‹packt›



2ND EDITION

Hands-On Time Series Analysis with R

Build accurate forecasting models using ARIMA, Prophet, TBATS, Bayesian methods, and deep learning

DIOGO ALVES DE RESENDE | SHUEN MEI

Hands-On Time Series Analysis with R

Second Edition

Build accurate forecasting models using ARIMA, Prophet, TBATS, Bayesian methods, and deep learning

Diogo Alves de Resende

Shuen Mei

‹packt›

Hands-On Time Series Analysis with R

Second Edition

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.
Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Sunith Shetty
Relationship Lead: Nilesh Kowadkar
Project Manager: Shashank Desai
Content Engineer: David Sugarman
Technical Editor: Shweta Amale
Copy Editor: David Sugarman
Indexer: Pratik Shirodkar
Proofreader: David Sugarman
Production Designer: Salma Patel
Growth Lead: Anjitha Murali

First published: May 2019
Second edition: September 2026

Production reference: 1100926

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83620-331-5
www.packt.com

Contributors

About the authors

Diogo Alves de Resende is the founder of The Data Hero and Bitte, with deep expertise in time series and commercial analytics. Formerly a senior commercial analyst at Zalando, he has implemented forecasting models for demand planning and return rates. As a popular Udemy instructor with over 20,000 students, Diogo brings his educational background and entrepreneurial experience into accessible, real-world teaching. He holds an MSc in Management and continues to blend technical insight with applied business strategy.

Shuen Mei is a senior technology executive with more than 25 years of experience in the financial services industry. He has designed and delivered large-scale, mission-critical systems supporting equities, fixed income, ETFs, foreign exchange (FX), credit, derivatives, trading platforms, order management systems, settlement, and clearing operations. Shuen holds a Master of Science degree in Management Information Systems (MIS) from the University of Illinois and maintains professional certifications in AWS, Databricks, Apache Spark, Cloudera, and HBase. He specializes in cloud-native architectures, real-time data platforms, machine learning, and artificial intelligence. Passionate about leveraging emerging technologies to solve complex business challenges, Shuen continues to drive innovation at the intersection of data, AI, software engineering, and financial technology.

About the reviewers

Ankush Singal is a Senior Data Scientist, AI Architect, and technology leader with over 15 years of experience in data science, machine learning, data engineering, and enterprise analytics. His expertise includes time-series forecasting, machine learning, distributed computing, data engineering, Generative AI, LLM-powered applications, and cloud-native analytics platforms. He has worked with leading organizations including Northrop Grumman, the FBI, Booz Allen Hamilton, and large enterprise clients, delivering advanced analytics and AI solutions at scale.

He is passionate about helping practitioners bridge the gap between statistical theory and real-world implementation, with a focus on scalable, production-ready analytics solutions.

Eitbar Aliyev is an AI architect and data scientist with over a decade of experience in machine learning, large language models, and analytics. He is Head of AI at Nucleus Labs AI, where he owns end-to-end AI architecture across model serving, retrieval, and evaluation. He previously led LLM research and evaluation at Google (via Turing), built marketing mix models at WPP, and developed time series forecasting systems at Volvo Group using LSTM, Neural Prophet, and classical statistical methods. He holds an M.S. in Informatics and Analytics from UNC Greensboro and is a Senior Member of the IEEE. He thanks the Packt team for the opportunity to contribute to this book.

Table of Contents

Preface xv

Free benefits with your book xxi

Chapter 1: Getting Started with the Basics of R 1

Technical requirements 2

Introducing R 2

Introducing RStudio • 3

Installing R and RStudio • 4

Installing R • 4

Installing RStudio • 5

Setting up your workspace • 6

Tips for success • 7

Understanding fundamental R syntax for programming 7

The basics of R syntax • 7

Simple operations • 8

Function calls • 8

Exploring various data types and structures in R 8

Exploring data structures in R • 9

Code examples • 9

Mastering basic operations and functions for data manipulation 11

Arithmetic operations • 11

Basic functions for data analysis • 13

Summary 14

Chapter 2: Understanding Time Series Data 17

Technical requirements 18

Defining time series data 18

Exploring the components of time series data 20

Trends • 23

Seasonality • 24

Cyclical changes • 25
Noise • 26
Autocorrelation • 27
Types of time series data • 27
Univariate time series • 27
Multivariate time series • 28
Continuous and discrete time series • 28
Challenges in time series data analysis • 29
Dealing with missing data • 29
Dynamic and evolving data • 30
Managing noise • 31
Handling time series data in R 31
Creating a time series object in R • 31
Using zoo and xts packages • 34
Transitioning to xts • 35
Plotting with zoo • 36
Plotting with xts • 36
Learning fundamental operations specific to time series data 37
Time indexing and slicing • 38
Time indexing • 38
Slicing data ranges • 39
Frequency conversion and resampling • 40
Frequency conversion • 40
Resampling • 40
Summary 42

Chapter 3: Working with Dates and Times 43

Technical requirements 43
Understanding R's date-time classes 44
Introducing date-time classes in R • 44
The Date class • 45
Common functions used with Date objects • 45
Practical examples of creating and manipulating date objects • 45
The POSIXct and POSIXlt classes • 46
POSIXct is suitable for arithmetic operations and large datasets • 46

POSIXlt offers handling of hours, minutes, and seconds • 47
Converting between POSIXct and POSIXlt • 47
Handling time zones • 48
Formatting dates and times • 50
Using the format() function • 50
Customizing format to include or exclude components • 51
Parsing date-time strings into R objects • 51
Using as.Date() and as.POSIXct() • 51
Handling common parsing issues • 53
Advanced formatting and parsing techniques • 53
Best practices in handling date-time data • 55
Using the right R date-time classes • 55
Consistent format and documentation • 55
Establishing enterprise standards • 56
Common issues and error checks • 57
Performing operations specific to temporal data..... 58
Time series arithmetic • 59
Calculating the daily change in stock price • 59
Adjusting time series data for inflation or other external factors • 60
Window functions and rolling calculations • 61
Time shifts and date adjustments • 63
Practical examples of manipulating time series data 66
Data cleaning and normalization • 66
Feature engineering • 69
Summary 71

Chapter 4: Visualizing Time Series Data 73

Technical requirements..... 73
Overview of plotting in R 74
Plotting time series data • 75
Enhancing basic plots • 77
Using ggplot2 for time series visualization..... 81
Installation and setup of ggplot2 • 82
Creating advanced time series plots • 83
Advanced plot types • 84

Customizing plots with ggplot2 • 89
Interactive visualizations with shiny and plotly 97
Tools for interactive visualization • 98
Building dynamic plots with shiny • 99
Getting started with shiny • 99
Introducing the architecture of a shiny application • 99
UI • 99
Server logic • 100
Creating a basic shiny app • 101
Advanced features in shiny • 103
Reactive expressions • 103
Observer functions • 106
Enhancing interactivity with plotly • 106
Overview of plotly syntax and functions • 106
Creating a time series plot with plotly • 107
Creating interactive time series plots with plotly • 108
Common pitfalls in visualizing time series..... 109
Identifying common errors • 109
Overplotting • 109
Misleading scales • 110
Ignoring seasonality • 110
Failing to highlight anomalies • 110
Summary 111

Chapter 5: Exploratory Data Analysis for Time Series 113

Technical requirements..... 114
Trend identification and analysis..... 114
Methods for detecting trends • 115
Visual inspection • 115
Statistical tests • 118
Understanding smoothing techniques..... 120
Moving averages • 120
Simple moving average • 121
Exponential moving average (EMA) • 122
Weighted moving average (WMA) • 124

Advanced smoothing techniques • 125
Applying LOESS in R • 125
Comparing smoothing techniques • 127
Analyzing smoothed data • 128
Understanding seasonal decomposition 132
Introducing seasonal decomposition • 132
Implementing seasonal decomposition in R • 133
Using decompose() and stl() functions • 133
Evaluating the significance of the seasonal component • 135
Autocorrelation analysis 138
Impact of autocorrelation on statistical inference and model assumptions • 139
Detecting autocorrelation • 139
Lag plots • 140
Autocorrelation function (ACF) • 141
Partial Autocorrelation Function (PACF) • 142
Detecting anomalies 143
Visual techniques • 144
Statistical techniques • 145
Using z-scores • 145
Using the interquartile range (IQR) • 146
Automated methods – Twitter anomaly detection libraries • 147
Summary 148

Chapter 6: Exponential Smoothing and Holt-Winters 151

Technical requirements 151
Customer support case study 152
Training and test in time series • 159
Evaluating your model • 161
Simple exponential smoothing 162
Applying simple exponential smoothing to our data • 163
Double exponential smoothing 165
Putting double exponential into practice • 166
Triple exponential smoothing (also known as Holt-Winters) 168
Alpha, Beta, Gamma • 169
Practical applications • 169

Predicting the future period • 170
Model Assessment • 171
Handling daily data • 172
Summary **175**

Chapter 7: The ARIMA Model Family 177

Technical requirements **178**
Introducing the case study **178**
Introducing ARIMA **187**
Autoregressive (AR) • 188
Integrated (I) • 189
Stationarity and non-stationary visualization • 190
ADF test • 192
Moving average (MA) • 195
ARIMA in R • 196
AIC and BIC • 200
Exploring seasonality with SARIMA **201**
SARIMA in R • 202
Going beyond seasonality with SARIMAX **204**
The X factor – Exogenous variables • 205
SARIMAX in R • 205
Cross-validation • 208
Cross-validation in R • 210
Parameter tuning • 212
Parameter tuning in R • 213
Summary **217**

Chapter 8: Forecasting with TBATS Modeling 219

Technical requirements **220**
Overview of TBATS models **220**
Trigonometric seasonality • 221
Box-Cox transformation • 222
ARMA errors • 222
Trend • 223
Seasonal components • 224

Implementing TBATS in R 224
Introducing the AirPassengers dataset • 225
TBATS forecasting in R • 225
Applying TBATS to multiple seasonal time series 234
Introducing the Taylor electricity demand dataset • 234
Exploring multiple seasonal patterns • 235
Fitting the baseline TBATS model • 237
Understanding the fitted model • 239
T – Trigonometric seasonalities (multiple) • 239
B – Box-Cox transformation • 239
A – ARMA errors • 239
T – Trend • 240
S – Seasonal components • 240
Other parameters • 240
Forecasting with the default TBATS model • 241
Evaluating alternative model configurations • 241
Summary 246

