Topic: Characteristics of Time-Series Data

Prof. Victor M. Preciado

Contents

1 Characteristics of Time-Series Data I: Trends, Seasonality, and Noise

The primary distinction between time series data and other types of data lies in the **temporal order of observations**. In time series data, the temporal order is fundamental, and any *permutation of the data points would disrupt the inherent structure* of the data. In real-world applications, time series data present the following components:

- The **trend** component refers to the long-term movement in the data, which can be upward, downward, or stable. A trend represents a systematic change in the mean over time, indicating that the average value of the time series is not constant. Detrending, which involves subtracting a fitted trend function from the original time series, is a necessary step before analyzing the stationary properties of a time series, such as its variance and autocorrelation.
- The **seasonal** component refers to regular patterns in data that repeat over a specific period, such as daily, monthly, or yearly. Mathematically, seasonality can be modeled by identifying periodic components within the time series. Seasonal adjustment, which involves eliminating the seasonal fluctuations from the data, is also a necessary step before analyzing stationary properties of a time series.
- The **noise** component in a times series accounts the random variability in the data that cannot be explained by a deterministic model. Noise is the residual component after accounting for seasonality, trend, and other systematic patterns. It is typically modeled as a *white noise process*[1]. The variance of the noise component represents the degree of unpredictability in the time series, and its distribution can affect the reliability of forecasts and the detection of true underlying patterns.

1.1 Python Example: Seasonality, Trend and Noise

To better understand the characteristics of time series data, let us generate a synthetic time series and plot it using Python. This Python code generates and plots a synthetic time series that combines a linear trend, a seasonal component, and random noise, which is useful for illustrating how different components—trend, seasonality, and noise—contribute to the overall behavior of a time series.

This piece of Python code performs the following tasks (these pieces of code can be found in the following GitHub Repository.)

1. **Import Necessary Libraries:** `import numpy as np` imports the `numpy` library with the alias `np`, which provides support for arrays, mathematical operations, and random number generation; `import matplotlib.pyplot as plt` imports the `matplotlib.pyplot` module with the alias `plt`, which is commonly used for creating plots and visualizing data in Python. This setup is typically used in data analysis or machine learning tasks to handle numerical data and generate visualizations.

```
import numpy as np
import matplotlib.pyplot as plt
```

[1] A white noise is a stochastic process whose values at different times are i.i.d. with a mean of zero and constant variance. Notice that white noise has no autocorrelation, i.e., $\rho(h) = 0$ for all non-zero lags h.

2. **Random Seed:** The code begins by setting a random seed using `np.random.seed(0)` to ensure that the random numbers generated (in this case, the noise) are the same every time the code is run, making the results reproducible.

```
# Set the random seed for reproducibility
np.random.seed(0)
```

3. **Time Index Creation:** It creates a time index array `time` that ranges from 0 to 99 using `np.arange(100)`, representing 100 time points.

```
# Create a time index from 0 to 99
time = np.arange(100)
```

4. **Trend Generation:** The code generates a linear trend component by multiplying the time index by 0.1. This creates a steadily increasing trend over time.

```
# Generate a linear trend
trend = 0.1 * time
```

5. **Seasonal Component:** It then creates a seasonal component using a sine function with a period of 20. The sine function oscillates between -10 and 10, introducing regular, periodic fluctuations into the time series.

```
# Generate a seasonal component with a period of 20
seasonal = 10 * np.sin(2 * np.pi * time / 20)
```

6. **Noise Generation:** The code generates random noise from a normal distribution with a mean of 0 and a standard deviation of 2. This noise is added to the time series to introduce randomness that cannot be explained by the trend or seasonality.

```
# Generate some random noise
noise = np.random.normal(scale=2, size=100)
```

7. **Time Series Combination:** The final time series `series` is constructed by adding together the linear trend, the seasonal component, and the noise.

```
# Combine these components to form the time series
series = trend + seasonal + noise
```

8. **Plotting the Time Series:** The code then plots the time series, the trend, and the seasonal component, shown in the Fig. 8:

```
# Plot the time series
plt.figure(figsize=(10, 6))
plt.plot(time, series, label='Time Series')
plt.plot(time, trend, '--', label='Trend')
plt.plot(time, seasonal, '--', label='Seasonality')
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('Synthetic Time-Series with Trend, Seasonality, and Noise')
plt.legend()
plt.show()
```

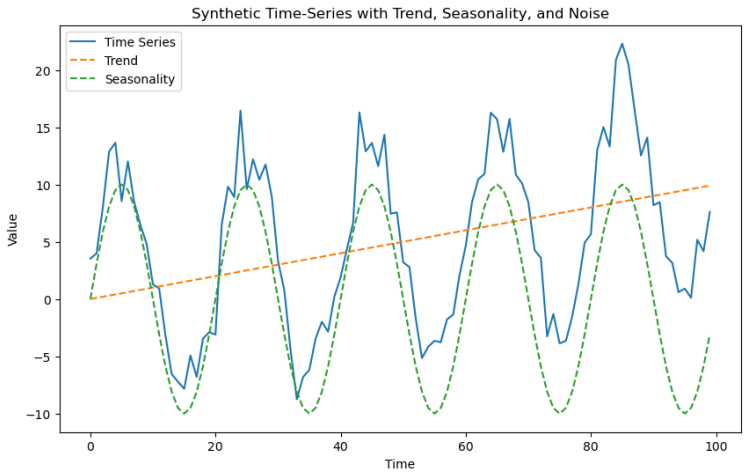

Figure 1: Synthetic Time-Series with Trend, Seasonality, and Noise

1.2 Extracting Seasonality, Trend, and Noise from a Time Series

Extracting seasonality, trend, and noise from a given time series is a fundamental task in time series analysis, often referred to as **time series decomposition**. The objective is to separate the observed time series into three main components:

- **Trend**: The long-term progression of the series, which may be upward, downward, or stable.
- **Seasonality**: The repeating patterns or cycles of behavior at regular intervals (e.g., yearly, quarterly).
- **Noise**: The residual component representing random variability or irregular fluctuations not explained by the trend or seasonality.

One of the most popular methods to separate these components is the the *classical decomposition*. The classical decomposition assumes that a time series Y_k can be expressed as the sum of its components in an *additive model*:

$$Y_k = \texttt{Trend}(k) + \texttt{Seasonal}(k) + \epsilon(k)$$

Here, $\texttt{Trend}(k)$ represents the trend component, $\texttt{Seasonal}(k)$ the seasonal component, and $\epsilon(k)$ the noise (also called *residuals*)[2]. Decomposing a time series into trend, seasonality, and noise is important in practice for several key reasons:

- **Improving Forecasting Accuracy:** Accurate forecasting often relies on the ability to model each component of the time series separately. By removing the effects of trend and seasonality, the remaining noise can be analyzed for potential predictive patterns. Additionally, trend and seasonal patterns can be extrapolated into the future, allowing for more precise forecasts that account for these systematic components.

[2]In some cases, such as with financial data, a multiplicative model, $Y_k = \texttt{Trend}(k) \times \texttt{Seasonal}(k) \times \epsilon(k)$, might be more appropriate. This multiplicative model can be converted to an additive form via a logarithmic transformation, as follows: $\log Y_k = \log \texttt{Trend}(k) + \log \texttt{Seasonal}(k) + \log \epsilon(k)$. This transformation simplifies the analysis and allows the use of additive methods on data that exhibits multiplicative behavior.

- **Modeling and Analysis:** Many statistical models and machine learning algorithms assume that the data is *WSS*, meaning that its statistical properties, such as mean and variance, do not change over time. Decomposing the time series allows for the removal of non-stationary components, such as trend and seasonality, making the data more suitable for analysis and improving the performance of these models.
- **Detecting Anomalies:** Furthermore, decomposing a time series makes it easier to detect anomalies or outliers. By isolating the noise component, which should ideally be random and normally distributed, deviations from this expected behavior can be identified as potential anomalies. This is particularly useful in applications such as fraud detection, quality control, and monitoring systems.

Several advanced algorithms have been developed for seasonal, trend, and noise decomposition, each tailored to different types of data and requirements. These include:

- **X-11:** A popular method developed by the U.S. Census Bureau for decomposing monthly or quarterly economic time series into trend, seasonal, and irregular components. It handles both additive and multiplicative decomposition models and is implemented in Python via the `statsmodels` library as part of the `seasonal_decompose` method.
- **STL (Seasonal and Trend decomposition using Loess):** A versatile decomposition technique that iteratively separate the trend and seasonal components. It is highly flexible in handling various seasonal patterns and is available in Python through the `statsmodels` library's `STL` function.
- **Prophet:** A time series forecasting model developed by Meta that automatically decomposes time series data into trend, seasonal, and holiday effects. It is particularly suited for data with strong seasonal effects and allows users to specify custom seasonalities. In Python, Prophet is implemented in the `prophet` package, and the decomposition is accessible via the `Prophet` class's `predict` method, which returns estimates of the components.

1.3 Python Example: Classical Decomposition

In this programming example, we perform a classical decomposition of a time series into its trend, seasonal, and noise components using the STL algorithm implemented in the `statsmodels.tsa.seasonal` Python library. Here is a step-by-step explanation of what the code does (these pieces of code can be found in the following GitHub Repository):

1. **Import STL Decomposition:** Import the `STL` class from the `statsmodels.tsa.seasonal` module.

```
# Import STL function
from statsmodels.tsa.seasonal import STL
```

2. **Initialize STL Decomposition and Fit the STL Model:** Create an instance of the `STL` class, passing the time series data and the period of the seasonal component. The period parameter specifies the length of the seasonal cycle (e.g., 20 time points). The `fit()` method performs the decomposition. This step separates the time series into three components: trend, seasonal, and residual.

```python
# Perform STL decomposition of the time series
stl = STL(series, period=20)
result = stl.fit()
```

3. **Plot the Decomposed Components:** Use the `plot()` method on the result object to visualize the decomposed components, and display the plot with `plt.show()`. We also include a subplot where we can better observe how the values repeat over each period (in this case, $T = 20$). This plot is especially useful for understanding how cyclical behaviors evolve across different seasons.

```python
# Plot the decomposed components
result.plot()
plt.show()
```

