arima

October 15, 2024

1 About Dataset:

Context

The crude oil price movements are subject to diverse influencing factors. This dataset was retrieved from the U.S. Energy Information Administration: Europe Brent Spot Price FOB (Dollars per Barrel)

Content

The aim of this dataset and work is to predict future Crude Oil Prices based on the historical data available in the dataset. The data contains daily Brent oil prices from 17th of May 1987 until the 13th of November 2022.

Acknowledgements

Dataset is available on U.S. Energy Information Administration: Europe Brent Spot Price FOB (Dollars per Barrel) which is updated on weekly bases.

Inspiration

The vast competition in the Data Science field and the availability of the new Prophet method made it easier to predict future prices, that is what you may find when predicting the oil prices with this dataset.

2 ARIMA :

2.1 ARIMA (AutoRegressive Integrated Moving Average) is a popular statistical model used for time series forecasting. It combines autoregressive (AR) terms, differencing (I) to make the data stationary, and moving averages (MA). ARIMA is effective for capturing patterns in data with trends and seasonality.

2.2 Loading and Preprocessing the Data

Load the Dataset: Import the Brent oil prices dataset and parse the 'Date' column as datetime.

Set the Index: Convert the 'Date' column to a datetime format and set it as the index.

Resample Data to Monthly Frequency: Resample the data to monthly frequency, calculating the mean price for each month to reduce noise and prepare for ARIMA fitting.

[4]:
```
import pandas as pd

data=pd.read_csv('BrentOilPrices1.csv', parse_dates=['Date'])
```

```
C:\Users\SUKANNA DAS\AppData\Local\Temp\ipykernel_5488\636127598.py:3:
UserWarning: Could not infer format, so each element will be parsed
individually, falling back to `dateutil`. To ensure parsing is consistent and
as-expected, please specify a format.
  data=pd.read_csv('BrentOilPrices1.csv', parse_dates=['Date'])
```

[5]:
```
# Step 1: Convert 'Date' to datetime and set it as index
data['Date'] = pd.to_datetime(data['Date'], format='%d-%b-%y')
data.set_index('Date', inplace=True)

# Step 2: Resample to monthly frequency using mean
monthly_data = data.resample('M').mean()

# Step 3: Format the index to show only Month-Year
monthly_data.index = monthly_data.index.strftime('%b-%Y')
```

```
C:\Users\SUKANNA DAS\AppData\Local\Temp\ipykernel_5488\425692211.py:6:
FutureWarning: 'M' is deprecated and will be removed in a future version, please
use 'ME' instead.
  monthly_data = data.resample('M').mean()
```

[6]:
```
# Assuming your DataFrame is named `monthly_data`
monthly_data = monthly_data.iloc[1:-1]
```

[7]:
```
monthly_data.head(5)
```

[7]:

Date	Price
Jun-1987	18.860476
Jul-1987	19.856522
Aug-1987	18.979524
Sep-1987	18.313182
Oct-1987	18.757727

[8]:
```
monthly_data.tail(5)
```

[8]:

Date	Price
Nov-2019	63.211905
Dec-2019	67.310000
Jan-2020	63.824783
Feb-2020	55.702000
Mar-2020	32.470000

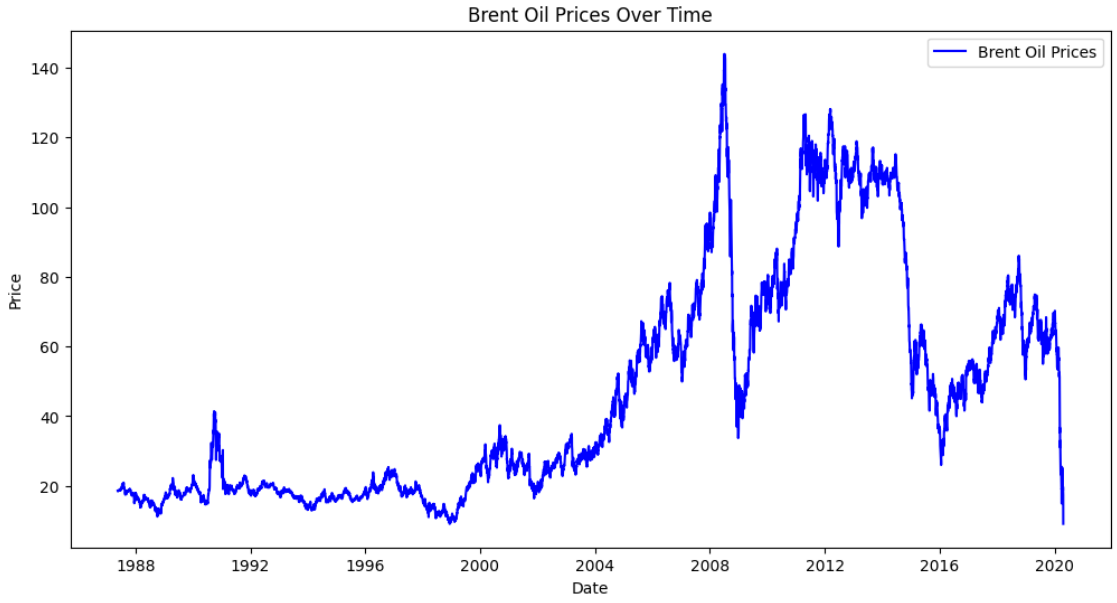
[9]:
```
monthly_data.isna().sum()
```

[9]:
```
Price    0
dtype: int64
```

[10]:
```
monthly_data.duplicated().sum()
```

[10]:
```
0
```

[11]:
```
import matplotlib.pyplot as plt

# Plot the Brent oil prices
plt.figure(figsize=(12, 6))
plt.plot(data.index, data['Price'], label='Brent Oil Prices', color='blue')
plt.title('Brent Oil Prices Over Time')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.show()
```



2.3 Time Series Decomposition

Perform seasonal decomposition using the multiplicative or additive method to break the series into:

Trend: The long-term movement in the series.

Seasonality: The repeating patterns or cycles within a fixed period (e.g., annual seasonality).

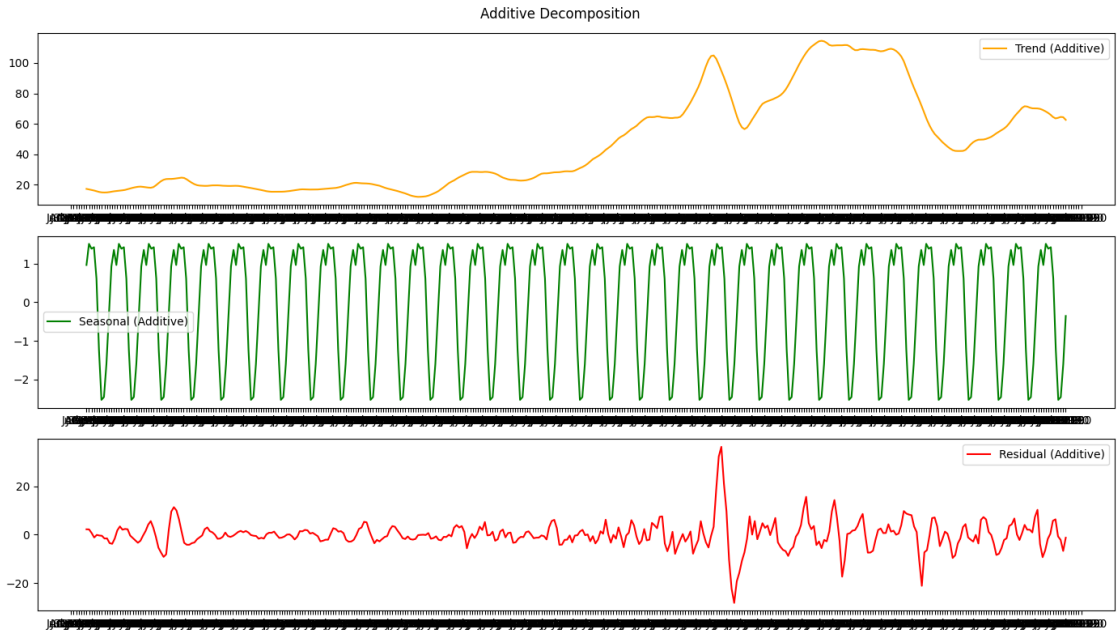
Residual: The remainder of the data after removing the trend and seasonality (random noise).

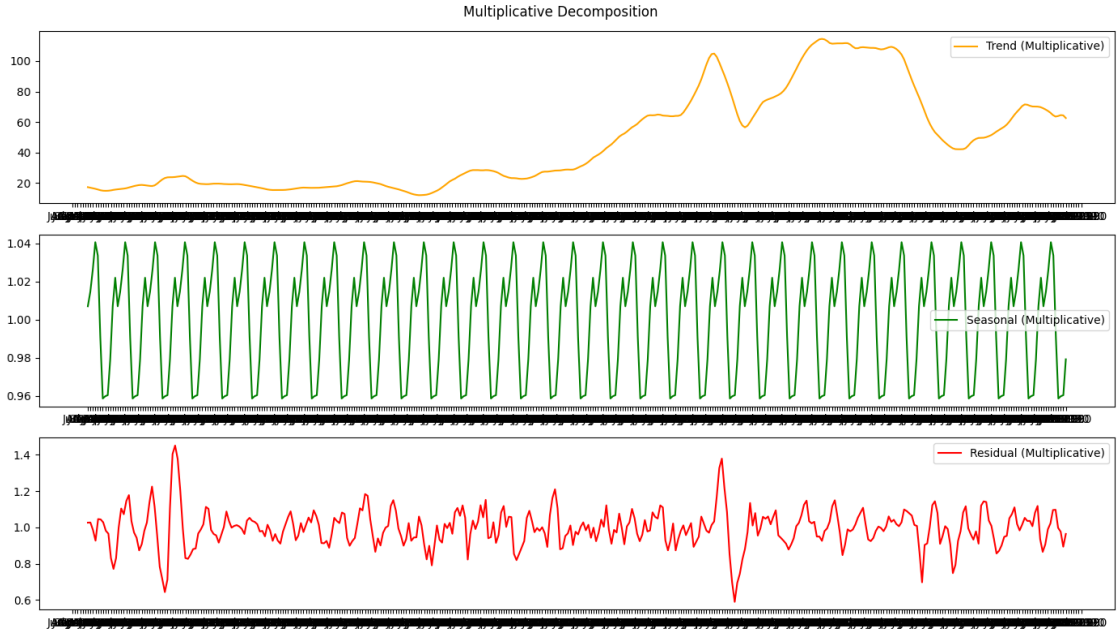
2.4 *Decomposition Interpretation:*

Both the MAE and RMSE are lower for the multiplicative model, suggesting that it provides a slightly better fit for this dataset. This might indicate that the relationship between the trend and seasonality in Brent oil prices is multiplicative in nature, meaning that seasonal fluctuations are proportional to the level of the trend. Multiplicative decomposition is more suitable for time series where seasonal effects increase as the level of the series increases, which could be the case for the Brent oil price data.

[12]:
```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose




# Additive decomposition
additive_decomposition = seasonal_decompose(monthly_data['Price'],␣
 ↪model='additive', period=12)

# Multiplicative decomposition
multiplicative_decomposition = seasonal_decompose(monthly_data['Price'],␣
 ↪model='multiplicative', period=12)

# Plot Additive Decomposition
plt.figure(figsize=(14, 8))

plt.subplot(3, 1, 1)
plt.plot(monthly_data.index, additive_decomposition.trend, label='Trend␣
 ↪(Additive)', color='orange')
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(monthly_data.index, additive_decomposition.seasonal, label='Seasonal␣
 ↪(Additive)', color='green')
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(monthly_data.index, additive_decomposition.resid, label='Residual␣
 ↪(Additive)', color='red')
plt.legend()

plt.suptitle('Additive Decomposition')
plt.tight_layout()
plt.show()
```

```
# Plot Multiplicative Decomposition
plt.figure(figsize=(14, 8))

plt.subplot(3, 1, 1)
plt.plot(monthly_data.index, multiplicative_decomposition.trend, label='Trend␣
 ↪(Multiplicative)', color='orange')
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(monthly_data.index, multiplicative_decomposition.seasonal,␣
 ↪label='Seasonal (Multiplicative)', color='green')
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(monthly_data.index, multiplicative_decomposition.resid,␣
 ↪label='Residual (Multiplicative)', color='red')
plt.legend()

plt.suptitle('Multiplicative Decomposition')
plt.tight_layout()
plt.show()
```





[13]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np

# Compute residual errors
additive_residuals = additive_decomposition.resid.dropna()
multiplicative_residuals = multiplicative_decomposition.resid.dropna()

# MAE
additive_mae = mean_absolute_error(monthly_data['Price'].loc[additive_residuals.
 ↪index], additive_residuals)
multiplicative_mae = mean_absolute_error(monthly_data['Price'].
 ↪loc[multiplicative_residuals.index], multiplicative_residuals)

print(f"Additive Model MAE: {additive_mae}")
print(f"Multiplicative Model MAE: {multiplicative_mae}")

# RMSE
additive_rmse = np.sqrt(mean_squared_error(monthly_data['Price'].
 ↪loc[additive_residuals.index], additive_residuals))
multiplicative_rmse = np.sqrt(mean_squared_error(monthly_data['Price'].
 ↪loc[multiplicative_residuals.index], multiplicative_residuals))

print(f"Additive Model RMSE: {additive_rmse}")
print(f"Multiplicative Model RMSE: {multiplicative_rmse}")
```

```
Additive Model MAE: 46.90755166966135
```

```
Multiplicative Model MAE: 45.90859384650473
Additive Model RMSE: 56.70789579414842
Multiplicative Model RMSE: 56.42679975107384
```

2.5 Time series analysis of Brent oil prices using a rolling mean and rolling standard deviation for a 12-month window.

Original Time Series (Blue Line):

This is the raw Brent oil price data. You can observe that the prices fluctuate over time, with some large spikes, particularly around the middle and later parts of the time period.

Rolling Mean (Red Line):

The rolling mean smooths out short-term fluctuations and highlights longer-term trends. It shows the general trend of the oil prices over the period, indicating periods of increase and decrease. In periods where the red line (rolling mean) is rising, prices are generally increasing, while when it is falling, prices are decreasing.

Rolling Standard Deviation (Black Line):

The rolling standard deviation indicates the variability in the data. Higher values mean that prices are fluctuating more widely from the mean during that period. For instance, in the middle of the graph where there is a large spike, the standard deviation also increases, showing high volatility during this period of price surges.

2.6 *Interpretation :*

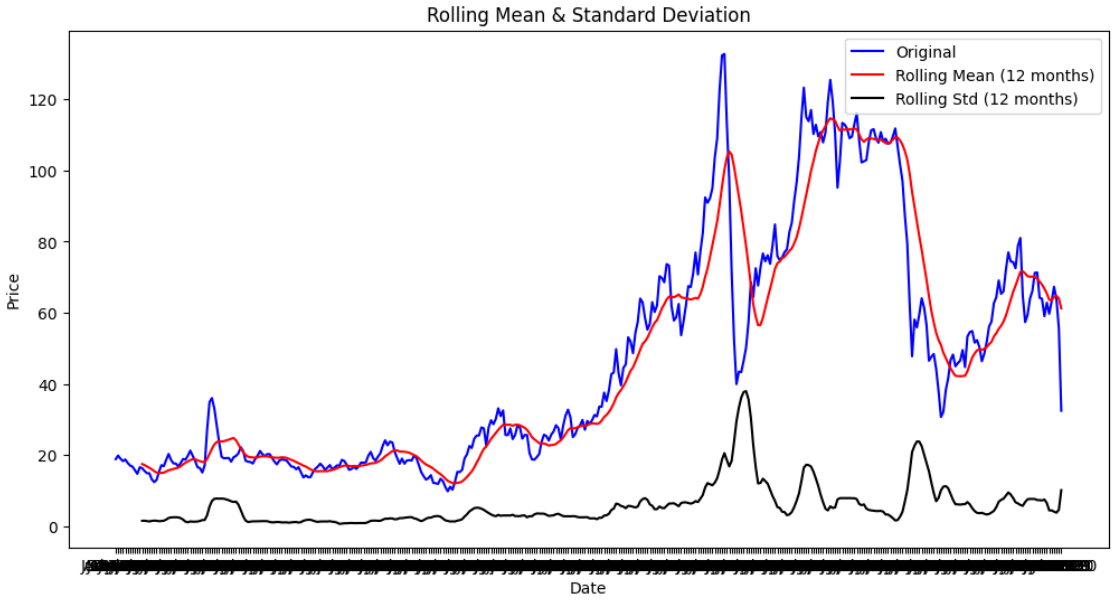
Periods of High Volatility: Around the mid-section of the graph (likely between 2006–2008 based on the general pattern of oil prices), there is a significant spike in both the price and the standard deviation, indicating a period of high volatility.

Long-Term Trend: The rolling mean shows a general trend of rising oil prices, followed by a sharp decline and some fluctuations later on.

Non-Stationarity: Since the rolling mean and standard deviation are not constant over time (they change significantly), this indicates the series is non-stationary. This means the statistical properties (like mean and variance) change over time, which is typical in financial time series data like oil prices.

[14]:
```
# Calculate rolling statistics
rolling_mean = monthly_data['Price'].rolling(window=12).mean()
rolling_std = monthly_data['Price'].rolling(window=12).std()

# Plot rolling statistics
plt.figure(figsize=(12, 6))
plt.plot(monthly_data.index,monthly_data['Price'], color='blue',␣
 ↪label='Original')
plt.plot(rolling_mean, color='red', label='Rolling Mean (12 months)')
plt.plot(rolling_std, color='black', label='Rolling Std (12 months)')
plt.title('Rolling Mean & Standard Deviation')
plt.xlabel('Date')
```

```
plt.ylabel('Price')
plt.legend(loc='best')
plt.grid(False)
plt.show()
```



2.7 Checking for Stationarity :

Apply the Augmented Dickey-Fuller (ADF) Test on the original series to confirm stationarity. If the p-value from the ADF test is below 0.05, the series is stationary, and you can proceed to fit the ARIMA model. Otherwise, differencing may be necessary.

Interpretation:

Null Hypothesis (H): The time series has a unit root, meaning it is non-stationary. Alternative Hypothesis (H): The time series does not have a unit root, meaning it is stationary.

The p-value is greater than 0.05, which means you fail to reject the null hypothesis. This suggests that the time series is non-stationary. The data does not have a constant mean and variance over time.

Since the series is non-stationary (as indicated by the p-value and the ADF statistic), you will need to apply differencing to make it stationary.

[15]:
```
from statsmodels.tsa.stattools import adfuller

# Perform ADF test
adf_result = adfuller(monthly_data['Price'])

# Extract and print results
```

```
print('ADF Statistic:', adf_result[0])
print('p-value:', adf_result[1])
print('Critical Values:')
for key, value in adf_result[4].items():
    print(f'   {key}: {value}')
```

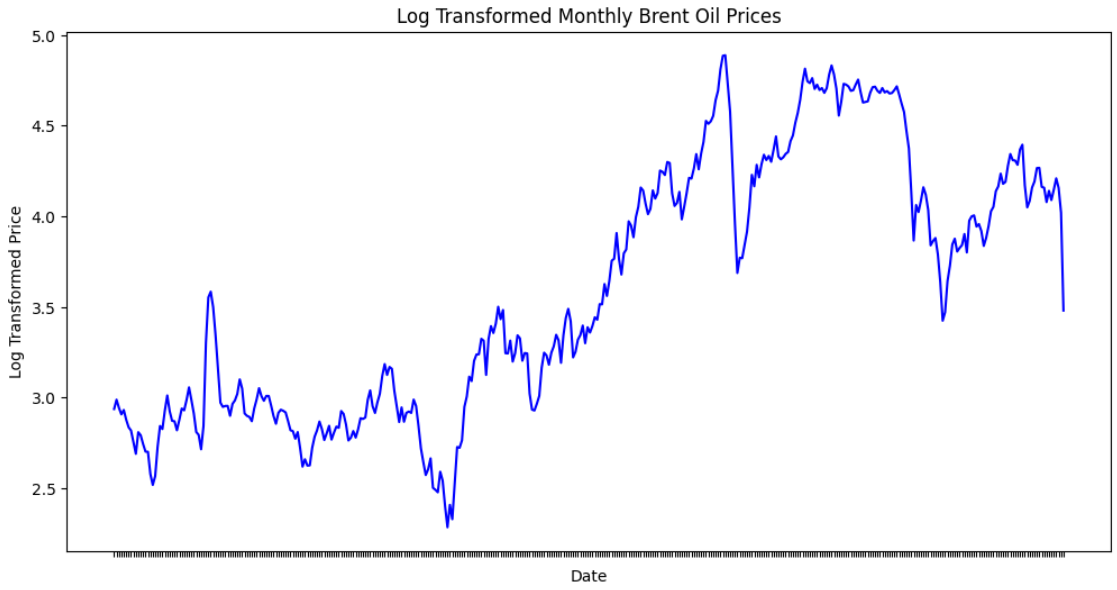
```
ADF Statistic: -2.120939367424105
p-value: 0.23617855568966656
Critical Values:
   1%: -3.44714244478345
   5%: -2.8689414326247995
   10%: -2.5707127699396084
```

2.8 Log Transformation

Apply a log transformation to stabilize the variance and ensure the data is stationary in variance.

This step is crucial for improving the model's performance when fitting ARIMA, as it helps to deal with heteroscedasticity.

[16]:
```
# Apply log transformation
monthly_data['Price_log'] = np.log(monthly_data['Price'])


plt.figure(figsize=(12, 6))
plt.plot(monthly_data.index,monthly_data['Price_log'], color='blue')
plt.title('Log Transformed Monthly Brent Oil Prices')
plt.xlabel('Date')
plt.ylabel('Log Transformed Price')
plt.gca().xaxis.set_major_formatter(plt.NullFormatter())
plt.grid(False)
plt.show()
```



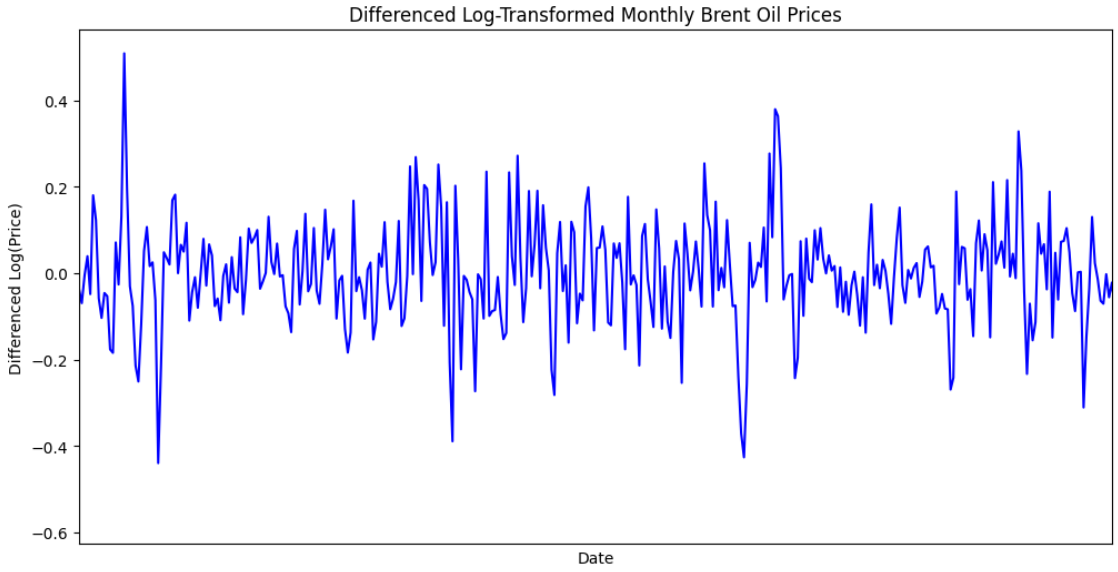
2.9 Differencing to Remove Trends

First-order differencing: Apply first-order differencing to remove any linear trends and make the series stationary in mean.

Seasonal Differencing: Perform seasonal differencing with a lag equal to 12 (for monthly data) to account for any seasonal trends in the data. Drop any resulting NaN values from differencing.

[17]:
```
# First-order differencing
monthly_data['Price_log_diff'] = monthly_data['Price_log'].diff()

