Chapter 1

ARIMA

Amit Dua

Put abstract here with \abstract.

Change with \papernote

Chapter 2

Time Series Analytics

Abstract

This chapter presents time series analysis theory and practice using YouTube, SEO, and NSE data. It provides implementations in both Excel and Python.

1 Foundations of Time Series Analysis

1.1 Theory: What is Time Series Data?

Definition: A time series is a sequence of data points indexed in time order. Unlike cross-sectional data, the temporal ordering matters because values are often dependent on previous observations.

Mathematical Representation:

$$Y_t = f(Y_{t-1}, Y_{t-2}, \dots, Y_{t-k}, \epsilon_t) \tag{2.1}$$

Where:

- Y_t = Value at time t
- ϵ_t = Random error term
- k = Number of lags

Key Properties:

- **Temporal Dependence**: Today's stock price depends on yesterday's price
- **Autocorrelation**: Correlation between observations at different time points
- **Trend**: Long-term movement in data
- **Seasonality**: Regular patterns that repeat over fixed periods

Types of Time Series:

- **Stationary:** Constant mean, variance, and covariance over time
- **Non-stationary:** Mean/variance changes over time
- **Seasonal:** Regular patterns (daily, monthly, yearly)
- **Irregular:** Random fluctuations

1.1.1 Python Implementation (Gold Price Data)

```
# Using yfinance (Gold Futures GC=F)
import yfinance as yf

gold = yf.download('GC=F', start='2015-06-01',
end='2025-06-01', interval='1d')
gold = gold.reset_index()[['Date', 'Close']]
gold.to_csv('gold_10y.csv', index=False)
```

Notes:

- Uses COMEX Gold Futures as proxy for spot prices
- Automatically handles weekend gaps in futures data
- For physical gold prices, use `GLD` ETF instead

1.1.2 Python Implementation (NSE Stock Data)

```
# Download MRF stock from NSE India
mrf = yf.download('MRF.NS', start='2015-06-01',
end='2025-06-01', interval='1d')
mrf = mrf.reset_index()[['Date', 'Close']]
mrf.to_csv('mrf_10y.csv', index=False)
```

Notes:

- Add `.NS` suffix for any NSE-listed stock
- Contains adjusted closing prices automatically
- 15-minute delay for live Indian market data

1.1.3 Python Implementation (Cryptocurrency Data)

```
# Bitcoin historical data (daily)
btc = yf.download('BTC-USD', start='2015-06-01',
end='2025-06-01', interval='1d')
btc = btc.reset_index()[['Date', 'Close']]
btc.to_csv('btc_10y.csv', index=False)
```

Notes:

- 24/7 trading data - no date gaps
- For Ethereum: use `ETH-USD` ticker
- Includes all trading pairs (USD, INR, etc.)

1.1.4 Python Implementation (Economic Data)

```
# COVID-19 data from Our World in Data
import pandas as pd

covid_url = "https://covid.ourworldindata.org/data/owid-covid-data.csv"
pd.read_csv(covid_url).to_csv('covid_global.csv', index=False)

# Temperature data (New Delhi example)
!pip install meteostat
from meteostat import Point, Daily

data = Daily(Point(28.6139, 77.2090),
start='2015-06-01', end='2025-06-01').fetch()
data.to_csv('delhi_temp.csv')
```

Notes:

- COVID data updated daily with 100+ metrics
- Temperature data requires latitude/longitude
- Meteorological data has 1-day latency

1.2 Practical: Data Preparation & Visualization

1.2.1 Excel Implementation (NSE MRF Stock Data)

Step 1: Data Import

```
1. Download MRF.csv from NSE India.
2. Data > From Text/CSV > Transform > Load.
3. Ensure the Date column is in Date format: Format > Cells > Date.
```

Step 2: Prepare Data for Trend Analysis

```
1. Identify columns:
- Column A: Date
- Column G: close (closing price)
2. Create a helper column for trend analysis:
- In column N (Row Number), enter 1 for the first data row, 2 for the second, and
    so on (or use =ROW()-1 and fill down).
```

Step 3: Linear Trend Analysis

```
1. Calculate the slope of the trend:
=SLOPE(G2:G38, N2:N38)

2. Calculate the intercept:
=INTERCEPT(G2:G38, N2:N38)

3. Generate fitted (trend) values for each row:
=TREND(G2:G38, N2:N38, N2:N38)
(Enter in O2 and fill down to O38)

4. Obtain full regression statistics (slope, intercept, R-squared, etc.):
- Select a 5×2 block of empty cells (e.g., P2:Q6)
- Enter:
=LINEST(G2:G38, N2:N38, TRUE, TRUE)
- Press Ctrl+Shift+Enter (for array formula)
```

Step 4: Summary Statistics

```
=AVERAGE(G2:G38)           // Mean closing price
=STDEV.S(G2:G38)           // Standard deviation of closing price
=MAX(G2:G38)-MIN(G2:G38)   // Range of closing price
```

Interpreting the Results:

- **Slope**: The average change in closing price per time period (Row Number). A positive value indicates an upward trend; a negative value indicates a downward trend.
- **Intercept**: The estimated closing price when Row Number is zero (theoretical starting value).
- **Fitted Values**: The predicted closing price for each period, based on the linear trend.

- **LINEST Output:**
  - Top left: Slope
  - Top right: Intercept
  - 2nd row: Standard errors
  - 3rd row: R-squared (goodness of fit), standard error of estimate
  - 4th row: F-statistic, degrees of freedom
  - 5th row: Regression sum of squares, residual sum of squares
- **Mean, Standard Deviation, Range:** Basic descriptive statistics for the closing price.

Tip: To visualize the trend, insert a scatter plot of Date vs. close, then add a linear trendline (right-click on data points > Add Trendline > Linear) and display the equation on the chart.

1.2.2 Python Implementation (YouTube Views Data)

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from datetime import datetime

# Load and prepare data
df = pd.read_csv('youtube_views.csv', parse_dates=['Date'],
index_col='Date')

# Basic statistics
print("Time Series Summary:")
print(f"Mean: {df['Views'].mean():.2f}")
print(f"Std Dev: {df['Views'].std():.2f}")
print(f"Min: {df['Views'].min()}")
print(f"Max: {df['Views'].max()}")

# Visualization
fig, axes = plt.subplots(2, 1, figsize=(12, 8))

# Time series plot
axes[0].plot(df.index, df['Views'], label='Daily Views')
axes[0].set_title('YouTube Views Over Time')
axes[0].set_ylabel('Views')
axes[0].legend()

```

```
# Distribution plot
axes[1].hist(df['Views'], bins=30, alpha=0.7)
axes[1].set_title('Distribution of Views')
axes[1].set_xlabel('Views')
axes[1].set_ylabel('Frequency')

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

Interpretation Guidelines:

- Look for trends (upward/downward movement)
- Identify seasonal patterns (regular cycles)
- Spot outliers (unusual spikes/drops)
- Check for volatility clustering

2 Moving Averages & Smoothing Techniques

2.1 Theory: Moving Averages

Definition: A moving average smooths time series data by creating a series of averages from different subsets of the full dataset.

Mathematical Formulation:

Simple Moving Average (SMA):

$$SMA_t = \frac{1}{k}\sum_{i=0}^{k-1} Y_{t-i} \tag{2.2}$$

Exponential Moving Average (EMA):

$$EMA_t = \alpha Y_t + (1-\alpha)EMA_{t-1} \tag{2.3}$$

Where α is the smoothing parameter ($0 < \alpha < 1$).

Weighted Moving Average (WMA):

$$WMA_t = \frac{\sum_{i=0}^{k-1} w_i Y_{t-i}}{\sum_{i=0}^{k-1} w_i} \tag{2.4}$$

Where w_i are weights assigned to each observation.

Trade-offs:

- **Lag**: All moving averages lag behind the actual data
- **Smoothness vs. Responsiveness**: Longer windows = smoother but less responsive
- **End-point problem**: Cannot compute MA for the most recent $k-1$ periods

2.2 Practical: Smoothing Techniques

2.2.1 Excel Implementation (Stock Data)

```
Simple Moving Average:
1. Data > Data Analysis > Moving Average
2. Input Range: B2:B100
3. Interval: 5 (for 5-day MA)
4. Output Range: C2
5. Chart Output: Yes
```

2.2.2 Python Implementation (Gold Prices)

```python
import pandas as pd
import matplotlib.pyplot as plt

# Load gold price data
df = pd.read_csv('gold_prices.csv', parse_dates=['Date'],
index_col='Date')

# Simple Moving Averages
df['SMA_5'] = df['Price'].rolling(window=5).mean()
df['SMA_20'] = df['Price'].rolling(window=20).mean()
df['SMA_50'] = df['Price'].rolling(window=50).mean()

# Exponential Moving Average
df['EMA_12'] = df['Price'].ewm(span=12).mean()

# Adaptive Smoothing using Holt-Winters
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# For trend and seasonality
model = ExponentialSmoothing(
df['Price'],
trend='add',            # Additive trend
seasonal='add',         # Additive seasonality
seasonal_periods=12     # Monthly seasonality
)
```

```python
fit = model.fit()
df['Holt_Winters'] = fit.fittedvalues

# Visualization
plt.figure(figsize=(14, 8))
plt.plot(df.index, df['Price'], label='Original', alpha=0.7)
plt.plot(df.index, df['SMA_5'], label='SMA(5)')
plt.plot(df.index, df['SMA_20'], label='SMA(20)')
plt.plot(df.index, df['EMA_12'], label='EMA(12)')
plt.plot(df.index, df['Holt_Winters'], label='Holt-Winters')
plt.title('Gold Prices: Various Smoothing Techniques')
plt.legend()
plt.show()

# Calculate smoothing errors
mse_sma5 = ((df['Price'] - df['SMA_5'])**2).mean()
mse_ema12 = ((df['Price'] - df['EMA_12'])**2).mean()
print(f"MSE SMA(5): {mse_sma5:.2f}")
print(f"MSE EMA(12): {mse_ema12:.2f}")
```

Interpretation:

- Shorter MA periods = More responsive to changes, more noise
- Longer MA periods = Smoother, less responsive
- EMA gives more weight to recent observations
- Choose smoothing method based on forecasting horizon

Scenario	Recommended MA	Why?	Example
Day Trading	5-10 period SMA	Need quick signals	Gold intraday
Swing Trading	20-50 period SMA	Balance noise/lag	MRF weekly
Long-term Investing	200 period SMA	Major trends only	Index funds
High Volatility	Longer periods or EMA	Reduce false signals	Crypto
Trend Following	Multiple MAs (50/200)	Confirmation	Stock portfolios
Seasonal Data	Period = Season length	Match cycle	Monthly sales

3 Stationarity & Data Transformations

3.1 Theory: Stationarity

Definition: A time series is stationary if its statistical properties (mean, variance, covariance) remain constant over time.

Mathematical Definition: A time series $\{Y_t\}$ is **strictly stationary** if:

$$P(Y_{t_1}, Y_{t_2}, \dots, Y_{t_k}) = P(Y_{t_1+h}, Y_{t_2+h}, \dots, Y_{t_k+h}) \tag{2.5}$$

For any lag h and any time points $t_1, t_2, \dots, t_k$.
Weak Stationarity (more practical):

1. **Constant Mean**: $E[Y_t] = \mu$ for all t
2. **Constant Variance**: $Var[Y_t] = \sigma^2$ for all t
3. **Constant Covariance**: $Cov[Y_t, Y_{t+h}] = \gamma(h)$ depends only on lag h

Why Stationarity Matters:

- Most time series models assume stationarity
- Non-stationary series can lead to spurious regression
- Forecasting requires stable statistical relationships

Types of Non-Stationarity:

1. **Trend Stationarity**: Deterministic trend, remove by detrending
2. **Difference Stationarity**: Stochastic trend, remove by differencing
3. **Structural Breaks**: Parameters change at specific points

Stationarity Tests:

1. **Augmented Dickey-Fuller (ADF):** Tests for unit root
   - H_0: Series has unit root (non-stationary)
   - H_1: Series is stationary
2. **KPSS Test:** Tests for stationarity around trend
   - H_0: Series is stationary
   - H_1: Series has unit root

3.2 Why Does Time Series Data Need to Be Stationary?

1. **Most Models Assume Stationarity**
   Statistical models like ARIMA, SARIMA, and many forecasting techniques assume that the data's statistical properties (mean, variance, autocorrelation) do not change over time. If this assumption is violated, model estimates become unreliable, forecasts become inaccurate, and hypothesis tests can be misleading.

2. **Stationarity Ensures Reliable Forecasting**
   Forecasting depends on the idea that patterns in the past will persist into the future. If the mean or variance keeps changing, the model cannot "learn" a stable relationship, making future predictions untrustworthy.

3. **Prevents Spurious Regression**
   Spurious regression is a statistical illusion where two unrelated non-stationary series appear to be strongly related simply because they both wander over time.
   Example: Gold prices and global temperature may both trend upward over decades, but this does not mean one causes the other.

Real-World Examples:

- **Gold Prices:** Raw gold price series often show a long-term upward trend due to inflation and macroeconomic factors. Making the series stationary (by differencing or detrending) removes this artificial relationship and reveals the true underlying dynamics.

- **Stock Prices (e.g., MRF):** Stock prices are typically non-stationary. Stock returns (percentage change in price), however, are often stationary. Models like ARIMA or GARCH require stationary input, so we analyze returns, not prices.

- **Cryptocurrency (e.g., Bitcoin):** Crypto prices are highly volatile and often exhibit structural breaks. Stationarity is required to model volatility or forecast future values.

- **Meteorological Data (e.g., Temperature):** Daily temperature shows strong seasonality. Removing seasonality (e.g., subtracting monthly average) makes the series stationary, enabling accurate modeling and anomaly detection.

Data Type	Raw Data Stationary?	How to Achieve Stationarity
Gold Price	No (trending)	Differencing, detrending
Stock Price	No (trending)	Use returns (log or percent diff)
Crypto Price	No (volatile, breaks)	Returns, structural break tests
Temperature	No (seasonal)	Remove seasonality, detrending

Summary Table:
Key Takeaways:

- Stationarity is the foundation for most time series modeling and forecasting.
- Non-stationary data can mislead models, cause spurious results, and produce unreliable forecasts.
- Always check for stationarity using visual inspection and statistical tests (ADF, KPSS).
- Transform your data (differencing, detrending, removing seasonality) before modeling.

> Stationarity ensures that the patterns we find in our data are real and persistent, not just artifacts of time. Without it, our models are like compasses spinning without direction.

Practice: Try plotting your gold price, stock price, crypto price, and temperature data. Observe the trends and seasonality. Then, difference or detrend the series and see how the statistical properties stabilize—this is the first step to building robust, reliable time series models!

3.3 Python Scripts: Transforming Data to Stationarity

3.3.1 Gold Price (Differencing)

```python
import pandas as pd

gold = pd.read_csv('gold_10y.csv', parse_dates=['Date'])
gold['Gold_Diff'] = gold['Close'].diff()
gold_stationary = gold[['Date', 'Gold_Diff']].dropna()
gold_stationary.to_csv('gold_stationary.csv', index=False)
```

3.3.2 Stock Price (Log Returns)

```
import pandas as pd
import numpy as np

stock = pd.read_csv('mrf_10y.csv', parse_dates=['Date'])
stock['Stock_Returns'] = np.log(stock['Close'] / stock['Close'].shift(1))
stock_stationary = stock[['Date', 'Stock_Returns']].dropna()
stock_stationary.to_csv('stock_stationary.csv', index=False)
```

3.3.3 Crypto Price (Log Returns)

```
import pandas as pd
import numpy as np

btc = pd.read_csv('btc_10y.csv', parse_dates=['Date'])
btc['BTC_Returns'] = np.log(btc['Close'] / btc['Close'].shift(1))
btc_stationary = btc[['Date', 'BTC_Returns']].dropna()
btc_stationary.to_csv('btc_stationary.csv', index=False)
```

3.3.4 Temperature (Remove Seasonality)

```
import pandas as pd

temp = pd.read_csv('delhi_temp_10y.csv', parse_dates=['time'])
temp = temp[['time', 'tavg']].dropna()
temp['Month'] = temp['time'].dt.month
monthly_means = temp.groupby('Month')['tavg'].transform('mean')
temp['Deseasonalized'] = temp['tavg'] - monthly_means
temp_stationary = temp[['time', 'Deseasonalized']].dropna()
temp_stationary.to_csv('temp_stationary.csv', index=False)
```

Colab Download:

```
from google.colab import files
files.download('gold_stationary.csv')
files.download('stock_stationary.csv')
files.download('btc_stationary.csv')
files.download('temp_stationary.csv')
```

3.4 Practical: Testing and Achieving Stationarity

3.4.1 Excel Implementation (NSE Data Stationarity)

Visual Tests:

```
// Plot time series
1. Insert > Charts > Line Chart (Date vs Price)
2. Add trendline: Right-click series > Add Trendline
3. Check if trend is significant

// Calculate first differences
// In column C (assuming prices in B)
=B3-B2  // First difference
```

Statistical Tests (Using NumXL Add-in):

```
1. Install NumXL add-in
2. NumXL > Tests > Unit Root Tests > ADF Test
3. Input: B2:B100 (Price series)
4. Include: Constant + Trend
5. Interpret: p-value < 0.05 indicates stationarity
```

3.4.2 Python Implementation (Comprehensive Stationarity Analysis)

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from scipy import stats

def test_stationarity(timeseries, title='Time Series'):
"""
Comprehensive stationarity testing and visualization
"""
# Plot the time series
fig, axes = plt.subplots(3, 2, figsize=(15, 12))

# Original series
axes[0,0].plot(timeseries)
axes[0,0].set_title(f'{title} - Original')
axes[0,0].set_ylabel('Value')

# Rolling statistics
rolling_mean = timeseries.rolling(window=12).mean()
rolling_std = timeseries.rolling(window=12).std()

axes[0,1].plot(timeseries, label='Original')
axes[0,1].plot(rolling_mean, color='red', label='Rolling Mean')
axes[0,1].plot(rolling_std, color='black', label='Rolling Std')
axes[0,1].set_title('Rolling Statistics')
axes[0,1].legend()
```

```
# Distribution
axes[1,0].hist(timeseries.dropna(), bins=30, alpha=0.7)
axes[1,0].set_title('Distribution')
axes[1,0].axvline(timeseries.mean(), color='red',
linestyle='--', label='Mean')

# Q-Q plot
stats.probplot(timeseries.dropna(), dist="norm",
plot=axes[1,1])
axes[1,1].set_title('Q-Q Plot')

# ACF and PACF
plot_acf(timeseries.dropna(), ax=axes[2,0], lags=20)
plot_pacf(timeseries.dropna(), ax=axes[2,1], lags=20)

plt.tight_layout()
plt.show()

# Statistical tests
print(f"Stationarity Tests for {title}")
print("="*50)

# ADF Test
adf_result = adfuller(timeseries.dropna(), autolag='AIC')
print(f"ADF Test:")
print(f"  Test Statistic: {adf_result[0]:.4f}")
print(f"  p-value: {adf_result[1]:.4f}")
print(f"  Critical Values:")
for key, value in adf_result[4].items():
print(f"    {key}: {value:.4f}")

# KPSS Test
kpss_result = kpss(timeseries.dropna(), regression='ct')
print(f"\nKPSS Test:")
print(f"  Test Statistic: {kpss_result[0]:.4f}")
print(f"  p-value: {kpss_result[1]:.4f}")
print(f"  Critical Values:")
for key, value in kpss_result[3].items():
print(f"    {key}: {value:.4f}")

# Interpretation
print("\nInterpretation:")
if adf_result[1] < 0.05 and kpss_result[1] > 0.05:
print("Series is stationary")
elif adf_result[1] > 0.05 and kpss_result[1] < 0.05:
print("Series is non-stationary (unit root)")
elif adf_result[1] > 0.05 and kpss_result[1] > 0.05:
```

```
print("Series is trend stationary")
else:
print("Results are inconclusive")

return adf_result, kpss_result

def make_stationary(timeseries, method='difference'):
"""
Transform series to achieve stationarity
"""
if method == 'difference':
# First differencing
diff_series = timeseries.diff().dropna()
return diff_series, 1

elif method == 'log_difference':
# Log transformation followed by differencing
log_series = np.log(timeseries)
log_diff = log_series.diff().dropna()
return log_diff, 1

elif method == 'detrend':
# Linear detrending
from scipy import signal
detrended = signal.detrend(timeseries)
return pd.Series(detrended, index=timeseries.index), 0

elif method == 'seasonal_difference':
# Seasonal differencing
seasonal_diff = timeseries.diff(12).dropna()  # Monthly
return seasonal_diff, 12

# Load data
df = pd.read_csv('website_traffic.csv', parse_dates=['Date'],
index_col='Date')

# Test original series
print("Testing Original Series")
adf_orig, kpss_orig = test_stationarity(df['Traffic'],
'Website Traffic')

# If non-stationary, apply transformations
if adf_orig[1] > 0.05:
print("\nApplying Transformations...")

# Try different methods
methods = ['difference', 'log_difference', 'detrend']
results = {}
```

```python

for method in methods:
print(f"\nTrying {method}...")
transformed, order = make_stationary(df['Traffic'],
method)
adf_trans, kpss_trans = test_stationarity(
transformed, f'Traffic - {method}')
results[method] = {
  'series': transformed,
  'adf_pvalue': adf_trans[1],
  'kpss_pvalue': kpss_trans[1],
  'order': order
}

# Select best transformation
best_method = min(results, key=lambda x: results[x]['adf_pvalue'])
print(f"\nBest transformation: {best_method}")
print(f"ADF p-value: {results[best_method]['adf_pvalue']:.4f}")

# Box-Cox transformation
def box_cox_transform(data):
"""
Apply Box-Cox transformation
"""
# Find optimal lambda
fitted_data, fitted_lambda = stats.boxcox(data[data > 0])
print(f"Optimal lambda: {fitted_lambda:.4f}")
return fitted_data, fitted_lambda

# Apply Box-Cox
transformed_data, lambda_val = box_cox_transform(df['Traffic'])

# Inverse transformation for forecasting
def inverse_box_cox(data, lambda_val):
if lambda_val == 0:
return np.exp(data)
else:
return np.power(lambda_val * data + 1, 1/lambda_val)
```

Decision Framework:

- ADF: Stationary, KPSS: Stationary → **Series is stationary**
- ADF: Non-stationary, KPSS: Stationary → **Trend stationary (detrend)**
- ADF: Non-stationary, KPSS: Non-stationary → **Difference stationary (difference)**
- ADF: Stationary, KPSS: Non-stationary → **Need further investigation**

4 ARMA & ARIMA Models

4.1 Theory: ARMA Models

Autoregressive (AR) Model: An AR(p) model expresses current value as a linear combination of past values:

$$Y_t = c + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \ldots + \phi_p Y_{t-p} + \epsilon_t \tag{2.6}$$

Where:

- c = constant
- ϕ_i = autoregressive parameters
- ϵ_t = white noise error term

Real-World Example (Gold Prices):

- *AR(2) Model:* Current gold price depends on past 2 months' prices
- $\phi_1 = 0.6$: 60% weight to last month's price
- $\phi_2 = 0.3$: 30% weight to price from two months ago
- Shock decay: \$100 price spike ⟶ \$80 ⟶ \$64 ⟶ ... (stationary)

Analogy:

- Your current mood (Y_t) depends on:
- 70% of yesterday's mood ($\phi_1 = 0.7$)
- 20% of the mood from two days ago ($\phi_2 = 0.2$)
- Plus today's random events (ϵ_t)

Moving Average (MA) Model: An MA(q) model expresses current value as a linear combination of past error terms:

$$Y_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_q \epsilon_{t-q} \tag{2.7}$$

Where:

- μ = mean of the series

- θ_i = moving average parameters

Real-World Example (Bitcoin Prices):

- *MA(2) Model:* Current BTC price reflects recent market shocks
- $\theta_1 = 0.6$: 60% of yesterday's news impact
- $\theta_2 = 0.3$: 30% of news from two days ago
- $5,000 hack effect: -$3,000 ⟶ -$1,500 ⟶ $0 (finite memory)

Analogy:

- Today's stock price (Y_t) reacts to:
- Today's earnings report (ϵ_t)
- 50% residual impact from yesterday's FDA approval ($\theta_1 = 0.5$)
- 20% lingering effect from last week's CEO scandal ($\theta_2 = 0.2$)

ARMA Model: ARMA(p,q) combines both AR and MA components:

$$Y_t = c + \phi_1 Y_{t-1} + \ldots + \phi_p Y_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q} \tag{2.8}$$

Stationarity Conditions:

- For AR(p): All roots of $\phi(z) = 1 - \phi_1 z - \ldots - \phi_p z^p = 0$ must lie outside unit circle
- For MA(q): All roots of $\theta(z) = 1 + \theta_1 z + \ldots + \theta_q z^q = 0$ must lie outside unit circle

ARIMA Models: ARIMA(p,d,q) extends ARMA to handle non-stationary data:

- **p**: Order of autoregression
- **d**: Degree of differencing
- **q**: Order of moving average

$$\phi(B)(1 - B)^d Y_t = \theta(B)\epsilon_t \tag{2.9}$$

Where B is the backshift operator: $BY_t = Y_{t-1}$.
Model Selection:

- **ACF/PACF Patterns:** Identify model order
- **Information Criteria:** AIC, BIC for model comparison
- **Residual Analysis:** Check for white noise residuals

Autoregressive Integrated Moving Average (ARIMA) Model: An ARIMA(p,d,q) model combines differencing with AR and MA components to handle non-stationary data:

$$(1 - \phi_1 B - \cdots - \phi_p B^p)(1 - B)^d Y_t = (1 + \theta_1 B + \cdots + \theta_q B^q)\epsilon_t \tag{2.10}$$

Where:

- p = AR order (past values)
- d = Differencing order (make series stationary)
- q = MA order (past errors)
- B = Backshift operator: $BY_t = Y_{t-1}$

Real-World Example (Gold Prices):

- *ARIMA(1,1,1) Model:*
- Raw gold prices need 1 differencing ($d = 1$) to remove trend: $Y'_t = Y_t - Y_{t-1}$
- $\phi_1 = 0.4$: 40% of yesterday's *price change* affects today
- $\theta_1 = 0.3$: 30% of yesterday's shock lingers
- A \$100 price jump becomes \$40 $\longrightarrow$ \$16 $\longrightarrow$ ... (controlled decay)

Analogy:

- Your *weight loss journey* (Y_t):
- First difference ($d = 1$): Focus on *weekly changes* rather than absolute weight
- AR(1): 50% of last week's progress ($\phi_1 = 0.5$) influences this week
- MA(1): 30% impact from last week's cheat meal ($\theta_1 = 0.3$)
- Combines persistent habits (AR) with temporary setbacks (MA)

Key Differences from AR/MA:

- Handles *trends* through differencing (d)
- Models *changes* rather than raw values
- Example: Bitcoin prices after removing bubble effects

Component	Role	Gold Price Example
AR(p)	Past price changes	60% of last month's change
I(d)	Remove trend	1 differencing
MA(q)	Past market shocks	30% of last month's news impact

4.2 Python Implementation: AR, MA, and ARIMA Models

4.2.1 Autoregressive (AR) Model

```
import pandas as pd
from statsmodels.tsa.ar_model import AutoReg
import matplotlib.pyplot as plt

# Load gold price data
gold = pd.read_csv('gold_10y.csv', parse_dates=['Date'])
gold = gold.set_index('Date').asfreq('B')  # business days frequency
gold['Close'] = gold['Close'].interpolate()  # fill missing values

# Fit AR(1) model (current value regressed on previous value)
model_ar = AutoReg(gold['Close'], lags=1).fit()
gold['AR1_pred'] = model_ar.fittedvalues

# Plot results
gold[['Close', 'AR1_pred']].plot(figsize=(12,4),
title='AR(1) Model: Gold Price')
plt.show()

# Print model summary
print(model_ar.summary())
```

Output Interpretation:

- **Plot Analysis:** The AR(1) prediction line closely follows the actual gold price but with a slight lag

- **Model Behavior:** AR models capture persistence (today's price depends heavily on yesterday's price)
- **Limitation:** Cannot capture sudden jumps or structural breaks well
- **Best Use:** When data shows strong autocorrelation and gradual changes

4.2.2 Moving Average (MA) Model

```
from statsmodels.tsa.arima.model import ARIMA

# Fit MA(1) model (current value depends on current and previous error)
model_ma = ARIMA(gold['Close'], order=(0,0,1)).fit()
gold['MA1_pred'] = model_ma.fittedvalues

# Plot results
gold[['Close', 'MA1_pred']].plot(figsize=(12,4),
title='MA(1) Model: Gold Price')
plt.show()

# Print model summary
print(model_ma.summary())
```

Output Interpretation:

- **Plot Analysis**: MA prediction appears smoother and may lag behind sudden price movements
- **Model Behavior**: Captures short-term shocks and their immediate effects
- **Strength**: Good for data where recent surprises influence current values
- **Limitation:** May not capture long-term trends as effectively as AR models

4.2.3 ARIMA Model

```
# First, difference the series to remove trend (d=1)
gold['Close_diff'] = gold['Close'].diff()

# Fit ARIMA(1,1,1): AR(1) + 1 difference + MA(1)
model_arima = ARIMA(gold['Close'], order=(1,1,1)).fit()
gold['ARIMA_pred'] = model_arima.fittedvalues

# Plot differenced series and ARIMA predictions
gold[['Close_diff', 'ARIMA_pred']].dropna().plot(figsize=(12,4),
```

```python
title='ARIMA(1,1,1) Model: Gold Price Changes')
plt.show()

# Print model summary
print(model_arima.summary())
```

Output Interpretation:

- **Plot Analysis:** Shows the differenced series (price changes) and model predictions
- **Key Insight:** ARIMA models changes, not levels—focuses on whether price goes up/down
- **Advantage:** Handles non-stationary data by differencing first
- **Practical Use:** Best for trending data like financial prices

4.3 Model Comparison Guidelines

What to Look For in Plots:

- **Closeness of Fit:** How well does the predicted line track the actual data?
- **Lag Behavior:** Does the model prediction lag behind sudden changes?
- **Residual Patterns:** Are there systematic patterns in prediction errors?

Model Selection Criteria:

Model	Best For	Key Indicator
AR(p)	Persistent, stationary data	High autocorrelation
MA(q)	Shock-driven, stationary data	Short memory effects
ARIMA(p,d,q)	Trending, non-stationary data	Unit root present

Diagnostic Checks:

```python
# Check residuals for white noise
residuals = model_arima.resid
print("Ljung-Box Test p-value:",
acorr_ljungbox(residuals, lags=10, return_df=True)['lb_pvalue'].iloc[0])

# Plot residuals
residuals.plot(title='Model Residuals')
plt.show()
```

Note: A good model should have residuals that look like white noise (random, no patterns). If patterns remain, consider different model orders or transformations.

4.4 Practical: Building ARMA/ARIMA Models

4.4.1 Excel Implementation (Using NumXL Add-in)

ARMA Model Building:

```
1. Ensure data is stationary (use ADF test from Module 3)
2. NumXL > Model > ARMA
3. Input Range: Stationary series
4. Model Order: Start with ARMA(1,1)
5. Estimation Method: Maximum Likelihood
6. Generate Forecasts: Yes
```

Model Diagnostics:

```
1. NumXL > Model > Residual Analysis
2. Check ACF of residuals (should be white noise)
3. Ljung-Box test: p-value > 0.05 for good model
```

4.4.2 Python Implementation (Complete ARIMA Workflow)

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.stats.diagnostic import acorr_ljungbox
import warnings
warnings.filterwarnings('ignore')

# Load YouTube views data
df = pd.read_csv('youtube_views.csv', parse_dates=['Date'],
index_col='Date')

def identify_arima_order(data, max_p=5, max_q=5):
"""
Identify optimal ARIMA order using ACF/PACF and
information criteria
"""
# Plot ACF and PACF
fig, axes = plt.subplots(1, 2, figsize=(15, 5))
plot_acf(data.dropna(), ax=axes[0], lags=20)
plot_pacf(data.dropna(), ax=axes[1], lags=20)
plt.show()

# Grid search for optimal parameters
best_aic = np.inf
```

```
best_order = None
aic_results = []

for p in range(max_p + 1):
for q in range(max_q + 1):
try:
model = ARIMA(data, order=(p, 0, q))
results = model.fit()
aic = results.aic
aic_results.append((p, q, aic))

if aic < best_aic:
best_aic = aic
best_order = (p, 0, q)
except:
continue

# Display AIC table
aic_df = pd.DataFrame(aic_results,
columns=['p', 'q', 'AIC'])
print("AIC Values for Different Orders:")
print(aic_df.pivot(index='p', columns='q',
values='AIC').round(2))

print(f"\nBest order: ARMA{best_order[0], best_order[2]}")
print(f"Best AIC: {best_aic:.2f}")

return best_order

def fit_arima_model(data, order):
"""
Fit ARIMA model and perform diagnostics
"""
model = ARIMA(data, order=order)
results = model.fit()

print(results.summary())

# Diagnostic plots
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# Residuals plot
residuals = results.resid
axes[0, 0].plot(residuals)
axes[0, 0].set_title('Residuals')
axes[0, 0].axhline(y=0, color='red', linestyle='--')

# Residuals distribution
```

```
axes[0, 1].hist(residuals, bins=30, alpha=0.7)
axes[0, 1].set_title('Residuals Distribution')

# ACF of residuals
plot_acf(residuals, ax=axes[1, 0], lags=20)
axes[1, 0].set_title('ACF of Residuals')

# Q-Q plot
from scipy import stats
stats.probplot(residuals, plot=axes[1, 1])
axes[1, 1].set_title('Q-Q Plot')

plt.tight_layout()
plt.show()

# Ljung-Box test
lb_test = acorr_ljungbox(residuals, lags=10,
return_df=True)
print("\nLjung-Box Test:")
print(lb_test)

if lb_test['lb_pvalue'].min() > 0.05:
print("Residuals appear to be white noise (Good!)")
else:
print("Residuals show autocorrelation (Model may need improvement)")

return results

def forecast_arima(model_results, steps=30):
"""
Generate forecasts with confidence intervals
"""
forecast = model_results.forecast(steps=steps)
conf_int = model_results.get_forecast(steps=steps).conf_int()

# Plot forecasts
plt.figure(figsize=(12, 6))

# Historical data (last 100 points)
plt.plot(model_results.fittedvalues.index[-100:],
model_results.fittedvalues[-100:],
label='Fitted Values', color='red')

# Forecasts
forecast_index = pd.date_range(
start=model_results.fittedvalues.index[-1] + pd.Timedelta(days=1),
periods=steps, freq='D')
plt.plot(forecast_index, forecast, label='Forecast',
```

```
color='blue')
plt.fill_between(forecast_index,
conf_int.iloc[:, 0],
conf_int.iloc[:, 1],
alpha=0.3, color='blue')

plt.title('ARIMA Forecast')
plt.legend()
plt.show()

return forecast, conf_int

# Complete workflow
print("Step 1: Check Stationarity")
adf_result = adfuller(df['Views'])
print(f"ADF p-value: {adf_result[1]:.4f}")

if adf_result[1] > 0.05:
print("Series is non-stationary, applying first difference")
df['Views_diff'] = df['Views'].diff().dropna()
data_for_modeling = df['Views_diff'].dropna()
d = 1
else:
print("Series is stationary")
data_for_modeling = df['Views']
d = 0

print("\nStep 2: Identify Model Order")
if d == 0:
best_order = identify_arima_order(data_for_modeling)
else:
# For differenced data, identify ARMA order then add back integration
arma_order = identify_arima_order(data_for_modeling)
best_order = (arma_order[0], d, arma_order[2])

print(f"\nStep 3: Fit ARIMA{best_order} Model")
final_model = fit_arima_model(df['Views'], best_order)

print("\nStep 4: Generate Forecasts")
forecasts, conf_intervals = forecast_arima(final_model, steps=30)

# Model evaluation metrics
def evaluate_model(actual, fitted):
"""Calculate evaluation metrics"""
mse = np.mean((actual - fitted)**2)
rmse = np.sqrt(mse)
mae = np.mean(np.abs(actual - fitted))
mape = np.mean(np.abs((actual - fitted) / actual)) * 100
```

```python

print("\nModel Evaluation Metrics:")
print(f"MSE: {mse:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"MAE: {mae:.2f}")
print(f"MAPE: {mape:.2f}%")

# Evaluate on in-sample data
evaluate_model(df['Views'], final_model.fittedvalues)
```