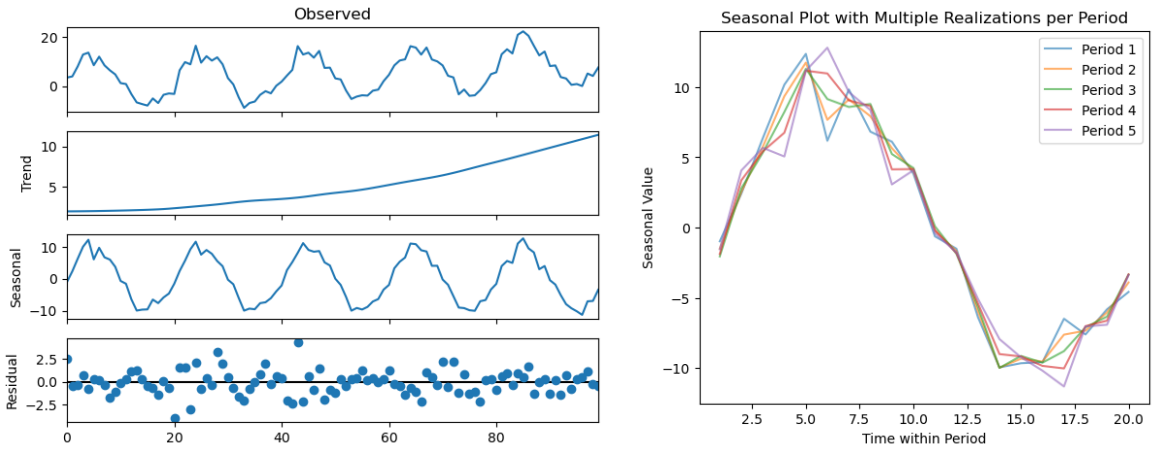

Figure 2: (Left) Decomposition of a time series into a Trend, Seasonal, and Noise (Residual) components. (Right) Cyclical behavior across different seasons.

2 Characteristics of Time-Series Data II: Temporal Dependencies

Apart from seasonality, trends and noise, time series data present temporal dependencies, since each observation typically dependent on previous observations. Temporal dependency introduces temporal autocorrelation in the data. Identifying the presence of autocorrelation in a time series is essential for both selecting the appropriate model and validating the assumptions that underpin the chosen approach. When autocorrelation is detected, it signals that simple models that assume uncorrelated errors are inadequate, thereby necessitating the use of more sophisticated models that account for these dependencies. By recognizing and addressing autocorrelation, analysts can enhance the accuracy and reliability of their models, leading to more robust forecasts and insights.

The analysis of autocorrelation is most effectively conducted on the residual (or noise) component of a time series, rather than on the original data, which includes trend and seasonal elements that often follow predictable patterns. Evaluating autocorrelation directly on the original time series can be misleading, as the observed correlations may merely reflect the inherent structure of the trend or seasonal components, rather than uncovering any genuine autocorrelation within the noise. By focusing on the residuals, the analysis aims to determine whether any dependencies

remain after the trend and seasonality have been accounted for, thereby offering a more accurate and insightful assessment of the underlying stochastic processes.

2.1 Detecting Autocorrelation

Below, we present several key techniques for analyzing the presence and significance of autocorrelation in time series data.

- **Visual Inspection:** Start by plotting the time series data. Clear patterns, such as trends or cycles, can indicate the presence of autocorrelation, as these patterns often reflect underlying dependencies between observations.
- **Autocorrelation Function (ACF) Plot:** Compute and plot the sample autocorrelation function (ACF), which displays the autocorrelation coefficients $\widehat{\rho}(h)$ at various lags h. In an ACF plot, significant autocorrelation is typically observed as spikes that extend beyond the confidence bounds, usually set at $\pm 2/\sqrt{T}$, where T denotes the length of the series.
- **Statistical Tests:** We can formally evaluate the presence of autocorrelation using hypothesis testing such as the *Ljung-Box test*. A small p-value (typically less than 0.05) from this test suggests that significant autocorrelation is present in the time series.

By combining these methods, you can effectively determine whether a real time series exhibits autocorrelation and to what extent, guiding further analysis or model selection.

2.2 Python Example: Detecting Autocorrelation

Below is the Python code that illustrates how to use Visual Inspection, the Autocorrelation Function (ACF) plot, and statistical tests to detect autocorrelation in the noise (residual) component obtained from the STL decomposition.

Here is a step-by-step explanation of the code (these pieces of code can be found in the following GitHub Repository):

- **Import Necessary Functions:** The `plot_acf` function is used to plot the Autocorrelation Function (ACF) of a time series, which helps in identifying the dependencies between observations at different lags. The `acorr_ljungbox` function performs the **Ljung-Box test** to check whether any group of autocorrelations in a time series is different from zero. The `probplot` function generates a **probability plot**, which compares the distribution of a dataset to a theoretical distribution (such as the normal distribution) to visually assess how closely the data follows the specified distribution. These functions are typically used together in time series analysis to check for autocorrelation, perform diagnostic tests, and assess the distribution of residuals or other datasets.

```
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.stats.diagnostic import acorr_ljungbox
from scipy.stats import probplot
```

- **Residual Extraction:** The residuals are extracted using `result.resid`, which represent the variability in the data that cannot be explained by the trend or seasonal components.

```
# Assuming 'result' contains the STL decomposition
# Extract the residual (noise) component
residuals = result.resid
```

- **Visual Inspection:** The residuals are then plotted over time to visually inspect for any obvious patterns, trends, or cycles. If the residuals show any systematic patterns, this could indicate that the decomposition has not fully captured the structure of the time series, and some autocorrelation may still be present.

```
# 1. Visual Inspection of Residuals
plt.figure(figsize=(7, 6))
plt.plot(residuals, label='Residuals (Noise)')
plt.title('Residuals from STL Decomposition')
plt.xlabel('Time')
plt.ylabel('Residuals')
plt.legend()
plt.show()
```

