Forecasting Truck Sales using SARIMA

Time Series Analysis and Predictive Modeling

- Nevil Hitesh Mehta

Agenda

Introduction

Data Overview

Data Pre-Processing

Model Development

- Train-Test Split
- Time Series Decomposition
- Rolling Statistics
- Stationarity Test
- ACF & PACF Plots
- SARIMA (Manual & Automatic)

Future Sales Forecast

Introduction

Purpose: To forecast future sales of Velocity Wheels (Truck)

Tools Used: Python

Model Implemented: SARIMA

Importance of Forecasting Sales in Business:

- **Informed Decision-Making**: Accurate forecasts guide strategic planning and resource allocation, mitigating risks.
- **Optimal Inventory Control**: Predicting demand prevents overstocking and stockouts, enhancing customer satisfaction.
- **Enhanced Operational Efficiency**: Streamlined logistics and supplier management improve order processing and performance.

Data Overview

Source: Dataset has been downloaded from Kaggle

Weblink: https://www.kaggle.com/datasets/willianoliveiragibin/velocity-wheels

First 5 rows of Dataset are as follows:

	Month-Year	Number_Trucks_Sold
0	03-jan.	155
1	03-Feb	173
2	03-mar.	204
3	03-Apr	219
4	03-May	223

Data Pre-Processing

- Checking the shape of data type
- Checking presence of null values
- Formatting the Date column
- Removing unnecessary columns

```
df.shape
```

```
(144, 2)
```

```
df.isnull().sum()
```

	0
Month-Year	0
Number_Trucks_Sold	0

dtype: int64

Data Pre-Processing: Formatting Date Column

```python
def convert_to_mm_yyyy(Month_Year):
  year,month = Month_Year.split('-')
  cleaned_month = month.strip('.').capitalize()[:3]
  full_year = '20' + year
  full_date = f'{cleaned_month}{full_year}'
  return pd.to_datetime(full_date, format = '%b%Y').strftime('%Y-%m')

df['Date'] = df['Month-Year'].apply(convert_to_mm_yyyy)

print(df)
```

```
    Month-Year  Number_Trucks_Sold     Date
0      03-jan.                 155  2003-01
1       03-Feb                 173  2003-02
2      03-mar.                 204  2003-03
3       03-Apr                 219  2003-04
4       03-May                 223  2003-05
..         ...                 ...      ...
139     14-Aug                 933  2014-08
140     14-Sep                 704  2014-09
141     14-Oct                 639  2014-10
142    14-nov.                 571  2014-11
143     14-Dec                 666  2014-12

[144 rows x 3 columns]
```

Data Pre-Processing: Removing Unnecessary Columns

```
[9] df = df.drop('Month-Year', axis=1)
```

```
df.head()
```

	Number_Trucks_Sold	Date
0	155	2003-01
1	173	2003-02
2	204	2003-03
3	219	2003-04
4	223	2003-05

```
[11] df = df.iloc[:,[1,0]]

     df.head()
```

	Date	Number_Trucks_Sold
0	2003-01	155
1	2003-02	173
2	2003-03	204
3	2003-04	219
4	2003-05	223

Train-Test Split

Split ratio taken here is 70:30

```
# Train-Test split

train_data = df[:101]
test_data = df[101:144]

train_data = train_data.set_index('Date')
```

```
train_data
```

	Number_Trucks_Sold
Date	
2003-01	155
2003-02	173
2003-03	204
2003-04	219
2003-05	223
...	...
2011-01	437
2011-02	440
2011-03	548
2011-04	590
2011-05	656

