In [1]:
```
'''
Ignore Warnings to prevent cluttering the output
'''
import warnings
warnings.filterwarnings('ignore')

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm

'''
Importing all functions from scipy.stats for statistical analysis
'''
from scipy.stats import *

'''
Importing timedelta and datetime for handling time-related operations
'''
from datetime import timedelta
import datetime

'''
This is to enable inline plotting in matplotlib
'''
%matplotlib inline

'''
This line is to set the seaborn plot style to dark
which is a built-in style in seaborn
'''
sns.set_style('dark')
```

This code snippet performs several important tasks to enable data analysis and visualization:

1. **Ignoring Warnings**: The `warnings.filterwarnings('ignore')` line ensures that warning messages are suppressed, which helps in keeping the output clean and focused.

2. **Importing Libraries**:
    - `pandas` : Used for data manipulation and analysis.
    - `matplotlib.pyplot` : Enables plotting functionalities.
    - `seaborn` : Provides high-level interface for creating informative statistical graphics.
    - `statsmodels.api` : Used for statistical analysis and modeling.
3. **Importing Functions**:
    - The line `from scipy.stats import *` imports all functions from the `scipy.stats` module, facilitating statistical analysis.
4. **Handling Time-related Operations**:
    - `timedelta` from `datetime` module: Used for arithmetic operations on dates and times.
    - `datetime` : Provides classes for manipulating dates and times.
5. **Enabling Inline Plotting**: `%matplotlib inline` command ensures that plots generated using `matplotlib` are displayed inline within Jupyter notebooks or IPython environments.
6. **Setting Plot Style**:
    - `sns.set_style('dark')` : Sets the default style for seaborn plots to a dark background, enhancing visual appeal and readability of plots.

In [15]:
```
# Reading the dataset from the provided URL using pandas
# This dataset contains daily Covid-19 data for different states in India
train = pd.read_csv('https://api.covid19india.org/csv/latest/state_wise_daily.csv')

# Converting the 'Date' column to datetime format for better handling of dates
train['Date'] = pd.to_datetime(train['Date'], format="%d-%b-%y")

# Displaying the last few rows of the dataset to verify the changes
train.tail()
```

Out[15]:

	Date	Date_YMD	Status	TT	AN	AP	AR	AS	BR	CH	...	PB	RJ	SK	TN	TG	TR	UP	UT	WB	UN
1786	2021-10-30	2021-10-30	Recovered	14672	1	535	10	371	2	2	...	27	1	3	1172	191	8	9	6	880	0
1787	2021-10-30	2021-10-30	Deceased	445	0	2	0	4	0	0	...	1	0	0	14	1	0	0	0	13	0
1788	2021-10-31	2021-10-31	Confirmed	12907	0	385	1	212	8	5	...	26	2	21	1009	121	12	6	5	914	0
1789	2021-10-31	2021-10-31	Recovered	13152	0	675	9	236	9	3	...	25	2	8	1183	183	2	6	9	913	0
1790	2021-10-31	2021-10-31	Deceased	251	0	4	0	1	0	0	...	1	0	1	19	1	0	0	0	15	0

5 rows × 42 columns

1. **We read the dataset** from the provided URL, which contains daily Covid-19 data for different states in India, and store it in the variable `train`.
2. We **convert** the '`Date`' column in the dataset to *datetime* format using `pd.to_datetime()` for better handling of dates.
3. We display the **last few rows** of the dataset using `train.tail()` to verify the changes made.

In [16]:
```
# List of state-wise columns to be dropped as we are predicting total cases ('TT')
cols = ['AN','AP',      'AR',   'AS',   'BR',   'CH',   'CT',   'DD',   'DL',   'DN',   'GA',   'GJ',   'HP',   'HR',   'JH',

# Dropping state-wise columns from the dataset
train.drop(cols, axis=1, inplace=True)

# Setting the index of the dataframe to 'Status' column
train = train.set_index('Status')

# Dropping 'Recovered' and 'Deceased' rows as we are focusing on total cases
train.drop(['Recovered', 'Deceased'], inplace=True)

# Resetting the index after dropping unnecessary rows
train = train.reset_index()

# Dropping the 'Status' column as it is no longer needed for analysis
train.drop(["Status"], axis=1, inplace=True)
train.tail()
```

```
# Copying data from the 'train' DataFrame to the 'train_df' DataFrame to preserve the original dataset
train_df = train
train_df.head()
```

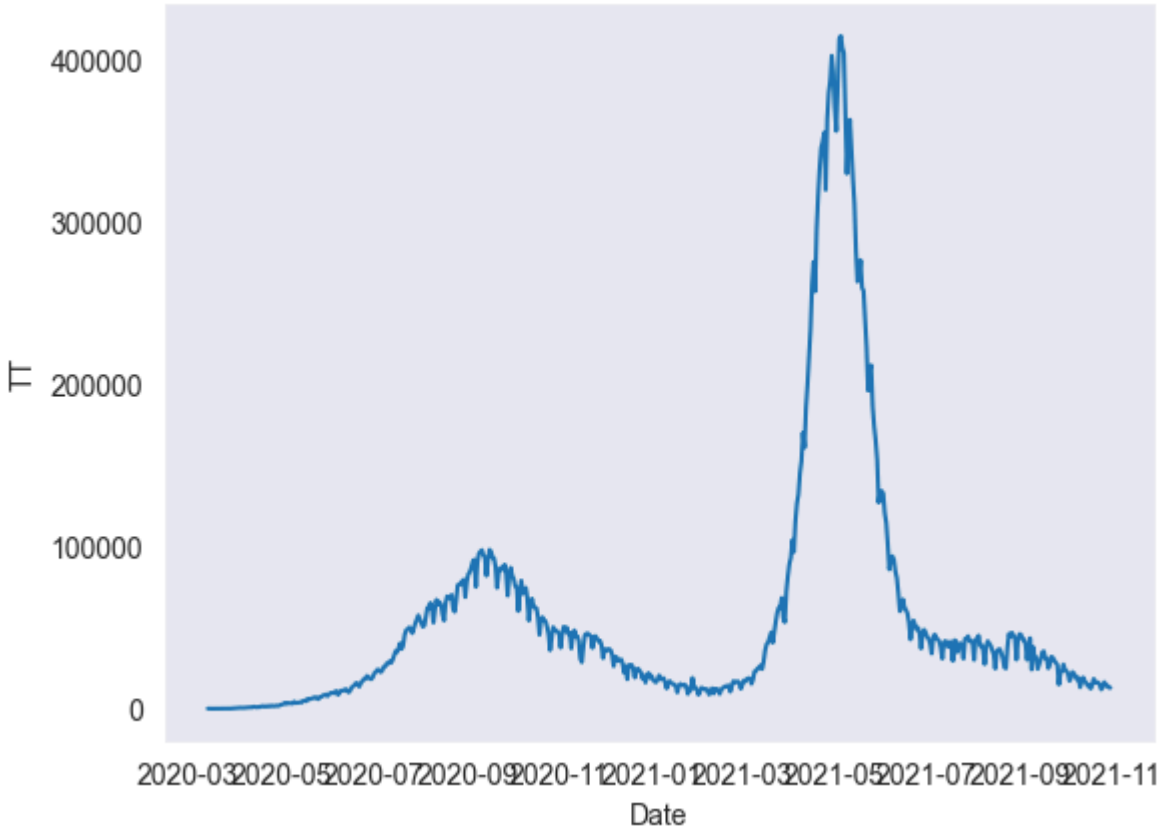
Out[16]:

	Date	Date_YMD	TT	UN
0	2020-03-14	2020-03-14	81	0
1	2020-03-15	2020-03-15	27	0
2	2020-03-16	2020-03-16	15	0
3	2020-03-17	2020-03-17	11	0
4	2020-03-18	2020-03-18	37	0

1. **We define a list** `cols` containing the names of state-wise columns to be dropped.
2. We use `drop()` method along `axis=1` to **drop the specified columns** from the `train` dataframe in-place.
3. We **set the index of the dataframe** to the 'Status' column, assuming it contains categories like 'Confirmed', 'Recovered', and 'Deceased'.
4. We **drop rows corresponding to** 'Recovered' and 'Deceased' status to focus on the total cases ('TT').
5. We **reset the index to default** after dropping rows to avoid any index inconsistencies.
6. We **drop the** `Status` column as it is no longer needed for further analysis.
7. Finally, **we create a copy of** `train` to `train_df` to preserve the original data and be able to work on the dataframe.

In [17]:
```
sns.lineplot(x="Date", y="TT",legend = 'full' , data=train_df)
```

Out[17]:
```
<Axes: xlabel='Date', ylabel='TT'>
```



In [18]:
```
# Set 'Date' column as the index column as forecasting will be done for this column
train_df.set_index('Date', inplace=True)

# Convert 'TT' column to float for statistical calculations
train_df['TT'] = train_df['TT'].astype(float)

# Display the first few rows of the DataFrame
train_df.head()
```

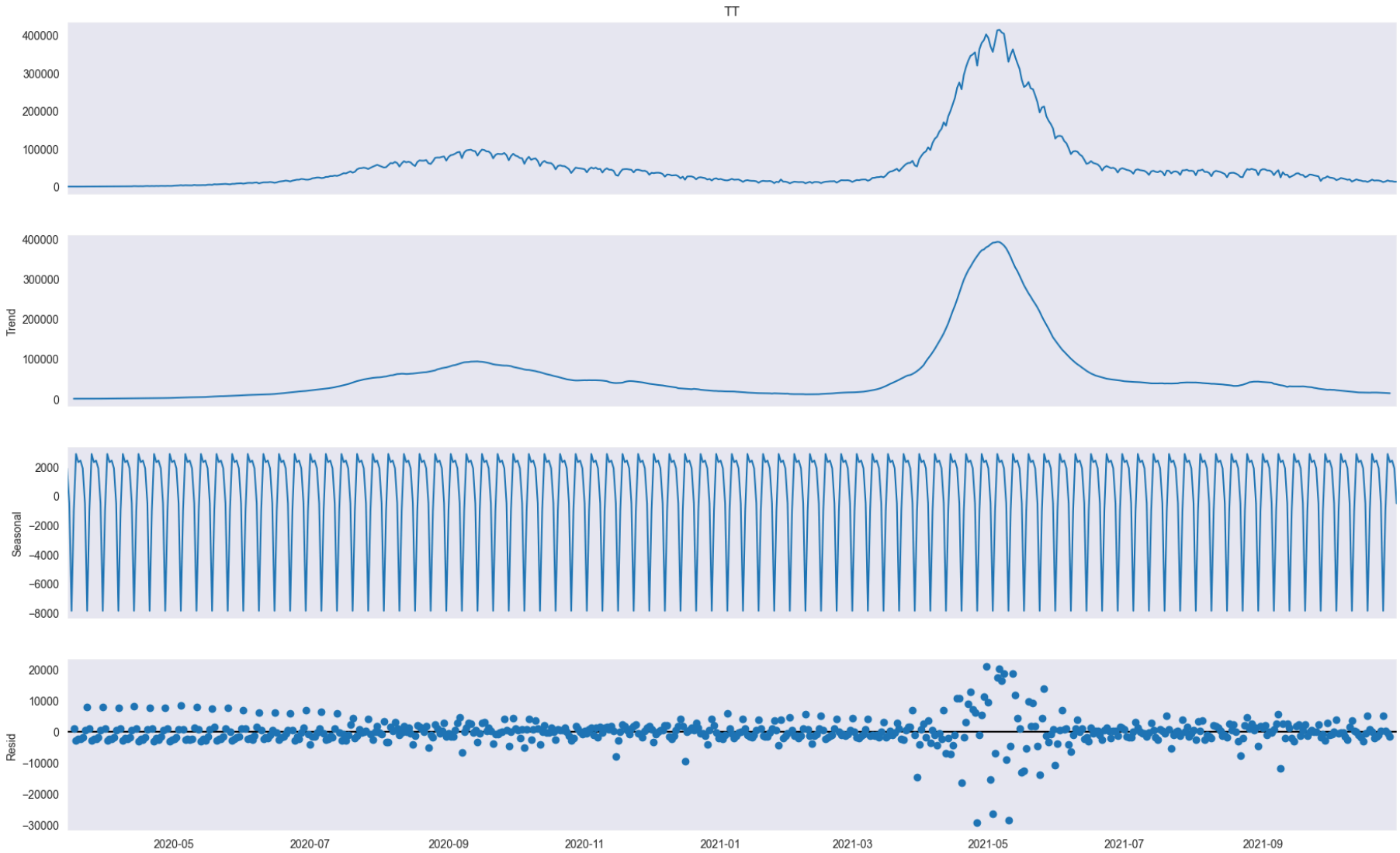
Out[18]:

Date	Date_YMD	TT	UN
2020-03-14	2020-03-14	81.0	0
2020-03-15	2020-03-15	27.0	0
2020-03-16	2020-03-16	15.0	0
2020-03-17	2020-03-17	11.0	0
2020-03-18	2020-03-18	37.0	0

In [19]:

```
# Import the necessary library for seasonal decomposition
from statsmodels.tsa.seasonal import seasonal_decompose

# Perform seasonal decomposition on the 'TT' column using an additive model
result = seasonal_decompose(train_df['TT'], model='additive')

# Create a figure object for plotting
fig = plt.figure()

# Plot the decomposed time series components (trend, seasonality, residual)
fig = result.plot()

# Set the size of the figure
fig.set_size_inches(20, 12)
```

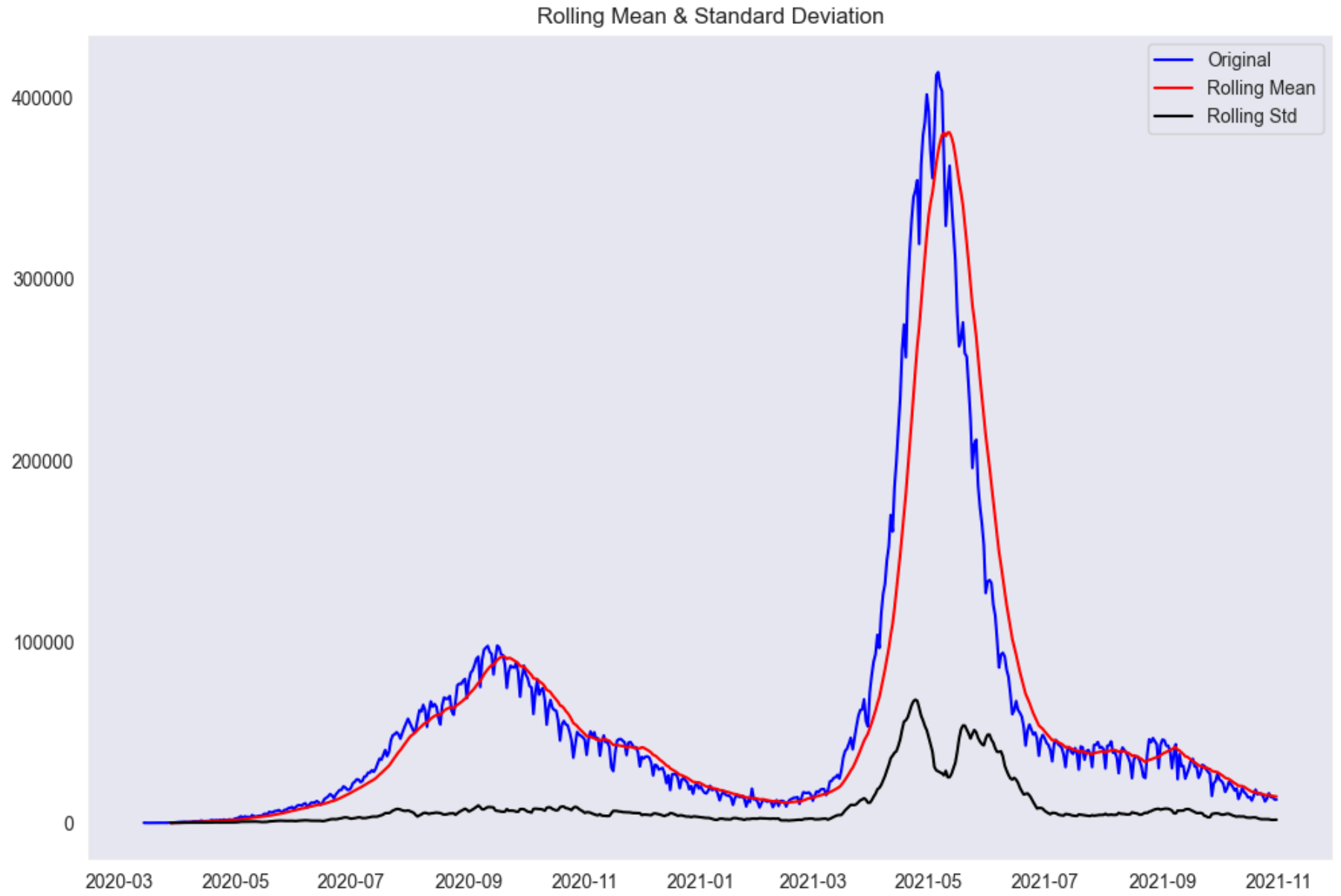
```
<Figure size 640x480 with 0 Axes>
```



The **trend** graph appears stationary. Time Series forecasting can be done on stationary data and hence no further processing to make it stationery should be required but we can check if it is stationary using the **Dickey-Fuller Test**

In [21]:
```
# Import necessary libraries
from statsmodels.tsa.stattools import adfuller  # Import Augmented Dickey-Fuller unit root test
import pandas as pd
```

```python
import matplotlib.pyplot as plt

# Define a function to test stationarity of a time series
def test_stationarity(timeseries, window=15, cutoff=0.01):
    # Calculate rolling mean and rolling standard deviation
    rolmean = timeseries.rolling(window).mean()
    rolstd = timeseries.rolling(window).std()

    # Plot original time series, rolling mean, and rolling standard deviation
    fig = plt.figure(figsize=(12, 8))
    orig = plt.plot(timeseries, 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()

    # Perform Augmented Dickey-Fuller test
    print('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic', 'p-value', '#Lags Used', 'Number of Observations Used'])

    # Add critical values to the output
    for key, value in dftest[4].items():
        dfoutput['Critical Value (%s)' % key] = value

    # Check p-value against cutoff for stationarity
    pvalue = dftest[1]
    if pvalue < cutoff:
        print('p-value = %.4f. The series is likely stationary.' % pvalue)
    else:
        print('p-value = %.4f. The series is likely non-stationary.' % pvalue)

    # Print Dickey-Fuller test results
    print(dfoutput)

# Example usage:
# Assuming train_df['TT'] is the time series to be tested for stationarity
test_stationarity(train_df['TT'])
```


Rolling Mean & Standard Deviation
Original
Rolling Mean
Rolling Std
400000
300000
200000
100000
0
2020-03
2020-05
2020-07
2020-09
2020-11
2021-01
2021-03
2021-05
2021-07
2021-09
2021-11

```
Results of Dickey-Fuller Test:
p-value = 0.0201. The series is likely non-stationary.
Test Statistic                  -3.198105
p-value                          0.020095
#Lags Used                      19.000000
Number of Observations Used    577.000000
Critical Value (1%)             -3.441734
Critical Value (5%)             -2.866562
Critical Value (10%)            -2.569445
dtype: float64
```

Let's break down the provided code and explain each part:

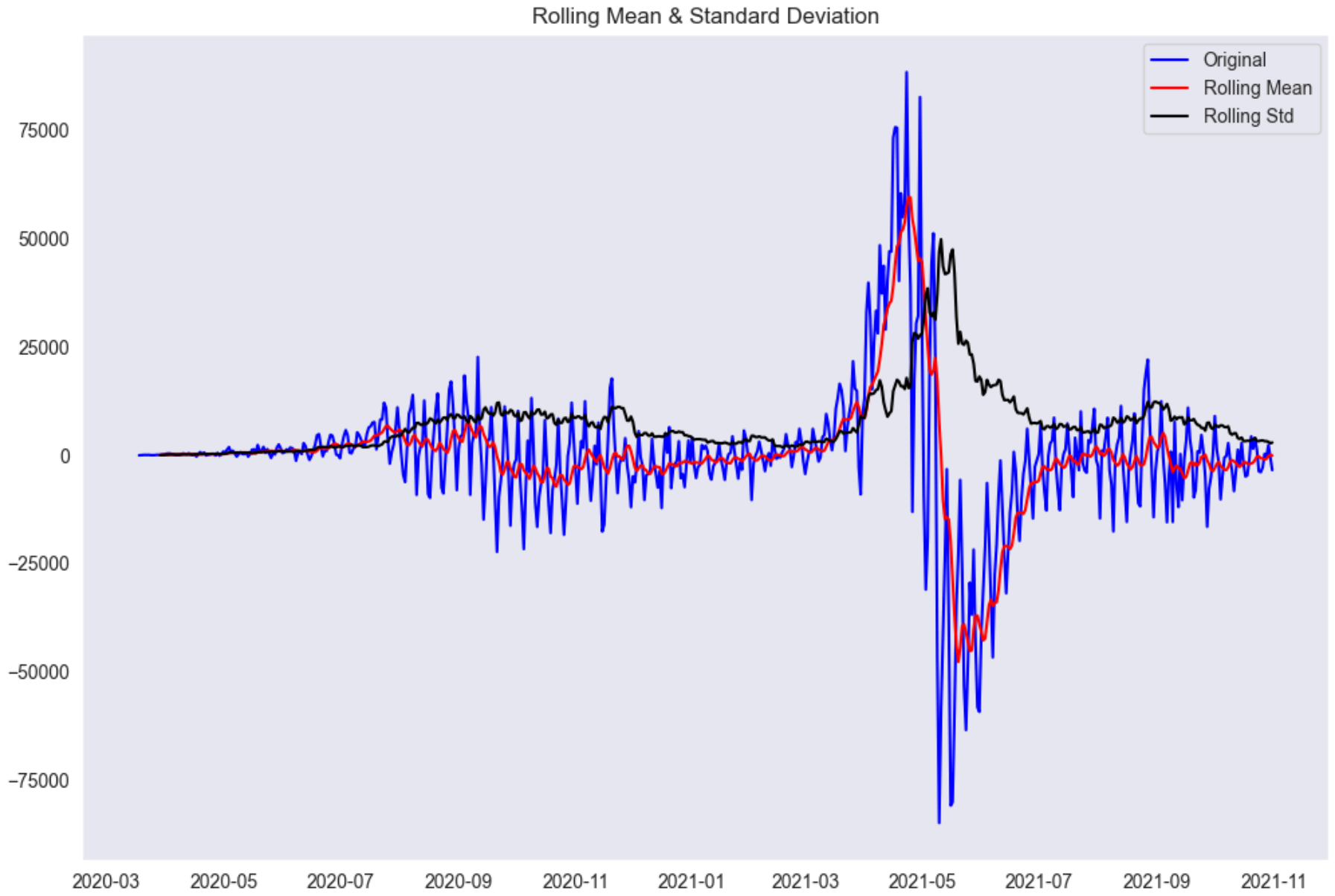
1. **Import Necessary Libraries**:
   - `from statsmodels.tsa.stattools import adfuller`: Imports the Augmented Dickey-Fuller (ADF) unit root test from statsmodels for testing stationarity.
   - `import pandas as pd`: Imports the pandas library for data manipulation.
   - `import matplotlib.pyplot as plt`: Imports matplotlib for plotting.
2. **Define `test_stationarity` Function**:
   - This function tests the stationarity of a given time series using the Augmented Dickey-Fuller (ADF) test and visualizes rolling statistics (mean and standard deviation).
   - Parameters:
     - `timeseries`: The input time series data to be tested for stationarity.
     - `window=15`: The size of the rolling window for computing the rolling mean and standard deviation (default is 15).
     - `cutoff=0.01`: The significance level for the p-value to determine stationarity (default is 0.01).
3. **Inside `test_stationarity` Function**:
   - Computes the rolling mean (`rolmean`) and rolling standard deviation (`rolstd`) using the specified window size.
   - Plots the original time series, rolling mean, and rolling standard deviation in a single plot for visual inspection.
   - Performs the Augmented Dickey-Fuller (ADF) test (`adfuller`) on the time series to check for stationarity.
   - Prints the results of the ADF test including the test statistic, p-value, critical values, and conclusion about stationarity based on the p-value and cutoff.
4. **Example Usage**:

- Calls the `test_stationarity` function with an example time series data `train_df['TT']` to test for stationarity.

The key steps in the `test_stationarity` function are calculating rolling statistics, plotting the original time series and rolling statistics, conducting the ADF test, and printing the results. The example usage at the end demonstrates how to apply this function to analyze the stationarity of a specific time series (`train_df['TT']` in this case).

1. Calling the function gives below result , where we can observe the huge gap between orignal data and mean,std
2. Also the `pvalue` is `0.020095` which is not so good and hence , the output says **"The series is likely non-stationary."**

In [22]:
```
# Using differencing to make the time series stationary
# Shift the 'TT' column of the DataFrame by 4 places and take the difference

# Calculate the first difference
first_diff = train_df['TT'] - train_df['TT'].shift(4)  # Shift by 4 periods (quarterly data)
first_diff = first_diff.dropna(inplace=False)  # Drop the NaN values created by differencing

# Test stationarity of the differenced time series
test_stationarity(first_diff, window=12)  # Using a rolling window of 12 for mean and std calculation
```

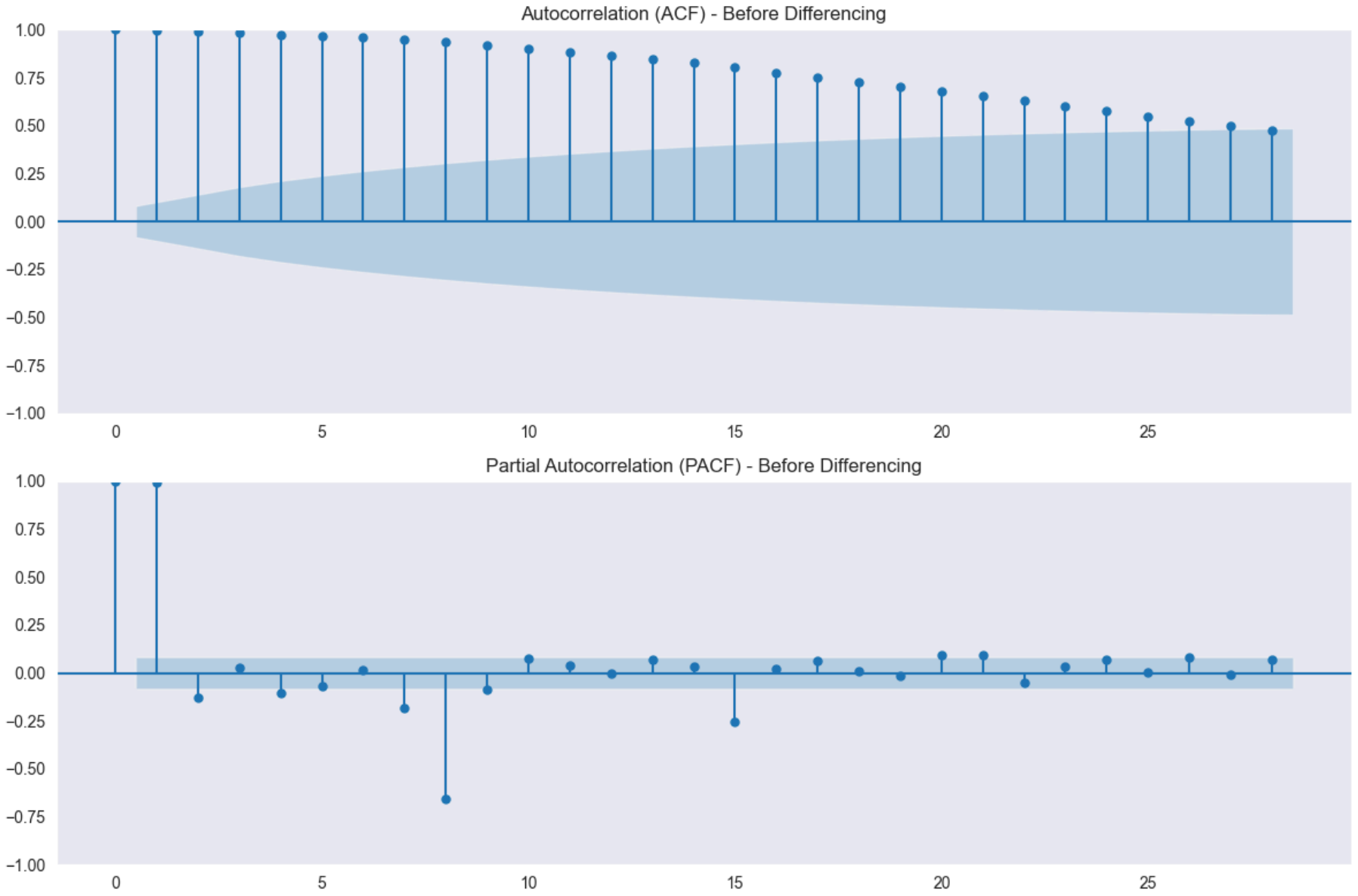

Rolling Mean & Standard Deviation
Original
Rolling Mean
Rolling Std
75000
50000
25000
0
−25000
−50000
−75000
2020-03
2020-05
2020-07
2020-09
2020-11
2021-01
2021-03
2021-05
2021-07
2021-09
2021-11

```
Results of Dickey-Fuller Test:
p-value = 0.0000. The series is likely stationary.
Test Statistic                -6.240639e+00
p-value                        4.706636e-08
#Lags Used                     1.900000e+01
Number of Observations Used    5.730000e+02
Critical Value (1%)           -3.441814e+00
Critical Value (5%)           -2.866597e+00
Critical Value (10%)          -2.569463e+00
dtype: float64
```

1. Calculate the **first difference** of the `TT` column in the DataFrame `train_df` by **subtracting the value** at each time point from the value at the same time point shifted by 4 periods. This helps remove seasonality if the data is quarterly.
2. Drop the `NaN` values resulting from the differencing operation using the `dropna` method with `inplace=False` to keep the modified DataFrame.
3. Test the **stationarity** of the differenced time series (`first_diff`) using the `test_stationarity` function defined earlier, with a **rolling window size** of `12` for calculating the **mean** and **standard deviation**. The choice of window size can be adjusted based on the frequency of the data and the desired level of smoothing.

In [23]:
```python
# Import necessary libraries
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Create a figure object with two subplots for autocorrelation and partial autocorrelation plots
fig = plt.figure(figsize=(12, 8))
ax1 = fig.add_subplot(211)  # Add subplot for autocorrelation
ax2 = fig.add_subplot(212)  # Add subplot for partial autocorrelation

# Plot autocorrelation (ACF) for the 'TT' column of the DataFrame before differencing
fig = sm.graphics.tsa.plot_acf(train_df['TT'], ax=ax1)  # Default value of lag used
ax1.set_title('Autocorrelation (ACF) - Before Differencing')

# Plot partial autocorrelation (PACF) for the 'TT' column of the DataFrame before differencing
fig = sm.graphics.tsa.plot_pacf(train_df['TT'], ax=ax2)  # Default value of lag used
ax2.set_title('Partial Autocorrelation (PACF) - Before Differencing')

# Show the plots
```

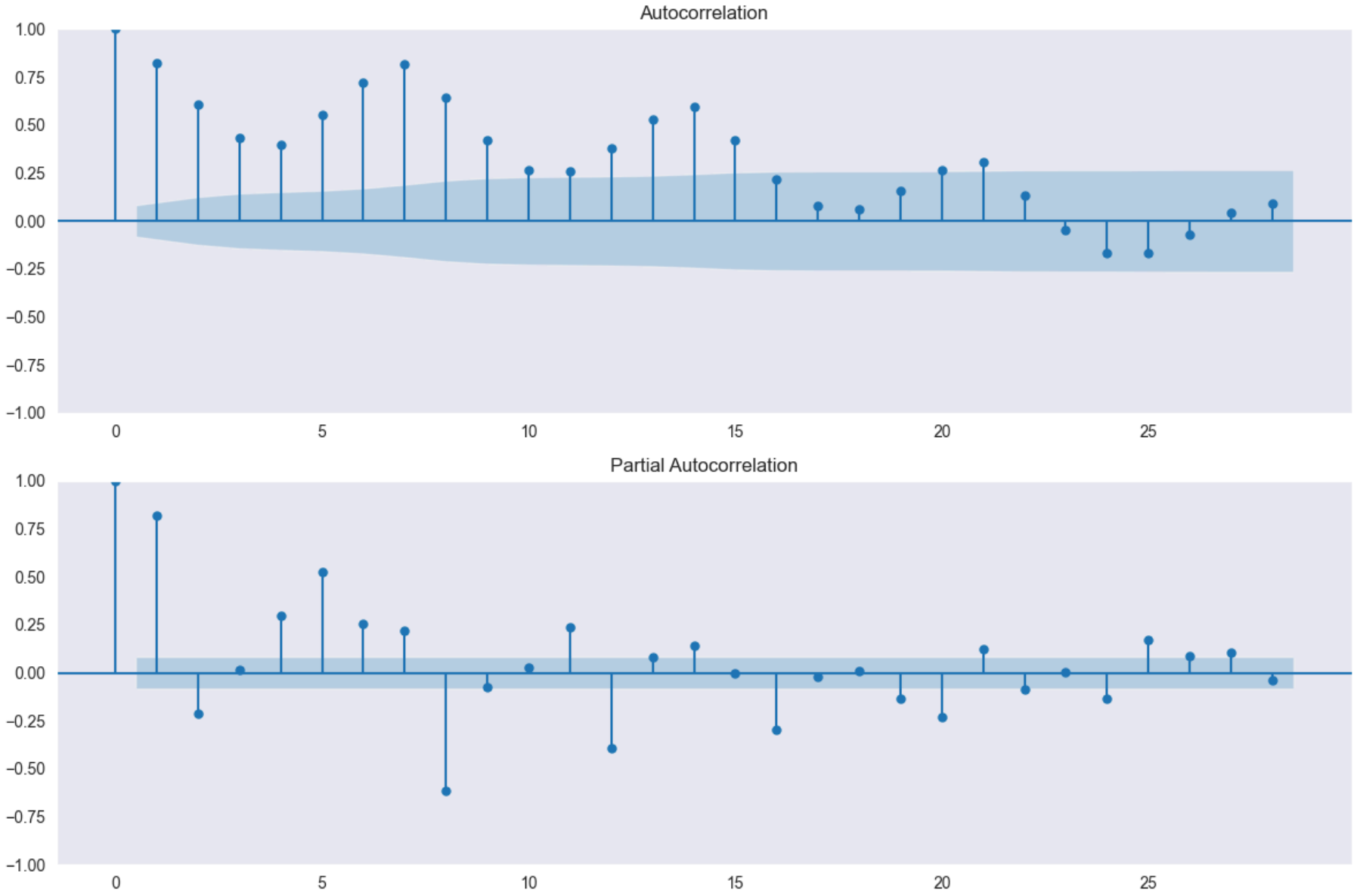
```
plt.tight_layout()
plt.show()
```



1. Import the necessary libraries (`statsmodels.api` for statistical modeling and `matplotlib.pyplot` for plotting).

2. Create a figure object (`fig`) with a size of 12 inches by 8 inches to hold the subplots for autocorrelation and partial autocorrelation plots.
3. Add two subplots (`ax1` and `ax2`) to the figure (`fig`) using `add_subplot` . These subplots will be used for autocorrelation and partial autocorrelation plots, respectively.
4. Plot the autocorrelation (ACF) of the 'TT' column in the DataFrame `train_df` using `sm.graphics.tsa.plot_acf` on `ax1` . The default value of lag is used to compute autocorrelation.
5. Set the title for the first subplot (`ax1`) as 'Autocorrelation (ACF) - Before Differencing'.
6. Plot the partial autocorrelation (PACF) of the 'TT' column in the DataFrame `train_df` using `sm.graphics.tsa.plot_pacf` on `ax2` . The default value of lag is used to compute partial autocorrelation.
7. Set the title for the second subplot (`ax2`) as 'Partial Autocorrelation (PACF) - Before Differencing'.
8. Use `plt.tight_layout()` to adjust the spacing between subplots for better readability.
9. Finally, display the plots using `plt.show()` .

In [25]:
```
# Create a new figure object with two subplots for autocorrelation and partial autocorrelation plots
fig = plt.figure(figsize=(12, 8))  # Create a figure with specified size (12 inches wide by 8 inches tall)

# Add the first subplot for autocorrelation (ACF) to the figure
ax1 = fig.add_subplot(211)  # 1st subplot for autocorrelation

# Plot autocorrelation (ACF) for the differenced time series (first_diff) on the first subplot (ax1)
fig = sm.graphics.tsa.plot_acf(first_diff, ax=ax1)  # Default value of lag used

# Add the second subplot for partial autocorrelation (PACF) to the figure
ax2 = fig.add_subplot(212)  # 2nd subplot for partial autocorrelation

# Plot partial autocorrelation (PACF) for the differenced time series (first_diff) on the second subplot (ax2)
fig = sm.graphics.tsa.plot_pacf(first_diff, ax=ax2)  # Default value of lag used

# Show the plots
plt.tight_layout()  # Adjust spacing between subplots for better readability
plt.show()
```



1. Create a new figure object (`fig`) with a specified size of 12 inches wide by 8 inches tall for better visualization of subplots.
2. Add the first subplot (`ax1`) for autocorrelation (ACF) to the figure. The `211` notation means a grid of 2 rows and 1 column, with this subplot being the 1st subplot.

3. Plot the autocorrelation (ACF) for the differenced time series (`first_diff`) on the first subplot (`ax1`) using `sm.graphics.tsa.plot_acf` . The default lag values are used for computing autocorrelation.
4. Add the second subplot (`ax2`) for partial autocorrelation (PACF) to the figure. The `212` notation means a grid of 2 rows and 1 column, with this subplot being the 2nd subplot.
5. Plot the partial autocorrelation (PACF) for the differenced time series (`first_diff`) on the second subplot (`ax2`) using `sm.graphics.tsa.plot_pacf` . The default lag values are used for computing partial autocorrelation.
6. Use `plt.tight_layout()` to adjust the spacing between subplots for better readability.
7. Finally, display the plots using `plt.show()` .

In [27]:
```
import statsmodels.api as sm

# Define SARIMAX model parameters and fit the model
# We assume a constant average trend with growth (I = 1) and no moving average component (MA = 0)
sarimax_mod = sm.tsa.statespace.SARIMAX(train_df['TT'], trend='n', order=(14, 1, 0)).fit()

# Print the summary of the SARIMAX model
print(sarimax_mod.summary())
```

```
C:\Users\mailr\AppData\Local\Programs\Python\Python311\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarning:
No frequency information was provided, so inferred frequency D will be used.
  self._init_dates(dates, freq)
C:\Users\mailr\AppData\Local\Programs\Python\Python311\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarning:
No frequency information was provided, so inferred frequency D will be used.
  self._init_dates(dates, freq)
```

```
                               SARIMAX Results
==========================================================================================
Dep. Variable:                         TT   No. Observations:                  597
Model:               SARIMAX(14, 1, 0)   Log Likelihood               -5840.113
Date:                 Sun, 31 Mar 2024   AIC                          11710.226
Time:                         19:53:05   BIC                          11776.079
Sample:                     03-14-2020   HQIC                         11735.869
                          - 10-31-2021
Covariance Type:                   opg
==========================================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.0979      0.027     -3.625      0.000      -0.151      -0.045
ar.L2          0.0975      0.027      3.675      0.000       0.045       0.149
ar.L3          0.1132      0.024      4.741      0.000       0.066       0.160
ar.L4          0.0839      0.028      2.959      0.003       0.028       0.139
ar.L5          0.1381      0.026      5.284      0.000       0.087       0.189
ar.L6          0.2109      0.021      9.903      0.000       0.169       0.253
ar.L7          0.6451      0.022     28.725      0.000       0.601       0.689
ar.L8          0.0888      0.028      3.190      0.001       0.034       0.143
ar.L9         -0.1538      0.022     -6.987      0.000      -0.197      -0.111
ar.L10        -0.1248      0.025     -4.941      0.000      -0.174      -0.075
ar.L11        -0.0698      0.032     -2.207      0.027      -0.132      -0.008
ar.L12        -0.1642      0.028     -5.901      0.000      -0.219      -0.110
ar.L13        -0.1322      0.024     -5.417      0.000      -0.180      -0.084
ar.L14         0.1628      0.025      6.436      0.000       0.113       0.212
sigma2      1.923e+07   5.72e-10   3.36e+16      0.000    1.92e+07    1.92e+07
==========================================================================================
Ljung-Box (L1) (Q):                   0.07   Jarque-Bera (JB):              1502.95
Prob(Q):                              0.79   Prob(JB):                         0.00
Heteroskedasticity (H):               8.96   Skew:                            -0.63
Prob(H) (two-sided):                  0.00   Kurtosis:                        10.68
==========================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 2.22e+31. Standard errors may be unstable.
```

Based on the recurring correlation observed in both the autocorrelation function (ACF) and partial autocorrelation function (PACF), it is evident that our time series data exhibits a certain level of seasonality. Following a common rule in time series modeling, the choice of the SARIMAX

model is appropriate as it can effectively handle seasonality.

In terms of differencing, the decision is guided by the behavior of the original series. A model with no orders of differencing assumes the original series is stationary and mean-reverting. On the other hand, a model with one order of differencing implies a constant average trend in the original series, such as a random walk or SES-type model, with or without growth. Since our series demonstrates a constant average trend with growth, taking the differencing order (I) as 1 and the moving average order (MA) as 0 (I-1) is suitable.

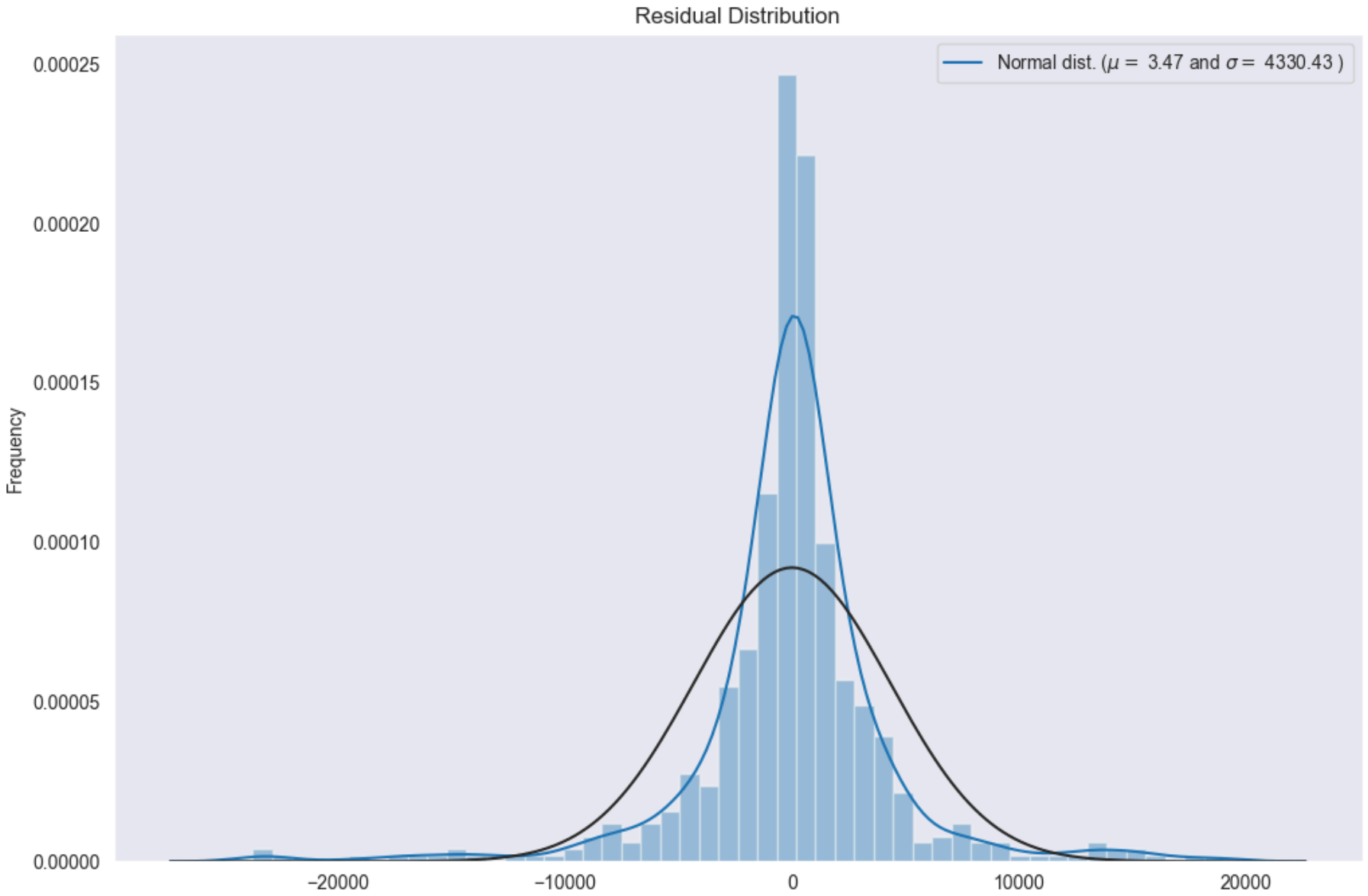
Therefore, we would set up a SARIMAX model with an integrated order (I) of 1 and a moving average order (MA) of 0 to capture the characteristics of our time series data appropriately, accounting for its seasonality and growth trend.

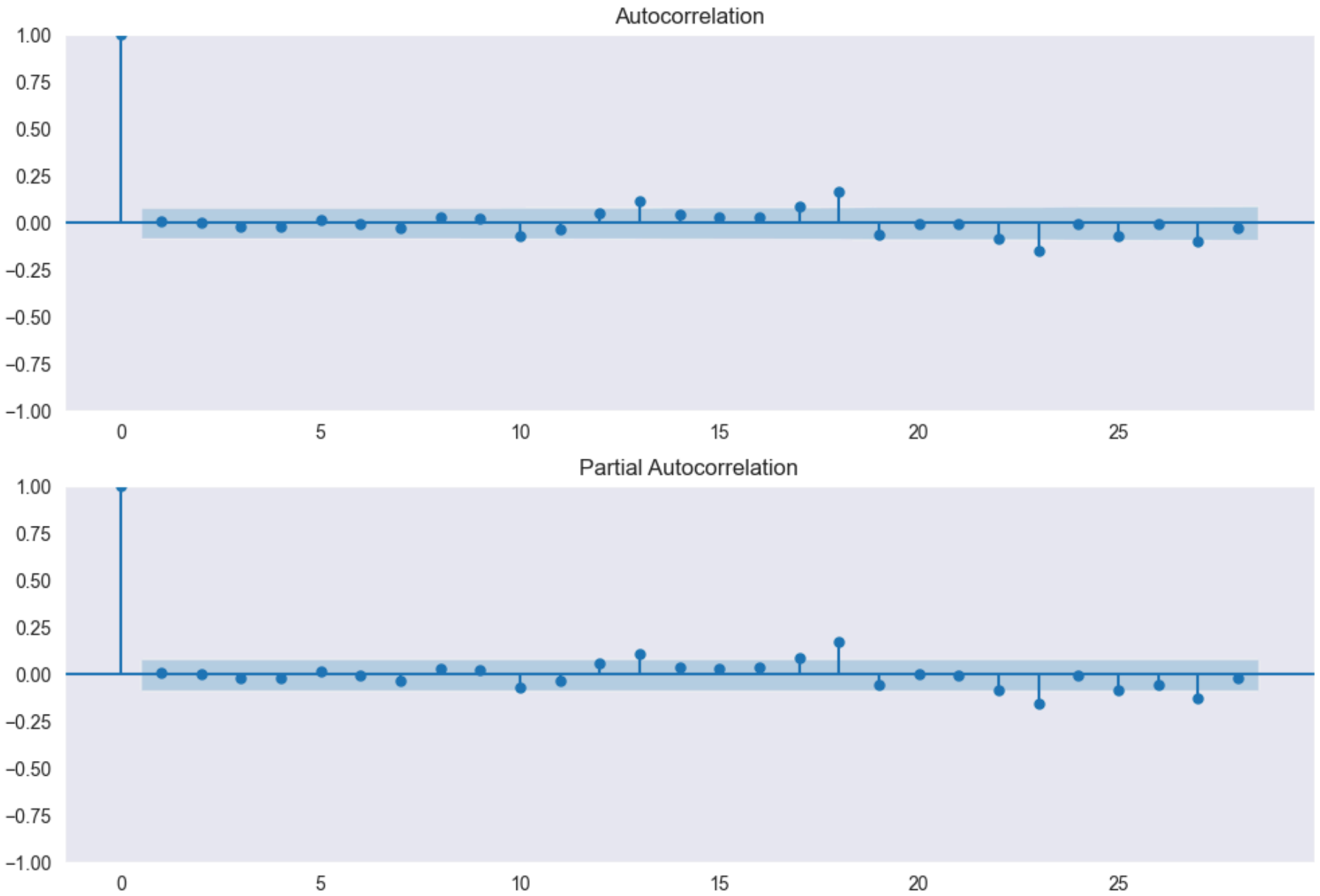
1. Import the necessary library `statsmodels.api` as `sm` to use SARIMAX models.
2. Define the SARIMAX model with the specified parameters:
   - `train_df['TT']`: Time series data to be used for modeling.
   - `trend='n'`: No trend component in the model (assumed constant average trend with growth).
   - `order=(14, 1, 0)`: SARIMAX model order parameters - (p, d, q).
     - `p = 14`: AutoRegressive (AR) order, representing the number of lag observations included in the model.
     - `d = 1`: Integrated (I) order, representing the number of times the differencing operation is performed to achieve stationarity.
     - `q = 0`: Moving Average (MA) order, representing the number of lagged forecast errors included in the model.
3. Fit the SARIMAX model to the data using the `.fit()` method.
4. Print the summary of the SARIMAX model using `.summary()` to view model statistics, parameter estimates, and diagnostic information.

In [29]:
```
import scipy.stats as stats
import seaborn as sns

# Get the residuals from the SARIMAX model
resid = sarimax_mod.resid

# Perform the normality test on the residuals using the Jarque-Bera test
jb_test = stats.normaltest(resid)
print(jb_test)

# Create a figure object for plotting the residual distribution
fig = plt.figure(figsize=(12, 8))
ax0 = fig.add_subplot(111)
```

```python
# Plot the distribution of residuals using seaborn's distplot and fit it to a normal distribution
sns.distplot(resid, fit=stats.norm, ax=ax0)

# Get the fitted parameters (mu, sigma) from the normal distribution fit
(mu, sigma) = stats.norm.fit(resid)

# Add legend with fitted parameters to the plot
plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)], loc='best')
plt.ylabel('Frequency')
plt.title('Residual Distribution')

# ACF and PACF plots for residuals
fig = plt.figure(figsize=(12, 8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(sarimax_mod.resid, ax=ax1)  # Autocorrelation (ACF) plot
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(sarimax_mod.resid, ax=ax2)  # Partial autocorrelation (PACF) plot
```

```
NormaltestResult(statistic=133.7420896904873, pvalue=9.083939322120471e-30)
```

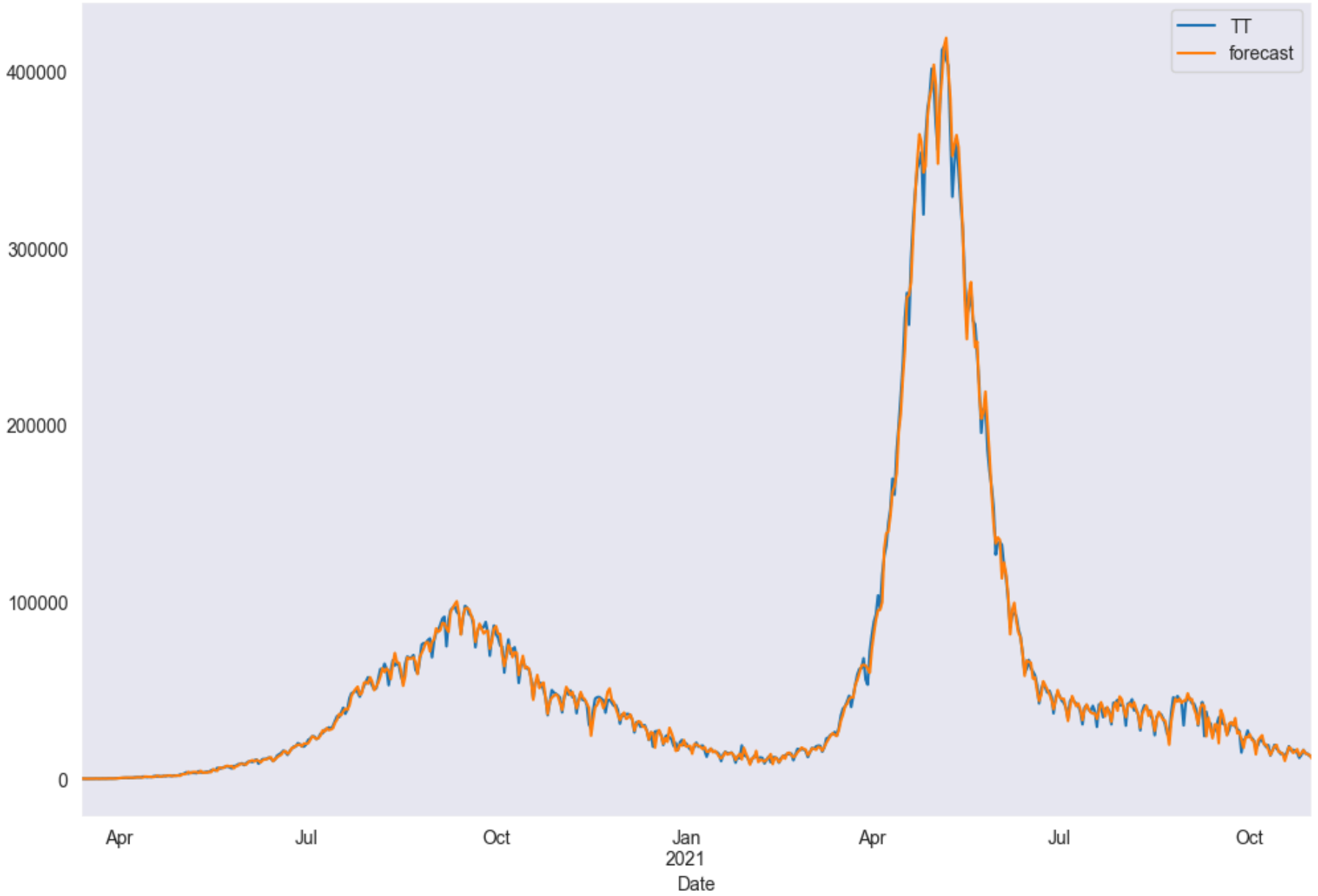

Residual Distribution
Normal dist. (μ = 3.47 and σ = 4330.43)
Frequency
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
−20000
−10000
0
10000
20000



1. Import the necessary libraries (`scipy.stats` for statistical functions and `seaborn` for plotting).
2. Get the residuals (`resid`) from the SARIMAX model (`sarimax_mod`), which represent the model errors.

3. Perform the normality test (`normaltest`) on the residuals using the Jarque-Bera test, which checks for normality in the residuals.
4. Print the results of the Jarque-Bera test to assess the normality of the residuals.
5. Create a figure object (`fig`) for plotting the residual distribution with a specified size.
6. Add a subplot (`ax0`) to the figure for the residual distribution plot.
7. Plot the distribution of residuals using `sns.distplot` from seaborn, fitting it to a normal distribution using `stats.norm` .
8. Get the fitted parameters (`mu` and `sigma`) from the normal distribution fit to display in the legend.
9. Add a legend to the plot indicating the fitted parameters and set axis labels and title.
10. Create another figure object (`fig`) for plotting the autocorrelation and partial autocorrelation of residuals.
11. Add subplots (`ax1` and `ax2`) to the figure for the autocorrelation and partial autocorrelation plots, respectively.
12. Plot the autocorrelation (ACF) of residuals on the first subplot (`ax1`) and the partial autocorrelation (PACF) on the second subplot (`ax2`) using `sm.graphics.tsa.plot_acf` and `sm.graphics.tsa.plot_pacf` , respectively.

In [30]:
```
import datetime

# Get today's date
today = datetime.date.today() - datetime.timedelta(days=1)

# Define the start and end date indices for forecasting
start_index = '14-Mar-20'  # Start date for the forecast
end_index = today.strftime("%Y-%m-%d")  # End date for the forecast (today's date)

# Add forecasted values to the DataFrame and plot the results
train_df['forecast'] = sarimax_mod.predict(start=start_index, end=end_index, dynamic=False)

# Plot the actual 'TT' values along with the forecasted values
train_df[start_index:][['TT', 'forecast']].plot(figsize=(12, 8))

# Show the plot
plt.show()
```



1. Import the necessary libraries, including `datetime` for working with dates.
2. Get today's date (`today`) using `datetime.date.today()` and subtract one day (`datetime.timedelta(days=1)`).

3. Define the start date (`start_index`) for forecasting, which is '14-Mar-20'.
4. Define the end date (`end_index`) for forecasting as today's date formatted as '%Y-%m-%d'.
5. Use the SARIMAX model (`sarimax_mod`) to forecast values from `start_index` to `end_index` using the `predict` method.
    - `start=start_index` : Start date for forecasting.
    - `end=end_index` : End date for forecasting.
    - `dynamic=False` : Use the model's fitted values for forecasting (no dynamic forecasting).
6. Add the forecasted values to the DataFrame `train_df` under the 'forecast' column.
7. Plot the actual 'TT' values and the forecasted values ('forecast') for the time period from `start_index` to the end of the dataset.
8. Set the figure size to (12, 8) for a larger plot.
9. Display the plot using `plt.show()` to visualize the actual and forecasted values.

In [32]:
```
future_predict = sarimax_mod.predict(start=datetime.date.today(), end=datetime.date.today() + datetime.timedelta(days=7), dyna
future_predict
```

```
C:\Users\mailr\AppData\Local\Programs\Python\Python311\Lib\site-packages\statsmodels\tsa\statespace\kalman_filter.py:2473: Va
lueWarning: Dynamic prediction specified to begin during out-of-sample forecasting period, and so has no effect.
  warn('Dynamic prediction specified to begin during'
```

Out[32]:
```
2024-03-31    10235.207098
2024-04-01    10235.209855
2024-04-02    10235.209777
2024-04-03    10235.207255
2024-04-04    10235.207223
2024-04-05    10235.208081
2024-04-06    10235.206978
2024-04-07    10235.207275
Freq: D, Name: predicted_mean, dtype: float64
```

1. `sarimax_mod.predict(...)` : Calls the `predict` method of the SARIMAX model (`sarimax_mod`), which generates forecasts based on the trained model.

2. `start=datetime.date.today()` : Specifies the start date for forecasting. In this case, it's set to the current date using `datetime.date.today()` .

3. `end=datetime.date.today() + datetime.timedelta(days=7)` : Specifies the end date for forecasting. Here, it's set to the current date plus 7 days (`timedelta(days=7)`), which forecasts for the next 7 days from the current date.

4. `dynamic=True`: Indicates whether to use dynamic forecasting. When `dynamic=True`, the forecasted values are used as input for subsequent forecasted values. This parameter can help capture evolving patterns in the time series.

5. `future_predict`: Stores the forecasted values generated by the SARIMAX model for the specified date range.

Explanation:

- The `predict` method is used to forecast future values based on the SARIMAX model (`sarimax_mod`) that has been trained on historical data.
- By setting `start` to the current date and `end` to the current date plus 7 days, we are forecasting for the next 7 days starting from today.
- Setting `dynamic=True` means that the model will use its own forecasted values to predict subsequent values, which can be useful for capturing evolving patterns in the time series.
- The `future_predict` variable will contain the forecasted values for the specified date range. These values can then be used for analysis, plotting, or any further modeling tasks as needed.