- **Autocorrelation Function (ACF) Plot:** The Autocorrelation Function (ACF) plot is generated using the `plot_acf` function. This plot displays the numerically estimated autocorrelation coefficients $\widehat{\rho}(h)$ at different lags h.
  - The vertical lines (spikes) in the ACF plot represent the autocorrelation at each lag.
  - The shaded area around the horizontal axis reflects the confidence interval for the autocorrelations.
  - If a spike extends beyond the shaded area, it indicates that the autocorrelation at that lag is statistically significant, meaning it is unlikely to have occurred by chance. Significant spikes suggest the presence of autocorrelation in the residuals at those specific lags. In this case, there is a significant spike at $h = 20$, corresponding to the period of the seasonal component. This suggests that the periodic behavior of the time series has not been fully captured by the seasonal component obtained from the STL decomposition.

```
# 2. Autocorrelation Function (ACF) Plot
plt.figure(figsize=(7, 6))
plot_acf(residuals, lags=40, alpha=0.05)
plt.title('ACF of Residuals')
plt.show()
```

- **Ljung-Box Test:** The Ljung-Box test is applied to the residuals to statistically assess the presence of autocorrelation. This test is performed using the `acorr_ljungbox` function. The Ljung-Box test evaluates the null hypothesis that the residuals are independently distributed (i.e., no autocorrelation) against the alternative hypothesis that autocorrelation is present. In this case, the p-value is 7×10^{-6}, which is much smaller than 0.05. Therefore, the test strongly suggests rejecting the null hypothesis, indicating the presence of significant autocorrelation. This result further supports the conclusion that the STL algorithm did not fully succeed in removing the trend and seasonality from the data.

```
# 3. Statistical Tests: Ljung-Box test for autocorrelation
# Perform the Ljung-Box test on the residuals
ljung_box_results = acorr_ljungbox(residuals, lags=[20], return_df=True)

```

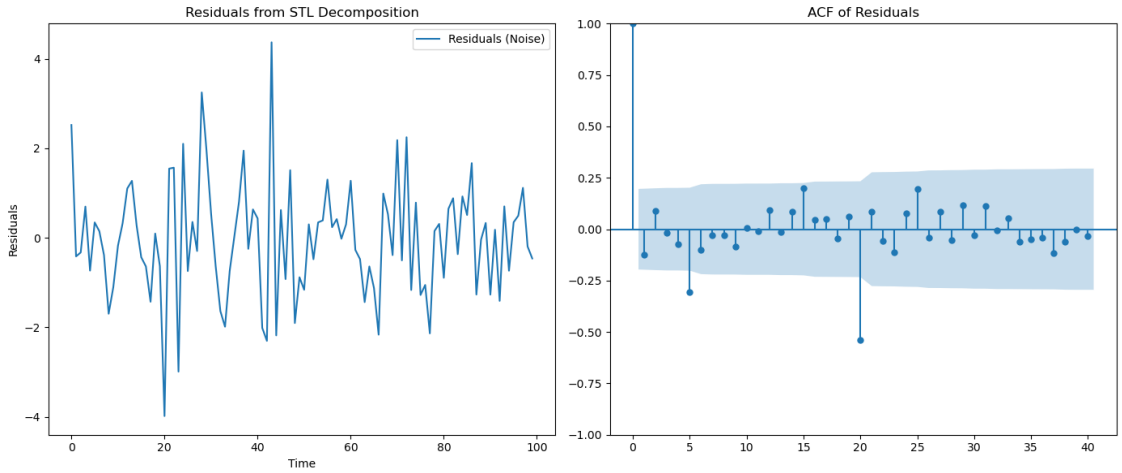

Figure 3: Residual time series and its ACF.

```
print("Ljung-Box test results:")
print(ljung_box_results)
```

2.3 Differencing

Differencing is a powerful technique used to transform a non-stationary time series into a stationary one, which is a prerequisite for many time series models, particularly those used in forecasting. Non-stationarity, often caused by trends or seasonal patterns, can violate the assumptions of these models, leading to inaccurate forecasts and poor model performance. To address this, differencing involves subtracting the previous observation from the current observation, effectively removing the trend or other systematic components from the series. This process can be expressed using the **backshift operator** B, which is defined as $BY_k = Y(t-1)$, shifting the time series back by one period. Using this operator, the **first-order difference** of a time series Y_k is defined as:

$$\Delta Y_k = Y_k - Y(t-1) = (1-B)Y_k$$

This first-order differencing transforms the original series into a new series that represents the change between consecutive observations. If the original series had a linear trend, the first-order differencing would typically remove this trend, making the series more stationary.

In some cases, first-order differencing may not be sufficient to achieve stationarity, particularly when the time series exhibits more complex trends or patterns. In such situations, higher-order differencing can be applied. The **d-th order difference** of a series is given by:

$$\Delta^d Y_k = (1-B)^d Y_k$$

where d is the order of differencing. For example, the second-order difference (when $d = 2$) is obtained by applying the difference operator twice:

$$\Delta^2 Y_k = (1-B)^2 Y_k = (Y_k - Y(t-1)) - (Y(t-1) - Y(t-2)) = \Delta Y_k - \Delta Y(t-1)$$

Higher-order differencing is useful when the series exhibits a non-linear trend or a more complex form of non-stationarity that is not removed by first-order differencing alone. By expressing differencing in terms of the backshift operator B, we can generalize this process efficiently for any order d.

While differencing is an effective tool for achieving stationarity, it is important to apply it carefully. Over-differencing can lead to too much of the original information is lost, resulting in a series that is difficult to model and forecast accurately. Therefore, it is often recommended to use the minimum differencing order necessary to achieve stationarity.

2.4 Python Example: Differencing

In this example, we apply first-order differencing and analyze the differenced series. Here is a step-by-step explanation of the code (these pieces of code can be found in the following GitHub Repository):

- **First-Order Differencing of Residuals:** The code begins by calculating the first-order difference of the residuals using the `np.diff` function. This process creates a new series in which each value represents the difference between consecutive observations, effectively removing any linear trend present in the residuals.

```
# Assuming 'residuals' is readily available
# Create first-order difference of the residuals
diff_residuals = np.diff(residuals, n=1)
```

- **Adjusting the Time Variable:** As differencing reduces the length of the series by one, the time variable is adjusted accordingly by excluding the first time point. The resulting time variable, `time_diff`, aligns with the time points of the differenced residuals.

```
# Create time variable for the differenced series
time_diff = time[1:]  # Adjust time for differencing
```

- **Plotting the First-Order Difference and ACF:** The code generates two plots. The first plot represents the first-order differenced residuals against the adjusted time variable, allowing for a visual inspection of the differenced series. The second plot displays the Autocorrelation Function (ACF) of the differenced residuals, using the `plot_acf` function. The ACF is computed for up to 40 lags, providing a visual representation of any remaining autocorrelation in the differenced series.

```
# Plot the first-order difference of the residuals
plt.figure(figsize=(7, 6))
plt.plot(time_diff, diff_residuals)
plt.xlabel('Time')
plt.ylabel('Differenced Residuals')
plt.title('First-Order Differencing of Residuals')
plt.show()

# Plot the ACF of the first-order differenced residuals
plt.figure(figsize=(7, 6))
plot_acf(diff_residuals, lags=40)
plt.title('ACF of First-Order Differenced Residuals')
plt.show()
```

- **Performing the Ljung-Box Test:** After plotting, the code performs the Ljung-Box test on the differenced residuals using the `acorr_ljungbox` function. This statistical test evaluates whether significant autocorrelation remains at multiple lags. The test is performed for a lag of 20. In this case, the p-value of the statistical test is 1.8×10^{-12}, strongly indicating the presence of significant autocorrelation in the differenced residuals.

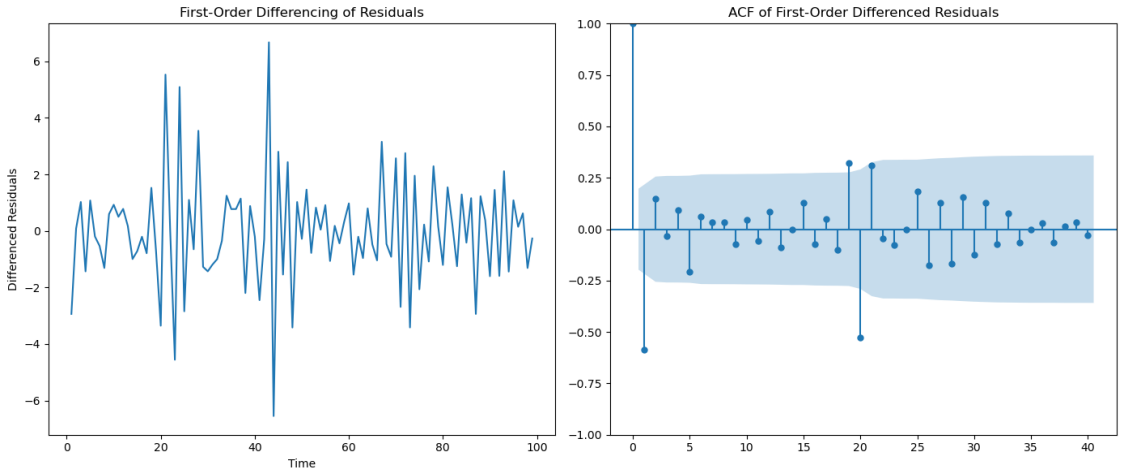

Figure 4: First-order difference of the residuals and its ACF.

```
# Perform and plot  the Ljung-Box test on the differenced residuals
ljung_box_results_diff = acorr_ljungbox(diff_residuals, lags=[20],
    return_df=True)
print("Ljung-Box test results for the differenced residuals:")
print(ljung_box_results_diff)
```

In this example, differencing has not been successful in fully eliminating the autocorrelation from the residuals. In the following chapter, we will explore strategies to address this issue by building models that inherently account for the presence of autocorrelation in the time series.

3 Practical Consideration

Handling outliers, missing values, and non-stationarity is crucial for accurate time series analysis. These issues can introduce bias and significantly affect forecasts if not properly addressed.

3.1 Outliers

Outliers are observations that differ significantly from the majority of the data. They may indicate data errors or genuinely unusual events. Careful consideration should be given before removing outliers, as they might provide important information about the underlying data-generating process. One approach to detect outliers is using the STL decomposition with a robust option, available in Python via the `statsmodels` library using the `STL` function with the `robust=True` argument. Another common method for detecting outliers is using boxplots and interquartile range (IQR) techniques, which can be implemented using the `pandas` or `scipy.stats` libraries.

3.2 Missing Values

Missing data can arise due to various reasons, such as data recording issues or planned events like holidays. If the *missingness* is systematic (e.g., due to holidays), special treatment, such as adding dummy variables in a regression model, may be necessary. For random missingness, several Python

methods can be used to impute missing values. The `pandas` library provides methods like `fillna()` and `interpolate()` for simple imputation. More advanced techniques like `KNNImputer` from the `sklearn.impute` module offer sophisticated imputation strategies. Some models handle missing values natively, while others may require explicit data imputation. For non-seasonal data, linear interpolation is commonly used, whereas seasonal data might require seasonal decomposition, for example, using the STL method, before applying interpolation.

3.3 Heteroskedasticity

Heteroskedasticity refers to the non-constant variance of residuals in time series data, commonly seen in financial time series with varying volatility over time. In classical forecasting models, homoskedasticity, or constant variance, is assumed, and ignoring heteroskedasticity can lead to biased estimates and inaccurate predictions. To address this, specialized models such as **ARCH** and **GARCH** are introduced, which model the time-varying volatility of residuals. While heteroskedasticity is critical in traditional statistical methods, machine learning models like decision trees and neural networks are flexible enough to accommodate varying data spreads without requiring explicit handling of heteroskedasticity.

3.4 Non-Linear Transformations

In time series analysis, it is often beneficial to adjust historical data in order to obtain a simpler and more tractable form. The primary goal of these adjustments is to eliminate known sources of variability or to standardize patterns across the data set. Simplified data patterns are typically easier to model, thereby improving the accuracy of forecasts.

When the magnitude of fluctuations in a time series grows or shrinks with the level of the series, a transformation can be applied to stabilize this variability. For example, in economic data such as inflation rates or GDP growth, larger economies tend to exhibit higher absolute fluctuations compared to smaller ones. In these cases, applying a logarithmic transformation to the data helps stabilize these fluctuations, allowing for a more consistent variance and facilitating more accurate forecasting and analysis.

Python Example: Stock Price Transformation

In the analysis of stock prices, it is common to measure returns rather than prices directly. A widely used method is to compute the log difference between consecutive prices. Let Y_k represent the stock price at time t. The logarithmic return is defined as:

$$w(k) = \log\left(\frac{Y_k}{Y(t-1)}\right).$$

This transformation captures the percentage change in stock price between two consecutive time points. It provides a convenient way to analyze stock returns, as it stabilizes variance and makes the distribution of returns closer to normal. The following Python code retrieves historical stock data, computes log returns, and generates histograms of stock prices and log returns. Below is a breakdown of the code using an itemized format (the code can be found in GitHub):

1. **Import Required Libraries.** We import necessary packages such as `pandas` for data handling, `numpy` for numerical operations, `matplotlib` for plotting, and `yfinance` for downloading stock price data.

```
# Import required libraries
!pip install yfinance
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import yfinance as yf
```

The `yfinance` library allows us to retrieve historical market data from Yahoo! Finance.

2. **Download Historical Stock Data.** Using the `yfinance` library, we download the adjusted closing prices for the Apple stock over a given period.

```
# Download historical stock data (e.g., for Apple stock 'AAPL')
stock_data = yf.download('AAPL', start='2020-01-01', end='2023-01-01')
# Extract adjusted closing prices
prices = stock_data['Adj Close']
```

3. **Plot Temporal Evolution and Histogram of Stock Prices.** We generate a temporal evolution plot and a histogram of the raw stock prices using the adjusted closing price.

```
plt.figure(figsize=(10, 5))

# Subplot 1: Temporal evolution of AAPL stock prices
plt.subplot(1, 2, 1)
plt.plot(prices, label='Stock Prices', color='blue')
plt.title('Temporal Evolution of AAPL Stock Prices')
plt.xlabel('Date')
plt.ylabel('Price')
plt.grid(True)
plt.legend()

# Subplot 2: Histogram of AAPL stock prices
plt.subplot(1, 2, 2)
plt.hist(prices, bins=30, color='blue', alpha=0.7)
plt.title('Histogram of Stock Prices (AAPL)')
plt.xlabel('Price')
plt.ylabel('Frequency')

# Show the plot
plt.tight_layout()
plt.show()
```

4. **Plot Temporal Evolution and Histogram of Log Returns.** We apply the log ratio transformation to compute stock returns, which reflects the percentage change between consecutive stock prices.

```
# Compute log returns (log ratio transformation)
log_returns = np.log(prices / prices.shift(1)).dropna()

plt.figure(figsize=(10, 5))

# Subplot 1: Temporal evolution of log returns
plt.subplot(1, 2, 1)
plt.plot(log_returns, label='Log Returns', color='green')
plt.title('Temporal Evolution of AAPL Log Returns')
plt.xlabel('Date')
```

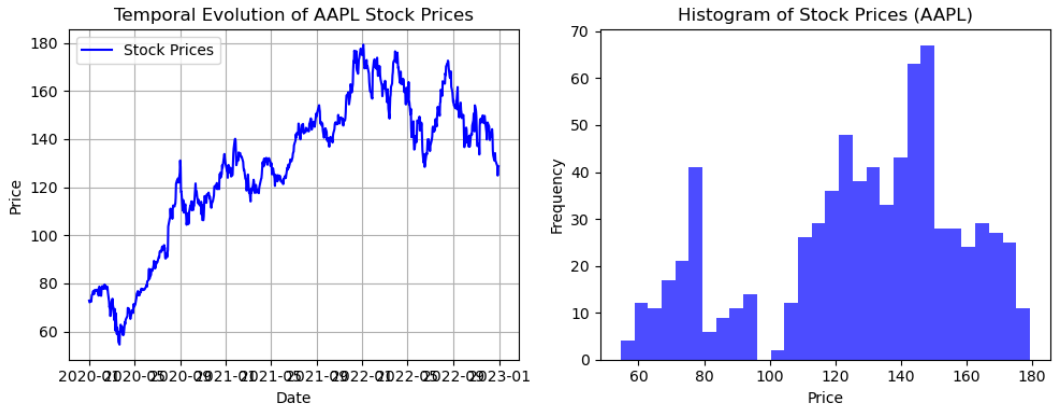

Figure 5: (Left) Histogram of AAPL stock prices. (Right) Histogram of log-transformed returns.

```
plt.ylabel('Log Return')
plt.grid(True)
plt.legend()

# Subplot 2: Plot histogram of log returns
plt.subplot(1, 2, 2)
plt.hist(log_returns, bins=30, color='green', alpha=0.7)
plt.title('Histogram of Log Returns (AAPL)')
plt.xlabel('Log Return')
plt.ylabel('Frequency')

# Show the plot
plt.tight_layout()
plt.show()
```

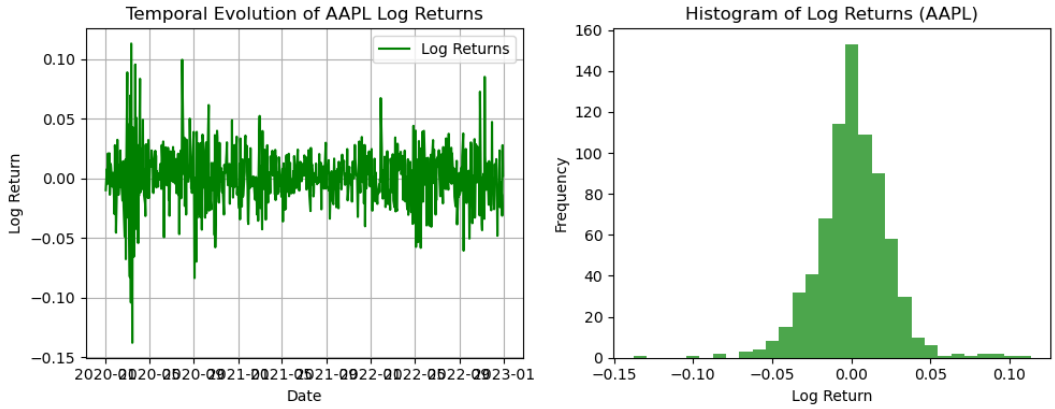

Figure 6: (Left) Histogram of AAPL stock prices. (Right) Histogram of log-transformed returns.

When comparing the time series and histograms of log-returns with the original stock prices, we observe that the log-returns eliminate the trend present in the original price series and stabilize the variance, leading to more consistent fluctuations around zero. While the original prices exhibit a skewed histogram, the log-returns have a more symmetric distribution, resembling a normal distribution centered around zero, though with occasional extreme movements

(fat tails). This transformation simplifies the analysis by normalizing changes and making the data more stationary.

In addition to logarithmic transformations, other types of transformations, such as power transformations or more complex methods, can be used to stabilize a time series and make the distribution closer to normal. While these transformations may be less interpretable than logarithms, they are still useful in specific contexts.

4 Appendix: List of Time-Series Datasets from Kaggle

For future reference, we include in this appendix a list of time series datasets available on *Kaggle* (an online platform and community focused on data science and machine learning), with direct links to each dataset for easy access.

Stock Market Data:

- Yahoo Finance Stock Prices - Stock market data from Yahoo Finance for various companies.
- S&P 500 Stock Data - Historical stock prices for companies listed in the S&P 500.
- Tesla Stock Price - Stock prices for Tesla Inc.
- Cryptocurrency Historical Prices - Historical data for Bitcoin and other cryptocurrencies.

COVID-19 Time Series:

- COVID-19 Global Forecasting - Time-series data tracking COVID-19 globally.
- COVID-19 in India Dataset - Daily time series tracking COVID-19 cases in India.

Retail Sales:

- Walmart Store Sales - Store sales data for Walmart across the U.S.
- Rossmann Store Sales - Time-series data of Rossmann drugstore sales in Germany.
- Favorita Store Sales - Grocery sales forecasting competition dataset.
- Store Item Demand Forecasting Challenge - Time-series data of store sales for demand forecasting.

Weather and Climate:

- Global Land Temperatures - Global surface temperatures, useful for climate change analysis.
- Daily Rainfall in Australia - Weather dataset tracking daily rainfall across Australia.
- Weather Data in Szeged, Hungary - Weather dataset recording temperatures and humidity over multiple years.

- Beijing PM2.5 Air Quality Data - Air pollution data in Beijing tracking particulate matter (PM2.5) over time.

Energy and Power Consumption:

- Household Electric Power Consumption - Electricity usage data for household energy consumption.
- Open Power Systems Data - European electricity production data over 30 years.
- Hourly Energy Consumption - Electricity consumption data for various regions.

Economic and Financial Data:

- M5 Forecasting - Accuracy - Retail sales and financial data used for hierarchical forecasting.
- GDP of Countries - Historical GDP data for countries around the world.

Health and Biomedical Data:

- ECG Signal Classification - Heartbeat data from electrocardiogram (ECG) signals, used for medical diagnosis.
- EEG Brainwave Dataset - Feeling Emotions - EEG data used to track emotional states from brain signals.

Miscellaneous Time-Series Data:

- Astronomy Dataset - Time-series data collected from space missions, useful for detecting anomalies in star behavior.
- CO2 Emissions by Vehicles - Dataset tracking CO2 emissions from vehicles over time.
- Metro Interstate Traffic Volume - Hourly traffic volume on Interstate 94.
- Earthquake Magnitudes - Dataset tracking earthquake magnitudes worldwide over time.

Exercises

1. In time series analysis, **white noise** refers to a stochastic process $\epsilon(k)$ where the elements in the sequence of random variables $\epsilon(k)$ are independent, identically distributed (i.i.d.), and have a mean of zero and constant variance σ^2. When a time series consists solely of white noise, it indicates the absence of any discernible pattern or structure, such as trends, seasonality, or autocorrelation.

   Answer the following questions:

   (a) Can the value of $\epsilon(k)$ be predicted from previous observations $\epsilon(t-1), \epsilon(t-2), \ldots$?

   (b) Compute the mean $\mu(k)$, the variance $\sigma^2(k)$, and the autocorrelation function $\rho(h)$ of white noise.

   (c) Is a white noise process wide-sense stationary (WSS)?

2. Generate a white noise process in Python and empirically verify the properties discussed above. Specifically, follow the steps below:

   (a) Use Python to generate a sequence of 1000 values of white noise $\epsilon(k)$ drawn from a normal distribution with mean 0 and variance 1.

   (b) Plot the generated white noise time series to visually inspect its behavior. Does the time series exhibit any noticeable trends or patterns?

   (c) Numerically estimate the mean and variance of the generated white noise process. Compare your estimates to the theoretical values.

   (d) Plot the autocorrelation function (ACF) of the numerically generated white noise process. Describe what you observe and how it compares to the theoretical expectations.

3. Consider a random process defined by the recursion: $Y_k = \phi_0 + \phi_1 Y(t-1) + \epsilon(k)$, where $\epsilon(k)$ is white noise with mean zero and constant variance σ_ϵ^2. Compute the correlation $\text{Corr}(Y_k, Y(t-2))$.

   Assume that we sum a trend component $\texttt{Trend}(k) = \alpha \cdot t$ to the random process defined above, resulting in a new random process $\widetilde{Y}(k) = Y_k + \texttt{Trend}(k)$. Answer the following questions:

   (a) Compute the theoretical mean, the theoretical variance, and the theoretical covariance of $\widetilde{Y}(k)$. Which of these theoretical values change and which remain unaffected by the addition of a trend?

   (b) Is the new process $\widetilde{Y}(k)$ wide-sense stationary? Explain your answer.

   (c) Is the new process $\widetilde{Y}(k)$ more or less autocorrelated than Y_k? Explain your answer.

4. Consider the random process Y_k defined as:

$$Y_k = 1 + \epsilon(k) + \theta_1 \epsilon(t-1) \tag{1}$$

   where $\epsilon(k)$ are independent and identically distributed (i.i.d.) random variables, each following a standard normal distribution $\mathcal{N}(0,1)$. Answer the following questions:

   (a) Compute the theoretical mean $\mu(k)$, the theoretical variance $\sigma^2(k)$, and the theoretical covariance $\text{Cov}(Y_k, Y_{k-h})$ of the random process Y_k.

(b) Is the random process wide-sense stationary?

(c) Is the random process autocorrelated?

5. Program a piece of Python code to generate a sample path of the random process Y_k defined above and estimate empirically the values of the mean, variance, and autocorrelation of the sample path. Do not use pre-programmed functions; instead, program the estimators from scratch. Compare your numerical estimates with the theoretical values you computed in the previous question.

6. In Section 2.3, we illustrated how first-order differencing does not remove the autocorrelation from the residual. Check numerically using Python whether second-order differencing would be successful in this task.

Project: Energy Consumption Data Analysis

Many electric companies and regional grid operators provide public access to historical energy consumption data. For this assignment, you will explore such data to conduct a time series analysis. One example of a data source is the PJM Interconnection, which offers a comprehensive data portal, PJM Data Miner 2, where you can search for and download energy consumption datasets. Please, perform the following tasks:

1. **Data Acquisition:**
   - Visit the PJM Data Miner 2 portal.
   - Download the"Historical Load Forecast" dataset in CSV format for the full month of July 2024
   - Import the time series from the column `forecast_load_mw` into Python.
2. **Data Analysis:** Using the tools and techniques learned in this chapter, analyze the extracted time series. Specifically, you are required to:
   - Perform a Seasonal-Trend decomposition using LOESS (STL). Plot the decomposed components and provide a critical analysis of your observations.
   - Plot the Autocorrelation Function (ACF) for both the original time series and the residuals from the STL decomposition.
   - Apply the Ljung-Box test to determine if there is significant autocorrelation in the residuals.
   - Perform first-order differencing on the residuals and compare the autocorrelations with the original residuals.
3. **Report:** Summarize your findings in a *Python Notebook* called `LastName_FirstName_STL2024.pynb`, including all relevant code, plots, and interpretations (in text cells) of your data analysis. Make sure to include at the end the **Mean-Square Error** (MSE) of the residual time series, defined as
$$\text{MSE} = \frac{1}{T}\sum_{k=1}^{T}\epsilon(k)^2$$
where $\epsilon(k)$ is the residual at time t and T is the number of observations.