101 rows × 1 columns

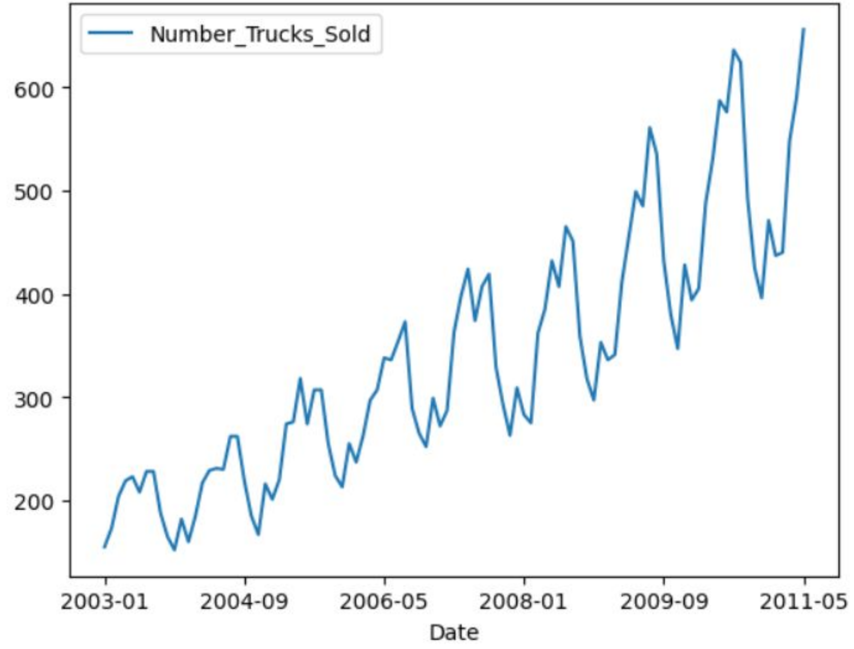
Time Series : EDA

- Crucial step in understanding time series characteristics.
- Helps in identifying trends, patterns and anomalies.
- Helps in identifying Seasonality and Stationarity.
- Key steps in EDA for time series data include:
  - **Visualization:** Plotting the time series to understand general trends.
  - **Descriptive Statistics:** Calculating metrics like mean, variance, and autocorrelation.
  - **Rolling Statistics:** Find the rolling mean and std.deviation to identify the trend.
  - **Stationarity Tests:** Checking if the statistical properties of the series are constant over time.

EDA : Visualization

```
train_data.plot()
```

```
<Axes: xlabel='Date'>
```



Descriptive Statistics

```
train_data.describe()
```

	Number_Trucks_Sold
count	101.000000
mean	337.148515
std	122.009949
min	152.000000
25%	231.000000
50%	309.000000
75%	419.000000
max	656.000000

Rolling Statistics

Calculations performed on a time series dataset over a fixed window of observations.

Useful for understanding trends and patterns over time.

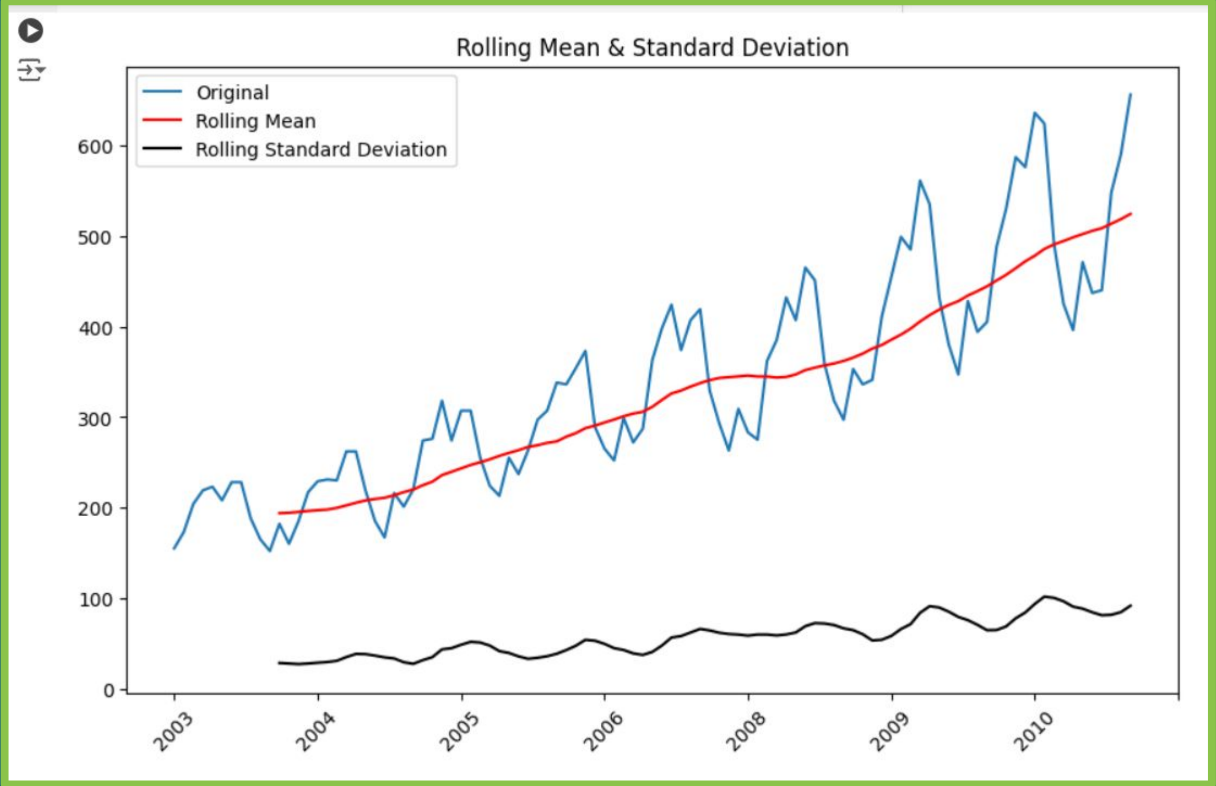
Common Rolling Statistics:

- **Rolling Mean**: The average of values within a specified window that moves across the data.
- **Rolling Standard Deviation**: Measures the variability of values within the window. It provides insights into the volatility of the data over time.
- **Rolling Variance**: Similar to the rolling standard deviation but provides the squared variability within the window.

