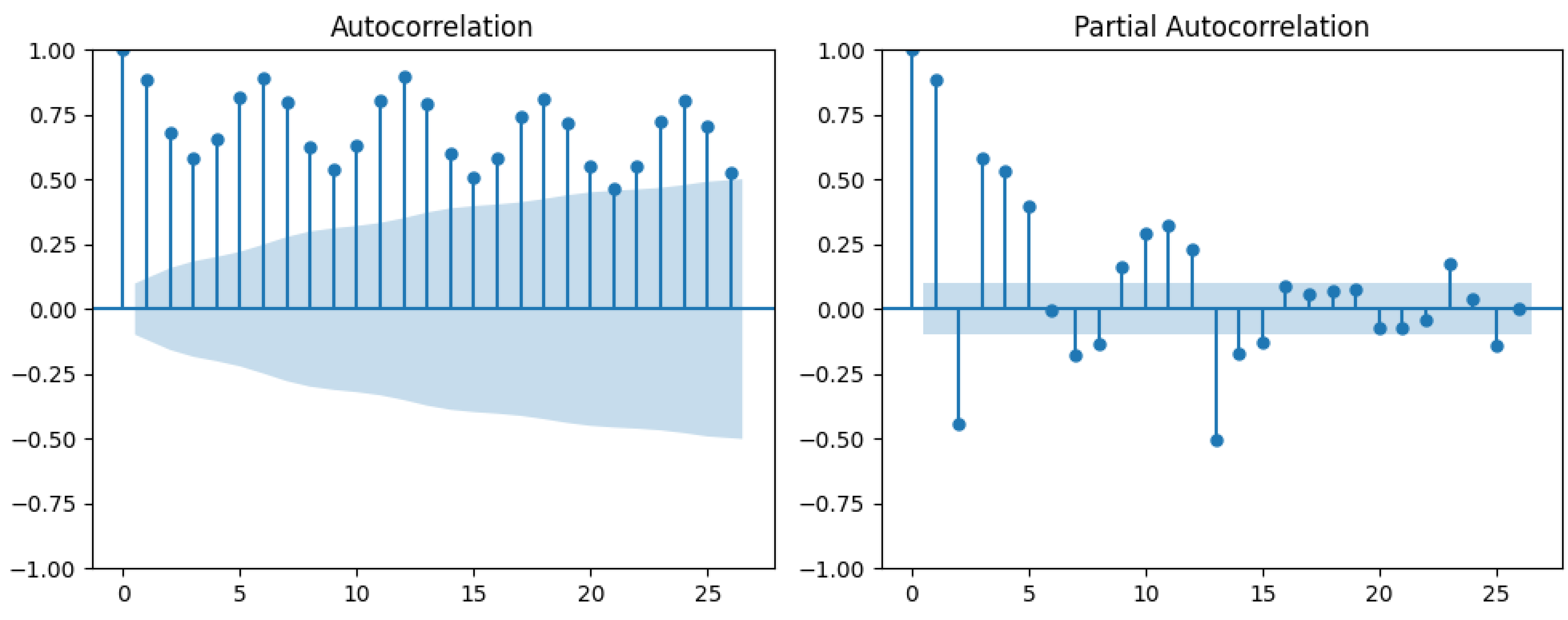
397 rows × 4 columns

In [87]:
```
result = adfuller(log_df['log'])
p_value = result[1]
if p_value < 0.05:
    print("The time series is stationary.")
else:
    print("The time series is non-stationary.")
```

```
The time series is stationary.
```

Autocorrelation Function and Partial Autocorrelation Function

In [88]:
```
# Plot the Autocorrelation Function (ACF)
plt.figure(figsize=(10, 4))
ax1 = plt.subplot(121)
plot_acf(log_df['log'], ax=ax1)

# Plot the Partial Autocorrelation Function (PACF)
ax2 = plt.subplot(122)
plot_pacf(log_df['log'], ax=ax2)

plt.tight_layout()
plt.show()
```



In [95]:
```
# ACF = tails off, PACF = cut off
# We can use AR(1)

from IPython.display import Image

image_url = "https://drive.google.com/uc?export=download&id=1rCIpCv3MwjAcKNPv7PrfB-NCLn9UcmI4"
Image(url=image_url)
#Source: University of Pittsburgh
```

Out[95]:

Choosing Model Specification

- Recall we have discussed that ACF and PACF can be used for determining ARIMA model hyperparamters p and q.