# Seasonal differencing (lag=12 for monthly data)
monthly_data['Price_log_diff_seasonal'] = monthly_data['Price_log_diff'] -␣
↪monthly_data['Price_log_diff'].shift(12)

# Drop NA values
stationary_data = monthly_data['Price_log_diff_seasonal'].dropna()

# Plot differenced series
plt.figure(figsize=(12, 6))
plt.plot(stationary_data, color='blue')
plt.title('Differenced Log-Transformed Monthly Brent Oil Prices')
plt.xlabel('Date')
plt.ylabel('Differenced Log(Price)')

# Customize x-axis labels to show years with a 5-year interval
plt.gca().xaxis.set_major_formatter(plt.matplotlib.dates.DateFormatter('%Y'))
plt.gca().xaxis.set_major_locator(plt.matplotlib.dates.YearLocator(5))
```

```
# Set x-axis limits to encompass the full range of your data
plt.xlim(stationary_data.index.min(), stationary_data.index.max())

plt.grid(False)
plt.show()
```



2.10 Checking for Stationarity

Apply the Augmented Dickey-Fuller (ADF) Test on the differenced series to confirm stationarity. If the p-value from the ADF test is below 0.05, the series is stationary, and you can proceed to fit the ARIMA model. Otherwise, further differencing may be necessary.

Interpretation :

The time series has become stationary after applying the differencing transformation. We can now proceed to fit the ARIMA model on the differenced, stationary data.

[18]:
```
# Apply the ADF test
adf_result = adfuller(stationary_data)

# Print the results
print('ADF Statistic:', adf_result[0])
print('p-value:', adf_result[1])
print('Critical Values:')
for key, value in adf_result[4].items():
    print(f'    {key}: {value}')
```

```
ADF Statistic: -8.690836465434414
p-value: 4.0448881740536107e-14
```

```
Critical Values:
    1%: -3.448196541708585
    5%: -2.869404683789669
    10%: -2.5709597356805545
```

2.11 Plot ACF and PACF

Plot the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) to determine the appropriate orders of p (autoregressive) and q (moving average) terms. Interpretation: Use the PACF plot to identify the order of the AR component (p) based on where the PACF cuts off. Use the ACF plot to identify the order of the MA component (q) based on where the ACF cuts off.

Interpretation :

From the ACF and PACF plots, we can estimate the parameters for the ARIMA model:

Autocorrelation Function (ACF):

The ACF plot shows a significant spike at lag 1 and then quickly dies off, with minor fluctuations beyond that. This indicates that the moving average (MA) part of the model might be MA(1) because the significant spike at lag 1 suggests a possible order of q = 1.

Partial Autocorrelation Function (PACF):

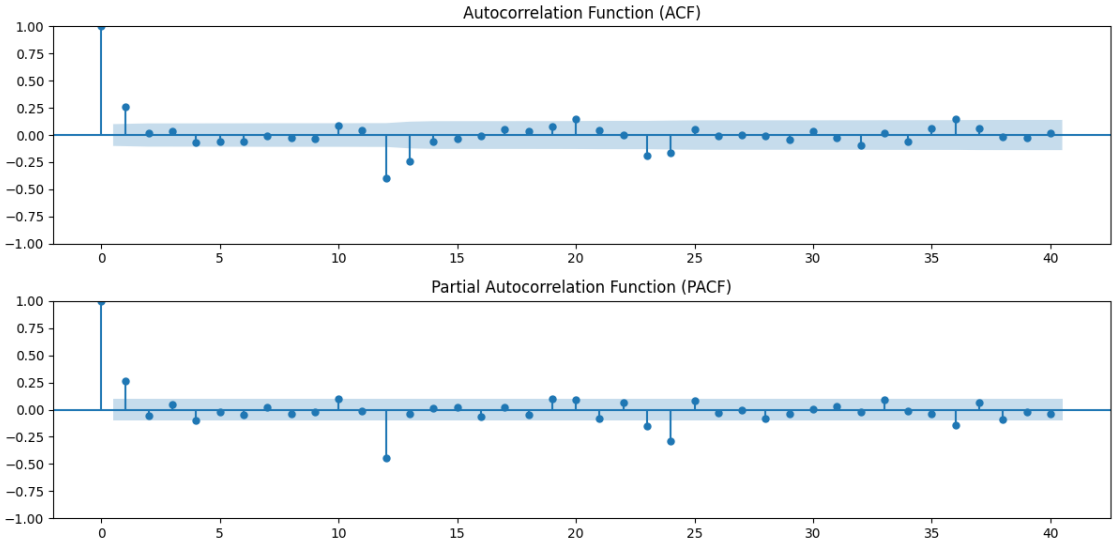
The PACF plot has a significant spike at lag 1 and shows a gradual decline after that. This indicates that the autoregressive (AR) part of the model might be AR(1), as the gradual decay beyond lag 1 suggests an order of p = 1.

Differencing (d):

Since the data was differenced to make it stationary (based on the ADF test), the order of differencing d = 1.

Suggested ARIMA model: ARIMA(1, 1, 1) (p=1, d=1, q=1)

[19]:
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

plt.figure(figsize=(12,6))

# Plot ACF
plt.subplot(211)
plot_acf(stationary_data, lags=40, ax=plt.gca())
plt.title('Autocorrelation Function (ACF)')

# Plot PACF
plt.subplot(212)
plot_pacf(stationary_data, lags=40, ax=plt.gca(), method='ywm')
plt.title('Partial Autocorrelation Function (PACF)')

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



2.12 Fit the ARIMA Model

Use the identified (p, d, q) values to fit the ARIMA model using the ARIMA function from the statsmodels library.

Fit the model to the transformed and differenced dataset.

2.13 Model Diagnostics

Check the residuals of the model to ensure they behave like white noise (i.e., no autocorrelation, normally distributed).

Use ACF/PACF of residuals to confirm that no significant patterns remain in the residuals.

Perform additional diagnostic tests like the Ljung-Box test to confirm that the residuals are independent.

2.14 Model Evaluation:

Evaluate the model's performance using metrics like:

AIC (Akaike Information Criterion) or BIC (Bayesian Information Criterion) to choose the best-fitting model.

Refit the model with slightly different parameters if necessary and choose the model with the lowest AIC/BIC.

2.15 *Interpretation:*

The SARIMAX model summary indicates that the best model fit is **ARIMA(2,1,1)(0,0,2)[12]**. Here's how we can interpret the output:

2.15.1 Model Summary:

- **ARIMA(2,1,1)**:
  - **AR (2)**: Two autoregressive (AR) terms are included, which means the model uses the past two observations to predict future values.
  - **I (1)**: First-order differencing is applied to make the time series stationary.
  - **MA (1)**: The model includes one moving average (MA) term, meaning it uses one lagged error term in prediction.
- **Seasonal (0,0,2)[12]**:
  - No seasonal autoregressive or differencing terms (indicated by the zeroes).
  - **MA (2)**: Two seasonal moving average terms are included to account for the seasonality with a period of 12 (monthly data).

2.15.2 Key Coefficients:

- **AR L1 (1.1803)**: The coefficient of the first autoregressive term is significant, with a positive value close to 1. This indicates that the series is highly dependent on its most recent past value.
- **AR L2 (-0.3028)**: The second AR term has a negative coefficient, showing a lesser but still significant influence of the second lag on future values.
- **MA L1 (-0.8973)**: The first moving average term has a large negative coefficient, meaning that past forecast errors influence the current forecast.
- **Seasonal MA L12 (0.0621)**: The first seasonal moving average term is not significant (p-value = 0.299), suggesting that the seasonality at lag 12 does not strongly affect the model.
- **Seasonal MA L24 (-0.1207)**: The second seasonal moving average term is significant (p-value = 0.044), implying that the seasonality at lag 24 (2 years back) plays a role.
- **sigma2 (0.0077)**: The variance of the residuals is small, which is desirable, indicating that the model fits the data well.

2.15.3 Model Fit and Statistics:

- **Log Likelihood (397.473)**: Higher values of log-likelihood suggest a better fit.
- **AIC (-782.947)**: The Akaike Information Criterion (AIC) is used for model selection. A lower AIC value indicates a better fit.
- **BIC (-759.104)**: The Bayesian Information Criterion (BIC) penalizes more complex models. Here, BIC is also low, but slightly higher than AIC.
- **HQIC (-773.498)**: This is another criterion used to evaluate model fit, again lower values are better.

2.15.4 Diagnostics:

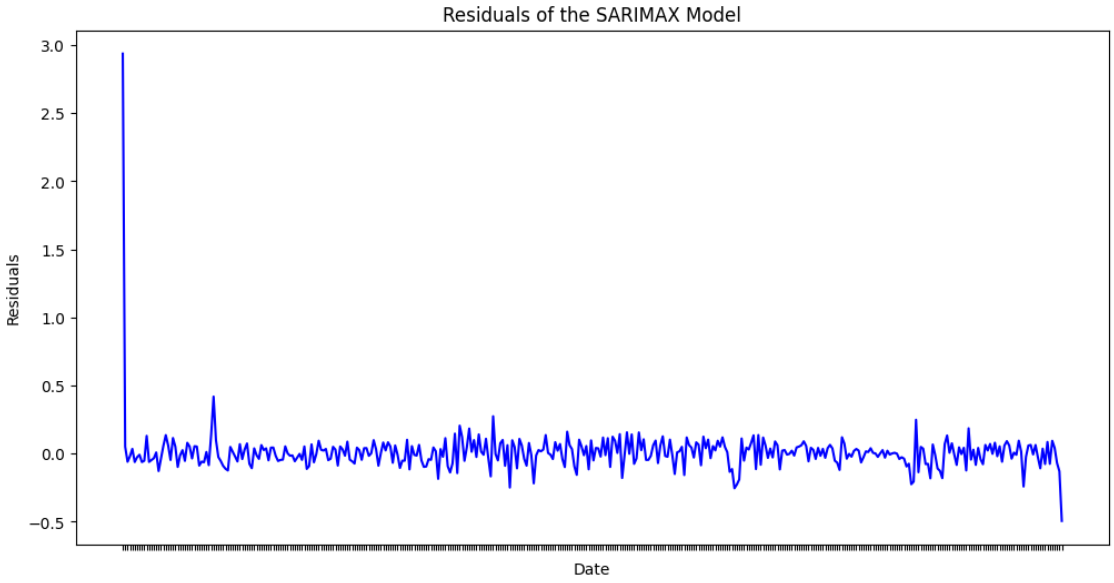
- **Ljung-Box (L1) Q = 0.01**: The Ljung-Box test for autocorrelation in the residuals shows a very small value (Prob(Q) = 0.94), indicating no significant autocorrelation left in the residuals, which is a good sign of a well-fitted model.
- **Jarque-Bera Test**: The JB statistic is 209.64 with a very low p-value, meaning the residuals are not normally distributed. This could be due to skewness (-0.40) or high kurtosis (6.49), suggesting the presence of extreme values or outliers.

[21]:
```
import pmdarima as pm

