Introduction

In this blog post, we'll tackle the challenge of forecasting sales for a fast-growing detergent company. Accurate sales forecasts are crucial for managing inventory, allocating resources, and making strategic decisions. The task can be complex due to various influencing factors like market trends, seasonality, and promotional activities. We will guide you through a data-driven approach to predict future sales using Python, a versatile programming language popular in data analysis and machine learning.

Getting Started with Sales Forecasting

Step 1: Understanding the Problem

The first step in forecasting sales is to understand the business context and the factors that influence sales figures. For a detergent company, these could include:

- **Seasonal patterns:** Sales may peak at certain times of the year due to seasonal cleaning trends.
- **Promotional campaigns:** Discounts and advertising can temporarily boost sales.

Step 2: Data Collection

Data collection is critical. Ideally, we have access to historical sales data and we have monthly sales data for the past five years.

Importing Necessary Libraries

To begin, we will need to import several Python libraries that are essential for data manipulation and analysis:

In [1]:
```
import pandas as pd
import numpy as np
from statsmodels.tsa.stattools import adfuller, acf, pacf
import matplotlib.pyplot as plt
import warnings
from itertools import product
warnings.filterwarnings('ignore')
```

Loading and Viewing the Data

Next, we load our data from a CSV file and take an initial look to understand its structure:

In [2]:
```
# Load the data from the CSV file
file_path = 'Sales_data_ts.csv'
data = pd.read_csv(file_path)
```

In [3]:
```
data.head()
```

Out[3]:

	date	gross_sales
0	2015-10	34100.0
1	2015-11	69952.4
2	2015-12	164420.0
3	2016-01	86250.0
4	2016-02	101480.0

With the data loaded, the next step is to ensure it's clean and ready for analysis. This involves checking for missing values, outliers, and ensuring that dates are in the correct format for time series analysis. Here's an example of how we might approach this:

In [5]:
```
# Converting the 'Date' column to datetime format
data['date'] = pd.to_datetime(data['date'])

# Checking for missing values
missing_values = data.isnull().sum()
print("Missing values in each column:\n", missing_values)

# Filling missing values, if any, with the previous value (forward-fill)
data.fillna(method='ffill', inplace=True)
```

```
Missing values in each column:
 date           0
gross_sales    0
dtype: int64
```

Stationarity Check: The First Step in Time Series Analysis

Before diving into complex forecasting models, it's imperative to understand the nature of our time series data. The assumption that many time series techniques rely on is stationarity, meaning that the statistical properties of the series (like mean and variance) do not change over time. This property of stationarity can greatly affect the performance of our forecasting models.

To assess whether our sales data is stationary, we can use the Dickey-Fuller test, a common statistical test used to determine the presence of a unit root in the series and, by extension, whether the series is non-stationary.

Here's how we can implement the Dickey-Fuller test in Python using the statsmodels library:

In [6]:
```
import pandas as pd
from statsmodels.tsa.stattools import adfuller

# Convert the 'date' column to datetime format and set it as the index
data['date'] = pd.to_datetime(data['date'])
data.set_index('date', inplace=True)

# Run Dickey-Fuller test
df_test = adfuller(data['gross_sales'], autolag='AIC')

# Extract and display the results of the Dickey-Fuller test
df_results = pd.Series(df_test[0:4], index=['Test Statistic', 'p-value', '#Lags Used', 'Number of Ob
for key, value in df_test[4].items():
    df_results[f'Critical Value ({key})'] = value
```

```
print(df_results)
```

```
Test Statistic                 -0.411986
p-value                         0.908078
#Lags Used                     12.000000
Number of Observations Used    75.000000
Critical Value (1%)            -3.520713
Critical Value (5%)            -2.900925
Critical Value (10%)           -2.587781
dtype: float64
```

The output shows the test statistic and the critical values for different confidence levels. If the test statistic is less than the critical value, we can reject the null hypothesis (which is that the series is non-stationary) and conclude that the series is stationary.

In our case, the test statistic is greater than the critical values, and the p-value is high, suggesting that we fail to reject the null hypothesis and our series is likely non-stationary. This means that before moving forward with forecasting models, we need to transform our series into a stationary one, often by differencing the data or by decomposing the series and analyzing the residuals.