	$AR(p)$	$MA(q)$	$ARMA(p,q)$
ACF	Tails off	Cuts off after lag q	Tails off
PACF	Cuts off after lag p	Tails off	Tails off

- Other criterions can be used for choosing q and q too, such as AIC (Akaike Information Criterion), AICc (corrected AIC) and BIC (Bayesian Information Criterion).
- Note that the selection for p and q is not unique.

ARIMA

In [96]:

```
# Fit the RA model
ar_model = ARIMA(log_df['log'], order=(2,1,2)).fit()

# Print the model summary
print(ar_model.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                    log   No. Observations:                  397
Model:                 ARIMA(2, 1, 2)   Log Likelihood                 754.803
Date:                Fri, 28 Jul 2023   AIC                          -1499.606
Time:                        14:42:01   BIC                          -1479.699
Sample:                    01-01-1985   HQIC                         -1491.719
                         - 01-01-2018
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.9993      0.001    984.991      0.000       0.997       1.001
ar.L2         -0.9998      0.000  -3735.875      0.000      -1.000      -0.999
ma.L1         -1.0338      0.068    -15.190      0.000      -1.167      -0.900
ma.L2          0.9986      0.127      7.876      0.000       0.750       1.247
sigma2         0.0013      0.000      7.221      0.000       0.001       0.002
===================================================================================
Ljung-Box (L1) (Q):                   6.59   Jarque-Bera (JB):                 8.06
Prob(Q):                              0.01   Prob(JB):                         0.02
Heteroskedasticity (H):               1.19   Skew:                             0.07
Prob(H) (two-sided):                  0.31   Kurtosis:                         3.69
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

In [97]:

```
log_df
```

Out[97]:

DATE	Value	log	rollMean	rollStd
1985-01-01	72.5052	4.283658	NaN	NaN
1985-02-01	70.6720	4.258049	NaN	NaN
1985-03-01	62.4502	4.134369	NaN	NaN
1985-04-01	57.4714	4.051287	NaN	NaN
1985-05-01	55.3151	4.013046	NaN	NaN
...	...	...	...	...
2017-09-01	98.6154	4.591227	4.613704	0.090198
2017-10-01	93.6137	4.539177	4.615619	0.088161
2017-11-01	97.3359	4.578168	4.619515	0.085080
2017-12-01	114.7212	4.742505	4.620945	0.087140
2018-01-01	129.4048	4.862945	4.630888	0.106964

397 rows × 4 columns

Predicted Value Based on Logarithm Approach with ARIMA

In [98]:

```
# Take the logarithm of the original data
train_log = np.log(log_df['Value'])

# Fit the ARIMA model to the log-transformed data
model_arima = ARIMA(train_log, order=(2, 1, 2))
model = model_arima.fit()

# Get the forecasts from the ARIMA model in log-transformed units
forecast_log = model.forecast(steps=12)

# Convert the forecasts back to the original units by taking the exponential
forecast_original_units_value = np.exp(forecast_log)

# Print the forecasts in the original units
print(forecast_original_units_value)
```

```
2018-02-01    124.440821
2018-03-01    109.838519
2018-04-01    100.825176
2018-05-01    104.859536
2018-06-01    118.799326
2018-07-01    129.403519
2018-08-01    124.410136
2018-09-01    109.812540
2018-10-01    100.826203
2018-11-01    104.885407
2018-12-01    118.827405
2019-01-01    129.402162
Freq: MS, Name: predicted_mean, dtype: float64
```

Train the Model

In [99]:

```
print(log_df.shape)
train=log_df['log'].iloc[:-12]
test=log_df['log'].iloc[-12:]
print(train.shape, test.shape)
```

```
(397, 4)
(385,) (12,)
```

In [100…

```
start=len(train)
end=len(train)+len(test)-1
pred=model.predict(start=start, end=end,type='levels') #ARIMA
# predictions_original_units = np.exp(pred)
print(pred) #predictions_original_units
pred.index=log_df.index[start:end+1]
```

```
2017-02-01    4.701892
2017-03-01    4.476696
2017-04-01    4.529986
2017-05-01    4.520083
2017-06-01    4.647055
2017-07-01    4.712697
2017-08-01    4.681536
2017-09-01    4.565866
2017-10-01    4.503922
2017-11-01    4.575488
2017-12-01    4.701479
2018-01-01    4.828078
Freq: MS, Name: predicted_mean, dtype: float64
```

In [101…