# Fit auto_arima
auto_model = pm.auto_arima(monthly_data['Price_log'],
                           seasonal=True,
                           m=12,
                           trace=True,
                           error_action='ignore',
                           suppress_warnings=True,
                           stepwise=True)

print(auto_model.summary())
```

```
Performing stepwise search to minimize aic
 ARIMA(2,1,2)(1,0,1)[12] intercept   : AIC=-778.104, Time=6.86 sec
 ARIMA(0,1,0)(0,0,0)[12] intercept   : AIC=-751.024, Time=0.06 sec
 ARIMA(1,1,0)(1,0,0)[12] intercept   : AIC=-777.251, Time=1.11 sec
 ARIMA(0,1,1)(0,0,1)[12] intercept   : AIC=-777.656, Time=1.21 sec
 ARIMA(0,1,0)(0,0,0)[12]             : AIC=-752.936, Time=0.14 sec
 ARIMA(2,1,2)(0,0,1)[12] intercept   : AIC=-776.394, Time=2.86 sec
 ARIMA(2,1,2)(1,0,0)[12] intercept   : AIC=-776.362, Time=1.76 sec
 ARIMA(2,1,2)(2,0,1)[12] intercept   : AIC=-778.754, Time=5.62 sec
 ARIMA(2,1,2)(2,0,0)[12] intercept   : AIC=-778.024, Time=4.65 sec
 ARIMA(2,1,2)(2,0,2)[12] intercept   : AIC=-776.607, Time=5.74 sec
 ARIMA(2,1,2)(1,0,2)[12] intercept   : AIC=-778.516, Time=4.24 sec
 ARIMA(1,1,2)(2,0,1)[12] intercept   : AIC=-776.936, Time=3.90 sec
 ARIMA(2,1,1)(2,0,1)[12] intercept   : AIC=-779.350, Time=4.87 sec
 ARIMA(2,1,1)(1,0,1)[12] intercept   : AIC=-779.427, Time=2.64 sec
 ARIMA(2,1,1)(0,0,1)[12] intercept   : AIC=-778.630, Time=1.92 sec
 ARIMA(2,1,1)(1,0,0)[12] intercept   : AIC=-778.474, Time=2.96 sec
 ARIMA(2,1,1)(1,0,2)[12] intercept   : AIC=-779.578, Time=4.90 sec
 ARIMA(2,1,1)(0,0,2)[12] intercept   : AIC=-781.205, Time=4.40 sec
 ARIMA(1,1,1)(0,0,2)[12] intercept   : AIC=-778.607, Time=2.54 sec
 ARIMA(2,1,0)(0,0,2)[12] intercept   : AIC=-778.633, Time=2.59 sec
 ARIMA(3,1,1)(0,0,2)[12] intercept   : AIC=-778.940, Time=6.16 sec
 ARIMA(2,1,2)(0,0,2)[12] intercept   : AIC=-778.290, Time=5.52 sec
 ARIMA(1,1,0)(0,0,2)[12] intercept   : AIC=-779.955, Time=1.33 sec
 ARIMA(1,1,2)(0,0,2)[12] intercept   : AIC=-777.609, Time=4.44 sec
 ARIMA(3,1,0)(0,0,2)[12] intercept   : AIC=-776.752, Time=2.46 sec
 ARIMA(3,1,2)(0,0,2)[12] intercept   : AIC=-777.555, Time=5.55 sec
 ARIMA(2,1,1)(0,0,2)[12]             : AIC=-782.947, Time=2.44 sec
 ARIMA(2,1,1)(0,0,1)[12]             : AIC=-780.324, Time=0.96 sec
 ARIMA(2,1,1)(1,0,2)[12]             : AIC=-782.242, Time=3.82 sec
 ARIMA(2,1,1)(1,0,1)[12]             : AIC=-781.372, Time=1.80 sec
 ARIMA(1,1,1)(0,0,2)[12]             : AIC=-780.568, Time=1.15 sec
 ARIMA(2,1,0)(0,0,2)[12]             : AIC=-780.593, Time=0.85 sec
 ARIMA(3,1,1)(0,0,2)[12]             : AIC=-780.961, Time=3.09 sec
```

```
 ARIMA(2,1,2)(0,0,2)[12]             : AIC=-781.082, Time=2.24 sec
 ARIMA(1,1,0)(0,0,2)[12]             : AIC=-781.924, Time=0.60 sec
 ARIMA(1,1,2)(0,0,2)[12]             : AIC=-780.432, Time=2.41 sec
 ARIMA(3,1,0)(0,0,2)[12]             : AIC=-778.706, Time=1.37 sec
 ARIMA(3,1,2)(0,0,2)[12]             : AIC=-779.486, Time=3.26 sec

Best model:  ARIMA(2,1,1)(0,0,2)[12]
Total fit time: 114.531 seconds
                                      SARIMAX Results
==========================================================================================
Dep. Variable:                                         y   No. Observations:
394
Model:             SARIMAX(2, 1, 1)x(0, 0, [1, 2], 12)   Log Likelihood
397.473
Date:                                 Tue, 15 Oct 2024   AIC
-782.947
Time:                                         08:46:55   BIC
-759.104
Sample:                                     06-01-1987   HQIC
-773.498
                                          - 03-01-2020
Covariance Type:                                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          1.1803      0.083     14.159      0.000       1.017       1.344
ar.L2         -0.3028      0.040     -7.534      0.000      -0.382      -0.224
ma.L1         -0.8973      0.087    -10.298      0.000      -1.068      -0.727
ma.S.L12       0.0621      0.060      1.038      0.299      -0.055       0.180
ma.S.L24      -0.1207      0.060     -2.010      0.044      -0.238      -0.003
sigma2         0.0077      0.000     21.139      0.000       0.007       0.008
==============================================================================
===
Ljung-Box (L1) (Q):                   0.01   Jarque-Bera (JB):
209.64
Prob(Q):                              0.94   Prob(JB):
0.00
Heteroskedasticity (H):               1.55   Skew:
-0.40
Prob(H) (two-sided):                  0.01   Kurtosis:
6.49
==============================================================================
===

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

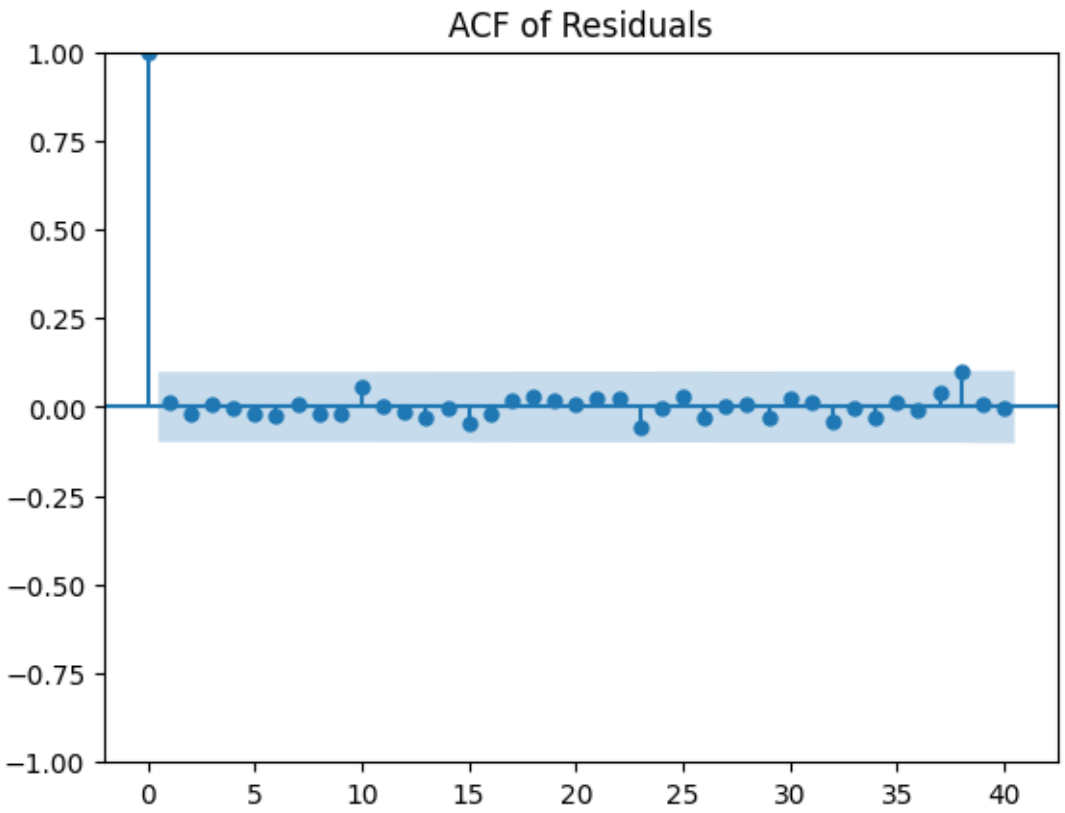
[22]:
```
# Get residuals
residuals = auto_model.resid()

# Plot residuals
plt.figure(figsize=(12, 6))
plt.plot(residuals, color='blue')
plt.title('Residuals of the SARIMAX Model')
plt.xlabel('Date')
plt.ylabel('Residuals')
plt.gca().xaxis.set_major_formatter(plt.NullFormatter())
plt.grid(False)
plt.show()
```



[23]:
```
# Calculate Z-scores for residuals
from scipy.stats import zscore

residuals_zscore = zscore(residuals)

# Identify outliers
outliers = residuals[np.abs(residuals_zscore) > 3]

print("Outliers in Residuals:")
print(outliers)
```

```
Outliers in Residuals:
Date
Jun-1987    2.937069
dtype: float64
```

[24]:
```
# ACF of residuals
plot_acf(residuals, lags=40)
plt.title('ACF of Residuals')
plt.show()
```



2.16 Forecasting:

Once the best model is selected, use it to forecast future values of the Brent oil prices. Plot the forecasted values along with the actual values to visualize the model's performance.

2.17 *Interpretation :*

The forecast indicates a steady upward trend from April 2020 to January 2021, with values increasing each month.

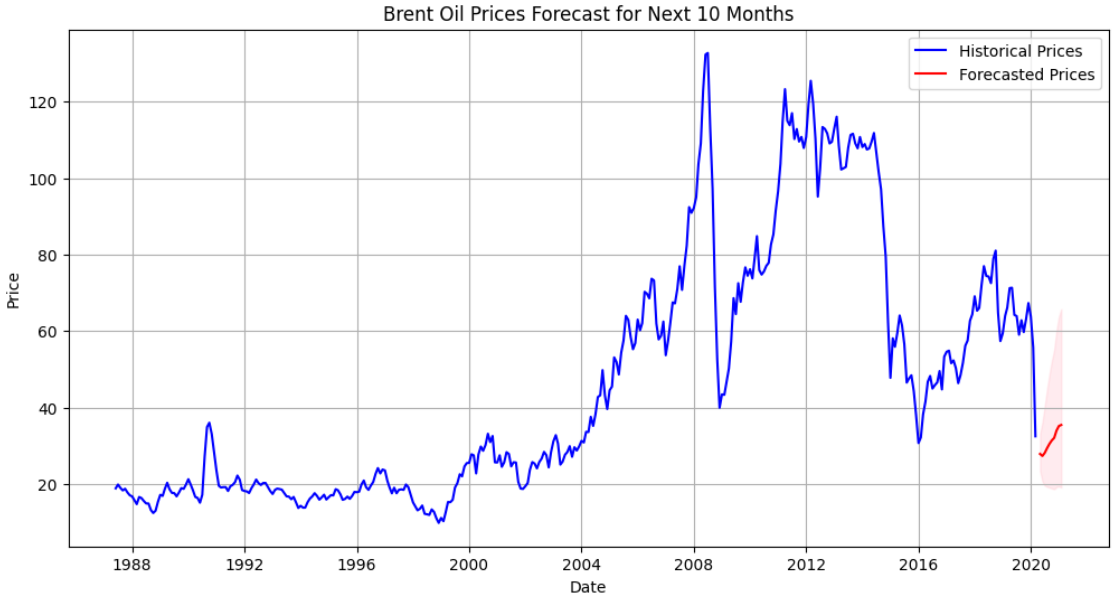
There is a slight dip between April and May, but the trend recovers and continues growing afterward.

The model likely reflects the underlying seasonal patterns captured in the ARIMA model, where seasonal components may contribute to periodic increases, such as the larger jumps in November and December 2020.