Rolling Statistics

```
# Rolling Statistics

from matplotlib.ticker import MaxNLocator

rolling_mean = train_data['Number_Trucks_Sold'].rolling(window = 12).mean() # 12 period rolling mean
rolling_std = train_data['Number_Trucks_Sold'].rolling(window = 12).std() # 12 period rolling standard deviation

# Plotting the values

plt.figure(figsize = (10,6))
plt.plot(train_data['Number_Trucks_Sold'], label = 'Original')
plt.plot(rolling_mean, label = 'Rolling Mean', color = 'red')
plt.plot(rolling_std, label = 'Rolling Standard Deviation', color = 'black')
plt.title('Rolling Mean & Standard Deviation')
truncated_labels = [str(label)[:4] for label in train_data.index]  # Truncate to first 7 characters
plt.xticks(train_data.index, truncated_labels, rotation=45)  # Rotate for readability
ax = plt.gca() # Gets current axis
ax.xaxis.set_major_locator(MaxNLocator(nbins=8))
plt.legend()
plt.show()
```

Rolling Statistics



Stationarity Test

Used to determine whether a time series data exhibits stationarity.

Stationary Series: Mean and Variance do not change with time.

Augmented Dickey Fuller (ADF) Test is used in this case.

Test Criteria : p-value (typically < 0.05) indicates that the series is stationary.

Stationarity Test

```python
# Stationarity Test

from statsmodels.tsa.stattools import adfuller
```

```python
def adf_test(series):
  result = adfuller(series)
  print('ADF Statistics: {}'.format(result[0]))
  print('p-value: {}'.format(result[1]))
  if result[1] <= 0.05:
    print('Strong evidence against the null hypothesis. Hence, null hypothesis is rejected indicating data is stationery.')
  else:
    print('Weak evidence against the null hypothesis. Hence, null hypothesis is accepted indicating data is not stationery.')
```

```python
adf_test(train_data['Number_Trucks_Sold'])
```

```
ADF Statistics: 0.9103273316294213
p-value: 0.9932236315543095
Weak evidence against the null hypothesis. Hence, null hypothesis is accepted indicating data is not stationery.
```

Differencing

Technique used for making a series stationary.

It removes any trends or seasonality present in the series.

For this we subtract the current observation from previous one.

Types of Differencing:

- **First Differencing**: The difference between consecutive data points (e.g., $Yt-Y(t-1)$) to remove linear trends.
- **Seasonal Differencing**: The difference between observations separated by a seasonal period (e.g., $Yt-Y(t-12)$), where 12 is the seasonal length) to remove seasonal patterns.

Differencing helps stabilize the mean and variance, making the data suitable for modeling.

1st Differencing

```
[ ] # Since our data is seasonal so we will need to do 12 months differencing

    train_data['Trucks_12_diff'] = train_data['Number_Trucks_Sold']-train_data['Number_Trucks_Sold'].shift(12)

    train_data.head(15)
```

```
[ ] train_data.head(15)
```

Date	Number_Trucks_Sold	Trucks_1st_diff	Trucks_2nd_diff	Trucks_12_diff
2003-01	155	NaN	NaN	NaN
2003-02	173	18.0	NaN	NaN
2003-03	204	31.0	13.0	NaN
2003-04	219	15.0	-16.0	NaN
2003-05	223	4.0	-11.0	NaN
2003-06	208	-15.0	-19.0	NaN
2003-07	228	20.0	35.0	NaN
2003-08	228	0.0	-20.0	NaN
2003-09	188	-40.0	-40.0	NaN
2003-10	165	-23.0	17.0	NaN
2003-11	152	-13.0	10.0	NaN
2003-12	182	30.0	43.0	NaN
2004-01	160	-22.0	-52.0	5.0
2004-02	185	25.0	47.0	12.0
2004-03	217	32.0	7.0	13.0

2nd and 3rd Differencing

```
train_data['Trucks_2nd_12_diff'] = train_data['Trucks_12_diff']-train_data['Trucks_12_diff'].shift(12)
```

```
adf_test(train_data['Trucks_2nd_12_diff'].dropna())
```

```
ADF Statistics: -2.256908311627945
p-value: 0.1862425827745111
Weak evidence against the null hypothesis. Hence, null hypothesis is accepted indicating data is not stationery.
```

```
train_data['Trucks_3rd_12_diff'] = train_data['Trucks_2nd_12_diff']-train_data['Trucks_2nd_12_diff'].shift(12)
```

```
adf_test(train_data['Trucks_3rd_12_diff'].dropna())
```

```
ADF Statistics: -5.266988473215438
p-value: 6.407011676003551e-06
Strong evidence against the null hypothesis. Hence, null hypothesis is rejected indicating data is stationery.
```

Differencing : Insights

1st Differencing (p-value = 0.058):

- The p-value is close to 0.05, but slightly above it. While technically, this suggests the series is not yet stationary, it’s quite close to being stationary.

2nd Differencing (p-value = 0.18):

- The p-value increases, which suggests that the second differencing is making the series less stationary.