Chapter 9: Bayesian Approaches in Time Series Analysis 249

Technical requirements 250
Foundations of Bayesian time series analysis 250
Prior distributions • 250
Likelihood and posterior distributions • 251
Bayes' theorem • 251
Practical example in R • 253
Analyzing posterior • 256
Bayesian inference in a time series context • 257
Second practical example in R • 258
Bayesian hierarchical models • 259
Third practical example in R • 260
Constructing a Bayesian model 262
Fourth practical example in R • 262
Choosing priors for time series models • 264
Choosing priors in R • 265
Model specification and diagnostics • 267

Practical example of model specifications and diagnostics in R • 268
Implementing Bayesian models in time series 270
Model selection for time series data • 270
Criteria for model selection • 271
Specifying model components • 272
Practical example in R • 273
Model validation and performance evaluation • 275
Residual plots • 277
Sensitivity analysis • 278
Summary 280

Chapter 10: Multi-Series Forecasting Techniques 283

Technical requirements..... 284
Introducing multi-series forecasting 284
Use cases of multi-series forecasting across industries • 285
Challenges in multi-series forecasting • 287
Data management complexities • 287
Varying data quality across series • 287
Computational demands • 288
Hierarchical forecasting methods 288
Introducing hierarchical forecasting • 288
Practical example – Aggregating and disaggregating data • 289
Bottom-up approach • 291
Practical example – Summing individual forecasts to create aggregated forecasts • 291
Middle-out approach • 293
Reconciliation and consistency in hierarchical forecasting • 295
Grouped time series analysis..... 296
Comparison with hierarchical forecasting • 297
Methods for grouping time series data • 297
Grouping criteria • 297
Techniques for identifying natural groupings in data • 298
Data preprocessing steps for grouped time series analysis • 298
Forecasting techniques for grouped time series • 299
Specialized models for grouped time series • 300
Grouped forecasting practical example • 300

Handling cross-sectional dependencies • 305
Techniques for modeling interactions between different groups • 305
Impact of cross-sectional dependencies on forecasting accuracy • 306
Summary **306**

Chapter 11: Prophet for Time Series **309**

Technical requirements **309**
Introducing Prophet **310**
Key components of Prophet • 311
When to use Prophet • 311
Implementing Prophet in R **312**
Tuning and customizing models **317**
Seasonality modes • 317
Adding Regressors • 318
Advanced Prophet techniques **322**
Holiday effects • 322
Cross-validation • 323
Parameter tuning • 324
Summary **327**

Chapter 12: Introducing Neural Networks and Time Series **329**

Technical requirements **330**
Understanding the basics of neural networks **330**
Activation functions • 333
Introducing simple neural networks **335**
Building a simple neural network model for time series in R **336**
Data preparation • 336
Model preparation • 337
Evaluating and optimizing simple neural network models **339**
Summary **343**

Chapter 13: Deep Learning with LSTM Models **345**

Technical requirements **346**
Introducing RNNs **346**
Typical structure of an RNN • 346

Limitations of traditional RNNs • 347

The vanishing gradient problem • 347

Short-term memory issue • 348

Introducing LSTM networks 349

How LSTM overcomes RNN limitations • 350

LSTM excels in time series forecasting • 351

Addressing common challenges in time series data • 351

Implementing LSTM in R 352

Tuning LSTM models for optimal performance 359

Finding the optimal parameters • 360

Overfitting and underfitting • 363

Summary 365

Final words 365

Other Books You May Enjoy 368

Index 371

Preface

Time series analysis has become an essential discipline in today's data-driven world, enabling organizations and individuals to understand historical patterns, identify trends, and generate accurate forecasts for future decisions. From financial industries to retail, healthcare, manufacturing, and supply chain management, time series forecasting plays a critical role in transforming raw data into actionable insights. If any of the above relates to you, you will find this book helpful. It provides learners with a comprehensive and practical understanding of time series analysis and forecasting using the R programing language.

The book begins with the fundamentals of R programming and gradually introduces the concepts, structures, and techniques required to work with time series data effectively. You will gain hands-on experience with data manipulation, data and time processing, visualization techniques, and exploratory data analysis; together, they form the foundation for successful forecasting.

Building upon these fundamentals, you will go through a variety of classical forecasting methodologies, including Exponential Smoothing, Holt-Winters Methods, ARIMA-family models, TBATS models, and Bayesian forecasting approaches. These techniques provide you with a broad toolkit for addressing different forecasting challenges and understanding the strengths and features of each modeling approach.

The final chapters introduce modern forecasting methods that have gained significant popularity both in industry and academia. You will explore Meta (Facebook) Prophet, neural networks, recurrent neural networks (RNNs), and long short-term memory (LSTM) networks for time series forecasting.

By the end of the book, you will possess the knowledge and practical hands-on skills necessary to develop, evaluate, and optimize forecasting models ranging from traditional statistical approaches to advanced deep learning solutions.

Who this book is for

This book is designed for students, business analysts, data analysts, financial analysts, researchers, data scientists, and professionals who wish to develop practical skills in time series analysis and forecasting using R. It is particularly valuable for individuals responsible for analyzing historical data, predicting future outcomes, and supporting data-driven decision-making processes.

The content is suitable for both beginners and intermediate learners. Readers with little or no prior experience in time series forecasting will benefit from the step-by-step introduction to R and fundamental forecasting concepts, while more experienced practitioners can expand their knowledge by exploring advanced forecasting techniques, machine learning models, and deep learning approaches for time series applications.

What this book covers

Chapter 1, *Getting Started with Basics of R*, introduces the R programming language and its role in data analysis and forecasting. You will learn the fundamental programming syntax, various data types and structures, and essential operations and functions for data manipulation.

Chapter 2, *Understanding ,Time Series Data*, provides a foundation in time series analysis by defining time series data and examining its key components. You will learn how to work with time series objects in R and perform common operations used in time-dependent data analysis.

Chapter 3, *Working with Dates and Times*, focuses on handling temporal data in R. Topics include date-time classes, time-based calculations and transformations, and practical techniques for managing and manipulating time series data.

Chapter 4, *Visualizing Time Series*, explores the importance of effective visualization in time series analysis. You will learn basic plotting techniques, advanced visualizations using `ggplot2`, interactive dashboards using `shiny` and `plotly`, and common mistakes to avoid when presenting time series data.

Chapter 5, *Exploratory Data Analysis for Time Series*, introduces exploratory techniques used to understand the behavior of time series data. Topics include trend analysis, smoothing methods, seasonal decomposition, autocorrelation analysis, and anomaly detection.

Chapter 6, *Exponential Smoothing and Holt-Winters*, presents forecasting techniques based on exponential smoothing. Through practical examples and case studies, you will learn simple, double, and triple exponential smoothing methods, including the Holt-Winters forecasting approach.

Chapter 7, *The ARIMA Model Family*, covers one of the most widely used forecasting frameworks in time series analysis. You will learn the principles and implementation of ARIMA, SARIMA, and SARIMAX models, along with methods for model selection and forecasting.

Chapter 8, *Forecasting with TBATS Modeling*, introduces TBATS models, which are designed to handle complex seasonal patterns and challenging forecasting scenarios. You will learn the underlying concepts, implementation techniques in R, and interpretation of model components and parameters.

Chapter 9, *Bayesian Approaches in Time Series Analysis*, explores Bayesian forecasting methodologies and their applications to time series problems. You will learn the fundamental concepts of Bayesian inference, implement Bayesian forecasting models in R, and interpret forecasting results within a probabilistic framework.

Chapter 10, *Multi-Series Forecasting Techniques*, introduces forecasting methods for multiple related time series. Topics include hierarchical forecasting structures, grouped time series analysis, and techniques for generating consistent forecasts across interconnected datasets.

Chapter 11, *Prophet for Time Series*, focuses on Prophet, a forecasting framework designed for business and operational forecasting tasks. You will learn how to implement Prophet models in R, customize model parameters, tune forecasting performance, and apply advanced Prophet features.

Chapter 12, *Introducing Neural Networks and Time Series,* introduces the fundamentals of neural networks and their application to forecasting problems. It explains how neural networks function, build forecasting models in R, evaluate performance, and optimize model accuracy through tuning techniques.

Chapter 13, *Deep Learning with LSTM Models*, explores advanced deep learning approaches for time series forecasting using `torch` package. You will be introduced to recurrent neural networks (RNNs) and long short-term memory (LSTM) networks, you will learn how to implement LSTM models in R, and you will apply optimization techniques to improve forecasting performance.

To get the most out of this book

All setup, installation, and packages are covered in the book.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to **packtpub.com**
2. Click your **profile picture** and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to `packtpub.com/unlock` or scan the following QR code:



2. Search for this book.
3. Sign up or log in to your free Packt account.
4. Upload your proof of purchase and download the code bundle locally.

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:`https://packt.link/gbp/<9781836203315>`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "The imported `qgsfunction` is a decorator used to register new QGIS expression functions."