```
print(test.index, train.index)
```

```
DatetimeIndex(['2017-02-01', '2017-03-01', '2017-04-01', '2017-05-01',
               '2017-06-01', '2017-07-01', '2017-08-01', '2017-09-01',
               '2017-10-01', '2017-11-01', '2017-12-01', '2018-01-01'],
              dtype='datetime64[ns]', name='DATE', freq=None) DatetimeIndex(['1985-01-01', '1985-02-01', '1985-03-01', '1985-04-01',
               '1985-05-01', '1985-06-01', '1985-07-01', '1985-08-01',
               '1985-09-01', '1985-10-01',
               ...
               '2016-04-01', '2016-05-01', '2016-06-01', '2016-07-01',
               '2016-08-01', '2016-09-01', '2016-10-01', '2016-11-01',
               '2016-12-01', '2017-01-01'],
              dtype='datetime64[ns]', name='DATE', length=385, freq=None)
```

In [102…

```
test_df = test.to_frame()
train_df = train.to_frame()
print(test_df['log'])
print(train_df['log'])
```

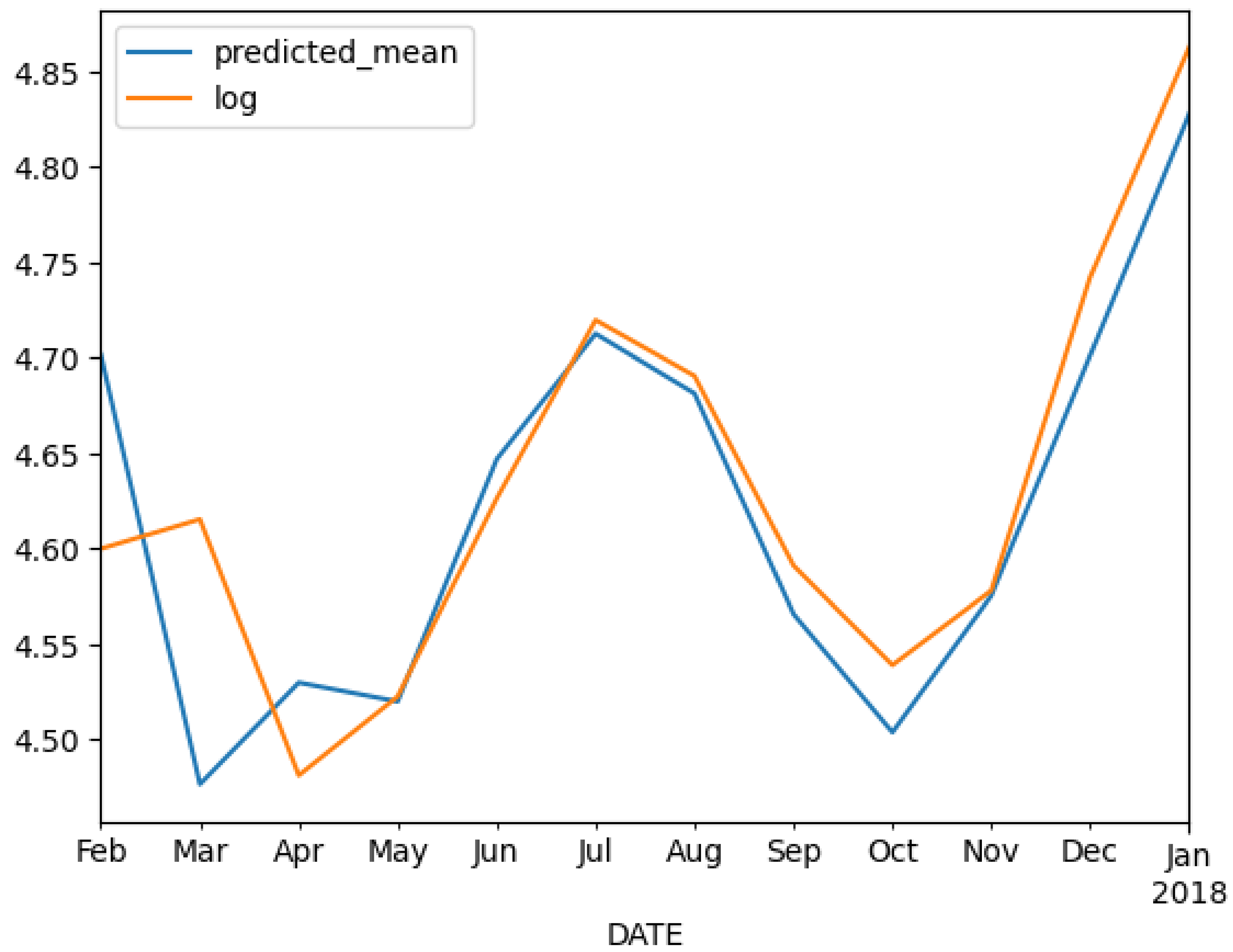
```
DATE
2017-02-01    4.600058
2017-03-01    4.615513
2017-04-01    4.481340
2017-05-01    4.522663
2017-06-01    4.626474
2017-07-01    4.719871
2017-08-01    4.690716
2017-09-01    4.591227
2017-10-01    4.539177
2017-11-01    4.578168
2017-12-01    4.742505
2018-01-01    4.862945
Name: log, dtype: float64
DATE
1985-01-01    4.283658
1985-02-01    4.258049
1985-03-01    4.134369
1985-04-01    4.051287
1985-05-01    4.013046
                ...
2016-09-01    4.632432
2016-10-01    4.516194
2016-11-01    4.531416
2016-12-01    4.725345
2017-01-01    4.743631
Name: log, Length: 385, dtype: float64
```

In [103…