3rd Differencing (p-value < 0.05):

- The p-value finally drops below 0.05, indicating stationarity. But this might indicate we've over-differenced the data, potentially removing too much of the original signal.

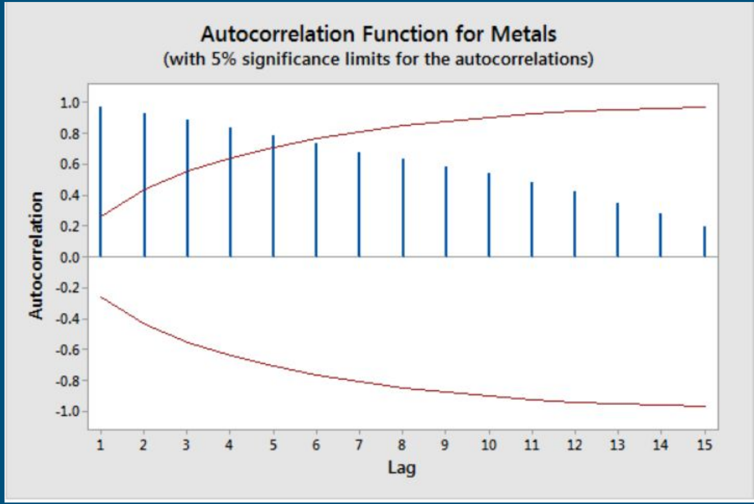
Hence, we go with 1st Differencing, since it is closer to p-value

Autocorrelation Function (ACF)

- Measures the correlation between observations of a time series separated by different time lags.
- Help in identifying the strength of correlation between current values and past values in the time series.
- Diagnosing whether the series is stationary or contains trends or seasonality.
- Determine the order of moving average (MA) components in ARIMA and SARIMA models.

Example is provided below.

Img: https://statisticsbyjim.com/time-series



Autocorrelation Function (ACF)

How the ACF Works:

- The ACF calculates the correlation of a time series with itself at various lags.
  - **Lag 1**: The correlation between the current value and the value 1 time step earlier.
  - **Lag 2**: The correlation between the current value and the value 2 time steps earlier, and so on.
- The correlations range between -1 and +1:
  - **+1**: Perfect positive correlation (the current and lagged values move in the same direction).
  - **-1**: Perfect negative correlation (the current and lagged values move in opposite directions).
  - **0**: No correlation (the lagged values have no linear relationship with the current values).

Autocorrelation Function (ACF): Interpretation

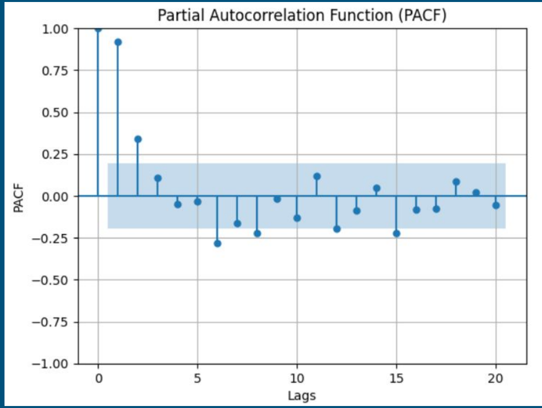
Significant Spikes: Spikes in the ACF plot outside the confidence bounds (often shown as a shaded area) indicate significant autocorrelation at that lag.

- **Gradual Decline**: A slow decay in the ACF indicates non-stationarity or the presence of a trend.
- **Seasonality**: Peaks at regular intervals suggest seasonality (e.g., a spike at lag 12 for monthly data with yearly seasonality).

White Noise: If all points in the ACF plot fall within the confidence interval, the time series is likely white noise, meaning it has no significant autocorrelation and is unpredictable.

Partial Autocorrelation Function (PACF)

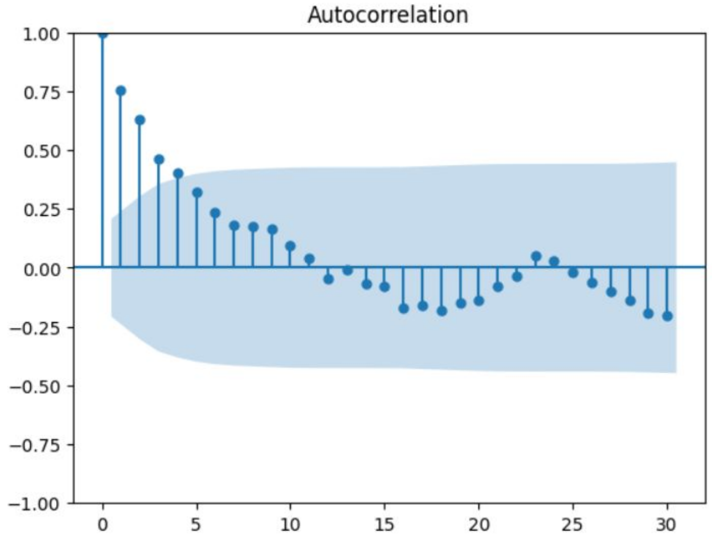
- Measures the correlation between a time series and its lagged values.
- Unlike ACF, PACF **controls for intermediate lags**.
- Isolates the **direct relationship** between the current observation and a specific lag.
- **How PACF works?**
  - At **lag 1**, PACF = ACF (no previous lags to account for).
  - At **lag 2**, PACF measures correlation between Yt and Yt−2 after removing effect of Yt−1.
  - Continues for higher lags by controlling for all shorter lags.
- Identifies **autoregressive (AR) order** in ARIMA models.
- Img: geeksforgeeks
  -