A block of code is set as follows:

```
ggplot(df, aes(x = Month.with.Year, y = revenue)) +
 geom_line(color = "blue", linewidth= 1) +
 geom_smooth(method = "lm", color = "red", se = FALSE) + # Add a linear trend line
 labs(title = "Analysis of Trend Types",
 subtitle = "Red line represents a linear fit to the data",
 x = "Date",
 y = "Value") +
 theme_minimal()
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
library(ggplot2)

ggplot(lag_data, aes(x = lag1, y = actual)) +
  geom_point(color = "blue") +
  labs(title = "Lag Plot of Time Series (Revenue)",
       x = "Lag 1",
       y = "Actual Values") +
  theme_minimal()
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Select the **Locations** layer in the project from the **Layers** panel."

Note

Warnings or important notes appear like this.

Tip

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com/`.

Share your thoughts

Once you've read *Hands-on Time Series Analysis with R*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



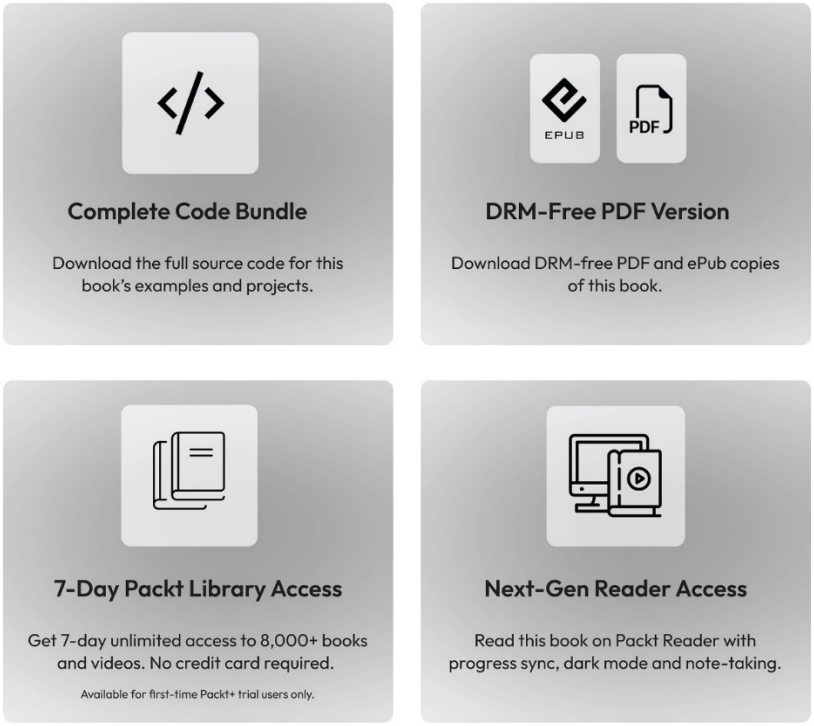
`https://packt.link/r/180323878X`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

1

Getting Started with the Basics of R

Back in the early 1990s, R started changing the game for everyone working with data. It was built with a lot of smart thinking about stats, ready to dive into complex data puzzles and turn them into clear, impactful stories. These stories help people make better decisions, all thanks to how R handles data.

R has a massive collection of tools, or **packages**, for just about any data task you can think of. Want to make your data look good? There's `ggplot2`. Need to tidy up your data? Say hello to `dplyr`. Dreaming of making your data interactive online? Check out `Shiny`. With these tools, you can push your data analysis way further than you thought possible.

R has a whole world of people behind it, sharing ideas, and building things together. You'll find a bunch of folks ready to help, inspire, and share some pretty neat tricks.

With R, you're joining a journey with a bunch of other data explorers, looking to uncover new insights, predict trends, or even advance research in all sorts of areas like finance, healthcare, and more.

We will cover the following topics in this chapter:

- Pros and Cons of the R programming language
- Understanding fundamental R syntax for programming
- Exploring various data types and structures in R
- Mastering basic operations and functions for data manipulation

By the end of this chapter, you'll not only have a solid understanding of R and RStudio's capabilities but also how to navigate their initial setup, dive into R's syntax, manage different types of data, and employ basic operations and functions for data analysis.

> **Note**
>
> **Your purchase includes a free PDF copy + exclusive extras**
>
> Your purchase includes a DRM-free PDF copy of this book, a 7-day trial to the Packt+ library (no credit card required), and additional exclusive extras. See the *Free benefits with your book* section in the *Preface* to unlock them instantly and maximize your learning.

Technical requirements

Before we dive headfirst into the world of R and RStudio, let's make sure you've got all the gear you need for the chapter:

- A computer: Whether you're team Mac, Windows, or Linux, you'll need a computer to run R and RStudio.
- An internet connection: You'll need to download R, RStudio, and any packages you decide to add to your toolkit. Once you've got everything set up, you can do a lot of work offline.

Furthermore, you will also need to have R and RStudio installed in your laptop, but we will go through detailed instructions for installation and setup later in this chapter.

Introducing R

R has become a key player in the data analysis and statistics world. It started in academic circles but quickly grew essential for data science thanks to its excellent tools for analyzing data, making graphs, and managing data efficiently.

Some of the pros of the R programming language include:

- Top-notch for stats and visualization: R is excellent for statistical analysis and creating professional-looking charts, making it a favorite among data analysts and researchers.
- Free and open-source: Anyone can use R without cost, which encourages a wide range of users to adopt it, contribute new tools, and help improve it.
- Rich collection of packages: R offers a vast array of packages, like `ggplot2` for charts and `dplyr` for data management, making it versatile for various data tasks.
- Strong community support: R's user community is active and welcoming, offering valuable resources for learning and problem-solving. For example, Reddit users often find active comments in r/stats, r/rprogramming, and other subreddits; on the other

hand, X (formerly Twitter) users can follow the #rstats tag to find R-related content, tips and blog posts.

Meanwhile, some of the cons of R include:

- Struggles with Big Data: R can slow down when working with very large datasets because it loads data into computer memory
- Learning curve: R's comprehensive capabilities come with a complexity that can be daunting for beginners
- Focused on data tasks: R is specialized for data analysis and not suited for general programming tasks like web development

R is a powerful tool for data analysis, widely used for its statistical analysis and visualization capabilities. It's supported by a strong, collaborative community, and a wealth of packages that extend its functionality. However, new users should be prepared for its learning curve and understand its limitations with large datasets and non-data tasks. Despite these challenges, R's benefits make it a valuable tool for professionals working with data.

Introducing RStudio

RStudio elevates R programming to new heights, distinguishing itself as the go-to **Integrated Development Environment** (**IDE**) for data scientists, statisticians, and analysts who prefer using R. At its core, RStudio streamlines the process of data analysis and statistical modeling, offering a cohesive and intuitive interface that dramatically enhances productivity and the overall coding experience. Its comprehensive nature integrates all necessary tools—console, script editor, and data visualization panels—into a single environment, simplifying workflow and minimizing the need to switch between different applications or windows.

Moreover, RStudio's robust features for project management and package handling empower users to organize their work more efficiently and to seamlessly incorporate a vast array of R packages into their projects. The ability to view plots, manage variables, and read documentation within the same window not only boosts efficiency but also aids in understanding and troubleshooting code.

The real game changer, however, lies in RStudio's commitment to fostering a community and supporting open-source development. By offering both free and professional versions, RStudio makes advanced R programming accessible to a wide audience, from students and academics to professionals in industry. This approach has cultivated a rich ecosystem of packages and resources that continually expand the capabilities of R.

Furthermore, RStudio's features like R Markdown and Shiny applications extend R's utility beyond traditional statistical analysis, allowing users to create dynamic reports and interactive web applications directly from R scripts. These capabilities make RStudio not just a tool for

statistical analysis but a comprehensive platform for data science projects that require reproducibility, collaboration, and communication.

In essence, RStudio stands out not only for its ability to make R programming more accessible and efficient but also for its role in advancing the field of data science. Its comprehensive environment, coupled with strong community support and the integration of cutting-edge features, makes RStudio the best choice for anyone looking to leverage the full power of R.

Installing R and RStudio

To get started with R and RStudio, follow the instructions below. This setup will equip you with one of the most powerful tools in data science and statistical computing.

Installing R

The following is a step-by-step guide on how to install Windows, Mac, and Linux:

1. Start by downloading R. Visit the **Comprehensive R Archive Network** (**CRAN**) at `https://cran.r-project.org/`.
2. Select the download link that corresponds to your operating system (Windows, Mac, or Linux).
3. The installation process for R varies for different operating systems:
    - **Windows**: Run the downloaded `.exe` file and follow the installation instructions.
    - **Mac**: Open the `.pkg` file and follow the on-screen prompts.
    - **Linux**: Use your distribution's package manager. For Debian-based distributions, you can use `sudo apt-get install r-base`, and for Red Hat-based distributions, `sudo yuminstall R`.
4. Open your terminal or command prompt and type `R`. If R is successfully installed, you'll enter the R environment, indicated by the > prompt. The following figure shows the R console after a successful installation:

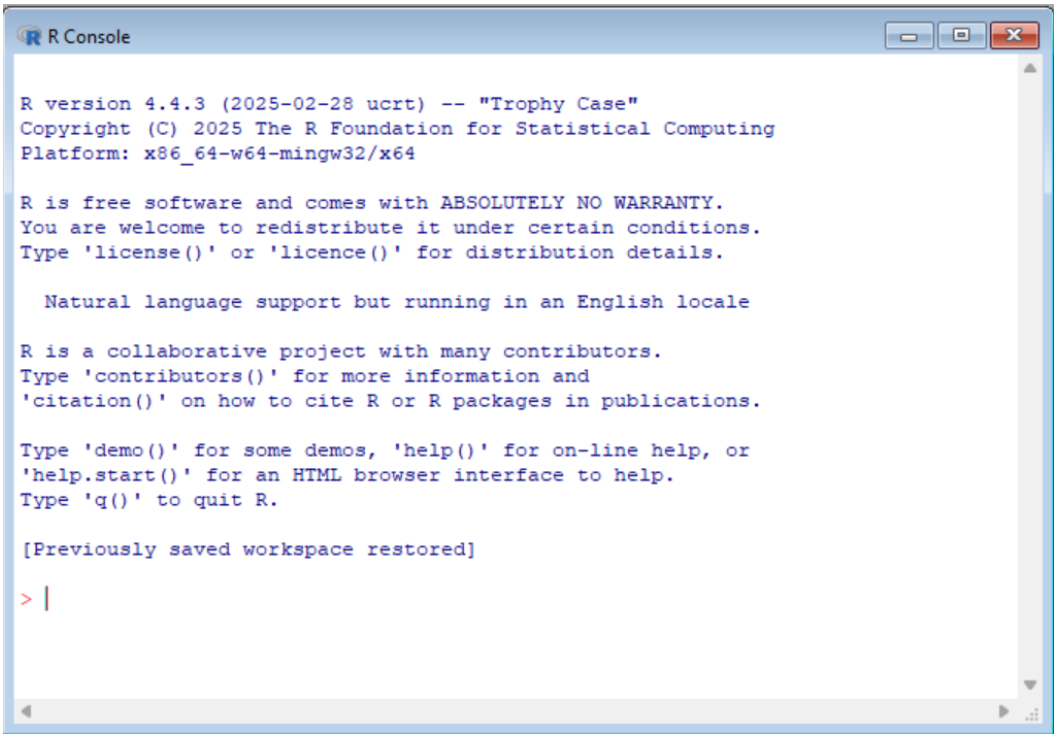


Figure 1.1: R Console after installation

Next, let's go through installing RStudio.

Installing RStudio

Follow the instructions below to install RStudio, the software to program and code R:

1. Begin by downloading RStudio. Go to the RStudio website at `https://docs.posit.co/ide/user/#rstudio-ide-oss-downloads`.
2. Choose the free version of RStudio Desktop, then download it for your operating system.
3. Installation Process:
    - **Windows**: Run the downloaded `.exe` file and follow the setup wizard
    - **Mac**: Open the `.dmg` file, then drag and drop RStudio to your Applications folder
    - **Linux**: Depending on the package format, use either `dpkg -i rstudio-x.yy.zzz-amd64.deb` for Debian-based distributions or `yum install`

`rstudio-x.yy.zzz-x86_64.rpm` for Red Hat-based distributions, replacing `x.yy.zzz` with the version number of the downloaded package

4. Once installed, open RStudio like any other application. RStudio will automatically detect your R installation. The following figure shows the layout of RStudio after installation:

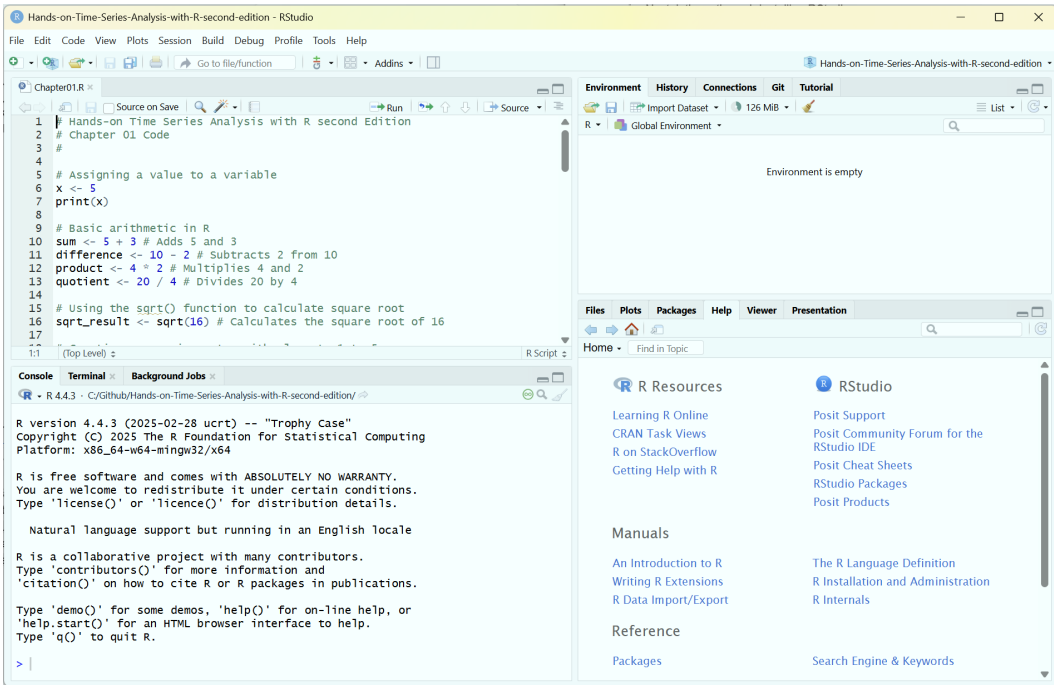


Figure 1.2: RStudio layout diagram after successful installation

Setting up your workspace

Here is how to start a new project, creating a file to start programming with R:

1. In RStudio, go to **File | New Project** to start a new workspace. This helps in organizing your scripts, data, and outputs.
2. You can install R packages using the command `install.packages("packageName")` in the console. Replace `"packageName"` with the name of the package you want to install. In the following chapter, we will install `zoo`, `xts` and `quantmod` packages and demonstrate the usage of them.

Tips for success

Here are some quick tips that will make the introduction to R and RStudio easier for you:

- Familiarize yourself with the layout: Script, Console, Environment/History, and Files/Plots/Packages/Help tabs.
- The best way to learn is by doing. Start simple and gradually take on more complex projects.
- The R community is vast and supportive. Use forums, online tutorials, and books (like this one!) to enhance your skills.

By following these steps, you'll have R and RStudio installed and be well on your way to performing powerful data analysis and statistical computing.

Understanding fundamental R syntax for programming

Getting started with R programming means diving straight into its syntax, the rules that guide how you write code. R syntax is your toolkit for instructing R to perform tasks, from basic arithmetic to complex data analysis. Understanding this syntax is crucial; it's the language you use to turn data into insights.

Syntax in R, as in any language, sets the groundwork for communication. It's what you use to create variables, control the flow of operations, and manipulate data. For those new to R, understanding syntax is your first step. For the pros, it's what everything else is built upon. Get the syntax down, and R's analytical power is at your fingertips.

In this section, you will learn about the basic syntax in R programming, perform simple arithmetic and learn about function calls.

The basics of R syntax

R's syntax is known for being logical and user-friendly. It revolves around objects like variables and functions. For instance, assigning a value to a variable in R is straightforward:

