Chapter 2: Traditional Forecasting Techniques with Hands-on Implementation Guide

In this chapter, we explore traditional forecasting techniques—Exponential Moving Average (EMA) and Auto-Regressive Integrated Moving Average (ARIMA)—to establish a foundation in demand forecasting. Widely used in stable market environments, these models are implemented in *Google Colab*, providing an interactive guide to forecasting setup, Python libraries, and essential functions. Beyond technical implementation, this chapter also highlights how these methods can support effective decision-making in scenarios like inventory planning and demand prediction, reinforcing core concepts with practical applications.

Learning Outcomes

- **Understand and implement traditional demand forecasting methods** (such as EMA and ARIMA) to gain foundational knowledge in time series forecasting.
- **Develop proficiency with Python libraries** and Functions for data manipulation, model building, performance evaluation, and visualization to streamline the implementation of forecasting models.
- **Gain hands-on experience** by implementing, testing, and modifying forecasting models in *Google Colab* using provided code examples.
- **Evaluate forecasting model performance** using metrics like MAE, RMSE, and MAPE, and visualizations to assess model accuracy and suitability for different demand scenarios.

Understanding Performance Evaluation: Start with the End in Mind

To assess the performance of demand forecasting models, it’s crucial to use metrics that effectively measure how well the model's predictions align with actual demand. These metrics can be thought of as "report cards" for forecasting models, with each highlighting a different

aspect of their performance. The three commonly used metrics are *Root Mean Squared Error (RMSE)*, *Mean Absolute Error (MAE)*, and *Mean Absolute Percentage Error (MAPE)*.

Please note, the math equations for these metrics are provided as technical references; however, the primary focus should be on understanding how to use these metrics in practice and the pros and cons of each, rather than the math and equations themselves.

Root Mean Squared Error (RMSE)

The **Root Mean Squared Error (RMSE)** measures the average magnitude of the errors in the same units as the forecasted values (e.g., units sold). RMSE gives more weight to larger errors, making it sensitive to outliers and large deviations. Mathematically, RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(y_t - \hat{y}_t)^2}$$

Where:

- n is the number of observations in the dataset.
- y_t is the actual value at time t.
- $\hat{y}_t$ is the forecasted value at time t.

Interpretation:

- RMSE is expressed in the same units as the dependent variable (e.g., number of products sold), making it easy to understand in the context of business operations.
- A lower RMSE indicates a better fit of the model to the actual data, meaning the forecasted values are closer to the actual values.

Advantages:

- RMSE is highly sensitive to large errors, which makes it particularly useful in business contexts where large deviations (e.g., underestimating demand during a promotional event) are more critical than small errors.

Limitations:

- RMSE can be skewed by extreme outliers, especially if the data contains significant variations or sudden spikes.

Mean Absolute Error (MAE)

The **Mean Absolute Error (MAE)** is a commonly used metric that measures the average magnitude of errors between actual values and predicted values in a dataset. It is calculated as the average of the absolute differences between actual observations and the predicted values:

$$\text{MAE} = \frac{1}{n} \sum_{t=1}^{n} |y_t - \hat{y}_t|$$

Where:

- n is the number of observations in the dataset.
- y_t is the actual value at time t.
- $\hat{y}_t$ is the forecasted value at time t.

Interpretation:

- MAE measures the average size of the errors in a set of predictions, without considering their direction (whether the errors are positive or negative). It provides a straightforward way to understand how far off the model's predictions are from the actual values, in the same units as the data.
- A lower MAE indicates a better fit of the model, meaning the average error between actual and predicted values is smaller. For example, if the MAE is 5 units, it means that, on average, the model's predictions are off by 5 units from the actual values.

Advantages:

- **Easy to Understand**: MAE is easy to interpret because it represents the average magnitude of errors in the same units as the original data. This makes it intuitive for business managers and those with limited quantitative backgrounds to grasp.
- **Equal Weight to All Errors**: MAE treats all errors equally, making it a reliable metric when you want to get a general sense of the model's accuracy without giving more weight to larger errors.

Limitations:

- **No Penalty for Large Errors**: Unlike other metrics such as RMSE, MAE does not emphasize larger errors, which can be a limitation if your goal is to minimize the impact of large deviations.
- **Sensitive to Outliers**: MAE can still be affected by outliers in the data, as extreme values can distort the overall average, making it less representative of typical performance.

Mean Absolute Percentage Error (MAPE)

The **Mean Absolute Percentage Error (MAPE)** measures the average percentage error between actual and forecasted values. It is defined as:

$$\text{MAPE} = \frac{1}{n} \sum_{t=1}^{n} \left| \frac{y_t - \hat{y}_t}{y_t} \right| \times 100$$

Where:

- n is the number of observations in the dataset.
- y_t is the actual value at time t.
- $\hat{y}_t$ is the forecasted value at time t.

Interpretation:

- MAPE is expressed as a percentage, which makes it useful for comparing the accuracy of models across different datasets or variables.
- A lower MAPE value indicates a smaller percentage error, meaning the model's forecasts are closer to the actual values in relative terms.

Advantages:

- MAPE is easy to interpret as a percentage, making it intuitive for business managers to understand how accurate the model is.
- It is not affected by the scale of the data, allowing it to be used across different datasets with varying units.

Limitations:

- MAPE can become inflated when actual values are close to zero, leading to misleading results in such cases.
- It does not differentiate between overestimating and underestimating errors, treating both equally.

Combining All Three Metrics for Comprehensive Evaluation

Each metric offers unique insights into a model's performance:

- **RMSE** is valuable for highlighting significant errors, particularly during volatile periods.
- **MAE** provides a clear view of the average error size, giving an overall indication of model accuracy.
- **MAPE** contextualizes errors as percentages, which helps understand the impact of mistakes relative to actual demand and makes it easier to compare performance across different scenarios.

By using these three metrics together, you gain a comprehensive view of your model's performance, helping you decide on the best forecasting approach for your specific needs—whether your goal is to minimize large surprises, understand the general accuracy, or compare performance across diverse datasets and conditions. In the later chapters, we will also explore $\mathbf{R^2}$ to provide additional insights into the goodness-of-fit for more advanced models.

The Scene: Forecasting Demand Fluctuations for a Grocery Store

Managing inventory levels in a grocery store is always a balancing act—ensuring popular items are available while keeping waste to a minimum. Unexpected events, such as sudden changes in the weather, often make this balancing act even more challenging. For example, a cold snap can lead to a spike in demand for comfort foods, blankets, and other essentials, while a heatwave can result in increased sales of cold drinks, ice cream, and fans. These sudden shifts in customer behavior can be difficult to predict and, if not managed properly, can leave shelves empty or lead to costly overstock.

Store managers face the challenge of anticipating how much inventory to order to meet these fluctuating demands without overcommitting to products that may not sell. With a weather event approaching, they need to make decisions about how to adjust inventory levels to ensure they are prepared. Stocking too much could mean excess goods that go to waste, while stocking too little might mean disappointed customers and missed opportunities.

The need to respond effectively to sudden changes like these highlights the importance of having flexible inventory management strategies that can adapt quickly to emerging situations. By understanding demand patterns and being able to adjust swiftly to external factors like weather, the grocery store aims to keep shelves stocked, reduce waste, and ultimately ensure customer satisfaction—even when conditions are unpredictable.

The Dataset

- **Date Range** spans from January 1, 2019, to December 31, 2021, with daily frequency, providing a continuous timeline for analyzing demand.
- **Demand Data** represents actual daily sales, showing natural seasonal variations throughout the year as well as daily fluctuations due to the natural changes in consumer behavior.
- **Weather-Induced Spikes data** includes significant changes in demand on certain days.
- **Date Index**: The 'Date' column is used as the index of the dataset, enabling efficient time series analysis and visualization.

Descriptive Statistics

Before diving into modeling and forecasting, it’s essential to first examine the descriptive statistics of the demand data. This initial analysis provides a foundational understanding of the dataset, helping us identify important characteristics such as the average demand level, variability, and any patterns or anomalies. By looking at metrics like the mean, median, standard deviation, and range, we can gain insights into the overall stability and fluctuations in demand.

For example, understanding the typical demand range can help set realistic expectations for the model’s performance, especially if the data contains sudden spikes or seasonal patterns. Visualizing the data distribution, such as through a histogram, allows us to see if demand is skewed, clustered around certain values, or if there are extreme outliers that could impact forecasting accuracy. By examining these aspects, we can make informed decisions on model selection and fine-tuning, ensuring that our approach aligns well with the data's underlying

patterns. This preparatory step ultimately helps us create a more accurate and reliable demand forecasting model.

Implementing Descriptive Statistics in *Google Colab:* Complete Code and Results

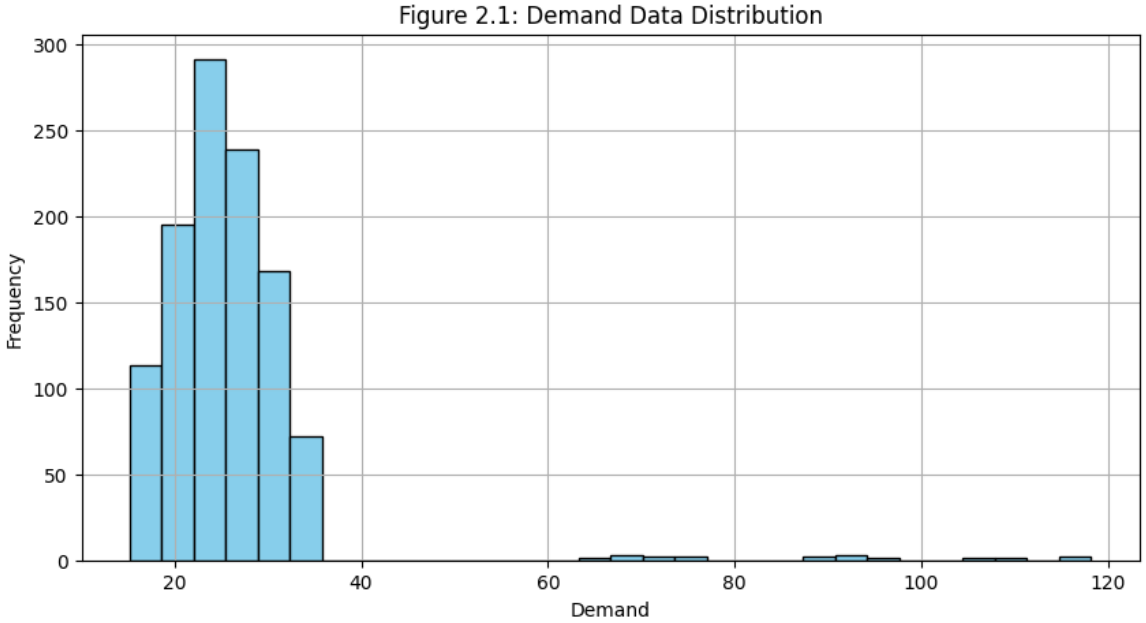
Below is a complete code snippet that loads the demand data, calculates descriptive statistics, and visualizes the demand distribution with a histogram. For a detailed explanation of the code, please refer to the technical appendix at the end of the chapter.

Technical note: in the following code, the # symbol is used to add ***comments****. These comments are intended to explain what the code is doing and provide additional context for the reader. Importantly, removing these comments will not affect the functionality of the code itself. The code will still be executed and perform all intended operations.*

```
# Step 1. Import the necessary libraries for data manipulation, numerical operations, visualization, and performance evaluation.
import pandas as pd  # For data manipulation and time series analysis.
import matplotlib.pyplot as plt  # For creating visualizations and charts.
import numpy as np  # For numerical operations and random number generation

# Step 2. Load the data file from GitHub into a pandas DataFrame.
url = "https://github.com/CharlesCLuo/Application-of-AI-in-Supply-Chain-Risk-Management-Series/blob/main/Demand_Forecsting/demand_data.csv?raw=true"
# Load data and set 'Date' as index.
data_single = pd.read_csv(url, parse_dates=['Date'], index_col='Date')

# Step 3: Descriptive Statistics
# Calculate descriptive statistics for the dataset.
print("Figure 2.1 Descriptive Statistics for the Demand Data:")
print(data_single['Demand'].describe())

# Step 4. Visualize the data distribution using a histogram.
plt.figure(figsize=(10, 5))
plt.hist(data_single['Demand'], bins=30, color='skyblue', edgecolor='black')
plt.title('Demand Data Distribution')
```

```
plt.xlabel('Demand')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

```
Descriptive Statistics for the Demand Data:
count    1096.000000
mean       25.968973
std         9.378127
min        15.204462
25%        21.578105
50%        25.007587
75%        28.652838
max       118.187573
```



The descriptive statistics and **Figure 2.1: Demand Data Distribution** provide valuable insights into the characteristics of the demand data, particularly how it behaves over the analyzed period:

- **Count**: The dataset contains 1,096 observations, representing daily demand values over approximately three years.
- **Mean (25.97)**: The average demand value is approximately 25.97. This mean value indicates that demand values tend to hover around this level.

- **Standard Deviation (9.38)**: The standard deviation of 9.38 suggests moderate variability in demand, indicating relatively stable demand with some notable fluctuations.
- **Minimum (15.20)**: The lowest demand value is 15.20, showing that even on the lowest-demand days, there is still a base level of demand.
- **25th Percentile (21.58)**: At the 25th percentile, the demand is 21.58, meaning that 25% of the observations fall below this level, representing lower demand days.
- **Median (25.01)**: The median demand value is 25.01, with half of the demand values falling below and half above this level. The closeness of the median to the mean suggests a fairly balanced distribution with slight right-skewness.
- **75th Percentile (28.65)**: At the 75th percentile, the demand is 28.65, showing that 75% of the data points are below this level, with the majority of demand values clustered between 15 and 40.
- **Maximum (118.19)**: The maximum demand value is 118.19, representing the highest observed demand, likely due to exceptional events or demand spikes.

Figure 2.1 displays the distribution of demand data over the entire dataset. The histogram reveals that the majority of demand values are concentrated between 15 and 40, with only a few instances where demand increases significantly. This indicates that, on most days, demand remains relatively stable, with occasional, sharp spikes in demand values, potentially due to specific events like weather changes or other influential factors. The clustering of demand values in the lower range, along with the presence of these rare spikes (shown on the far right of the histogram), suggests a right-skewed distribution. In summary, the descriptive statistics and Figure 2.1 reveal a demand pattern that is typically low and stable, punctuated by significant spikes. These spikes likely represent impactful events that cause short-term surges in demand.

Exponential Moving Average (EMA)

Exponential Moving Average (EMA) is a type of moving average that gives more weight to recent observations, making it more responsive to changes in data than a simple moving average. By applying a smoothing factor that emphasizes newer data points, EMA effectively tracks

short-term trends while filtering out random noise from daily fluctuations.

The basic formula for Single Exponential Smoothing is defined as:

$$\hat{y}_{t+1} = \alpha y_t + (1 - \alpha)\hat{y}_t$$

Where:

- $\hat{y}_{t+1}$ = forecasted value at time $t + 1$
- α = smoothing constant (0 < α < 1)
- y_t = actual value at time t
- $\hat{y}_t$ = forecasted value at time t

Intuition: EMA as a Manager's Radar

The EMA can be thought of as a store manager's radar for spotting demand trends. When there is a sudden increase in customers buying a specific product—perhaps due to an approaching weather event—it is essential for this radar to reflect the uptick immediately, enabling the store to adjust inventory appropriately. EMA ensures that decisions are based on the most current information, rather than relying too heavily on outdated sales patterns. It helps maintain focus on present demand while still acknowledging historical data, without letting outdated trends overly influence inventory decisions.

In summary, EMA is responsive to recent changes, as it places more weights on the latest data while smoothing out less relevant older data. Additionally, the EMA smooths out random fluctuations, effectively filtering out noise while still capturing the underlying trends in demand. In the next section, EMA will be demonstrated in action.

Implementing EMA in *Google Colab:* Complete Code and Results

Below is a complete code snippet. For a detailed explanation of the code, please refer to the technical appendix at the end of the chapter.

*Technical note: in the following code, the # symbol is used to add **comments**. These comments are intended to explain what the code is doing and provide additional context for the reader. Importantly, removing these comments will not affect the functionality of the code itself. The code will still be executed and perform all intended operations.*

```
# Step 1. Import the necessary libraries for data manipulation, numerical operations, visualization, and performance evaluation.
import pandas as pd  # For data manipulation and time series analysis.
from statsmodels.tsa.holtwinters import ExponentialSmoothing  # For Exponential Moving Average (EMA) modeling.
import matplotlib.pyplot as plt  # For creating visualizations and charts.
from sklearn.metrics import mean_absolute_error, mean_squared_error  # For calculating performance metrics.
import numpy as np  # For numerical operations and random number generation

# Step 2. Load the data file from GitHub into a pandas DataFrame.
url = "https://github.com/CharlesCLuo/Application-of-AI-in-Supply-Chain-Risk-Management-Series/blob/main/Demand_Forecsting/demand_data.csv?raw=true"
# Load data and set 'Date' as index.
data_single = pd.read_csv(url, parse_dates=['Date'], index_col='Date')

# Step 3. Split the dataset into training and testing sets.
# Allocate 80% of the data to training and 20% to testing.
train_size = int(len(data_single) * 0.8)  # Calculate the number of rows to include in the training set (80% of the total data).
train, test = data_single.iloc[:train_size], data_single.iloc[train_size:]  # Use slicing to separate training and testing data.

# Step 4: Fit the Exponential Moving Average (EMA) model on the training data.
# We use the ExponentialSmoothing function from the statsmodels library.
# Here, we only include a trend component (additive trend) and no seasonal component (set to None).
ema_model = ExponentialSmoothing(train['Demand'], trend='add', seasonal=None).fit()  # Fit the EMA model on the training data.

# Step 5: Forecast using the EMA model.
# Forecast values for the training period using fitted values and predict the next steps for the test period.
train['EMA_Forecast'] = ema_model.fittedvalues  # Use fitted values to get the model's predictions on the training data.
```

```
test['EMA_Forecast'] = ema_model.forecast(steps=len(test))  # Forecast future values for the test set.

# Step 6: Visualize the EMA results.
plt.figure(figsize=(14, 7))  # Set the figure size to 14 inches by 7 inches for better readability.

# Plot the actual demand values for the training period.
plt.plot(train.index, train['Demand'], label='Training Data', color='blue', linestyle='-')  # Training data in blue.
# Plot the actual demand values for the testing period.
plt.plot(test.index, test['Demand'], label='Test Data', color='orange', linestyle='--')  # Test data in orange.
# Plot the EMA forecast values for the testing period.
plt.plot(test.index, test['EMA_Forecast'], label='EMA Forecast', color='red', linestyle='-')  # EMA forecast in red.

# Highlight the point where the training data ends and the test data begins.
plt.axvline(x=train.index[-1], color='black', linestyle=':', linewidth=1.5, label='Train-Test Split')  # Vertical line indicating the split.

# Add labels, title, and legend to make the plot easier to understand.
plt.title('Exponential Moving Average (EMA) Model - Actual vs. Forecasted Demand')  # Title of the plot.
plt.xlabel('Date')  # X-axis label indicating the dates.
plt.ylabel('Demand')  # Y-axis label indicating demand values.
plt.legend()  # Display the legend to identify each line in the plot.
plt.grid(True)  # Add a grid for easier interpretation of the values.

# Display the plot.
plt.show()  # Show the visualization.

# Step 7: Calculate Performance Metrics for the EMA Model
# Calculate Mean Absolute Error (MAE) to measure the average magnitude of the errors between actual and forecasted values.
mae_EMA = mean_absolute_error(test['Demand'], test['EMA_Forecast'])
print(f"Mean Absolute Error (MAE): {mae_EMA:.4f}")

# Calculate Root Mean Squared Error (RMSE) to measure the square root of the average of squared differences between actual and forecasted values.
# This metric penalizes larger errors more than MAE.
```

```
rmse_EMA = mean_squared_error(test['Demand'], test['EMA_Forecast'], squared=False)  # Set squared=False to get RMSE.
print(f"Root Mean Squared Error (RMSE): {rmse_EMA:.4f}")

# Calculate Mean Absolute Percentage Error (MAPE) to express the error as a percentage of the actual values.
# Adding a small epsilon value (1e-10) helps avoid division by zero if any demand values are zero.
epsilon = 1e-10  # A small value to handle division by zero, if any demand values are zero.
mape_EMA = np.mean(np.abs((test['Demand'] - test['EMA_Forecast']) / (test['Demand'] + epsilon))) * 100
print(f"Mean Absolute Percentage Error (MAPE): {mape_EMA:.4f}%")
```

```
Mean Absolute Error (MAE): 4.7470
Root Mean Squared Error (RMSE): 10.7998
Mean Absolute Percentage Error (MAPE): 17.4612%
```

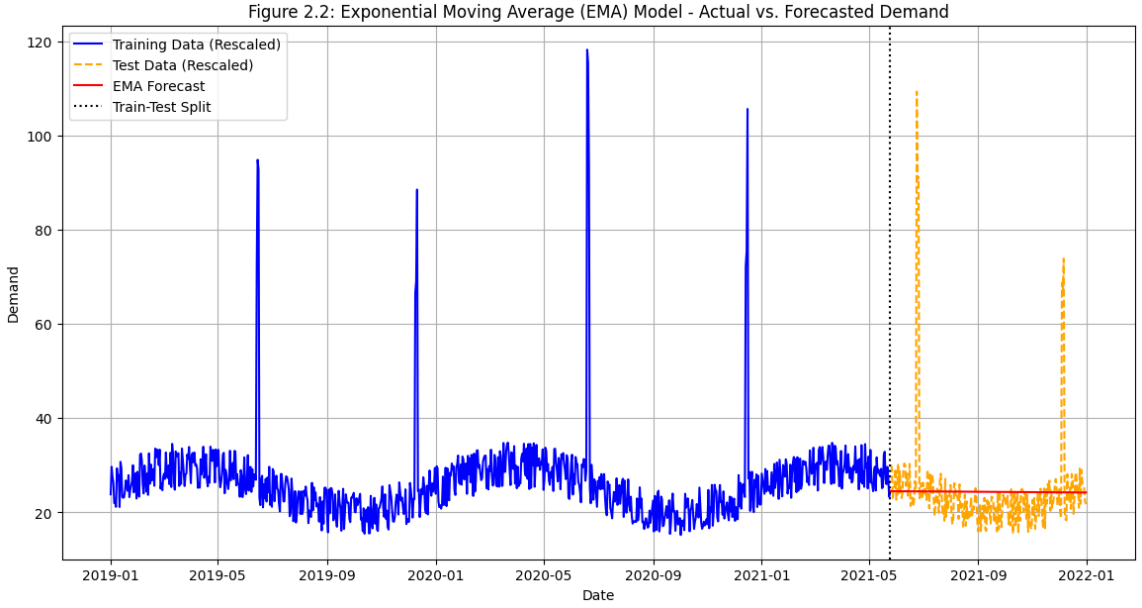


Figure 2.2: Exponential Moving Average (EMA) Model - Actual vs. Forecasted Demand provides a visual comparison between the actual demand data and the forecasts generated by the EMA model over a three-year period. The dataset was split into two segments: training and testing. The blue line represents the training data, covering demand from January 2019 to early 2021. The orange line represents the test data, covering the rest of 2021. The vertical dotted line indicates the point of division between training and testing.

EMA Forecast and Actual Values: The red line represents the forecasted demand values produced by the EMA model. After training, the model predicted demand for the test period, aiming to capture both regular fluctuations and sudden spikes. The EMA model was able to track the overall trend in demand reasonably well, especially during the stable portions of the data. It effectively smoothed out minor fluctuations, providing a general view of ongoing demand. However, the EMA model also smoothed out extreme values (demand spikes), leading to under-forecasting during periods of sharp increases. This limitation in accurately predicting significant spikes is evident from the disparity between the forecast and the observed values during peak events. The performance metrics obtained for the Exponential Moving Average (EMA) model are as follows:

- *Mean Absolute Error (MAE): 4.7470.* MAE represents the average absolute difference between the predicted and actual values. An MAE of 4.7470 indicates that, on average, the EMA model's predictions are approximately 4.75 units away from the actual demand values. This level of error suggests that the model is generally able to track the underlying trend in demand but does encounter some deviation, especially during fluctuations.
- *Root Mean Squared Error (RMSE): 10.7998.* RMSE measures the square root of the average of the squared differences between predicted and actual values. It places greater emphasis on larger errors compared to MAE. An RMSE of 10.7998 suggests a moderate level of error, with a higher penalty on larger deviations. This indicates that while the EMA model is generally aligned with the actual demand trend, it struggles more during periods of high variability or sudden demand spikes, which impact the RMSE value due to the squaring of larger errors.
- *Mean Absolute Percentage Error (MAPE): 17.4612%.* MAPE represents the average absolute percentage difference between predicted and actual values, expressed as a percentage. A MAPE of 17.4612% implies that, on average, the EMA model's predictions are about 17.46% off from the actual demand values. While this percentage is acceptable for stable demand periods, it indicates higher error rates during sudden demand changes, leading to relatively larger deviations.

Why is MAPE Higher? The EMA model, as a smoothing technique, performs well with stable data but struggles with capturing **sudden spikes** or sharp fluctuations in demand. If the actual data contains abrupt increases or decreases (e.g., due to events or external factors), the EMA model will likely underestimate these, resulting in relatively large errors.

Insights: The MAE and RMSE values indicate that the EMA model has a reasonable level of accuracy when predicting stable demand. However, the MAPE value highlights its limitations in capturing sudden changes or demand spikes. The EMA model's smoothing effect might cause it to adapt slowly to abrupt shifts in demand, leading to higher relative errors in those cases.

Implications for Supply Chain Managers: A forecasting model that cannot handle unpredictable spikes may lead to inventory shortages during unexpected high-demand periods. To mitigate this risk, it is essential to consider more advanced models that can better adapt to sudden changes, improving responsiveness and accuracy in volatile demand environments.

Autoregressive Integrated Moving Average (ARIMA)

For demand forecasting, another popular traditional model worth considering is ARIMA, which is particularly effective at capturing trends and patterns when there is a clear linear relationship between past and present values. ARIMA combines three components—Autoregression (AR), Differencing (I), and Moving Average (MA), enabling it to model both trend and seasonality in the data. This flexibility allows ARIMA to adapt to complex patterns in demand more effectively than simpler models like EMA, potentially reducing forecasting errors and improving overall accuracy.

- **Autoregression (AR)**: This component involves regressing the time series on its own lagged values. In simpler terms, it predicts future values based on a combination of past values.
- **Differencing (I)**: Differencing is a way of removing trends or seasonality from the data to make it stationary. Essentially, it subtracts the current observation from the previous one, which helps eliminate patterns that could mislead forecasts.
- **Moving Average (MA)**: The moving average component looks at the errors made by previous forecasts and adjusts future forecasts accordingly.

The technical details of the ARIMA model are provided below; however, understanding the mathematics is not essential for effective application in our context.

1. **Autoregressive (AR) Component**:

The AR part involves regressing the variable against its own lagged values. The mathematical equation for an AR model of order p is:

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \epsilon_t$$

Where:

- y_t is the value at time t.
- ϕ_i are the coefficients to be estimated.
- ϵ_t is the error term (white noise).

2. **Differencing (I) Component**:

Differencing involves subtracting the previous value from the current value to remove trends and make the data stationary. For first-order differencing:

$$\Delta y_t = y_t - y_{t-1}$$

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

The MA part models the error term as a linear combination of previous error terms. For an MA model of order q:

$$y_t = \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q}$$

Where:

- θ_i are the coefficients to be estimated.

The full ARIMA model combines these components as:

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \Delta y_t - \theta_1 \epsilon_{t-1} - \ldots - \theta_q \epsilon_{t-q} + \epsilon_t$$

Intuition: Imagine you're running a retail store, and you want to predict future sales. Sales usually increase during certain periods of the year. But how do you capture these patterns in a way that can give you reliable sales forecasts? When applying ARIMA forecasting method, we look at three main components:

- **Past Sales Data (Autoregressive Component)**: ARIMA uses past sales numbers to predict future values. It assumes that recent sales have been influenced by the sales in the previous few periods. For example, if sales have been steadily increasing over the last few months, ARIMA will factor that into its predictions. Imagine you sold 100 units last month, 95 units the month before, and 90 units before that—ARIMA would recognize this upward trend and use it to make future forecasts.

- **Trend Adjustments (Differencing Component)**: Sales data may not always be stable—there could be a consistent upward or downward trend. To handle this, ARIMA smooths out these changes by calculating the differences between consecutive periods to make the data more stable. This allows ARIMA to focus on the core movement in your sales without being thrown off by persistent trends. For example, if your sales are consistently increasing by 5 units per month, differencing helps ARIMA capture this growth and adjust its model accordingly.
- **Random Fluctuations (Moving Average Component)**: There are always random spikes or drops in sales that cannot be predicted by trend alone. These fluctuations may be due to promotions, supply chain disruptions, or even unexpected weather changes. ARIMA tries to reduce the impact of these random variations by incorporating a moving average of past forecast errors. This allows ARIMA to make more accurate predictions by accounting for past unpredictability. For instance, if there was an unexpected surge in sales in the last season, ARIMA would learn from the past error to make a more accurate adjustment in future forecasts.

Implementing ARIMA in *Google Colab: Complete Code and Results*

Below is a complete code snippet. For a detailed explanation of the code, please refer to the technical appendix at the end of the chapter.

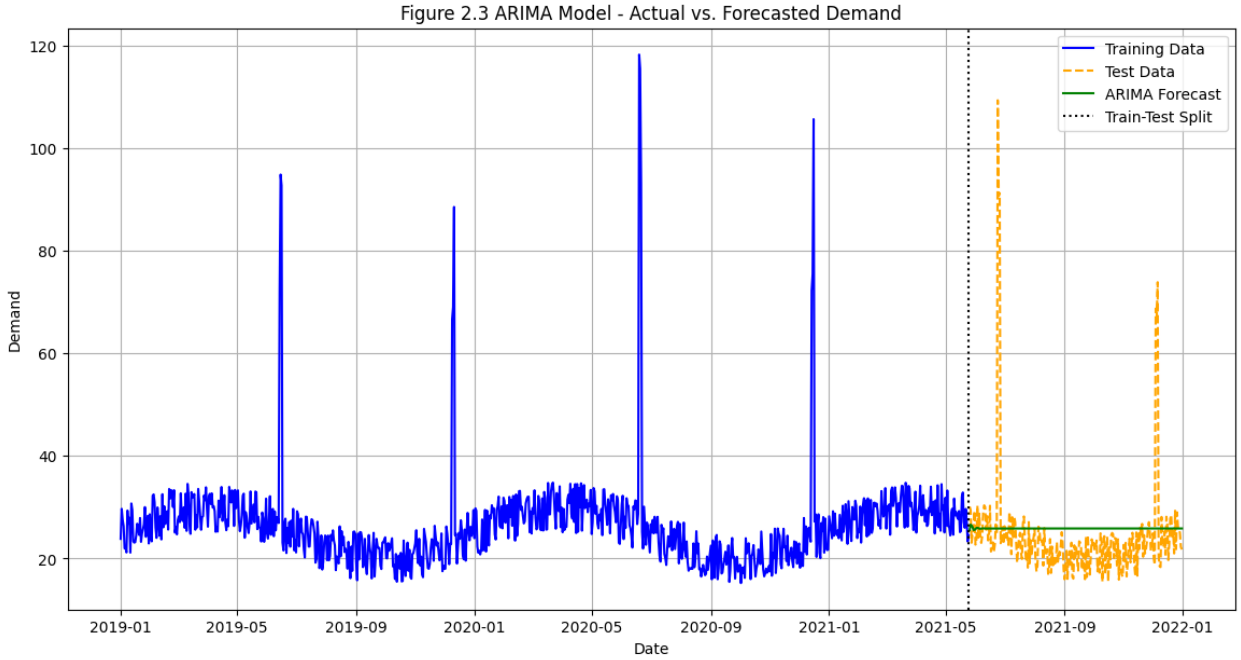
*Technical note: in the following code, the # symbol is used to add **comments**. These comments are intended to explain what the code is doing and provide additional context for the reader. Importantly, removing these comments will not affect the functionality of the code itself. The code will still be executed and perform all intended operations.*

```
# Step 1. Import necessary libraries for data loading, ARIMA modeling, and visualization
import pandas as pd  # For data manipulation and time series analysis.
from statsmodels.tsa.arima.model import ARIMA  # ARIMA is used for modeling time series data with auto-regression and moving averages.
from sklearn.metrics import mean_absolute_error, mean_squared_error  # For calculating evaluation metrics.
import matplotlib.pyplot as plt  # For visualizing the results.
import numpy as np  # For mathematical operations.

# Step 2.  Load the data file from GitHub into a pandas DataFrame.
url = "https://github.com/CharlesCLuo/Application-of-AI-in-Supply-Chain-Risk-Management-Series/blob/main/Demand_Forecsting/demand_data.csv?raw=true"
# Load data and set 'Date' as index.
data_single = pd.read_csv(url, parse_dates=['Date'], index_col='Date')

# Step 3. Split the dataset into training and testing sets.
train_size = int(len(data_single) * 0.8)  # Calculate the number of rows for the training set (80% of the total data).
train, test = data_single.iloc[:train_size], data_single.iloc[train_size:]  # Use slicing to split the data into training and testing sets.

# Step 4. Fit the ARIMA model on the training demand data.
# Parameters: p = 5 (lag observations), d = 1 (difference the data once), q = 0 (no moving average component)
arima_model = ARIMA(train['Demand'], order=(5, 1, 0)).fit()  # Fit the ARIMA model on the training data.

# Step 5. Make predictions using the ARIMA model.
train['ARIMA_Forecast'] = arima_model.fittedvalues  # Get the model's predictions on the training data (fitted values).
test['ARIMA_Forecast'] = arima_model.forecast(steps=len(test))  # Forecast future values for the test set using the model.

# Step 6. Calculate evaluation metrics for the ARIMA model.
```

```
mape_ARIMA = np.mean(np.abs((test['Demand'] - test['ARIMA_Forecast']) / test['Demand'])) *
100  # Calculate MAPE.
mae_ARIMA = mean_absolute_error(test['Demand'], test['ARIMA_Forecast'])  # Calculate MAE.
rmse_ARIMA = np.sqrt(mean_squared_error(test['Demand'], test['ARIMA_Forecast']))  # Calculate
RMSE.

# Print the evaluation metrics.
print(f'MAPE: {mape_ARIMA:.2f}%')
print(f'MAE: {mae_ARIMA:.2f}')
print(f'RMSE: {rmse_ARIMA:.2f}')

# Step 7. Visualize the ARIMA model results.
plt.figure(figsize=(14, 7))  # Set the figure size to 14 inches by 7 inches for better readability.

# Plot the actual demand values for the training period.
plt.plot(train.index, train['Demand'], label='Training Data', color='blue', linestyle='-')  # Training
data in blue.
# Plot the actual demand values for the testing period.
plt.plot(test.index, test['Demand'], label='Test Data', color='orange', linestyle='--')  # Test data in
orange.
# Plot the ARIMA forecast values for the testing period.
plt.plot(test.index, test['ARIMA_Forecast'], label='ARIMA Forecast', color='green', linestyle='-')  #
ARIMA forecast in green.

# Highlight the point where the training data ends and the test data begins.
plt.axvline(x=train.index[-1], color='black', linestyle=':', linewidth=1.5, label='Train-Test Split')  #
Vertical line indicating the train-test split.

# Add labels, title, and legend to make the plot more understandable.
plt.title('Figure 2.3 ARIMA Model - Actual vs. Forecasted Demand')  # Set the title of the plot.
plt.xlabel('Date')  # Label for the x-axis indicating dates.
plt.ylabel('Demand')  # Label for the y-axis indicating demand values.
plt.legend()  # Display the legend to identify each line in the plot.
plt.grid(True)  # Add a grid for better readability of the data points.

# Show the plot to visualize the results.
plt.show()  # Display the visualization.
```

MAPE: 21.46%
MAE: 5.48
RMSE: 10.93

Figure 2.3 ARIMA Model - Actual vs. Forecasted Demand



Visualization: In Figure 2.3, the blue line represents the actual demand during the training period, while the orange line represents the actual demand during the testing period. The green line shows the demand forecasted by the ARIMA model. A vertical line is added to indicate where the training data ends and the test data begins, making it clear where the forecast starts. You can see that the forecasted demand remains quite flat, failing to capture the sharp spikes that are evident in the actual demand. This flat forecast line highlights the challenges ARIMA faces when dealing with sudden changes in demand.

Performance Metrics:

- The **MAPE** of 21.46% indicates notable differences between predicted and actual values, especially during sharp spikes. While lower MAPE values suggest better accuracy, here the model's predictions diverge significantly from the true demand during volatile periods, making it unreliable in such scenarios.
- The **MAE** of 5.48 suggests that, on average, the forecasted values are 5.48 units away from the actual demand. However, this metric can be somewhat misleading, as ARIMA captures the general trend but struggles with large spikes. MAE doesn’t fully reflect the model's challenges in predicting extreme values.

- The **RMSE** of 10.93 penalizes larger errors more heavily than MAE, providing insight into the model's performance during periods of significant error. Although the RMSE isn't excessively high, it emphasizes the model's limitations during spikes.

Overall, the metrics and the flat forecast line in figure 2.3 make it clear that ARIMA, with the current parameter settings, struggles with extreme variability in demand. These spikes are essential for effective supply chain management because they dictate when extra inventory is needed or when operational adjustments are necessary: If the model underestimates demand, companies might not have enough inventory on hand during critical times, leading to lost sales opportunities and dissatisfied customers; If sudden drops in demand aren't captured, companies may overproduce or overstock, which increases holding costs and the risk of obsolescence.

Fine-Tuning Parameters for improvement: Experimenting with different parameter values can help the model better capture sudden demand spikes and adapt to diverse demand patterns. We encourage readers to try various settings to see how the model responds under different scenarios. For instance, keeping d=0 may retain essential fluctuations, while testing different combinations of p and q can help the model learn from past values and adjust for recent forecast errors, enhancing its responsiveness to abrupt changes:

- Since the demand is generally stable, a lower autoregressive order (e.g., p=1 or p=2) is typically sufficient. This will allow the model to consider recent observations without overly complicating the model.
- The moving average term q helps account for past forecast errors. For data with sudden spikes, a moderate q value (e.g., q=2 or q=3) can help capture the effects of recent errors and adjust forecasts accordingly.

While fine-tuning can improve performance to some degree, ARIMA's linear nature ultimately limits its effectiveness in dynamic environments with abrupt changes in demand. When facing volatile demand in a complex and dynamic environments, alternative or complementary approaches may be necessary for more reliable forecasting.

Discussion

In evaluating EMA and ARIMA models, it becomes clear that both capture general demand trends effectively, especially within stable environments. EMA, by weighting recent data more heavily, smooths out short-term fluctuations while staying responsive to recent shifts, making it well-suited for overall trend tracking. Similarly, ARIMA, through its AR, I, and MA components, models linear trends and seasonality, making it effective for consistent demand patterns. This performance is reflected in relatively low MAE and RMSE values, underscoring the models' reliability in stable environments.

However, traditional models like EMA and ARIMA struggle to accurately capture demand spikes, as indicated by MAPE. Unlike MAE and RMSE, MAPE focuses on proportional error, which reveals these models' limitations in handling sharp, unpredictable changes. EMA's smoothing effect averages out anomalies, leading to an under-representation of spikes. ARIMA, with its dependence on historical linear patterns, similarly lacks responsiveness to sudden shifts, resulting in high MAPE values during spike events.

Overall, while EMA and ARIMA are reliable for capturing general trends, more adaptable methods may be necessary in highly volatile environments. Machine learning-based techniques, which can better account for abrupt changes and nonlinear patterns, might provide more accurate predictions in scenarios with frequent demand surges and unexpected fluctuations.

Summary

This chapter lays a foundation, equipping us with essential tools and understanding for implementing and evaluating demand forecasting methods in supply chain contexts. We introduced traditional demand forecasting concepts, focusing on EMA and ARIMA—widely used models suited to different forecasting needs, with EMA for short-term trends and ARIMA for complex patterns. Beginning with descriptive statistics and visualizations, we gained insights into demand patterns, which set the stage for implementing EMA and ARIMA using Python and libraries like *Pandas, Matplotlib,* and *Statsmodels*. Model performance was evaluated with key

metrics—MAE, MAPE, and RMSE—revealing that while both EMA and ARIMA effectively capture general trends in stable settings, their limitations with sudden demand spikes became apparent through MAPE analysis. Such spikes are critical in supply chain management, where missing unexpected changes can lead to inventory shortages or lost sales.

Key Takeaways

1. **Importance of Descriptive Statistics**: Descriptive analysis of demand data reveals key patterns and characteristics essential for informed forecasting.
2. **Exponential Moving Average (EMA)**: EMA effectively captures short-term trends by emphasizing recent data, helping to smooth fluctuations while remaining responsive to changes.
3. **ARIMA Model Versatility**: With its autoregressive, differencing, and moving average components, ARIMA is suited for data with trends and seasonality but requires careful tuning of parameters for optimal accuracy.
4. **Performance Metrics**: MAE, MAPE, and RMSE each offer unique insights into model accuracy, helping to assess both average errors and areas where models may struggle.
5. **Python Libraries in Forecasting**: Practical use of libraries like Pandas, Matplotlib, and Statsmodels streamlines data manipulation, visualization, and model implementation, bridging theory and application.
6. **Hands-on Learning**: Coding exercises provide valuable experience in implementing and evaluating models, reinforcing concepts through direct application in Google Colab.

Exercise

In this exercise, we analyze demand data that shows a steady increase over a three-year period, representing business growth and market expansion. Use the following code to apply RNN and LSTM models, and assess how well each model effectively captures the underlying demand patterns in comparison to EMA and ARIMA.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Load the dataset from GitHub
url = 'https://raw.githubusercontent.com/CharlesCLuo/Application-of-AI-in-Supply-Chain-Risk-Management-Series/main/Demand_Forecsting/demand_data_exercise.csv'
data = pd.read_csv(url, parse_dates=['Date'], index_col='Date')

# Display the first few rows of the dataset
print("First few rows of the dataset:")
print(data.head())

# Generate descriptive statistics for the 'Demand_with_Trend' variable
print("\nDescriptive Statistics for 'Demand_with_Trend':")
print(data['Demand_with_Trend'].describe())

# Plot the demand data
plt.figure(figsize=(12, 6))
plt.plot(data.index, data['Demand_with_Trend'], label='Demand_with_Trend')
plt.title('Demand with Trend Over Time')
plt.xlabel('Date')
plt.ylabel('Demand')
plt.legend()
plt.grid(True)
plt.show()

# Plot histogram of 'Demand_with_Trend'
plt.figure(figsize=(10, 5))
plt.hist(data['Demand_with_Trend'], bins=30, color='skyblue', edgecolor='black')
plt.title('Distribution of Demand with Trend')
plt.xlabel('Demand')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()

# Split the data into training and testing sets
train_size = int(len(data) * 0.8)
train, test = data.iloc[:train_size], data.iloc[train_size:]

# Apply Exponential Moving Average (EMA) to the training set only
train['EMA'] = train['Demand_with_Trend'].ewm(span=10, adjust=False).mean()

# Extend EMA forecast to the test set by using the last value of EMA from the training set
last_train_ema = train['EMA'].iloc[-1]
```

```
test['EMA'] = last_train_ema  # Use last training EMA as the constant EMA forecast for testing

# Plot EMA on the training and testing data
plt.figure(figsize=(12, 6))
plt.plot(train.index, train['Demand_with_Trend'], label='Training Data - Demand_with_Trend')
plt.plot(train.index, train['EMA'], label='Training EMA', color='red')
plt.plot(test.index, test['Demand_with_Trend'], label='Testing Data - Demand_with_Trend')
plt.plot(test.index, test['EMA'], label='Testing EMA (Constant)', color='red', linestyle='--')
plt.title('Exponential Moving Average (EMA) Applied Only on Training Data')
plt.xlabel('Date')
plt.ylabel('Demand')
plt.legend()
plt.grid(True)
plt.show()

# Fit ARIMA model on 'Demand_with_Trend'
arima_model = ARIMA(train['Demand_with_Trend'], order=(5, 1, 0)).fit()

# Forecast using ARIMA model
train['ARIMA_Forecast'] = arima_model.fittedvalues
test['ARIMA_Forecast'] = arima_model.forecast(steps=len(test))

# Plot ARIMA forecast
plt.figure(figsize=(12, 6))
plt.plot(train.index, train['Demand_with_Trend'], label='Training Data')
plt.plot(test.index, test['Demand_with_Trend'], label='Test Data')
plt.plot(test.index, test['ARIMA_Forecast'], label='ARIMA Forecast', color='green')
plt.title('ARIMA Model - Actual vs. Forecasted Demand_with_Trend')
plt.xlabel('Date')
plt.ylabel('Demand')
plt.legend()
plt.grid(True)
plt.show()

# Calculate evaluation metrics for EMA and ARIMA models on the test set
mae_ema = mean_absolute_error(test['Demand_with_Trend'], test['EMA'])
mape_ema = np.mean(np.abs((test['Demand_with_Trend'] - test['EMA']) /
test['Demand_with_Trend'])) * 100
rmse_ema = np.sqrt(mean_squared_error(test['Demand_with_Trend'], test['EMA']))

mae_arima = mean_absolute_error(test['Demand_with_Trend'], test['ARIMA_Forecast'])
mape_arima = np.mean(np.abs((test['Demand_with_Trend'] - test['ARIMA_Forecast']) /
test['Demand_with_Trend'])) * 100
rmse_arima = np.sqrt(mean_squared_error(test['Demand_with_Trend'], test['ARIMA_Forecast']))

# Print evaluation metrics for both EMA and ARIMA
print("Evaluation Metrics for EMA (Testing Set):")
print(f"Mean Absolute Error (MAE): {mae_ema:.2f}")
```

```
print(f"Mean Absolute Percentage Error (MAPE): {mape_ema:.2f}%")
print(f"Root Mean Squared Error (RMSE): {rmse_ema:.2f}")

print("\nEvaluation Metrics for ARIMA (Testing Set):")
print(f"Mean Absolute Error (MAE): {mae_arima:.2f}")
print(f"Mean Absolute Percentage Error (MAPE): {mape_arima:.2f}%")
print(f"Root Mean Squared Error (RMSE): {rmse_arima:.2f}")
```

Understanding the Data:

Q1: Review and interpret the descriptive statistics and visualizations for the dataset. What insights can you gain about the overall demand pattern and any unusual characteristics?

Evaluating Model Performance:

Q2: Evaluate the performance of the EMA model using the metrics MAE, MAPE, and RMSE. How well does EMA capture the characteristics of the data, and where does it show strengths or weaknesses?

Q3: Evaluate the performance of the ARIMA model using MAE, MAPE, and RMSE. How effectively does ARIMA capture the data's characteristics, and how does its performance compare to EMA?

Practical Considerations for Business Decisions:

Q4: As a supply chain manager, consider how the insights from each model could inform inventory planning during periods with demand spikes.

Glossary

Exponential Moving Average (EMA): A type of moving average that gives more weight to recent observations, making it responsive to recent trends in the data. EMA is commonly used in demand forecasting to capture short-term patterns while smoothing out noise.

AutoRegressive Integrated Moving Average (ARIMA): A time series forecasting model that combines three components: autoregressive (AR), differencing (I), and moving average (MA). ARIMA is particularly effective for data with a clear trend or seasonality.

Descriptive Statistics: Basic statistical calculations, including mean, median, standard deviation, and percentiles, that provide an overview of data characteristics. Descriptive statistics help analysts understand demand patterns before applying forecasting models.

Mean Absolute Error (MAE): A performance metric that calculates the average absolute difference between forecasted and actual values. MAE provides an indication of the model's accuracy in units of the data.

Mean Absolute Percentage Error (MAPE): A performance metric that expresses the error as a percentage of the actual values, making it useful for understanding forecast accuracy in relative terms. MAPE is particularly useful for comparing model performance across datasets of varying scales.

Root Mean Squared Error (RMSE): A performance metric that calculates the square root of the average squared differences between forecasted and actual values. RMSE places more weight on larger errors, making it valuable for assessing the impact of outliers on forecasting accuracy.

Time Series: A sequence of data points collected over time at regular intervals. Time series data is often used in demand forecasting to analyze trends, patterns, and seasonality.

Spikes: Sudden increases in demand that occur at predictable intervals, often influenced by unexpected events. Spikes can make forecasting more challenging as they may not follow the general trend.

Trend: A long-term increase or decrease in data values over time. In demand forecasting, trends indicate an overall shift in demand and can be positive or negative.

Autoregressive (AR) Component: The AR part of the ARIMA model that uses previous observations to predict future values. It captures the correlation between a data point and a specified number of preceding data points.

Differencing (I) Component: The differencing part of the ARIMA model helps make a time series stationary by removing trends. Differencing is essential for ARIMA models to work effectively on data with a clear trend.

Moving Average (MA) Component: The MA part of the ARIMA model that accounts for patterns in the forecast errors, helping to reduce noise by smoothing out random fluctuations.

Python Libraries: Essential libraries used in this chapter include `Pandas` for data manipulation, `Matplotlib` for data visualization, and `Statsmodels` for implementing EMA and ARIMA models. Each library provides specialized functions for data analysis.

Technical Appendix 2.A. Implementing Descriptive Statistics: A Step by Step Explanation

Here, we provide a step-by-step guide on the coding implementation, walking through each part to ensure a clear and practical understanding of how to create descriptive statistics and visualize them effectively.

```
# Step 1. Import the necessary libraries for data manipulation, numerical operations, visualization, and performance evaluation.
import pandas as pd  # For data manipulation and time series analysis.
import matplotlib.pyplot as plt  # For creating visualizations and charts.
import numpy as np  # For numerical operations and random number generation
```

To start our demand forecasting journey, we first bring in a set of powerful Python libraries that simplify data handling, visualization, and mathematical operations. These libraries are like different sets of tools we need to complete a task. Imagine we're about to cook a meal—we'll need different utensils, like a knife, a pan, and a spatula. Similarly, we need different tools to manage data, visualize results, and build a model:

`numpy` and `pandas`: These are our "data knives." They help chop, mix, and prepare data so it's ready to be used:

- **Pandas** is our go-to library for managing data. It helps us load, arrange, and manipulate time-based data, which is essential for understanding demand trends. With Pandas, we can index data by date, perform calculations on specific columns, and seamlessly organize our demand dataset in a way that sets us up for effective analysis.
- **NumPy** is an essential library for numerical and scientific computing in Python. It provides fast and efficient tools for handling arrays and performing mathematical operations, making it fundamental in data science, machine learning, and forecasting tasks. NumPy's capabilities range from basic calculations, such as mean and standard deviation, to more complex operations, such as element-wise computations and random number generation.

Matplotlib is our "presentation plate." It allows us to visualize our data, like presenting our meal nicely on a plate. Its flexible tools allow users to create a wide variety of charts and plots—from simple line graphs to more complex visualizations like histograms and scatter plots.

In demand forecasting, NumPy is particularly useful when calculating performance metrics like RMSE and MAPE. These metrics often require multiple mathematical steps, including squaring values, taking square roots, or calculating percentages—all of which NumPy handles with ease. Its speed and efficiency make it possible to process large datasets and perform complex calculations quickly, ensuring that we can evaluate our model's accuracy effectively.

```
# Step 2. Load the data file from GitHub into a pandas DataFrame.
url = "https://github.com/CharlesCLuo/Application-of-AI-in-Supply-Chain-Risk-Management-Series/blob/main/Demand_Forecsting/demand_data.csv?raw=true"
# Load data and set 'Date' as index.
data_single = pd.read_csv(url, parse_dates=['Date'], index_col='Date')
```

Here, we're accessing our demand dataset stored on GitHub and loading it into a format that's easy to work with in Python. By specifying the GitHub URL, we import the data straight into a **Pandas DataFrame**—a table-like structure ideal for handling and analyzing data:

- The **`parse_dates`** argument converts the 'Date' column into a date format, which makes it easier to handle time-based data.
- Setting **`index_col='Date'`** allows us to use each date as a unique index, organizing our data by time. This setup is particularly helpful for time series analysis, where understanding trends over time is essential.

With our data in a structured format, we're now ready to explore demand patterns, examine trends, and prepare the data for forecasting. This step ensures that we have the data organized by date, giving us a solid foundation for analyzing how demand has changed over time.

```
# Step 3: Descriptive Statistics
```

```
# Calculate descriptive statistics for the dataset.
print("Figure 2.1 Descriptive Statistics for the Demand Data:")
print(data_single['Demand'].describe())
```

In this step, we calculate descriptive statistics to get a quick overview of our demand data. This initial analysis helps us understand basic characteristics of the dataset, such as average demand, variability, and the range of values.

By calling **`data_single['Demand'].describe()`**, we generate several helpful metrics:

- **Count**: The total number of demand entries in our dataset, giving us an idea of the dataset's size.
- **Mean**: The average demand, offering insight into typical demand levels over the analyzed period.
- **Standard Deviation**: A measure of how much demand fluctuates from the mean, indicating how stable or variable demand tends to be.
- **Min and Max**: The minimum and maximum demand values show us the range, highlighting any extreme low or high points in demand.
- **Percentiles (25%, 50%, 75%)**: These values provide a sense of how demand is distributed across different levels. For example, the 50% percentile (or median) tells us the middle point of demand, where half the values are above and half are below.

Viewing these statistics helps us quickly grasp the data's overall structure and variability. With this information, we can assess whether demand is mostly stable, highly variable, or if there are any unusual peaks. This foundational understanding guides our approach to modeling and helps us determine if adjustments are needed to capture demand trends accurately.

```
# Step 4. Visualize the data distribution using a histogram.
plt.figure(figsize=(10, 5))
plt.hist(data_single['Demand'], bins=30, color='skyblue', edgecolor='black')
plt.title('Demand Data Distribution')
```

```
plt.xlabel('Demand')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

Creating a histogram provides a visual summary of the demand data, showing how demand is distributed across different levels:

We first create a new figure with `plt.figure()`, specifying the size with **`figsize=(10, 5)`**. This sets the width of the plot to 10 units and the height to 5 units. Adjusting the plot's size can make it easier to read and interpret, especially for larger datasets.

Next, we generate the histogram itself using **`plt.hist()`**:

- **`data_single['Demand']`**: Specifies the data we're analyzing—the 'Demand' column from our `data_single` DataFrame.
- **`bins=30`**: Divides the range of demand values into 30 intervals, or bins. Each bin represents a range of demand values, and the histogram counts how many data points fall within each bin. Choosing 30 bins gives us a good level of detail, revealing the data's spread while avoiding too much granularity.
- **`color='skyblue'`**: Fills each bar in the histogram with sky blue, making it visually appealing and easy to interpret.
- **`edgecolor='black'`**: Adds a black border around each bar, which helps distinguish individual bins and improves readability.

Further, we label the plot to make it easy to interpret:

- **`plt.title('Demand Data Distribution')`**: Adds a title that describes what the histogram represents—the distribution of demand values.
- **`plt.xlabel('Demand')`** and **`plt.ylabel('Frequency')`**: Label the x-axis as "Demand" and the y-axis as "Frequency." This clearly indicates that we're plotting the frequency of demand values, making the visualization intuitive.

- Adding a grid with **`plt.grid(True)`** creates horizontal lines across the plot. These grid lines align with y-axis values, making it easier to judge the height of each bar and quickly see which demand ranges are most common.
- Finally, **`plt.show()`** displays the histogram. Without this command, the plot would not appear. `plt.show()` finalizes the figure and opens the histogram so we can visually interpret it.

Technical Appendix 2.B. Implementing EMA: A Step by Step Explanation

```
# Step 1. Import the necessary libraries for data manipulation, numerical operations, visualization, and performance evaluation.
import pandas as pd  # For data manipulation and time series analysis.
from statsmodels.tsa.holtwinters import ExponentialSmoothing  # For Exponential Moving Average (EMA) modeling.
import matplotlib.pyplot as plt  # For creating visualizations and charts.
from sklearn.metrics import mean_absolute_error, mean_squared_error  # For calculating performance metrics.
import numpy as np  # For numerical operations and random number generation
```

The code begins by importing essential libraries for data processing, modeling, visualization, and evaluation. Besides the libraries explained in Technical Appendix A, a few libraries are specific to EMA.

- **`statsmodels`** is a powerful Python library tailored for statistical modeling, hypothesis testing, and data exploration.
- **`ExponentialSmoothing`** from `statsmodels`: This class lets us create an Exponential Moving Average (EMA) model.
- `Scikit-Learn Metrics` **(`from sklearn.metrics import mean_absolute_error, mean_squared_error`)** is a library provides essential performance metrics (MAE, MSE, RMSE) for evaluating the accuracy of our forecasts, giving quantitative insights into prediction errors.

```
# Step 2. Load the data file from GitHub into a pandas DataFrame.
url = "https://github.com/CharlesCLuo/Application-of-AI-in-Supply-Chain-Risk-Management-Series/blob/main/Demand_Forecsting/demand_data.csv?raw=true"
# Load data and set 'Date' as index.
data_single = pd.read_csv(url, parse_dates=['Date'], index_col='Date')
```

Next, we load the demand data from a GitHub URL and bring it into a format that is easy to work with.

- The data is loaded directly into a **Pandas DataFrame** using `pd.read_csv()`.
- **`parse_dates=['Date']`** automatically converts the 'Date' column into a date format, which allows us to index the data by time.
- **`index_col='Date'`** sets the 'Date' column as the index, so we can easily work with time-series data and analyze patterns over time.

```
# Step 3. Split the dataset into training and testing sets.
# Allocate 80% of the data to training and 20% to testing.
train_size = int(len(data_single) * 0.8)  # Calculate the number of rows to include in the training set (80% of the total data).
train, test = data_single.iloc[:train_size], data_single.iloc[train_size:]  # Use slicing to separate training and testing data.
```

In this step, we split our dataset into training and testing sets, a crucial step in evaluating the forecasting model's effectiveness. By allocating 80% of the data to the training set and the remaining 20% to the testing set, we ensure that the model can learn from a substantial portion of historical data while still having separate data for unbiased performance evaluation.

We calculate `train_size` as 80% of the dataset's length by multiplying `len(data_single)` by 0.8 and converting the result to an integer. This calculation gives us the exact number of rows to use for training. Using `iloc`, we then slice the data: `train` contains the first `train_size` rows, while `test` contains the remaining rows.

This train-test split is essential because it mirrors real-world scenarios where historical data is used to predict future outcomes. The training set enables the model to learn patterns, while the testing set allows us to assess its generalization on unseen data.

```
# Step 4: Fit the Exponential Moving Average (EMA) model on the training data.
# We use the ExponentialSmoothing function from the statsmodels library.
# Here, we only include a trend component (additive trend) and no seasonal component (set to None).
ema_model = ExponentialSmoothing(train['Demand'], trend='add', seasonal=None).fit()  # Fit the EMA model on the training data.
```

In Step 4, we apply the Exponential Moving Average (EMA) model to the training data to capture demand trends. For this, we use the `ExponentialSmoothing` function from the `statsmodels` library. EMA is a common time series forecasting method that assigns greater weight to recent observations, allowing it to respond to recent changes more readily than traditional moving averages.

In this specific implementation, we configure the `ExponentialSmoothing` function to use only a trend component, which we set as `trend='add'` for an additive trend model. This choice means that any upward or downward shifts in demand over time are added directly to the forecasted values, making it effective for capturing linear trends. We omit a seasonal component by setting `seasonal=None`, which simplifies the model and makes it well-suited for non-seasonal demand data.

The `.fit()` method then trains the model on the training data. This fitting process adjusts the model's parameters to minimize the forecast error over the training period, enabling it to produce reliable forecasts based on the established trend. The resulting `ema_model` captures the demand trends from the training data, which we can later use for forecasting and evaluation.

```
# Step 5: Forecast using the EMA model.
# Forecast values for the training period using fitted values and predict the next steps for the test period.
```

```
train['EMA_Forecast'] = ema_model.fittedvalues  # Use fitted values to get the model's predictions on the training data.
test['EMA_Forecast'] = ema_model.forecast(steps=len(test))  # Forecast future values for the test set.
```

In Step 5, we use the trained Exponential Moving Average (EMA) model to generate forecasts for both the training and testing periods. The goal here is to see how the model performs on data it was trained on and assess its predictive accuracy on unseen test data.

First, we use the `fittedvalues` attribute of the `ema_model` to obtain the model's predictions over the training period. This step essentially retrieves the forecasted values that the model calculated based on its parameters during the training phase, stored in the `train['EMA_Forecast']` column. By comparing these fitted values to the actual demand values in the training dataset, we can examine how closely the model captures historical trends.

Next, we use the `forecast()` method of the `ema_model` to predict future demand for the length of the testing period, storing these forecasts in the `test['EMA_Forecast']` column. The `forecast()` function takes an argument specifying the number of steps to project, allowing the model to predict one time step at a time until it reaches the end of the testing dataset.

By creating forecasted values for both training and test periods, we set up a basis for evaluating how well the EMA model fits the historical data and how accurately it projects future demand. This two-fold approach provides a comprehensive view of the model's effectiveness in real-time forecasting scenarios, where consistency and accuracy across periods are essential.

```
# Step 6: Visualize the EMA results.
plt.figure(figsize=(14, 7))  # Set the figure size to 14 inches by 7 inches for better readability.

# Plot the actual demand values for the training period.
plt.plot(train.index, train['Demand'], label='Training Data', color='blue', linestyle='-')  # Training data in blue.
# Plot the actual demand values for the testing period.
plt.plot(test.index, test['Demand'], label='Test Data', color='orange', linestyle='--')  # Test data in orange.
# Plot the EMA forecast values for the testing period.
```

```
plt.plot(test.index, test['EMA_Forecast'], label='EMA Forecast', color='red', linestyle='-')  # EMA forecast in red.

# Highlight the point where the training data ends and the test data begins.
plt.axvline(x=train.index[-1], color='black', linestyle=':', linewidth=1.5, label='Train-Test Split')  # Vertical line indicating the split.

# Add labels, title, and legend to make the plot easier to understand.
plt.title('Exponential Moving Average (EMA) Model - Actual vs. Forecasted Demand')  # Title of the plot.
plt.xlabel('Date')  # X-axis label indicating the dates.
plt.ylabel('Demand')  # Y-axis label indicating demand values.
plt.legend()  # Display the legend to identify each line in the plot.
plt.grid(True)  # Add a grid for easier interpretation of the values.

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

In Step 6, we visualize the results of the Exponential Moving Average (EMA) model to understand how well it aligns with the actual demand data. This visualization is key for interpreting the model's performance visually, especially in capturing trends and patterns over time.

We start by defining a figure size of 14 by 7 inches for better readability. Then, we plot three sets of data to illustrate the EMA model's effectiveness in both training and testing periods:

1. **Training Data**: The actual demand values for the training period are plotted in blue with a solid line. This line represents the historical demand data the model was trained on, giving us a baseline to see how well the model learned these patterns.
2. **Test Data**: The actual demand values for the testing period are plotted in orange with a dashed line. This represents real-world demand that the model has not seen, providing a way to evaluate how well the model can forecast new, unseen data.
3. **EMA Forecast**: The EMA forecasted values for the testing period are plotted in red with a solid line. This line shows the model's predicted demand values for the test period,

allowing a direct comparison against the actual test data (in orange) to gauge forecasting accuracy.

To highlight the transition from training to testing data, we add a vertical line in black at the point where the training data ends and testing data begins. This division helps us see where the model transitions from fitting historical data to forecasting future demand.

Finally, we add titles and labels to make the plot clear and informative. The gridlines and legend further enhance readability by clarifying each line's role in the plot. Displaying this plot allows us to visually assess the EMA model's forecasting accuracy, particularly in terms of whether it captures overall trends and how it handles fluctuations in demand.

```
# Step 7: Calculate Performance Metrics for the EMA Model
# Calculate Mean Absolute Error (MAE) to measure the average magnitude of the errors between actual and forecasted values.
mae_EMA = mean_absolute_error(test['Demand'], test['EMA_Forecast'])
print(f"Mean Absolute Error (MAE): {mae_EMA:.4f}")

# Calculate Root Mean Squared Error (RMSE) to measure the square root of the average of squared differences between actual and forecasted values.
# This metric penalizes larger errors more than MAE.
rmse_EMA = mean_squared_error(test['Demand'], test['EMA_Forecast'], squared=False)  # Set squared=False to get RMSE.
print(f"Root Mean Squared Error (RMSE): {rmse_EMA:.4f}")

# Calculate Mean Absolute Percentage Error (MAPE) to express the error as a percentage of the actual values.
# Adding a small epsilon value (1e-10) helps avoid division by zero if any demand values are zero.
epsilon = 1e-10  # A small value to handle division by zero, if any demand values are zero.
mape_EMA = np.mean(np.abs((test['Demand'] - test['EMA_Forecast']) / (test['Demand'] + epsilon))) * 100
print(f"Mean Absolute Percentage Error (MAPE): {mape_EMA:.4f}%")
```

In Step 7, we evaluate the accuracy of the Exponential Moving Average (EMA) model by calculating three widely used performance metrics: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and Mean Absolute Percentage Error (MAPE). Each of these metrics provides unique insights into how well the model's forecasted values align with the actual demand values in the test dataset.

1. **Mean Absolute Error (MAE)**:
    - MAE represents the average of the absolute differences between the actual and forecasted values. It calculates the average error magnitude without considering whether the forecasted values are too high or too low.
    - MAE is calculated by using `mean_absolute_error` from `sklearn.metrics`. Here, it's applied to the actual test demand values (`test['Demand']`) and the model's forecasted values (`test['EMA_Forecast']`).
    - The resulting MAE gives a straightforward understanding of the model's typical error in predicting demand, presented as an absolute value.
2. **Root Mean Squared Error (RMSE)**:
    - RMSE measures the square root of the average of squared differences between actual and forecasted values. Unlike MAE, RMSE places a greater emphasis on larger errors due to the squaring operation, making it particularly sensitive to significant deviations between actual and predicted values.
    - This metric is calculated by setting `squared=False` in the `mean_squared_error` function to obtain the root of the mean squared differences.
    - A lower RMSE value indicates a model that is less likely to produce large errors in predictions.
3. **Mean Absolute Percentage Error (MAPE)**:
    - MAPE provides an error percentage by comparing the absolute differences between actual and forecasted values to the actual values. This metric shows how large the errors are in relation to the actual values, offering a percentage that helps interpret forecast accuracy in proportional terms.
    - An epsilon (`1e-10`) is added to the denominator to prevent division by zero in case any demand values are zero.

- A lower MAPE suggests that the forecasted values are closer to the actual values, proportionally, which is particularly helpful in comparing forecasting performance across different scales.

By examining these three metrics, we gain a rounded understanding of the EMA model's performance. MAE provides a basic measure of average error, RMSE highlights any large errors more prominently, and MAPE shows the error in terms of percentages, which can be particularly informative when dealing with varying scales in demand. Together, these metrics help assess how well EMA captures demand trends and pinpoint any significant deviations in forecasting accuracy.

Technical Appendix 2.C. Implementing ARIMA: A Step by Step Explanation

```
# Step 1. Import necessary libraries for data loading, ARIMA modeling, and visualization
import pandas as pd  # For data manipulation and time series analysis.
from statsmodels.tsa.arima.model import ARIMA  # ARIMA is used for modeling time series data with auto-regression and moving averages.
from sklearn.metrics import mean_absolute_error, mean_squared_error  # For calculating evaluation metrics.
import matplotlib.pyplot as plt  # For visualizing the results.
import numpy as np  # For mathematical operations.
```

This code imports several essential Python libraries that enable us to load, manipulate, model, evaluate, and visualize data. Besides the libraries explained before, a few libraries are specific to ARIMA.

Statsmodels is a Python library specifically designed for statistical modeling and econometrics, offering a range of tools for analyzing and modeling time series data. In this code, we import the ARIMA model from `statsmodels.tsa.arima.model`. This library streamlines the ARIMA implementation, allowing for straightforward setup and tuning by providing an accessible API to fit and forecast models. Once configured, the ARIMA model can generate forecasts, which are

invaluable for time series forecasting applications like demand prediction in supply chain management.

```
# Step 2.  Load the data file from GitHub into a pandas DataFrame.
url = "https://github.com/CharlesCLuo/Application-of-AI-in-Supply-Chain-Risk-Management-Series/blob/main/Demand_Forecsting/demand_data.csv?raw=true"
# Load data and set 'Date' as index.
data_single = pd.read_csv(url, parse_dates=['Date'], index_col='Date')

# Step 3. Split the dataset into training and testing sets.
train_size = int(len(data_single) * 0.8)  # Calculate the number of rows for the training set (80% of the total data).
train, test = data_single.iloc[:train_size], data_single.iloc[train_size:]  # Use slicing to split the data into training and testing sets.
```

In Step 2, we load the demand data from a GitHub URL using Pandas.
In Step 3, we split the data into training and testing sets, with the training set (80% of the data) used to train the ARIMA model, and the testing set (20% of the data) reserved for evaluating the model's performance on unseen data.

```
# Step 4. Fit the ARIMA model on the training demand data.
# Parameters: p = 5 (lag observations), d = 1 (difference the data once), q = 0 (no moving average component)
arima_model = ARIMA(train['Demand'], order=(5, 1, 0)).fit()  # Fit the ARIMA model on the training data.
```

In step 4, we use the ARIMA model to learn patterns in the training demand data. This step is essential as it “teaches” the ARIMA model the structure of the demand data using the training set, setting up the model to make informed predictions about future demand values. By carefully specifying the values for `p`, `d`, and `q`, we ensure the model captures the relevant patterns, removes noise, and provides reliable forecasts. Ultimately, this step enables the ARIMA model to generate informed forecasts that are grounded in the real patterns and fluctuations observed in the historical demand data.

The ARIMA model uses three parameters—**p**, **d**, and **q**—each play a distinct role in shaping how the model interprets and forecasts the data. Let's explore each parameter in turn to understand their impact on the model.

- **p = 5**: The **autoregressive (AR)** component. This parameter, represented by **p**, tells the model how many past observations to consider when making a prediction. By taking into account previous demand values, the model can "remember" recent trends or shifts that may carry forward. Setting **p=5** means that, to forecast demand for the next day, the model will look at the previous five days of demand data. By focusing on recent observations, the AR component allows the model to capture short-term relationships in the data, providing a more responsive prediction for the next time point. This is especially useful in contexts where demand tends to fluctuate based on recent patterns.
- **d = 1**: The **differencing (I)** component. Differencing is used to make the data stationary, which is a core requirement for many time series models. "Stationary" data means that overall patterns like trends are removed, allowing the model to focus on the underlying fluctuations or seasonal behaviors. By setting **d=1**, we instruct the model to calculate the difference between each value and its preceding value. This single differencing level removes the underlying trend in the data, allowing the model to analyze and forecast based on variations around a stable mean. This is crucial in demand forecasting, as we want to detect and respond to demand fluctuations rather than simply following a longer-term trend.
- **q = 0**: The **moving average (MA)** component. The moving average component, represented by **q**, allows the model to adjust predictions based on patterns in the residuals or errors of previous predictions. Here, we set **q=0**, meaning the model does not adjust based on past forecast errors. In this case, we assume that there are no error-based patterns significant enough to influence future predictions. This setup works when we believe past prediction errors are random and do not contain information that could help in future forecasts.

Fitting the Model: .fit() Method: After defining the ARIMA model with the specified values for **p**, **d**, and **q**, we use the `.fit()` method to train it on our training dataset. The `.fit()` function allows the model to learn the patterns, relationships, and seasonality within the

historical demand data by applying the ARIMA formula with our chosen parameters. Once trained, the model, stored in `arima_model`, is now equipped to make predictions on both the training data (fitted values) and new data (test data) by following the patterns it has learned.

```
# Step 5. Make predictions using the ARIMA model.
train['ARIMA_Forecast'] = arima_model.fittedvalues  # Get the model's predictions on the training data (fitted values).
test['ARIMA_Forecast'] = arima_model.forecast(steps=len(test))  # Forecast future values for the test set using the model.
```

In Step 5, we evaluate the model's reliability by comparing both its "learned" values and its forecasts against actual demand, giving us a clear picture of its forecasting ability.

train['ARIMA_Forecast'] = arima_model.fittedvalues

The `fittedvalues` attribute of the ARIMA model provides the model's predictions on the **training set** (the data it was trained on). These are not "new" predictions but rather the model's best fit for the known data points in the training set. By comparing these fitted values to the actual values in the training set, we can assess how well the ARIMA model has learned the underlying patterns. A close match between the fitted values and the actual values indicates that the model has effectively captured the historical demand pattern.

test['ARIMA_Forecast'] = arima_model.forecast(steps=len(test))

The `forecast()` method generates predictions for a specified number of future data points, which, in this case, is the size of the test set. Here, `steps=len(test)` directs the model to forecast demand values for each date in the testing set. These predictions simulate how well the model can forecast demand on **new, unseen data**. Evaluating the ARIMA model on the testing set helps determine if the patterns it learned from the training data apply effectively to future data points.

References

Chase, C. W. (2013). *Demand-Driven Forecasting: A Structured Approach to Forecasting*. Wiley.
Chase presents structured techniques for demand-driven forecasting, emphasizing practical applications to align forecasting efforts with business needs.

Hyndman, R. J., & Athanasopoulos, G. (2021). *Forecasting: Principles and Practice* (3rd ed.).
A practical, accessible guide to forecasting, covering statistical and machine learning techniques, aimed at both students and practitioners.

Makridakis, S., Wheelwright, S. C., & Hyndman, R. J. (2008). *Forecasting: Methods and Applications*. John Wiley & Sons.
This reference book offers an in-depth exploration of various forecasting methods, providing both theoretical understanding and practical applications.

McKinney, W. (2022). Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter (3rd ed.). O'Reilly Media.

This book is widely considered one of the best resources for learning data analysis with Python. It covers the use of pandas, NumPy, and IPython for data manipulation and analysis.