Practical 1: Time Series Visualization and Decomposition

This practical demonstrates how to create synthetic time series data, visualize it, and decompose it into its constituent components using Python.

1.1 Objective

- Generate synthetic time series data with known patterns
- Visualize time series data effectively
- Decompose time series into trend, seasonal, and residual components
- Understand the statsmodels seasonal_decompose function

1.2 Theory: Time Series Visualization

Time series visualization is crucial for exploratory data analysis. Effective visualization helps identify:
- Overall trends and patterns
- Seasonal effects
- Outliers and anomalies
- Data quality issues
- Structural breaks

Python libraries like Matplotlib and Seaborn provide powerful tools for creating informative time series plots.

1.3 Theory: Seasonal Decomposition

Seasonal decomposition is a statistical technique that separates a time series into its constituent components. The statsmodels library provides the seasonal_decompose function which uses moving averages to extract:

1. Trend: Computed using a centered moving average
2. Seasonal: Calculated as the average deviation from the trend for each period
3. Residual: What remains after removing trend and seasonal components

The decomposition process:
1. Calculate the trend using moving averages
2. Detrend the data by subtracting the trend
3. Calculate seasonal indices by averaging detrended values for each season

4. Calculate residuals as the remainder after removing trend and seasonal components

1.4 Code Implementation

 # Synthetic Data Creation

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose

# Create synthetic monthly sales data
np.random.seed(42)
date_rng = pd.date_range(start='2018-01-01', end='2022-12-01', freq='MS')
sales = 200 + np.sin(np.arange(len(date_rng)) * (2 * np.pi / 12)) * 20 + np.random.normal(0,
10, len(date_rng))

# Save to CSV
df = pd.DataFrame({'Date': date_rng, 'Sales': sales})
df.to_csv("synthetic_sales.csv", index=False)
print("CSV created: synthetic_sales.csv")
```

 # Time Series Visualization

```
# Load dataset
df = pd.read_csv("synthetic_sales.csv", parse_dates=['Date'])
df.set_index('Date', inplace=True)

# Plot time series
plt.figure(figsize=(12,5))
plt.plot(df['Sales'], label="Sales")
plt.title("Time Series Visualization - Sales")
plt.xlabel("Date")
plt.ylabel("Sales")
plt.legend()
plt.show()
```

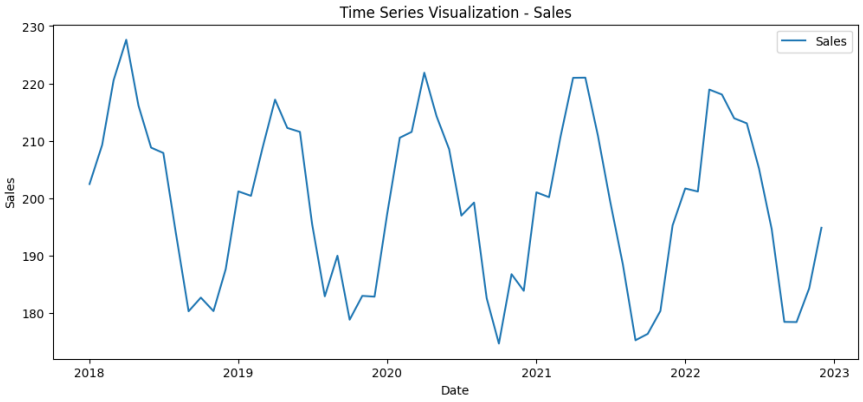

Fig 1.1 Time Series Visualisation

 # Time Series Decomposition