```
# Assigning a value to a variable
x <- 5
print(x) #print the value of x in console output, below is the output
[1] 5
```

This line tells R to assign the value `5` to `x`. The `<-(or =)` symbol is the assignment operator, a core part of R syntax.

Simple operations

Starting with R involves performing simple operations. These operations include arithmetic calculations, which are intuitive in R and are given as follows:

```
# Basic arithmetic in R
# contents after # are code comments
sum <- 5 + 3 # Adds 5 and 3
difference <- 10 - 2 # Subtracts 2 from 10
product <- 4 * 2 # Multiplies 4 and 2
quotient <- 20 / 4 # Divides 20 by 4
```

Each of these lines performs a basic arithmetic operation and assigns the result to a variable.

Function calls

Using functions is a big part of working with R. Functions are commands that perform specific tasks. Here's how to use a basic function:

```
sqrt_result <- sqrt(16) # Calculates the square root of 16
```

This code calculates the square root of `16` using the `sqrt()` function and assigns the result to `sqrt_result`.

As you get comfortable with these basics, you'll move on to more complex tasks, like manipulating data frames or creating plots. But even these advanced operations build on the simple principles of R syntax.

Now that we have seen the basics of syntax, let's look at the different types and structures that data can be in R.

Exploring various data types and structures in R

R handles a variety of data types, which are the building blocks of data manipulation and analysis. These include:

- **Numeric**: Represents decimal and whole numbers. For example, `2`, or `15.5`.
- **Integer**: A subtype of numeric that specifically represents whole numbers. Denoted by the `L` suffix, like `2L`.
- **Character**: Used for text or string data. Enclosed in quotes, such as `"Hello, World!"`.
- **Logical**: Contains Boolean (`TRUE` or `FALSE`) values. `TRUE` can also be abbreviated as `T` and `FALSE` as `F`.

- **Complex**: For complex numbers with real and imaginary parts, like `1 + 4i`.
- **Raw**: Handles raw binary data.

Exploring data structures in R

R's data structures organize data for efficient manipulation, analysis, and storage. They can be one-dimensional, two-dimensional, or multi-dimensional:

- **Vectors**: The simplest and most common data structure in R. A vector holds elements of the same data type. Use the `c()` function to create a vector, for instance, `c(1, 2, 3)` for a numeric vector.
- **Matrices**: Two-dimensional, rectangular data structures that contain elements of the same data type. You can create a matrix with the `matrix()` function.
- **Arrays**: Similar to matrices but can have more than two dimensions. Created using the `array()` function.
- **Data frames**: Two-dimensional, tabular data structures that can hold elements of different data types. They are similar to datasets in other statistical software. Use the `data.frame()` function to create one.
- **Lists**: The most complex R data structure, capable of holding elements of any type, including other lists. Create a list using the `list()` function.

Code examples

To illustrate, here are simple examples of how to create and work with these data types and structures in R:

- **Numeric vector creation**: Vectors are the simplest type of data structure in R. Here, we create a numeric vector containing a sequence of numbers. Vectors are one-dimensional and can only contain data of the same type:

  ```
  # Creating a numeric vector with elements 1 to 5
  numeric_vector <- c(1, 2, 3, 4, 5)
  ```

  This code snippet creates a numeric vector named `numeric_vector` by combining the numbers 1 through 5 using the `c()` function. Vectors in R are one-dimensional arrays that store elements of the same data type. Here, `numeric_vector` stores integers, making it useful for numerical data manipulation and analysis.

- **Character vector creation**: Character vectors are used to store text strings. Like numeric vectors, they are one-dimensional and all elements must be of the character type:

```
# Creating a character vector with three fruit names
character_vector <- c("apple", "banana", "cherry")
```

This line of code defines a character vector named `character_vector` that contains three elements: `"apple"`, `"banana"`, and `"cherry"`. The `c()` function is used to combine these text elements into a single vector. Character vectors are essential for handling textual data in R, allowing for operations on strings such as sorting, filtering, and text analysis:

- **Matrix creation**: Matrices are two-dimensional data structures that contain elements of the same type, organized in rows and columns. They are particularly useful for numerical and statistical computations:

```
# Creating a 3x3 matrix with numbers 1 to 9
my_matrix <- matrix(1:9, nrow=3, ncol=3)
print(my_matrix) #print my_matrix in console output
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Here, a matrix named `my_matrix` is created using the `matrix()` function, filled with the numbers 1 to 9. The arguments `nrow=3` and `ncol=3` specify that the matrix should have 3 rows and 3 columns, respectively.