```
pred.plot(legend=True)
test_df['log'].plot(legend=True)
```

Out[103]:

```
<Axes: xlabel='DATE'>
```



In [104…

```
test_mean = test_df['log'].mean()
rmse = sqrt(mean_squared_error(test_df, pred))
print(rmse) #Root Mean Squared Error (RMSE)
```

```
0.05580080263050337
```

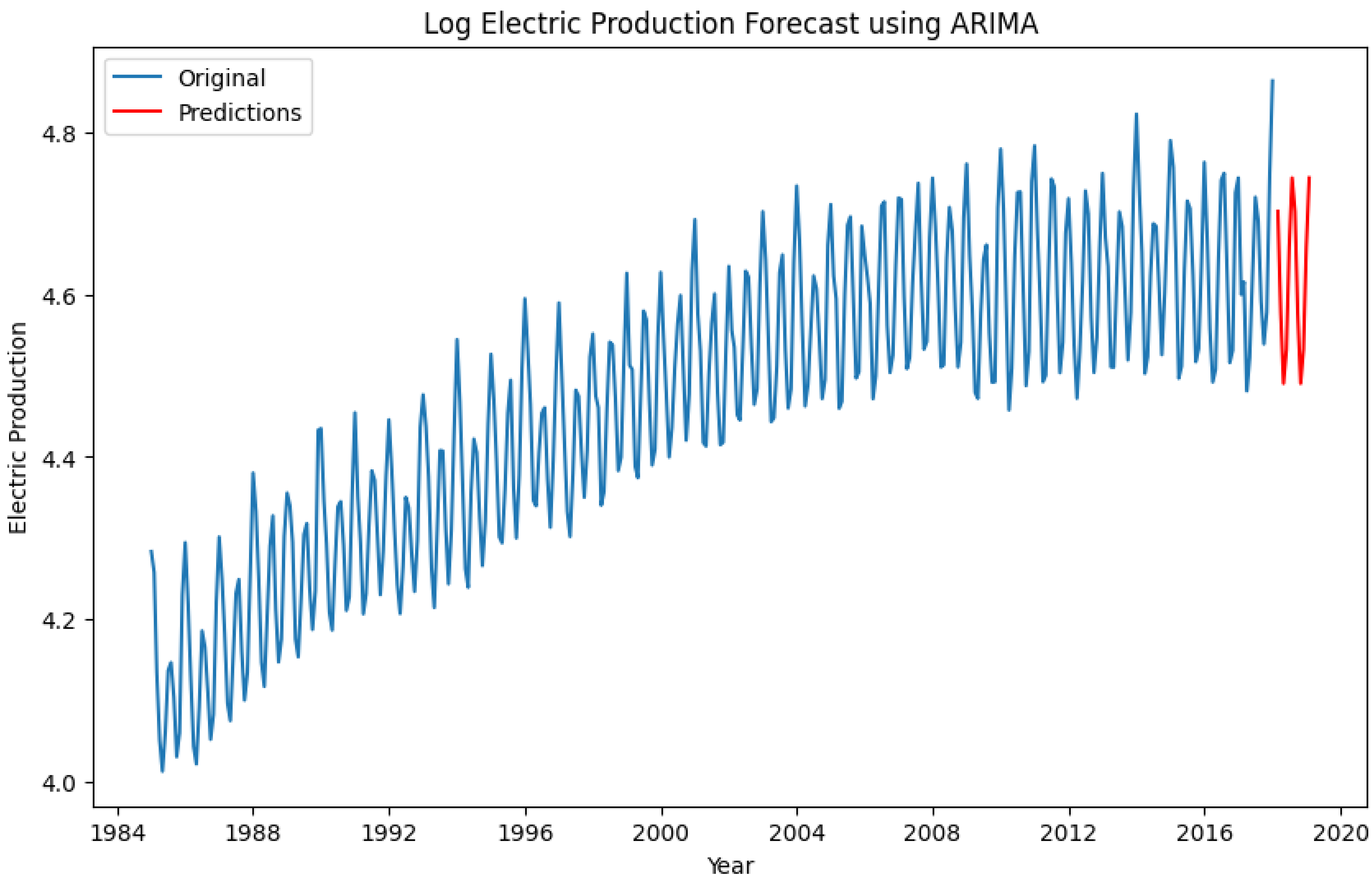
Comparing ARIMA and SARIMAX

ARIMA

In [105…