[25]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pmdarima as pm

# Ensure your index is a datetime index
monthly_data.index = pd.to_datetime(monthly_data.index)

# Define the number of periods to forecast (10 months)
n_steps = 10

# Forecast the next 10 months
forecast, conf_int = auto_model.predict(n_periods=n_steps, return_conf_int=True)

# Create forecast index for the next 10 months
last_date = monthly_data.index[-1]  # Get the last date of the complete dataset
forecast_index = pd.date_range(start=last_date + pd.DateOffset(months=1),
                               periods=n_steps,
                               freq='M')

# Convert forecasted log prices back to original scale (if you applied log␣
 ↪transformation)
forecast_prices = np.exp(forecast)  # Inverse the log transformation
conf_int_prices = np.exp(conf_int)  # Inverse the log transformation

# Plot
plt.figure(figsize=(12, 6))
plt.plot(monthly_data.index, monthly_data['Price'], label='Historical Prices',␣
 ↪color='blue')
plt.plot(forecast_index, forecast_prices, label='Forecasted Prices',␣
 ↪color='red')
plt.fill_between(forecast_index,
                 conf_int_prices[:, 0],
                 conf_int_prices[:, 1],
                 color='pink', alpha=0.3)
plt.title('Brent Oil Prices Forecast for Next 10 Months')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.show()
```

```
C:\Users\SUKANNA DAS\AppData\Local\Temp\ipykernel_5488\305801821.py:7:
UserWarning: Could not infer format, so each element will be parsed
individually, falling back to `dateutil`. To ensure parsing is consistent and
as-expected, please specify a format.
  monthly_data.index = pd.to_datetime(monthly_data.index)
```

```
C:\Users\SUKANNA DAS\AppData\Local\Temp\ipykernel_5488\305801821.py:17:
FutureWarning: 'M' is deprecated and will be removed in a future version, please
use 'ME' instead.
  forecast_index = pd.date_range(start=last_date + pd.DateOffset(months=1),
```



[26]:
```
fitted_values = auto_model.predict_in_sample()

# Calculate MAE and RMSE
mae = mean_absolute_error(monthly_data['Price'], np.exp(fitted_values))  #␣
 ↪Inverse log for comparison
rmse = np.sqrt(mean_squared_error(monthly_data['Price'], np.
 ↪exp(fitted_values)))  # Inverse log for comparison

# Print MAE and RMSE
print(f"Mean Absolute Error (MAE): {mae}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
```

```
Mean Absolute Error (MAE): 2.9741793492732493
Root Mean Squared Error (RMSE): 4.426013802720291
```

[27]:
```
forecast_prices
```

[27]:
```
2020-04-01    27.867559
2020-05-01    27.327398
2020-06-01    28.131114
2020-07-01    29.383190
2020-08-01    30.510137
```

```
2020-09-01    31.389567
2020-10-01    32.061025
2020-11-01    33.948707
2020-12-01    35.162472
2021-01-01    35.438983
Freq: MS, dtype: float64
```

2.18 Conclusion :

The analysis conducted through the Augmented Dickey-Fuller (ADF) test, autocorrelation function (ACF), and the fitting of the ARIMA model provides a comprehensive insight into the behavior of the time series data from 1987 to early 2020.

Stationarity Assessment: Initially, the ADF test indicated non-stationarity in the original series, leading to the necessity for differencing. However, subsequent differencing resulted in a stationary series, confirming the stability needed for effective time series modeling.

Model Selection: The selected model, ARIMA(2,1,1)(0,0,2)[12], was chosen based on optimal performance metrics, including the lowest Akaike Information Criterion (AIC) value. The significant coefficients indicate that the model effectively captures the underlying data patterns, including both autoregressive and moving average components, along with seasonal influences.

Forecasting Results: The forecasted values from April 2020 to January 2021 reveal a clear upward trend, reflecting an anticipated increase in the underlying metric (such as sales, consumption, or temperature) over this period. The gradual increase, despite a minor dip in May, demonstrates resilience in the series, suggesting that any potential fluctuations are likely to be temporary.

Practical Implications: The forecasting outcomes can provide valuable insights for strategic planning and decision-making. For instance, businesses can leverage this growth trend for inventory management, resource allocation, or marketing strategies, while policymakers can use the data to anticipate future needs and challenges.

exponential

October 15, 2024

[15]:
```
import numpy as np
import pandas as pd
```

0.1 Holt-Winters Exponential Smoothing

0.2 It is a time series forecasting method that extends simple exponential smoothing to capture trends and seasonality. It consists of three components: level, trend, and seasonal smoothing. It's commonly used for seasonal data to predict future values by adjusting for both trend and seasonal variations.

[16]:
```
data = pd.read_csv("BrentOilPrices1.csv")
data
```

[16]:

	Date	Price
0	20-May-87	18.63
1	21-May-87	18.45
2	22-May-87	18.55
3	25-May-87	18.60
4	26-May-87	18.63
…	…	…
8355	15-Apr-20	19.80
8356	16-Apr-20	18.69
8357	17-Apr-20	19.75
8358	20-Apr-20	17.36
8359	21-Apr-20	9.12

[8360 rows x 2 columns]

1 Data Preprocessing for Monthly Analysis

This code snippet outlines the steps to convert a dataset containing dates and corresponding values into a monthly aggregated format. The process involves several key steps, including date conversion, resampling, and formatting. Below is a breakdown of each step:

1.1 Step 1: Convert 'Date' to Datetime and Set as Index

1.2 Step 2: Resample to Monthly Frequency Using Mean

1.3 Step 3: Format the Index to Show Only Month-Year

[17]:
```
data['Date'] = pd.to_datetime(data['Date'], format='%d-%b-%y')
data.set_index('Date', inplace=True)


monthly_data = data.resample('M').mean()


monthly_data.index = monthly_data.index.strftime('%b-%Y')

print(monthly_data)
```

```
              Price
Date
May-1987  18.580000
Jun-1987  18.860476
Jul-1987  19.856522
Aug-1987  18.979524
Sep-1987  18.313182
…              …
Dec-2019  67.310000
Jan-2020  63.824783
Feb-2020  55.702000
Mar-2020  32.470000
Apr-2020  19.702308

[396 rows x 1 columns]
```

```
C:\Users\SUKANNA DAS\AppData\Local\Temp\ipykernel_16480\2295496914.py:5:
FutureWarning: 'M' is deprecated and will be removed in a future version, please
use 'ME' instead.
  monthly_data = data.resample('M').mean()
```

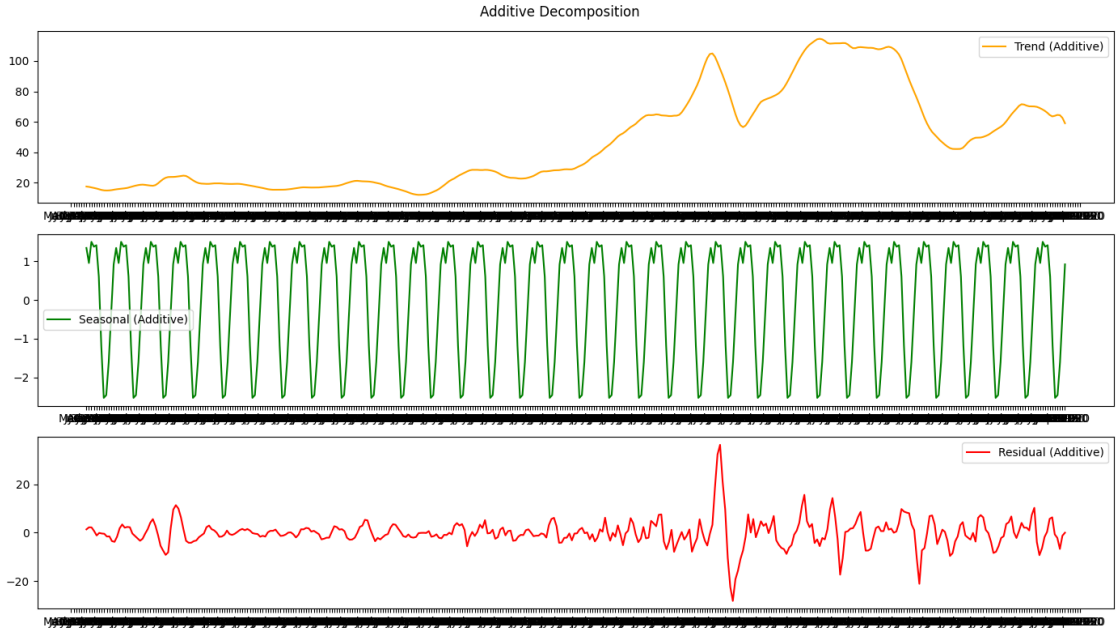
2 Seasonal Decomposition of Time Series Data

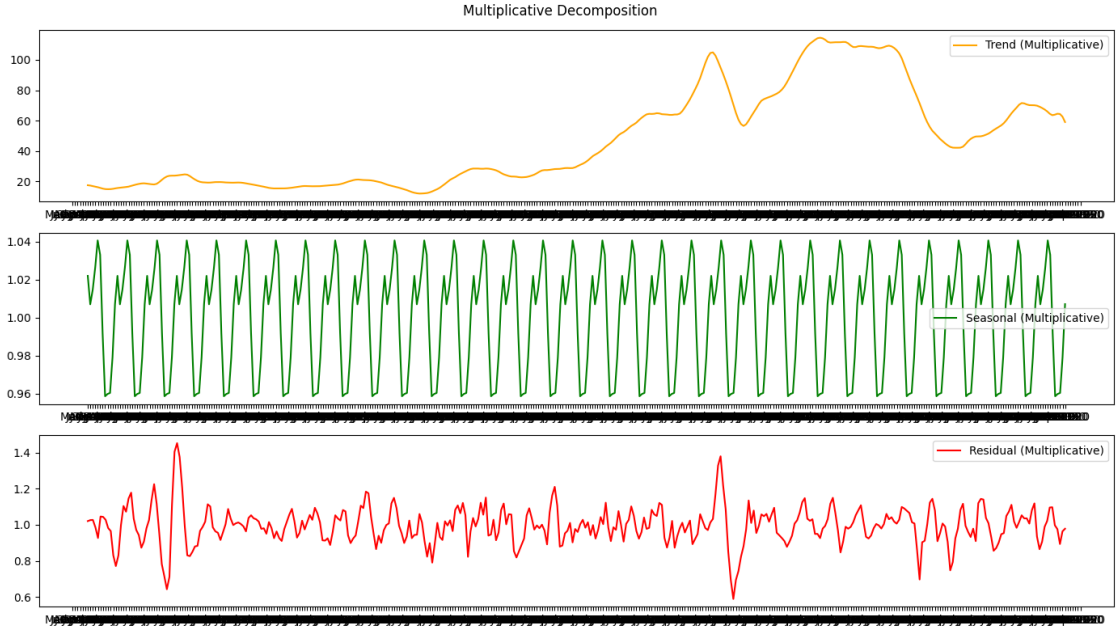
This code snippet demonstrates how to perform seasonal decomposition on a time series dataset using both additive and multiplicative models. The results are visualized to show the trend, seasonal, and residual components for each model.

[18]:
```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
```

```
# Additive decomposition
additive_decomposition = seasonal_decompose(monthly_data['Price'],␣
 ↪model='additive', period=12)

# Multiplicative decomposition
multiplicative_decomposition = seasonal_decompose(monthly_data['Price'],␣
 ↪model='multiplicative', period=12)

# Plot Additive Decomposition
plt.figure(figsize=(14, 8))

plt.subplot(3, 1, 1)
plt.plot(monthly_data.index, additive_decomposition.trend, label='Trend␣
 ↪(Additive)', color='orange')
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(monthly_data.index, additive_decomposition.seasonal, label='Seasonal␣
 ↪(Additive)', color='green')
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(monthly_data.index, additive_decomposition.resid, label='Residual␣
 ↪(Additive)', color='red')
plt.legend()

plt.suptitle('Additive Decomposition')
plt.tight_layout()
plt.show()

# Plot Multiplicative Decomposition
plt.figure(figsize=(14, 8))

plt.subplot(3, 1, 1)
plt.plot(monthly_data.index, multiplicative_decomposition.trend, label='Trend␣
 ↪(Multiplicative)', color='orange')
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(monthly_data.index, multiplicative_decomposition.seasonal,␣
 ↪label='Seasonal (Multiplicative)', color='green')
plt.legend()

plt.subplot(3, 1, 3)
```

```
plt.plot(monthly_data.index, multiplicative_decomposition.resid,␣
  ↪label='Residual (Multiplicative)', color='red')
plt.legend()

plt.suptitle('Multiplicative Decomposition')
plt.tight_layout()
plt.show()
```





[19]:
```
monthly_data.isnull().sum()
```

[19]:
```
Price    0
dtype: int64
```

[20]:
```
monthly_data
```

[20]:
```
              Price
Date
May-1987  18.580000
Jun-1987  18.860476
Jul-1987  19.856522
Aug-1987  18.979524
Sep-1987  18.313182
…              …
Dec-2019  67.310000
Jan-2020  63.824783
Feb-2020  55.702000
Mar-2020  32.470000
Apr-2020  19.702308

[396 rows x 1 columns]
```

[21]:
```
# Find rows with null values
rows_with_nulls = monthly_data[monthly_data.isnull().any(axis=1)]
print(rows_with_nulls)
```

```
Empty DataFrame
Columns: [Price]
Index: []
```

[ ]:
```

```

3 Error Metrics Calculation for Seasonal Decomposition

This code snippet demonstrates how to compute the Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) for both additive and multiplicative seasonal decomposition models of a time series dataset. These metrics are essential for evaluating the accuracy of the models.

3.1 Seeing the RMSE and MAE , we are selecting the model .

[22]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error
import numpy as np
```

```
# Compute residual errors
additive_residuals = additive_decomposition.resid.dropna()
multiplicative_residuals = multiplicative_decomposition.resid.dropna()

# MAE
additive_mae = mean_absolute_error(monthly_data['Price'].loc[additive_residuals.
 ↪index], additive_residuals)
multiplicative_mae = mean_absolute_error(monthly_data['Price'].
 ↪loc[multiplicative_residuals.index], multiplicative_residuals)

print(f"Additive Model MAE: {additive_mae}")
print(f"Multiplicative Model MAE: {multiplicative_mae}")

# RMSE
additive_rmse = np.sqrt(mean_squared_error(monthly_data['Price'].
 ↪loc[additive_residuals.index], additive_residuals))
multiplicative_rmse = np.sqrt(mean_squared_error(monthly_data['Price'].
 ↪loc[multiplicative_residuals.index], multiplicative_residuals))

print(f"Additive Model RMSE: {additive_rmse}")
print(f"Multiplicative Model RMSE: {multiplicative_rmse}")
```

```
Additive Model MAE: 46.86126861840189
Multiplicative Model MAE: 45.86609026000232
Additive Model RMSE: 56.64802709624784
Multiplicative Model RMSE: 56.36590943980757
```

3.2 here We choose Multiplicative model because of it's low MAE and low RMSE

4 Data Filtering: Removing Year 1987 from Monthly Data

This code snippet demonstrates how to filter a time series dataset by removing all data points corresponding to the year 1987.

[23]:
```
monthly_data.index = pd.to_datetime(monthly_data.index, format='%b-%Y')

# Remove all data points for the year 1987
monthly_data_filtered = monthly_data[monthly_data.index.year != 1987]

# Optionally, reset the index to the original format (Month-Year)
monthly_data_filtered.index = monthly_data_filtered.index.strftime('%b-%Y')

# Display the filtered data
print(monthly_data_filtered)
```

```
             Price
Date
```

```
Jan-1988  16.749444
Feb-1988  15.729524
Mar-1988  14.731304
Apr-1988  16.595263
May-1988  16.314091
…              …
Dec-2019  67.310000
Jan-2020  63.824783
Feb-2020  55.702000
Mar-2020  32.470000
Apr-2020  19.702308

[388 rows x 1 columns]
```

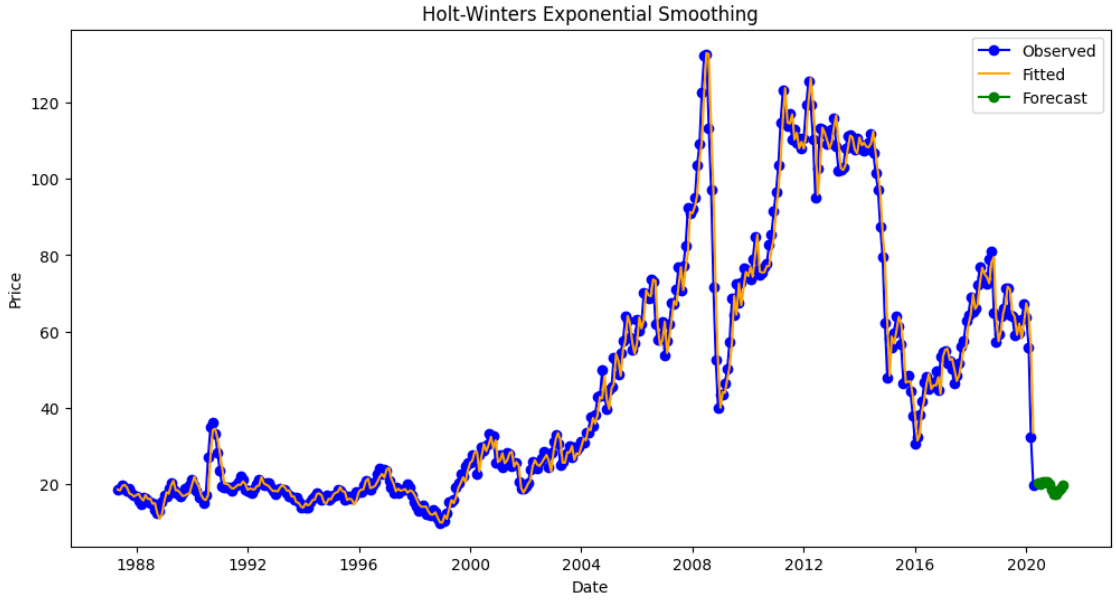
5 Holt-Winters Exponential Smoothing Forecasting

This code snippet demonstrates how to use the Holt-Winters Exponential Smoothing method for forecasting future values based on a time series dataset. The process involves fitting the model, making predictions, and visualizing the results.

[24]:
```
import pandas as pd
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt

model = sm.tsa.ExponentialSmoothing(monthly_data['Price'], seasonal='add',␣
 ↪seasonal_periods=12)
holt_winters_fit = model.fit()

forecast = holt_winters_fit.forecast(steps=12)

forecast_index = pd.date_range(start=monthly_data.index[-1] + pd.offsets.
 ↪MonthBegin(), periods=12, freq='M')
forecast_df = pd.DataFrame(data=forecast.values, index=forecast_index,␣
 ↪columns=['Forecasted Price'])

plt.figure(figsize=(12, 6))
plt.plot(monthly_data['Price'], label='Observed', marker='o', color='blue')
plt.plot(holt_winters_fit.fittedvalues, label='Fitted', color='orange')
plt.plot(forecast_df['Forecasted Price'], label='Forecast', color='green',␣
 ↪marker='o')
plt.title('Holt-Winters Exponential Smoothing')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.show()
```

```
print("Forecasted Values:")
print(forecast_df)
```

```
C:\Users\SUKANNA DAS\AppData\Roaming\Python\Python312\site-
packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarning: No frequency
information was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
C:\Users\SUKANNA DAS\AppData\Local\Temp\ipykernel_16480\3572402039.py:11:
FutureWarning: 'M' is deprecated and will be removed in a future version, please
use 'ME' instead.
  forecast_index = pd.date_range(start=monthly_data.index[-1] +
pd.offsets.MonthBegin(), periods=12, freq='M')
```




```
Forecasted Values:
            Forecasted Price
2020-05-31         20.233860
2020-06-30         19.982452
2020-07-31         20.661849
2020-08-31         20.619566
2020-09-30         20.703954
2020-10-31         19.954532
2020-11-30         18.441680
2020-12-31         17.379459
2021-01-31         17.452227
2021-02-28         18.137859
2021-03-31         18.741191
2021-04-30         19.702308
```

[30]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Step 4: Calculate MAE and RMSE for fitted values (train)
mae = mean_absolute_error(monthly_data['Price'], holt_winters_fit.fittedvalues)
rmse = np.sqrt(mean_squared_error(monthly_data['Price'], holt_winters_fit.
 ↪fittedvalues))

# Print the MAE and RMSE
print(f"Mean Absolute Error (MAE): {mae}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
```

```
Mean Absolute Error (MAE): 3.0161121799205595
Root Mean Squared Error (RMSE): 4.638151637499927
```

5.1 The forecasted prices from May 2020 to April 2021 show fluctuations, peaking at approximately 20.66 in July and dropping to 17.38 by December. Despite a slight recovery to 18.74 in March 2021, the trend indicates market volatility, necessitating continuous monitoring of external factors influencing prices.

lstm

October 15, 2024

0.1 Long Short-Term Memory (LSTM):

0.1.1 It is a type of recurrent neural network (RNN) designed to handle sequential data and capture long-term dependencies. LSTMs are equipped with memory cells and gating mechanisms that regulate the flow of information, allowing them to retain or forget information over time. This architecture addresses the vanishing gradient problem commonly faced by standard RNNs. LSTMs are widely used in various applications, including natural language processing, time series forecasting, and speech recognition, due to their ability to learn patterns and dependencies in complex sequences, making them effective for tasks requiring context and temporal dynamics.

[2]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from datetime import datetime
from pylab import rcParams
import matplotlib.pyplot as plt
import warnings
import itertools
import statsmodels.api as sm
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from sklearn.metrics import mean_squared_error
from keras.callbacks import ReduceLROnPlateau, EarlyStopping, ModelCheckpoint
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
import seaborn as sns
sns.set_context("paper", font_scale=1.3)
sns.set_style('white')
import math
from sklearn.preprocessing import MinMaxScaler
# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list␣
 ↪all files under the input directory
warnings.filterwarnings("ignore")
plt.style.use('fivethirtyeight')
```

```
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# Any results you write to the current directory are saved as output.
```

[3]:
```
#Convert date coulmns to specific format
dateparse = lambda x: datetime.strptime(x, '%d-%b-%y')
#Read csv file
df = pd.read_csv(r'BrentOilPrices1.csv',parse_dates=['Date'],␣
 ↪date_parser=dateparse)
print("jos")
#Sort dataset by column Date
df = df.sort_values('Date')
df = df.groupby('Date')['Price'].sum().reset_index()
df.set_index('Date', inplace=True)
#df=df.loc[datetime.date(year=2000,month=1,day=1):]
df=df.loc[datetime.strptime('2000-01-01',"%Y-%d-%m"):]
```

```
jos
```

[3]:
```
# Print some data rows.
df.head()
```

[3]:

Date	Price
2000-01-04	23.95
2000-01-05	23.72
2000-01-06	23.55
2000-01-07	23.35
2000-01-10	22.77

0.1.2 DataFrame Info Function

The following function `DfInfo` provides a summary of a given DataFrame, including the column types, the number of null values, and the percentage of null values

[4]:
```
#Read dataframe info
def DfInfo(df_initial):
    # gives some infos on columns types and numer of null values
    tab_info = pd.DataFrame(df_initial.dtypes).T.rename(index={0: 'column␣
 ↪type'})
    tab_info = tab_info.append(pd.DataFrame(df_initial.isnull().sum()).T.
 ↪rename(index={0: 'null values (nb)'}))
    tab_info = tab_info.append(pd.DataFrame(df_initial.isnull().sum() /␣
 ↪df_initial.shape[0] * 100).T.
```

```
                                rename(index={0: 'null values (%)'}))
    return tab_info
```

[5]:
```
df.index
```

[5]:
```
DatetimeIndex(['2000-01-04', '2000-01-05', '2000-01-06', '2000-01-07',
               '2000-01-10', '2000-01-11', '2000-01-12', '2000-01-13',
               '2000-01-14', '2000-01-17',
               …
               '2020-04-06', '2020-04-07', '2020-04-08', '2020-04-09',
               '2020-04-14', '2020-04-15', '2020-04-16', '2020-04-17',
               '2020-04-20', '2020-04-21'],
              dtype='datetime64[ns]', name='Date', length=5160, freq=None)
```

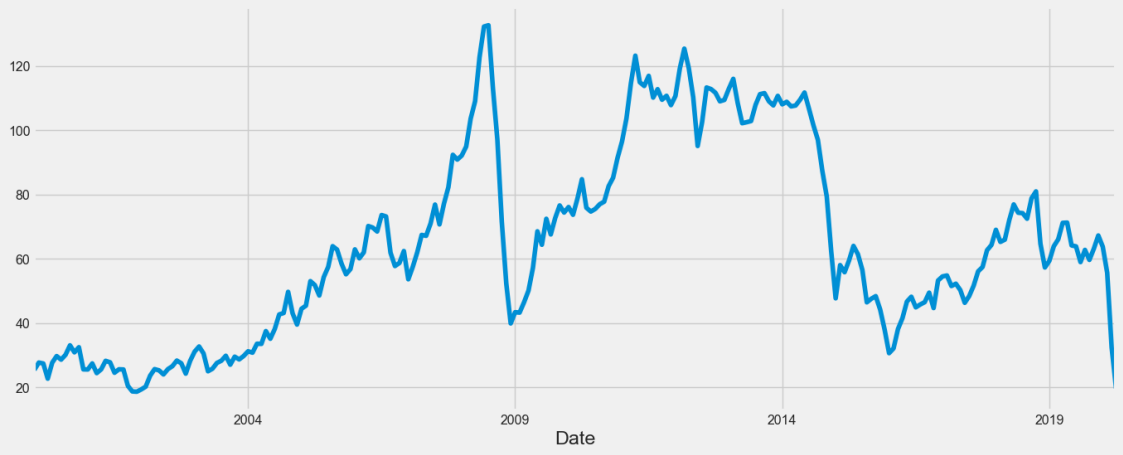
0.1.3 Resampling Data to Monthly Frequency

The following line of code resamples the `Price` column in a DataFrame to a monthly frequency and calculates the average price for each month:

[5]:
```
y = df['Price'].resample('MS').mean()
```

0.2 Plotting the 'Price' column

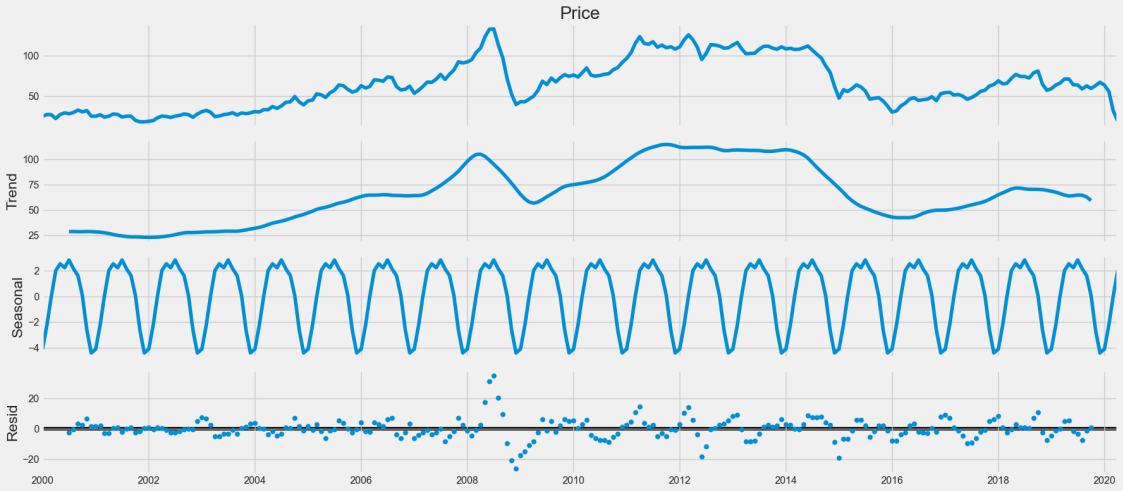
[6]:
```
y.plot(figsize=(15, 6))
plt.show()
```



0.2.1 Seasonal Decomposition of Time Series

The following code performs seasonal decomposition of a time series into its components: trend, seasonal, and residual.

[7]:
```
rcParams['figure.figsize'] = 18, 8
decomposition = sm.tsa.seasonal_decompose(y, model='additive')
fig = decomposition.plot()
plt.show()
```



0.3 from the graph bwe can see that here , trend , seasonal components are present .

0.3.1 Normalizing the Dataset Using MinMaxScaler

In data preprocessing, it's common to normalize the features before applying machine learning algorithms. The following code snippet demonstrates how to normalize a dataset using `MinMaxScaler` from `sklearn.preprocessing`. This scales the values in the dataset so that they lie within the specified range, typically between 0 and 1.

[8]:
```
# normalize the data_set
sc = MinMaxScaler(feature_range = (0, 1))
df = sc.fit_transform(df)
```

0.3.2 Splitting the Dataset into Training and Testing Sets

Before applying machine learning models, it's essential to divide the dataset into training and testing sets. This allows the model to be trained on one portion of the data (training set) and evaluated on another portion (testing set). The following code splits the dataset into 70% training and 30% testing sets.

[9]:
```
# split into train and test sets
train_size = int(len(df) * 0.70)
test_size = len(df) - train_size
train, test = df[0:train_size, :], df[train_size:len(df), :]
```

0.3.3 Converting an Array of Values into a Dataset Matrix

To prepare the data for time series forecasting, a dataset matrix can be created where each input `X` consists of a certain number of previous time steps (`look_back`), and the corresponding output `Y` is the next value in the sequence. The following function transforms a dataset into such a format.

[10]:
```
# convert an array of values into a data_set matrix def
def create_data_set(_data_set, _look_back=1):
    data_x, data_y = [], []
    for i in range(len(_data_set) - _look_back - 1):
        a = _data_set[i:(i + _look_back), 0]
        data_x.append(a)
        data_y.append(_data_set[i + _look_back, 0])
    return np.array(data_x), np.array(data_y)
```

0.3.4 Reshaping Data into X=t and Y=t+1

To prepare the data for a sequence-to-sequence forecasting model like an LSTM (Long Short-Term Memory) network, the data needs to be reshaped into sequences of `X=t` (inputs) and `Y=t+1` (outputs). Each sequence of past observations (e.g., `t` time steps) is used to predict the next value in the time series (`t+1`).

The following code achieves this by: - Splitting the data into training and testing sets. - Using the `create_data_set` function to generate the sequences of inputs (`X`) and corresponding outputs (`Y`). - Reshaping the inputs to fit the 3D shape required by LSTMs (samples, timesteps, features).

[11]:
```
# reshape into X=t and Y=t+1
look_back =90
X_train,Y_train,X_test,Ytest = [],[],[],[]
X_train,Y_train=create_data_set(train,look_back)
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test,Y_test=create_data_set(test,look_back)
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
```

0.3.5 Creating and Fitting an LSTM Network Regressor

In this section, we create and train a Long Short-Term Memory (LSTM) network for time series forecasting. The LSTM model is well-suited for learning patterns in sequential data.

[12]:
```
# create and fit the LSTM network regressor = Sequential()
regressor = Sequential()

regressor.add(LSTM(units = 5, return_sequences = True, input_shape = (X_train.
 ↪shape[1], 1)))
regressor.add(Dropout(0.1))

regressor.add(LSTM(units = 5, return_sequences = True))
regressor.add(Dropout(0.1))
# # following two added by JJ
```

```
# regressor.add(LSTM(units = 60, return_sequences = True))
# regressor.add(Dropout(0.1))

regressor.add(LSTM(units = 5))
regressor.add(Dropout(0.1))

regressor.add(Dense(units = 1))


regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')
reduce_lr = ReduceLROnPlateau(monitor='val_loss',patience=5)
history =regressor.fit(X_train, Y_train, epochs = 20, batch_size =␣
 ↪15,validation_data=(X_test, Y_test), callbacks=[reduce_lr],shuffle=False)
```

```
Epoch 1/20
235/235             22s 70ms/step -
loss: 0.0124 - val_loss: 0.0845 - learning_rate: 0.0010
Epoch 2/20
235/235             17s 73ms/step -
loss: 0.0442 - val_loss: 0.0592 - learning_rate: 0.0010
Epoch 3/20
235/235             18s 75ms/step -
loss: 0.0329 - val_loss: 0.0292 - learning_rate: 0.0010
Epoch 4/20
235/235             18s 76ms/step -
loss: 0.0124 - val_loss: 0.0103 - learning_rate: 0.0010
Epoch 5/20
235/235             16s 67ms/step -
loss: 0.0048 - val_loss: 0.0070 - learning_rate: 0.0010
Epoch 6/20
235/235             14s 60ms/step -
loss: 0.0035 - val_loss: 0.0080 - learning_rate: 0.0010
Epoch 7/20
235/235             16s 70ms/step -
loss: 0.0032 - val_loss: 0.0058 - learning_rate: 0.0010
Epoch 8/20
235/235             16s 67ms/step -
loss: 0.0029 - val_loss: 0.0070 - learning_rate: 0.0010
Epoch 9/20
235/235             22s 74ms/step -
loss: 0.0027 - val_loss: 0.0043 - learning_rate: 0.0010
Epoch 10/20
235/235             13s 56ms/step -
loss: 0.0021 - val_loss: 0.0052 - learning_rate: 0.0010
Epoch 11/20
235/235             12s 50ms/step -
loss: 0.0026 - val_loss: 0.0054 - learning_rate: 0.0010
```

```
Epoch 12/20
235/235             12s 50ms/step -
loss: 0.0026 - val_loss: 0.0038 - learning_rate: 0.0010
Epoch 13/20
235/235             12s 53ms/step -
loss: 0.0023 - val_loss: 0.0075 - learning_rate: 0.0010
Epoch 14/20
235/235             15s 63ms/step -
loss: 0.0030 - val_loss: 0.0051 - learning_rate: 0.0010
Epoch 15/20
235/235             14s 61ms/step -
loss: 0.0025 - val_loss: 0.0027 - learning_rate: 0.0010
Epoch 16/20
235/235             13s 53ms/step -
loss: 0.0022 - val_loss: 0.0052 - learning_rate: 0.0010
Epoch 17/20
235/235             12s 50ms/step -
loss: 0.0029 - val_loss: 0.0079 - learning_rate: 0.0010
Epoch 18/20
235/235             23s 97ms/step -
loss: 0.0035 - val_loss: 0.0059 - learning_rate: 0.0010
Epoch 19/20
235/235             27s 117ms/step -
loss: 0.0030 - val_loss: 0.0067 - learning_rate: 0.0010
Epoch 20/20
235/235             40s 111ms/step -
loss: 0.0035 - val_loss: 0.0055 - learning_rate: 0.0010
```

0.3.6 Making Predictions with the Trained LSTM Network

After training the LSTM network, we can use it to make predictions on both the training and test datasets. This allows us to evaluate the model's performance and analyze the results.

[13]:
```
train_predict = regressor.predict(X_train)
test_predict = regressor.predict(X_test)
```

```
110/110             4s 22ms/step
46/46             2s 35ms/step
```

0.3.7 Inverting Predictions

After training the LSTM model and making predictions on the training and test datasets, the next step is to revert the scaled predictions back to their original value range. This is particularly important if we applied normalization (such as Min-Max scaling) to your data before training. Below is the process for inverting both the predicted values and the true values.