- **Data frame creation**: Data frames are two-dimensional, but unlike matrices, they can hold different types of data in each column, similar to a spreadsheet or SQL table. This makes them incredibly useful for statistical modeling and data analysis:

```
# Creating a data frame with Columns: Name, Age, and Salary
my_data_frame <- data.frame(
 Name = c("Alice", "Bob", "Charlie"),
 Age = c(25, 30, 35),
 Salary = c(50000, 60000, 65000))
print(my_data_frame)
     Name Age Salary
1   Alice  25  50000
```

```
2     Bob  30  60000
3 Charlie  35  65000
```

This code constructs a data frame named `my_data_frame`, which is a two-dimensional, table-like structure. Data frames can hold columns of different data types and are especially useful for datasets where each column represents a variable and each row represents an observation. In this example, the data frame includes columns for names, ages, and salaries, making it ideal for representing and analyzing related sets of data.

- **List creation**: Lists are the most flexible data structure in R. They can hold elements of any type and size, including other lists, making them ideal for organizing complex or heterogeneous data collections:

```
# Creating a list containing a name, an age, and a vector of hobbies
my_list <- list(Name="Alice", Age=25, Hobbies=c("Reading", "Cycling"))
```

The final snippet creates a list named `my_list`. Lists in R are generic vectors that can contain elements of different types and structures, including vectors, matrices, and even other lists. This makes lists extremely versatile for organizing and storing diverse data types. In `my_list`, various data types are included: a character string for `Name`, a numeric value for `Age`, and a character vector for `Hobbies`, showcasing the flexibility of lists in handling complex data structures.

Each of these data structures plays a vital role in R programming, allowing for efficient data manipulation, analysis, and visualization tailored to the specific needs of various tasks.

Mastering basic operations and functions for data manipulation

R offers a rich set of operations and functions that are essential for data manipulation. These tools allow you to perform calculations, transform data, and carry out statistical analyses efficiently. Mastering these basics is crucial for any aspiring data analyst or statistician working with R.

Arithmetic operations

R supports standard arithmetic operations which can be performed on numeric data. These operations are fundamental for data analysis, allowing for the manipulation and transformation of data.

Now, you are going to learn how to make simple arithmetic operations like, adding, subtracting or multiplying:

- Here, you can see how to add numbers, specifically 5 plus 3:

  ```
  sum_result <- 5 + 3
  ```

  This line adds the numbers 5 and 3 together, storing the result (8) in the variable `sum_result`.

- Here is an example of subtraction:

  ```
  difference_result <- 10 - 2
  ```

  In the example, 2 is subtracted from 10, with the result (8) saved in `difference_result`.

- To multiply numbers in R, you use the * symbol:

  ```
  product_result <- 4 * 2
  ```

  This code multiplies 4 by 2, storing the product (8) in `product_result`.

- To divide numbers you include the divisor on the right of the / symbol, which indicated the dividing symbol:

  ```
  quotient_result <- 20 / 4
  ```

  20 is divided by 4, and the quotient (5) is held in `quotient_result`.

- The ^ symbol allow to exponentiate numbers. The example below is 2 to the power of 3, or 2*2*2:

  ```
  power_result <- 2^3
  ```

  2 raised to the power of 3 (2 * 2 * 2 = 8) is calculated, with the result stored in `power_result`.

- Modulus operation:

  ```
  modulus_result <- 7 %% 3
  ```

This operation finds the remainder of 7 divided by 3 (which is 1), saving it in `modulus_result`.

But what about data analysis functions

Basic functions for data analysis

R comes with a plethora of built-in functions that facilitate data analysis. These include statistical functions, data manipulation functions, and general-purpose functions. Understanding and utilizing these functions is critical for effective data analysis. The functions include:

- Calculating the mean (average) of a numeric vector:

```
mean_value <- mean(c(1, 2, 3, 4, 5))
```

This function computes the average of the numbers 1 through 5, storing the result (3) in `mean_value`.

- Summing up elements of a vector created by `c()`:

```
sum_value <- sum(c(1, 2, 3, 4, 5))
```

Here, the sum of the vector elements (15) is calculated and stored in `sum_value`.

- Generating a sequence of numbers:

```
number_sequence <- seq(from = 1, to = 10, by = 2)
```

A sequence from 1 to 10 in increments of 2 (1, 3, 5, 7, 9) is generated, with the sequence stored in `number_sequence`.

- Sorting elements of a vector:

```
sorted_vector <- sort(c(5, 2, 3, 4, 1))
```

This sorts the elements of the vector in ascending order (1, 2, 3, 4, 5), storing the sorted vector in `sorted_vector`.

- Calculating Square Root for Numbers in a Vector Using `sapply` (Applying the `sqrt` function to each element of a vector (square root numbers)):

```
sqrt_vector <- sapply(c(1, 4, 9, 16, 25), sqrt)
```

The `sapply()` function applies another function to each element of a vector—in this case, `sqrt` the numbers. The square root numbers (1, 2, 3, 4, 5) are stored in `sqrt_vector`.

Becoming proficient in basic operations and functions in R is foundational for any data-related task. From simple arithmetic to complex statistical analysis, these tools empower you to manipulate and analyze data effectively. Practice and exploration of these operations and functions will significantly enhance your data manipulation skills in R.

Summary

This chapter is your starting point for diving into R and RStudio, giving you the tools and knowledge you need to kick off your data analysis journey. We began by talking about what R is good for, like stats and making sense of data through graphs, and where it might make you scratch your head a bit. Then, we introduced you to RStudio, a super helpful tool that makes working with R a lot easier, offering tips on how to set it all up. We walked you through installing R and RStudio step by step, so you're all set up and ready to go. Plus, we threw in some handy tips for making your work in RStudio smoother and more organized.

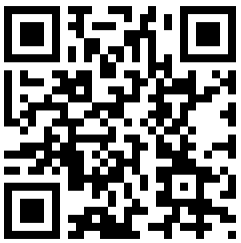
Next, we got into the nuts and bolts of R programming: the syntax. Think of syntax like the grammar rules of R. We showed you the basics, like how to do simple math, call functions, and start building up to more complex stuff. Understanding this is super important because it's the foundation for everything you'll do in R. We also explored different kinds of data you'll work with in R, like numbers, text, tables, and more complex mix-and-match types. With examples, we showed you how to create and play around with these, helping you get comfortable with managing and analyzing data in R.

Finally, we covered basic operations and functions—basically, the everyday tools you'll use to manipulate and analyze data. From adding and subtracting numbers to calculating averages and making sequences, we laid out how to do all that with R.

In the next chapter, we will introduce core concepts of time series data.

Get this book's PDF version and more

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require an invoice.

2

Understanding Time Series Data

If you've ever wondered how to peek into the future with data or understand why certain things happen when they do, you're in the right spot. This chapter is all about getting friendly with time series data using R, and by the time you're done, you'll be doing some pretty cool stuff with it.

Think of time series data like a diary that records what's happening, when, and then helps us spot the patterns. Ever wanted to predict stock market trends, figure out the best time to sell ice cream, or see how people react to big news on social media? Answering those questions is the kind of power you're about to have at your fingertips.

You'll learn how to turn your numbers into time series data in R, play around with both simple and more complex data sets, and even figure out the difference between data that's collected all the time and data that's not. And because not all data plays nice, we'll also cover how to deal with missing pieces, cut through the noise, and keep up with data that changes as you're looking at it.

By mastering a few key moves like slicing and dicing time, changing how often you're checking your data, and making sense of it all, you'll be all set to uncover hidden stories in your data. So, whether you're diving into data analysis for the first time or just looking to add some new tricks to your toolbox, this chapter has got you covered. Let's dive in and start turning those numbers into insights.

We are going to cover the following topics in this chapter:

- Defining time series data
- Exploring the components of time series data
- Handling time series data in R
- Learning fundamental operations specific to time series data

Technical requirements

R and RStudio are required for this chapter.

We will install the following R packages into our environment and use them for data analysis:

- **zoo**: Zoo takes its name from **Z's Ordered Observations**. *Z* is often used in statistical notation for a generic variable or dataset, while Ordered Observation provides methods for indexed and ordered data, which is key feature of time series data. The zoo package offers a versatile way to manage time series data, it provides various operations that are widely used in time series data practice.
- **xts**: The **eXtensible Time Series** (**xts**) package is a powerful and popular package designed for dealing with time series data. It extends zoo functionalities, optimized for financial time series data.
- **quantmod**: The **Quantitative Financial Modelling Framework** (**quantmod**) package is a powerful tool to handle development on financial models. It helps you to easily download and analyze financial time series data.

Note

R provides a built-in utility to access the help documentation for a function, a dataset, or a package. Simply use the `?` symbol before a package (or a function). For example, entering `?quantmod` will display the R documents for the `quantmod` package.

Defining time series data

Time series data is a sequence of data points collected or recorded at regular time intervals. This type of data is essential because it allows us to observe changes over time, uncover patterns, and make predictions. Time series data is not just a collection of values; it's a historical record that provides context, allowing analysts to identify trends, cycles, and seasonal variations.

Imagine taking a snapshot of something at regular intervals. Each snapshot captures a moment, and when you line these moments up, you get a story. That's essentially what time series data is—a story told over time. This could be anything from the daily closing price of a stock, the hourly temperature of a city, or the monthly sales figures of a product. The key components that define time series data are:

- **Sequential**: The data is collected in a chronological order and each data point corresponds to a timestamp

- **Regular Intervals**: The data is recorded at consistent periods, be it every second, hourly, daily, or yearly
- **Data Points Related Over Time**: Each data point in the series is related to its predecessors and successors, allowing for the analysis of trends, patterns, and forecasting

Let's consider an example. Think of the stock market: investors keep a close eye on these stories, tracking the daily closing prices of stocks. The following chart shows S&P 500 (SPY) Index values in the past 5 years. Those ups and downs, those peaks and troughs, they're like signposts. They tell you about the stock's health, its potential to grow, and hey, maybe even when it's the perfect time to jump on or off the stock market.



Figure 2.1: S&P500 index from 2019 to 2024 (Source: `https://www.google.com/finance/quote/.INX:INDEXSP?hl=en&window=5Y` *)*

Meteorologists use time series data to chart the course of weather patterns, temperature shifts, and rainfall. This isn't just about deciding if you need an umbrella tomorrow; it's also about predicting severe weather events, understanding climate change, and so much more.