```
train = log_df['log'].iloc[:-12]
test = log_df['log'].iloc[-12:]
# Fit the ARIMA model
arima_model = ARIMA(train, order=(2, 1, 2))
fitted_model = arima_model.fit()
```

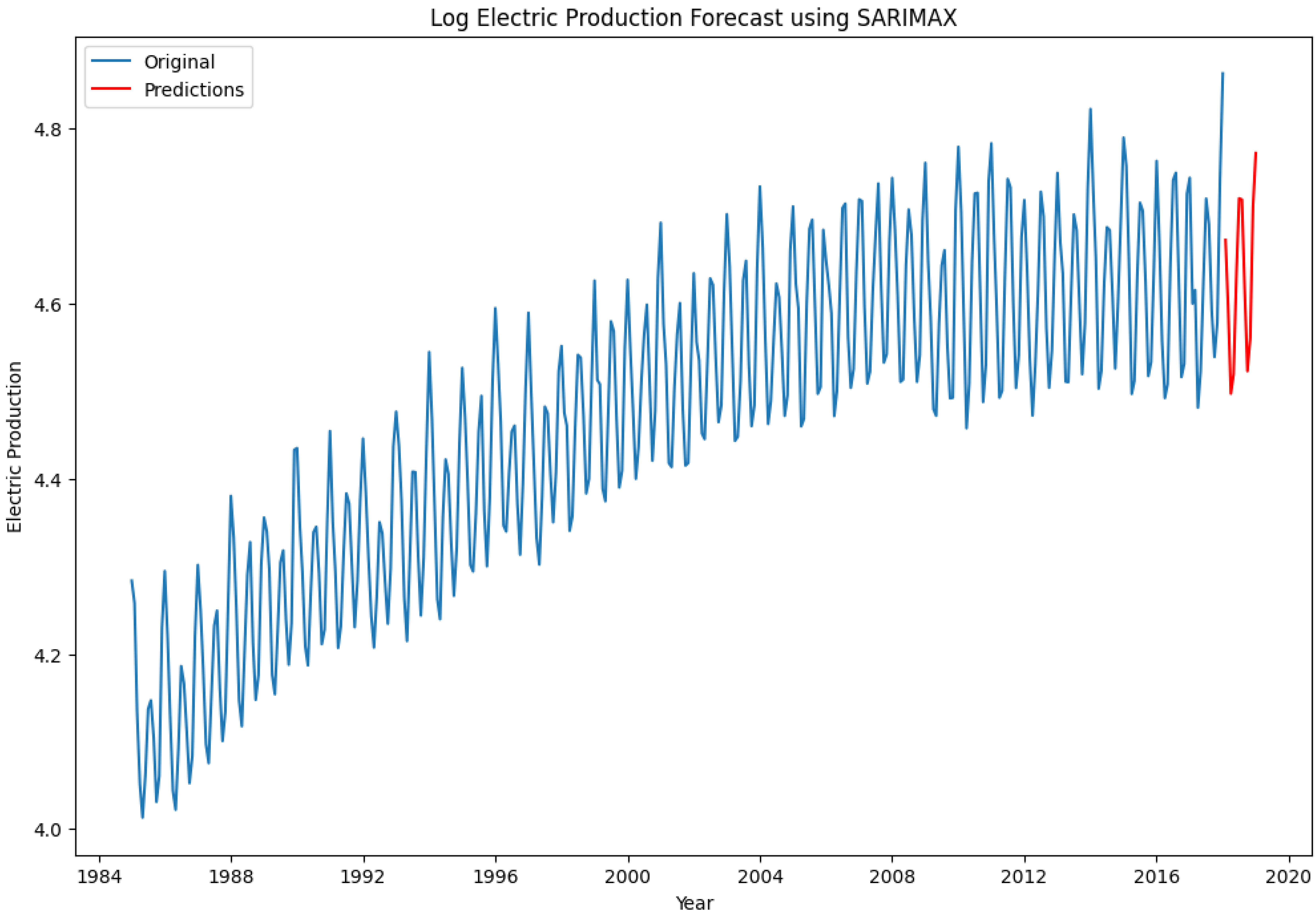
```
# Make predictions
predictions = fitted_model.forecast(steps=len(test))