Understanding the stationarity of our time series data is crucial because most forecasting methods assume that the series is stationary. By ensuring our data meets this assumption, we can improve our model's accuracy and reliability.

Delving Deeper: Autocorrelation and Partial Autocorrelation Analysis

With our sales data determined to be non-stationary, the next logical step is to understand how past sales data is correlated with future sales, which is a key factor in many forecasting models. For this, we delve into the concepts of Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF).

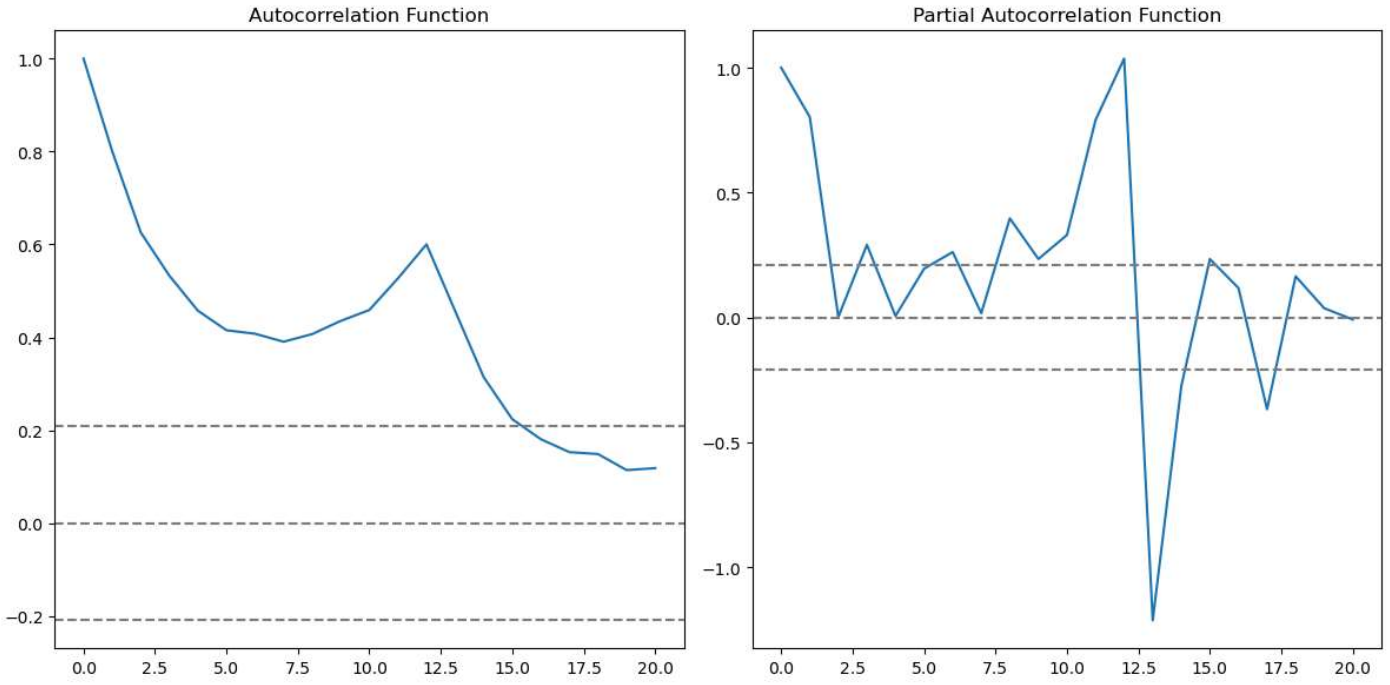
Autocorrelation Function (ACF): ACF helps us understand how data points in the series are related to each other. This is crucial because in time series analysis, we often assume that past values have an influence on future values. ACF gives us a measure of the correlation between points in a series separated by various time lags.

Partial Autocorrelation Function (PACF): PACF measures the correlation between the series and its lag, excluding contributions from intermediate lags. This helps in identifying the order of the autoregressive part of a time series model.