Figure 2.2 shows a graph that predicts the weather for a location in the upcoming 7 days:

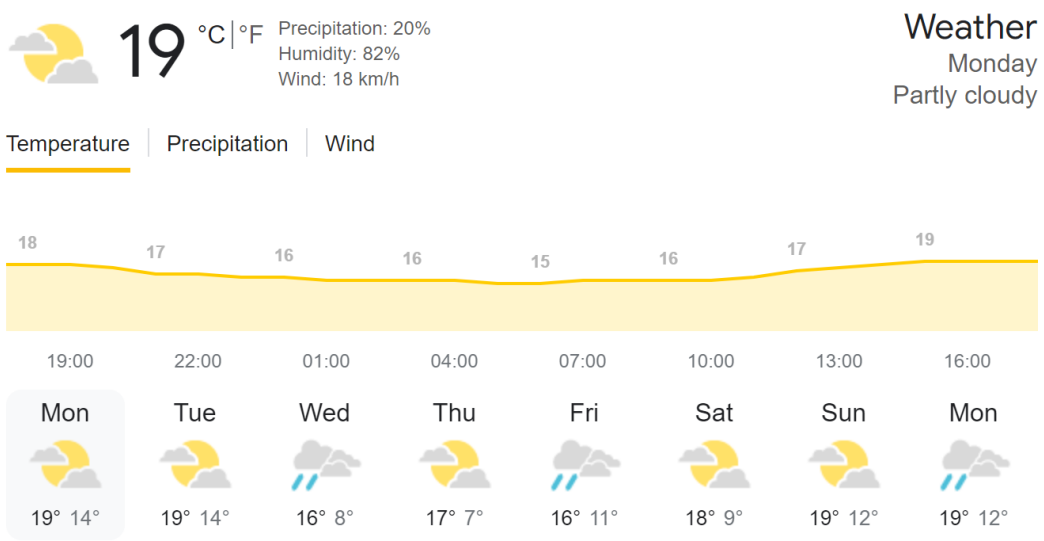


Figure 2.2: Weather forecast example (Source: Google Weather)

Now that you have the basics, let's start to get into the crucial components of this kind of data.

Exploring the components of time series data

In time series data, the chronological sequence of data points is paramount, knowing not just what happened, but *when* it happened. This temporal framework allows us to track changes, understand progressions, and even rewind the tape to analyze historical events in their specific contexts.

The **frequency** at which time series data is collected—a concept that ranges from the rapid-fire tick of milliseconds in financial markets to the gradual progression of years in climate studies—plays a pivotal role in shaping our understanding and analysis of the world.

In financial transactions, data can zip by at a breakneck pace, capturing every heartbeat of the market. This high-frequency data allows traders and algorithms to make split-second decisions, reacting to market movements with the agility of a sprinter. It's a world where timing is everything, and the ability to analyze data at such a fine resolution can mean the difference between profit and loss.

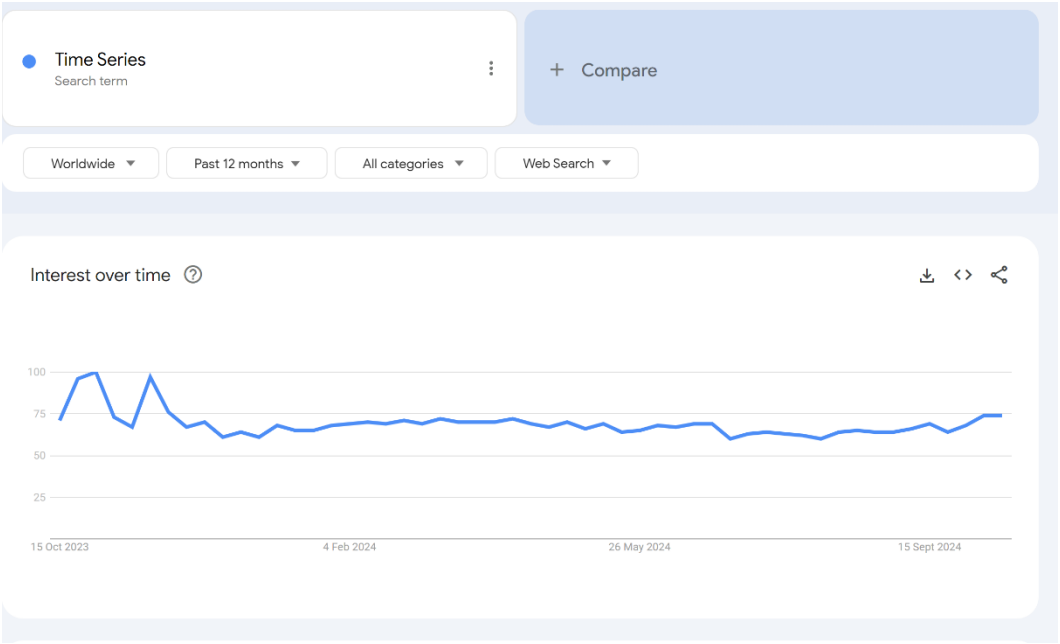


Figure 2.3: Google trends of the time series topic in the last 12 months (Source: `https://trends.google.com/trends/explore?q=Time%20Series&hl=en-GB`*)*

If we chart the data over different time periods, in this case comparing the last 12 months to the last 5 years, the insights we get are different: if we just look at the last 12 months, we can see that interest was up in late October to November; however, if we look at the whole of the 5 years, we can see interest peaking repeatedly in the tracked period. Shorter periods allow us to zoom in on specific spikes, whereas longer periods, like the following image focus more on trend and seasonal insights.

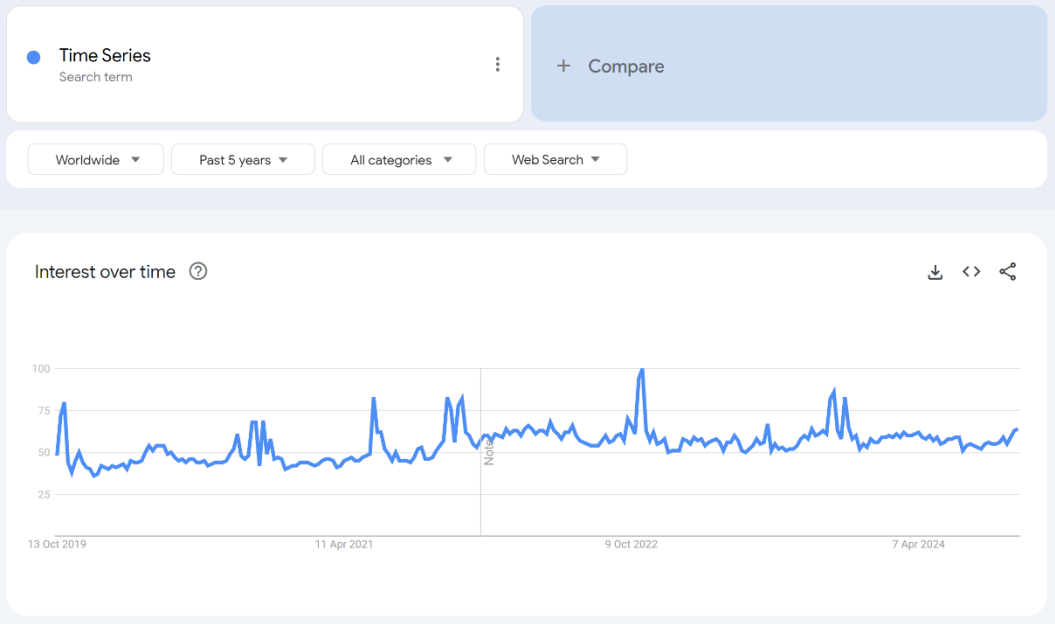


Figure 2.4: Google trends of the time series topic last 5 years (Source:`https://trends.google.com/trends/explore?date=today%205-y&q=Time%20Series&hl=en-GB`*)*

On the other end of the spectrum, the slow, measured cadence of data collection in fields like climate change research offers a different kind of insight. Here, data points spread over months, years, or even decades paint a picture of gradual transformations, revealing trends that unfold at a glacial pace. This long-term data helps scientists and policymakers understand broader patterns, such as rising global temperatures or shifting weather patterns, enabling decisions that shape future generations.

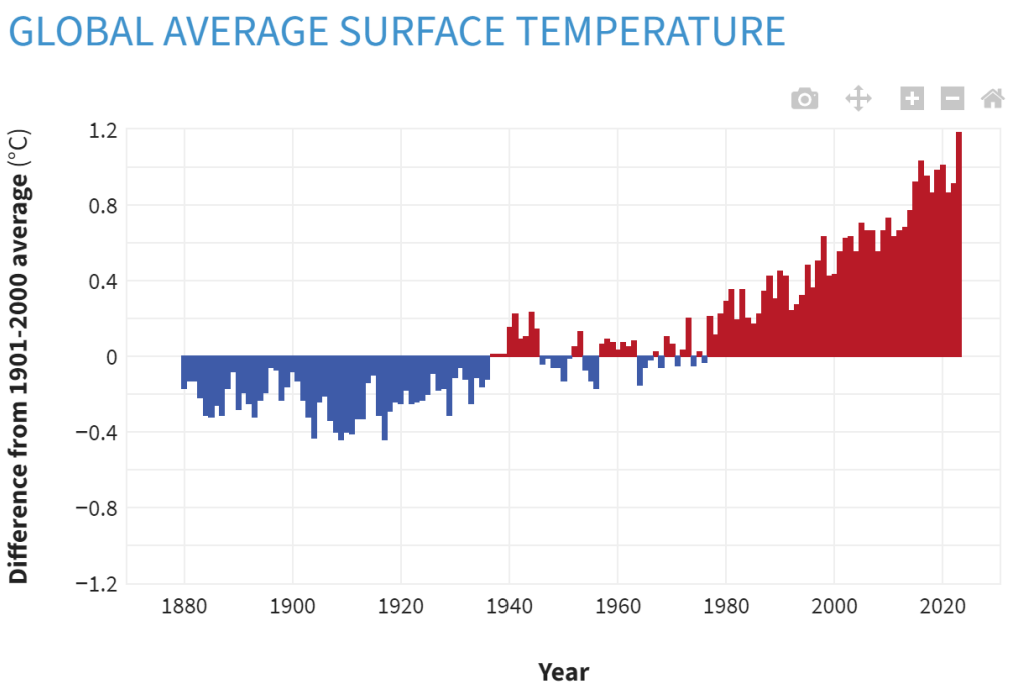


Figure 2.5: Global average temperature since 1870 (Source: `https://www.climate.gov/news-features/understanding-climate/climate-change-global-temperature`*)*

Furthermore, the slow frequency of data collection does not diminish its impact; rather, it provides the lens to observe the earth's slow dance of changes, an interesting contrast to the frenetic rhythms of financial markets.