[14]:
```
# invert predictions
train_predict = sc.inverse_transform(train_predict)
Y_train = sc.inverse_transform([Y_train])
```

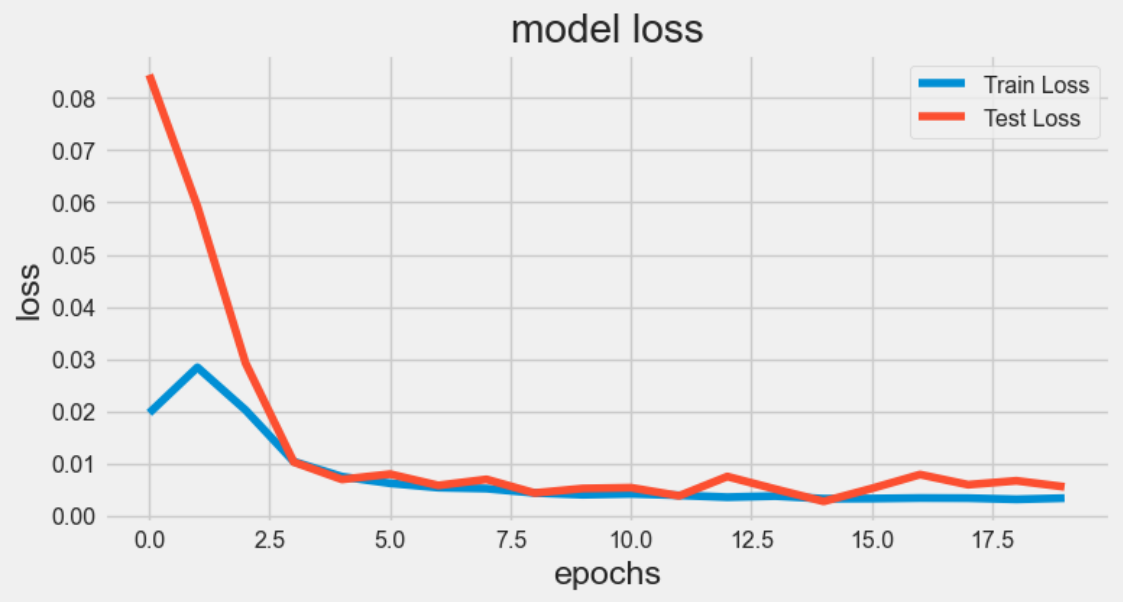
```
test_predict = sc.inverse_transform(test_predict)
Y_test = sc.inverse_transform([Y_test])
```

0.3.8 Model Performance Evaluation

After training the LSTM model, it's important to evaluate its performance using appropriate metrics. Here, we use Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) to assess the accuracy of the predictions on both the training and test datasets. Additionally, we visualize the loss during training to monitor the model's performance over epochs.

[15]:
```
print('Train Mean Absolute Error:', mean_absolute_error(Y_train[0],␣
 ↪train_predict[:,0]))
print('Train Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_train[0],␣
 ↪train_predict[:,0])))
print('Test Mean Absolute Error:', mean_absolute_error(Y_test[0], test_predict[:
 ↪,0]))
print('Test Root Mean Squared Error:',np.sqrt(mean_squared_error(Y_test[0],␣
 ↪test_predict[:,0])))
plt.figure(figsize=(8,4))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epochs')
plt.legend(loc='upper right')
plt.show();
```

```
Train Mean Absolute Error: 8.63926199427518
Train Root Mean Squared Error: 10.025784615489563
Test Mean Absolute Error: 9.301374057931994
Test Root Mean Squared Error: 10.025318171672788
```



0.3.9 Interpretation of Model Loss Plot

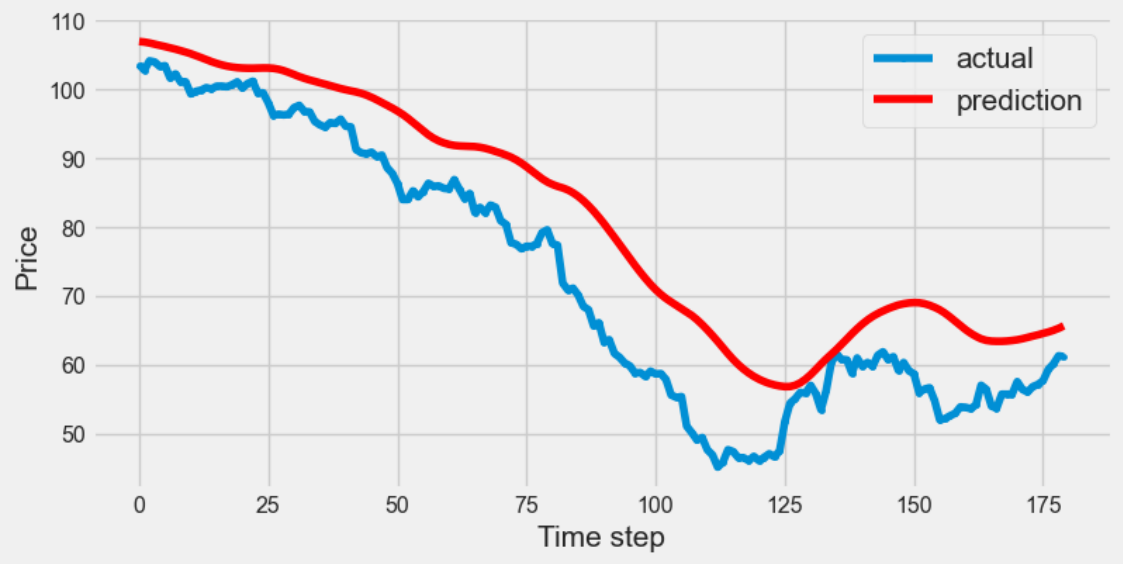
1. **Training Loss (Blue Line):**
   - Decreases significantly in the initial epochs, indicating effective learning.
   - Stabilizes at a low value after around 10 epochs, suggesting optimal learning.
2. **Test Loss (Red Line):**
   - Decreases but remains higher than training loss, indicating potential overfitting.
   - The gap suggests the model is tailored to training data but less so to unseen data.
3. **Stability:**
   - Both losses stabilize towards the end of training, indicating learned patterns.
   - Continuous monitoring is needed to assess if further training or adjustments are required.

0.3.10 Actual vs. Predicted Values Comparison

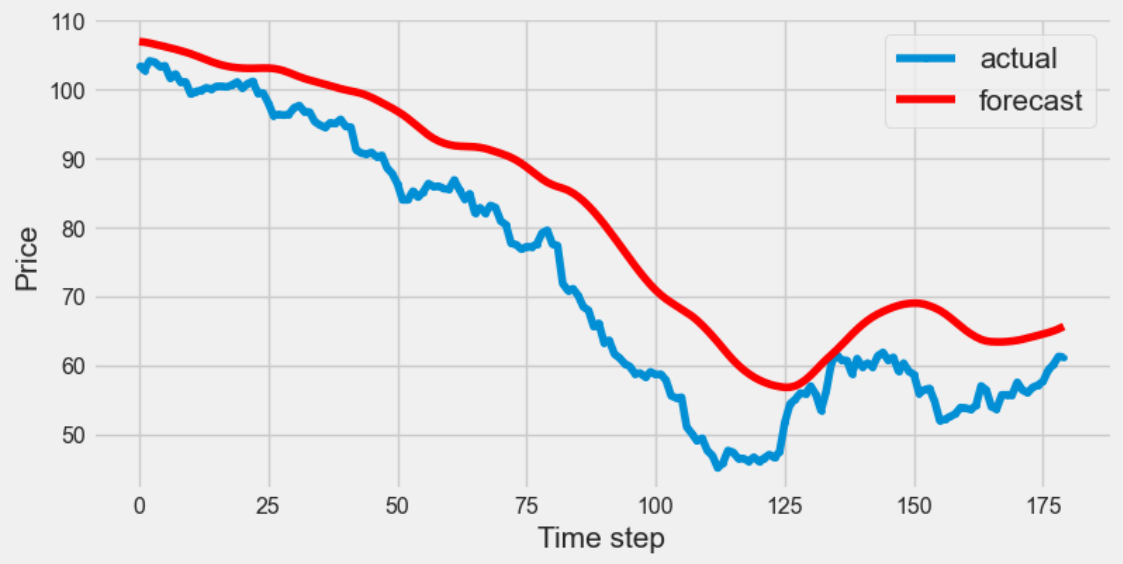
To evaluate the performance of our LSTM model visually, we can compare the actual prices against the predicted prices. This allows us to assess how well the model captures the underlying trends in the data.

[16]:
```
#Compare Actual vs. Prediction
aa=[x for x in range(180)]
plt.figure(figsize=(8,4))
plt.plot(aa, Y_test[0][:180], marker='.', label="actual")
plt.plot(aa, test_predict[:,0][:180], 'r', label="prediction")
plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=0.07)
plt.ylabel('Price', size=15)
plt.xlabel('Time step', size=15)
```

```
plt.legend(fontsize=15)
plt.show();
```



[19]:
```
# Compare Actual vs. Forecast
aa = [x for x in range(180)]
plt.figure(figsize=(8,4))
plt.plot(aa, Y_test[0][:180], marker='.', label="actual")
plt.plot(aa, test_predict[:,0][:180], 'r', label="forecast")
plt.tight_layout()
sns.despine(top=True)
plt.subplots_adjust(left=0.07)
plt.ylabel('Price', size=15)
plt.xlabel('Time step', size=15)
plt.legend(fontsize=15)
plt.show()

# Print forecasted values
print("Forecasted values for the first 180 time steps:")
print(test_predict[:,0][:180])
```




```
Forecasted values for the first 180 time steps:
[106.98494  106.90821  106.76974  106.60293  106.42807  106.24823
 106.0717   105.877    105.67208  105.448906 105.21175  104.9416
 104.64805  104.34894  104.06545  103.80755  103.588036 103.41164
 103.2759   103.18163  103.13151  103.10553  103.10294  103.12517
 103.139854 103.13427  103.08235  102.95084  102.744286 102.479485
 102.180786 101.88722  101.62664  101.3968   101.196266 101.00105
 100.79605  100.57538  100.35556  100.146385  99.96581   99.803314
  99.653595  99.4586    99.196655  98.87244   98.509224  98.11962
  97.72602   97.315346  96.8792    96.40162   95.85052   95.236046
  94.608574  93.989746  93.41539   92.92921   92.53943   92.244934
  92.03056   91.88078   91.80653   91.7762    91.74504   91.70761
  91.611244  91.4582    91.24625   91.00875   90.76361   90.488815
  90.176025  89.7842    89.31202   88.768974  88.18863   87.603355
  87.04833   86.576324  86.21617   85.934425  85.70645   85.41449
  85.00321   84.47932   83.85419   83.13028   82.33003   81.44771
  80.520645  79.53972   78.53581   77.51042   76.47814   75.45247
  74.45156   73.48153   72.5608    71.69849   70.91998   70.232445
  69.63969   69.12587   68.64284   68.167145  67.69753   67.16851
  66.55286   65.84539   65.076965  64.256454  63.399513  62.507355
  61.613216  60.77732   60.029907  59.371895  58.802406  58.312176
  57.90278   57.561325  57.284645  57.07445   56.91879   56.81999
  56.835743  57.018948  57.379677  57.906467  58.560757  59.31274
  60.102177  60.842342  61.529606  62.226006  62.969273  63.74442
  64.52438   65.247955  65.91844   66.51206   67.02681   67.45387
  67.82202   68.15545   68.44446   68.693115  68.869644  68.98666
  69.036064  69.01481   68.8851    68.65708   68.35805   67.98512
  67.50785   66.94034   66.32063   65.69197   65.10267   64.57955
  64.13182   63.77047   63.541233  63.44192   63.4142    63.415283
```

```
63.454365  63.52979   63.633846  63.78688   63.97014   64.16028
64.35477   64.55361   64.76173   65.00377   65.2961    65.65412 ]
```

0.3.11 Interpretation of Actual vs. Forecasted Prices Plot

1. **Trend Alignment:** Both actual prices (blue line) and predicted prices (red line) show a downward trend, indicating that the model captures the overall direction of price movement.
2. **Initial Accuracy:** In the early time steps (0 to 50), predictions closely follow actual values, showing strong initial performance.
3. **Divergence:** From about time steps 100 to 130, the predicted prices diverge significantly from actual prices, indicating the model struggles during this period.
4. **Subsequent Recovery:** After the divergence, the model's predictions improve and align more closely with actual prices towards the end of the time series (140 to 175).
5. **Smoothing Effect:** The predictions are smoother than actual prices, suggesting the model averages out noise, which may result in missed sudden changes.

0.4 The ARIMA model is the best among the three, as it has the lowest Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE). Lower error metrics indicate that ARIMA provides more accurate predictions compared to Exponential Smoothing and LSTM for this dataset.