Autocorrelation Function (ACF) Plot

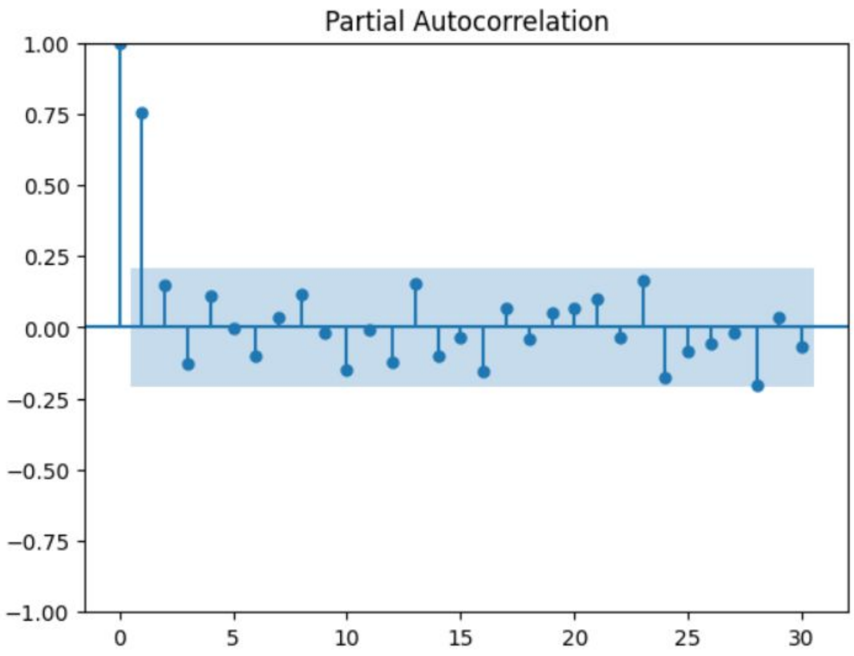
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

```
acf = plot_acf(train_data['Trucks_12_diff'].dropna(), lags = 30)
```



Partial Autocorrelation Function (PACF) Plot

```
pacf = plot_pacf(train_data['Trucks_12_diff'].dropna(), lags = 30)
```



✓ 0s completed at 8:17 PM

SARIMA Model

- Seasonal Autoregressive Integrated Moving Averages (SARIMA)
- Extension of the ARIMA model.
- Captures both non-seasonal and seasonal patterns in time series data.
- Key Components of SARIMA
  - **p, d, q**: Non-seasonal ARIMA components.
    - **p**: Number of autoregressive terms (AR).
    - **d**: Differencing to make data stationary (I).
    - **q**: Number of moving average terms (MA).
  - **P, D, Q, s**: Seasonal ARIMA components.
    - **P**: Number of seasonal autoregressive terms.
    - **D**: Seasonal differencing.
    - **Q**: Number of seasonal moving average terms.
    - **s**: Length of the seasonal cycle (e.g., 12 for monthly data with yearly seasonality).

Manual Detection of Parameters

For Non-Seasonal Model:

- ACF is decaying exponentially.
- In PACF we have 1 significant spike.
- Value of d = 2 (As we saw, we needed to difference it twice in non-seasonal).
- Hence, we will consider AR(1) model ($p = 1$, $q = 0$)

For Seasonal Model:

- D = 1 (Considering value of $p = 0.058$ as satisfactory)
- No seasonalities are found in ACF, PACF plots, we will go with $P = 0$, $Q = 0$.
- S = 12 for accounting monthly data

SARIMA: Manual

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

```
# Prediction using manual detection of SARIMA parameters

# For non-seasonal model, as stated above we will go with AR(1) model
p,d,q = 1,2,0

P,D,Q,s = 0,1,0,12

sarima_model = SARIMAX(train_data['Number_Trucks_Sold'], order = (p,d,q), seasonal_order = (P,D,Q,s))
sarima_model_fit = sarima_model.fit(disp = False)
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency information was provided, so inferred f
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency information was provided, so inferred f
  self._init_dates(dates, freq)
```

```
# Forecasting

n_test = len(test_data['Number_Trucks_Sold'])
forecast = sarima_model_fit.forecast(steps = n_test)
```

SARIMA: Manual

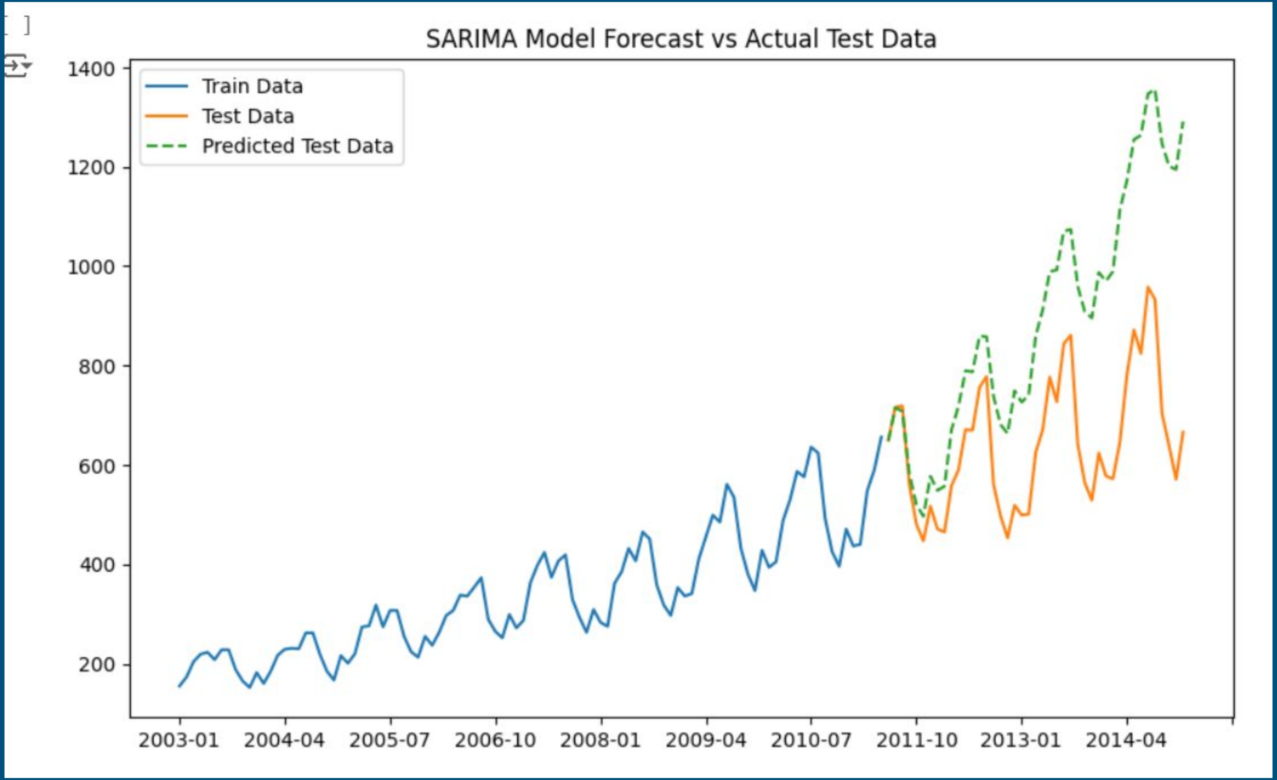
```
[ ]  # Finding RMSE
     from sklearn.metrics import mean_squared_error

     test_actual = test_data['Number_Trucks_Sold'].values
     test_predicted = forecast.values
     rmse = np.sqrt(mean_squared_error(test_actual, test_predicted))
     print(f"Root Mean Squared Error (RMSE): {rmse}")
```

```
Root Mean Squared Error (RMSE): 302.70261731074197
```

SARIMA: Manual

```
[ ]  import matplotlib.pyplot as plt

     plt.figure(figsize = (10,6))
     plt.plot(train_data.index, train_data['Number_Trucks_Sold'].values, label='Train Data')
     plt.plot(test_data.index, test_actual, label='Test Data')
     plt.plot(test_data.index, test_predicted, label='Predicted Test Data', linestyle='--')
     plt.legend(loc='upper left')
     plt.title('SARIMA Model Forecast vs Actual Test Data')
     ax = plt.gca() # Gets current axis
     ax.xaxis.set_major_locator(MaxNLocator(nbins=12))
     plt.show()
```

SARIMA: Manual



SARIMA: Autodetection of Parameters

```
# Trying to find best values using iterations for m = 12
import itertools
from statsmodels.tsa.statespace.sarimax import SARIMAX
import warnings

# Ignore warnings related to statsmodels convergence
warnings.filterwarnings("ignore")

series = train_data['Number_Trucks_Sold']

# Define the range of parameters to test
p = d = q = range(0, 3)
P = D = Q = range(0, 3)
m = 12

# Create a list with all combinations of p, d, q, P, D, Q values
param_combinations = list(itertools.product(p, d, q, P, D, Q))

# Initialize variables to store the best parameters and AIC
best_aic = float("inf")
best_params = None
```

SARIMA: Autodetection of Parameters

```python
# Loop through all parameter combinations
for params in param_combinations:
    try:
        # Extract parameters
        non_seasonal_order = (params[0], params[1], params[2])
        seasonal_order = (params[3], params[4], params[5], m)

        # Fit SARIMA model
        model = SARIMAX(series,
                        order=non_seasonal_order,
                        seasonal_order=seasonal_order,
                        enforce_stationarity=False,
                        enforce_invertibility=False)

        results = model.fit(disp=False)

        # Compare AIC values
        if results.aic < best_aic:
            best_aic = results.aic
            best_params = (non_seasonal_order, seasonal_order)

        # Print current combination and AIC
        print(f"SARIMA{non_seasonal_order}x{seasonal_order} - AIC: {results.aic}")

    except Exception as e:
        print(f"Error for SARIMA{non_seasonal_order}x{seasonal_order}: {e}")
        continue
```

SARIMA: Autodetection of Parameters

```python
# Print the best combination of parameters and its AIC
print("\nBest Model:")
print(f"SARIMA{best_params[0]}x{best_params[1]} - AIC: {best_aic}")
```

```
SARIMA(2, 2, 2)x(2, 2, 1, 12) - AIC: 424.418387587633
SARIMA(2, 2, 2)x(2, 2, 2, 12) - AIC: 411.2543575593715

Best Model:
SARIMA(2, 1, 1)x(0, 1, 2, 12) - AIC: 12.0
```

SARIMA: Autodetection of Parameters

```
# Prediction using auto detection of SARIMA parameters by for loop
# Here we take the best model given by m = 12 as it has the lowest AIC

p,d,q = 2,1,1

P,D,Q,s = 0,1,2,12

sarima_model2 = SARIMAX(train_data['Number_Trucks_Sold'], order = (p,d,q), seasonal_order = (P,D,Q,s))
sarima_model2_fit = sarima_model2.fit(disp = False)
```

```
[ ] # Forecasting

n_test = len(test_data['Number_Trucks_Sold'])
forecast = sarima_model2_fit.forecast(steps = n_test)
```

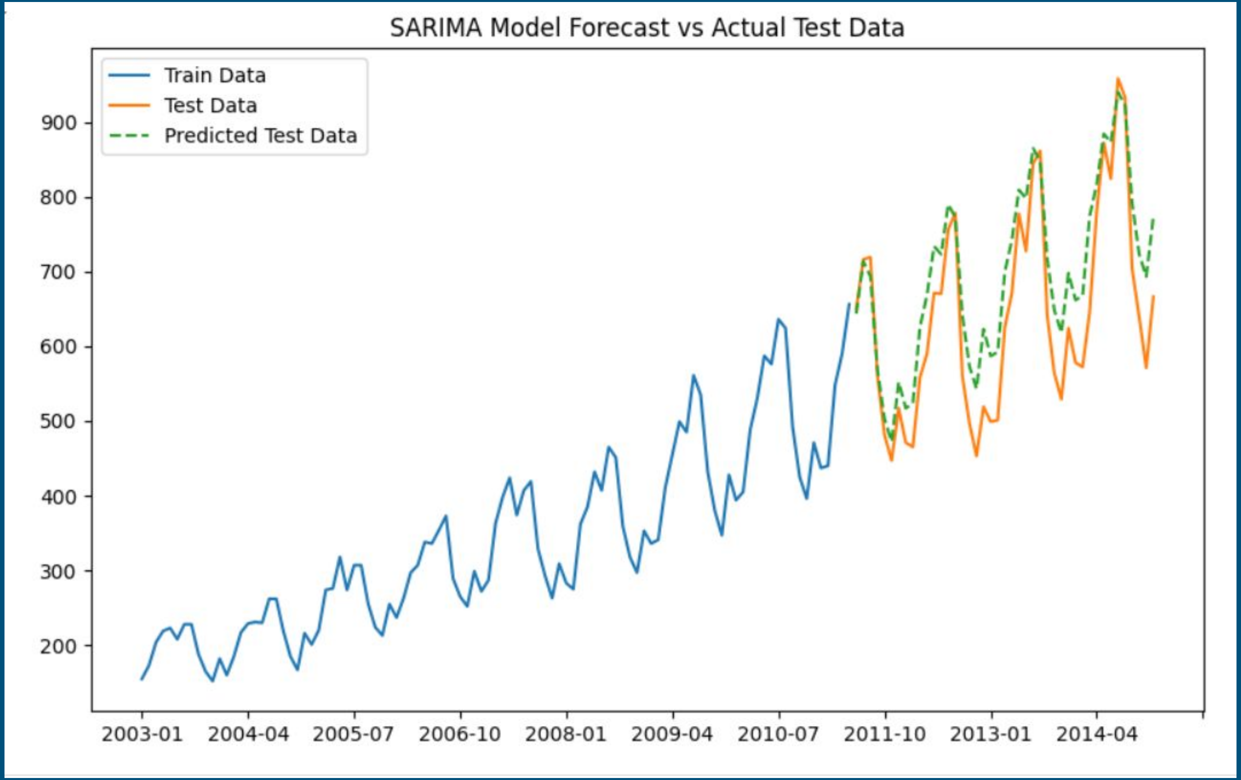
```
[ ] # Finding RMSE
from sklearn.metrics import mean_squared_error

test_actual = test_data['Number_Trucks_Sold'].values
test_predicted2 = forecast.values
rmse = np.sqrt(mean_squared_error(test_actual, test_predicted2))
print(f"Root Mean Squared Error (RMSE): {rmse}")
```

```
Root Mean Squared Error (RMSE): 67.40059180983118
```

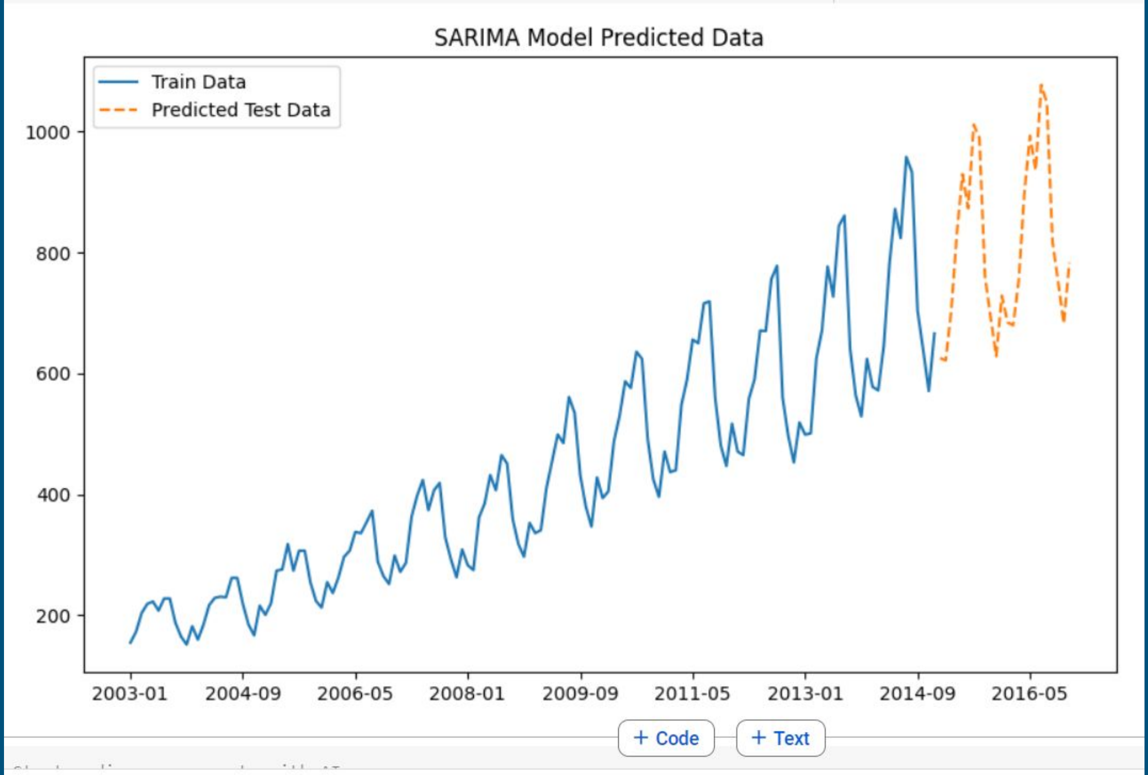
SARIMA: Autodetection of Parameters

```
[ ]  import matplotlib.pyplot as plt

     plt.figure(figsize = (10,6))
     plt.plot(train_data.index, train_data['Number_Trucks_Sold'].values, label='Train Data')
     plt.plot(test_data.index, test_actual, label='Test Data')
     plt.plot(test_data.index, test_predicted2, label='Predicted Test Data', linestyle='--')
     plt.legend(loc='upper left')
     plt.title('SARIMA Model Forecast vs Actual Test Data')
     ax = plt.gca() # Gets current axis
     ax.xaxis.set_major_locator(MaxNLocator(nbins=12))
     plt.show()
```

SARIMA: Autodetection of Parameters



Forecasting Future Data



Findings

RMSE Error through Manual Detection: 302.702

RMSE Error through Auto Detection: 67.4

Percentage Error Reduction: 77.73%

Forecast made for next 2 yrs using Auto detection of SARIMA Model parameters.

Advantages of Auto Detection

Integrating Automation helps in improving prediction by reducing error

Auto detection iterated through 729 combinations by taking a couple of mins!

Manual Detection would have been a lot time consuming as well as error-prone.

```python
import itertools

p = d = q = range(0, 3)
P = D = Q = range(0, 3)
m = 4

# Create a list with all combinations of p, d, q, P, D, Q values
param_combinations = list(itertools.product(p, d, q, P, D, Q))
len(param_combinations)
```

```
729
```

Thank You