Understanding the frequency of data collection is essential for navigating the complexities of time series analysis. It influences the tools and techniques we use, from the algorithms designed to process high-speed financial transactions to the models predicting long-term environmental shifts.

Trends

Identifying **trends** in data guides decisions and predictions. Analysts sift through data to distinguish lasting trends from random fluctuations, using these insights for various applications like financial forecasting or climate change policies, highlighting the importance of understanding data's overarching direction. The following screenshot of Google Trends shows how interest in Generative AI over the past 5 years has grown, as we might expect.

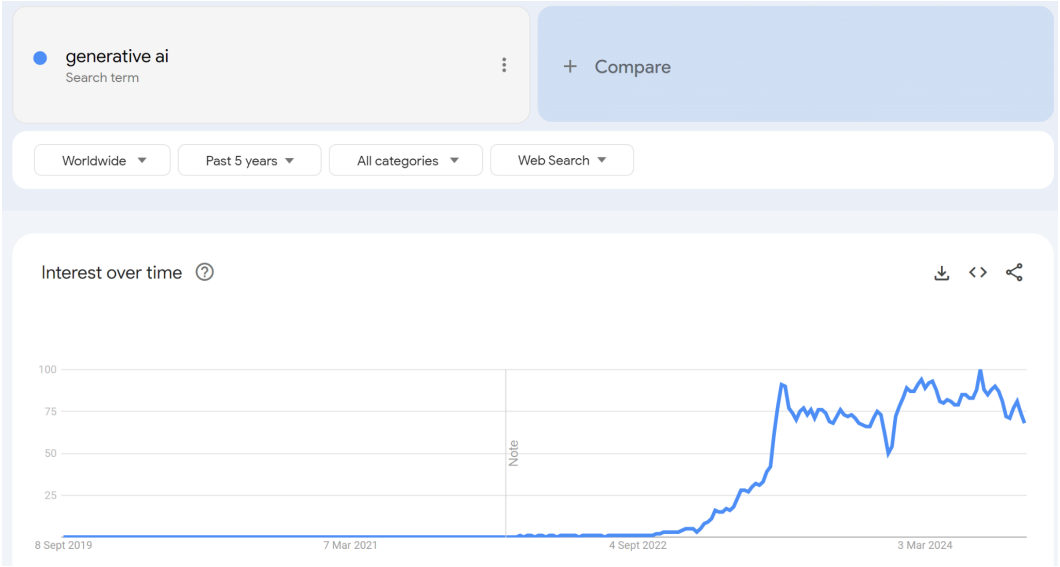


Figure 2.6: Google trends for Generative AI in the past 5 years (Source: Google Trends for Generative AI: `https://trends.google.com/trends/explore?date=today%205-y&q=generative%20ai&hl=en-GB`*)*

Techniques in time series analysis, such as moving averages or exponential smoothing explained in the later chapters in the book, help clarify trends by filtering out noise. One example might be a fashion editor curating styles, emphasizing the importance of discerning the true direction of data for forecasting and strategic planning.

The analogy between fashion and statistical trends illustrates a narrative of change. Recognizing these trends in time series data is vital for predicting future developments and making informed decisions, paralleling how fashion trends encapsulate societal attitudes and shifts over time.

Seasonality

Seasonality in time series data, like the surge in ice cream sales during summer in the following figure, highlights predictable, repeating patterns across specific periods. This cyclical phenomenon is crucial for understanding consumer behavior, allowing businesses to anticipate changes in demand based on the time of year, thus optimizing their strategies and stock levels.

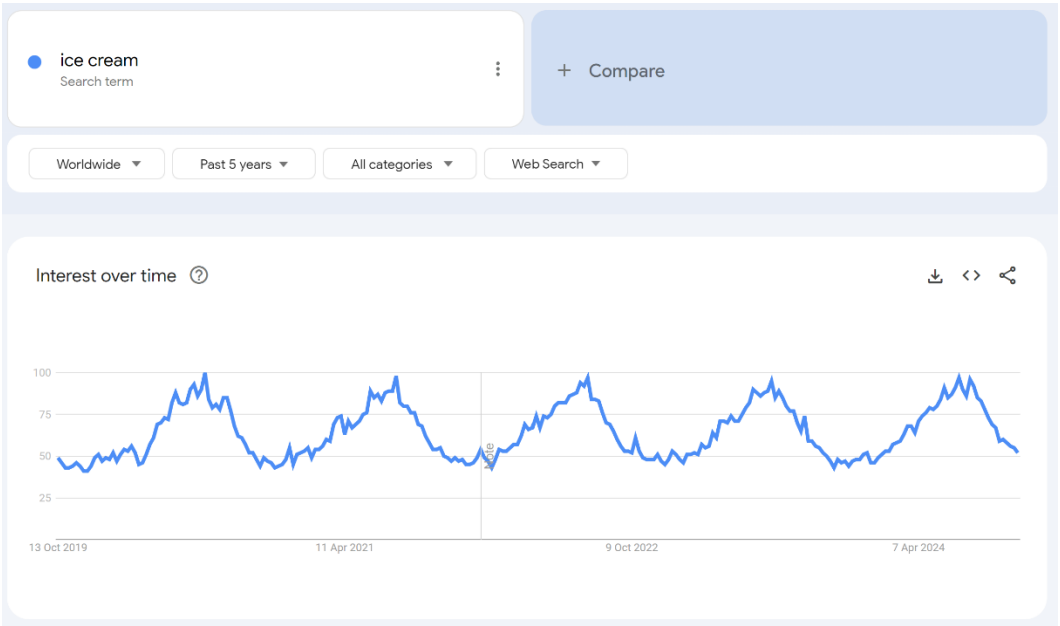


Figure 2.7: Ice cream Google trends from 2019 to 2024 (Source: `https://trends.google.com/trends/explore?date=today%205-y&q=ice%20cream&hl=en-GB`*)*

For businesses, recognizing these patterns is key to planning. By analyzing past sales data, companies can predict future demand spikes, such as the increase in ice cream sales when temperatures rise. This foresight enables them to adjust inventory, staffing, and marketing efforts accordingly, maximizing profitability and customer satisfaction.

Moreover, seasonality aids in strategic decision-making beyond mere sales. It influences various aspects of business operations, including supply chain management and product development. Understanding seasonal trends allows companies to launch timely promotions, ensuring they capitalize on periods of high demand while navigating the ebbs and flows of the market cycle.

Cyclical changes

Cyclical changes in time series data refer to patterns of increases and decreases that are not tied to a specific calendar schedule, differing significantly from seasonality. While seasonality is predictable and occurs at regular intervals throughout the year, like increased ice cream sales during the summer months, cyclical changes are more irregular. They can last for years or even decades and are often influenced by broader economic conditions, technological changes, or other external factors. These cycles reflect the underlying rhythms of an economy or a particular market, showing expansions and contractions that aren't bound by the seasons.

Bitcoin's 4-year cycle is a prime example of these cyclical changes at play, standing out starkly from the concept of seasonality. Unlike seasonal patterns that are directly linked to specific times of the year, the Bitcoin cycle revolves around the cryptocurrency's halving event, which occurs approximately every four years. This halving event reduces the reward for mining new Bitcoin by half, which can influence the supply and, potentially, the price of Bitcoin. This cycle is driven by the design of the cryptocurrency itself rather than external seasonal factors, highlighting how cycles in time series data can arise from unique, non-seasonal causes.

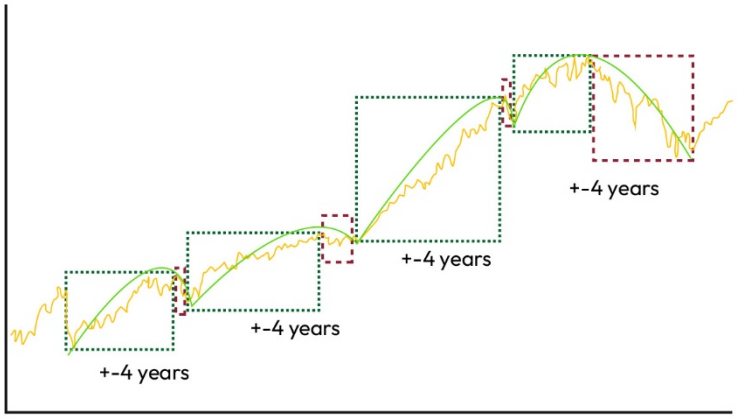


Figure 2.8: Bitcoin cycle over time

Understanding the difference between cyclical changes and seasonality, and recognizing examples like the Bitcoin 4-year cycle, is crucial for analyzing time series data. It allows us to see beyond the immediate, predictable patterns of seasonality to the deeper, more complex cycles that drive long-term changes. This deeper insight can help investors, economists, and analysts forecast future trends and make more informed decisions, showcasing the dynamic and multifaceted nature of time series data.

Noise

In the world of time series data, **noise** refers to the random variations that clutter our data sets, the unpredictable fluctuations that seem to defy explanation. This noise can stem from a multitude of sources—measurement errors, unexpected events, or simply the inherent randomness of certain processes.

For analysts and data scientists, filtering out this noise to uncover the underlying trends and patterns is a critical step in ensuring that decisions and forecasts are based on solid, meaningful information rather than being misled by misleading data.

The challenge of separating noise from signal is a fundamental aspect of time series analysis. This process involves a variety of statistical techniques designed to smooth out the data,

enhancing the clarity of the signal. Techniques such as moving averages or more sophisticated filtering methods are employed to strip away the randomness, allowing the true structure of the data to emerge.