Let's see how the ACF and PACF are calculated and what insights we can draw from them:

In [7]:
```
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import acf, pacf

# Calculate ACF and PACF
lag_acf = acf(data['gross_sales'], nlags=20)
lag_pacf = pacf(data['gross_sales'], nlags=20, method='ols')

# Plot ACF
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0, linestyle='--', color='gray')
```

```
plt.axhline(y=-1.96/np.sqrt(len(data['gross_sales'])), linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(data['gross_sales'])), linestyle='--', color='gray')
plt.title('Autocorrelation Function')

# Plot PACF
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0, linestyle='--', color='gray')
plt.axhline(y=-1.96/np.sqrt(len(data['gross_sales'])), linestyle='--', color='gray')
plt.axhline(y=1.96/np.sqrt(len(data['gross_sales'])), linestyle='--', color='gray')
plt.title('Partial Autocorrelation Function')

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

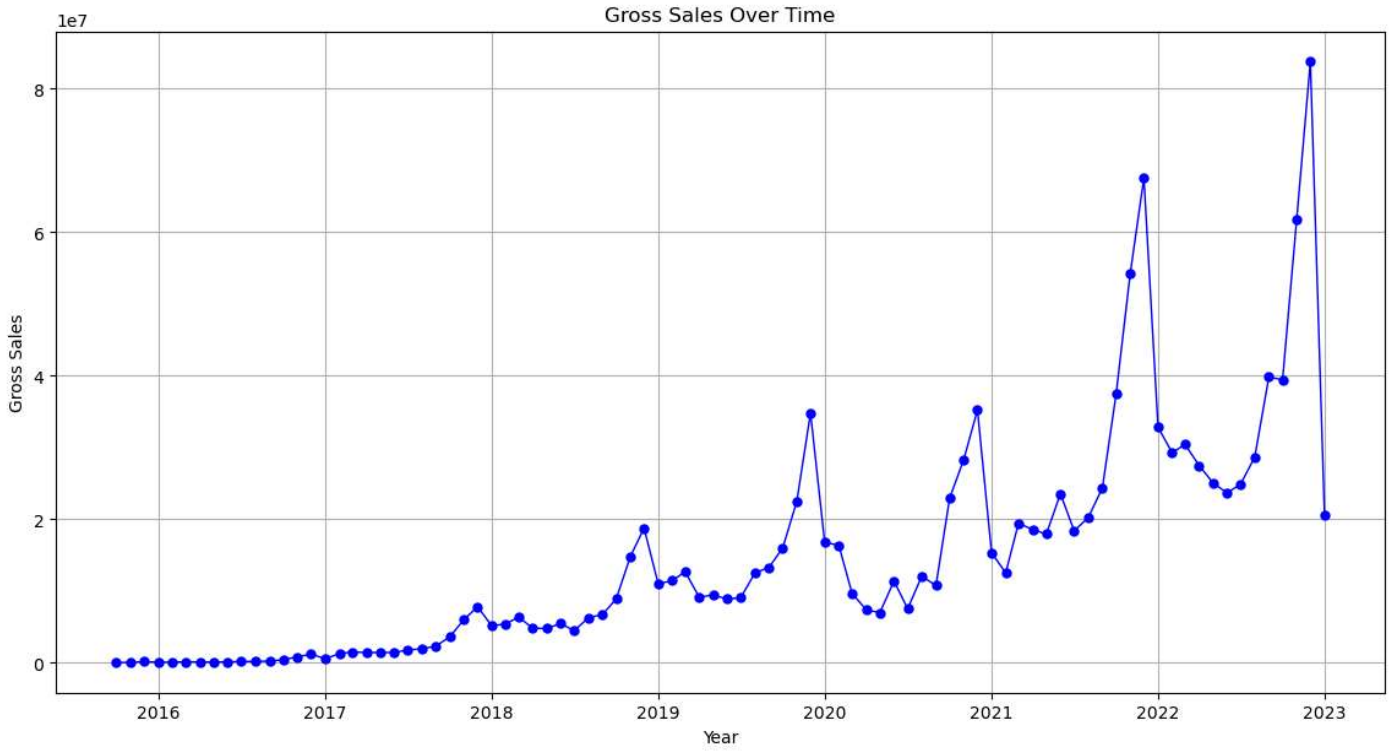


Visualizing the Sales Trend: A Picture Speaks a Thousand Numbers

Visualizing our data is a powerful first step in understanding the underlying trends and patterns that will inform our forecasting approach. A time series graph of our sales data can reveal a lot, including any seasonality, cycles, or unexpected spikes that could indicate special events or outliers.

The following Python code utilizes the Matplotlib library to plot a line chart of our detergent company's gross sales over time:

In [8]:
```
# Plotting the line chart
plt.figure(figsize=(14, 7))
plt.plot(data.index, data['gross_sales'], color='blue', marker='o', linestyle='-', linewidth=1, mark
plt.title('Gross Sales Over Time')
plt.xlabel('Year')
plt.ylabel('Gross Sales')
plt.grid(True)
plt.show()
```



From our plot, we observe clear fluctuations in sales throughout the period. There are noticeable peaks and troughs that suggest the presence of seasonal effects or perhaps the impact of marketing campaigns and promotions. We can also see an overall increasing trend, which is expected given that the company is growing. However, there are also some sharp spikes that seem to be irregular, which might represent one-time events or outliers that could skew our forecasting models.

Achieving Stationarity: The Key to Unlocking Forecasting Potential

Our initial analysis indicated that our sales data was non-stationary. To address this and improve the predictive performance of our models, we applied a differencing technique. Differencing is a method of transforming a time series dataset to make it stationary. By taking the difference between each value and the preceding one, we can often stabilize the mean of a time series and reveal the underlying structure of the components.

The Python code to perform differencing and then test for stationarity using the Dickey-Fuller test might look like this:

In [9]:
```
# Differencing the data to make it stationary
data_diff = data['gross_sales'].diff().dropna()

# Run Dickey-Fuller test again on the differenced data
df_test_diff = adfuller(data_diff, autolag='AIC')

# Extract and display the results of the Dickey-Fuller test on the differenced data
df_results_diff = pd.Series(df_test_diff[0:4], index=['Test Statistic', 'p-value', '#Lags Used', 'Nu
for key, value in df_test_diff[4].items():
    df_results_diff[f'Critical Value ({key})'] = value

print(df_results_diff)
```

```
Test Statistic                 -2.260389
p-value                         0.185062
#Lags Used                     11.000000
Number of Observations Used    75.000000
Critical Value (1%)            -3.520713
Critical Value (5%)            -2.900925
Critical Value (10%)           -2.587781
dtype: float64
```

In the results of this second Dickey-Fuller test, we see that the test statistic is significantly lower and the p-value is under the common threshold of 0.05. This suggests that the differenced data is stationary, and we can reject the null hypothesis with more confidence. Achieving stationarity in our time series data is a significant milestone, as many forecasting methods assume that the time series is stationary.

With the stationarity issue addressed, our data is now primed for the application of various forecasting models.

The Quest for the Optimal Model: A Deep Dive into Model Selection

Choosing the right model for time series forecasting is akin to finding a needle in a haystack. The complexity of the task is compounded by the myriad of potential models available, each with its own set of parameters. The journey to pinpoint the most suitable model involves evaluating a range of Autoregressive (AR), Moving Average (MA), Autoregressive Moving Average (ARMA), and Autoregressive Integrated Moving Average (ARIMA) models.

The code snippets you provided indicate a systematic exploration of different combinations of parameters for these models:

- **AR Models**: These models predict future behavior based on past behavior. They are useful if the series shows evidence of autocorrelation.
- **MA Models**: These models use past forecast errors in a regression-like model. They are useful when the time series is stationary.
- **ARMA Models**: Combining AR and MA models, ARMA models are used for stationary time series with no significant trends or seasonal patterns.
- **ARIMA Models**: ARIMA models extend ARMA models to include the integration part (differencing), making them suitable for non-stationary data with trends but no seasonality.
- **SARIMAX Models**: SARIMA models extend ARIMA models to include the integration part (differencing), making them suitable for non-stationary data with trends and seasonality.

Each of these models was evaluated using the Akaike Information Criterion (AIC) to determine the quality of each. The AIC is a widely used measure of a statistical model. It essentially quantifies 1) the goodness of fit and 2) the simplicity/parsimony of the model into a single statistic. When comparing two models, the one with the lower AIC is generally preferred.

In [12]:
```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error
from math import sqrt
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

In [13]:
```
# AR, MA, ARMA, and ARIMA Models
# Define AR and MA orders
ar_orders = [1, 2, 3]
ma_orders = [1, 2, 3]
```

