7 Fitting time series models

This chapter covers

- Time series components and analysis
- ARIMA models
- Exponential smoothing models
- Model evaluation and diagnostics
- Forecasting

Transitioning from traditional models like linear and logistic regression and decision trees and random forests to time series analysis represents a shift from working with unordered data sets to analyzing sequential, time-ordered data where temporal patterns, such as trends and seasonality, may play a significant role. In many statistical methods, we often work with independent and identically distributed data, where predictions are made based on historical data without considering the order of the data points. However, in time series analysis, the temporal order of observations is crucial, as each data point may depend on prior values.

This chapter will examine the fundamentals of time series analysis, providing a comprehensive overview of models like ARIMA (used for capturing patterns based on past values) and exponential smoothing (which forecasts future values by weighing recent observations more heavily). We will explore practical applications by forecasting closing stock prices of a public company and comparing competing models fit to the same data. Note, however, that stock prices are influenced by numerous unpredictable factors, such as public opinion, which can limit the accuracy of forecasts. By the end of this chapter, you will be equipped with foundational techniques and tools to perform reliable time series forecasting, thereby enabling you to make informed decisions based on temporal data.

7.1 Distinguishing forecasts from predictions

In time series analysis, we focus on making *forecasts* rather than *predictions*. Although these terms are often used interchangeably, they have distinct meanings in this context. Most machine learning models make predictions based on patterns in the input data, but they often do not account for time order unless specifically designed for time-dependent data. Forecasts, on the other hand, involve predicting future values based on past observations, accounting for the time-based structure of the data. This transition necessitates understanding the temporal dependencies and patterns that characterize time series data.

Working with time series or longitudinal data involves analyzing data points collected or recorded at specific time intervals (seconds, minutes, hours, days, months, etc.). This contrasts with cross-sectional data, where observations are collected at a single point in time. Time series data can be found in various fields, including finance (e.g., closing stock prices), economics (e.g., gross domestic product), healthcare (e.g., patient vital signs), and digital analytics (e.g., web traffic). Analyzing such data often aims to uncover trends, seasonal patterns, and cyclic behaviors that can provide insights and support forecasting.

The two most common methods for time series forecasting are ARIMA (autoregressive integrated moving average) and exponential smoothing. ARIMA models are powerful tools that combine autoregression (AR), differencing (I for *integration*), and moving averages (MAs) to model time series data. These models are particularly useful for capturing various temporal dependencies and trends within the data. On the other hand, exponential smoothing techniques, including simple, double, and Holt–Winters exponential smoothing, are used to smooth out the noise in the data and highlight underlying trends and seasonal patterns.

Both ARIMA and exponential smoothing methods offer valuable approaches to time series forecasting, each suited to different data characteristics. ARIMA models are powerful for capturing complex temporal dependencies and trends, especially in data with autocorrelations or nonstationarity. Exponential smoothing methods, on the other hand, are valued for their simplicity and effectiveness in highlighting short-term fluctuations and long-term trends in relatively stable data sets. By selecting the appropriate method, whether ARIMA for more intricate patterns or exponential smoothing for consistent trends, we can generate forecasts that better inform decision-making across various domains.

7.2 Importing and plotting the data

Forecasting stock prices is one of the most popular and challenging time series use cases due to its direct effect on financial markets and investment decisions. Stock prices are influenced by a multitude of factors, including economic indicators, company performance, mergers and acquisitions, market sentiment, geopolitical events, and even psychological factors such as investor behavior. The complex interplay of these variables makes predicting stock prices accurately a formidable task.

Several reasons contribute to the difficulty of accurately forecasting stock prices. First, stock prices exhibit nonstationary and volatile behavior, meaning they can vary significantly over time and are influenced by sudden shifts in market conditions or investor sentiment. These abrupt changes can make it challenging to capture long-term trends and forecast short-term movements. Second, financial markets are generally efficient and incorporate new information rapidly. Even small pieces of news or economic data can cause significant price movements, making it challenging for models to consistently outperform the market—although high-frequency trading may exploit brief inefficiencies. Third, the underlying dynamics driving stock prices are often nonlinear and may not adhere to simple linear relationships. Traditional statistical methods like ARIMA and exponential smoothing models may struggle to capture these complex patterns effectively.

Despite these challenges, similar methods and approaches in time series analysis apply across different domains. Techniques such as ARIMA and exponential smoothing are versatile tools used not only for stock price forecasting but also for predicting demand in retail, traffic patterns in transportation, energy consumption, and many other applications. The fundamental principles of identifying patterns, incorporating seasonality or trends, and handling noisy data remain consistent, underscoring the broad applicability of time series analysis techniques.

Although forecasting stock prices poses unique difficulties due to the intricate nature of financial markets and the rapid dissemination of information, the methodologies employed in time series analysis offer valuable insights and predictive capabilities across diverse fields beyond finance. This versatility highlights the robustness and utility of time series models in tackling forecasting challenges across various domains.

7.2.1 Fetching financial data

When it comes to retrieving historical market data directly in a Python script, establishing a connection to Yahoo Finance is no doubt the best way to go. The yfinance library is a powerful tool that streamlines the process. This library offers seamless access to a wide range of financial data, including historical stock prices, trading volumes, dividends, and more. By specifying the ticker symbol and date range, analysts and researchers can efficiently fetch and utilize data for various financial analyses and modeling tasks. Furthermore, yfinance integrates well with other Python libraries such as pandas, allowing for easy manipulation, visualization, and further analysis of the fetched data.

This makes it an invaluable resource for anyone involved in financial research, market analysis, or investment strategy development within the Python ecosystem.

Our first order of business is to import the yfinance library (assuming it has already been installed, of course). Importing it is no different than importing any other Python library:

```
>>> import yfinance as yf
```

We now have direct access to every method and attribute offered by the yfinance library, which is easily accessible via the `yf` alias.

Our next line of code is a straightforward assignment statement that initializes a variable called `ticker_symbol` with the ticker symbol `AAPL`. Because `AAPL` is a character string, it must be bounded by single or double quotation marks:

```
>>> ticker_symbol = 'AAPL'
```

In finance, a ticker symbol is a unique series of letters assigned to a security for trading purposes; it identifies a specific stock or security on a stock exchange or trading platform. For example, `AAPL` is the ticker symbol for Apple, Inc.

Ticker symbols

There are several efficient methods to retrieve ticker symbols. Financial websites—not just Yahoo Finance, but also Google Finance, for instance—allow you to search by company name to find the corresponding ticker symbol. Similarly, financial news websites like MarketWatch, Bloomberg, and CNBC provide search functions for this purpose. Stock exchange websites, including the NYSE and NASDAQ, offer search tools to locate ticker symbols for listed companies. Additionally, financial data platforms such as Morningstar, Reuters, and S&P Capital IQ include ticker symbol lookup features.

Assigning the ticker symbol to a variable is optional, but doing so significantly simplifies code maintenance. Rather than manually replacing every instance of a specific ticker symbol (like `AAPL`) throughout the code, defining it in one place allows you to change it more easily whenever needed.

To efficiently fetch historical stock data, we've created a function, `fetch_stock_data()`, that encapsulates the data-fetching steps. This approach makes the code more modular, reusable, and adaptable to different stocks, date ranges, and intervals. By using a function, you can quickly fetch data for any ticker symbol without duplicating code, which also reduces the chance of errors when changing parameters:

```
>>> def fetch_stock_data(ticker_symbol, start_date, end_date,
>>>                      interval = '1d'):
>>>     ticker_data = yf.Ticker(ticker_symbol)
>>> stock_data = ticker_data.history(start = start_date,
                                 end = end_date,
                                 interval = interval)
>>> return stock_data
```

This function works in two main steps: first, it initializes a Ticker object associated with the given ticker symbol, connecting Python to Yahoo Finance's data for that stock. Second, it calls the history() method on the Ticker object to fetch the stock's historical data. Passing the start and end dates (formatted in the ISO 8601 format YYYY-MM-DD) and specifying the desired interval allows for precise and accurate retrieval of historical data.

You can then use this function to fetch stock data for any ticker symbol and date range by providing the relevant parameters. Because the interval parameter defaults to '1d', you don't need to specify it unless you want data at a different frequency, like hourly ('1h') or weekly ('1wk'):

```
>>> stock_data = fetch_stock_data('AAPL', '2023-10-01', '2024-04-30')
```

This approach makes your code more flexible, allowing you to retrieve data for any stock symbol or time period simply by changing the function parameters. Additionally, specifying dates in ISO 8601 format ensures compatibility with the history() method and prevents interpretation errors. In this example, the fetch_stock_data() function streamlines the data-fetching process. Specifying the start and end dates in ISO 8601 format ensures compatibility, as it is the required date format for the code to run without throwing an error.

ISO 8601 date formats

ISO 8601 is an internationally recognized standard for representing dates and times in a clear and unambiguous manner. It specifies formats such as YYYY-MM-DD for dates, ensuring consistency across different countries and systems. In programming and data analysis, adhering to ISO 8601 ensures compatibility and simplifies date handling, as it reduces ambiguity about the order of year, month, and day components. This standard also includes formats for times, durations, and combined date-time representations, providing a comprehensive framework for precise date and time representation in various applications, from financial data analysis to software development and beyond.

In Python, the datetime module facilitates working with ISO 8601-compliant dates and times, allowing for easy parsing. Embracing ISO 8601 promotes clarity and interoperability, which are essential for accurate data processing and communication across diverse platforms and regions.

We now have a pandas data frame called stock_data that contains seven months of Apple market information. Apart from the temporal aspect of the data, our primary interest lies in the daily closing price of the stock. Therefore, we subset stock_data to extract the Close variable and assign the resulting data to a new data frame named close_prices:

```
>>> close_prices = stock_data[['Close']]
```

This dropped the date index: financial data series are typically indexed by date, or timestamp, which means each observation corresponds to one interval of market activity. Our next line of code resets the `close_prices` index with a default numeric index starting from zero:

```
>>> close_prices = close_prices.reset_index(drop = True)
```

This was another discretionary operation, but replacing the timestamp with a numeric index, which squarely corresponds to trading days (days when financial markets are open), will simplify our upcoming time series charts. Using numerical labels on the *x* axis makes the charts cleaner and more straightforward, thereby avoiding the clutter of date timestamps. However, note that this approach depends on the specific market's open and close times, which may differ across exchanges.

7.2.2 Understanding the data

The `info()` method returns summary information about a pandas data frame, including row and column counts, non-null counts, and data types. It's a quick and easy way to get an additional understanding of a data frame before considering further analysis or preprocessing steps:

```
>>> print(close_prices.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145 entries, 0 to 144
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   145 non-null    float64
dtypes: float64(1)
memory usage: 1.3 KB
None
```

As expected, the data retrieved from Yahoo Finance shows no null values, and the closing price of Apple's stock, representing the price at the end of each trading day, is consistently displayed as floating-point numbers. Stock prices are typically presented as floating-point numbers (floats) due to their inherent need for precision in representing fractional values, such as cents and smaller price increments. This format ensures that financial data retains accuracy and granularity, which is crucial for comprehensive analysis and decision-making in trading and investment contexts. By using floats, financial data sets can accommodate a wide range of price fluctuations while maintaining computational efficiency and compatibility across various financial platforms and systems.

The `head()` and `tail()` methods print the top and bottom of a data frame, defaulting to five rows each when no parameters are specified. Note that historical stock prices may change slightly over time due to adjustments for dividends, stock splits, or corrections by data providers; so, your results may differ, albeit slightly:

```
>>> print(close_prices.head())
        Close
0  173.300247
1  171.953735
2  173.210495
3  174.457260
4  177.030579

print(close_prices.tail())
          Close
140  166.899994
141  169.020004
142  169.889999
143  169.300003
144  173.500000
```

Interestingly, the stock's closing price on April 30, 2024, remained nearly unchanged compared to October 1, 2023. However, as we are about to discover, there were substantial fluctuations in the price over these seven months.

7.2.3 Plotting the data

Plotting time series data, particularly daily stock prices, is essential for visualizing trends, patterns, and fluctuations in financial markets. These plots provide valuable insights into the performance of stocks over time, helping analysts and investors identify key price movements and make informed decisions. By plotting daily stock prices, we can observe how prices evolve throughout each trading day, capture trends such as upward or downward movements, and detect potential support and resistance levels. Visualizations of daily stock prices also facilitate the comparison of historical performance, enabling stakeholders to assess volatility, trading volumes, and overall market sentiment. These visual representations serve as foundational tools in technical analysis and quantitative research, offering a clear and intuitive way to interpret complex market dynamics.

The following snippet of `matplotlib` code generates a time series chart displaying the closing prices of Apple's stock from October 1, 2023, through April 30, 2024 (see figure 7.1):

```
>>> import matplotlib.pyplot as plt
>>> plt.subplots()
>>> plt.plot(close_prices)
>>> plt.xlabel('Trading Days\n'
>>>     'October 1, 2023 through April 30, 2024')
>>> plt.ylabel('Closing Price (USD)')
>>> plt.title('Daily Closing Stock Price - AAPL')
>>> plt.grid()
>>> plt.show()
```

Library must be imported before running subsequent code

Initializes a new figure

Plots data stored in close_prices as a line chart by default

Sets the top line in the x-axis label; \n is the newline character and acts as a carriage return

Sets the bottom line in the x-axis label

Sets the y-axis label

Sets the title

Adds a grid; enhances visual clarity by displaying horizontal and vertical grid lines

Displays the plot

Although Apple's stock was priced at roughly $173 when the market closed on October 1, 2023, and again when the market closed on April 30, 2024, the price was highly volatile over these seven months, soaring to nearly $198 in December 2023 and plunging to $165 in April 2024. Such volatility is hardly unusual in financial markets, where stock prices frequently experience significant fluctuations due to various economic and company-specific factors. This inherent unpredictability makes it challenging to accurately forecast future stock prices when using such volatile prices as a baseline.

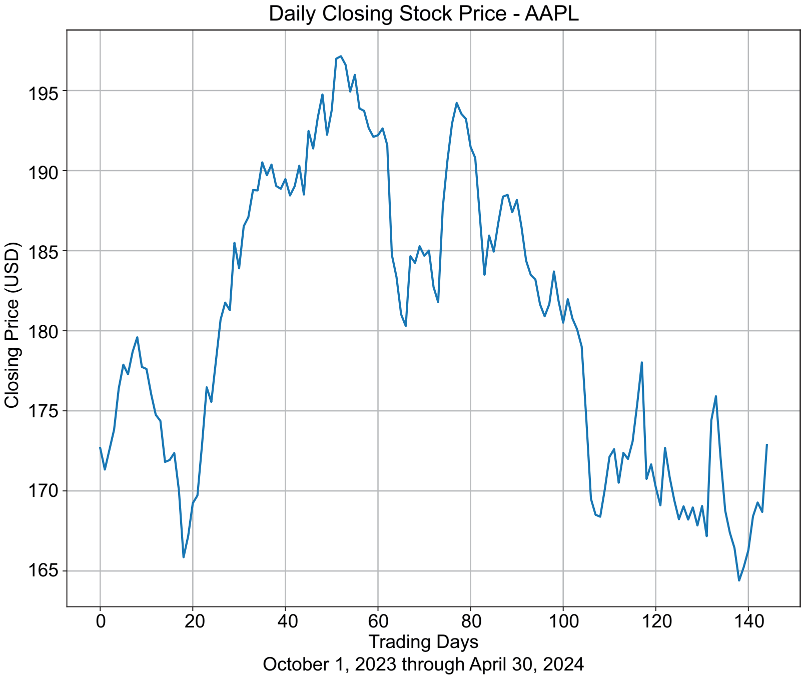

Figure 7.1 The daily closing price of Apple (AAPL) stock between October 1, 2023, and April 30, 2024. The more volatility in time series data, stock prices, and so on, the more challenging it is to fit an accurate forecast.

Nevertheless, we will fit an ARIMA model and then a series of exponential smoothing models to the first six months of data and then generate a forecast for the following month. This forecast will be compared to the actual stock prices to evaluate the accuracy and effectiveness of these time series models in predicting future price movements.

7.3 Fitting an ARIMA model

An ARIMA (autoregressive integrated moving average) model is a fundamental tool in time series analysis and forecasting, capable of capturing complex patterns and dynamics within sequential data. ARIMA models combine three key components—

autoregression (AR), integration (I), and moving averages (MAs)—to effectively model time-dependent data. Each component addresses different aspects of the time series, making ARIMA versatile for a wide range of applications in finance, economics, weather forecasting, and more. Next, we will define each component and briefly explain how they work together to make ARIMA a powerful forecasting tool.

7.3.1 Autoregression (AR) component

The AR part of ARIMA models the linear relationship between an observation and a specified number of its previous values, known as *lagged observations* or *autoregressive terms*. This component assesses how past values of the series influence its current value. For instance, in an $AR(1)$ model, the current value is regressed on its immediate past value. By incorporating these past observations, ARIMA can capture underlying trends and patterns in the data.

7.3.2 Integration (I) component

The I part signifies the differencing of the time series to make it stationary. Stationarity implies that the statistical properties of the series, such as mean and variance, do not change over time. Differencing involves subtracting the previous observation from the current observation, which removes trends or seasonal patterns that could affect the series' stability.

7.3.3 Moving average (MA) component

The MA component models the relationship between the current value of the series and random noise in the past. It captures short-term fluctuations that are not accounted for by the autoregression and differencing components. In an MA model, the current value is expressed as a linear combination of past error terms, helping to smooth out irregularities in the data.

7.3.4 Combining ARIMA components

Typically, the I component is addressed first in ARIMA modeling. This involves determining the order of differencing, d, required to make the time series stationary. Once the data has been differenced adequately, the subsequent steps involve identifying the AR and MA components, p and q, respectively. This sequential approach ensures that each component of the ARIMA model is appropriately specified to capture the underlying patterns and dynamics of the time series data. When fitting an ARIMA, the nonnegative integers assigned to p, d, and q are provided to the model to generate the forecast.

Before diving into finer details, it is prudent to introduce the concepts of stationarity and differencing. These concepts are foundational as they pave the way for understanding how to determine the appropriate degree of differencing, d, essential in preparing time series data for ARIMA modeling. Stationarity is crucial because it ensures that the statistical properties of the time series, such as mean and variance, remain consistent over time. Differencing, on the other hand, plays a key role in transforming nonstationary data into a stationary form by removing trends or seasonal patterns. Understanding these concepts sets a solid foundation for effectively applying ARIMA models in time series analysis and forecasting.

7.3.5 Stationarity

Stationarity is a fundamental concept in time series analysis, defining the behavior of a stochastic process where statistical properties remain constant over time. In simpler terms, a stationary time series is one whose mean, variance, and autocovariance (covariance between two time points) do not change over time. This stability allows for reliable forecasting and statistical inference.

Statistical properties

Stationarity in time series analysis entails specific properties that ensure the statistical characteristics of the data remain consistent over time, facilitating reliable modeling and forecasting.

First, a *constant mean* implies that the average value of the time series does not change across different time points. Mathematically, for any given time t, $E(Y_t)$, where E denotes the expected value or mean, remains the same. This property suggests that the central tendency of the data does not shift over time, providing a stable reference point for understanding the series' behavior.

Second, *constant variance* dictates that the variability or spread of the time series observations remains uniform across all time points. Formally, $Var(Y_t)$, representing the variance of $Y_{t,}$ does not depend on t. This characteristic ensures that the magnitude of fluctuations around the mean remains consistent, indicating that the data points scatter evenly around the mean without systematic changes in dispersion over time.

Third, *constant autocovariance* involves the relationship between the series' values at different time points. Autocovariance measures the covariance between two time points, t and s, denoted as $\mathrm{Cov}(Y_t, Y_\mathrm{s})$. Importantly, in a stationary time series, autocovariance only depends on the lag $|t - s|$, the absolute difference between t and s, not on the specific values of t and s themselves. This means the degree of dependence between observations remains consistent regardless of when the observations occur, reflecting a stable correlation structure over time.

Achieving stationarity typically involves identifying and addressing trends and seasonality within the data. *Trends* refer to long-term movements or patterns, whereas *seasonality* involves repeating patterns at regular intervals. Differencing is a common technique used to achieve stationarity by subtracting the previous observation from the current observation. This process removes trends or seasonal patterns, thereby stabilizing the mean and variance of the time series.

In summary, the properties of stationarity—constant mean, constant variance, and constant autocovariance—provide a foundational framework for understanding and analyzing time series data. These characteristics ensure that the statistical properties of the data remain consistent over time, facilitating accurate modeling, forecasting, and inference.

Visualizing nonstationary and stationary data in a quadrant of plots provides a clear comparison of their distinct characteristics in time series analysis (see figure 7.2). Nonstationary data typically exhibits trends, seasonality, or changing variances over time, making patterns less predictable and statistical inference challenging without proper preprocessing. In contrast, stationary data remains stable with constant statistical properties such as mean and variance, enabling more reliable modeling and forecasting. This visual comparison helps in understanding the effect of stationarity on data analysis and highlights the importance of techniques like differencing to achieve stationarity before applying models such as ARIMA for accurate predictions.

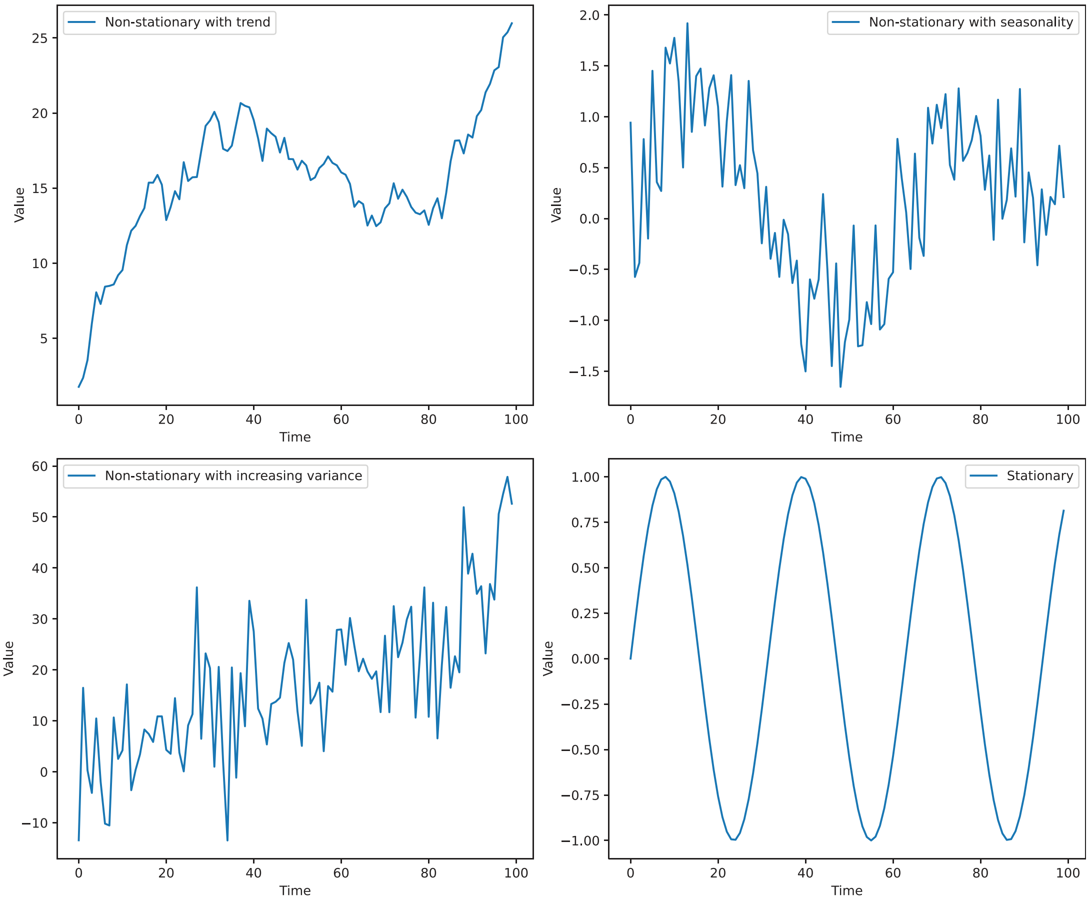

Figure 7.2 Four illustrative time series charts. Three of the subplots display nonstationary data; in contrast, the plot in the lower-right quadrant displays stationary time series data.

Three of the subplots display nonstationary data: one with an increasing trend, not unlike the closing price of Apple's stock during the last quarter of 2023; another with a seasonal pattern; and a third with increasing variance over time. The fourth subplot, in the lower-right quadrant, shows a stationary time series that oscillates around a constant mean.

7.3.6 Differencing

Differencing is a fundamental technique in time series analysis used to transform a nonstationary time series into a stationary one. Stationarity, characterized by constant statistical properties over time (such as mean, variance, and autocovariance), is crucial for many time series models like ARIMA, which assume stationary data for accurate forecasting and inference.

PURPOSE

The primary objective of differencing is to remove trends and seasonality present in the time series data. Trends represent long-term movements or shifts in the data, whereas seasonality refers to periodic fluctuations that occur at fixed intervals. Both trends and seasonality can introduce nonconstant mean and variance, making the data nonstationary and especially challenging from which to base accurate forecasts.

TYPES

There are two primary ways to differentiate nonstationary data: first-order differencing and seasonal differencing. First-order differencing involves subtracting each observation from its previous observation (or, subtracting the daily closing price of Apple stock from the previous day's closing price):

$$Y_t' = Y_t - Y_{t-1}$$

Here, once more, are the closing prices from the first five trading days of October 2023 (these were previously returned by calling the `head()` method):

[173.300247, 171.953735, 173.210495, 174.457260, 177.030579]

A first-order differenced series from five observations is derived by following these four steps:

1. Calculate the difference between the second and first closing prices:

$$171.953735 - 173.300247 = -1.346512$$

2. Calculate the difference between the third and second closing prices:

$$173.210495 - 171.953735 = 1.256760$$

3. Calculate the difference between the fourth and the third closing prices:

$$174.457260 - 173.210495 = 1.246765$$

4. Calculate the difference between the fifth and fourth closing prices:

$$177.030579 - 174.457260 = 2.573319$$

So, the first-order differenced series is

$$[-1.346512, 1.256760, 1.246765, 2.573319]$$

The record count inevitably drops by one because it's impossible, of course, to subtract from the first entry in a data series.

First-order differencing effectively transformed the time series by stabilizing its mean and other statistical properties, which are essential for accurate time series analysis. By subtracting each data point from its previous value, we removed the linear trend that was also present in the original data. The stabilized series is now (presumably) stationary, with constant variance and autocovariance, making it more suitable for modeling and forecasting purposes. This transformation ensures that the intrinsic dynamics of the data are captured more accurately, without the distorting effects of any trend.

However, first-order differencing does not always completely address nonstationarity, particularly when there are more complex trends. In such cases, second-order differencing is necessary, where the differencing process is applied twice:

$$Y_t'' = Y_t' - Y_{t-1}'$$

This additional step helps to further stabilize the series by removing any residual trends that first-order differencing could not entirely eliminate.

The second type of differencing, known as seasonal differencing, is particularly useful for time series data exhibiting regular, repeating patterns due to seasonal effects. This technique involves subtracting the value of a data point from the value at the same position in the previous season. For instance, in monthly data with yearly seasonality, the differencing period would be 12. Mathematically, if s represents the seasonal period, seasonal differencing is performed as follows:

$$Y_t' = Y_t - Y_{t-s}$$

Seasonal differencing helps to remove these repeating patterns, thereby stabilizing the mean and other statistical properties of the data, just like first-order differencing. By eliminating seasonal effects, the series becomes more stationary, making it suitable for accurate modeling and forecasting. This process is essential in domains like weather forecasting, where temperatures and precipitation patterns recur annually, or in tourism and hospitality, where travel and hotel bookings typically show seasonal peaks and troughs. A second order of seasonal differencing may sometimes be required if the first order of seasonal difference fails to stabilize the time series sufficiently.

The plot of Apple's daily closing stock price from October 1, 2023, to April 30, 2024, does not exhibit any clear seasonal patterns. Instead, the data shows significant volatility and general trends typical of stock price movements without any distinct periodicity. This lack of repeating cycles indicates that the series is nonseasonal. Given this characteristic, we will apply first-order differencing to the data, as necessary, to stabilize its statistical properties and remove any linear trends, rather than using seasonal differencing.

7.3.7 Stationarity and differencing applied

Of course, we are assuming Apple's daily closing stock price is nonstationary, based on our understanding of stationarity and our observations so far. However, rather than immediately applying first-order differencing, we will first demonstrate two methods to determine whether a time series is nonstationary.

For accurate forecasting of closing stock prices, it's crucial to partition the historical data into separate sets for training and testing. The initial six months of data will be used to train the forecasting model. Subsequently, the model's predictions for the next month will be plotted alongside the actual closing prices to assess its accuracy. So, instead of determining the stationarity (or nonstationarity) of the entire time series, we will assess it against just the training set.

The following line of code generates a Boolean mask (`bln_msk`) by evaluating the `close_prices` index. It selects all observations from our time series that satisfy the condition specified by the `len()` method, creating a distinct subset:

```
>>> bln_msk = (close_prices.index < len(close_prices) - 22)
```

This subset is assigned to a new data frame called `train`, which includes the closing prices from October 1, 2023, through March 31, 2024. The remaining 22 entries, corresponding to the trading days in April 2024, are assigned to another data frame called `test`:

```
>>> train = close_prices[bln_msk].copy()
>>> test = close_prices[~bln_msk].copy()
```

To reiterate, our ARIMA model will be fitted to the `train` subset, meaning all subsequent preprocessing steps should be, and will be, applied to this same subset.

Visual inspection

Visual inspection is typically the first method used in time series analysis to evaluate whether a time series is stationary or, alternatively, shows trends and seasonality. Among various techniques, examining autocorrelation function (ACF) and partial autocorrelation function (PACF) plots provides valuable insights into the temporal dependencies and structure of the data.

The ACF quantifies the correlation between a time series and its lagged values (previous observations) at various time lags k. It helps analysts understand how past values influence current and future observations. Mathematically, the ACF at lag k, denoted as ρ_k, is defined as

$$\rho_k = \frac{\text{Cov}(X_t, X_{t-k})}{\sqrt{\text{Var}\,(X_t) \times \text{Var}(X_{t-k}}}$$

Here's a breakdown of the components:

- The *covariance* ($\text{Cov}(X_tX_{t-k})$) measures how two random variables (X_t and X_{t-k}) change together. Specifically, it quantifies the extent to which X_t and X_{t-k} deviate from their respective means together. The formula for covariance at lag k is given by

$$\text{Cov}\,(X_t, X_{t-k}) = \frac{1}{T}\sum_{t=1}^{T-k}(X_t - \mu)(X_{t+k} - \mu)$$

  where T is the total number of observations and μ is the mean of the time series.
- The *variance* ($\text{Var}(X_t)$) quantifies the variances of X_t and $X_{t-k,}$ which measure the spread of the data points around their mean. Variance is defined as the average of the squared differences from the mean. The variance of the time series X_t is

$$\text{Var}\,(X_t) = \frac{1}{T}\sum_{t=1}^{T}(X_t - \mu)^2$$

To generate an ACF plot, autocovariances are computed for lags from 0 up to a maximum lag K, normalized by the square root of the variances, and plotted against lag k. The plot visually displays how each lag correlates with the current values in the time series, thereby providing a clear picture of the temporal dependencies within the data.

The PACF measures the direct relationship between observations separated by exactly k time units while adjusting for effects of the intervening lags. PACF at lag k, denoted as ϕkk, isolates the correlation between X_t and X_{t-k} by removing the influence of the intervening terms X_{t-1}, X_{t-2}, …, X_{t-k+1}

Mathematically, PACF at lag k is computed with the following equation:

$$\phi kk = \frac{\text{Cov}(X_t, X_{t-k}|X_{t-1}, X_{t-2}, \ldots X_{t-k+1})}{\text{Var}(X_t) \times \text{Var}(X_{t-k}|X_{t-1}, X_{t-2}, \ldots X_{t-k+1})}$$

Here's a breakdown of the individual components:

- The *conditional covariance* ($\text{Cov}(X_tX_{t-k}))|(X_{t-1}, X_{t-2}, \ldots, X_{t-k+1})$) measures how X_t and X_{t-1} change together while accounting for the influence of the intermediate lags X_{t-1}, X_{t-2}, …, X_{t-k+1}.
- The *conditional variance* ($\text{Var}(X_t)|\text{Var}(X_{t-1}, X_{t-2}, \ldots, X_{t-k+1})$) measures the spread of X_{t-k} values around their mean, conditioned on the intermediate lags X_{t-1}, X_{t-2}, …, X_{t-k+1}

To generate a PACF plot, these partial autocorrelations ϕkk are computed for lags from 0 up to a chosen maximum lag K and plotted against lag k. This plot highlights the lags where significant direct relationships exist between the time series values, providing insight into stationarity and nonstationarity, as well as the appropriate number of autoregressive terms to include in a time series model.

ACF PLOT

The ACF is plotted by calling the `plot_acf()` method from the `statsmodels` library, with `train` as our time series data:

```
>>> from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
>>> acf = plot_acf(train)
```

Python subsequently returns a Matplotlib figure (see figure 7.3).

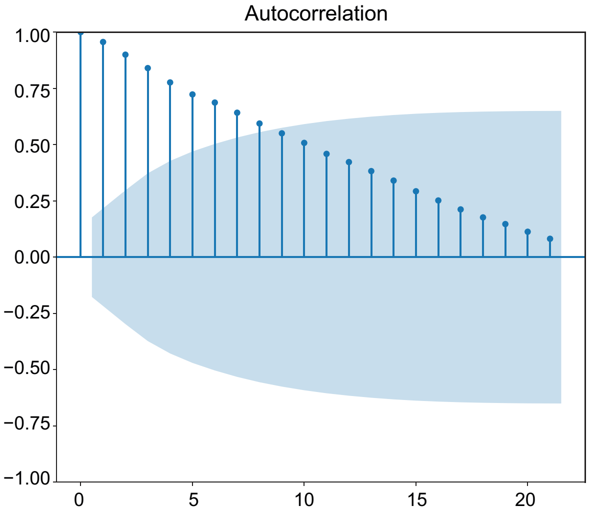

Figure 7.3 An ACF plot showing the correlation of a time series with its own past values (lags). The vertical lines represent the autocorrelation coefficients for different lags, with the shaded area indicating the confidence interval. Values outside this shaded region suggest significant autocorrelation, indicating a pattern in the data that could be used for time series forecasting. The gradual decline in the bars signifies the "memory" effect of the time series, where past values influence future observations.

The ACF plot displays the autocorrelation between the time series and its lagged values, providing insights into how past observations influence current and future values. (Autocorrelation measures how a time series relates to its own past values, whereas correlation typically measures the relationship between two distinct variables. Despite their differences in application, both autocorrelation and correlation are quantified in a similar manner and should be interpreted similarly.) On the horizontal axis (*x* axis), the plot shows different lags *K*, representing the number of time steps by which the series is shifted. The vertical axis (*y* axis) shows the correlation coefficients. These coefficients quantify the strength and direction of the linear relationship between the time series and its lagged versions.

Correlation coefficients always equal some number between −1 and +1:

- A coefficient of +1 indicates a perfect positive correlation: as the time series increases, the lagged values increase proportionally.
- A coefficient of −1 indicates a perfect negative relationship: as the time series increases, the lagged values decrease proportionally.

- A coefficient of 0 implies no linear relationship between the time series and its lagged values.

In an ACF plot, each bar represents the autocorrelation coefficient for a specific lag. The height and direction of the bar indicate the magnitude and sign of the correlation. The shaded area around the horizontal line at zero represents the confidence intervals, typically at a 95% confidence level. Correlations within this shaded area are not statistically significant, suggesting that observed correlations could be due to random noise rather than a genuine pattern in the data.

Interpreting an ACF plot involves assessing the decay of correlations over lags:

- Rapid decay in correlations suggests the time series may be stationary, with little dependence on past observations beyond a few lags.
- Persistent significant correlations across many lags indicate potential nonstationarity or the presence of long-term dependencies in the data.
- Peaks at regular intervals indicate seasonal patterns, with significant correlations repeating at those lags.

The ACF plot clearly indicates nonstationary data, characterized by significant correlations that persist over multiple lags, suggesting a lack of decay in autocorrelation and the presence of temporal dependencies within the series.

PACF PLOT

The PACF is plotted by making a call to the `plot_pacf()` method, also from the `statsmodels` library:

```
>>> pacf = plot_pacf(train)
```

Once more, Python returns a Matplotlib figure (see figure 7.4).

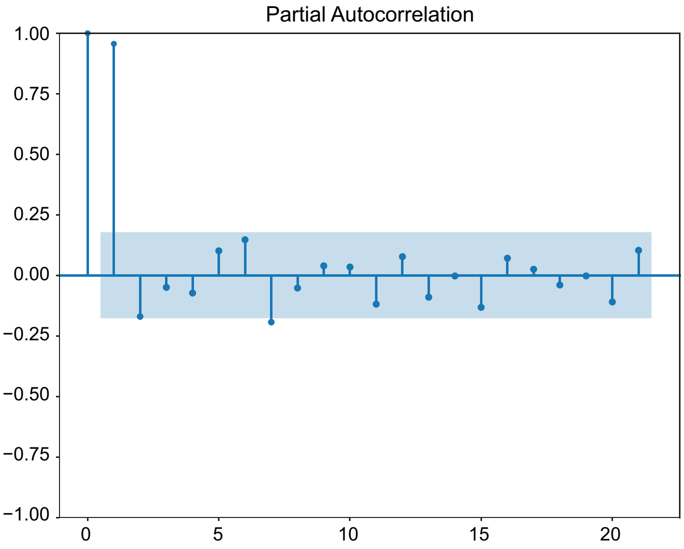

Figure 7.4 A PACF plot showing the correlation between a time series and its own past values (lags), controlling for the effects of earlier lags. The vertical lines represent the partial autocorrelation coefficients for each lag, with the shaded area indicating the confidence interval. Values outside this shaded region suggest significant partial autocorrelation, which can help identify the appropriate number of autoregressive terms in a time series model. The sharp drop after the first few lags indicates that only the first few lags have a significant direct effect on the current value of the series.

The PACF plot shows the direct relationship between a time series observation and its lagged values while accounting for the effects of intervening lags; it provides insights into how each lagged observation directly influences the current observation, adjusting for other lags in between. The *x* axis represents the number of time units, or lags, between the current observation and its lagged values. The *y* axis shows the partial autocorrelation coefficients, ranging from −1 to +1. These coefficients measure the strength and direction of the relationship between the current observation and its lagged values, after adjusting for the influence of intermediate lags.

Each bar in the PACF plot represents the partial autocorrelation coefficient for a specific lag. The height and direction of the bar indicate the magnitude and sign of the partial correlation. Similar to the ACF plot, the PACF plot typically includes a shaded region around the horizontal line at zero. This area represents the confidence intervals, typically at a 95% confidence level. Coefficients within this shaded area are statistically insignificant, indicating correlations that could arise due to random noise rather than meaningful patterns in the data.

High partial autocorrelation at lag 1 indicates a strong direct relationship between the current value and its immediate past value, which is common in both stationary and nonstationary series. The key is what happens at higher lags.

The fact that correlations from lag 2 onward fall within the shaded area implies these correlations are not statistically significant. This pattern suggests that the influence of past values diminishes quickly. The rapid decay in partial autocorrelation after lag 1, with subsequent lags showing insignificant correlations, typically suggests that the time series is stationary. In a stationary series, any dependence on past values tends to weaken rapidly.

In summary, whereas the ACF plot indicates nonstationarity in the time series, the PACF plot suggests the possibility of stationarity. To gain a definitive understanding, we will now pivot toward a more precise method by running a statistical test to determine whether the daily closing prices of Apple stock are stationary or nonstationary.

Augmented Dickey–Fuller test

The Augmented Dickey–Fuller (ADF) test is a widely used statistical test in time series analysis for determining whether a time series is stationary. The ADF test helps to identify a stochastic trend in the data that causes its statistical properties, such as mean and variance, to change over time, thereby indicating nonstationarity. One of the main advantages of the ADF test is its ability to handle complex time series with autocorrelation by including lagged terms. However, a limitation of the ADF test is its sensitivity to the chosen lag length, which can affect results, and its tendency to have lower power in detecting stationarity in short or noisy time series.

The null hypothesis of the ADF test is that the data is nonstationary. We will therefore require a p-value from the test below the typical 5% threshold to reject the null hypothesis and conclude the data is stationary. The test involves estimating the following regression model:

$$\Delta Y_t = \alpha + \beta t + \gamma Y_{t-1} + \sum_{i=1}^{p} \delta_i \Delta Y_{t-1} + \epsilon_t$$

where

- ΔY_t is the first difference of the time series Y_t
- α is a constant term (drift).
- β_t is a time trend.
- yY_{t-1} represents the lagged value of the time series.
- $\Sigma_{i=1}^{\mathrm{p}} \sigma_i \Delta Y_{t-1}$ includes the lagged differences of the time series to account for higher-order autoregressive processes.
- ϵ_t is the error term.

The key parameter in this model is γ (gamma). If γ is significantly different from zero, it suggests the data does not have a stochastic trend, leading to a low p-value and therefore a rejection of the null hypothesis. Alternatively, if γ is not significantly different from zero, it suggests the data does, in fact, contain a stochastic trend; the ADF test will return a p-value above the 5% threshold, meaning we should fail to reject the null hypothesis. By running the ADF test, analysts can make a more precise determination of the stationarity of a time series, complementing visual inspections from ACF and PACF plots.

The following snippet of Python code runs an ADF test on the `train` data frame and outputs the associated p-value:

```
>>> from statsmodels.tsa.stattools import adfuller
>>> adf_test = adfuller(train)
>>> print('p-value:', adf_test[1])
p-value: 0.5761116196186049
```

Because the p-value of the test is above 5%, we should fail to reject the null hypothesis and conclude that the daily closing prices of Apple stock, at least between October 1, 2023, and March 31, 2024, is nonstationary. This, of course, means we need to difference the data before fitting our ARIMA model.

First-order differencing

First-order differencing is a technique used in time series analysis to remove trends and make the data stationary by subtracting each observation from the previous one. This process helps stabilize the mean of a time series by eliminating changes in the level of the series.

In the next line of Python code, we use the `diff()` method to compute the difference between consecutive observations in the `train` data set, effectively applying first-order differencing. The `dropna()` method then removes any NaN values created by the differencing operation, resulting in a cleaned data series stored in a new object called `train_diff`:

```
>>> train_diff = train.diff().dropna()
```

Next, we plot the differenced data in a `matplotlib` time series chart (see figure 7.5):

```
>>> train_diff.plot()
>>> plt.xlabel('Trading Days:\n'
>>>     'October 1, 2023 through March 31, 2024')
>>> plt.ylabel('Differences in Closing Prices')
>>> plt.title('1st Order Differencing')
>>> plt.legend().set_visible(False)
>>> plt.show()
```

Initializes a line plot

Sets the x-axis label

Sets the y-axis label

Sets the title. Note that the sequence of labels and titles does not affect the plot.

Disables the legend

Displays the plot

After applying first-order differencing to the time series data, the resulting series exhibits characteristics indicative of stationarity. Specifically, the differenced data oscillates around a constant mean, which suggests that any underlying trend present in the original series has been effectively removed. This transformation stabilizes the mean of the series, making it more suitable for further time series analysis and modeling techniques that assume stationarity. The visual inspection of the differenced data plot confirms the absence of pronounced trends or varying levels, reinforcing the appearance of stationarity.

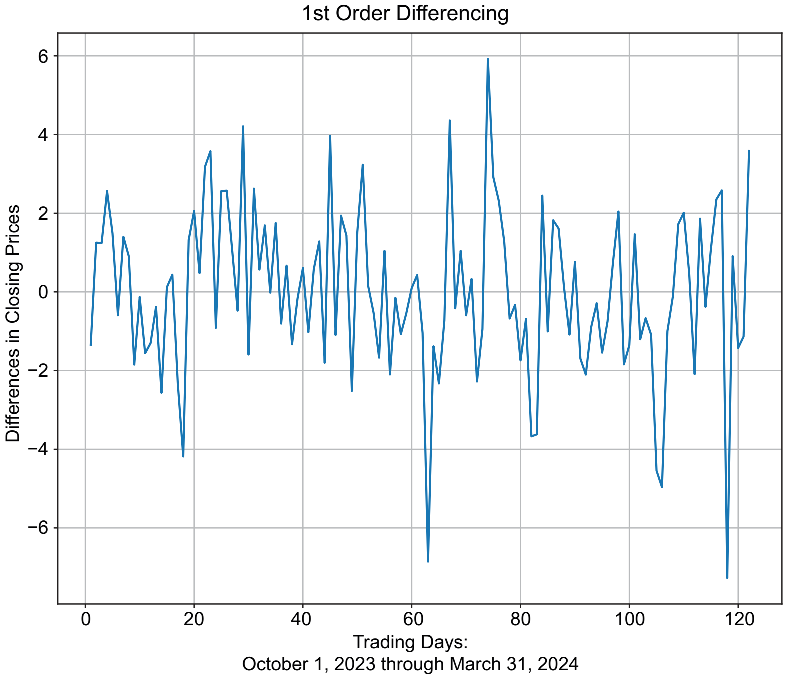

Figure 7.5 Time series data that was converted from nonstationary to stationary by first-order differencing

Although the time series now *appears* stationary, we should nevertheless create a new set of ACF and PACF plots and then conduct a second ADF test to confirm stationarity. The following snippet of `matplotlib` code creates ACF and PACF plots for the `train_diff` data frame and combines them with our previous ACF and PACF plots into a single figure:

```
>>> fig, axes = plt.subplots(2, 2)
>>> plot_acf(train, ax = axes[0, 0])
>>> axes[0, 0].set_title('ACF - Original')
>>> plot_pacf(train, ax = axes[0, 1])
>>> axes[0, 1].set_title('PACF - Original')
>>> plot_acf(train_diff, ax = axes[1, 0])
>>> axes[1, 0].set_title('ACF - Differenced')
>>> plot_pacf(train_diff, ax = axes[1, 1])
>>> axes[1, 1].set_title('PACF - Differenced')
>>> plt.tight_layout()
>>> plt.show()
```

Creates a 2 × 2 grid of subplots

Plots the original ACF plot in the top-left subplot

Sets title of the first ACF subplot

Plots the original PACF plot in the top-right subplot

Sets title of the first PACF subplot

Plots the differenced ACF plot in the bottom-left subplot

Sets the title of the second ACF subplot

Plots the differenced PACF plot in the bottom-right subplot

Sets the title of the second PACF subplot

Adjusts the layout to prevent overlapping titles and labels

Displays the plot

Figure 7.6 presents the autocorrelation and partial autocorrelation functions for both the original and differenced time series data. The top row displays the ACF (left) and

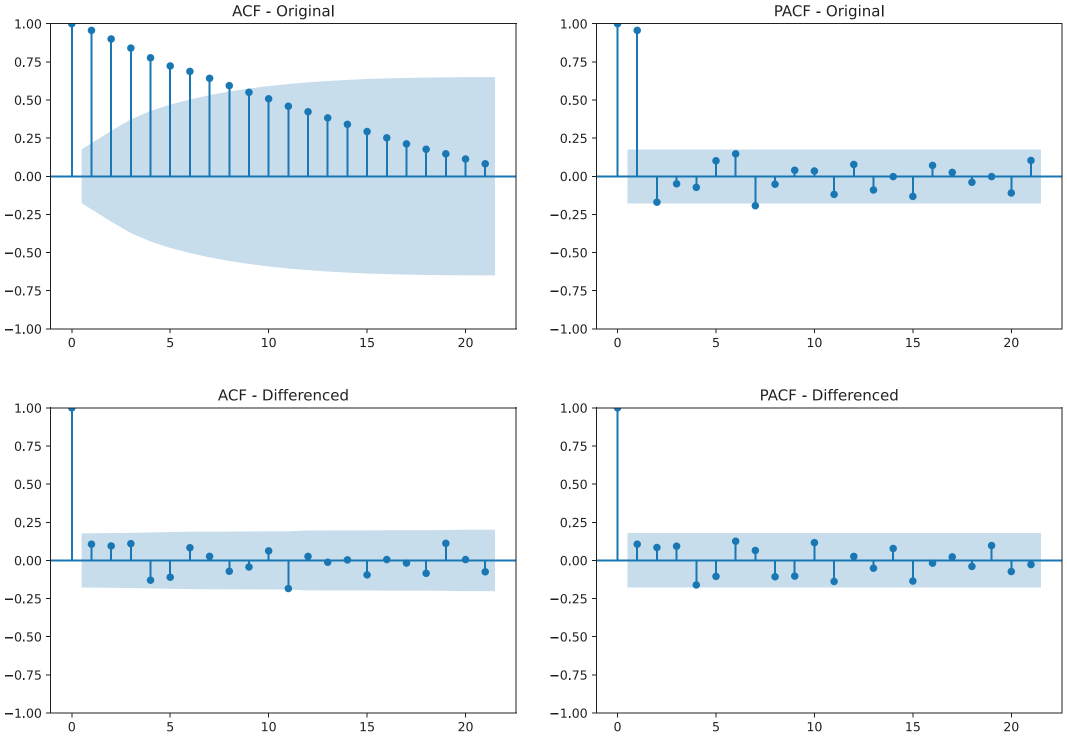

Figure 7.6 ACF and PACF plots for the original time series (top row) and first-order differenced time series (bottom row). The left column displays the ACF, showing how each observation is correlated with its previous values, whereas the right column presents the PACF, illustrating the direct effect of each lag. These plots are provided to compare the effect of first-order differencing on the time series' correlation structure.

PACF (right) for the original data, whereas the bottom row shows the ACF (left) and PACF (right) for the differenced data. The effects of first-order differencing are highlighted here, with both the original and differenced plots provided for easy comparison of how differencing affects the time series' autocorrelation structure.

After subjecting the data to first-order differencing, the resulting ACF and PACF plots reveal notable characteristics. Both plots exhibit a strong positive correlation at lag 0, which is expected due to the autocorrelation with itself. However, beyond lag 0, there are no statistically significant correlations observed in either plot. This lack of significant correlations suggests that the differenced data has effectively removed any systematic patterns or trends that were present in the original series. This transformation indicates that the differenced series may now be stationary, as evidenced by the absence of autocorrelation at lags other than 0.

Encouraging as these results are, let's verify the stationarity of the differenced time series through an ADF test to ensure reliability:

```
>>> adf_test_diff = adfuller(train_diff)
>>> print('p-value:', adf_test_diff[1])
p-value: 1.2486140415792743e-16
```

With a remarkably low p-value of 1.25×10^{-16} obtained from the ADF test, we confidently reject the null hypothesis of nonstationarity for the differenced time series. This strongly suggests that the differenced data is stationary, as there is substantial evidence against the presence of a stochastic trend.

ARIMA models, as a reminder, are structured into three components: autoregression (AR), integration (I), and moving average (MA). Each component corresponds to a parameter in the model notation ARIMA(p, d, q), where p represents the order of the AR component (AR(p)), d signifies the number of differencing operations needed to achieve stationarity (I(d)), and q denotes the order of the MA component (MA(q)). Having determined that d should be 1, reflecting the single differencing operation that rendered our initially nonstationary data stationary, our focus now shifts to identifying suitable values for p and q.

7.3.8 AR and MA components

Although we have established only one of the three necessary parameters for our ARIMA model, we have nevertheless completed roughly 90% of the groundwork. Determining the AR(p) and MA(q) components should be straightforward, as we have already identified the I(d) component to achieve stationarity in the time series data.

Autoregression component

The AR component in an ARIMA model captures the linear relationship between a time series and its own past values, reflecting the idea that future values of the series can be predicted using its own historical data. After creating ACF and PACF plots for the differenced time series data, we observe that there is a significant correlation only at lag 0 in both the ACF and PACF plots, whereas correlations at other lags are statistically insignificant.

In the ACF plot, a significant correlation at lag 0 indicates that each observation is highly correlated with its immediate predecessor, suggesting a strong dependence on the previous time point. This pattern is mirrored in the PACF plot, where a significant spike at lag 0 directly reflects this immediate correlation without additional significant spikes at subsequent lags.

Given this singular significant correlation at lag 0 in both plots, the appropriate choice for the AR component is likely AR(1). AR(1) implies that the current observation depends linearly on its immediate previous observation, aligning with the observed autocorrelation structure in the time series data. The lack of significant correlations at other lags reinforces the simplicity of the model, indicating that additional AR terms beyond AR(1) are unnecessary to explain the data's temporal dependencies.

Choosing AR(1) as the AR component ensures that the model captures the essential autocorrelation pattern without introducing unnecessary complexity. This decision is supported by both the theoretical underpinnings of ARIMA modeling and the empirical evidence provided by the ACF and PACF plots.

Moving average component

The MA component models the relationship between the current observation and the residual errors from a moving average model applied to lagged observations. After generating ACF and PACF plots for the differenced time series data, we observe that there are no significant correlations at any lags in the ACF plot beyond lag 0. This lack of significant correlations indicates that there is no remaining autocorrelation in the differenced series after accounting for the AR component.

Similarly, in the PACF plot, there are no significant spikes beyond lag 0, reinforcing the absence of direct correlations between the current observation and its lagged values. This pattern suggests that the observed variations in the time series are well explained by the immediate past values rather than by longer-term trends captured by a moving average process.

Given the absence of significant correlations in both the ACF and PACF plots, the appropriate choice for the MA component is likely MA(0). MA(0) implies that there is no residual autocorrelation in the series beyond the immediate lagged values, aligning with the stationary nature of the differenced time series data. This simplicity in the model specification avoids unnecessary complexity and ensures that the model focuses on the most relevant dependencies in the data.

Choosing MA(0) as the MA component is supported by the empirical evidence from the ACF and PACF plots, which indicate no significant autocorrelation at any lags not previously accounted for. This determination allows us to confidently specify the ARIMA(p, d, q) model with an MA component of MA(0).

Although our analysis supports the choice of AR(1) and MA(0), alternative configurations, such as AR(2) and MA(1), might be appropriate in cases where the ACF and PACF plots show significant correlations at additional lags. An AR(2) model, for instance, would suggest that the current observation depends not only on the previous value but also on the value two time steps prior, capturing a deeper dependency. Similarly, an MA(1) model would imply that residual errors from one previous time step influence the current observation, potentially capturing more complex noise patterns in the data. In this case, however, the simpler AR(1) and MA(0) specifications are well-suited to our observed autocorrelation structure.

When deciding between applying another differencing operation versus increasing the AR order, the key is to determine whether the data still exhibits nonstationarity. If the ACF shows a slow decay or persistent correlations across many lags, this may indicate the need for additional differencing. In contrast, if the data appears stationary but shows significant spikes at specific lags in the PACF, it's generally more appropriate to increase the AR order rather than differencing further.

7.3.9 Fitting the model

The following snippets of code demonstrate how to fit an ARIMA model to time series data using the `statsmodels` library. Fitting an ARIMA model, particularly when setting the AR component to 1, at least somewhat resembles fitting a simple linear regression, as both involve estimating parameters to minimize prediction errors based on historical data.

The first snippet imports the `ARIMA` class from the `statsmodels.tsa.arima.model` module:

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

The next snippet outlines the specifications for an ARIMA model fitted to the `train` subset of our original time series. The `order` parameter is a tuple that specifies the three key components of the model:

1. *Autoregressive order (*`p`*):* The first element in the `order` tuple is `p`, which is set to `1` in this instance. This parameter defines the number of lagged observations included in the model. By setting `p = 1`, we instruct the model to use one past value of the time series to predict the current value. This choice was based on our examination of the autocorrelation plots after the data was differenced and identifying significant versus insignificant lagged correlations, suggesting that one lag is sufficient to capture the dependencies in the data.
2. *Differencing Order (*`d`*):* The second element in the `order` tuple is `d`, which is also set to `1`. This parameter represents the number of times the data needs to be differenced to achieve stationarity. Setting `d = 1` means we are applying first-order differencing—subtracting each observation from the previous one—to stabilize the mean and other statistical properties of the series.
3. *Moving average order (*`q`*):* The third element in the `order` tuple is `q`, set to `0` in this case. This parameter indicates the number of lagged forecast errors included in the model. By setting `q = 0`, we specify that the model does not need to include any past forecast errors to predict future values. This decision was made based on the analysis of autocorrelation plots, where no significant cor-

relations are observed at higher lags, thereby suggesting that past errors do not add predictive power.

The train and order parameters are passed to the ARIMA class to create an instance of that class called model:

```
>>> model = ARIMA(train, order = (1, 1, 0))
```

In short, the preceding line of code creates an ARIMA model tailored to the characteristics of the train data frame, where

- p = 1 indicates using one past observation.
- d = 1 applies first-order differencing to ensure stationarity.
- q = 0 excludes past forecast errors from the algorithm.

After configuring the model with the appropriate parameters and providing it with the preprocessed training data, the next step is to fit the model. This process entails estimating the model's parameters based on the provided data. When the fit() method is called, the model begins adjusting its internal coefficients to minimize the difference between observed values in the training data and those predicted by the model. The fitting process primarily relies on maximum likelihood estimation (MLE), a statistical method used to find the parameters that maximize the likelihood of observing the given data under the model assumptions. This approach ensures that the ARIMA model effectively captures the temporal dependencies present in the differenced time series (train_diff), which include AR, I, and MA components. By iteratively optimizing these parameters, the model enhances its capability to forecast future values based on historical patterns discerned from the training data:

```
>>> model_fit = model.fit()
```

Once the fitting process is complete, model_fit encapsulates the trained ARIMA model with optimized parameters. This fitted model is then ready to be utilized to make predictions on new data or for further analysis and diagnostics. Overall, model.fit() represents the pivotal stage where theoretical model specifications are aligned with empirical data.

Finally, the line print(model_fit.summary()) outputs a summary table providing key statistical information—goodness of fit measures and parameter estimates, for instance—about the fitted model. The summary is crucial for evaluating the significance and effectiveness of the model in capturing the underlying patterns and dependencies of the time series data:

```
>>> print(model_fit.summary())
                               SARIMAX Results
==============================================================================
Dep. Variable:                  Close   No. Observations:                  123
Model:                 ARIMA(1, 1, 0)   Log Likelihood                -264.006
Date:                Thu, 20 Jun 2024   AIC                            532.013
Time:                        16:20:48   BIC                            537.621
Sample:                             0   HQIC                           534.291
```

```
                                  -123
Covariance Type:                   opg
==============================================================================
                  coef    std err          z       P>|z|       [0.025   0.975]
------------------------------------------------------------------------------
ar.L1           0.1084      0.091      1.196       0.232       -0.069    0.286
sigma2          4.4371      0.441     10.068       0.000        3.573    5.301
==============================================================================
Ljung-Box (L1) (Q):                     0.01   Jarque-Bera (JB):         12.14
Prob(Q):                                0.92   Prob(JB):                  0.00
Heteroskedasticity (H):                 1.57   Skew:                     -0.36
Prob(H) (two-sided):                    0.15   Kurtosis:                  4.37
==============================================================================
Warnings:
[1] Covariance matrix calculated using the outer product
of gradients (complex-step).
```

This concludes the model fitting process. Next, we will demonstrate how to interpret these results and evaluate the overall fit of our ARIMA model. After that, we will forecast April 2024 closing prices and plot them alongside the actual closing prices.

7.3.10 Evaluating model fit

The summary results are broadly divided into four sections: general information (upper left), goodness of fit (upper right), parameters (middle), and residuals (bottom).

General information

The general information section provides basic details about the model and the data used to train the model:

- `Dep. Variable`—The name of the variable containing the original time series data.
- `Model`—The order of the fitted model, or the nonnegative integers assigned to `p`, `d`, and `q`.
- `Sample`—The number of samples, or records, in the time series.
- `Covariance Type`—The method used to estimate the covariance matrix of the model's parameter estimates, which is crucial for assessing parameter uncertainty. In time series models, the `opg` method (short for outer product of gradients) is often used to calculate this matrix by approximating it based on the outer product of gradients from the likelihood function. This approach provides an efficient and consistent way to estimate parameter variances, which are essential for constructing confidence intervals and hypothesis tests on the model's parameters. Other covariance types may also be used depending on the model, each with different trade-offs in terms of computational efficiency and accuracy.

Goodness of fit

This section displays metrics that assess the model's overall fit to the data. They are typically used to compare and contrast competing models fit to the same data:

- `No. Observations`—Number of samples used to train the model.
- `Log Likelihood`—Measures how well the model's parameters explain the observed data, calculated based on the likelihood function. A higher log-likelihood value suggests that the model fits the data more closely, as it implies that the observed values are more probable under the model's parameters. Although this metric is useful for comparing models, it does not account for model complexity, which is where criteria like AIC, BIC, and HQIC come in.
- `AIC` (short for Akaike Information Criterion)—Balances model fit with simplicity by introducing a penalty for adding more parameters. It is calculated by taking twice the number of parameters in the model and subtracting this value from twice the log-likelihood. Lower AIC values indicate a better model, as they reflect both strong fit and parsimony. This criterion helps in selecting models that generalize well to new data, although it may lean toward more complex models in large data sets.
- `BIC` (short for Bayesian Information Criterion)—Like AIC, balances model fit and complexity, but applies a stronger penalty for the number of parameters, especially as sample size increases. It is calculated by taking the number of parameters, multiplying it by the natural logarithm of the sample size, and then subtracting this from twice the log-likelihood. Due to this stronger penalty for additional parameters, BIC tends to favor simpler models, making it particularly useful when avoiding overfitting, which is a priority. In model comparison, a lower BIC indicates a better model, as it suggests an optimal balance of fit and simplicity.
- `HQIC` (short for Hannan–Quinn Information Criterion)—Like AIC and BIC, evaluates the trade-off between model fit and complexity, with an emphasis on sample size. It is calculated by taking the number of parameters, multiplying it by twice the natural logarithm of the natural logarithm of the sample size, and then subtracting this from twice the log-likelihood. This penalty grows with sample size but at a slower rate than BIC, making HQIC particularly useful for smaller data sets, where it allows for a more balanced evaluation of model complexity. As with AIC and BIC, a lower HQIC value indicates a preferable model, reflecting an effective balance between fit and simplicity.

PARAMETERS

This section lists the estimated coefficients for the model's parameters:

- `ar.L1`—Represents the only parameter in an ARIMA(1, 1, 0) model. For an ARIMA model with $p = 1$ (one AR parameter), $d = 1$ (one order of differencing), and $q = 0$ (no MA parameters), the model equation is given as

  $$X_t = \phi_1 X_{t-1} + \epsilon_t$$

  where

  - X_t is the value of the time series at time t.
  - ϕ_1 is the coefficient of the AR parameter at lag 1.

  - X_{t-1} is the value of the time series at the previous time step.
  - ϵ_t is the error term at time t, assumed to be white noise.

- `coeff`—The estimated coefficient for the model's one parameter.
- `std err`—The standard error of the coefficient estimate, indicating the precision of the same. A lower standard error suggests that the estimate is more precise and likely to be closer to the true population value. In statistical terms, a standard error of 0.091 implies that the coefficient estimate could vary by approximately ±0.091 units around its approximate mean if the model were estimated repeatedly from different data samples.
- `z`—Represents the z-statistic for the coefficient, or the number of standard deviations by which the estimated coefficient differs from zero. In statistical terms, a z-statistic equal to 1.196 suggests that the coefficient is relatively close to zero and may not be statistically significant, as it does not exceed the conventional threshold of approximately 1.196 standard deviations (which corresponds to a 5% significance level).
- `P>|z|`—The p-value associated with the z-statistic, indicating the significance of the coefficient estimate. Because the p-value (0.232) is greater than the typical 5% threshold for significance, the coefficient may not have a material effect on the model's forecast.
- `[0.025 and 0.975]`—The 95% confidence interval for the coefficient estimate, showing the range within which the true coefficient value is likely to fall.
- `sigma2`—The estimated variance of the residual errors (white noise) in the model. More precisely, it represents the estimated variance of the residual errors left unexplained by the ARIMA model. It indicates how much the actual data points typically deviate from the predicted values. A higher `sigma2` implies larger fluctuations or errors in the predictions, whereas a lower `sigma2` suggests that the model fits the data more closely with smaller residual errors. Similar to measures of goodness of fit, `sigma2` is more appropriately used for comparative purposes rather than to assess an individual model in isolation.

Residuals

This section provides statistics and diagnostic test results evaluating the residuals—that is, the differences between observed and predicted values—to identify issues like autocorrelation and heteroskedasticity. Rather than reviewing these from the summary table, visual diagnostic methods are typically used for thorough assessment (see chapter 4).

The following `matplotlib` code snippet produces two side-by-side plots that depict the distribution of residuals. As in regression analysis, the Residuals plot should reveal no noticeable trend or pattern, indicating randomness and unbiased errors. Meanwhile, the Density plot is expected to exhibit a bell-shaped curve, signifying a normal distribution centered around a mean of zero (see figure 7.7):

```
>>> residuals = model_fit.resid[1:]
>>> fig, ax = plt.subplots(1, 2)
>>> residuals.plot(title = \
>>>                'Residuals', ax = ax[0])
>>> residuals.plot(title = \
>>>                'Density',
>>>                kind = 'kde',
>>>                ax = ax[1])
>>> plt.tight_layout()
>>> plt.show()
```

Extracts the residuals from the fitted ARIMA model (model_fit)

Creates a single figure containing two plots arranged in a 1 × 2 configuration

Sets the plot type and title of the first plot

Sets the plot type and title of the second plot

Adjusts subplot parameters, as necessary, to prevent overlapping elements

Displays the plots as a single figure

Both plots exhibit the expected characteristics.

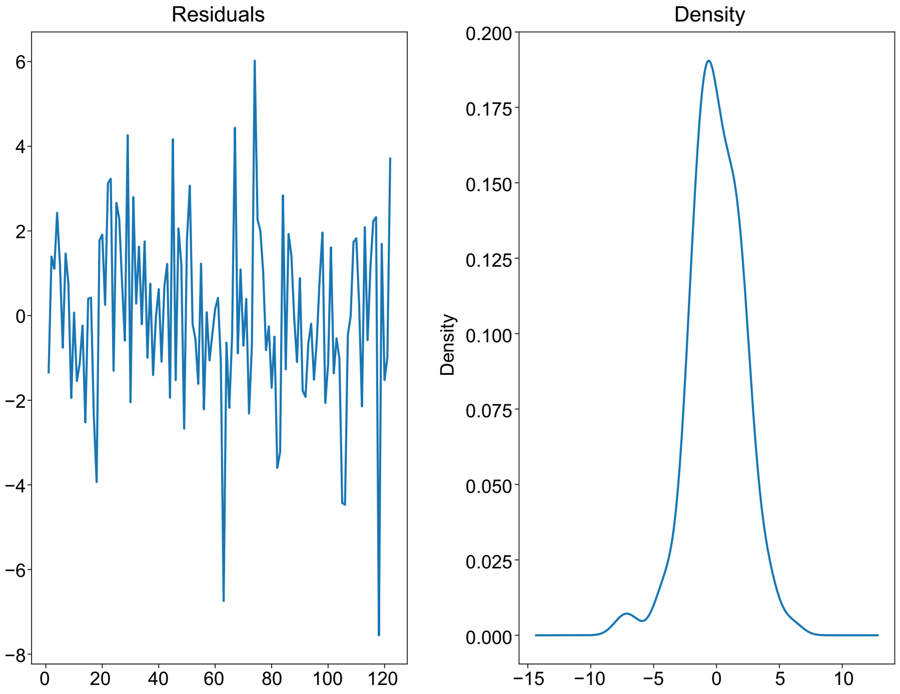

Figure 7.7 Left: The model residuals display no obvious pattern or trend. Right: The same residuals are normally distributed around a mean of zero.

When residuals show no obvious pattern or trend and are normally distributed, it suggests that the model adequately captures the underlying patterns in the data and that the assumptions of the model, such as constant variance and independence of errors, are likely met. Furthermore, examining the ACF and PACF plots of the residuals reveals that nearly every lag exhibits no statistically significant correlations, indicative of white noise. This reinforces confidence that our ARIMA model effectively captured the underlying data trends (see figure 7.8).

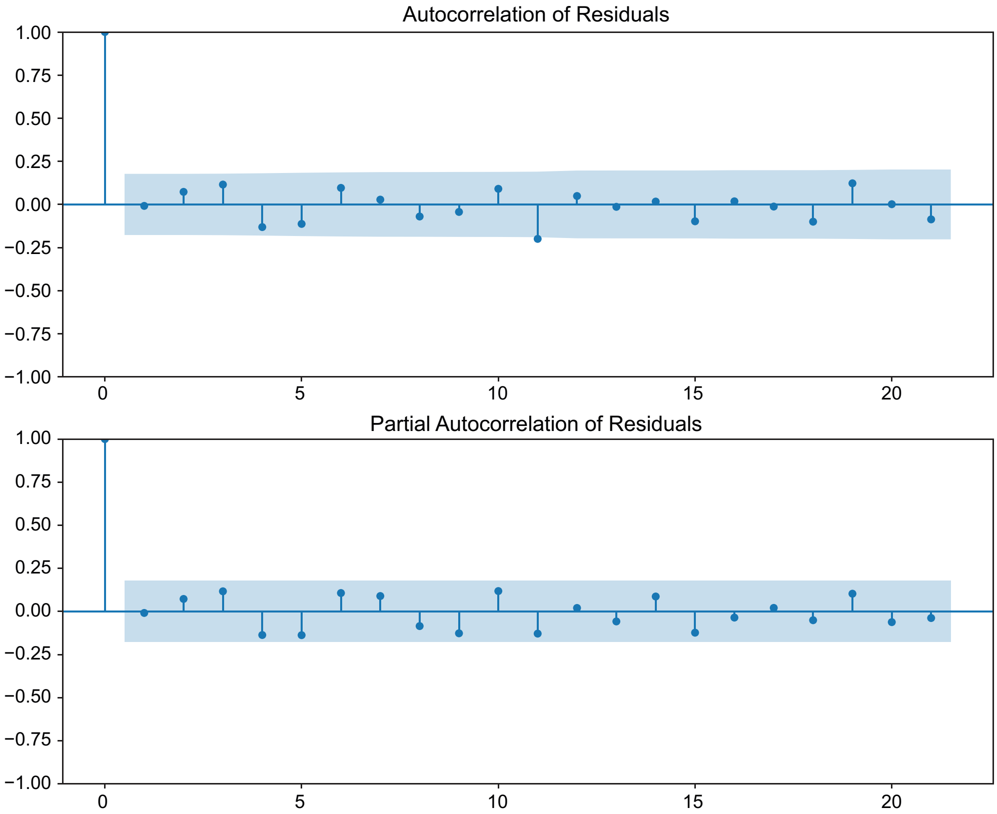

Figure 7.8 ACF plot (top) and PACF plot (bottom) of the model residuals. The lack of statistically significant correlations further suggests that the ARIMA model sufficiently captured trends in the time series.

Our evaluation of the ARIMA model provided mixed results: for instance, although the autoregressive parameter in the ARIMA(1, 1, 0) model has a p-value well above the standard 5% threshold for significance, the diagnostic tests on the model residuals were notably promising. Next, we will forecast the April 2024 closing prices using this model and visually compare the predicted values against the actual values to evaluate its predictive accuracy.

7.3.11 Forecasting

The next step in our analysis involves forecasting future values—April 2024 closing prices—from the data used to train our ARIMA model: closing prices from October 1, 2023, through March 31, 2024. This is achieved with the next line of code: `model_fit.forecast()` generates out-of-sample forecasts based on the length of the `test` data frame previously subset from the original time series. Specifically, it uses the estimated parameters of the ARIMA model to predict the values for the specified number of periods. This method is crucial for evaluating the model's predictive performance, as it provides a direct comparison between the forecasted values and the actual observed values in the `test` set. By assessing these predictions, we can better understand how well our model captures the underlying patterns in the data and its effectiveness in making future projections:

```
>>> forecast_test = model_fit.forecast(len(test))
```

After generating the forecast values, the next step is to incorporate these predictions into our `close_prices` data frame for comparison purposes. The next line of code appends a new variable to `close_prices` called `Forecast`. It populates `Forecast` with `NaN` for those trading days that correspond to the training period, and otherwise the forecasted closing prices forecasted by the model:

```
>>> close_prices['Forecast'] = [None] * len(train) + list(forecast_test)
```

When the data is plotted, we have a single time series from October 1, 2023 through March 31, 2024, and two time series for April 2024—the actual closing prices and the predicted closing prices:

```
>>> close_prices.plot()
>>> plt.xlabel('Trading Days:\n'
>>>    'October 1, 2023 through April 30, 2024')
>>> plt.ylabel('Differences in Closing Prices')
>>> plt.title('Actual vs. \
>>>            Forecasted Closing Price - AAPL')
>>> plt.grid()
>>> plt.show()
```

Generates a plot of all variables in the close_prices data

Sets the x-axis label

Sets the y-axis label

Sets the title

Adds a grid

Displays the plot

Figure 7.9 shows the result. The ARIMA model forecasted a daily closing price of approximately 173.51 for April 2024. The actual daily closing price during the same month, which experienced considerable volatility, averaged around 169.43. This discrepancy indicates that although the ARIMA model provided a reasonable estimate, it overpredicted the average closing price, albeit by a small margin. This result underscores the importance of accounting for both forecast accuracy and the inherent volatility of stock price movements. This flat forecast is a result of the ARIMA model structure, which—after differencing—tends to predict future values as a continuation of the most recent trend, leading to a constant forecast when no strong momentum or seasonal pattern is present in the training data.

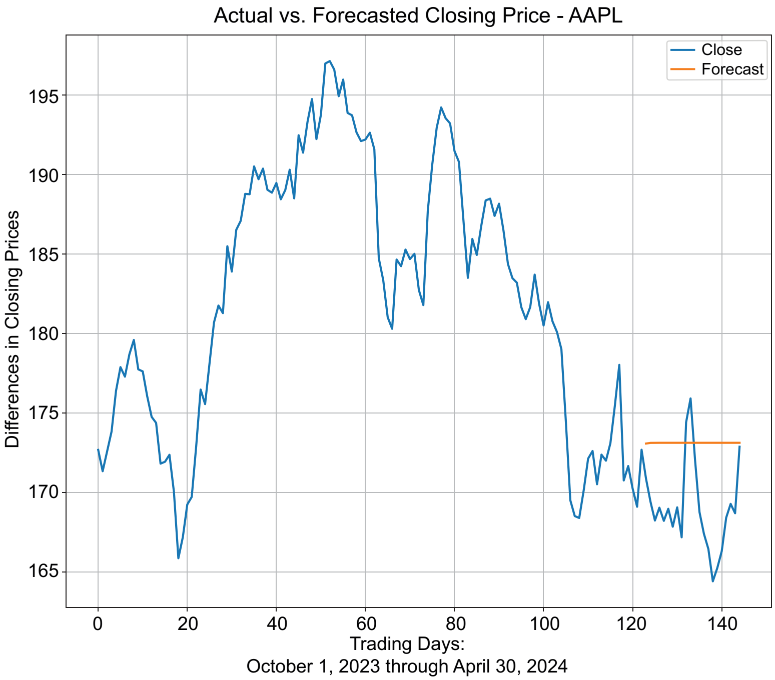

Figure 7.9 The actual closing prices of Apple stock from October 1, 2023, through April 30, 2024, plus the forecasted closing price of the stock throughout April 2024 generated from an ARIMA model.

As we've seen, the ARIMA model is a powerful tool for capturing various patterns in time series data, particularly when there are clear trends or seasonal components. However, although ARIMA is effective in certain scenarios, it might not always be the best fit for every type of time series, especially when the data exhibits smooth, gradual changes over time. This brings us to the next approach in our forecasting toolkit: exponential smoothing models. Unlike ARIMA, which focuses on autoregression and moving averages, exponential smoothing directly models the level, trend, and seasonality in the data, making it especially useful for time series that require a more responsive, adaptive approach to forecasting. Let's explore how exponential smoothing can be applied to our time series data.

7.4 Fitting exponential smoothing models

Exponential smoothing models are a class of forecasting techniques that apply weighting factors that decrease exponentially over time. Unlike ARIMA models, which focus on understanding the underlying data structure and stochastic nature of the time series, exponential smoothing models emphasize smoothing the data to make forecasts. They are particularly useful for time series where the most recent observations are more relevant for forecasting than older observations.

7.4.1 Model structure

Whereas ARIMA models rely heavily on the statistical properties of the data and typically require detailed analysis of autocorrelation and partial autocorrelation functions to identify the appropriate model parameters, exponential smoothing models—including simple, double, and Holt–Winters exponential smoothing—apply weighted averages of past observations with weights that decay exponentially. They do not explicitly model the underlying data structure but rather smooth the data to identify trends and seasonality.

7.4.2 Applicability

Although ARIMA models may be best suited for time series data that exhibits clear patterns over time, including trends and seasonality, and where the goal is to understand the data-generating process, exponential smoothing models are more effective for time series that are primarily concerned with making short-term forecasts where the most recent data points are (presumably) more indicative of future values. These models are less complex and often easier to implement.

7.4.3 Mathematical properties

Whereas ARIMA models incorporate differencing to make the time series stationary and use lagged values of the series and/or lagged forecast errors to train the model, exponential smoothing models use weighted averages of past observations, where weights decrease exponentially. For instance, the simple exponential smoothing model is

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

where α is the smoothing parameter between 0 and 1, y_t is the actual observation at time t, and $\hat{y}_{t+1}$ is the forecast for the next period.

7.4.4 Types of exponential smoothing models

As previously mentioned, there are three types of exponential smoothing models:

- Simple exponential smoothing (SES) is most suitable for forecasting time series with no clear trend or seasonal pattern. It works best for time series that are stationary around a constant mean.
- Double exponential smoothing (DES) extends SES to data with trends. It incorporates a trend component to account for linearity in the time series.
- Holt–Winters exponential smoothing (H-W) extends DES to handle seasonality and therefore works well with data that contains trends and a seasonal component. It comes in two variations: additive (for constant seasonal variation) and multiplicative (for changing seasonal variation).

We will demonstrate how to fit each type of exponential smoothing model in Python. We will then compare the Holt–Winters model to the ARIMA model by evaluating the AIC and BIC measures of both models. Finally, we will generate a forecast using the Holt–Winters model and plot the forecasted closing prices against the actual closing prices.

7.4.5 Choosing between ARIMA and exponential smoothing

Exponential smoothing models provide a simpler and often more intuitive approach to time series forecasting compared to ARIMA models. Although ARIMA models excel at capturing the underlying data-generating process and are powerful for long-term forecasting, exponential smoothing models are advantageous for their simplicity and effectiveness in short-term forecasting, especially when the most recent observations are more predictive of future values. By fitting both ARIMA and Holt–Winters exponential smoothing models, we can compare their forecasts and determine which method better suits our specific forecasting needs.

7.4.6 SES and DES models

The following code snippet demonstrates how to fit an SES model to the training data using the `statsmodels` library. This type of model is particularly suited for time series data without trend or seasonal components, where more recent observations are given exponentially greater weights:

```
>>> from statsmodels.tsa.holtwinters import SimpleExpSmoothing
>>> ses_model = SimpleExpSmoothing(train).fit()
```

Here, the `SimpleExpSmoothing` class is imported from `statsmodels`, and an SES model is fitted to `train` using the `fit()` method. This operation results in an SES model that assigns exponentially decreasing weights to older observations, effectively smoothing the data for better forecasting. The fitted model can then be used to make predictions based on the most recent patterns in the data.

The next code snippet demonstrates how to fit a DES model using the `statsmodels` library. This model, often referred to as *Holt's linear trend model*, is particularly useful for time series data that exhibit a trend component:

```
>>> from statsmodels.tsa.holtwinters import ExponentialSmoothing
>>> des_model = ExponentialSmoothing(train, trend = 'add').fit()
```

The `ExponentialSmoothing` class is imported from `statsmodels`, and an instance of the DES model is created with the `train` data set. The `trend = 'add'` parameter indicates that an additive trend component should be included in the model. By then calling the `fit()` method, the model parameters are optimized to best capture any underlying trend in the data.

The key difference between the DES model and the SES model lies in the ability to handle trends. Although the SES model focuses solely on smoothing the data without considering any trend, the DES model incorporates an additive trend component. This makes the DES model more suitable for time series where the data exhibits a linear trend over time.

Both the SES and DES models assign exponentially decreasing weights to older observations, giving more importance to recent data points. However, the DES model takes it a step further by smoothing the trend component, which helps capture the data dynamics better.

The SES model is ideal for time series without trends or seasonal patterns, providing a straightforward method for smoothing. In contrast, the DES model is designed for time series data with a trend, making it more robust for such data sets. This enhancement allows the DES model to deliver more accurate forecasts for data with an underlying trend.

7.4.7 Holt–Winters model

H-W, also known as triple exponential smoothing, builds on the foundations of SES and DES by incorporating a seasonal component in addition to the level and trend components. Whereas SES focuses on smoothing data without trends and DES adds the capability to model linear trends, H–W takes it a step further by addressing seasonal variations. This makes it particularly effective for time series data that exhibits both trend and seasonal patterns, such as monthly sales data or daily stock prices with recurring cycles. By simultaneously modeling level, trend, and seasonality, H–W provides a comprehensive framework for more accurate and insightful forecasting.

MODEL FITTING

The following snippet of code utilizes the H–W method, advancing from DES by integrating seasonal patterns into the model. It employs the `ExponentialSmoothing` class from `statsmodels` to fit a Holt–Winters model to the training data set, incorporating both trend and seasonal components in the process. The `'add'` parameter for `trend` specifies an additive trend component, whereas `'add'` for `seasonal` indicates an additive seasonal component. With `seasonal_periods = 5`, the model assumes recurring seasonal cycles within the six-month training period, anticipating a pattern every five days:

```
>>> hw_model = ExponentialSmoothing(train,
>>>                                  trend = 'add',
>>>                                  seasonal = 'add',
>>>                                  seasonal_periods = 5).fit()
```

This enhancement over DES extends its capability to forecast data featuring regular seasonal fluctuations, offering a more comprehensive approach to time series forecasting.

MODEL EVALUATION AND COMPARISON

Rather than generating a detailed summary table akin to our ARIMA model approach, our focus now shifts to retrieving and printing just the AIC and BIC measures for our Holt–Winters model. This streamlined approach allows us to swiftly compare these goodness-of-fit metrics with those from our ARIMA model, aiding in the assessment of which model better fits the underlying patterns within the data:

```
>>> print(f"AIC: {hw_model.aic}")
AIC: 198.23435477953498

>>> print(f"BIC: {hw_model.bic}")
BIC: 223.54401397788672
```

On comparing the AIC and BIC values between our Holt–Winters and ARIMA models, it becomes evident that both metrics are notably lower for the Holt–Winters model. This outcome is indicative of the Holt–Winters model providing a better fit to the data than the ARIMA model. Lower AIC and BIC values suggest that the Holt–Winters model not only captures the underlying patterns and trends in the time series data more effectively but also demonstrates superior predictive performance. This model comparison perspective underscores the potential advantages of utilizing the Holt–Winters approach, particularly when dealing with time series data that exhibit seasonal variations and trends.

Forecasting

A forecast of April 2024 closing prices is generated just as before, except that we swap out the pointer to our ARIMA model (`model_fit`) in favor of our Holt–Winters model (`hw_model`). The forecast is then plotted against the actual closing prices by repurposing the same snippet of `matplotlib` code (see figure 7.10).

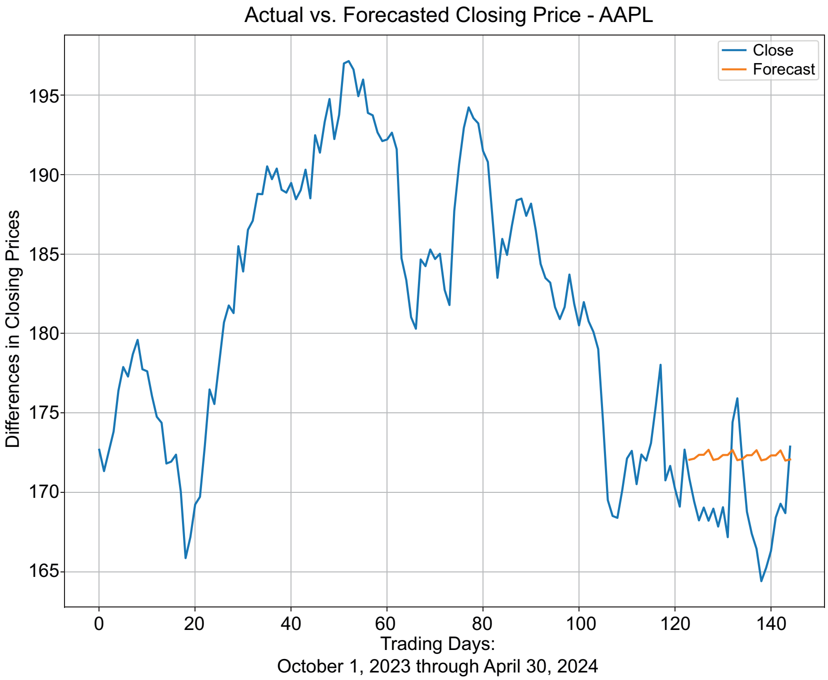

Figure 7.10 The actual closing prices of Apple stock from October 1, 2023, through April 30, 2024, plus the forecasted closing price of the stock throughout April 2024 generated from a Holt–Winters exponential smoothing model. The H–W forecast is slightly more accurate than the forecast from our ARIMA model.

The H–W model predicted a daily closing price ranging from about 172.39 to 173.06, with an average of approximately 172.66. In contrast, the ARIMA model forecasted a constant closing price of 173.51. This indicates that the Holt–Winters forecast is slightly more accurate than the ARIMA forecast. Although exponential smoothing models are generally easier to fit than ARIMA models, their predictive accuracy may vary. Therefore, organizations often adopt a strategy of fitting multiple time series models using different methods and combining their forecasts to derive a final prediction.

Disclaimer about time series models

Time series models offer powerful tools for analyzing data that changes over time, allowing us to forecast trends, seasonal patterns, and cyclic behaviors with a structured approach. They're particularly valuable in fields like finance, economics, and inventory management, where understanding future trends can drive critical decisions. For example, models such as ARIMA and exponential smoothing capture both short- and long-term dependencies, making them adaptable to various applications. Time series models can also handle temporal autocorrelation, helping to produce more accurate forecasts when the order of observations matters.

However, time series models have limitations. They rely on historical patterns to predict future values, which may not always hold true, especially in volatile or unpredictable markets. Sudden changes or external shocks, like economic crises or regulatory changes, can reduce their accuracy. Additionally, time series models assume stationarity or stable statistical properties, which can require extensive data transformation. It's essential to validate and test these models carefully, especially when using them in high-stakes scenarios. Users should remember that forecasts are probabilistic, not certain, and should be complemented with other analytical methods for a balanced perspective.

Throughout this journey, we've explored a wide array of machine learning and time series models, each offering unique strengths for different types of data and predictive challenges. From the foundational techniques of linear and logistic regression, which allowed us to model relationships and probabilities, to the more complex decision trees and random forests, which excel at capturing nonlinear patterns and interactions, we've built a solid foundation in predictive modeling. We then ventured into time series analysis, applying ARIMA models to uncover trends and seasonality, and exponential smoothing to create responsive forecasts for dynamic data. These comprehensive analyses have equipped us with a versatile toolkit for tackling a wide range of predictive tasks. As we transition from model fitting to linear programming, and specifically to constrained optimization, we shift our focus to optimizing decision-making processes, using the powerful insights gained from our modeling efforts to drive effective resource allocation and strategic planning.

Summary

- A time series model is a statistical tool used to understand and predict the behavior of data points indexed by time. It analyzes patterns and trends within sequential data, aiming to capture dependencies and variations over time. Time series models typically account for seasonality, trends, and irregular fluctuations in data, making them essential for forecasting future values or understanding historical patterns.
- ARIMA (autoregressive integrated moving average) is a popular time series forecasting model that combines autoregressive (AR), differencing (I), and moving average (MA) components. ARIMA models are versatile for handling a wide range of time series data by capturing their temporal structure, seasonality, and trend. The AR component models the relationship between an observation and a lagged value, whereas the MA component models the dependency between an observation and a residual error from a moving average model. The differencing component handles nonstationary data by transforming it into a stationary series.
- Exponential smoothing models do not require as much preprocessing as ARIMA models because they primarily focus on smoothing past data to make forecasts. They do not involve complex parameter determination through differencing or identifying autoregressive and moving average components, as ARIMA models do. This simplicity in preprocessing makes exponential smoothing models easier and quicker to implement for forecasting time series data with less historical analysis and adjustment.
- Simple exponential smoothing is a forecasting technique that assigns exponentially decreasing weights to past observations. It is suitable for time series data without trends or seasonal patterns. SES is characterized by its reliance on a single smoothing factor, which controls the rate of decay of older observations. Despite its simplicity, SES can provide effective short-term forecasts by emphasizing recent data over historical values.
- Double exponential smoothing extends SES by incorporating a trend component into the forecasting process. In addition to the smoothing parameter for level smoothing, DES introduces a trend smoothing parameter. This model is suitable for time series data exhibiting a trend but no seasonal pattern. DES forecasts are influenced by both recent observations and the trend observed in previous periods, making it more robust than SES for data with a linear trend.
- Holt–Winters exponential smoothing extends double exponential smoothing by adding a seasonal component to handle time series data with seasonal variations. It therefore includes three smoothing parameters: level smoothing, trend smoothing, and seasonal smoothing. This model is effective for forecasting data with both trend and seasonal patterns, providing a flexible approach to capture and forecast seasonal variations in time series data.