# Extend the time index for predictions
future_index = pd.date_range(start=log_df.index[-1], periods=len(test) + 1, freq='M')
predictions.index = future_index[1:]

# Visualize the original time series and the predictions
plt.figure(figsize=(10, 6))
plt.plot(log_df['log'], label='Original')
plt.plot(predictions, color='red', label='Predictions')
plt.xlabel('Year')
plt.ylabel('Electric Production')
plt.title('Log Electric Production Forecast using ARIMA')
plt.legend()
plt.show()
```



SARIMAX

In [106…

```
# Split the data into training and test sets
train = log_df['log'].iloc[:-12]
test = log_df['log'].iloc[-12:]

# Fit the SARIMAX model
sarima_model = SARIMAX(train, order=(2, 1, 2), seasonal_order=(2, 1, 2, 12))
fitted_model = sarima_model.fit()

# Make predictions
predictions = fitted_model.get_forecast(steps=len(test))

# Get the confidence intervals for the predictions
pred_confidence = predictions.conf_int()

# Extend the time index for predictions
future_index = pd.date_range(start=log_df.index[-1], periods=len(test), freq='M')

# Visualize the original time series and the predictions
plt.figure(figsize=(12, 8))
plt.plot(log_df['log'], label='Original')
plt.plot(future_index, predictions.predicted_mean, color='red', label='Predictions')
plt.xlabel('Year')
plt.ylabel('Electric Production')
plt.title('Log Electric Production Forecast using SARIMAX')
plt.legend()
plt.show()
```



ARIMA and SARIMAX Future Values

In [107…

```
# ARIMA
# Take the logarithm of the data original
train_log = np.log(log_df['Value'])

# Fit the ARIMA model to the log-transformed data
model_arima = ARIMA(train_log, order=(2, 1, 2))
model = model_arima.fit()

# Get the forecasts from the ARIMA model in log-transformed units
forecast_log = model.forecast(steps=12)

# Convert the forecasts back to the original units by taking the exponential
forecast_original_units_arima = np.exp(forecast_log)

# Print the forecasts in the original units
print(forecast_original_units_arima)
```

```
2018-02-01    124.440821
2018-03-01    109.838519
2018-04-01    100.825176
2018-05-01    104.859536
2018-06-01    118.799326
2018-07-01    129.403519
2018-08-01    124.410136
2018-09-01    109.812540
2018-10-01    100.826203
2018-11-01    104.885407
2018-12-01    118.827405
2019-01-01    129.402162
Freq: MS, Name: predicted_mean, dtype: float64
```

In [108…

```
# SARIMAX
# Take the logarithm of the original data
train_log = np.log(log_df['Value'])