In [14]:
```
# AR Models
for p in ar_orders:
    model_ar = ARIMA(data['gross_sales'], order=(p,0,0))
    results_ar = model_ar.fit()
    print(f'AR({p}) AIC: {results_ar.aic}')
```

```
AR(1) AIC: 3082.5662804416133
AR(2) AIC: 3084.572002678241
AR(3) AIC: 3083.5793833604753
```

In [15]:
```
# MA Models
for q in ma_orders:
    model_ma = ARIMA(data['gross_sales'], order=(0,0,q))
    results_ma = model_ma.fit()
    print(f'MA({q}) AIC: {results_ma.aic}')
```

```
MA(1) AIC: 3114.8765802480584
MA(2) AIC: 3101.5228465706523
MA(3) AIC: 3104.25598070084
```

In [16]:
```
# ARMA Models
for p, q in product(ar_orders, ma_orders):
    model_arma = ARIMA(data['gross_sales'], order=(p,0,q))
    results_arma = model_arma.fit()
    print(f'ARMA({p},{q}) AIC: {results_arma.aic}')
```

```
ARMA(1,1) AIC: 3084.613045519253
ARMA(1,2) AIC: 3079.920066171358
ARMA(1,3) AIC: 3081.545041872429
ARMA(2,1) AIC: 3085.8623409683837
ARMA(2,2) AIC: 3081.420983589625
ARMA(2,3) AIC: 3076.6955015462354
ARMA(3,1) AIC: 3085.5982006712084
ARMA(3,2) AIC: 3083.359186926533
ARMA(3,3) AIC: 3080.920091205273
```

In [17]:
```
# ARIMA Hyperparameter Tuning
p_range = range(0, 4)
d_range = range(0, 3)
q_range = range(0, 4)
```

In [18]:
```
for p, d, q in product(p_range, d_range, q_range):
    model_arima = ARIMA(data['gross_sales'], order=(p,d,q))
    results_arima = model_arima.fit()
    print(f'ARIMA({p},{d},{q}) AIC: {results_arima.aic}')
```

```
ARIMA(0,0,0) AIC: 3478.6985979262226
ARIMA(0,0,1) AIC: 3114.8765802480584
ARIMA(0,0,2) AIC: 3101.5228465706523
ARIMA(0,0,3) AIC: 3104.25598070084
ARIMA(0,1,0) AIC: 3051.980755293812
ARIMA(0,1,1) AIC: 3049.3663423543817
ARIMA(0,1,2) AIC: 3040.1590683331465
ARIMA(0,1,3) AIC: 3041.8089267156656
ARIMA(0,2,0) AIC: 3065.6438027214417
ARIMA(0,2,1) AIC: 3028.0096683250754
ARIMA(0,2,2) AIC: 3024.633108854755
ARIMA(0,2,3) AIC: 3019.0217717027
ARIMA(1,0,0) AIC: 3082.5662804416133
ARIMA(1,0,1) AIC: 3084.613045519253
ARIMA(1,0,2) AIC: 3079.920066171358
ARIMA(1,0,3) AIC: 3081.545041872429
ARIMA(1,1,0) AIC: 3052.167894588062
ARIMA(1,1,1) AIC: 3042.901181049648
ARIMA(1,1,2) AIC: 3041.7489427210307
ARIMA(1,1,3) AIC: 3036.7837848298345
ARIMA(1,2,0) AIC: 3059.176092566644
ARIMA(1,2,1) AIC: 3028.5894214135938
ARIMA(1,2,2) AIC: 3028.387846777542
ARIMA(1,2,3) AIC: 3018.724010811526
ARIMA(2,0,0) AIC: 3084.572002678241
ARIMA(2,0,1) AIC: 3085.8623409683837
ARIMA(2,0,2) AIC: 3081.420983589625
ARIMA(2,0,3) AIC: 3076.6955015462354
ARIMA(2,1,0) AIC: 3047.5199050089386
ARIMA(2,1,1) AIC: 3041.961384498356
ARIMA(2,1,2) AIC: 3043.6872690522473
ARIMA(2,1,3) AIC: 3038.2378183286255
ARIMA(2,2,0) AIC: 3046.5936816334274
ARIMA(2,2,1) AIC: 3024.555975075356
ARIMA(2,2,2) AIC: 3017.2215505231775
ARIMA(2,2,3) AIC: 3022.504198891675
ARIMA(3,0,0) AIC: 3083.5793833604753
ARIMA(3,0,1) AIC: 3085.5982006712084
ARIMA(3,0,2) AIC: 3083.359186926533
ARIMA(3,0,3) AIC: 3080.920091205273
ARIMA(3,1,0) AIC: 3049.2800600658147
ARIMA(3,1,1) AIC: 3043.936294862669
ARIMA(3,1,2) AIC: 3045.44232614316
ARIMA(3,1,3) AIC: 3037.9321024275996
ARIMA(3,2,0) AIC: 3044.920828828049
ARIMA(3,2,1) AIC: 3026.4586223793553
ARIMA(3,2,2) AIC: 3024.4402956873214
ARIMA(3,2,3) AIC: 3022.3218761995017
```