```
decomposition = seasonal_decompose(df['Sales'], model='additive', period=12)
decomposition.plot()
plt.show()
```

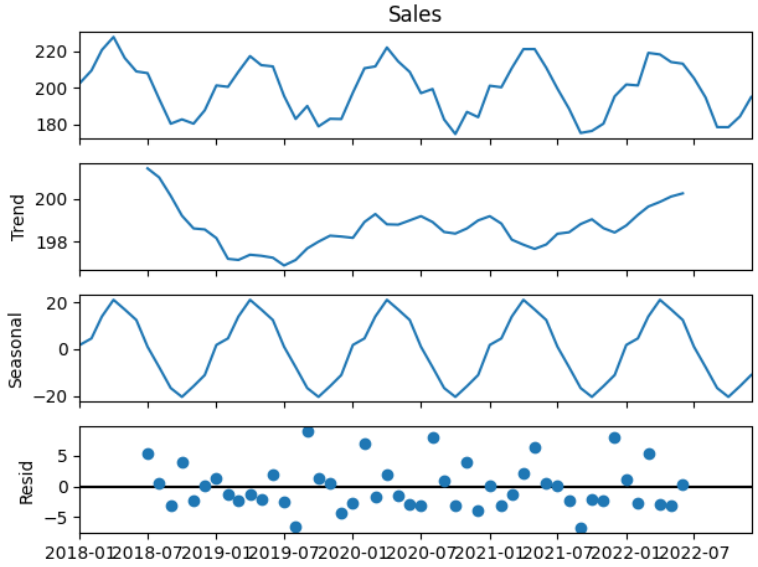

Fig 1.2 Trend , Seasonal , Resid

1.5 Expected Results

The decomposition should reveal:
- Trend: A relatively stable long-term pattern around 200
- Seasonal: Clear 12-month cyclical pattern with amplitude of ±20
- Residual: Random fluctuations representing noise in the data

This decomposition helps identify the underlying structure of the time series and can inform forecasting strategies.

Practical 2: Forecasting with Holt-Winters Exponential Smoothing

This practical demonstrates time series forecasting using the Holt-Winters exponential smoothing method, which is particularly effective for data with both trend and seasonal patterns.

2.1 Objective

- Understand the Holt-Winters exponential smoothing algorithm
- Implement forecasting using ExponentialSmoothing from statsmodels
- Generate multi-step ahead forecasts
- Evaluate forecast accuracy and visualize results

2.2 Theory: Exponential Smoothing

Exponential smoothing is a time series forecasting technique that applies exponentially decreasing weights to past observations. The weights decrease exponentially as observations get older, giving more importance to recent data points.

Simple Exponential Smoothing formula:
$St = \alpha Xt + (1-\alpha)St-1$

Where:
- St is the smoothed value at time t
- α is the smoothing parameter ($0 < \alpha < 1$)
- Xt is the actual observation at time t

2.3 Theory: Holt-Winters (Triple Exponential Smoothing)

The Holt-Winters method extends simple exponential smoothing to handle time series with both trend and seasonal patterns. It uses three smoothing equations:

1. Level equation: $Lt = \alpha(Yt/St-m) + (1-\alpha)(Lt-1 + bt-1)$
2. Trend equation: $bt = \beta(Lt - Lt-1) + (1-\beta)bt-1$
3. Seasonal equation: $St = \gamma(Yt/Lt) + (1-\gamma)St-m$

Where:
- Lt = level (smoothed observation)
- bt = trend component
- St = seasonal component

- α, β, γ = smoothing parameters
- m = seasonal period
- Yt = actual observation at time t

Forecast equation: $\hat{Y}t+h = (Lt + hbt)St-m+h$

The method comes in two variants:
1. Additive: Y(t) = Level + Trend + Seasonal + Error
2. Multiplicative: Y(t) = Level × Trend × Seasonal × Error

2.4 Algorithm Steps

1. Initialize level, trend, and seasonal components
2. For each time period:
   a. Update level component using current observation and seasonal factor
   b. Update trend component based on level changes
   c. Update seasonal component using current observation and level
3. Generate forecasts using the forecast equation
4. Evaluate forecast performance using appropriate metrics

2.5 Code Implementation

 # Import Libraries and Load Data

```
import pandas as pd
from matplotlib import pyplot as plt
df = pd.read_csv('synthetic_sales.csv')
```

 # Holt-Winters Exponential Smoothing

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Apply forecasting model (Holt-Winters Exponential Smoothing)
model = ExponentialSmoothing(df['Sales'], trend="add", seasonal="add",
seasonal_periods=12)
fit = model.fit()

# Forecast next 12 months
forecast = fit.forecast(12)

# Plot actual vs forecast
plt.figure(figsize=(12,5))
plt.plot(df['Sales'], label="Actual")
plt.plot(forecast, label="Forecast", color='red')
plt.title("Sales Forecasting with Holt-Winters")
plt.xlabel("Date")
```

```
plt.ylabel("Sales")
plt.legend()
plt.show()
```

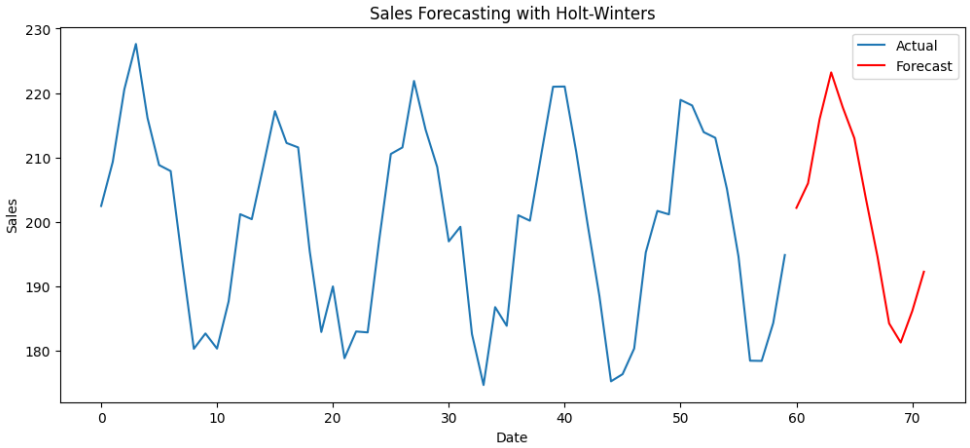

Fig 2 .1 Sales Forcasting with Holt -Winters

Practical 3: Simple Moving Average and AutoRegressive Models

This practical explores two fundamental time series modeling approaches: Simple Moving Average (SMA) for trend smoothing and AutoRegressive (AR) models for capturing temporal dependencies.

3.1 Objective

- Understand and implement Simple Moving Average for trend detection
- Learn AutoRegressive model theory and applications
- Compare SMA smoothing with original data
- Build and evaluate AR models for forecasting
- Analyze model performance and interpretation

3.2 Theory: Simple Moving Average (SMA)

Simple Moving Average (SMA) is a fundamental time series smoothing technique that calculates the arithmetic mean of a fixed number of consecutive observations. It's used to:
- Identify underlying trends
- Reduce noise in data
- Provide baseline forecasts
- Detect trend changes

Mathematical Formula:
$SMA(t) = (X(t) + X(t-1) + X(t-2) + ... + X(t-n+1)) / n$

Where:
- SMA(t) = Simple Moving Average at time t
- X(t) = Observation at time t
- n = Window size (number of periods)

Key Characteristics:
1. Equal weight to all observations in the window
2. Window size determines smoothness vs responsiveness trade-off
3. Larger windows = smoother but less responsive
4. Smaller windows = more responsive but noisier

Applications:
- Trend identification in financial markets
- Signal processing and noise reduction
- Quality control in manufacturing

- Baseline forecasting method

3.3 Theory: AutoRegressive (AR) Models

AutoRegressive (AR) models predict future values based on a linear combination of past values from the same time series. The model assumes that current values depend on previous observations.

Mathematical Representation:
AR(p): $X(t) = c + \varphi_1 X(t-1) + \varphi_2 X(t-2) + ... + \varphi_p X(t-p) + \varepsilon(t)$

Where:
- $X(t)$ = Current observation
- c = Constant term
- φ_i = Autoregressive coefficients ($i = 1, 2, ..., p$)
- p = Order of the autoregressive model (number of lags)
- $\varepsilon(t)$ = White noise error term

Key Components:
1. Order (p): Number of previous time points used in the model
2. Coefficients (φ): Measure the influence of each lagged value
3. Error term (ε): Random component not explained by the model

Model Selection:
- Use Autocorrelation Function (ACF) and Partial ACF (PACF) plots
- Information criteria (AIC, BIC) for optimal order selection
- Cross-validation for performance evaluation

Stationarity Requirement:
AR models require stationary data where:
- Constant mean over time
- Constant variance over time
- Covariance depends only on the lag, not time

Applications:
- Economic forecasting
- Stock price prediction
- Weather forecasting
- Signal processing

3.4 Autocorrelation and Model Diagnostics

Autocorrelation Function (ACF):
Measures the correlation between a time series and its lagged values. For AR(p) models, ACF decays gradually.

Partial Autocorrelation Function (PACF):
Measures the correlation between X(t) and X(t-k) after removing the linear dependence on X(t-1), X(t-2), ..., X(t-k+1). For AR(p) models, PACF cuts off after lag p.

Model Selection Rules:
- AR(p): PACF cuts off after lag p, ACF decays exponentially
- MA(q): ACF cuts off after lag q, PACF decays exponentially
- ARMA(p,q): Both ACF and PACF decay exponentially

Diagnostic Tests:
1. Ljung-Box test for residual autocorrelation
2. Jarque-Bera test for normality of residuals
3. Heteroskedasticity tests for constant variance
4. Stability tests for coefficient significance

3.5 Code Implementation

 # Import Libraries and Load Data

```
import pandas as pd
from matplotlib import pyplot as plt
df = pd.read_csv('synthetic_sales.csv')
```

 # Simple Moving Average Implementation

```
from statsmodels.tsa.ar_model import AutoReg
# --- Simple Moving Average (SMA) ---
df['SMA_6'] = df['Sales'].rolling(window=6).mean()
plt.figure(figsize=(12,5))
plt.plot(df['Sales'], label="Original")
plt.plot(df['SMA_6'], label="6-Month SMA", color='red')
plt.title("Simple Moving Average (SMA)")
plt.xlabel("Date")
plt.ylabel("Sales")
plt.legend()
plt.show()
```

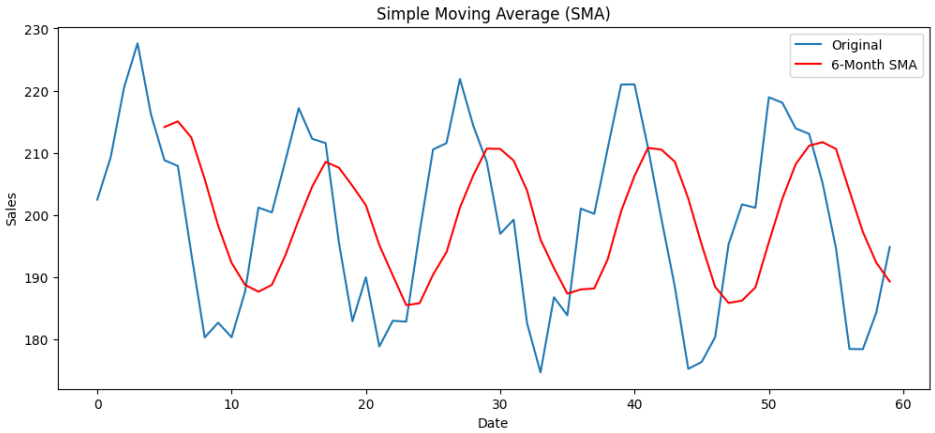

Fig 3.1 Simple Moving Average (SMA)

 # AutoRegressive Model Implementation

```
train = df['Sales'][:-12]
test = df['Sales'][-12:]
ar_model = AutoReg(train, lags=6).fit()
pred = ar_model.predict(start=len(train), end=len(train)+len(test)-1, dynamic=False)
plt.figure(figsize=(12,5))
plt.plot(train.index, train, label="Train")
plt.plot(test.index, test, label="Test", color='blue')
plt.plot(test.index, pred, label="AR Predictions", color='red')
plt.title("AutoRegressive (AR) Model")
plt.xlabel("Date")
plt.ylabel("Sales")
plt.legend()
plt.show()
```

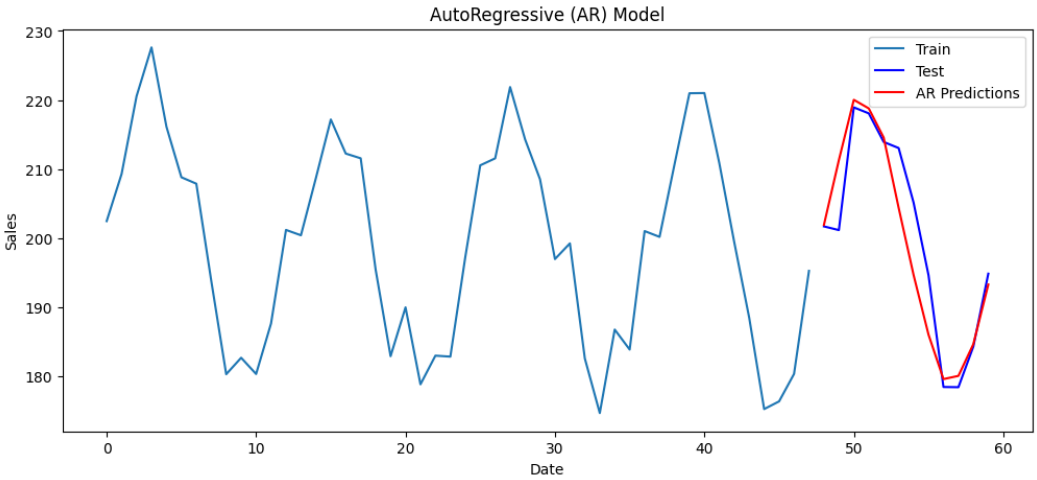

Fig 3.2 Autoregressive (AR) Model