# Fit the SARIMAX model to the log-transformed data
sarimax_model = SARIMAX(train_log, order=(2, 1, 2), seasonal_order=(2, 1, 2, 12))
fitted_model = sarimax_model.fit()

# Get the forecasts from the SARIMAX model in log-transformed units
forecast_log = fitted_model.forecast(steps=12)

# Convert the forecasts back to the original units by taking the exponential
forecast_original_units_sarimax = np.exp(forecast_log)

# Print the forecasts in the original units
print(forecast_original_units_sarimax)
```

```
2018-02-01    114.308770
2018-03-01    105.480066
2018-04-01     92.207340
2018-05-01     93.816401
2018-06-01    104.391530
2018-07-01    113.643464
2018-08-01    112.271775
2018-09-01    101.749260
2018-10-01     93.949382
2018-11-01     97.401239
2018-12-01    111.889007
2019-01-01    123.187538
Freq: MS, Name: predicted_mean, dtype: float64
```

Comparing Results

In [109…
```
print(forecast_original_units_arima.index, forecast_original_units_sarimax.index)
```

```
DatetimeIndex(['2018-02-01', '2018-03-01', '2018-04-01', '2018-05-01',
               '2018-06-01', '2018-07-01', '2018-08-01', '2018-09-01',
               '2018-10-01', '2018-11-01', '2018-12-01', '2019-01-01'],
              dtype='datetime64[ns]', freq='MS') DatetimeIndex(['2018-02-01', '2018-03-01', '2018-04-01', '2018-05-01',
               '2018-06-01', '2018-07-01', '2018-08-01', '2018-09-01',
               '2018-10-01', '2018-11-01', '2018-12-01', '2019-01-01'],
              dtype='datetime64[ns]', freq='MS')
```

In [110…
```
# ARIMA
arima_forecast = forecast_original_units_arima.to_frame()
test_mean_arima = arima_forecast.mean()
rmse1 = sqrt(mean_squared_error(arima_forecast, pred))
print(rmse1) #Root Mean Squared Error (RMSE)
```

```
110.549056918131
```

In [111…
```
# SARIMAX
sarimax_forecast = forecast_original_units_sarimax.to_frame()
test_mean_sarimax = sarimax_forecast.mean()
rmse2 = sqrt(mean_squared_error(sarimax_forecast, pred))
print(rmse2) #Root Mean Squared Error (RMSE)
```

```
101.17101092483746
```

In [ ]:
```
# SARIMAX test result's is more better than ARIMA
```

In [112…
```
# Then, we will use SARIMAX predicted value
predicted_value_sarimax1 = forecast_original_units_sarimax.to_frame()
print(predicted_value_sarimax1)
```