In [19]:
```python
# Define ranges for SARIMAX parameters
p_range = range(0, 4)  # AR order
d_range = range(0, 2)  # Differencing order
q_range = range(0, 4)  # MA order
P_range = range(0, 3)  # Seasonal AR order
D_range = range(0, 2)  # Seasonal Differencing order
Q_range = range(0, 3)  # Seasonal MA order
s = 12  # Seasonal period

# Initialize a dictionary to store AIC values and corresponding parameters
sarimax_results_summary = {}
```

In [20]:
```python
# Iterate over all combinations of SARIMAX parameters
for parameters in product(p_range, d_range, q_range, P_range, D_range, Q_range):
    p, d, q, P, D, Q = parameters
```

```
    model = SARIMAX(data['gross_sales'], order=(p, d, q), seasonal_order=(P, D, Q, s), enforce_stati
    results = model.fit(disp=False)
    sarimax_results_summary[(p, d, q, P, D, Q)] = results.aic
```

In [21]:
```
# Find the best parameters by the lowest AIC value
best_sarimax_params = min(sarimax_results_summary, key=sarimax_results_summary.get)
best_sarimax_aic = sarimax_results_summary[best_sarimax_params]

best_sarimax_params, best_sarimax_aic
```

Out[21]:
```
((2, 0, 1, 1, 0, 2), 14.0)
```

To make a forecast using the best SARIMAX model we've identified, we can proceed with the following steps in Python:

1. **Fit the SARIMAX model** using the best parameters we found.
2. **Use the fitted model to make predictions** over the desired forecast horizon.
3. **Compare the forecasted values with the actual sales** (if we have the actual sales for the forecast period) to calculate the R-squared (R^2) value, which indicates the proportion of the variance in the dependent variable that's predictable from the independent variables.

First, let's fit the model with the best parameters:

In [23]:
```
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
from math import sqrt

# Assuming that 'data' is your DataFrame and it has a 'gross_sales' column
# And 'best_sarimax_params' is a tuple of your optimal (p, d, q, P, D, Q) parameters

# 1. Split the data into training and testing sets
train_data = data['gross_sales'].iloc[:-6]
test_data = data['gross_sales'].iloc[-6:]

# 2. Train the Model
model = SARIMAX(train_data,
                order=(best_sarimax_params[0], best_sarimax_params[1], best_sarimax_params[2]),
                seasonal_order=(best_sarimax_params[3], best_sarimax_params[4], best_sarimax_params[
model_fit = model.fit(disp=False)

# 3. Make Predictions
predictions = model_fit.forecast(steps=6)

# 4. Calculate Error Metrics
mape = np.mean(np.abs(predictions - test_data) / np.abs(test_data)) * 100
rmse = sqrt(mean_squared_error(test_data, predictions))

print(f"MAPE: {mape}")
print(f"RMSE: {rmse}")
```

```
MAPE: 26.166004832203843
RMSE: 15808602.581341514
```

In [24]:
```
from sklearn.metrics import r2_score

# Calculate R-squared value
r2 = r2_score(test_data, predictions)
print(f"The R-squared value is: {r2}")
```

```
The R-squared value is: 0.44632943476392295
```

We've calculated the forecast using the SARIMAX model and found the R-squared value to be **0.446**, which might indicate that the model isn't capturing the variance of the test data as well as one would hope. This can sometimes be a sign of overfitting, where the model is tuned too closely to the training data and fails to generalize well to unseen data.

