EDA and Forecasting Brent Oil Prices

In [1]:
```
import numpy as np
import pandas as pd
```

In [3]:
```
df = pd.read_csv("BrentOilPrices.csv")
# Please use the csv in the same git folder to compare analysis
df.head()
```

Out[3]:

	Date	Price
0	May 20, 1987	18.63
1	May 21, 1987	18.45
2	May 22, 1987	18.55
3	May 25, 1987	18.60
4	May 26, 1987	18.63

Data Preprocessing

1. Need to convert Date column to standard format

In [6]:
```
import seaborn as sns
from matplotlib import pyplot as plt

df['Date'] = pd.to_datetime(df['Date'], format="%b %d, %Y")
df.head()
```

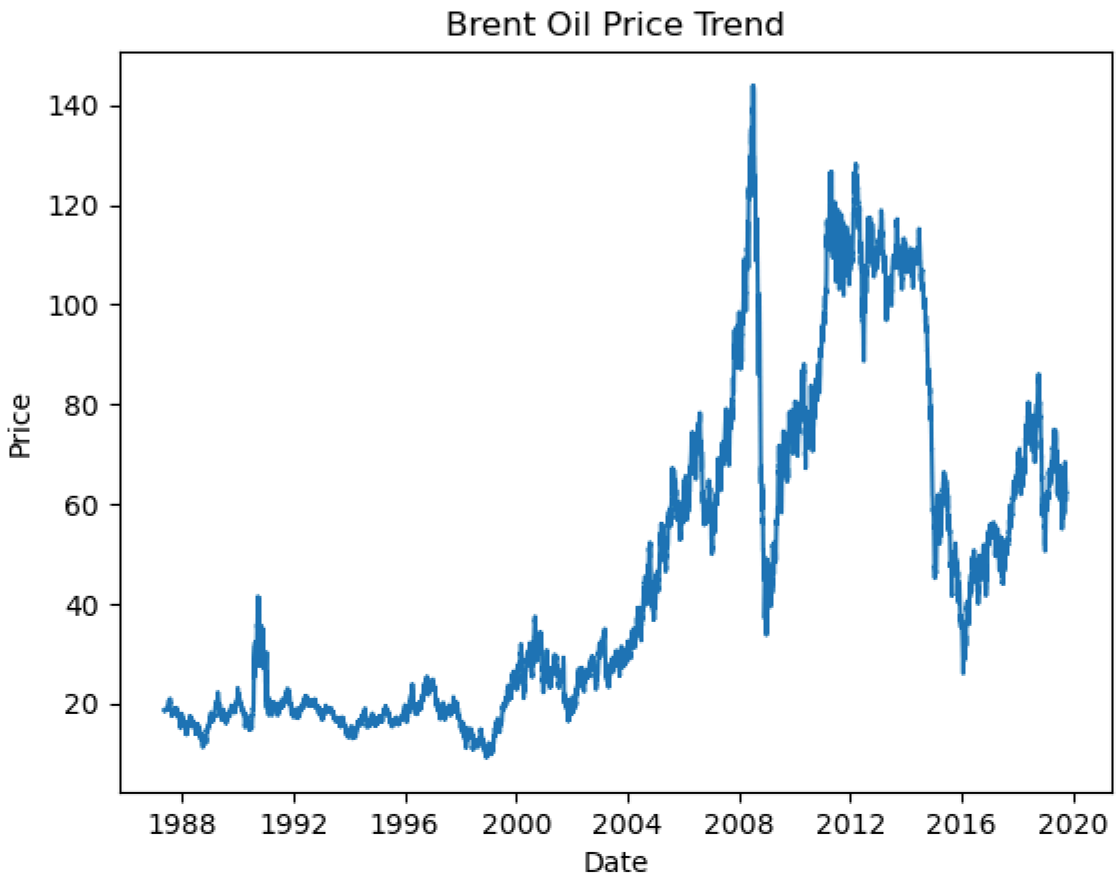
Out[6]:

	Date	Price
0	1987-05-20	18.63
1	1987-05-21	18.45
2	1987-05-22	18.55
3	1987-05-25	18.60
4	1987-05-26	18.63

Visualizing Full Data as a line plot

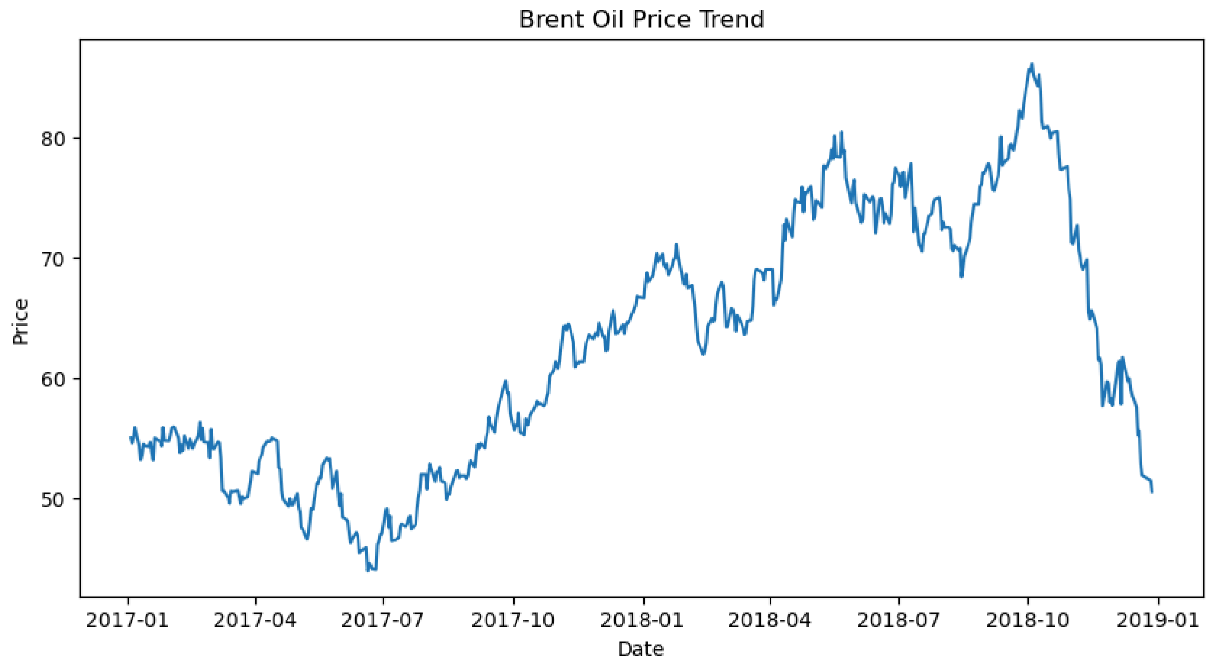
In [8]:
```
g = sns.lineplot(x='Date',y='Price',data = df)
plt.title("Brent Oil Price Trend")
```

Out[8]: Text(0.5, 1.0, 'Brent Oil Price Trend')



In [9]:
```
# Function to plot Oil Price Trend between specific period

def plot_price_trend(df, start_date, end_date):
    """
    This function filters the dataframe for the specified date range and
    plots the line plot of the data using seaborn.

    The dataframe may not be indexed on any Datetime column.
    In this case, we use mask to filter out the date.

    PS - There is another function provided later in the notebook
    which used indexed column to filter data
    """
    mask = (df['Date'] > start_date) & (df['Date'] <= end_date)
    sdf = df.loc[mask]
    plt.figure(figsize = (10,5))
    chart = sns.lineplot(x='Date',y='Price',data = sdf)
#     chart.set_xticklabels(chart.get_xticklabels(), rotation=45)
    plt.title("Brent Oil Price Trend")
```

In [10]:
```
# Test our function

plot_price_trend(df,'2017-01-01','2019-01-01')
```



Forecast Model Using Prophet

Step 1 - First we import the Prophet class from fbprophet module and then create an instance of this.

In [14]:
```
from prophet import Prophet
m = Prophet()
```

Step 2 - Note that Prophet requires the date column as 'ds' and outcome varible as 'y'. So we change this in our dataframe and check its data

In [17]:
```
pro_df = df
pro_df.columns = ['ds','y']
pro_df.head()
```

Out[17]:

	ds	y
0	1987-05-20	18.63
1	1987-05-21	18.45
2	1987-05-22	18.55
3	1987-05-25	18.60
4	1987-05-26	18.63

Step 3 - Next we fit this dataframe into the model object created and then create a forecast for the Oil Price for the next 90 days.

This might take ~1mins

In [20]:
```
m.fit(pro_df)
future = m.make_future_dataframe(periods = 90)
forecast = m.predict(future)
```

```
11:46:20 - cmdstanpy - INFO - Chain [1] start processing
11:46:28 - cmdstanpy - INFO - Chain [1] done processing
```

Step 4 - We check the forecast data has several components - trend, weakly and yearly seasonality - and for each of these components, we have the lower and upper confidence intervals data.

In [23]:
```
forecast.head()
```

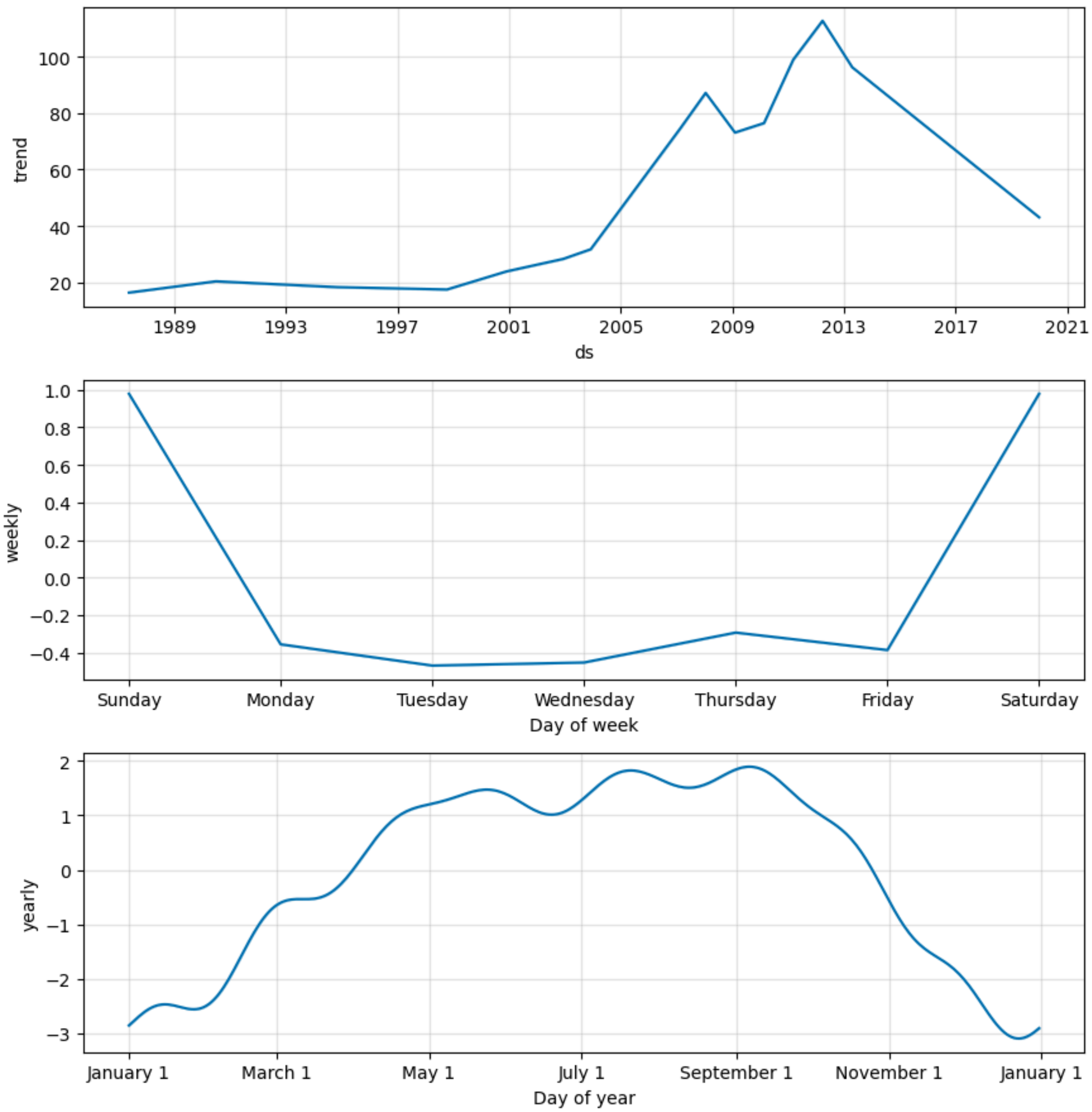
Out[23]:

	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	additive_terms	ad
0	1987-05-20	16.436499	0.928637	32.786393	16.436499	16.436499	0.994033	
1	1987-05-21	16.440049	2.722570	32.719206	16.440049	16.440049	1.162842	
2	1987-05-22	16.443599	2.499184	32.790695	16.443599	16.443599	1.077276	
3	1987-05-25	16.454249	2.095926	33.098442	16.454249	16.454249	1.115841	
4	1987-05-26	16.457799	1.438579	33.463511	16.457799	16.457799	1.000086	

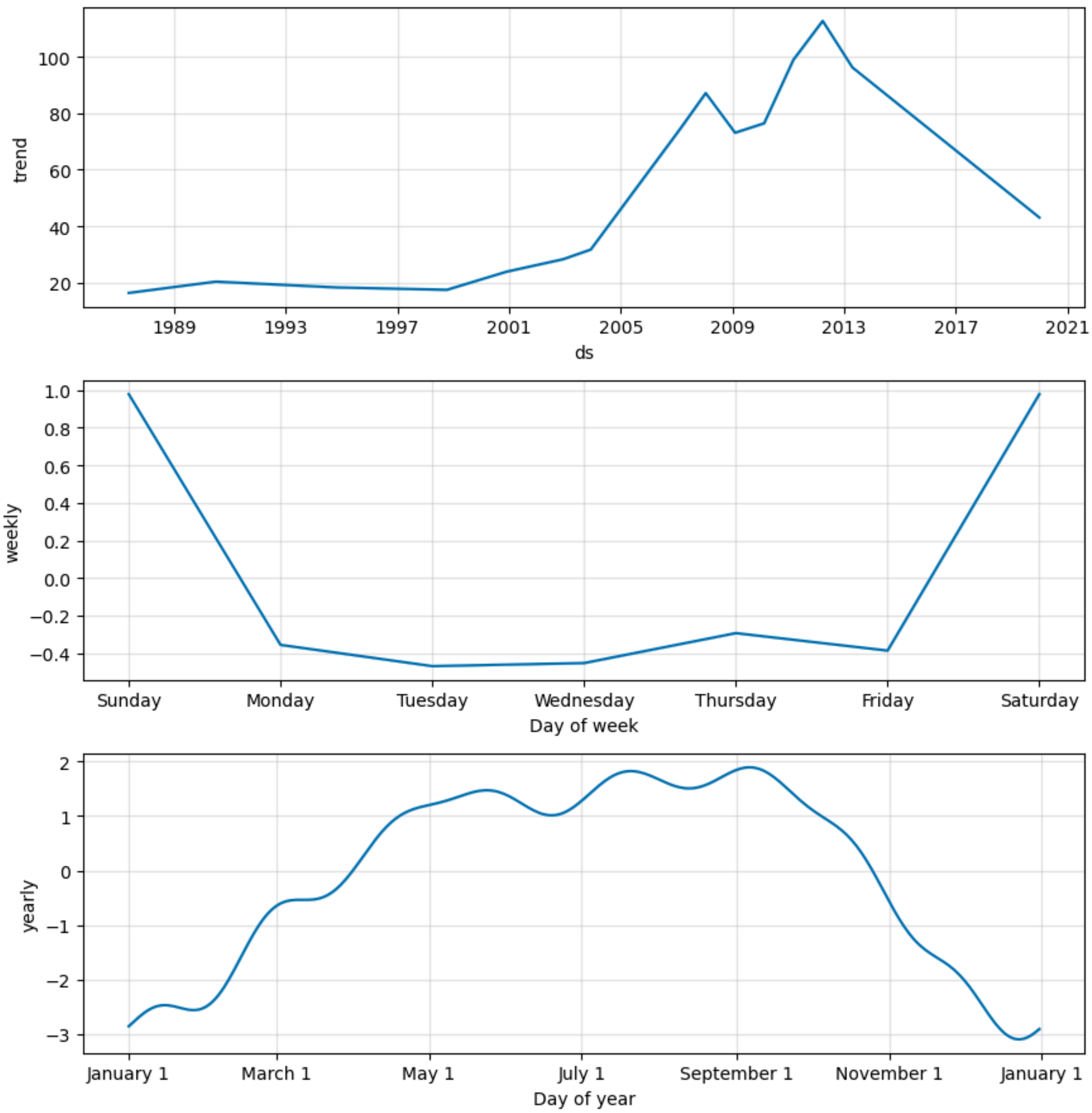
Step 5 - We plot these components of the forecast fit model.

In [26]:
```
m.plot_components(forecast)
```

Out[26]:

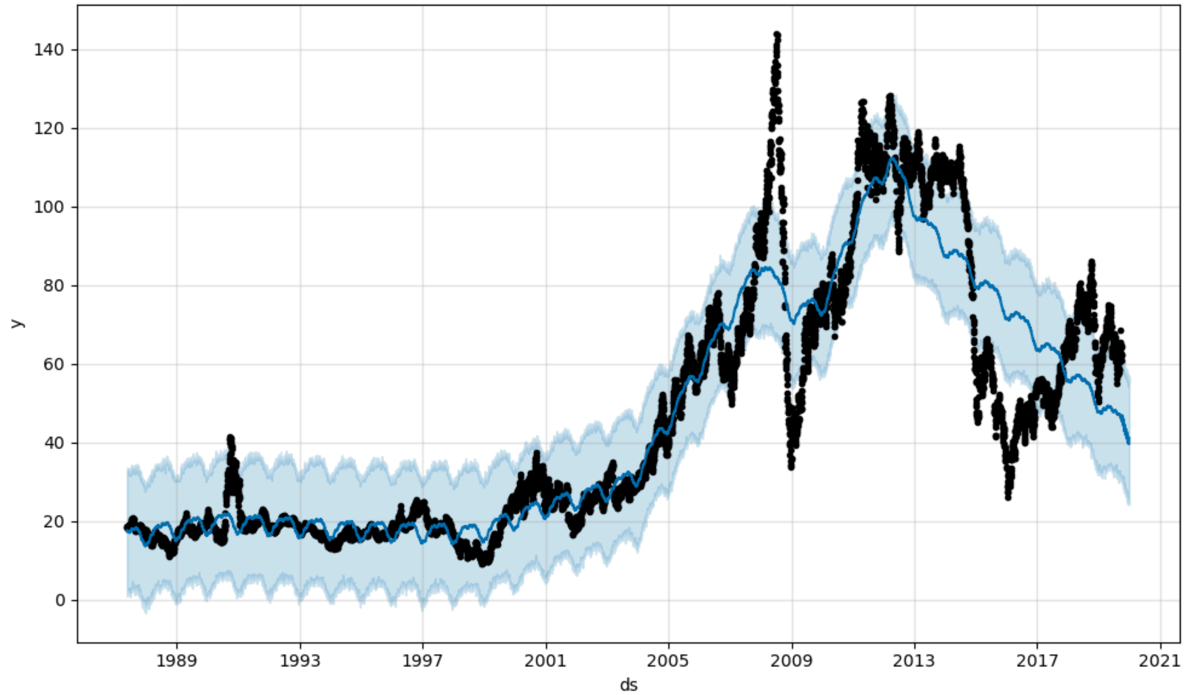


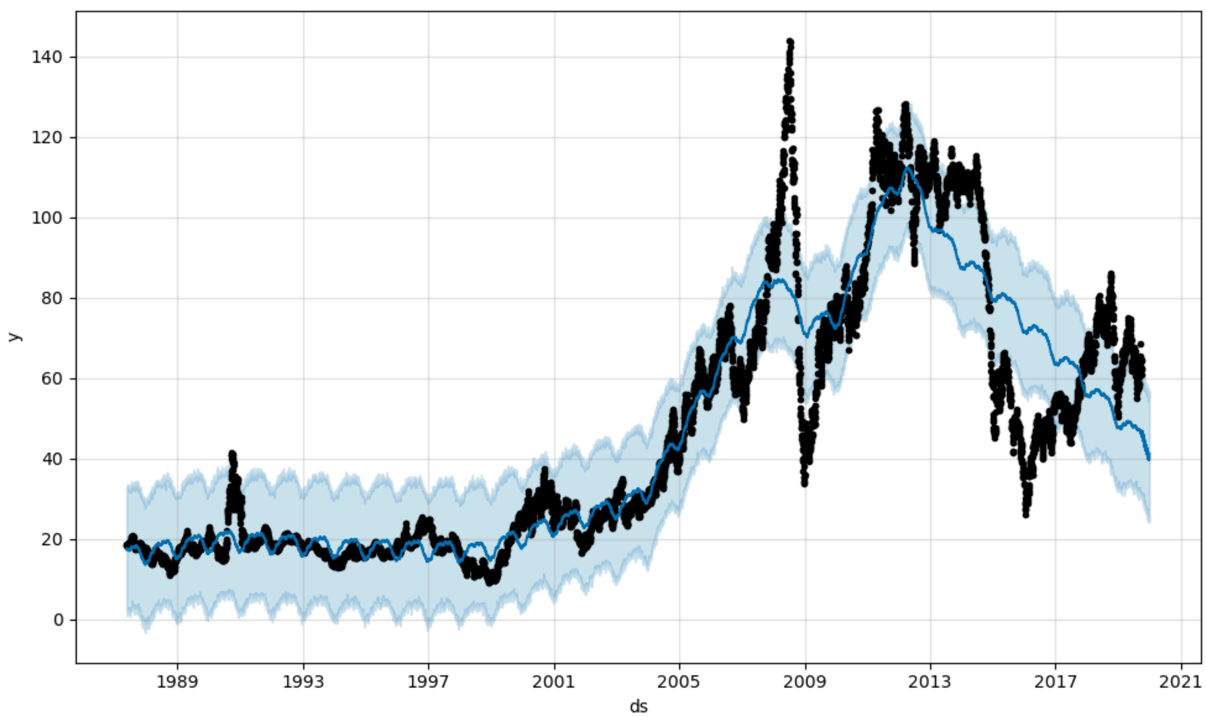
trend
100
80
60
40
20
1989
1993
1997
2001
2005
2009
2013
2017
2021
ds
weekly
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Day of week
yearly
2
1
0
−1
−2
−3
January 1
March 1
May 1
July 1
September 1
November 1
January 1
Day of year



In [27]:
```
m.plot(forecast)
```

Out[27]:





Step 6 - Next we want to visualize side by side the original data and the forecast data. So for this, we join the original and forecast data on the column 'ds'

In [29]:
```
cmp_df = forecast.set_index('ds')[['yhat','yhat_lower','yhat_upper']].join(pro_df.s
cmp_df.head()
```

Out[29]:

ds	yhat	yhat_lower	yhat_upper	y
1987-05-20	17.430531	0.928637	32.786393	18.63
1987-05-21	17.602891	2.722570	32.719206	18.45
1987-05-22	17.520875	2.499184	32.790695	18.55
1987-05-25	17.570090	2.095926	33.098442	18.60
1987-05-26	17.457885	1.438579	33.463511	18.63

In [30]:

```
cmp_df.tail(5)
```

Out[30]:

ds	yhat	yhat_lower	yhat_upper	y
2019-12-25	39.651475	25.274974	55.986695	NaN
2019-12-26	39.803295	24.213094	55.812940	NaN
2019-12-27	39.708538	24.420255	55.292734	NaN
2019-12-28	41.078033	25.229344	55.659367	NaN
2019-12-29	41.086199	25.883526	56.846418	NaN

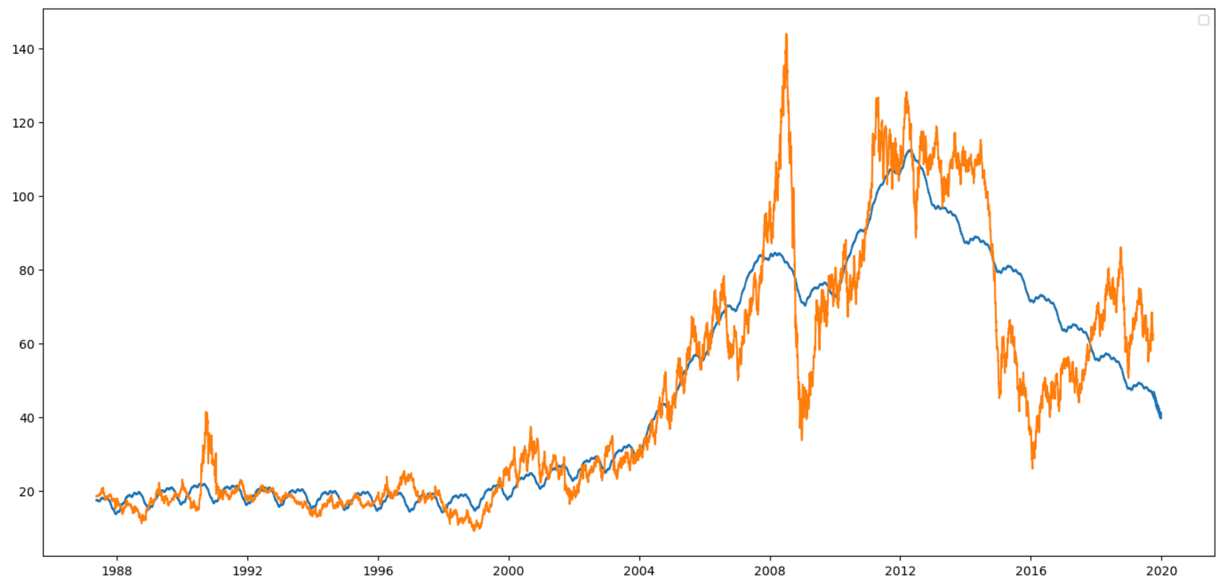
Note that the original y data is NaN towards the end because, these are the predicted dates.

Step 7 - Then, we visualize the original and forecast data alongside each other

In [32]:

```
plt.figure(figsize=(17,8))
#plt.plot(cmp_df['yhat_lower'])
#plt.plot(cmp_df['yhat_upper'])
plt.plot(cmp_df['yhat'])
plt.plot(cmp_df['y'])
plt.legend()
plt.show()
```

```
C:\Users\Mahaveera Foods\AppData\Local\Temp\ipykernel_16940\1898636383.py:6: UserWar
ning: No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
  plt.legend()
```



Step 8 - From above graph, we are not able to readily see how many months data was forecast.

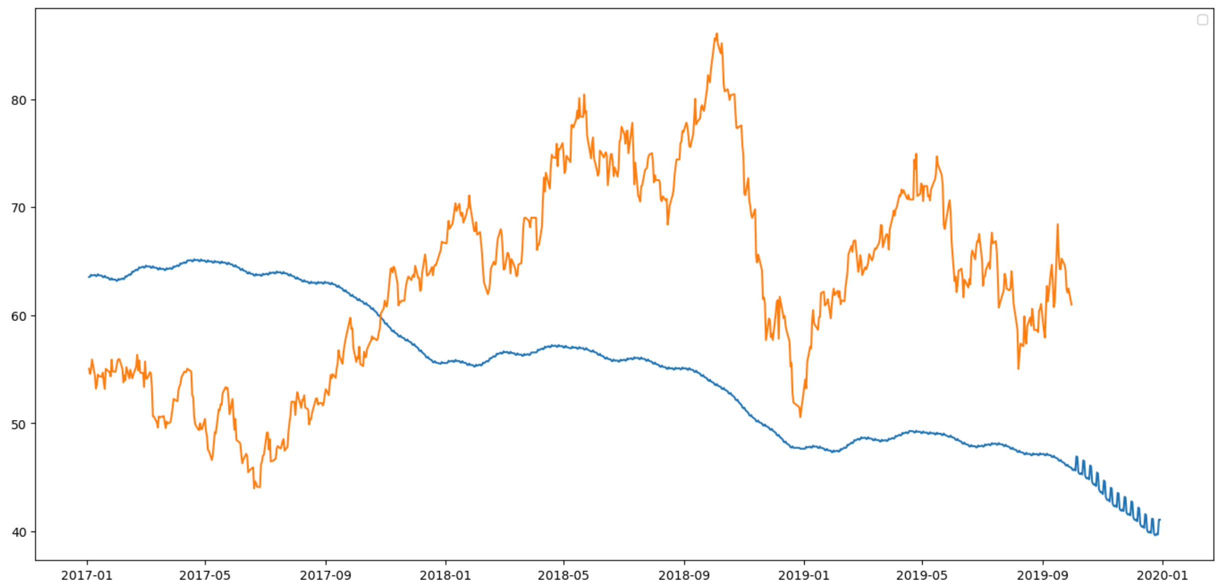
So, We need a function which will show us the original and forecast data between a specified date range.

In [34]:
```
def plot_price_forecast(df,start_date, end_date):
    """
    This function filters the dataframe for the specified date range and
    plots the actual and forecast data.

    Assumption:
    - The dataframe has to be indexed on a Datetime column
    This makes the filtering very easy in pandas using df.loc
    """
    cmp_df = df.loc[start_date:end_date]
    plt.figure(figsize=(17,8))
    plt.plot(cmp_df['yhat'])
    plt.plot(cmp_df['y'])
    plt.legend()
    plt.show()
```

Stpe 9 - Using this function, we can see that, the original graph (orange) does not have data towards the end. This data can be taken from the forecasted graph (blue).

In [36]:
```
plot_price_forecast(cmp_df,'2017-01-01','2020-01-01')
```

```
C:\Users\Mahaveera Foods\AppData\Local\Temp\ipykernel_16940\220737787.py:14: UserWar
ning: No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
  plt.legend()
```



Using ARIMA

Step 1 - First we import the required libraries

In [40]:
```
from statsmodels.tsa.arima_model import ARIMA    # ARIMA Modeling
from statsmodels.tsa.stattools import adfuller   # Augmented Dickey-Fuller Test for
from statsmodels.tsa.stattools import acf, pacf  # Finding ARIMA parameters using A
from statsmodels.tsa.seasonal import seasonal_decompose # Decompose the ARIMA Forec
```

Step 2 - Arima requires the date column to be set as index

In [42]:
```
arima_df = df.set_index('ds')
arima_df.head()
```

Out[42]:

	y
ds	
1987-05-20	18.63
1987-05-21	18.45
1987-05-22	18.55
1987-05-25	18.60
1987-05-26	18.63

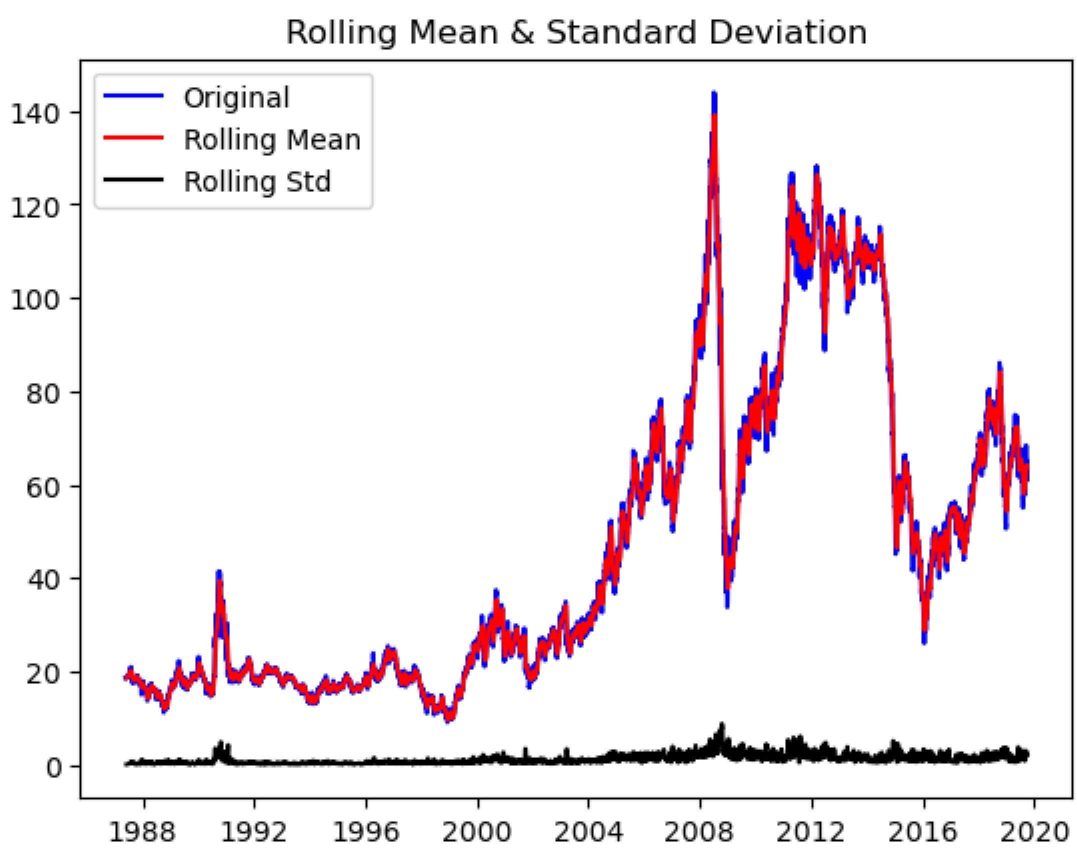
Step 3 - Next we write a function that plots the Rolling mean and standard deviation and then checks the stationarity of the time series using Augmented Dickey - Fuller Test

Credit - https://www.kaggle.com/freespirit08/time-series-for-beginners-with-arima

In [44]:
```
# Perform Augmented Dickey-Fuller test to check if the given Time series is station
def test_stationarity(ts):

    #Determing rolling statistics
    rolmean = ts.rolling(window=12).mean()
    rolstd = ts.rolling(window=12).std()

    #Plot rolling statistics:
    orig = plt.plot(ts, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)

    #Perform Dickey-Fuller test:
    print('Results of Dickey-Fuller Test:')
    dftest = adfuller(ts['y'], autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print(dfoutput)
```

Step 4 - Next, we use this function to check if our given timeseries data is stationary or not

In [46]:
```
test_stationarity(arima_df)
```



```
Results of Dickey-Fuller Test:
Test Statistic                   -1.954749
p-value                           0.306759
#Lags Used                       35.000000
Number of Observations Used    8180.000000
Critical Value (1%)              -3.431150
Critical Value (5%)              -2.861893
Critical Value (10%)             -2.566958
dtype: float64
```

Observation - The null hypothesis of ADF test is the Time series is NOT stationary. We see that the Test Statistic (-1.95) is higher than 10% Critical Value (-2.56). This means this result is statistically significant at 90% confidence interval and so, we fail to reject the null hypothesis.

This means that our time series data is NOT stationary.

Step 5 - Some definitions -

Correlation - Describes how much two variables depend on each other.

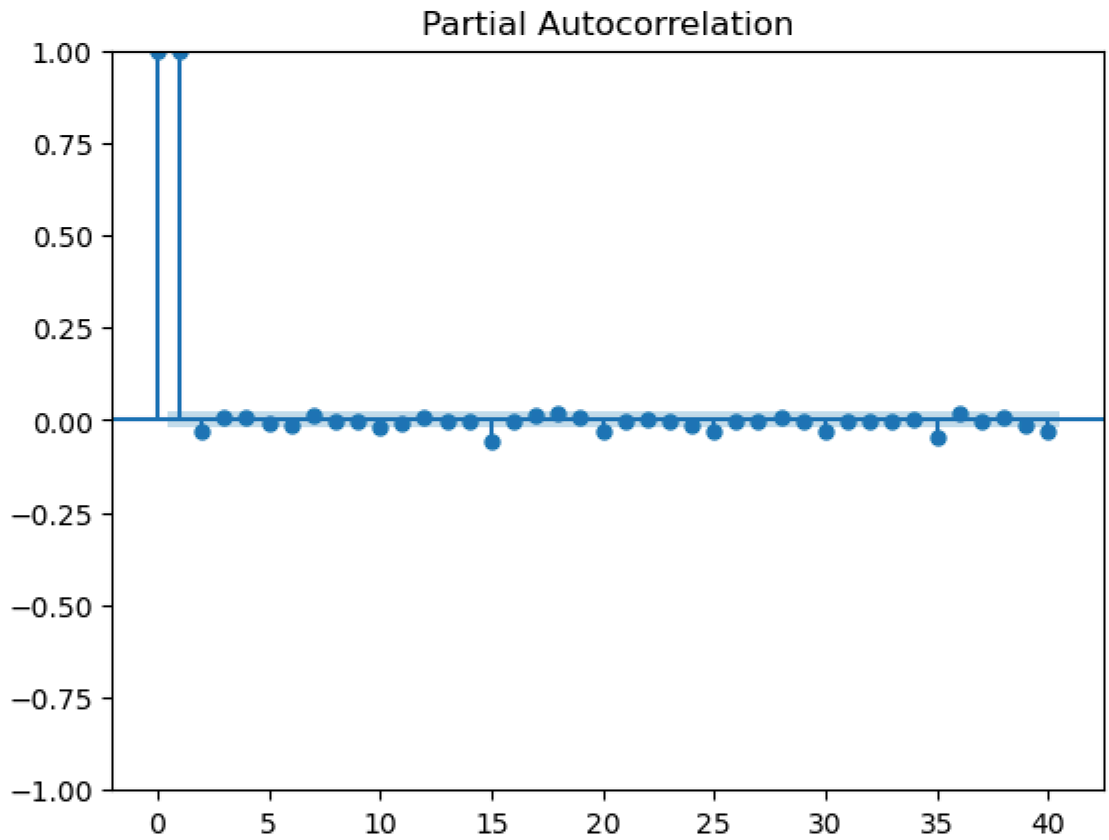
Partial Correlation - When multiple variables are involved, two variables may have direct relation as well as indirect relation (i.e x1 and x3 are related and x2 and x3 are related. Due to this indirect relation, x1 and x2 might be related). This is called partial correlation.

Auto Correlation - In a time series data, variable at a time step is dependent upon its lag values. This is called auto-correlation (i.e. variable depending upon its own values)

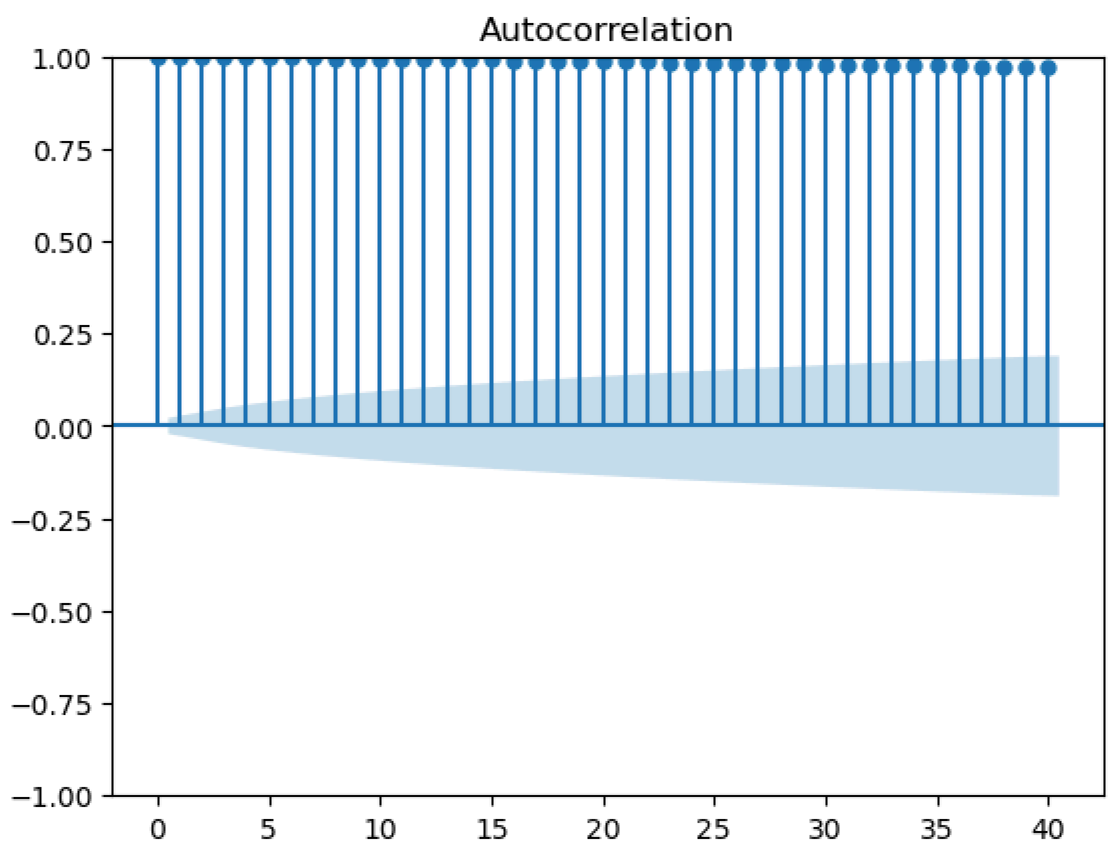
Partial Autocorrelation - describes correlation of a variable with its lag values after removing the effect of indirect correlation.

In [51]:
```
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
plot_acf(arima_df)
plot_pacf(arima_df)
```

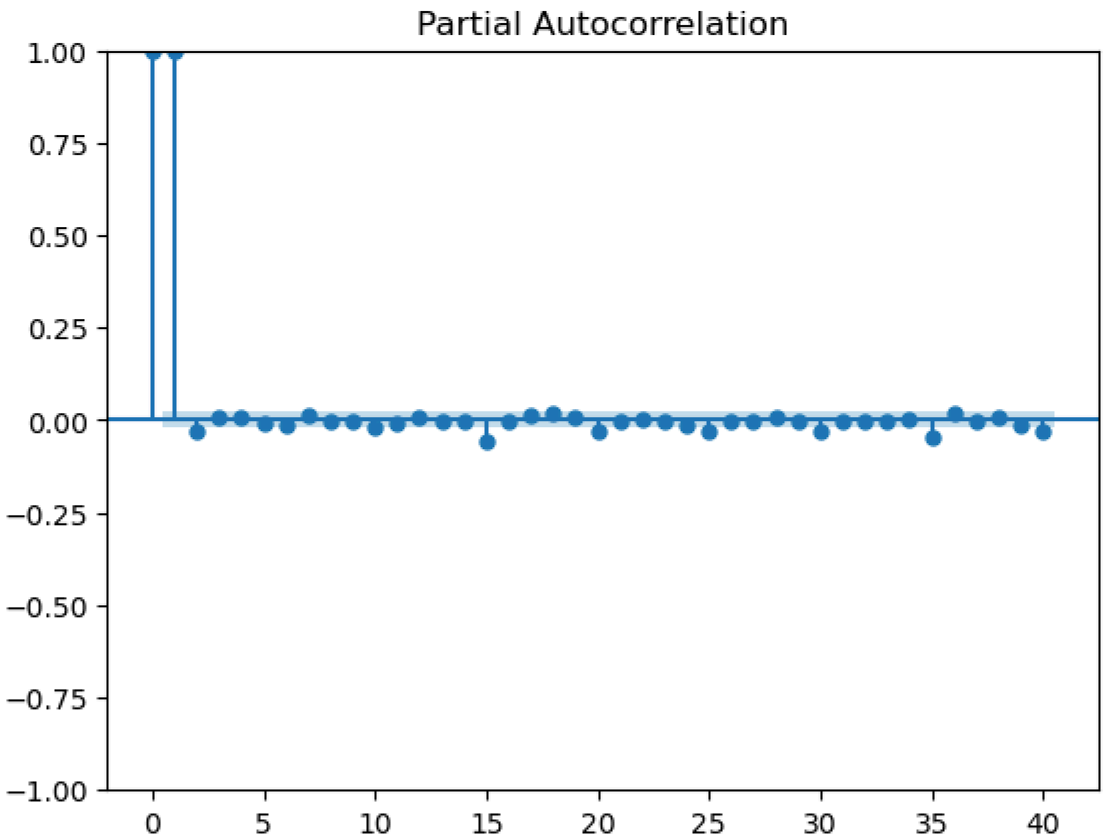
Out[51]:



Partial Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
25
30
35
40

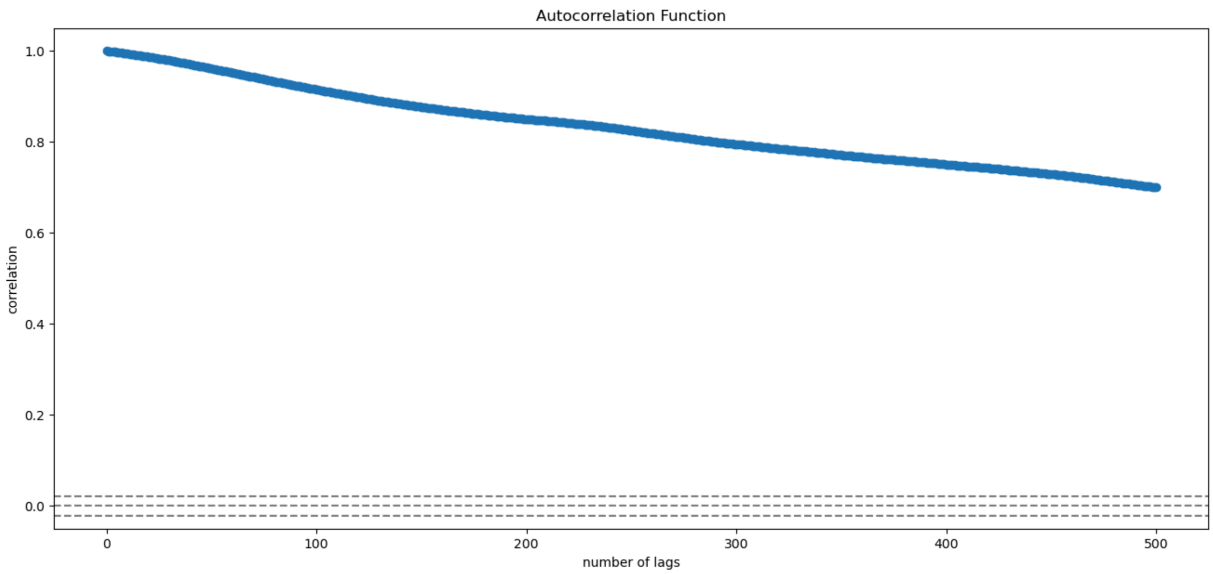


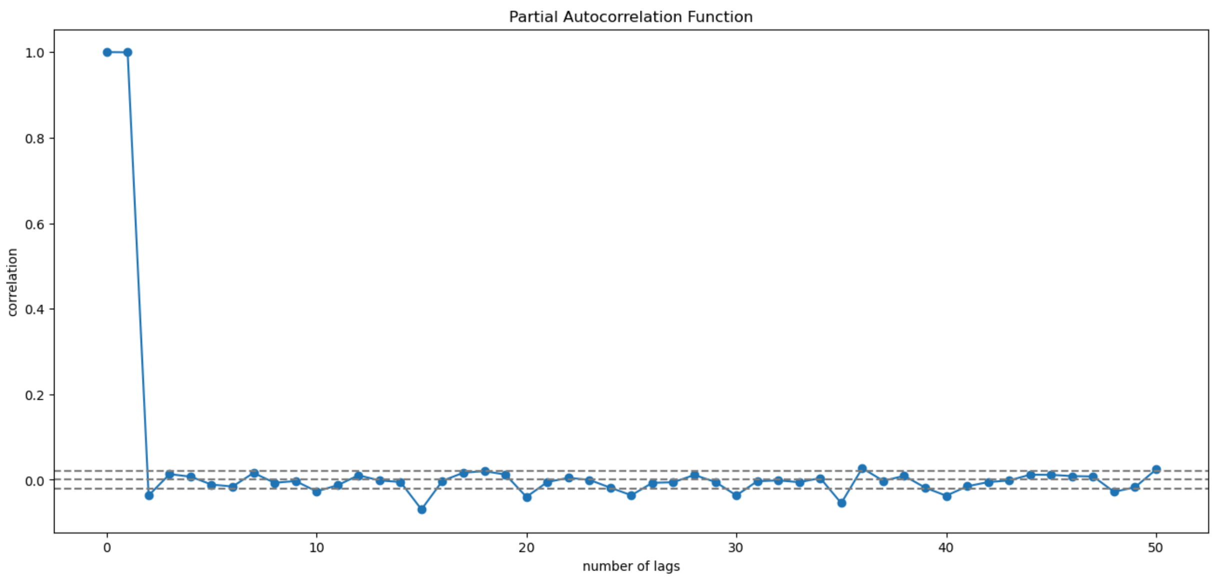
Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
25
30
35
40



In [52]:
```
# Implementing own function to create ACF plot
def get_acf_plot(ts):
    #calling acf function from stattools
    y = ts['y']
    lag_acf = acf(y, nlags=500)
    plt.figure(figsize=(16, 7))
    plt.plot(lag_acf, marker="o")
    plt.axhline(y=0,linestyle='--',color='gray')
    plt.axhline(y=-1.96/np.sqrt(len(y)),linestyle='--',color='gray')
    plt.axhline(y=1.96/np.sqrt(len(y)),linestyle='--',color='gray')
    plt.title('Autocorrelation Function')
    plt.xlabel('number of lags')
    plt.ylabel('correlation')

def get_pacf_plot(ts):
    #calling pacf function from stattools
    y = arima_df['y']
    lag_pacf = pacf(y, nlags=50)
    plt.figure(figsize=(16, 7))
    plt.plot(lag_pacf, marker="o")
    plt.axhline(y=0,linestyle='--',color='gray')
    plt.axhline(y=-1.96/np.sqrt(len(y)),linestyle='--',color='gray')
    plt.axhline(y=1.96/np.sqrt(len(y)),linestyle='--',color='gray')
    plt.title('Partial Autocorrelation Function')
    plt.xlabel('number of lags')
    plt.ylabel('correlation')
```

In [53]:
```
get_acf_plot(arima_df)
get_pacf_plot(arima_df)
```

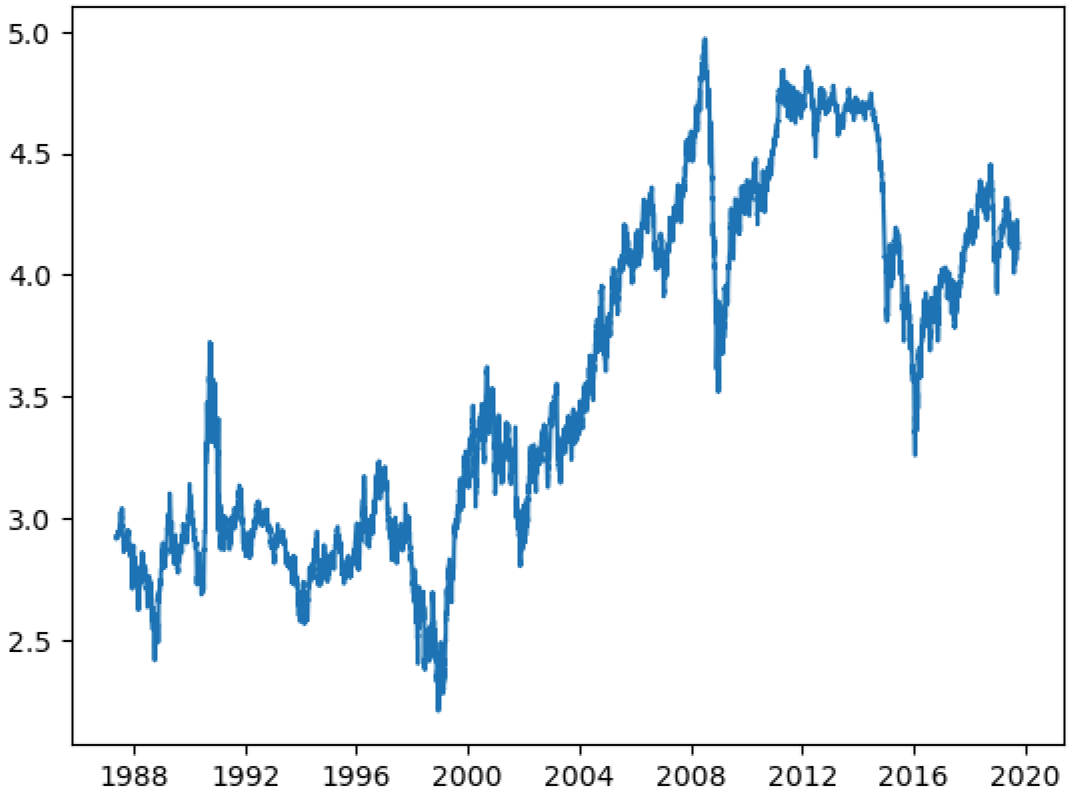




Step 6 - Next we see some methods to make the data stationary

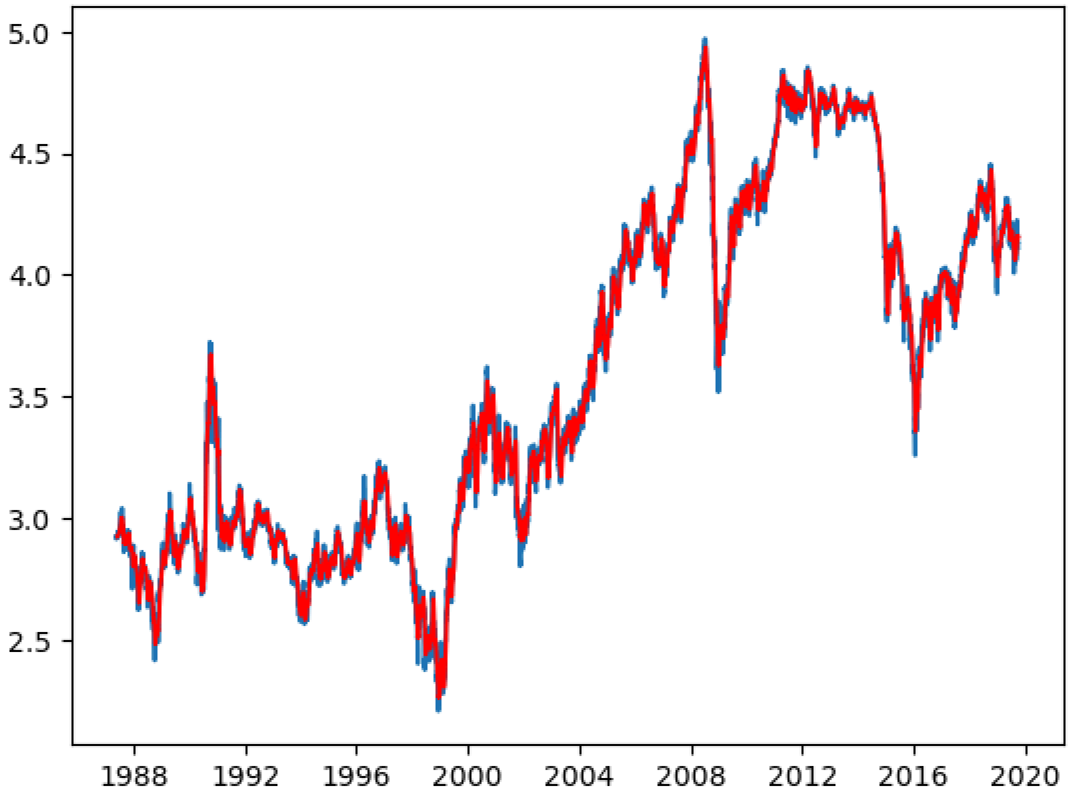
In [57]:
```
# Log Transformation
ts_log = np.log(arima_df)
plt.plot(ts_log)
```

Out[57]:
```
[<matplotlib.lines.Line2D at 0x1c501c75880>]
```



In [58]:
```
# Moving Average of last 12 values
moving_avg = ts_log.rolling(12).mean()
plt.plot(ts_log)
plt.plot(moving_avg, color='red')
```

Out[58]:
```
[<matplotlib.lines.Line2D at 0x1c501df1460>]
```

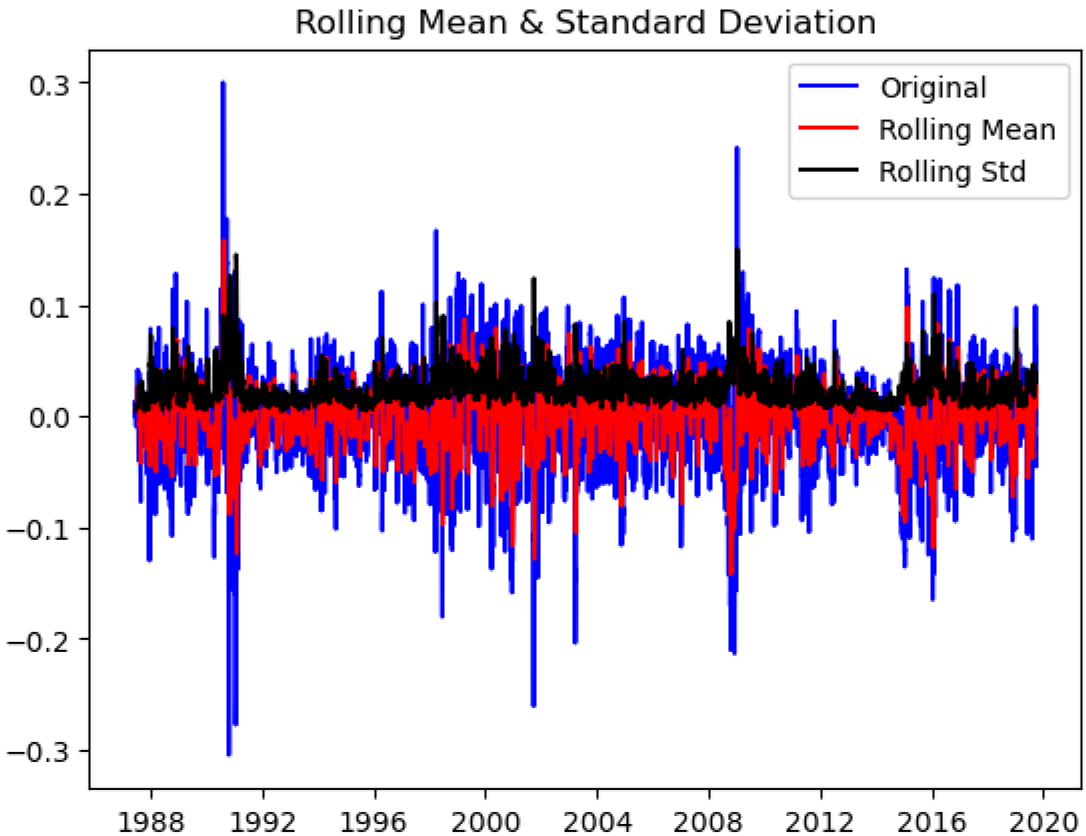


In [60]:
```
# Differencing
ts_log_ma_diff = ts_log - moving_avg
ts_log_ma_diff.head(12)
```

Out[60]:

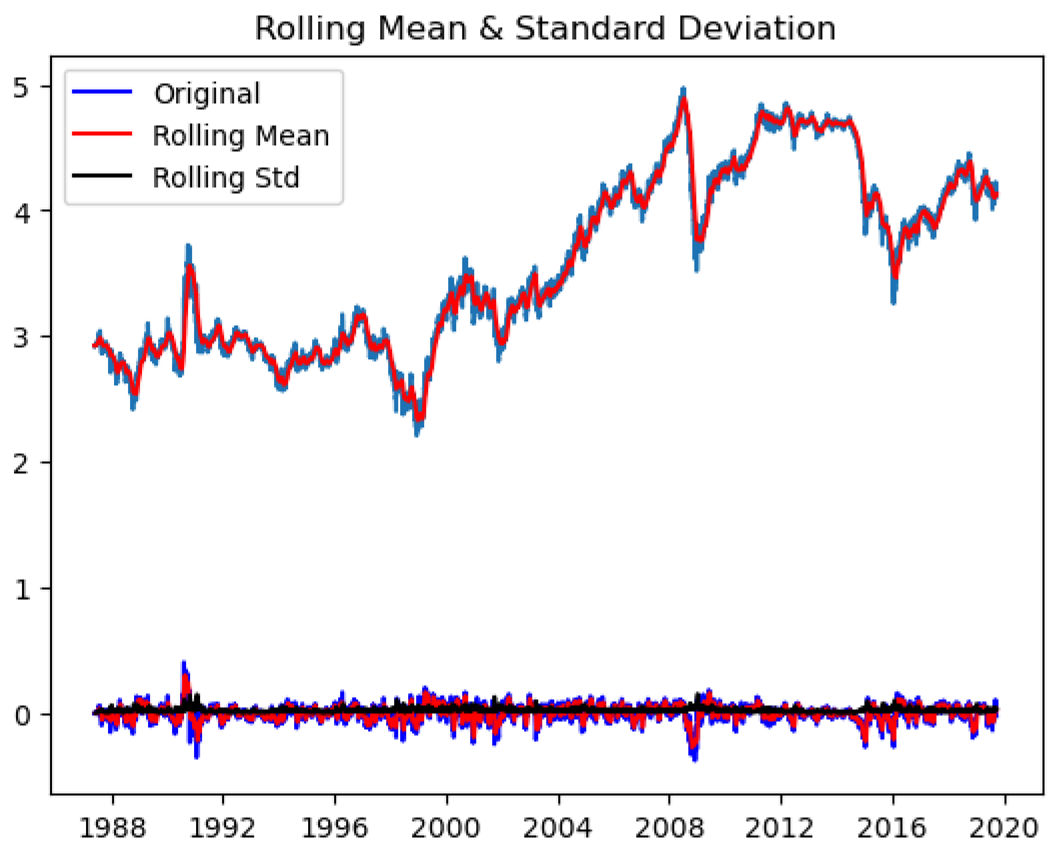
ds	y
1987-05-20	NaN
1987-05-21	NaN
1987-05-22	NaN
1987-05-25	NaN
1987-05-26	NaN
1987-05-27	NaN
1987-05-28	NaN
1987-05-29	NaN
1987-06-01	NaN
1987-06-02	NaN
1987-06-03	NaN
1987-06-04	0.008298

In [61]:
```
ts_log_ma_diff.dropna(inplace=True)
test_stationarity(ts_log_ma_diff)
```




```
Results of Dickey-Fuller Test:
Test Statistic                -1.879553e+01
p-value                        2.023172e-30
#Lags Used                     1.500000e+01
Number of Observations Used    8.189000e+03
Critical Value (1%)           -3.431149e+00
Critical Value (5%)           -2.861893e+00
Critical Value (10%)          -2.566958e+00
dtype: float64
```

In [62]:
```
# Exponentially weighted moving average
expwighted_avg = ts_log.ewm(halflife=12).mean()

plt.plot(ts_log)
plt.plot(expwighted_avg, color='red')
ts_log_ewma_diff = ts_log - expwighted_avg
test_stationarity(ts_log_ewma_diff)
```



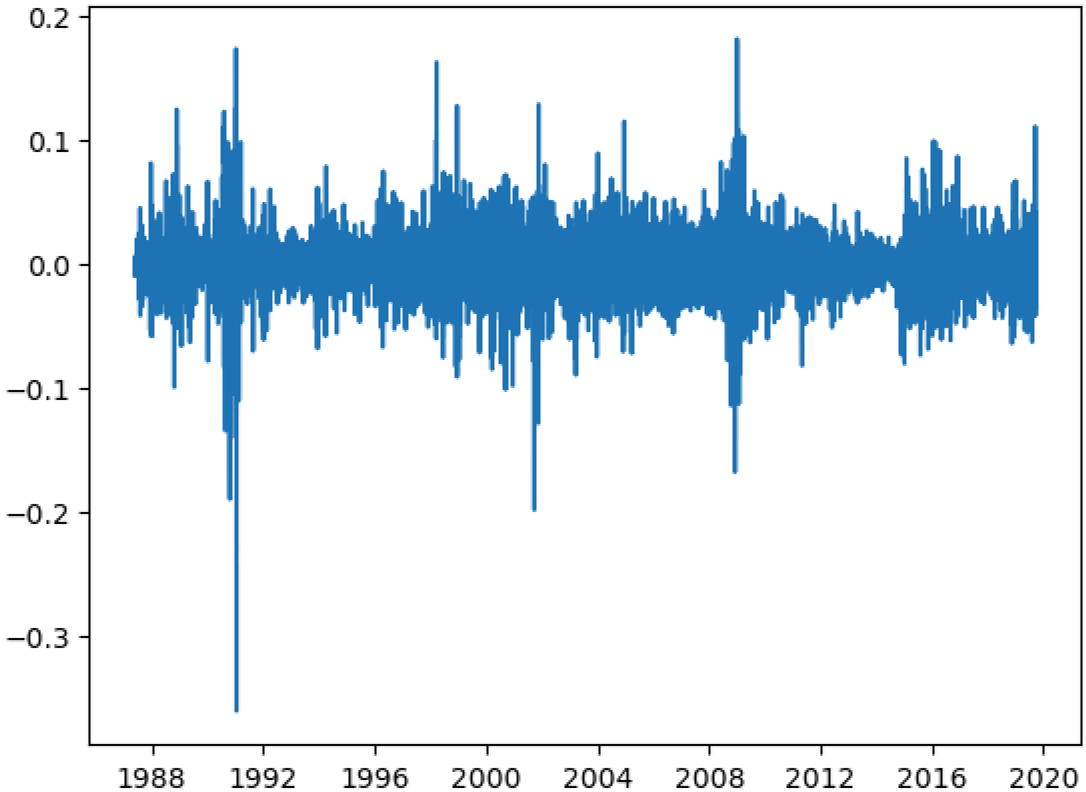

```
Results of Dickey-Fuller Test:
Test Statistic                -1.283031e+01
p-value                        5.892793e-24
#Lags Used                     1.500000e+01
Number of Observations Used    8.200000e+03
Critical Value (1%)           -3.431148e+00
Critical Value (5%)           -2.861893e+00
Critical Value (10%)          -2.566958e+00
dtype: float64
```

Step 8 - ARIMA models

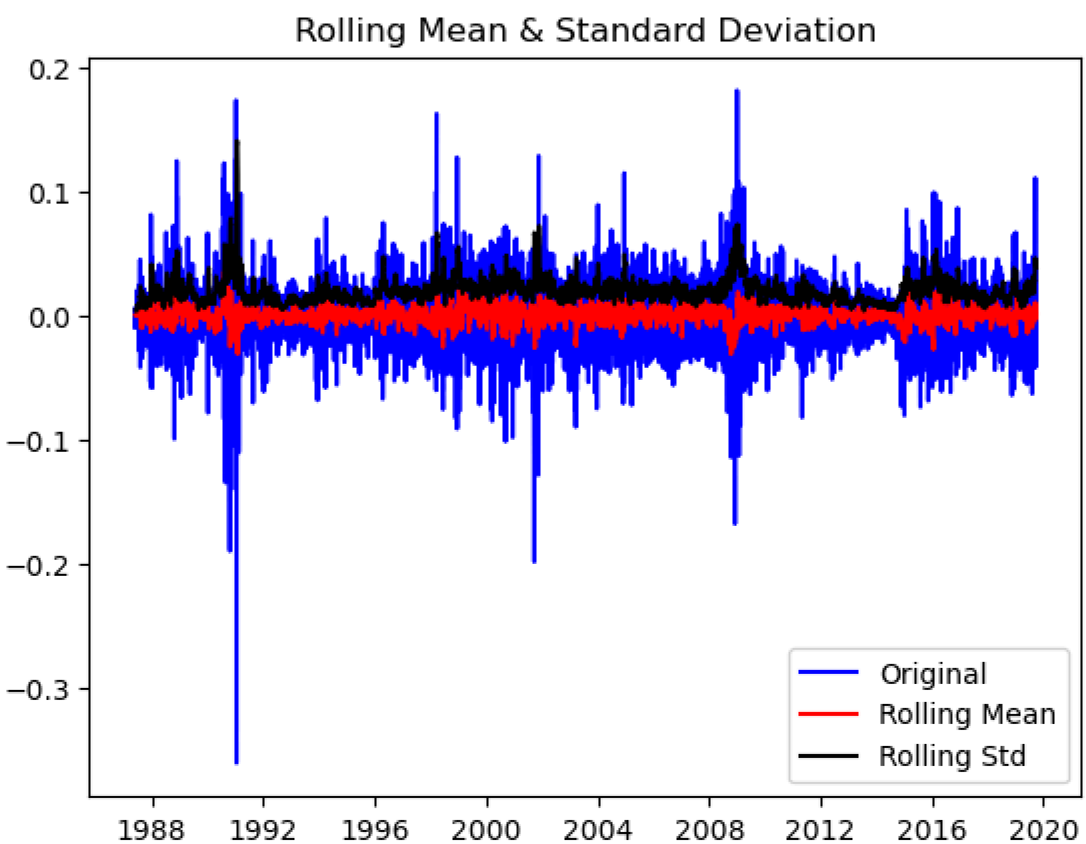
In [67]:

```
ts_log_diff = ts_log - ts_log.shift()
plt.plot(ts_log_diff)
```

Out[67]: [<matplotlib.lines.Line2D at 0x1c501697da0>]



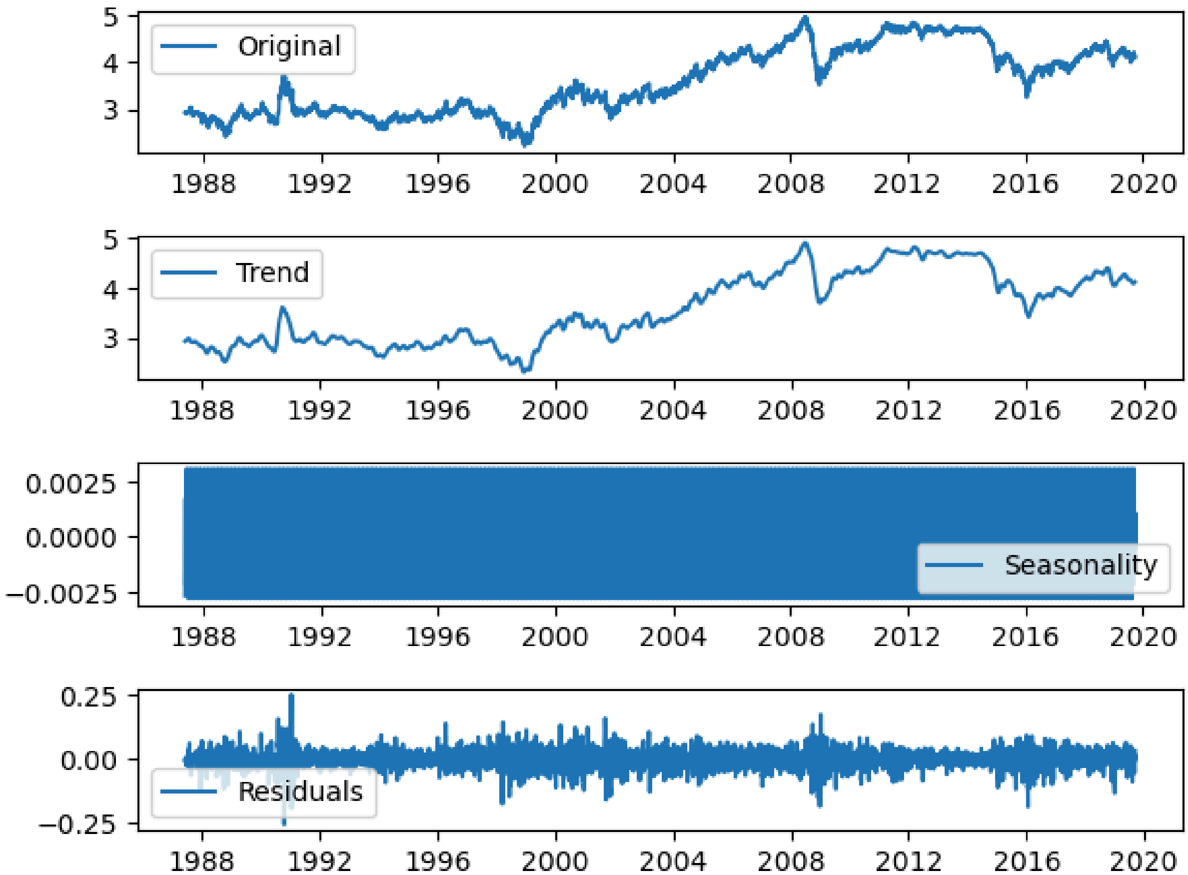

```
In [68]: ts_log_diff.dropna(inplace=True)
         test_stationarity(ts_log_diff)
```



```
Results of Dickey-Fuller Test:
Test Statistic                   -21.500963
p-value                            0.000000
#Lags Used                        14.000000
Number of Observations Used     8200.000000
Critical Value (1%)               -3.431148
Critical Value (5%)               -2.861893
Critical Value (10%)              -2.566958
dtype: float64
```

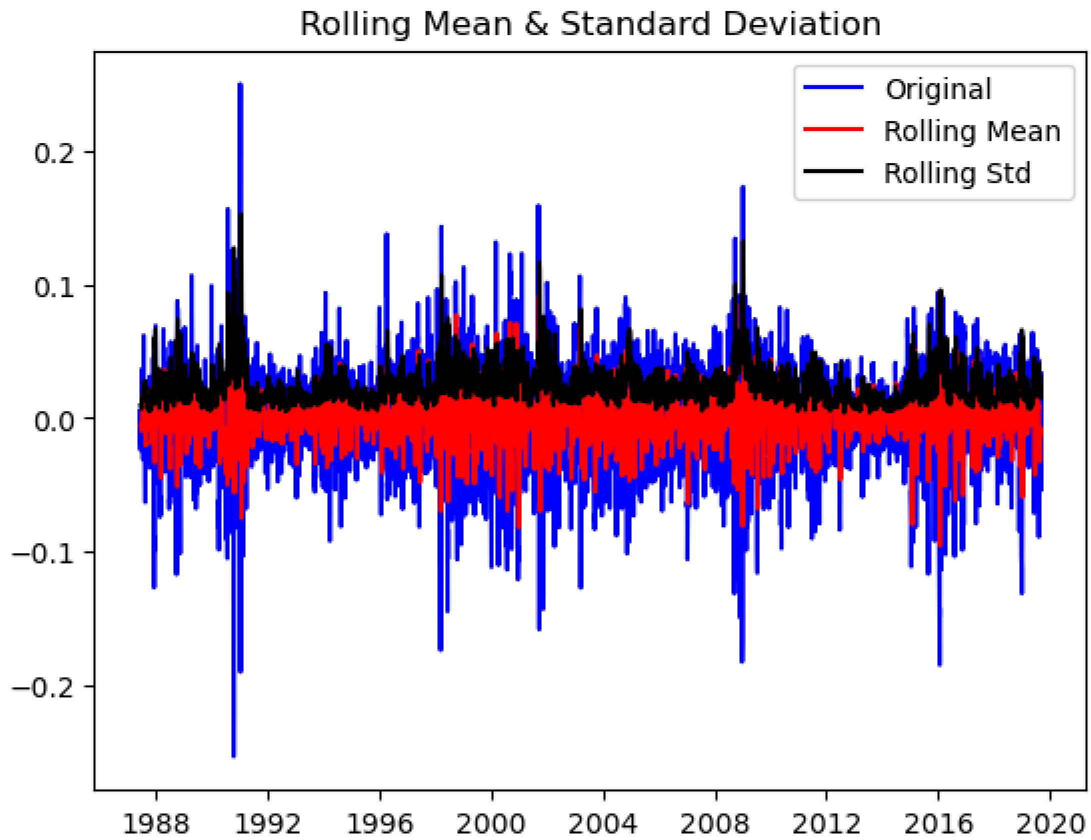
In [76]:

```python
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(ts_log, period = 30)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(ts_log, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



In [78]:

```
ts_log_decompose = residual
ts_log_decompose.dropna(inplace=True)
test_stationarity(ts_log_decompose)
```




```
Results of Dickey-Fuller Test:
```

```
---------------------------------------------------------------------------
DateParseError                            Traceback (most recent call last)
File C:\ProgramData\anaconda3\Lib\site-packages\pandas\core\indexes\datetimes.py:60
3, in DatetimeIndex.get_loc(self, key)
    602 try:
--> 603     parsed, reso = self._parse_with_reso(key)
    604 except (ValueError, pytz.NonExistentTimeError) as err:

File C:\ProgramData\anaconda3\Lib\site-packages\pandas\core\indexes\datetimes.py:55
9, in DatetimeIndex._parse_with_reso(self, label)
    558 def _parse_with_reso(self, label: str):
--> 559     parsed, reso = super()._parse_with_reso(label)
    561     parsed = Timestamp(parsed)

File C:\ProgramData\anaconda3\Lib\site-packages\pandas\core\indexes\datetimelike.py:
293, in DatetimeIndexOpsMixin._parse_with_reso(self, label)
    291     label = str(label)
--> 293 parsed, reso_str = parsing.parse_datetime_string_with_reso(label, freqstr)
    294 reso = Resolution.from_attrname(reso_str)

File parsing.pyx:442, in pandas._libs.tslibs.parsing.parse_datetime_string_with_reso
()

File parsing.pyx:666, in pandas._libs.tslibs.parsing.dateutil_parse()

DateParseError: Unknown datetime string format, unable to parse: y

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In[78], line 3
      1 ts_log_decompose = residual
      2 ts_log_decompose.dropna(inplace=True)
----> 3 test_stationarity(ts_log_decompose)

Cell In[44], line 18, in test_stationarity(ts)
     16 #Perform Dickey-Fuller test:
     17 print('Results of Dickey-Fuller Test:')
---> 18 dftest = adfuller(ts['y'], autolag='AIC')
     19 dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags U
sed','Number of Observations Used'])
     20 for key,value in dftest[4].items():

File C:\ProgramData\anaconda3\Lib\site-packages\pandas\core\series.py:1121, in Serie
s.__getitem__(self, key)
   1118     return self._values[key]
   1120 elif key_is_scalar:
-> 1121     return self._get_value(key)
   1123 # Convert generator to list before going through hashable part
   1124 # (We will iterate through the generator there to check for slices)
   1125 if is_iterator(key):

File C:\ProgramData\anaconda3\Lib\site-packages\pandas\core\series.py:1237, in Serie
s._get_value(self, label, takeable)
   1234     return self._values[label]
   1236 # Similar to Index.get_value, but we do not fall back to positional
```

```
-> 1237 loc = self.index.get_loc(label)
   1239 if is_integer(loc):
   1240     return self._values[loc]

File C:\ProgramData\anaconda3\Lib\site-packages\pandas\core\indexes\datetimes.py:60
5, in DatetimeIndex.get_loc(self, key)
    603     parsed, reso = self._parse_with_reso(key)
    604 except (ValueError, pytz.NonExistentTimeError) as err:
--> 605     raise KeyError(key) from err
    606 self._disallow_mismatched_indexing(parsed)
    608 if self._can_partial_date_slice(reso):

KeyError: 'y'
```

In [ ]:

```
model = ARIMA(ts_log, order=(2, 1, 2))
results_ARIMA = model.fit(disp=-1)
plt.plot(ts_log_diff)
plt.plot(results_ARIMA.fittedvalues, color='red')
# plt.title('RSS: %.4f'% sum((results_ARIMA.fittedvalues-ts_log_diff)**2))
```

In [ ]: