CASE STUDY ON Real Estate Sales of the state of Conneticut in USA between 2001-2022

By -
Hussain Ujjainwala (62)
Shannan Rodrigues (10)
Yash Chheda (11)
Moaviya Sayyed(48)

INDEX

Introduction: This report details the Time Series Analysis project, covering dataset handling, stationarity tests, and various time series forecasting techniques, including Exponential Smoothing and ARIMA. The project is primarily focused on forecasting using historical data, exploring different smoothing methods and statistical tests to ensure accurate predictions.

1. Dataset Overview:

The dataset consists of real estate sales records from Connecticut spanning from 2001 to 2022. It includes various attributes such as:

- Town property address
- Date of sale
- Property type (residential, apartment, commercial, industrial, or vacant land)
- Sales price
- Property assessment

This historical data is used for time series forecasting, which is crucial for predicting future sales trends and real estate market analysis.

2. Data Cleaning and Processing:

The dataset undergoes a cleaning process to handle any missing or inconsistent data. Data cleaning ensures the dataset is ready for time series analysis. Steps in this phase include:

- Ensuring consistency in the data format (especially for dates and categorical variables).

This ensures that the data remains stable and accurate across different time periods,sources and variables,allowing for accurate and meaningful insights to be derived.Data consistency in analytics is vital to ensure the integrity and reliability of analytical results.

```
# Strip white spaces from the 'Date Recorded' column
data['Date Recorded'] = data['Date Recorded'].str.strip()

# Now, convert to datetime and handle errors
data['Date Recorded'] = pd.to_datetime(data['Date Recorded'], format='%m/%d/%Y', errors='coerce')
```

Here we are Converting Date Field in the correct Month Day Year Format

This line converts the values in the 'Date Recorded' column to datetime objects. The errors=”coerce” argument ensures that any invalid date values (e.g., wrong formats or missing data) are replaced with Nat(not a Timestamp).

- Converting non-numeric fields to numerical ones for analysis.

```
# Convert Sale Amount to numeric
data_cleaned.loc[:, 'Sale Amount'] = pd.to_numeric(data_cleaned['Sale Amount'], errors='coerce')
```

Why Convert to Numeric?

- **Data Consistency**: Converting the 'Sale Amount' column to a numeric type ensures that all values are consistently handled as numbers, which is crucial for performing mathematical operations, aggregations, and statistical analysis.

- Removing or imputing missing values.

```
print(data.isnull().sum())
```

```
Serial Number             0
List Year                 0
Date Recorded             2
Town                      0
Address                  51
Assessed Value            0
Sale Amount               0
Sales Ratio               0
Property Type        382446
Residential Type     398389
Non Use Code         784172
Assessor Remarks     926395
OPM remarks         1084592
Location             799514
dtype: int64
```

Here the variable of our interest are Date Recorded and Sales amount out of which Date Recorded has 2 null values so we will drop them

```
# Clean the data: remove rows with missing Sale Amount or Date Recorded
data_cleaned = data.dropna(subset=['Date Recorded'])
```

3.Visualising The Data

After preprocessing the dataset, it's essential to visualize the data to better understand its underlying patterns and trends over time. Visualizing time series data helps in identifying key features such as:

- **Trends**: Upward or downward movements in the data over time.
- **Seasonality**: Repeating cycles or patterns that occur at regular intervals.
- **Outliers**: Abnormally high or low values compared to the overall pattern.
- **Stationarity**: Whether the statistical properties of the series, such as mean and variance, remain constant over time.

The visualization step also guides the selection of appropriate models for forecasting and helps in making the data more interpretable for further statistical analysis.

```python
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter

# Define a custom formatter to display y-axis in regular numbers with commas
def millions_formatter(x, pos):
    return f'{int(x):,}'

# Plotting the Monthly Sale Amount
plt.figure(figsize=(10,6))
plt.plot(monthly_data, label='Sale Amount')

# Apply the custom formatter to y-axis
ax = plt.gca()
ax.yaxis.set_major_formatter(FuncFormatter(millions_formatter))

plt.title('Monthly Sale Amount')
plt.legend()
plt.show()
```

1. Imports:

`matplotlib.pyplot` and `FuncFormatter` are imported for creating and customizing plots.

2. Custom Formatter:

The `millions_formatter` function formats y-axis values with commas.

3. Plot Creation:

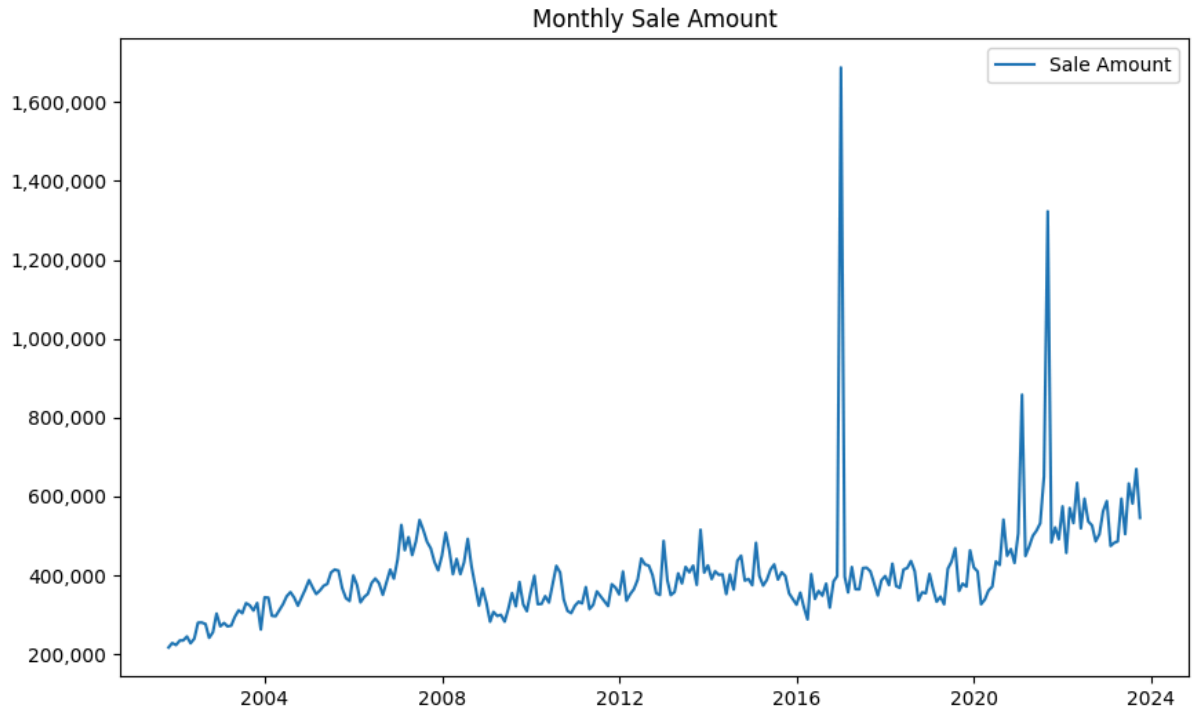
A plot is generated with `monthly_data`, labeled as "Sale Amount", in a 10x6 figure.

4. Axis Customization:

`ax.yaxis.set_major_formatter()` applies the custom comma formatting to the y-axis.

5. Final Touches:

The plot gets a title, legend, and is displayed with `plt.show()`.



Graph Interpretation:

In the graph, we are visualizing the **monthly sale amount** from the given dataset, which represents real estate sales between 2001 and 2022.

1. **Overall Trend**: The graph shows significant fluctuations in sales amounts over the period, with the data generally following an upward trend until around 2015, after which there are sharp spikes.
2. **Spikes**: There are two notable spikes in the data:
    - The first major spike occurs around 2015–2016, where the sale amount drastically increases to more than 1.6 million.
    - The second spike is visible around 2020, which may indicate a one-time event (such as a large sale or market anomaly).
3. These spikes are likely outliers and could be the result of specific events in the real estate market, such as unusually high property transactions.
4. **Post-Spike Trends**: After the spikes, the sales amount decreases but follows a more consistent upward trajectory with fluctuations. There are still periodic small increases, suggesting some degree of seasonality or cyclical behaviour.

4.Exponential Smoothing

Exponential Smoothing is the weighted average of the past data, with the recent data points given more weight than earlier data points.Exponential smoothing is a forecasting technique used to predict future values in time series data by averaging past observations with exponentially decreasing weights. The main idea is to give more importance to recent data while still considering older observations, making it particularly useful for short-term forecasting.

Single Exponential Smoothing

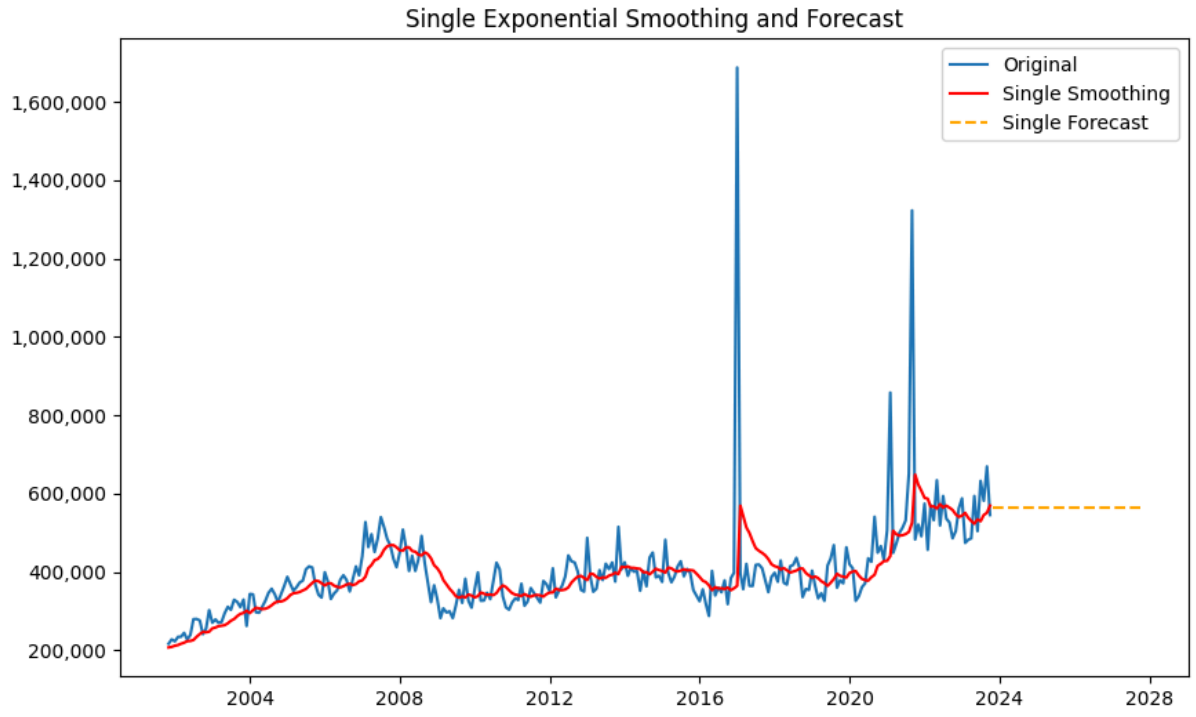
The simplest of the exponentially smoothing methods is naturally called simple exponential smoothing .This method is suitable for forecasting data with no clear trend or seasonal pattern. Single Exponential Smoothing (SES) is a forecasting method used to predict future values in a time series by smoothing past observations with an exponentially decreasing weight.

Purpose: SES smooths time series data for forecasting, giving more weight to recent values. Best for data without trends or seasonality.

Formula: $S_t = \alpha Y_t + (1 - \alpha) S_{t-1}$, where α is the smoothing factor, controlling how much weight is given to recent data.

Key Parameter: α (smoothing factor) between 0 and 1, with higher values giving more weight to recent observations.
Use Cases: Ideal for short-term forecasts in stable time series without trends or seasonality.
Limitations: Not suitable for data with trends or seasonal patterns.

```python
import matplotlib.pyplot as plt
from statsmodels.tsa.api import ExponentialSmoothing
from matplotlib.ticker import FuncFormatter
import pandas as pd

# Number of steps to forecast
forecast_steps = 48

# Single Exponential Smoothing
single_smooth = ExponentialSmoothing(monthly_data.dropna(), trend=None).fit()
single_forecast = single_smooth.forecast(forecast_steps)
# Create a future date range that matches the forecast period
last_date = monthly_data.index[-1]
forecast_index = pd.date_range(start=last_date, periods=forecast_steps + 1, freq='M')[1:]

# Plot Single Exponential Smoothing with forecast
plt.figure(figsize=(10,6))
plt.plot(monthly_data, label='Original')
plt.plot(single_smooth.fittedvalues, label='Single Smoothing', color='red')
plt.plot(forecast_index, single_forecast, label='Single Forecast', color='orange', linestyle='dashed')
ax = plt.gca()
ax.yaxis.set_major_formatter(FuncFormatter(millions_formatter))
plt.title('Single Exponential Smoothing and Forecast')
plt.legend()
plt.show()
```



Brief Summary:

- Original Data (Blue Line): Shows the actual sales data with noticeable spikes around 2015-2016 and 2020.
- SES Smoothing (Red Line): Smooths the data, revealing a more stable trend by reducing noise but doesn't capture large spikes.

- SES Forecast (Dashed Yellow Line): Predicts a flat future trend, indicating SES struggles to handle data with trends or seasonality.

Insights:

- SES does a good job of smoothing out the noise in the time series and providing a clearer view of the long-term trend.
- However, it is not well-suited for forecasting in this case because it cannot capture the complex dynamics, such as the sharp spikes and trends in the data.
- The flat forecast indicates that SES is assuming the future sales will stabilize around the smoothed values, which may not reflect real market conditions.

Double Exponential Smoothing

Double Exponential Smoothing extends Single Exponential Smoothing by incorporating trend information, making it suitable for data with linear trends. It uses two smoothing equations to account for both level and trend components in the time series.Double Exponential Smoothing model is suitable to model the time series with trend but without seasonality.

Purpose: DES is used for forecasting time series with trends by applying smoothing twice—once for the level and once for the trend.

Formula:

- Level: $L_t = \alpha Y_t + (1 - \alpha)(L_{t-1} + T_{t-1})$
- Trend: $T_t = \beta(L_t - L_{t-1}) + (1 - \beta)T_{t-1}$
- Forecast: $F_{t+m} = L_t + mT_t$ α controls the level smoothing, and β controls trend smoothing.

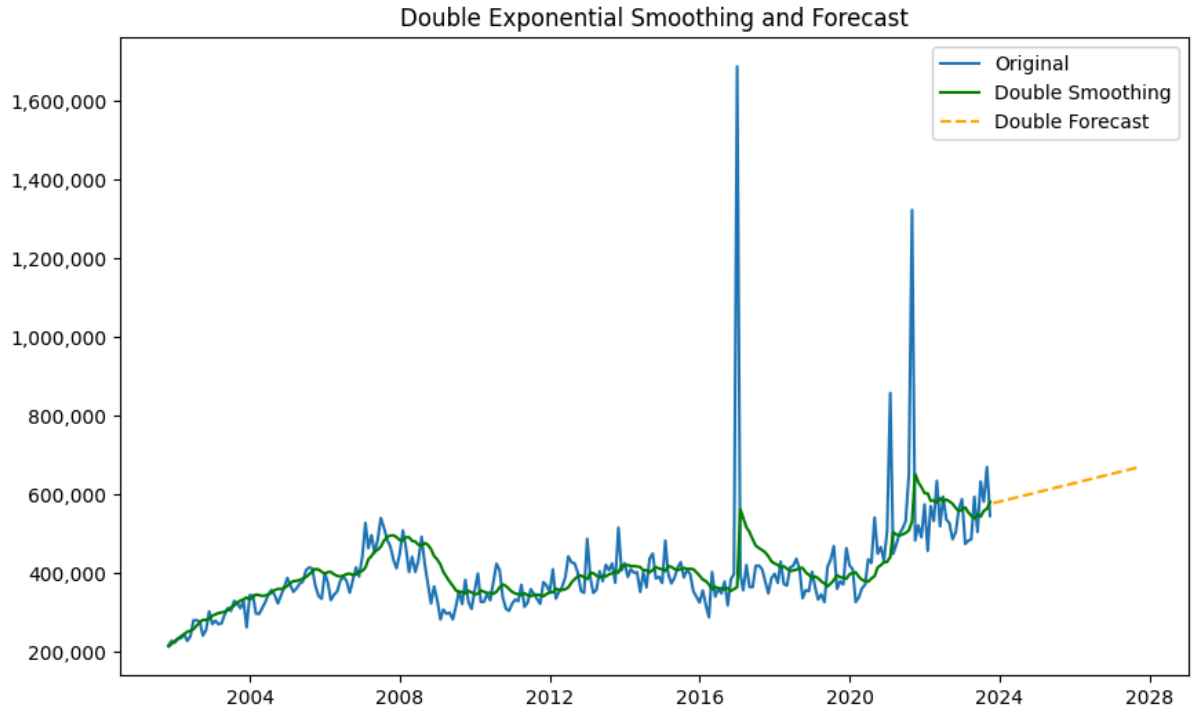
Key Parameters:

  - α for level smoothing.
  - β for trend smoothing.

Use Cases: Suitable for data with an upward or downward trend but without seasonality.

Limitations: Cannot handle seasonal patterns; for seasonality, Triple Exponential Smoothing (Holt-Winters) is used.

```python
# Double Exponential Smoothing
double_smooth = ExponentialSmoothing(monthly_data.dropna(), trend="add").fit()
double_forecast = double_smooth.forecast(forecast_steps)

# Plot Double Exponential Smoothing with forecast
plt.figure(figsize=(10,6))
plt.plot(monthly_data, label='Original')
plt.plot(double_smooth.fittedvalues, label='Double Smoothing', color='green')
plt.plot(forecast_index, double_forecast, label='Double Forecast', color='orange', linestyle='dashed')
ax = plt.gca()
ax.yaxis.set_major_formatter(FuncFormatter(millions_formatter))
plt.title('Double Exponential Smoothing and Forecast')
plt.legend()
plt.show()
```



Brief Summary:

1. **Original Data (Blue Line)**:
    - This represents the actual monthly sales data. The blue line shows the volatility in sales, including significant spikes in 2015-2016 and 2020.
2. **Double Exponential Smoothing (Green Line)**:
    - Double Exponential Smoothing (DES) is applied to model the trend in the data. This smoothing accounts for both the level and trend in the time series.
    - The green line follows the trend more closely than Single Exponential Smoothing, adapting to changes in direction, but still struggles with large spikes.
3. **Double Forecast (Dashed Yellow Line)**:
    - The dashed line represents the forecasted values using DES. The forecast predicts an upward trend, indicating that the method captures the data's

underlying trend better than SES but may still not handle large, sudden fluctuations well.

Insights:

- **Improved Trend Detection**: Compared to SES, DES captures the data's overall trend more effectively, as seen in the smoother line following the actual sales values.
- **Forecast**: DES forecasts an increasing trend, reflecting its ability to adapt to gradual changes, though it's not equipped to handle sharp spikes like those seen in the data.

Triple/Exponential Smoothing (with seasonal component)

Triple Exponential Smoothing, also known as the Holt-Winters method, is an advanced forecasting technique designed to handle time series data with trends and seasonal patterns. Triple exponential smoothing produces an exponential moving average that takes into account the tendency of data to repeat itself in intervals over time.

Purpose: Triple Exponential Smoothing accounts for level, trend, and seasonality in time series forecasting, making it suitable for data with periodic patterns.

Formula:

- Level: $L_t = \alpha(Y_t/S_{t-s}) + (1-\alpha)(L_{t-1} + T_{t-1})$
- Trend: $T_t = \beta(L_t - L_{t-1}) + (1-\beta)T_{t-1}$
- Seasonal: $S_t = \gamma(Y_t/L_t) + (1-\gamma)S_{t-s}$
- α, β, and γ are smoothing factors for level, trend, and seasonality respectively.

Key Parameters:

- α for level smoothing.
- β for trend smoothing.
- γ for seasonal smoothing.
- `seasonal_periods` defines the length of a seasonal cycle.

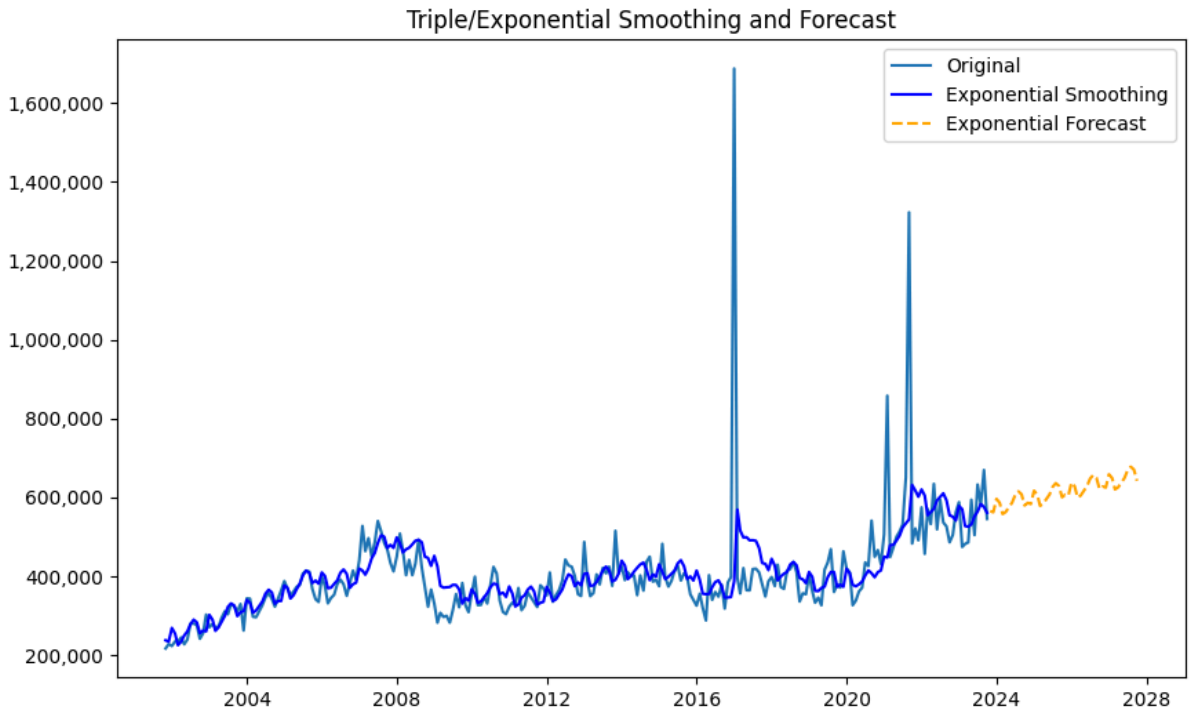
Key Parameters:

- α for level smoothing.
- β for trend smoothing.
- γ for seasonal smoothing.
- seasonal_periods defines the length of a seasonal cycle.

Use Cases: Ideal for data with seasonal patterns, like monthly sales or annual temperature cycles.

Limitations: Requires sufficient historical data to capture seasonality, and inappropriate for non-seasonal data.

```
# Triple/Exponential Smoothing (with seasonal component)
exp_smooth = ExponentialSmoothing(monthly_data.dropna(), trend='add', seasonal='add', seasonal_periods=12).fit()
exp_forecast = exp_smooth.forecast(forecast_steps)

# Plot Triple/Exponential Smoothing with forecast
plt.figure(figsize=(10,6))
plt.plot(monthly_data, label='Original')
plt.plot(exp_smooth.fittedvalues, label='Exponential Smoothing', color='blue')
plt.plot(forecast_index, exp_forecast, label='Exponential Forecast', color='orange', linestyle='dashed')
ax = plt.gca()
ax.yaxis.set_major_formatter(FuncFormatter(millions_formatter))
plt.title('Triple/Exponential Smoothing and Forecast')
plt.legend()
plt.show()
```



Brief Summary:

1. Original Data (Blue Line):
    - This represents the actual historical sales data, showing significant fluctuations and spikes around 2015-2016 and 2020.
2. Triple Exponential Smoothing (Dark Blue Line):
    - Triple Exponential Smoothing (Holt-Winters method) is applied to account for level, trend, and seasonality in the data. The dark blue line smooths the data more effectively by factoring in these components, providing a clearer representation of the underlying trend despite the volatility and spikes.
3. Triple Exponential Forecast (Dashed Yellow Line):

    - The forecasted values (dashed yellow line) predict an upward trend, reflecting the seasonal patterns detected in the data. Unlike Single and Double Exponential Smoothing, this method captures the seasonality, allowing for a more realistic forecast that includes periodic fluctuations.

Insights:

- Improved Modeling: Triple Exponential Smoothing handles both trend and seasonality better than previous methods. The dark blue line closely follows the original data, smoothing out noise while capturing the overall pattern.
- Forecast: The forecast shows an upward trend with periodic fluctuations, indicating that the model has successfully captured both the trend and seasonality, making it more suitable for forecasting in complex time series data like real estate sales.

5. Tests for Stationarity:

What is Stationarity?
A stationary time series is one where the statistical properties (mean, variance, and autocorrelation) remain constant over time. This consistency simplifies the process of modeling and forecasting and is a critical assumption for many time series analysis techniques. Testing and transforming data to achieve stationarity are essential steps in effective time series modeling.

Properties of a Stationary Time Series:
Constant Mean: The average value of the series remains the same over time.
Constant Variance: The variability or dispersion of the series remains consistent over time.
Constant Autocovariance: The relationship between observations at different time lags is constant.

How to Check Stationarity?
Checking for stationarity in a time series is a crucial step in time series analysis, as many statistical methods and models assume that the data is stationary. To check for stationarity in a time series, start with visual inspection and rolling statistics, then use statistical tests like the ADF, PP, and KPSS tests for more formal analysis. If the series is non-stationary, apply transformations such as differencing or logarithmic transformations to stabilize the series and make it suitable for further analysis.

1) **ADF Test**

The Augmented Dickey-Fuller (ADF) test is a statistical test used to determine if a time series data set is stationary or if it contains a unit root, which implies non-stationarity.The Augmented Dickey-Fuller (ADF) test is a widely used method for assessing stationarity in time series data. By testing the null hypothesis of a unit root, it helps determine whether the series is stationary or requires further transformation. The test involves estimating a model with lagged differences and comparing the test statistic to critical values or using p-values to make inferences about the stationarity of the series.

Purpose: Tests for stationarity in a time series by checking for the presence of a unit root.

Null Hypothesis: The time series does not have a unit root (i.e., it is non-stationary).

Alternative Hypothesis: The time series has a unit root (i.e., it is stationary).

Key Metric: If the p-value is less than a significance level (e.g., 0.05), the null hypothesis is rejected, indicating the series is stationary.

Formula: Involves regressing the series on lagged values, differences, and a deterministic trend to identify the presence of a unit root.

Use Case: Used to determine whether differencing is needed for time series modeling (e.g., in ARIMA).

```
from statsmodels.tsa.stattools import adfuller, kpss
from arch.unitroot import PhillipsPerron

# ADF Test
adf_result = adfuller(monthly_data.dropna())
print(f"ADF Statistic: {adf_result[0]}")
print(f"p-value For ADF: {adf_result[1]}")
```

```
ADF Statistic: -2.6364241768173877
p-value For ADF: 0.08570034707326246
```

Conclusion: Here the p-value is greater than 0.05 therefore we don't reject H0 and conclude that the Data is non-stationary.

2) Philips-Perron Test

The Phillips-Perron (PP) test is a statistical test used to assess whether a time series has a unit root, indicating non-stationarity. It is similar to the Augmented Dickey-Fuller (ADF) test but adjusts for serial correlation and heteroscedasticity in the residuals.The Phillips-Perron (PP) test is a method for detecting unit roots in time series data, indicating non-stationarity. It adjusts for serial correlation and heteroscedasticity in the residuals, providing a robust alternative to the Augmented Dickey-Fuller (ADF) test. By testing the null hypothesis of a unit root, it helps determine whether a time series is stationary or requires transformation. The PP test's results are interpreted based on p-values and critical values, with attention to the test's robustness and potential limitations.

Purpose: Another test for stationarity, like ADF, but it uses non-parametric adjustments to account for autocorrelation and heteroscedasticity in errors.

Null Hypothesis: The time series has a unit root (non-stationary).

Alternative Hypothesis: The time series has a unit root (i.e., it is stationary).

Key Metric: Similar to ADF, where a low p-value indicates rejection of the null hypothesis and stationarity in the series.

Improvement: PP test improves on ADF by adjusting for serial correlation without adding lagged difference terms.

Use Case: Useful when ADF might be unreliable due to autocorrelation in the residuals.

```
# PP Test
pp_test = PhillipsPerron(monthly_data.dropna())
print(f"PP Test Statistic: {pp_test.stat}")
print(f"p-value For PP-Test: {pp_test.pvalue}")
```

```
PP Test Statistic: -13.7501342151867
p-value For PP-Test: 1.0543308318689678e-25
```

Conclusion: Here the p-value is greater than 0.05 therefore we don't reject H0 and conclude that the Data is non-stationary.

3) KPSS Test

The Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test is a statistical test used to assess the stationarity of a time series. It tests for stationarity around a deterministic trend and is often used in conjunction with other tests like the Augmented Dickey-Fuller (ADF) test to provide a comprehensive assessment of stationarity.
The Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test is used to assess whether a time series is stationary around a deterministic trend (or a constant level). It complements other unit root tests like the Augmented Dickey-Fuller (ADF) test by focusing on the null hypothesis of stationarity. The KPSS test is interpreted through its test statistic and p-values, with critical values used to determine whether the series is stationary or non-stationary.

Purpose: Tests for stationarity by testing the null hypothesis that a time series is trend-stationary.

Null Hypothesis: The time series is stationary (stationary around a deterministic trend).

Alternative Hypothesis: The time series does not have a unit root (i.e., it is non-stationary).

Key Metric: A high p-value indicates the series is stationary, whereas a low p-value suggests non-stationarity.

Formula: Decomposes the series into deterministic trend, random walk, and stationary error components.

Use Case: Complements ADF and PP tests by testing the opposite hypothesis, helping verify results.

```
# KPSS Test
kpss_stat, p_value, lags, critical_values = kpss(monthly_data.dropna(), regression='c') # c for level stationarity
print(f"KPSS Statistic: {kpss_stat}")
print(f"p-value for KPSS: {p_value}")
```

```
KPSS Statistic: 1.4351531034767144
p-value for KPSS: 0.01
```

Conclusion: Here the p-value is lesser than 0.05 therefore we reject H0 and conclude that the Data is non-stationary

6) Stationarity Transformation:

Why to make data Stationary?
Stationarity is really the foundational concept in time series analysis, so its importance cannot be overstated. The following points are considered while making the data stationary:

- **Assumptions of many time series models:** Stationarity is a fundamental assumption for numerous time series models such as ARIMA (Auto Regression Integrated Moving Average) and SARIMA (Seasonal ARIMA). These models rely on the constancy of statistical properties like mean and variance over time. Non-stationary data can lead to unreliable model outputs and inaccurate predictions, just because the models aren't expecting it.
- **Easier modeling and forecasting:** Stationarity simplifies the complexities within time series data, making it easier to model and forecast than non-stationary time series. When the statistical properties of a time series remain constant over time, it's much easier to use historical data to develop accurate models of the time series and forecast future values of the series.
- **Interpretability of trends and patterns:** Trends and patterns in stationary time series are easier to interpret because relationships between data points remain constant, which means that we can be confident that any trends or patterns we observe are not simply due to random fluctuations in the data.
- **Enhanced diagnostic checks:** Stationarity enables more robust diagnostic checks in time series analysis. By confirming stationarity, analysts can identify any potential issues in the data that might violate this essential assumption.
- **Improved model performance:** A stationary time series typically results in improved model performance. The constancy of key statistical properties ensures that models can better capture the underlying dynamics, leading to more accurate predictions.

Differencing:

Differencing is a method of transforming a time series to remove trends or seasonal patterns by subtracting the previous observation from the current observation.

Why it's used: It is mainly used to stabilize the mean of a time series by removing trends. By applying differencing, you eliminate the time-dependent structure (trend) from the data.

The primary goal of differencing is to remove trends from a time series, making the data stationary, which is a key requirement for many time series forecasting models such as ARIMA. Here's why differencing is crucial:

- **Stabilizes the mean**: If your time series shows a clear upward or downward trend, the mean will change over time, which makes it hard to model. Differencing helps by removing this trend, ensuring the mean remains constant over time.
- **Prepares data for ARIMA and other models**: Many statistical models, including ARIMA, assume that the time series is stationary. By removing the trend or seasonality, differencing allows these models to better capture the relationship between data points.
- **Removes autocorrelation**: Differencing reduces the dependence between time steps (also known as autocorrelation), especially when there is a time-based pattern in the data.

- **Mathematical Formula:**

$$Y_t' = Y_t - Y_{t-1}$$

where:

- Y_t' is the differenced value at time t.
- Y_t is the value at time t.
- Y_{t-1} is the value at time $t - 1$.

Second-order differencing:

What it does: If the first-order differencing does not completely remove the trend (or if the trend is non-linear), you may need to apply second-order differencing. This method involves taking the first-order differences of the first-order differences (i.e., the difference of differences).

- **Mathematical Formula:**

$$Y_t'' = Y_t' - Y_{t-1}' = (Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2})$$

where:

- Y_t'' is the second-order differenced value at time t.
- Y_t' is the first-order differenced value at time t.
- Y_{t-1} is the first-order differenced value at time $t-1$.

```
# Step 3: First-order differencing
differenced_data = np.diff(transformed_data, n=1)  # First-order differencing

# Step 4: Create a new time index for differenced data (it will be 1 less than the original index)
differenced_index = cleaned_data.index[1:]
```

Box-cox:
The Box-Cox transformation is a family of power transformations that stabilize variance and make the data more normally distributed. It is a more flexible transformation because it can handle a range of data characteristics, including data with increasing variance over time.
The Box-Cox transformation generalizes several specific transformations, including logarithmic and square root transformations:
Logarithmic transformation (when $\lambda = 0$) is a special case of Box-Cox and is often used when the data has exponential growth or non-constant variance.
Square root transformation (when $\lambda = 0.5$) is another special case that can be applied when variance increases with the level of the time series.
The goal of the Box-Cox transformation is to make the data more stationary and normally distributed.

Why it's used:

Stabilizing variance: Time series data often exhibits heteroscedasticity, where the variance changes over time. This can make modeling difficult, as many statistical models assume constant variance. data.