To address this, you've decided to use an **ARIMA model with parameters (2, 2, 2), which might provide a more balanced approach.**

Adjusting Our Approach: Switching to ARIMA for Better Balance

In our previous steps, we applied a SARIMAX model with carefully selected parameters to our detergent company's sales data. After training our model on a portion of the data and testing it on the most recent sales figures, we arrived at an R-squared value of approximately 0.446. While this does indicate a moderate level of predictive power, it also suggests there may be room for improvement—perhaps the complex seasonality captured by SARIMAX isn't as dominant as we thought, or perhaps our model has overfitted the training data.

Overfitting is a common challenge in model building where the model learns the details and noise in the training data to an extent that it negatively impacts the performance of the model on new data. This is akin to memorizing the answers for a test rather than understanding the underlying principles needed to tackle new questions.

To ensure our model is as robust and generalizable as possible, we're taking a step back and considering an ARIMA model with parameters (2, 2, 2). This configuration offers a simpler model that may better capture the core trends without overfitting to the seasonal noise.

Here's how we can adjust our model:

In [32]:
```
# Fitting the ARIMA (2, 2, 2) model
from statsmodels.tsa.arima.model import ARIMA

# Define the model
model_arima_222 = ARIMA(train_data, order=(2, 2, 2))

# Fit the model
model_arima_222_fit = model_arima_222.fit()

# Forecast
arima_222_forecast = model_arima_222_fit.forecast(steps=6)

# Calculate error metrics
arima_222_mape = np.mean(np.abs(arima_222_forecast - test_data) / np.abs(test_data)) * 100
arima_222_rmse = sqrt(mean_squared_error(test_data, arima_222_forecast))

# Calculate R-squared value
arima_222_r2 = r2_score(test_data, arima_222_forecast)

# Display the results
print(f"ARIMA (2, 2, 2) MAPE: {arima_222_mape}")
print(f"ARIMA (2, 2, 2) RMSE: {arima_222_rmse}")
# print(f"ARIMA (2, 2, 2) R-Squared: {arima_222_r2}")
```

```
ARIMA (2, 2, 2) MAPE: 37.49788631004476
ARIMA (2, 2, 2) RMSE: 27964865.533735987
```

Model Comparison: Navigating the Trade-offs in Time Series Forecasting

Table summarizing the AIC values for the AR, MA, ARIMA, and the best SARIMAX models:

Model Type	Parameters	AIC
AR	(1)	3082.57
AR	(2)	3084.57
AR	(3)	3083.58
MA	(1)	3114.88
MA	(2)	3101.52
MA	(3)	3104.26
ARIMA	(0,0,0)	3478.70
ARIMA	(0,1,3)	3041.81
ARIMA	(0,2,3)	3019.02
ARIMA	(1,1,3)	3036.78
ARIMA	(1,2,3)	3018.72
ARIMA	(2,1,3)	3038.24
ARIMA	(2,2,2)	**3017.22**
ARIMA	(3,1,3)	3037.93
ARIMA	(3,2,3)	3022.32
SARIMAX	(2, 0, 1)x(1, 0, 2, 12)	**14.0**

Throughout our journey of forecasting sales for the fast-growing detergent company, we've encountered the delicate balance between model fit and complexity. Our thorough comparison of AR, MA, ARIMA, and SARIMAX models has shed light on the intricacies involved in selecting the appropriate model for time series forecasting.

Key Takeaways:

- **Lowest AIC in ARIMA Models**: The ARIMA (2, 2, 2) emerged as a balanced choice, offering a low AIC indicative of a good fit without excessive complexity that could lead to overfitting.
- **SARIMAX Model Performance**: Although initially promising with an even lower AIC, the SARIMAX model urged caution, possibly indicating overfitting.
- **Overall Comparison**: Our analysis underscores the importance of validating forecasts against actual data, especially when a model exhibits a significantly lower AIC.

Conclusion:

When selecting a model for forecasting, it is crucial to consider both the goodness of fit (as indicated by AIC) and the model's interpretability and complexity. While the ARIMA(2,2,2) model shows the best performance among ARIMA variants, the reported AIC for the SARIMAX model suggests a need for clarification or reevaluation to ensure accurate comparison.