Interpretation Guidelines:

- **Parameters:** Significant coefficients (p-value < 0.05)
- **Residuals:** Should be white noise (no autocorrelation)
- **Model Selection:** Lower AIC/BIC indicates better model
- **Forecast Intervals:** Wider intervals indicate more uncertainty

5 Seasonal ARIMA (SARIMA) Models

5.1 Theory: Seasonal ARIMA

Definition: SARIMA models extend ARIMA to handle seasonal patterns in time series data.

Mathematical Formulation: SARIMA$(p, d, q) \times (P, D, Q)_s$ model:

$$\phi(B)\Phi(B^s)(1 - B)^d(1 - B^s)^D Y_t = \theta(B)\Theta(B^s)\epsilon_t \tag{2.11}$$

Where:

- (p, d, q) = Non-seasonal ARIMA order
- (P, D, Q) = Seasonal ARIMA order
- s = Seasonal period (e.g., 12 for monthly data)
- $\Phi(B^s)$ = Seasonal AR polynomial
- $\Theta(B^s)$ = Seasonal MA polynomial

Identification Process:

1. Check for seasonal patterns in ACF/PACF

2. Apply seasonal differencing if needed
3. Identify seasonal and non-seasonal orders
4. Estimate parameters jointly

5.2 Practical: Building SARIMA Models

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import warnings
warnings.filterwarnings('ignore')

# Load seasonal data (e.g., monthly sales)
df = pd.read_csv('monthly_sales.csv', parse_dates=['Date'],
index_col='Date')
df = df.asfreq('MS')  # Monthly start frequency

def seasonal_analysis(data, period=12):
"""
Perform seasonal decomposition and analysis
"""
# Seasonal decomposition
decomposition = seasonal_decompose(data, model='multiplicative',
period=period)

# Plot components
fig, axes = plt.subplots(4, 1, figsize=(12, 10))

data.plot(ax=axes[0], title='Original Series')
decomposition.trend.plot(ax=axes[1], title='Trend')
decomposition.seasonal.plot(ax=axes[2], title='Seasonal')
decomposition.resid.plot(ax=axes[3], title='Residual')

plt.tight_layout()
plt.show()

# Check seasonal strength
seasonal_strength = 1 - (decomposition.resid.var() /
(decomposition.resid + decomposition.seasonal).var())
print(f"Seasonal Strength: {seasonal_strength:.3f}")

return decomposition
```

```python

def identify_sarima_order(data, seasonal_period=12, max_order=2):
"""
Identify SARIMA order using grid search
"""
# Define parameter ranges
p = d = q = range(0, max_order + 1)
P = D = Q = range(0, 2)

# Grid search
best_aic = np.inf
best_order = None
best_seasonal_order = None

for param in itertools.product(p, d, q):
for param_seasonal in itertools.product(P, D, Q):
try:
model = SARIMAX(data,
order=param,
seasonal_order=param_seasonal + (seasonal_period,),
enforce_stationarity=False,
enforce_invertibility=False)
results = model.fit(disp=False)

if results.aic < best_aic:
best_aic = results.aic
best_order = param
best_seasonal_order = param_seasonal + (seasonal_period,)
except:
continue

print(f"Best SARIMA order: {best_order} x {best_seasonal_order}")
print(f"Best AIC: {best_aic:.2f}")

return best_order, best_seasonal_order

def fit_sarima_model(data, order, seasonal_order):
"""
Fit SARIMA model with diagnostics
"""
model = SARIMAX(data,
order=order,
seasonal_order=seasonal_order,
enforce_stationarity=False,
enforce_invertibility=False)

results = model.fit()

```

```
print(results.summary())

# Diagnostic plots
results.plot_diagnostics(figsize=(15, 12))
plt.show()

return results

# Analysis workflow
print("Seasonal Time Series Analysis")
print("="*50)

# Step 1: Seasonal decomposition
decomposition = seasonal_analysis(df['Sales'])

# Step 2: Identify SARIMA order
order, seasonal_order = identify_sarima_order(df['Sales'])

# Step 3: Fit model
sarima_model = fit_sarima_model(df['Sales'], order, seasonal_order)

# Step 4: Forecast
forecast_steps = 24  # 2 years ahead
forecast = sarima_model.get_forecast(steps=forecast_steps)
forecast_mean = forecast.predicted_mean
forecast_conf_int = forecast.conf_int()