Normalizing the data: Many time series models, such as ARIMA, assume that the data is normally distributed. The Box-Cox transformation helps by making the distribution of the data more normal (i.e., bell-shaped), which improves model accuracy.

Handling positive values: The Box-Cox transformation is only applicable to positive data. If the data has negative values, a constant must be added to shift the data before applying the transformation.

When to use: When the series shows both non-constant variance and non-normality.

How to do it:

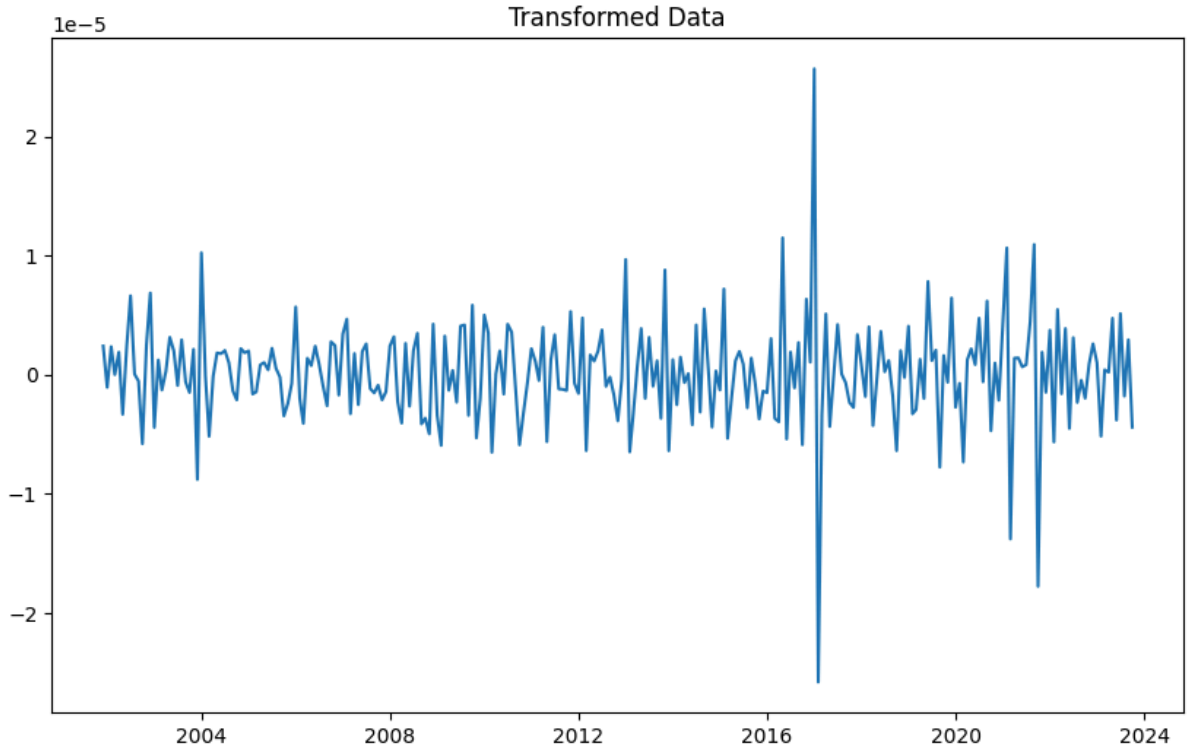
- Box-Cox transformation:

$$Y_t' = \frac{Y_t^\lambda - 1}{\lambda}$$

```
# Step 2: Apply Box-Cox transformation
transformed_data, lambda_value = boxcox(cleaned_data)
```

Visualising the Data Again after applying Box-Cox transformation and Differencing

```
# Step 5: Plot the differenced data using the correct index (differenced_index)
plt.figure(figsize=(10,6))
plt.plot(differenced_index, differenced_data)
plt.title('Transformed Data')
plt.show()
```



Now that we have applied the transformation we have to check if the data is stationary now

```
ADF Statistic: -6.520525033792338
p-value For ADF: 1.0449593909068573e-08
PP Test Statistic: -36.0768046809192
p-value For PP-Test: 0.0
KPSS Statistic: 0.18761249882182346
p-value for KPSS: 0.1
```

After running all the 3 tests again we got the following output
Here for ADF test the p-value is less than 0.05, therefore we reject H0 and conclude that the Data is Stationary
Similarly for PP test the p-value is less than 0.05, therefore we reject H0 and conclude that the Data is Stationary
For KPSS test the p-value is greater than 0.05, therefore we do not reject H0 and conclude that the Data is Stationary

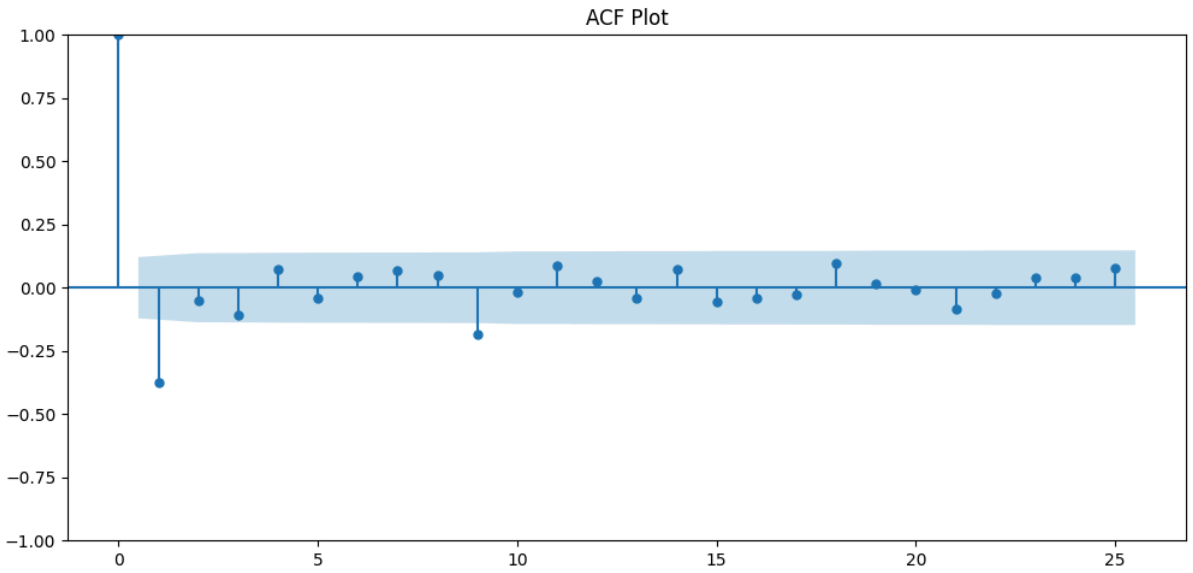
7) Autocorrelation and Partial Autocorrelation Functions

ACF:

The Autocorrelation Function (ACF) is a key tool in time series analysis that measures the correlation between a time series and lagged versions of itself. It helps identify patterns, trends, and relationships within the time series data.The Autocorrelation Function (ACF) measures the correlation between a time series and its lagged versions, providing insights into the internal structure of the data. It helps identify patterns, trends, and the appropriate modeling approach. By examining the ACF plot, analysts can make informed decisions about model selection and evaluate the goodness-of-fit for time series models.

- **Purpose**: Measures the correlation between a time series and its lagged values to identify patterns and dependencies in data over time.
- **Key Metric**: Autocorrelation coefficients for different lag values; high values indicate strong correlation at specific lags.
- **Use Case**: Useful in identifying the order of the **MA** (Moving Average) term in ARIMA models and detecting seasonality.

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# ACF and PACF plots
fig, ax = plt.subplots(2, 1, figsize=(12, 12))

# ACF plot
plot_acf(differenced_data, ax=ax[0])
ax[0].set_title('ACF Plot')
```



Interpretation:

1. **Lag 0 (Correlation = 1)**:
    - The autocorrelation at lag 0 is always 1 because each data point is perfectly correlated with itself.
2. **First Few Lags**:
    - The autocorrelation quickly drops after lag 0, showing a slight negative correlation at lag 1. This suggests that the series is not strongly correlated with its immediate past.
3. **Lags Beyond Lag 1**:
    - Most of the points fall within the shaded confidence interval, which means that the autocorrelation at these lags is statistically insignificant.
    - The values of autocorrelations beyond the first few lags hover around zero, indicating that the series does not have a strong autocorrelation structure over time.
4. **Confidence Interval (Shaded Area)**:
    - The shaded area represents the confidence intervals for the correlations at each lag. Any autocorrelation points outside this shaded region are statistically significant, while those within are not.

Insights:

- **Lack of Strong Autocorrelation**: Since most of the values after lag 1 fall within the confidence interval, the time series does not have significant autocorrelation for most lags.
- **Stationarity**: The quickly decaying autocorrelation suggests that the series may be stationary or close to stationary.

PACF:

PACF (Partial Autocorrelation Function) is a statistical tool used in time series analysis to measure the correlation between a time series and its lagged values, while controlling for the correlations at shorter lags. It helps determine the direct relationship between an observation and its lagged values, excluding the influence of intermediate observations.

Why Use PACF?

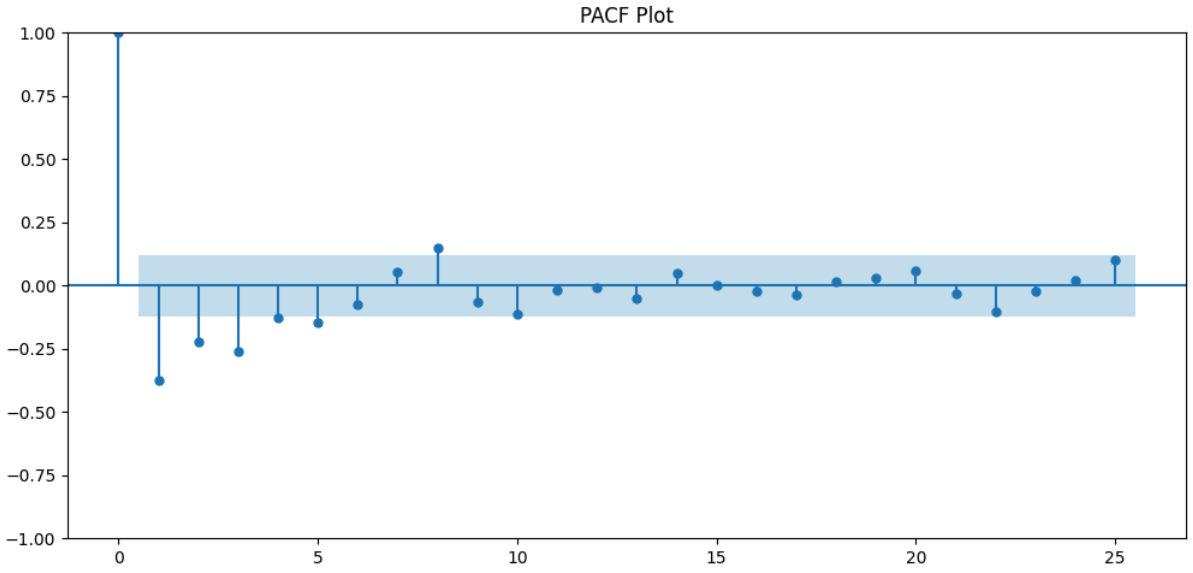
- In an autocorrelation plot (ACF), each lag may be influenced by previous lags, making it hard to determine the direct influence of a specific lag.
- PACF removes this indirect influence, showing only the correlation that a lag has with the current observation while controlling for the other lags.

Identifying AR Terms Using PACF:

- The PACF plot is commonly used to identify the order of the **AR (AutoRegressive)** component in ARIMA models.

- A **sharp cut-off** after a certain lag in the PACF plot suggests that an AR model up to that lag order is appropriate. For example, if the PACF plot shows significant correlation only for the first lag, an AR(1) model might be suitable.

```
# PACF plot
plot_pacf(differenced_data, ax=ax[1])
ax[1].set_title('PACF Plot')