```
            predicted_mean
2018-02-01      114.308770
2018-03-01      105.480066
2018-04-01       92.207340
2018-05-01       93.816401
2018-06-01      104.391530
2018-07-01      113.643464
2018-08-01      112.271775
2018-09-01      101.749260
2018-10-01       93.949382
2018-11-01       97.401239
2018-12-01      111.889007
2019-01-01      123.187538
```

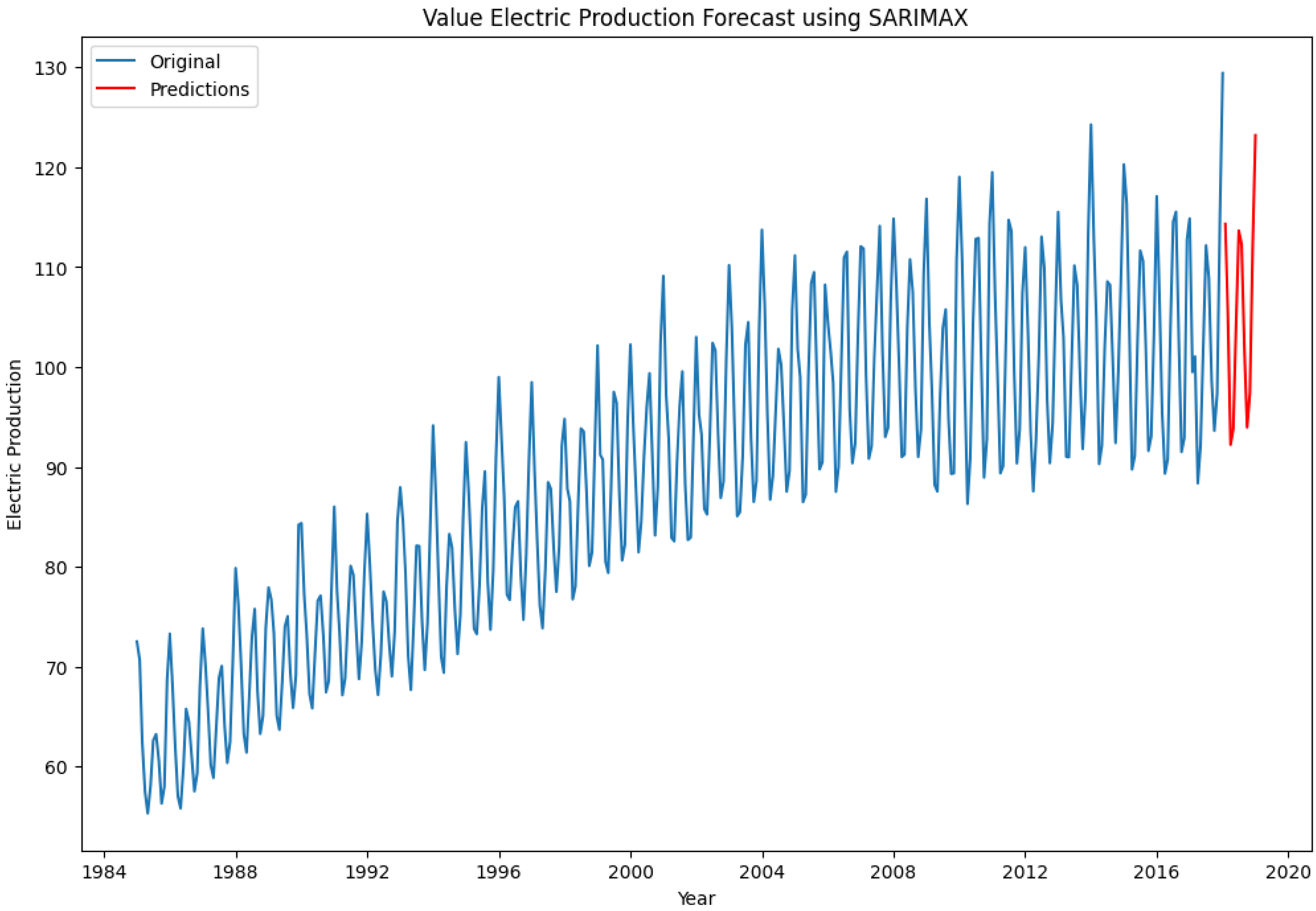
Final Result for The Prediction

In [113…
```
# Convert the Series to a 1-dimensional array using .values or .to_numpy()
predicted_value_sarimax_array = predicted_value_sarimax1.values.ravel()

# Create a DataFrame from the 1-dimensional array
predicted_df = pd.DataFrame({
    'Date': predicted_value_sarimax1.index,
    'predicted value': predicted_value_sarimax_array
})
print(predicted_df)
```

```
         Date  predicted value
0  2018-02-01       114.308770
1  2018-03-01       105.480066
2  2018-04-01        92.207340
3  2018-05-01        93.816401
4  2018-06-01       104.391530
5  2018-07-01       113.643464
6  2018-08-01       112.271775
7  2018-09-01       101.749260
8  2018-10-01        93.949382
9  2018-11-01        97.401239
10 2018-12-01       111.889007
11 2019-01-01       123.187538
```

In [114…
```
# Original Value
plt.figure(figsize=(12, 8))
plt.plot(log_df['Value'], label='Original')
plt.plot(future_index, predicted_df['predicted value'], color='red', label='Predictions')
plt.xlabel('Year')
```

```python
plt.ylabel('Electric Production')
plt.title('Value Electric Production Forecast using SARIMAX')
plt.legend()
plt.show()

predicted_df
```



Out[114]:

	Date	predicted value
0	2018-02-01	114.308770
1	2018-03-01	105.480066
2	2018-04-01	92.207340
3	2018-05-01	93.816401
4	2018-06-01	104.391530
5	2018-07-01	113.643464
6	2018-08-01	112.271775
7	2018-09-01	101.749260
8	2018-10-01	93.949382
9	2018-11-01	97.401239
10	2018-12-01	111.889007
11	2019-01-01	123.187538

In [ ]: