Topic 1B: Time-Series Data Preprocessing

Victor M. Preciado

Contents

1 Characteristics of Time-Series Data

A distinctive attribute of time series data, as opposed to other data types such as images or tabular datasets, is the **temporal ordering of observations**. In time series, the sequence of the data points is essential, as any permutation of the data points would disrupt the inherent temporal structure. This ordering imposes unique challenges in modeling, as the value of an observation at a given point in time often depends on its previous observations, making standard statistical and machine learning techniques insufficient without appropriate modifications.

1.1 Trends, Seasonality, and Noise

The importance of this temporal dependency becomes even more pronounced in real-world applications, where time series data frequently exhibit distinct patterns and behaviors that evolve across time. Understanding and identifying these characteristics is critical for effective forecasting. In particular, real-world time series commonly exhibit the following components:

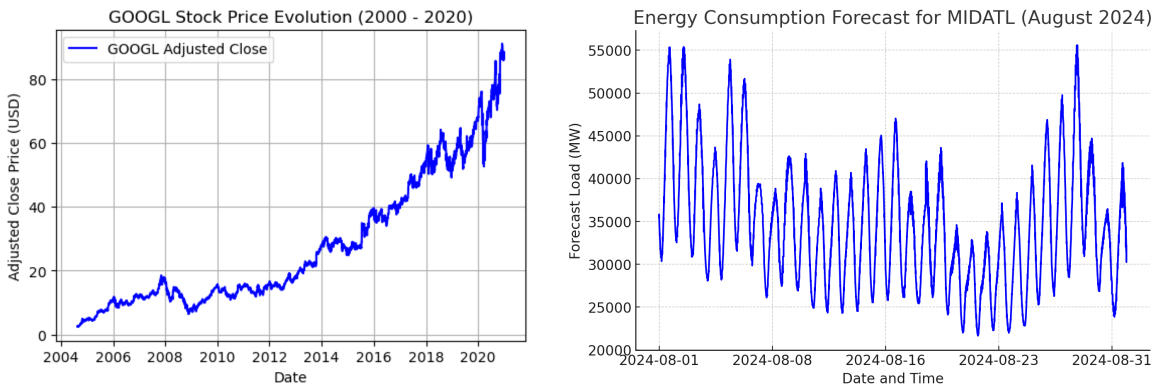

Figure 1: (Left) Evolution of Google stock prices from 2000 to 2020. (Right) Energy consumption in the Mid-Atlantic U.S. region during August 2024.

- A **trend** component representing the long-term averaged evolution of a time series. It captures the underlying direction in which the series is moving, abstracting from short-term fluctuations and cyclical patterns. This component reflects fundamental shifts in the data-generating process over time (see the red line in Fig. 2). A trend reflects a systematic shift in the average level of the series over time, often driven by factors such as economic growth, population changes, or technological advancements. To rigorously analyze the statistical properties of a time series—including its variance and autocorrelation—it is crucial to *detrend* the data. Detrending typically involves fitting a slowly varying function, such as a linear growth, non-linear patterns, or piecewise functions (e.g., splines), to the time series and subtracting it from the original observations, thereby removing the systematic variation in the average level.

- The **seasonal** component captures cyclic patterns that recur at regular intervals, such as daily, monthly, or annually (see the green periodic signal in Fig. 2). These periodic fluctuations often arise in natural phenomena or human activities, such as the daily energy consumption cycle or retail sales during holiday seasons. Mathematically, seasonality can be characterized by identifying periodic components, often through tools from Fourier analysis or other spectral methods (for a detailed treatment, see [1]). To facilitate the modeling and analysis, *seasonal adjustment* is frequently applied to remove these recurring fluctuations. This process allows for a clearer examination of the underlying, non-periodic dynamics of the series.

- A **noise** component (also called **residual**) representing the random variation that remains in the time series after accounting for the trend and seasonal effects. In classical time series forecasting, this residual noise is typically modeled using a WSS process. The noise component introduces random fluctuations that cannot be explained by the deterministic trend and seasonality. Properly identifying and modeling the noise component is crucial for accurate forecasting and for understanding the uncertainty and inherent risks within the time series. The statistical characteristics of this noise—such as its variance and autocorrelation structure—significantly affect the reliability of predictions and the detection of patterns in the data.

Example 1: The Importance of Detrending and Seasonal Adjustment

Consider the following time series model, which consists of a linear trend, a seasonal component, and noise:

$$Y_k = \underbrace{\beta_0 + \beta_1 k}_{\text{Trend}} + \underbrace{\alpha\cos(2\pi k/T)}_{\text{Seasonal}} + \underbrace{\nu_k}_{\text{Noise}}.$$

In this model, β_0 and β_1 define the intercept and slope of the linear trend, α is the amplitude of the periodic seasonal component with period T, and ν_k represents a noise, assumed to be a WSS process with zero mean, variance σ_ν^2, and autocovariance function $\mathrm{Cov}(\nu_k, \nu_{k-h}) = \gamma(h)$, where h is the lag. The presence of both the linear trend and periodic seasonal component introduces non-stationarity into the process, violating the assumption of constant mean and variance required for WSS. This non-stationarity complicates the application of many time series models, which are based on the assumption of stationarity.
To address this issue, we must preprocess the data. Assume that we have observed a sample path of length L, i.e., $\{y_k : k \le L\}$. In order to apply a WSS stochastic process model to this sample path, the following steps are necessary:

1. **Detrending:** The linear trend $\beta_0 + \beta_1 k$ and the seasonal component induce a time-varying mean:
$$\mathbb{E}[Y_k] = \beta_0 + \beta_1 k + \alpha\cos\left(\frac{2\pi k}{T}\right),$$
which clearly violates the WSS requirement for a constant mean. To remove the trend and thereby detrend the series, we fit a linear function, $\widehat{Y}_k = \widehat{\beta}_0 + \widehat{\beta}_1 k$, to the data set $\mathcal{D} = \{(k, y_k) : k \le L\}$, using a least squares regression method. This provides estimates for the intercept and slope of the trend component. The detrended series is then obtained by subtracting the fitted trend from the original series:
$$Y_k^{\text{det}} = Y_k - (\widehat{\beta}_0 + \widehat{\beta}_1 k).$$
At this stage, we have removed the linear trend, but periodic seasonality still induces non-stationarity in the remaining data.

2. **Seasonal Adjustment:** Even after detrending, the periodic seasonal component $\alpha\cos(2\pi k/T)$ prevents Y_k^{det} from being WSS. To achieve stationarity, we must remove this seasonal component, a process known as seasonal adjustment. This can be

done by subtracting the estimated seasonal term from the detrended series:

$$Y_k^{\text{res}} = Y_k^{\text{det}} - \widehat{\alpha} \cos\left(\frac{2\pi k}{\widehat{T}}\right),$$

where $\widehat{\alpha}$ and $\widehat{T}$ are estimates of the amplitude and period, respectively, which can be obtained using spectral analysis techniques such as Fourier transforms (see [1] for more details).

3. **Modeling the Residual Term:** The resulting series $\mathcal{Y}^{\text{res}} = \{Y_k^{\text{res}} : k \in \mathbb{N}\}$ is a residual series composed primarily of the measurement noise ν_k. Assuming that the residual term is wide-sense stationary (WSS), the transformed series can be modeled using methods that rely on stationarity, such as autoregressive (AR) models. These methods allow for further characterization of the noise process and provide insights into any underlying patterns or structure that might remain in the residuals, ensuring that no significant information is left unexplained. In the next chapter, we will cover a few of these techniques in detail, exploring their applications to residual analysis and time series modeling.

It is important to note that failing to remove the trend and seasonal component from a time series can obscure the true properties of the residual noise, leading to incorrect conclusions about the stationarity of the data and limiting the effectiveness of certain time series models. For instance, if the seasonal component is left intact, the variance and autocovariance structure will vary over time, preventing the use of stationary stochastic models for analysis.

Moreover, it is important to emphasize that the residual term is *not* a pure white noise process. While it may contain elements of randomness, it often exhibits statistical patterns—such as autocorrelation or time-dependent structure—that can be exploited for predictive purposes. A common approach to extract useful information from the residual term is to employ stochastic models that transform a white noise input into an output process reflecting the statistical properties of the residual. The next chapter will introduce some of the most important models for accomplishing this task.

Extracting Seasonality, Trend, and Noise from a Time Series

The ability to identify and disentangle the components of seasonality, trend, and noise is critical in time series analysis. The process of isolating these components is called **time series decomposition** and it is a foundational step in modeling the temporal dynamics in a time series. By breaking down a time series into its constituent parts, one can gain deeper insights into the factors driving the observed behavior, which is essential for reliable forecasting and analysis. A widely adopted approach to decomposing time series is the so-called **classical decomposition**. This method assumes that a sample path $\{Y_k : k \leq L\}$ can be expressed as a sum of its components in an **additive model**:

$$Y_k = \texttt{Trend}(k) + \texttt{Seasonal}(k) + \texttt{Residual}(k) \text{ for all } k \leq L.$$

In certain cases, such as financial time series where relative changes are more meaningful than absolute changes, a *multiplicative model*, $Y_k = \texttt{Trend}(k) \times \texttt{Seasonal}(k) \times \texttt{Residual}(k)$, might be more appropriate.

Several advanced algorithms are available for time series decomposition, each designed to handle specific types of data and modeling requirements. Notable methods include:

- **STL (Seasonal and Trend decomposition using Loess):** STL is a versatile and iterative decomposition technique that separates the trend and seasonal components of a time series using LOESS (Locally Estimated Scatterplot Smoothing), which is a regression technique used to smooth data in order to reveal underlying trends. STL is often preferred when seasonality changes over time or when the data requires a more robust and adaptive method. In Python, STL can be accessed through the `STL` function in the `statsmodels` library.
- **MSTL (Multiple Seasonal-Trend Decomposition using Loess):** MSTL extends STL by allowing the decomposition of time series with multiple seasonal components of varying periodicities, such as daily, weekly, and monthly cycles. This approach is particularly useful when the series exhibits complex seasonal behavior that cannot be captured by a single seasonal period. MSTL iteratively fits multiple STL decompositions, each targeting a specific seasonal period, and then combines the results to isolate the trend and residuals more accurately. This flexibility makes MSTL ideal for applications requiring precise modeling of complex seasonality patterns, which can be implemented in Python using the `MSTL` function from the `statsmodels` library.
- **Prophet:** Developed by Meta, Prophet is a forecasting model that automatically decomposes time series data into trend, seasonal, and *holiday effects*. It is particularly suited for time series with strong seasonal components, such as online sales, where patterns follow daily, weekly, or monthly cycles, often driven by shopping habits and holiday events. Prophet also allows users to specify custom seasonalities, making it especially useful when the seasonality is not strictly periodic or when there are external factors influencing seasonality, such as holidays. This algorithm is implemented in Python via the `prophet` package and the `predict` method, which estimates each component.

While some advanced techniques, such as deep learning approaches, can handle certain types of non-stationary data directly, performing a time series decomposition remains valuable for several reasons. On the one hand, it aids in data interpretation, allowing analysts to understand the underlying patterns driving the time series. On the other hand, the decomposition serves as a crucial diagnostic tool, revealing potential issues in the data that might not be apparent in the raw series. Therefore, even when using models capable of handling non-stationary data, performing a time series decomposition can provide insights that inform model selection and parameter tuning, ultimately leading to more accurate and interpretable results.

Python Lab: Classical Decomposition

In this programming example, we perform a classical decomposition of a time series into its Trend, Seasonal, and Residual components using the STL algorithm implemented in the `statsmodels` Python library. The following step-by-step guide walks you through the code (available in this GitHub Repository):

1. **Import necessary libraries:** The code imports the following Python libraries for time series analysis: `statsmodels.tsa.seasonal.STL` for STL decomposition, `plot_acf` for plotting the ACF, and `statsmodels.stats.diagnostic.acorr_ljungbox` for testing the absence of serial correlation in residuals.

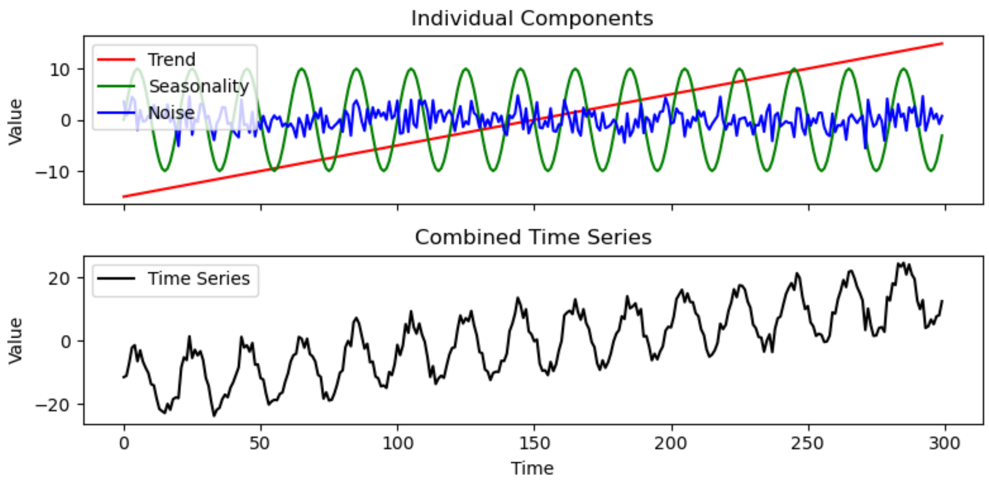

Figure 2: Synthetic time-series with a linear trend, a sinusoidal seasonal component, and an added noise.

```python
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import STL
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.stats.diagnostic import acorr_ljungbox
```

2. **Generate a synthetic time series:** The following code creates a time series by combining a linear trend, a sinusoidal seasonal component with a period of T = 20 , and Gaussian white noise. The resulting time series is shown in Fig. 2.

```python
# Set the random seed for reproducibility
np.random.seed(0)
# Create a time index from 0 to 299
time = np.arange(300)
# Generate a linear trend starting at -10 with the same slope
trend = 0.1 * time - 15  # This will start at -10 and increase with the
    same slope
# Generate a seasonal component with a period of 20
seasonal = 10 * np.sin(2 * np.pi * time / 20)
# Generate some random noise
noise = np.random.normal(scale=2, size=300)

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

# Create the figure and subplots
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 4), sharex=True)
# Plot components in the first subplot
ax1.plot(time, trend, color='red', label='Trend')
ax1.plot(time, seasonal, color='green', label='Seasonality')
ax1.plot(time, noise, color='blue', label='Noise')
ax1.set_ylabel('Value')
ax1.legend()
ax1.set_title('Individual Components')
# Plot combined time series in the second subplot
```

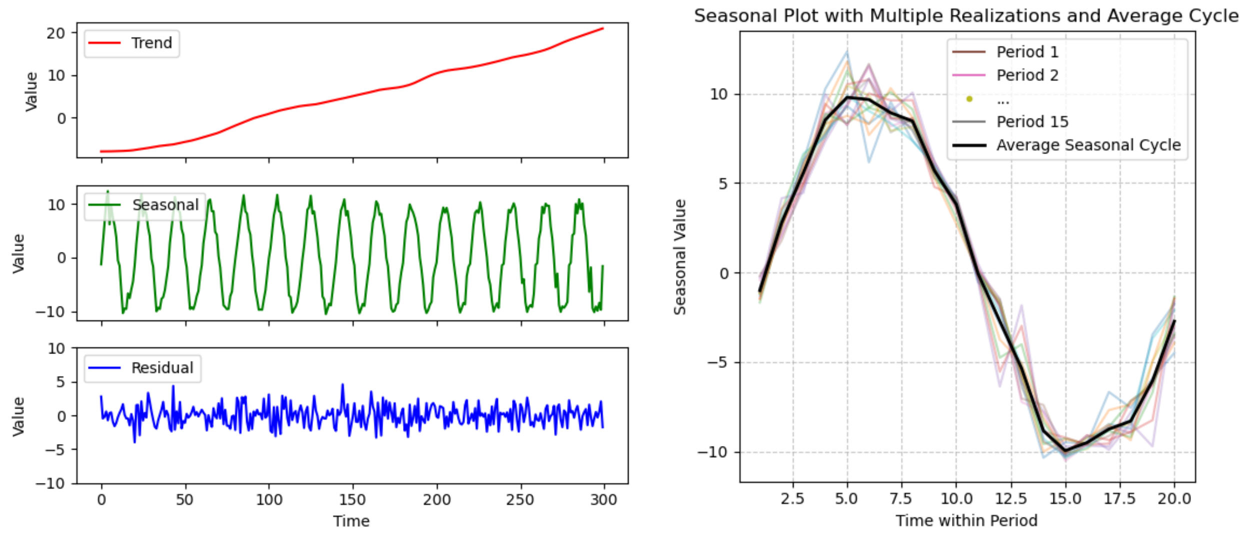

Figure 3: (Left) Decomposition of a time series into Trend, Seasonal, and Residual components. (Right) Cyclical behavior across different seasons of length 20 time units.

```python
ax2.plot(time, series, color='black', linewidth=1.5, label='Time Series
    ')
ax2.set_xlabel('Time')
ax2.set_ylabel('Value')
ax2.legend()
ax2.set_title('Combined Time Series')
# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```

3. **Perform the STL Decomposition:** The code uses the `STL` class to decompose the time series into Trend, Seasonal, and Residual components, specifying a seasonal period of 20 time units. The `fit()` method performs the decomposition, and the components are visualized using the `plot()` function (see Fig. 3-(left)).

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

# Create a plot with three subplots
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(6, 5), sharex=True)
fig.suptitle('STL Decomposition of Time Series', fontsize=16)
# Plot trend component
ax1.plot(time, result.trend, color='red', label='Trend')
ax1.set_ylabel('Value')
ax1.legend(loc='upper left')
# Plot seasonal component
ax2.plot(time, result.seasonal, color='green', label='Seasonal')
ax2.set_ylabel('Value')
ax2.legend(loc='upper left')
# Plot residual component (noise)
ax3.plot(time, result.resid, color='blue', label='Residual')
ax3.set_xlabel('Time')
ax3.set_ylabel('Value')
```

```python
ax3.set_ylim(-10, 10)  # Set y-axis limits for residual plot
ax3.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

4. **Compute the Average Seasonal Cycle:** Since the original time series exhibits a perfectly repeating seasonal pattern, we extract the seasonal component from the STL decomposition and compute an average seasonal cycle to account for variations that occur in each extracted cycle. In Fig. 3-(right), the seasonal cycles for each period ($T = 20$) are superimposed, which helps to visualize their consistency over time. To obtain the average seasonal cycle, the seasonal component is reshaped into individual cycles, and each point is averaged across the cycles, providing a clear representation of the typical seasonal pattern. This approach smooths out irregularities, offering a concise view of the consistent behavior of the time series across its cycles.

```python
# Assuming 'result' is the output from STL decomposition
# Extract the seasonal and trend components
seasonal = result.seasonal
detrended = series - result.trend
# Define the period (e.g., 20 as in the STL decomposition)
period = 20

# Reshape the seasonal component into realizations of each period
n_periods = len(seasonal) // period
seasonal_reshaped = seasonal[:n_periods * period].reshape(n_periods,
    period)

# Calculate the average seasonal cycle
average_seasonal = np.mean(seasonal_reshaped, axis=0)

# Create the seasonal plot
plt.figure(figsize=(5, 5))
# Plot individual seasonal cycles
for i in range(n_periods):
    plt.plot(np.arange(1, period + 1), seasonal_reshaped[i], alpha=0.3)
# Plot the average seasonal cycle
average_line = plt.plot(np.arange(1, period + 1), average_seasonal,
    color='black', linewidth=2, label='Average Seasonal Cycle')
plt.title('Seasonal Plot with Multiple Realizations and Average Cycle')
plt.xlabel('Time within Period')
plt.ylabel('Seasonal Value')
# Customize the legend
if n_periods > 2:
    first_line = plt.plot([], [], label='Period 1')  # Empty plot for
        legend
    second_line = plt.plot([], [], label='Period 2')  # Empty plot for
        legend
    last_line = plt.plot([], [], label=f'Period {n_periods}')  # Empty
        plot for legend
    dots_line = plt.plot([], [], label='...', linestyle='None', marker=
        '.')  # Dots for ...
    plt.legend(handles=first_line + second_line + dots_line + last_line
         + average_line)
else:
    plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

5. **Detrend and Seasonal Adjustment:** This code removes the trend from the original time series using the trend component identified by the STL decomposition. It then adjusts the detrended series for seasonality by subtracting the average seasonal cycle (shown as the black curve in Fig. 3-(right)), resulting in residuals that should be very similar to the noise component of the data. The residual time series is plotted in Fig. 4-(left).

```python
# Assuming the previous STL decomposition has been performed and we
    have the 'result' object
# Extract the trend component from the STL decomposition
trend = result.trend

# Detrend the original series by subtracting the trend component
detrended_series = series - trend

# Repeat the average seasonal cycle to match the length of the original
     series
average_seasonal_cycle = np.tile(average_seasonal, 15)

# Adjust the detrended series for seasonality by subtracting the
    repeated average seasonal cycle
residuals = detrended_series - average_seasonal_cycle
```

In summary, detrending a time series before analysis or modeling offers several practical benefits. First, it enhances stationarity, a key assumption for many time series models, by stabilizing the mean. Second, detrending improves model performance by allowing algorithms to better capture and model the true underlying seasonal and cyclical components without being biased by the overall trend. This separation enables more precise forecasting of short-term movements and improves interpretability. Finally, detrending reduces the risk of misleading correlations, which can occur when the trend is mistakenly interpreted as a meaningful relationship between variables. Overall, detrending helps in achieving more reliable, interpretable, and accurate analytical results in time series modeling.

Furthermore, seasonal adjustment removes predictable seasonal fluctuations that can obscure the underlying patterns, making it easier to identify and analyze non-seasonal trends and irregularities. This is crucial for accurate forecasting, as models can then focus on capturing the genuine movements in the data rather than being distracted by repetitive seasonal variations. Second, seasonal adjustment improves comparability across time periods by eliminating seasonal effects, allowing for a clearer assessment of performance or changes across different times of the year. Lastly, it helps prevent misleading interpretations of data, as raw seasonal patterns can often mask important short-term changes or lead to incorrect conclusions about trends and relationships. By adjusting for trend and seasonality, the data becomes more stable, facilitating a deeper understanding of the true stochastic behavior of the time series.

1.2 Temporal Dependencies

In addition to seasonality, trends, and noise, time series data exhibit **temporal dependencies**, meaning that each observation depends exclusively on prior observations. This characteristic reflects a fundamental aspect of time series data known as **temporal causality**[1], where the current state of the process is influenced by its past but not by its future. Temporal causality captures the inherent one-directional flow of information through time, dictating that past observations drive the current and future values of the series.

Temporal causality implies that the joint density of a sequence of observations cannot be naïvely factored as the product of independent marginal densities, i.e.,

$$f_{\mathcal{Y}}(y_1, \ldots, y_L) \neq f_{Y_1}(y_1) \cdot \ldots \cdot f_{Y_L}(y_L),$$

unless the random process is a collection of independent random variables (e.g., Gaussian white noise). Instead, the presence of causal temporal dependencies means that the joint distribution can be recursively decomposed using conditional probabilities.

Instead, the joint density of the observations can be decomposed into a sequence of conditional densities that capture the influence of past observations on future values. Let $\mathcal{F}_0 = \{Y_0 = y_0, Y_{-1} = y_{-1}, \ldots\}$ denote the information set containing the observations prior to time index 1, which serves as the initial conditions of the process[2]. Conditioned on these initial conditions, the joint density of the stochastic process $\mathcal{Y}$ can be factored as (see Exercise 4 at the end of this chapter):

$$f_{Y_1,\ldots,Y_L|\mathcal{Y}_{\leq 0}}(y_1, \ldots, y_L|\mathcal{F}_0) = \prod_{k=1}^{L} f_{Y_k|\mathcal{Y}_{\leq k-1}}(y_k|\mathcal{F}_{k-1}), \tag{1}$$

where each factor $f_{Y_k|\mathcal{Y}_{\leq k-1}}$ represents the forecast density of Y_k given all available past information up to time $k-1$. This factorization elegantly captures the temporal structure of the data, highlighting how each observation is probabilistically dependent on its predecessors, thereby embodying the principle of temporal causality.

Example 2: Causal Factorization of the AR(1) Model

Consider an AR(1) process defined by the recursion

$$Y_{k+1} = \phi Y_k + \epsilon_{k+1}, \quad \text{with } \epsilon_{k+1} \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2) \text{ and } Y_0 = y_0.$$

To explicitly illustrate the causal factorization of the joint density of this process, consider the factorization of the joint density of $(Y_1, \ldots, Y_L)$ conditioned on the initial condition $Y_0 = y_0$. For the AR(1) process, the factorization simplifies significantly because each observation depends only on its immediate predecessor. Thus, each factor satisfies:

$$f_{Y_k|\mathcal{Y}_{\leq k-1}}(y_k \mid \mathcal{F}_{k-1}) = f_{Y_k|Y_{k-1}}(y_k \mid y_{k-1}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_k - \phi \cdot y_{k-1})^2}{2\sigma^2}\right).$$

Using (1), we obtain the following expression for the joint density of a time series of length L, given the initial condition y_0:

$$f_{Y_1,\ldots,Y_L|Y_0}(y_1, \ldots, y_L \mid y_0) = \left(2\pi\sigma^2\right)^{-L/2} \prod_{k=1}^{L} \exp\left(-\frac{(y_k - \phi \cdot y_{k-1})^2}{2\sigma^2}\right).$$

[1] It is important to distinguish temporal causality from Granger causality. While Granger causality determines whether one time series can predict another, temporal causality implies that the value of a time series at any point depends causally on past values, not future ones.

[2] By convention, non-positive time indices are reserved for initial conditions.

This factorization explicitly shows that the joint density of the observations in an AR(1) process cannot be expressed as a product of independent densities but instead reflects the sequential dependence of each observation on its immediate predecessor. The causal structure of the AR(1) model is thus encoded in these conditional densities, demonstrating how temporal dependencies shape the overall distribution of the time series data.

1.3 Empirical Autocorrelation

Temporal dependencies give rise to **temporal autocorrelation**, reflecting the persistence of patterns across time as past values shape future ones. 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 simply reflect the inherent structure of the trend or seasonal components, rather than revealing any genuine autocorrelation within the noise.

Python Lab: Detecting Autocorrelation

In this lab, we demonstrate how analyze the autocorrelation of the residual component obtained from an STL decomposition. Below, we outline a series of steps using Python:

- **Autocorrelation Function (ACF) Plot:** An analysis of the autocorrelation function of the residuals aims to determine whether any dependencies remain after accounting for the trend and seasonality, thereby providing a more accurate and insightful assessment of the noise process. The sample autocovariance of a sequence of empirical residuals, denoted by $(r_1, r_2, \dots, r_L)$, at lag h can be estimated as:

$$\widehat{C}_{\text{res}}(h) = \frac{1}{L-h}\sum_{i=1}^{L-h}(r_i - \widehat{\mu}_{\text{res}})(r_{i+h} - \widehat{\mu}_{\text{res}}) \quad \text{where} \quad \widehat{\mu}_{\text{res}} = \frac{1}{L}\sum_{i=1}^{L} r_i.$$

  This estimate converges to the true values of the autocovariance as $L \to \infty$ if the residual is *wide-sense stationarity*. From the autocovariance, we can then compute the autocorrelation function as:

$$\widehat{R}_{\text{res}}(h) = \frac{\widehat{C}_{\text{res}}(h)}{\widehat{\sigma}^2_{\text{res}}} \quad \text{where} \quad \widehat{\sigma}^2_{\text{res}} = \frac{1}{L-1}\sum_{i=1}^{L}(r_i - \widehat{\mu})^2.$$

  The ACF plot visualizes the sample autocorrelation at various lags, represented by *spikes*, which are vertical lines indicating the strength of autocorrelation at each lag. In the case of a white noise process, the height of these spikes should close to zero for all $h \neq 0$, indicating no

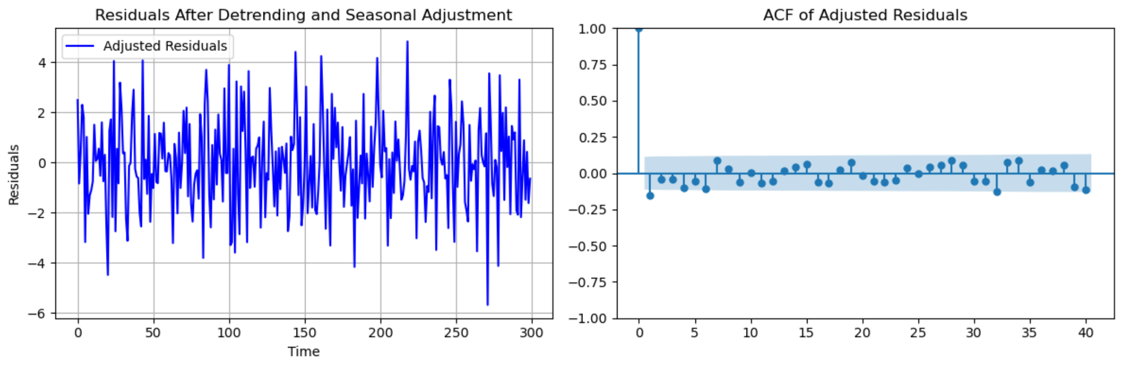

Figure 4: Residual time series from the STL decomposition performed in § 1.1 and its ACF.

significant autocorrelation between lagged observations. However, due to random fluctuations inherent in finite samples, the sample autocorrelation values will rarely be exactly zero.

To account for this natural variability, a confidence interval (CI) is defined around the autocorrelation estimates. Assuming that the residuals follow a white noise process, the 95% CI at lag h is given by $\pm 2/\sqrt{L-h}$, where L is the length of the time series (see Example 3 for a theoretical justification of this CI). Approximately 95% of the spikes in the ACF plot should fall within this confidence interval if the residuals are truly white noise. When a large number of spikes fall outside the CI—significantly more than 5%—it suggests the presence of statistically significant autocorrelation in the residual signal, which could indicate the presence of some underlying temporal pattern not captured by the model.

In Python, we can use the `plot_acf` function from the `statsmodels` library to generate the ACF plot. This plot offers an intuitive visualization of the dependencies between time series values at different lags, aiding in the identification of potential autocorrelation within the residuals.

```python
# Create a figure with two subplots horizontally arranged
fig, axs = plt.subplots(1, 2, figsize=(12, 4))

# Subplot 1: Plot the residuals time series
axs[0].plot(residuals, label='Adjusted Residuals', color='blue')
axs[0].set_title('Residuals After Detrending and Seasonal Adjustment')
axs[0].set_xlabel('Time')
axs[0].set_ylabel('Residuals')
axs[0].grid(True)
axs[0].legend()

# Subplot 2: Plot the ACF of the new residuals to assess
    autocorrelation
plot_acf(residuals, lags=40, alpha=0.05, ax=axs[1])
axs[1].set_title('ACF of Adjusted Residuals')
# Adjust layout to ensure no overlap
plt.tight_layout()
plt.show()
```

In Fig. 4, the residual time series and its ACF are displayed. The ACF plot shows no significant spikes apart from the one at $h = 0$, suggesting that the residuals closely resemble a white noise process, indicating that the trend and sesonality patterns have been effectively removed from the data.

- **Ljung-Box Test:** The Ljung-Box test evaluates whether the residuals of a time series model resemble white noise, which is crucial for validating the adequacy of the model. Approximately 5% of the spikes in the ACF plot are expected to fall outside the 95% confidence bounds purely due to random variation. To assess whether the observed frequency and magnitude of these spikes indicate significant autocorrelation, we apply the **Ljung-Box test**[3].

  The null hypothesis H_0 of the test posits that the residuals exhibit no autocorrelation, i.e., they behave as white noise, while the alternative hypothesis suggests the presence of significant autocorrelation. The Ljung-Box test statistic is defined as:

  $$Q = L(L+2)\sum_{h=1}^{H}\frac{\widehat{R}_{\mathcal{Y}}(h)^2}{L-h},$$

  where H is the number of lags considered in the test (a hyperparameter), and L is the length of the time series. Under H_0, the statistic Q follows a chi-squared distribution with H degrees of freedom [1], allowing us to compute the p-value. A small p-value (typically less than 0.05) indicates that the observed pattern of autocorrelation is unlikely due to random variation, leading to the rejection of the null hypothesis and suggesting significant autocorrelation in the residuals.

  The `acorr_ljungbox` method from the `statsmodels` library in Python can be used to perform this test, as illustrated in the code cell below:

```
# 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
    )

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

  In this case, the p-value obtained from the Ljung-Box test is 0.095. Since this value is above the typical threshold of 0.05, we fail to reject the null hypothesis, suggesting that the residuals do not exhibit significant autocorrelation, and the current model adequately captures the underlying structure of the data.

When the Ljung-Box test indicates significant autocorrelation in the residuals, it becomes necessary to apply time series models that explicitly account for these correlations. In the next chapter, we will introduce methods such as Autoregressive (AR), Moving Average (MA), and their combination, which can address the underlying temporal structure, refine residuals, and lead to more accurate forecasts. These models are capable of capturing both short-term and long-term dependencies by directly modeling the autocorrelations observed in the ACF.

Example 3: Justification of the ACF Confidence Interval

Consider a sample path of length L from a white noise process $\varepsilon_k \sim_{\text{iid}} \mathcal{N}(0,1)$, i.e., a Gaussian process with mean 0 and variance 1. The empirical estimate of the autocorrelation function

[3]For a detailed explanation, see [1]

at lag h is defined as:

$$\widehat{R}_{\boldsymbol{\varepsilon}}(h) = \frac{1}{L-h}\sum_{i=1}^{L-h} \varepsilon_i \varepsilon_{i+h} \approx \mathbb{E}[\varepsilon_i \varepsilon_{i+h}],$$

where the term $L-h$ accounts for the finite length of the time series, ensuring the summation is performed over valid pairs of observations.
Since the process has zero mean, the variance of this empirical estimate can be computed as follows:

$$\operatorname{Var}\left(\widehat{R}_{\boldsymbol{\varepsilon}}(h)\right) = \frac{1}{(L-h)^2}\operatorname{Var}\left(\sum_{i=1}^{L-h} \varepsilon_i\varepsilon_{i+h}\right) = \frac{1}{(L-h)^2}\mathbb{E}\left[\left(\sum_{i=1}^{L-h}\varepsilon_i\varepsilon_{i+h}\right)\left(\sum_{j=1}^{L-h}\varepsilon_j\varepsilon_{h+j}\right)\right].$$

Given that $\boldsymbol{\varepsilon}$ is a white noise process, only the terms where $i=j$ contribute to the expectation. This simplification leads to the following expression for the variance:

$$\operatorname{Var}\left(\widehat{R}_{\boldsymbol{\varepsilon}}(h)\right) = \frac{1}{(L-h)^2}\sum_{i=1}^{L-h}\mathbb{E}\left[\varepsilon_i^2\right]\mathbb{E}\left[\varepsilon_{i+h}^2\right] = \frac{1}{(L-h)^2}(L-h)\mathbb{E}\left[\varepsilon_i^2\right]^2 \text{ for } h \neq 0.$$

Since ε_i has variance 1, $\mathbb{E}\left[\varepsilon_i^2\right] = 1$, we have that:

$$\operatorname{Var}\left(\widehat{R}_{\boldsymbol{\varepsilon}}(h)\right) = \frac{1}{L-h}.$$

This result forms the basis for constructing the commonly used 95% confidence interval $\pm 2/\sqrt{L-h}$ in ACF plots, since the mean of $\widehat{R}_{\boldsymbol{\varepsilon}}(h)$ is equal to zero.

Interpreting the Autocorrelation Function

The ACF is an important tool in time series analysis, capturing the relationship between observations and their lagged values. By examining the ACF, we can identify key patterns such as trends, seasonality, and autocorrelation, which are essential for understanding the series' dynamics and selecting appropriate models. The ACF plot can reveal several characteristics of a time series:

- **Significant Spikes:** Significant spikes indicate statistically relevant autocorrelation at specific lags, suggesting that past values influence future values. Spikes at initial lags point to dependencies on recent past values, typical of Autoregressive (AR) processes, as shown in Fig. 5-(top left).

- **Trends:** Slowly decaying autocorrelations suggest the presence of a trend, which introduces long-term dependencies in the data. For instance, a time series Y_k with a linear trend and a stochastic component:

  $$Y_k = \alpha + \beta k + \phi Y_{k-1} + \epsilon_k \quad \epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2),$$

  exhibits an ACF with persistent tails, as shown in Fig. 5-(top right). This behavior indicates a non-stationary series, highlighting the impact of the trend on the ACF.

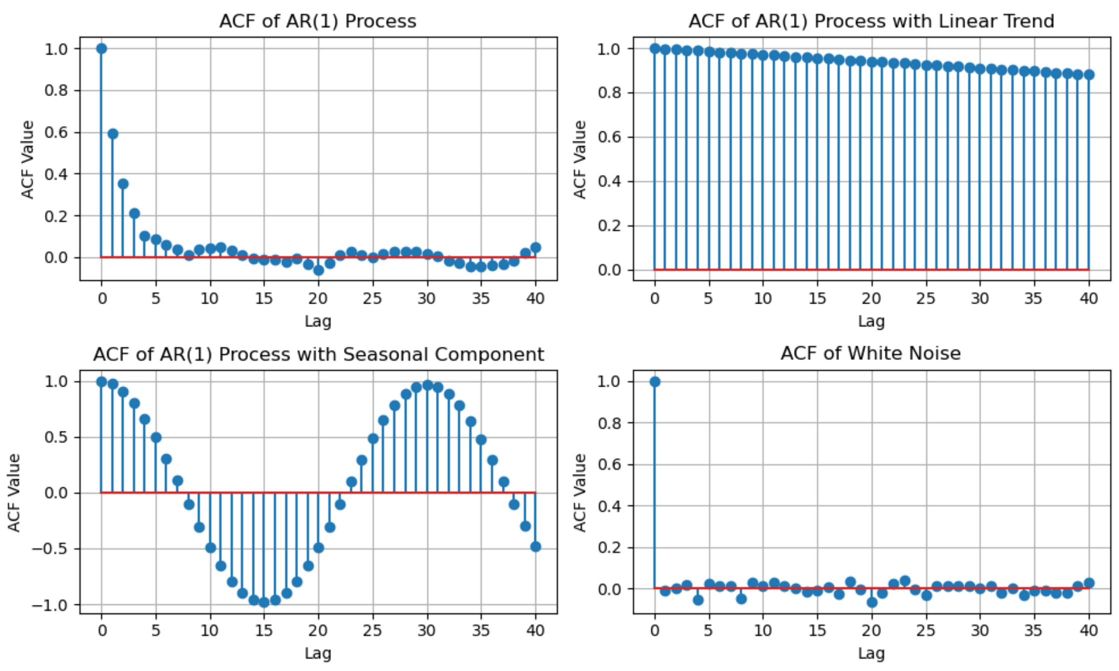

Figure 5: Various types of ACFs illustrating significant spikes, trends, seasonality, and white noise.

- **Seasonality:** Repeating patterns in the ACF at regular intervals indicate seasonality. For example, the ACF of a series like:

$$Y_k = \alpha \cos\left(\frac{2\pi k}{T}\right) + \phi Y_{k-1} + \epsilon_k \quad \epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2)$$

  shows clear oscillations, reflecting the periodic nature of the series (see Fig. 5-(bottom left)).

- **White Noise:** An ACF with no significant spikes, with values falling within the confidence bounds, suggests the series resembles white noise. This behavior, depicted in Fig. 5-(bottom right), indicates a lack of predictable structure, characteristic of stationary processes with rapidly decaying autocorrelations.

The ACF plot is a foundational tool in time series analysis, guiding the identification of necessary preprocessing steps such as detrending and seasonal adjustment, assisting in model selection, and validating model performance through residual analysis. In the ideal case, the model used to predict a time series should produce residuals with an ACF close to that of white noise, indicating that the model has successfully captured all relevant patterns and that no significant temporal structure remains in the residuals.

1.4 Partial Autocorrelation Function (PACF)

In time series analysis, understanding dependencies between observations at different time lags is critical for building effective models. The **Autocorrelation Function (ACF)** is commonly used to measure the correlation between observations at various lags. However, the ACF does not solely represent *direct* linear dependencies between observations. Instead, it reflects both direct and indirect relationships, making it difficult to pinpoint which past observations directly affect the current value in the series.

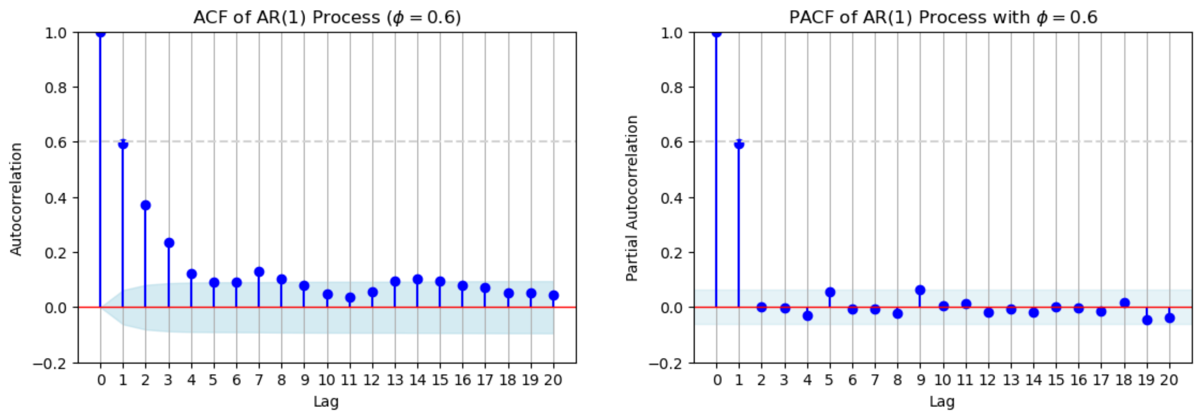

Figure 6: ACF versus PACF of an AR(1). Notice that that the height of the spike at lag 1 in the PACF is equal to the value of the autoregression coefficient $\phi = 0.6$.

For example, the AR(1) model displays exponentially decaying spikes for all lag values, suggesting that each observation influences all future observations. However, this is misleading because only the most recent observation, Y_k, directly impacts the next value, Y_{k+1}. The correlations observed at higher lags are due to indirect relationships where each lagged observation influences subsequent values through intermediate lags. Thus, the ACF captures a cascading effect of dependencies rather than direct causality.

The **Partial Autocorrelation Function (PACF)** addresses this limitation by measuring the *direct* linear relationship between an observation and its lagged values, specifically by removing the effects of intervening observations. The PACF at lag h represents the correlation between Y_k and Y_{k-h}, while controlling for the influence of all intermediate observations between them, specifically those at lags $1, 2, \ldots, h-1$. In essence, the PACF isolates the direct effect of the h-th lag on the current observation, stripping away the impact of any intermediate observations. In practice, the PACF can be computed using the Python library `statsmodels`, which provides the method `pacf()` for estimating the PACF from a sample path.

The PACF provides valuable insights into the direct dependencies within a time series. For an AR(p) model, the PACF will exhibit significant spikes only up to lag p, indicating that the direct influence of past observations only extends to this point. Beyond lag p, the PACF will show values close to zero, as any remaining correlations are indirect and have been accounted for by the previous lags. For example, in an AR(1) model, the PACF will have only significant spikes at lags 0 and 1, correctly indicating that only the immediate past value Y_{k-1} directly influences Y_k (see Fig. 6). In contrast, the ACF would show a gradual decay, reflecting not only direct but also indirect dependencies across the series.

It is important to remark that both the PACF (and the ACF) are only valid measures for Wide-Sense Stationaryprocesses, where the mean and autocovariance function do not change over time. This assumption ensures that the correlation structure remains stable, allowing meaningful interpretation of the PACF and ACF across different lags.

2 Practical Considerations

In this section, we explore some considerations for effectively managing real-world time-series data. Properly addressing challenges such as outliers, missing values, and non-stationarity is essential for accurate analysis. Failure to tackle these issues can introduce bias and significantly impair the quality of forecasts.

2.1 Outliers

Outliers are observations that differ significantly from the majority of the data. They may indicate data errors or genuinely unusual events, and it is essential to assess their relevance before deciding to remove them. In some cases, outliers might hold critical insights about rare or extreme conditions in the underlying data-generating process. However, if left unaddressed, they can skew model results and forecasts. Several methods are available for detecting and handling outliers, each suited to different types of time series data:

- **STL Decomposition with Robust Option:** The Seasonal and Trend decomposition using LOESS (STL) is an effective approach for detecting and handling outliers. By setting the `‘robust=True’` argument in the `STL` function from the `statsmodels` library, the decomposition minimizes the influence of outliers on the trend and seasonal components.

- **Boxplot and Interquartile Range (IQR) Method:** The boxplot is a widely-used graphical method that identifies outliers by highlighting data points that fall outside the typical interquartile range (IQR). This approach is straightforward to implement using the `pandas` or `scipy.stats` libraries and provides a clear visual representation of data dispersion and potential outliers. To calculate the IQR in Python, you can use the `numpy` function `np.percentile()` or the `scipy.stats.iqr()` method, which directly computes the interquartile range for given data.

- **Z-score Method:** This statistical method identifies outliers based on how many standard deviations a data point is from the mean. Data points with a Z-score (standard score) above a certain threshold (commonly 3) are considered outliers. The Z-score method assumes normality of the data, making it ideal when the time series distribution is roughly symmetric. In Python, the Z-scores can be easily computed using the `scipy.stats.zscore()` function, which standardizes the data points based on the mean and standard deviation of the dataset. For more robust outlier detection in normally distributed data, the Extreme Studentized Deviate (ESD) test can also be used, available via the `statsmodels` or `scipy` libraries.

  Isolation Forests: Isolation Forests are an ensemble-based machine learning method specifically designed for anomaly detection. Unlike traditional statistical methods, Isolation Forests do not rely on any assumptions about the distribution of the data. Instead, they work by randomly partitioning the data space and isolating observations that require fewer partitions, which are considered anomalies. This method is particularly effective for high-dimensional data and can handle non-linear relationships and complex data structures. Isolation Forests can be easily implemented in Python using the `IsolationForest` class from the `scikit-learn` library, providing a flexible and powerful tool for identifying outliers in time series and other datasets.

Each method has its unique strengths, tailored to different data characteristics and analytical contexts. Handling outliers thoughtfully is crucial to ensure that the resulting time series model remains accurate and truly reflective of the underlying data patterns, enhancing the robustness and reliability of forecasts. It is important not to remove outliers indiscriminately, as they may represent genuine and significant phenomena rather than mere noise. Moreover, in machine learning techniques, outlier correction is often unnecessary, as these models can adapt and learn directly from the full complexity of the data, including anomalous observations.

2.2 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., data is missing due to holidays or planned outages), special treatment, such as introducing dummy variables in a regression model, may be necessary to account for these periods explicitly.

For random missingness, several Python methods are available to impute (i.e., fill in) the missing values, each with its specific approach:

- `fillna()`: This method replaces missing values with a specified value, such as the mean, median, or a constant. It is a simple approach suitable for situations where the missing data is minimal and does not have a significant impact on the overall pattern.
- `interpolate()`: This method uses interpolation techniques to estimate missing values based on neighboring data points. It is particularly useful in creating a smooth transition that preserves the trend and seasonality.
- `KNNImputer`: This advanced imputation technique uses k-nearest neighbors to predict missing values. The algorithm finds the nearest data points based on similarity and fills in the missing values using the average of these neighbors. This approach can capture complex relationships within the data, making it more accurate for datasets where values are missing at random but show patterns similar to other data points.

Each imputation method has its strengths and is chosen based on the nature and pattern of the missing data. While simple methods are often quick and easy to implement, more advanced techniques like `KNNImputer` provide better performance when dealing with structured data. Choosing the right imputation method ensures that the integrity of the time series is maintained, allowing for more reliable analysis and modeling.

2.3 Non-Linear Transformations

In time series analysis, transforming historical data is a powerful technique that helps to reveal the underlying structure of the series and make it more amenable to modeling. Transformations serve to adjust data characteristics, such as reducing variability, stabilizing variance, or normalizing the scale, which can significantly enhance the interpretability and predictive power of statistical and machine learning models. By addressing non-linearities and heteroscedasticity, these transformations simplify complex patterns, making it easier to identify key dynamics such as trends, cycles, and dependencies, ultimately leading to more robust and accurate forecasts.

The main goal of applying transformations is to create a time series that adheres more closely to the assumptions required by various modeling techniques, such as stationarity and normality. These adjustments help to standardize fluctuations and align the series‘ behavior across different time points, thereby ensuring that the model captures the essential dynamics without being misled by irregularities or inconsistencies in the data. Below are some of the most common transformations used to refine time series data:

- **Logarithmic Transformation:** The logarithmic transformation is frequently applied to time series data exhibiting exponential growth or multiplicative relationships, where variability increases with the level of the series. This transformation compresses the range of the data, particularly affecting larger values more than smaller ones, thereby stabilizing variance and making patterns clearer. The logarithmic transformation is defined as:

  $$Y_k' = \log(Y_k), \quad Y_k > 0.$$

  This approach is especially beneficial in financial time series, where it normalizes returns and mitigates the influence of extreme changes, promoting more consistent forecasting performance.

- **Power Transformations:** Power transformations, including square root and cube root, adjust the scale of data to reduce skewness and stabilize variance, making the data less sensitive to outliers. The general form is:

  $$Y_k' = Y_k^{\lambda}, \quad \lambda > 0.$$

  For instance, the square root transformation ($\lambda = 0.5$) effectively reduces the impact of larger values, making the data distribution more symmetrical and manageable for subsequent analysis.

- **Box-Cox Transformation:** The Box-Cox transformation is a flexible power transformation that adapts to the data by optimizing the parameter λ, allowing it to handle a wide range of skewness and non-linearity. It is particularly useful for stabilizing variance and achieving approximate normality:

  $$Y_k' = \begin{cases} \frac{Y_k^{\lambda}-1}{\lambda}, & \lambda \neq 0, \\ \log(Y_k), & \lambda = 0. \end{cases}$$

  This transformation is invaluable when preparing data for linear regression or other modeling techniques that assume normally distributed residuals.

These transformations, while sometimes reducing the direct interpretability of the data, significantly enhance the ability of models to detect and predict patterns by reducing noise, skewness, and other data irregularities. By refining the data into a more stable and predictable form, these methods lay the groundwork for high-quality forecasting, ensuring that the series reflects the essential behavior without being skewed by scale or volatility effects. In an ideal scenario, a well-transformed and modeled time series should exhibit residuals with an ACF that closely resembles white noise, indicating that the model has successfully captured all temporal dependencies, validating both the transformation and the modeling approach.

Python Lab: Log-Transformation of Stock Prices

In the analysis of stock prices, it is typically more relevant to measure returns rather than prices directly. A widely used method to obtain a more natural scale in stock prices is to transform the time series using the **log-return** between consecutive prices, defined as:

$$Y_k' = \log\left(\frac{Y_k}{Y_{k-1}}\right).$$

This transformation captures the percentage change in stock price between two consecutive time points, offering several advantages over the direct analysis of raw prices. For example, the log-transformation stabilizes the variance, especially in financial data where volatility tends to grow with the level of the price series. Furthermore, the log-returns tend to make the distribution of returns more symmetric and closer to normality, which is an assumption in many financial models. These benefits make log returns a standard tool in the analysis of stock market data.

The following Python code retrieves historical data about the APPLE stocks, 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, and `numpy` for numerical operations, `matplotlib` for plotting. Additionally, we use `yfinance`, a convenient tool for downloading historical stock data directly from Yahoo! Finance, which seamlessly integrates with `pandas` for easy analysis.

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

2. **Download Historical Stock Data.** Using the `yfinance` library, we retrieve the daily prices for Apple Inc. (AAPL) over a three-year period from the beginning of 2020 to the end of 2023. This data provides a clean representation of the stock's price evolution, adjusted for corporate actions like dividends and splits, making it suitable for time series analysis and forecasting.

```python
# 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.** This code shows both the temporal evolution and a histogram of the raw stock prices using the adjusted closing price.

```python
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')

```

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

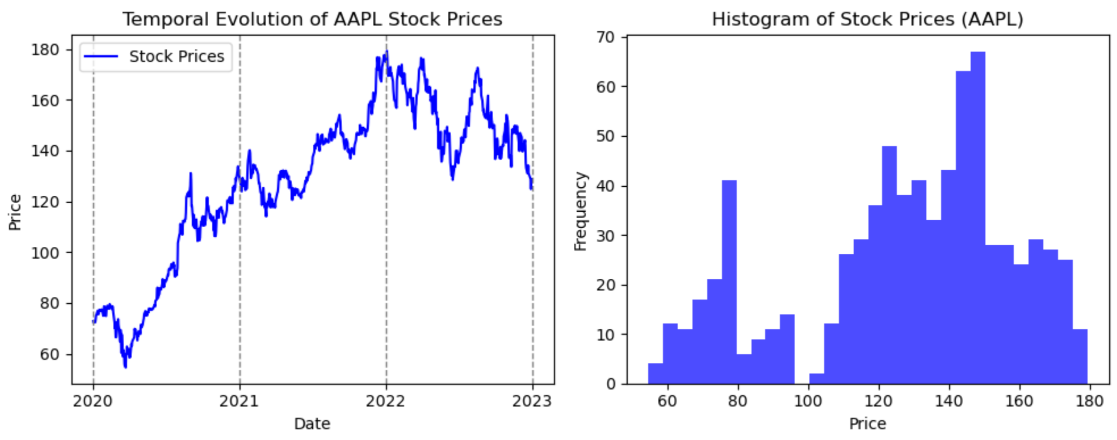

Figure 7: (Left) Histogram of AAPL stock prices. (Right) Histogram of log-transformed returns.

Notice that the histogram of the stock prices is far from normally distributed. Stock prices often display skewness, making traditional statistical methods that assume normality less effective. By transforming stock prices into log returns, we aim to stabilize the variance and make the distribution closer to normal. This transformation not only simplifies modeling but also aligns better with the underlying assumption of many time series models, enhancing the accuracy and reliability of forecasts.

4. **Log-Returns.** The following code applies the log return transformation to the original stock values. By computing the logarithmic returns, we capture the relative percentage change in stock prices between consecutive time points, which helps stabilize the variance and make the data more normally distributed. We then plot the non-linearly transformed time series and a histogram of the log-returns, providing insights into the distribution and temporal dynamics of the returns.

```python
# 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')
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()
```

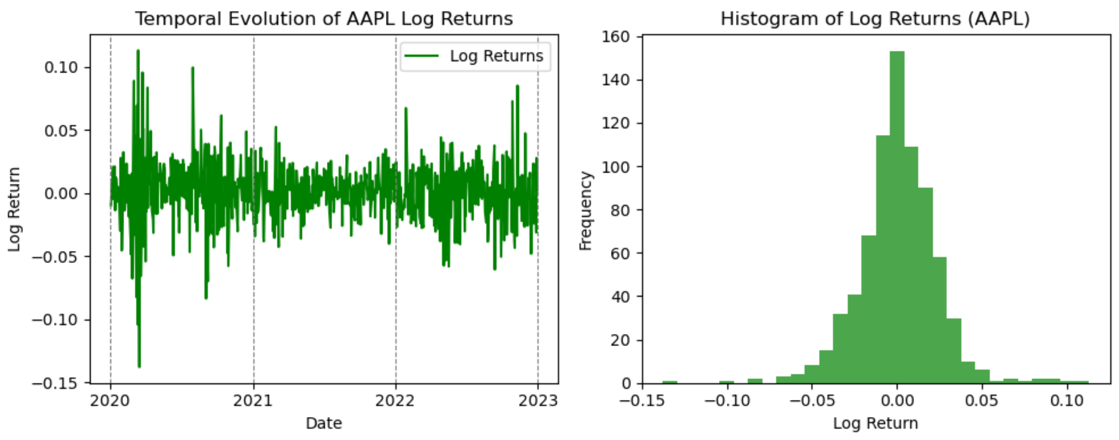

Figure 8: (Left) Time series of log returns. (Right) Histogram of log-returns.

2.4 Heteroskedasticity

Heteroskedasticity refers to the condition where the variance of residuals in a time series is not constant over time, often manifesting as periods of increased or decreased volatility (see Fig. 7-left). This phenomenon is particularly prevalent in financial time series, where markets experience alternating periods of high and low volatility, such as during economic crises or calm market phases. In classical forecasting models, it is generally assumed that residuals exhibit **homoskedasticity**, meaning the variance remains constant over time. However, ignoring heteroskedasticity when present can lead to biased parameter estimates, reduced model efficiency, and unreliable forecasts.

To detect heteroskedasticity in residuals, a common approach is to perform a hypothesis test, such as the **Breusch-Pagan test** or the **White test**. The null hypothesis of these tests is that the residuals exhibit constant variance (homoskedasticity). In Python, these tests can be implemented using the `het_bpg` method from the `statsmodels.stats.diagnostic` module. The Breusch-Pagan test returns a test statistic and a p-value. If the p-value is below the threshold (e.g., 0.05), we reject the null hypothesis, concluding that heteroskedasticity is present.

Furthermore, to address heteroskedasticity, specialized models such as **ARCH** (Autoregressive Conditional Heteroskedasticity) and **GARCH** (Generalized ARCH) have been developed [1]. ARCH models capture time-varying volatility by modeling the current variance of the residuals as a function of past squared errors, effectively allowing the variance to change over time in response to prior shocks. GARCH extends this approach by incorporating past variances as well, making the model more flexible and capable of capturing persistent volatility. These methods are particularly useful in financial applications where accurate modeling of risk and volatility is crucial. ARCH and GARCH models will be introduced in detail in the next chapter, where their structure and application to time series forecasting will be thoroughly explored.

While heteroskedasticity poses significant challenges for traditional statistical models, many machine learning approaches, including decision trees, random forests, and neural networks, are inherently robust to varying data spreads. These models do not rely on assumptions of constant variance, allowing them to handle heteroskedasticity without explicit modifications. For instance, tree-based models partition the data into subsets, each with its own variance, thereby accommodating shifts in volatility. Neural networks, particularly recurrent architectures like LSTMs, learn complex patterns in the data, including changes in variance, without being constrained by traditional statistical assumptions. This inherent flexibility makes machine learning methods valuable tools for modeling heteroskedastic time series without needing specialized statistical interventions.

In the ideal scenario, the model used to predict a time series should yield residuals whose ACF is close to white noise, indicating that all systematic patterns in the data, including heteroskedasticity, have been adequately captured by the model.

3 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. Check links

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. (1 point) Discuss the primary benefits of detrending a time series before further analysis or modeling.

2. (1 point) Outline the key advantages of removing the seasonal component from a time series prior to processing or forecasting.

3. (6 points) Consider a time series $\mathcal{X} = (X_1, X_2, \ldots\}$ defined by the following recursion:

$$X_k = \alpha \cos\left(\frac{2\pi k}{T}\right) + \beta \sin\left(\frac{2\pi k}{T}\right) + \phi X_{k-1} + \epsilon_k + \theta \epsilon_{k-1} \quad \text{with } \epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2), .$$

   Answer the following questions:

   (a) Write a Python script to generate and plot a sample path of length $L = 1000$ using the following parameters: $\alpha = 2$, $\beta = 0.5$, $T = 50$, $\phi = 0.9$, $\theta = 0.7$, $\sigma_\epsilon^2 = 1$, and $Y_0 = 0$.

   (b) Plot the autocorrelation function (ACF) of the generated sample path.

   (c) Perform a seasonal adjustment of the sample path generated in the previous step using the `stl`. Ensure that the seasonal component being removed is strictly periodic (e.g., compute the average of all seasons). Plot the residual signal after the seasonal adjustment.

   (d) Plot the ACF of the residuals.

   (e) Does the seasonal adjustment aids in the visual analysis of the ACF? Justify your answer.

4. (5 points) Given two continuous random variables X and Y with a joint density function $f_{X,Y}(x, y)$, the conditional density is defined as:

$$f_{X|Y}(x \mid y) = \frac{f_{XY}(x, y)}{f_Y(y)}.$$

   (a) Use the definition of conditional density to prove that:

$$f_{Y_1 Y_2}(y_1, y_2) = f_{Y_1}(y_1) f_{Y_2|Y_1}(y_2|y_1).$$

   (b) Let us define the set of random variable $\mathcal{Y}_{\leq k} = \{Y_k, Y_{k-1}, Y_{k-2}, \ldots\}$ and the related information set $\mathcal{F}_k = \{Y_k = y_k, Y_{k-1} = y_{k-1}, Y_{k-2} = y_{k-2}, \ldots\}$. Using the definition of conditional density, show how to prove:

$$f_{Y_1 Y_2|\mathcal{Y}_{\leq 0}}(y_1, y_2|\mathcal{F}_0) = f_{Y_1|\mathcal{Y}_{\leq 0}}(y_1|\mathcal{F}_0) f_{Y_2|\mathcal{Y}_{\leq 1}}(y_2|\mathcal{F}_1).$$

   (c) Using similar steps in a recursive manner, show how to prove the following identity:

$$f_{Y_1 Y_2 Y_3|\mathcal{Y}_{\leq 0}}(y_1, y_2, y_3|\mathcal{F}_0) = f_{Y_1|\mathcal{Y}_{\leq 0}}(y_1|\mathcal{F}_0) f_{Y_2|\mathcal{Y}_{\leq 1}}(y_2|\mathcal{F}_1) f_{Y_3|\mathcal{Y}_{\leq 2}}(y_3|\mathcal{F}_2)$$

5. (4 points) Consider the following stochastic process:

$$Y_{k+1} = \phi Y_k + \epsilon_{k+1} + \theta \epsilon_k, \quad \text{with } \epsilon_k \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2),\ |\phi| < 1, \text{ and } Y_0 = y_0.$$

   Using the *causal factorization* in (1), provide a closed-form expression for:

$$f_{Y_1, \ldots, Y_L|\mathcal{Y}_{\leq 0}}(y_1, \ldots, y_L|\mathcal{F}_0).$$

6. (1 point) Based on a visual inspection, determine whether the temporal evolution of AAPL stock prices (Fig. 7-left) appears stationary. Justify your answer with observations.

7. (1 point) Evaluate whether the temporal evolution of AAPL log returns (Fig. 8-left) is stationary. Conduct a formal hypothesis test in Python and interpret the results based on the observed p-value.

Project: Energy Consumption Data Analysis

(8 point)

The goal of this project is to analyze historical energy consumption data from the PJM Interconnection, a regional transmission organization that coordinates the movement of wholesale electricity in multiple U.S. states. For this project, you will work with the Historical Load Forecast data, which can be downloaded in CSV format for the period from January 1, 2024, to June 30, 2024, using this LINK. Your task is to perform a detailed analysis using the time series tools and techniques covered in this chapter.

Follow these steps below to complete the project:

- **Data Import:** Import the time series data from the column `forecast_load_mw` in the provided CSV into Python.

- **Seasonal-Trend Decomposition:** Perform a Seasonal-Trend decomposition using LOESS (STL) on the imported time series. Focus solely on the daily periodicity ($T = 24$ h.) and disregard any weekly or monthly seasonalities. The objective is to separate the time series into its trend, seasonal, and residual components.

- **Plot and Analyze the Components:** Plot the decomposed components (Trend, Seasonal, and Residual) obtained from the previous STL decomposition (include the '`robust=True`' option). Provide a critical analysis of each component, discussing how the identified patterns reflect the underlying structure of the energy consumption data. Highlight any observed trends, anomalies, or periodicities, and their potential implications.

- **Autocorrelation Analysis:** Plot the Autocorrelation Functions (ACF) up to 30 lags of the following time series:

  1. The original time series.
  2. The detrended time series.
  3. The residual series (after both trend and seasonal components are removed). Do not force a purely periodic seasonal component.

  Discuss how the ACF plots change after each decomposition step and what these changes imply about the presence of temporal dependencies in the data.

- **Ljung-Box Test:** Apply the Ljung-Box test to the residuals obtained from the STL decomposition using 30 lags. The test should help determine whether significant autocorrelation remains in the residuals, indicating that the residuals are not yet white noise. Report the p-value of the test and discuss its significance in the context of forecasting.

Report Submission: Compile your findings in a well-organized *Python Notebook* named

`LastName_FirstName_STL2024.pynb`.

Your notebook should include:

- All relevant Python code used for data import, decomposition, and analysis.
- Clearly labeled plots for the STL decomposition components and ACFs.
- Text cells with detailed interpretations of your results, including critical observations and implications of the analysis.
- A brief conclusion summarizing the key findings and any recommendations for further analysis or model improvement.

Ensure your work is clear, concise, and reflects an understanding of the concepts discussed in this chapter.

References

[1] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. *Time series analysis: forecasting and control.* John Wiley & Sons, 2015.