plt.show()
```



Interpretation:

1. **Lag 0 (Correlation = 1)**:
    - As expected, the correlation at lag 0 is 1 since the time series is perfectly correlated with itself at this lag.
2. **First Few Lags**:
    - The first lag has a strong negative correlation that falls outside the confidence interval, indicating a statistically significant partial autocorrelation.
    - For lags 2 to 5, there is a noticeable decrease in autocorrelation, and they remain outside the confidence interval, showing significance up to lag 5.
3. **Lags Beyond 5**:
    - Beyond lag 5, the partial autocorrelation values are mostly within the shaded confidence interval, indicating that these lags are not statistically significant.
4. **Confidence Interval (Shaded Area)**:
    - The shaded region represents the confidence intervals for the PACF values. Any point outside this region represents a significant correlation, while those inside the region are statistically insignificant.

Insights:

- **Significant PACF at Lag 1**: The large negative spike at lag 1 suggests a strong inverse relationship between the current value and the immediately preceding value. This can indicate a strong autoregressive (AR) component.
- **Decaying Autocorrelation**: The significance of the first few lags indicates that the AR component of the time series may be of low order.

8) ARIMA (AutoRegressive Integrated Moving Average)

ARIMA (AutoRegressive Integrated Moving Average) is a popular time series forecasting model that combines three components: AutoRegressive (AR), Integrated (I), and Moving Average (MA). It is designed to predict future values based on past values in a time series while accounting for patterns such as trends and noise.
Here's a breakdown of the components:

1) **AutoRegressive (AR) Component:**

- The AR part of the model refers to the use of past values (lags) to predict future values. The AR term, denoted by **p**, represents how many previous observations should be considered in the prediction.
- For example, in an AR(1) model, the current value depends on the immediately preceding value.

2) **Integrated (I) Component:**

- The "Integrated" part addresses the need for the time series to be stationary. Stationarity means that the mean and variance of the series do not change over time.
- If the series is not stationary, the I component applies **differencing** to the data (replacing the value with the difference between the value and its previous value) to remove trends and make the series stationary. The degree of differencing is represented by **d**.
- For example, if a time series has a trend, applying first-order differencing (d = 1) would remove the trend.

3) **Moving Average (MA) Component:**

- The MA part refers to modeling the relationship between an observation and the residual errors from past observations. The MA term, denoted by **q**, determines how many lagged forecast errors are used to predict future values.
- For instance, in an MA(1) model, the current value is influenced by the previous period's forecast error.

Components of ARIMA (Slide)

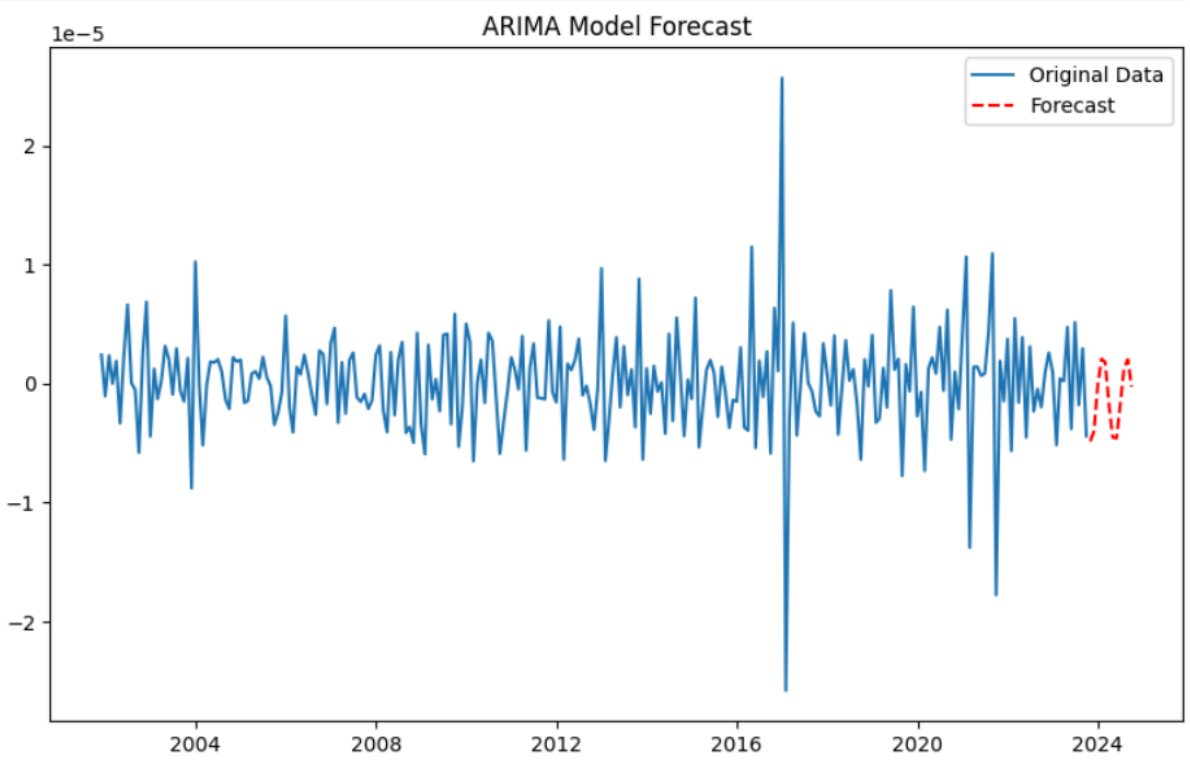
- **p (AutoRegressive)**: Determines the number of lag observations in the model. It refers to the number of terms in the autoregressive component.
- **d (Differencing)**: Determines the degree of differencing needed to make the time series stationary. A stationary series is one whose properties do not depend on the time at which the series is observed (constant mean, variance, etc.). If your series is not stationary, differencing can help achieve stationarity.
- **q (Moving Average)**: Determines the number of lagged forecast errors included in the model

Identifying p, d, q in ARIMA

1. **Determine d (Differencing)**:
    - Perform **differencing** until the series becomes stationary. Typically, use the ADF or KPSS test to check if differencing makes the series stationary.
    - Start with d=(first-order differencing) and check the stationarity using statistical tests or by examining the autocorrelation function (ACF) plot. If necessary, increase d.
2. **Determine p (AutoRegressive)**:
    - Look at the **Partial Autocorrelation Function (PACF) plot**. The value of p is determined by the number of lags after which the PACF cuts off (drops to zero). For example, if the PACF cuts off after 2 lags, p=2.
    - A strong PACF at the first lag suggests an AR(1) model might be appropriate.
3. **Determine q (Moving Average)**:
    - Look at the **Autocorrelation Function (ACF) plot**. The value of qqq is determined by the number of lags after which the ACF cuts off. For example, if the ACF cuts off after 1 lag, q=1.

```python
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA

# Step 1: Convert differenced data into a pandas Series, using the appropriate index
differenced_series = pd.Series(differenced_data, index=differenced_index)

# Step 2: Define the ARIMA model
model = ARIMA(differenced_series, order=(5, 1, 1))  # Adjust p, d, q based on ACF/PACF
arima_model = model.fit()



forecast_steps = 12  # Example: Forecast next 12 periods
forecast = arima_model.forecast(steps=forecast_steps)


forecast_index = pd.date_range(start=differenced_series.index[-1], periods=forecast_steps + 1, freq='M')[1:]


plt.figure(figsize=(10, 6))
plt.plot(differenced_series, label='Original Data')
plt.plot(forecast_index, forecast, label='Forecast', linestyle='--', color='red')
plt.title('ARIMA Model Forecast')
plt.legend()
plt.show()
```

Here we have decided the p,q based on the PACF and ADF graph that we have created earlier and D is 1 because differencing is done once.



Key Components:

1. **Original Data (Blue Line)**:
    - This line represents the original, differenced time series data. The data shows fluctuating patterns around a mean value close to zero, indicating that the differencing step effectively removed any trends from the data.
    - The time series appears to exhibit some noise and periodic fluctuations, but without obvious trends after differencing.
2. **Forecast (Dashed Red Line)**:
    - The dashed red line represents the forecasted values for the next 12 periods based on the ARIMA model.
    - The forecast shows slight fluctuations, following the pattern of the original data but with less noise.
    - Since ARIMA uses past values and residuals for forecasting, the forecast aligns with the observed behavior of the series, but it does not predict large deviations from the recent behavior.

Insights:

- **Stationarity**: The original data appears stationary after differencing, with the values oscillating around a mean of zero. This indicates that the ARIMA model was applied to stationary data, which is a requirement for this type of model.
- **Forecast Accuracy**: The forecasted values show small fluctuations, which indicates that the model anticipates the time series to continue fluctuating around the mean. The lack of large changes in the forecast suggests that ARIMA has captured the core

pattern of the data but expects it to remain stable, without significant upward or downward trends.

- **Model Fit**: The ARIMA model seems to have captured the general behavior of the time series, but the forecast does not show dramatic changes, which might be expected for data that doesn't have strong seasonal or trend components after differencing.