# Plot forecast
plt.figure(figsize=(14, 8))
plt.plot(df.index, df['Sales'], label='Historical')
plt.plot(sarima_model.fittedvalues.index, sarima_model.fittedvalues,
label='Fitted', alpha=0.7)
plt.plot(forecast_mean.index, forecast_mean, label='Forecast',
color='red')
plt.fill_between(forecast_conf_int.index,
forecast_conf_int.iloc[:, 0],
forecast_conf_int.iloc[:, 1],
alpha=0.3, color='red')
plt.title('SARIMA Forecast')
plt.legend()
plt.show()

# Seasonal evaluation
def evaluate_seasonal_model(actual, fitted, period=12):
"""
Evaluate seasonal forecasting performance
"""
# Overall metrics
```

```
rmse = np.sqrt(np.mean((actual - fitted)**2))
mae = np.mean(np.abs(actual - fitted))

# Seasonal metrics
seasonal_errors = []
for i in range(period):
seasonal_actual = actual[i::period]
seasonal_fitted = fitted[i::period]
if len(seasonal_actual) > 0 and len(seasonal_fitted) > 0:
seasonal_rmse = np.sqrt(np.mean((seasonal_actual - seasonal_fitted)**2))
seasonal_errors.append(seasonal_rmse)

print("Model Evaluation:")
print(f"Overall RMSE: {rmse:.2f}")
print(f"Overall MAE: {mae:.2f}")
print(f"Average Seasonal RMSE: {np.mean(seasonal_errors):.2f}")

evaluate_seasonal_model(df['Sales'], sarima_model.fittedvalues)
```

Excel Implementation:

```
// Seasonal modeling in Excel:
1. Create seasonal dummy variables
2. Use Data > Data Analysis > Regression with seasonal dummies
3. Create seasonal dummy variables for each month/quarter
4. NumXL > SARIMA Model (if available)
```

6 Transfer Function Models

6.1 Theory: Transfer Function Models

Definition: Transfer function models relate an output time series to one or more input time series, capturing the dynamic relationship between variables.

Mathematical Formulation:

$$Y_t = \frac{\omega(B)}{\delta(B)} B^b X_t + \frac{\theta(B)}{\phi(B)} \epsilon_t \tag{2.12}$$

Where:

- Y_t = Output series
- X_t = Input series
- $\omega(B), \delta(B)$ = Transfer function polynomials

- $\theta(B), \phi(B)$ = Noise model polynomials
- b = Pure delay parameter
- ϵ_t = White noise

Components:

1. **Transfer Function:** $\frac{\omega(B)}{\delta(B)}B^b$ describes how input affects output
2. **Noise Model:** $\frac{\theta(B)}{\phi(B)}\epsilon_t$ captures unexplained variation

Applications:

- Marketing: How advertising spend affects sales
- Economics: How interest rates affect inflation
- Digital: How content creation affects YouTube views

Model Building Steps:

1. **Prewhitening**: Transform input to white noise
2. **Cross-correlation:** Identify delay and transfer function structure
3. **Model Estimation:** Fit transfer function parameters
4. **Residual Analysis:** Check noise model adequacy

6.2 Practical: Building Transfer Function Models

6.2.1 Python Implementation (YouTube Views vs. Ad Spend)

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from scipy.stats import pearsonr
import warnings
warnings.filterwarnings('ignore')

# Load data with input (Ad_Spend) and output (Views)
df = pd.read_csv('youtube_ad_data.csv', parse_dates=['Date'],
```

```
index_col='Date')

def prewhiten_series(input_series, max_order=3):
"""
Prewhiten input series by fitting ARIMA model
"""
print("Prewhitening Input Series")
print("="*30)

# Find best ARIMA model for input
best_aic = np.inf
best_order = None

for p in range(max_order + 1):
for d in range(2):
for q in range(max_order + 1):
try:
model = ARIMA(input_series, order=(p, d, q))
results = model.fit()
if results.aic < best_aic:
best_aic = results.aic
best_order = (p, d, q)
except:
continue

print(f"Best order for input: ARIMA{best_order}")

# Fit model and get residuals (prewhitened series)
model = ARIMA(input_series, order=best_order)
results = model.fit()
prewhitened = results.resid

return prewhitened, results

def cross_correlation_analysis(input_clean, output_clean, max_lag=20):
"""
Calculate and plot cross-correlation function
"""
# Calculate cross-correlations
cross_corrs = []
lags = range(-max_lag, max_lag + 1)

for lag in lags:
if lag > 0:
corr, _ = pearsonr(input_clean[:-lag], output_clean[lag:])
else:
corr, _ = pearsonr(input_clean[-lag:], output_clean[:lag])
cross_corrs.append(corr)
```

```
# Plot cross-correlations
plt.figure(figsize=(12, 6))
plt.stem(lags, cross_corrs)
plt.axhline(y=0, color='black', linestyle='-', alpha=0.3)
plt.axhline(y=1.96/np.sqrt(len(input_clean)), color='red',
linestyle='--', alpha=0.5)
plt.axhline(y=-1.96/np.sqrt(len(input_clean)), color='red',
linestyle='--', alpha=0.5)
plt.title('Cross-Correlation Function')
plt.xlabel('Lag')
plt.ylabel('Cross-Correlation')
plt.grid(True, alpha=0.3)
plt.show()

# Identify significant lags
threshold = 1.96/np.sqrt(len(input_clean))
significant_lags = [lag for lag, corr in zip(lags, cross_corrs)
if abs(corr) > threshold]

print(f"Significant lags: {significant_lags}")

return cross_corrs, lags

def fit_transfer_function(output_series, input_series, delay=0,
transfer_order=(1,1), noise_order=(1,1)):
"""
Fit transfer function model using SARIMAX
"""
print(f"Fitting Transfer Function Model")
print(f"Delay: {delay}, Transfer Order: {transfer_order}, Noise Order: {
   noise_order}")
print("="*50)

# Prepare input with delay
if delay > 0:
input_delayed = input_series.shift(delay)
else:
input_delayed = input_series

# Align data
aligned_data = pd.concat([output_series, input_delayed], axis=1).dropna()
y = aligned_data.iloc[:, 0]
x = aligned_data.iloc[:, 1]

# Fit SARIMAX model (which can handle exogenous variables)
model = SARIMAX(y,
exog=x,
```

```python
order=noise_order + (0,),  # (p,d,q) for noise model
enforce_stationarity=False,
enforce_invertibility=False)

results = model.fit()

print(results.summary())

# Diagnostics
results.plot_diagnostics(figsize=(15, 12))
plt.show()

return results

def transfer_function_forecast(model_results, future_input, steps=30):
"""
Generate forecasts using transfer function model
"""
print("Transfer Function Forecasting")
print("="*30)

# Generate forecasts
forecast = model_results.get_forecast(steps=steps, exog=future_input)
forecast_mean = forecast.predicted_mean
forecast_ci = forecast.conf_int()

# Plot results
fig, axes = plt.subplots(2, 1, figsize=(15, 12))

# Historical and forecast for output
axes[0].plot(model_results.data.endog[-60:], label='Historical Output',
color='blue')
axes[0].plot(model_results.fittedvalues[-60:], label='Fitted',
color='red', alpha=0.7)

forecast_index = pd.date_range(
start=model_results.data.endog.index[-1] + pd.Timedelta(days=1),
periods=steps, freq='D')
axes[0].plot(forecast_index, forecast_mean, label='Forecast',
color='green', linewidth=2)
axes[0].fill_between(forecast_index,
forecast_ci.iloc[:, 0],
forecast_ci.iloc[:, 1],
alpha=0.3, color='green')
axes[0].set_title('Output Forecast')
axes[0].legend()

# Input series
```

```
axes[1].plot(model_results.data.exog[-60:], label='Historical Input',
color='orange')
axes[1].plot(forecast_index, future_input, label='Future Input',
color='purple', linewidth=2)
axes[1].set_title('Input Series')
axes[1].legend()

plt.tight_layout()
plt.show()

return forecast_mean, forecast_ci

# Complete Transfer Function Workflow
print("Transfer Function Modeling: YouTube Views vs Ad Spend")
print("="*60)

print("\nStep 1: Data Exploration")
# Plot input and output series
fig, axes = plt.subplots(2, 1, figsize=(15, 8))
axes[0].plot(df.index, df['Ad_Spend'], label='Ad Spend')
axes[0].set_title('Input Series: Ad Spend')
axes[0].legend()

axes[1].plot(df.index, df['Views'], label='Views')
axes[1].set_title('Output Series: Views')
axes[1].legend()
plt.tight_layout()
plt.show()

# Check correlation
correlation, p_value = pearsonr(df['Ad_Spend'], df['Views'])
print(f"Overall correlation: {correlation:.4f} (p-value: {p_value:.4f})")

print("\nStep 2: Prewhitening Input Series")
prewhitened_input, input_model = prewhiten_series(df['Ad_Spend'])

print("\nStep 3: Cross-Correlation Analysis")
cross_corrs, lags = cross_correlation_analysis(prewhitened_input, df['Views'])

print("\nStep 4: Fit Transfer Function Model")
# Based on cross-correlation, determine delay and orders
# For demonstration, using delay=1 and simple orders
tf_model = fit_transfer_function(df['Views'], df['Ad_Spend'],
delay=1,
transfer_order=(1,1),
noise_order=(1,0))

print("\nStep 5: Forecasting")
```

```
# Create future input scenario (e.g., constant ad spend)
future_ad_spend = pd.Series([df['Ad_Spend'].mean()] * 30)
forecast_mean, forecast_ci = transfer_function_forecast(tf_model,
future_ad_spend)

# Model evaluation
def evaluate_transfer_function(actual, fitted):
"""Evaluate transfer function model"""
mse = np.mean((actual - fitted)**2)
rmse = np.sqrt(mse)
mae = np.mean(np.abs(actual - fitted))
mape = np.mean(np.abs((actual - fitted) / actual)) * 100

print("\nTransfer Function Model Evaluation:")
print(f"MSE: {mse:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"MAE: {mae:.2f}")
print(f"MAPE: {mape:.2f}%")

evaluate_transfer_function(df['Views'], tf_model.fittedvalues)

# Scenario analysis
def scenario_analysis(model, base_input, scenarios):
"""
Analyze different input scenarios
"""
print("\nScenario Analysis")
print("="*20)

results = {}
for scenario_name, multiplier in scenarios.items():
scenario_input = base_input * multiplier
forecast = model.get_forecast(steps=len(scenario_input),
exog=scenario_input)
results[scenario_name] = {
  'forecast': forecast.predicted_mean,
  'total_impact': forecast.predicted_mean.sum()
}
print(f"{scenario_name}: Total predicted views = {results[scenario_name]['
   total_impact']:.0f}")

return results

# Run scenario analysis
base_spend = pd.Series([df['Ad_Spend'].mean()] * 30)
scenarios = {
  'Low Spend (50%)': 0.5,
  'Current Spend (100%)': 1.0,
```

```
    'High Spend (150%)': 1.5,
    'Maximum Spend (200%)': 2.0
}

scenario_results = scenario_analysis(tf_model, base_spend, scenarios)
```

Excel Implementation:

```
// Cross-correlation in Excel:
1. Use CORREL function with shifted data
2. =CORREL(A2:A100, B3:B101) for lag 1
3. Plot correlations vs lags
4. Use Regression with lagged inputs for transfer function
```

7 Comprehensive Project: Indian Stock Market Analysis

7.1 Final Project: NSE MRF Stock Prediction

```
# Complete end-to-end time series analysis
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import datetime

# Load NSE MRF data
df = pd.read_csv('MRF_stock_data.csv', parse_dates=['Date'],
index_col='Date')

print("="*60)
print("COMPREHENSIVE TIME SERIES ANALYSIS: MRF STOCK")
print("="*60)

# 1. Exploratory Data Analysis
print("\n1. EXPLORATORY DATA ANALYSIS")
print("-" * 40)
print(f"Data Range: {df.index.min()} to {df.index.max()}")
print(f"Total Observations: {len(df)}")
print(f"Missing Values: {df.isnull().sum().sum()}")

# 2. Decomposition
print("\n2. TIME SERIES DECOMPOSITION")
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(df['Close'],
model='multiplicative',
period=252)  # Annual
decomposition.plot()
```

```python
plt.show()

# 3. Stationarity Tests
print("\n3. STATIONARITY ANALYSIS")
from statsmodels.tsa.stattools import adfuller
adf_result = adfuller(df['Close'])
print(f"ADF Test p-value: {adf_result[1]:.6f}")

# 4. ARIMA Modeling
print("\n4. ARIMA MODELING")
# Auto ARIMA approach
from statsmodels.tsa.arima.model import ARIMA
# ... (use previous code)

# 5. SARIMA for Seasonality
print("\n5. SEASONAL MODELING")
# ... (use previous SARIMA code)

# 6. Transfer Function with Market Index
print("\n6. TRANSFER FUNCTION WITH NIFTY INDEX")
# Load NIFTY data and model relationship
# ... (use previous transfer function code)

# 7. Final Forecast
print("\n7. FINAL FORECAST AND TRADING STRATEGY")
# Combine models for robust forecast
# ... (implement ensemble approach)

print("\n" + "="*60)
print("ANALYSIS COMPLETE")
print("="*60)
```

8 Resources & Next Steps

8.1 Recommended Books

- *Forecasting: Principles and Practice* by Hyndman & Athanasopoulos
- *Time Series Analysis* by Hamilton
- *Applied Time Series Analysis* by Cryer & Chan

8.2 Python Libraries

- `statsmodels`: Statistical modeling

- `pmdarima`: Auto ARIMA
- `prophet`: Facebook's forecasting tool
- `arch`: GARCH models for volatility

8.3 Practice Datasets

- NSE historical data
- RBI economic indicators
- Weather data from IMD
- COVID-19 statistics

8.4 Advanced Topics (Next Level)

- GARCH models for volatility
- Vector Autoregression (VAR)
- State Space Models
- Machine Learning for Time Series