However, it's not just about applying mathematical formulas; it requires a deep understanding of the context in which the data was generated. What constitutes noise in one scenario might be the signal in another.

Autocorrelation

The term **autocorrelation** may sound complex, but it boils down to a simple yet profound concept within time series data: the idea that the value of a data point today could be influenced by its own previous values.

By examining how strongly current values are related to their predecessors, we can develop models that anticipate future trends, patterns, and swings.

For instance, in financial markets, the movement of stock prices today can often be traced to their performances in the recent past, allowing investors to make informed predictions. Similarly, in meteorology, today's weather patterns might give clues about tomorrow's conditions, thanks to the thread of autocorrelation running through the data.

You will learn more about autocorrelation in a later chapter, deep dive on its formula, and how it can be visualized compared to its cousin, the partial autocorrelation.

Now that we have an understanding of key concepts, let's get into the types of data that can be studied in time series analysis.

Types of time series data

When we dive into the world of time series data, we find two distinct paths: **Univariate** and **Multivariate** time series. Each path offers unique insights and challenges, guiding us through the landscape of data analysis with different perspectives.

Univariate time series

Univariate time series are all about simplicity and focus. This type involves data collected over time from a single source or variable, offering a clear, uncluttered view of how that specific variable changes.

Imagine tracking the daily temperature in a city. Each day brings a new reading, a new data point, forming a timeline of temperature changes. This simplicity is powerful, enabling us to forecast future values based solely on the past values of the same variable. For example, predicting tomorrow's weather based on the trends and patterns observed in the temperature over the past days or weeks.

Multivariate time series

Multivariate time series involve data collected over time from multiple sources or variables. Instead of looking at daily temperature readings in isolation, we consider multiple factors together: temperature, humidity, and wind speed, all recorded daily. This approach allows us to see a fuller picture, understanding not just the behavior of each variable but also how they interact with each other.

For example, let's imagine a study aiming to predict the peak tourist season in a city. Analysts wouldn't rely solely on temperature data; they'd also consider humidity levels and wind speeds. By analyzing these variables together, they might discover that tourists prefer days when the temperature is above 75°F, humidity is below 60%, and there's a gentle breeze. This nuanced understanding helps local businesses prepare for the influx of tourists, from stocking up on sunscreen to scheduling extra staff.

Expanding on this, consider a major sports event planned in a city. The organizers could use multivariate time series analysis to ensure the event's success. By examining past events' data on temperature, humidity, and wind speed, they can choose a date when the weather conditions are likely to be optimal, enhancing the comfort of athletes and spectators alike. This approach not only improves the experience for everyone involved but also demonstrates the power of analyzing multiple variables to make informed, strategic decisions.

Continuous and discrete time series

When we explore the distinctions within time series data, we encounter two pivotal categories: **Continuous** and **Discrete** time series. Each type serves unique purposes and is suited to different kinds of analysis, depending on the precision of data required and the context of its application.

Continuous time series

Continuous time series data collection is comprehensive, capturing data at every possible moment within a specified timeframe. This approach, similar to a non-stop recording, offers an uninterrupted flow of information.

Medical devices that continuously monitor heart rate exemplify this, constantly tracking cardiac rhythms to detect nuanced variations critical for patient care. Similarly, environmental monitoring systems measure air quality or radiation levels second by second, providing vital data for real-time decision-making and long-term studies.

In financial markets, high-frequency trading algorithms analyze stock prices tick by tick to exploit minute fluctuations. Such precision is essential in domains like healthcare, environmental science, and finance, where detailed data analysis can uncover patterns, predict outcomes, and drive innovations.

Discrete time series

Discrete time series data is collected at set intervals, providing periodic snapshots instead of a non-stop record. This method captures essential points over time.

Think about how a store checks its sales figures at the end of each week. These weekly check-ins help the store understand what's flying off the shelves and what's not, guiding them on what to stock up on and when to have a sale.

It's not just retail stores that rely on this kind of data; economists keep an eye on things like unemployment rates month by month to get a sense of how the economy's doing. This approach to collecting data at regular intervals keeps things straightforward but still gives everyone from store managers to policymakers the insights they need to make smart decisions.

Understanding whether data is continuous or discrete helps analysts and researchers choose the right tools and approaches for their investigations. While continuous data offers depth and detail, discrete data affords simplicity and manageability.

Challenges in time series data analysis

When tackling time series data analysis, several challenges can arise, complicating the process of extracting meaningful insights. Let's cover the main hurdles analysts might face.

Dealing with missing data

Missing data in time series refers to gaps in sequentially ordered observations where a data value is not recorded for a specific timestamp. Dealing with missing data in time series analysis is like trying to solve a puzzle when some pieces are missing. It's a common challenge that can significantly impact the quality of your insights and predictions. Here's how to tackle it:

- **Identification**: First, you need to spot the gaps. Missing data can be random or follow a pattern. Recognizing these gaps early is crucial because they can skew your analysis, leading to inaccurate conclusions.
- **Impact**: Missing data can distort your understanding of the time series. For instance, if temperature data is missing for certain months, it could affect trend analysis and seasonal adjustment calculations.

Fortunately, there are several strategies to handle missing data fill in the blanks:

- **Imputation**: This involves estimating the missing values based on other available data. Techniques vary from simple (like filling gaps with the mean or median of nearby points) to complex (like using machine learning models to predict missing values).
- **Interpolation**: For time series data, interpolation methods can estimate missing values by considering the time-ordered nature of the data. Linear interpolation is

straightforward and commonly used, but more sophisticated methods might provide better accuracy for complex data.

- **Analysis Adjustment**: Sometimes, it's about working around the missing data. This might mean adjusting your analysis methods or using models that can handle gaps in the data.

Managing missing data in time series analysis requires a careful blend of detection, understanding the potential impact, and applying the most appropriate method for filling in or accounting for those gaps. The goal is always to maintain the integrity of your analysis, ensuring that the story your data tells is as complete and accurate as possible.

Dynamic and evolving data

Dynamic and evolving data in time series analysis present a unique set of challenges, akin to shooting a moving target. As the world changes, so does the data, making it imperative for models and analysis techniques to adapt in real time or risk becoming obsolete.

- First, analyzing dynamic data requires tools and methods that can keep up with its pace. This involves setting up systems that can process and analyze data as it comes in, allowing for immediate insights and responses. It's like trying to read a book while it's being written, necessitating a flexible approach to understand and react to new information.
- Moreover, as new data arrives, it can reveal trends or shifts that were not apparent before. Models that were accurate yesterday may not be tomorrow, requiring continual re-evaluation and adjustment. This might involve updating model parameters regularly or switching to different modeling techniques altogether. The key is to maintain a balance between responsiveness to new data and stability to avoid being misled by short-term fluctuations.
- Finally, the nature of the data itself can change over time. This is known as **data drift**. This could be due to shifts in the underlying processes generating the data or changes in the data collection methods. Analysts must be vigilant, ready to identify these shifts and understand their implications. It's a process of continuous learning and adaptation, ensuring that the analysis remains relevant and accurate.

In dealing with dynamic and evolving data, the challenge is not just in the analysis itself but in staying ahead of the curve. It requires a proactive stance, with systems and strategies in place to ensure that analysis keeps pace with the data's evolution. This agility is crucial for making informed decisions in a world where change is the only constant.

Managing noise

Noise represents those random fluctuations that add no value to the analysis, merely clouding our understanding of the underlying trends.

The initial task is to distinguish the valuable data patterns from the random noise. It requires a discerning eye, sometimes aided by advanced statistical methods, to ensure that the true essence of the data isn't drowned out by irrelevant variations.

Employing techniques to filter out the noise brings the data into sharper focus. Simple tools like **moving averages** help by blending data points to smooth erratic fluctuations. For a more tailored approach, sophisticated filters such as the **Kalman filter** or **wavelet transforms** adjust data dynamically, preserving crucial details while discarding the noise.

Streamlining the data to mitigate noise is key for deriving accurate insights and making solid predictions. Without this crucial step, data analysis can lead to false leads, skewed interpretations, and suboptimal decisions. Clearing away the noise ensures that strategic choices are informed by genuine trends, not by the random scatter of data points.

The process of managing noise in time series data is essential for clarity and accuracy in analysis. It involves not just recognizing and filtering out the noise but doing so in a way that preserves the integrity and significance of the true signal. When executed well, it allows analysts to read the story of the data with confidence, free from the distraction of irrelevant details.

Handling time series data in R

Let's walk through five fundamental examples of handling time series data in R, from creation to transformation.

Creating a time series object in R

Creating a time series object in R is a foundational skill for anyone looking to dive into time series analysis. This process converts your dataset into a structured format that R can easily interpret.

First, ensure your dataset is ready for conversion. Typically, you'll need a sequence of observations recorded at consistent intervals over time. This could be anything from daily sales figures to monthly temperature readings.

R's `ts()` function is designed to create time series objects from your data. The basic syntax requires you to input your data vector and specify the start and end time periods, along with the frequency of the observations. For example:

1. Use the following code to create dummy data for your time series:

```
my_data <- data.frame(
  Date = seq(as.Date("2020-01-01"), as.Date("2020-12-01"), by = "month"),
  Value = runif(12)  # Random values for each month
)
```

2. Next, create the time series object:

```
my_time_series <- ts(data = my_data$Value, start = c(2020, 1), frequency =
12)
```

   This command tells R that your data starts in January 2020, ends in December 2020, and has monthly observations (`frequency = 12`).

3. After creating your time series object, it's useful to inspect it to ensure it reflects your data accurately. Print the time series object:

```
print(my_time_series)
```

   This produces the following output, which does not fit well on paper but will be easier to read on your screen. You might also see different random values created for your data set; this is totally fine and the different value will not impact the concepts in the book:

```
          Jan       Feb       Mar       Apr       May       Jun
Jul       Aug       Sep       Oct       Nov       Dec
2020 0.5222043 0.8034903 0.1078941 0.1092760 0.6525748 0.4363834 0.3700710
0.4291788 0.6885271 0.5957612 0.4922222 0.1170296
```

4. Get a summary of the time series using the following code:

```
summary(my_time_series)
```

Which gives us the following:

```
   Min.    1st Qu.  Median   Mean     3rd Qu.    Max.
   0.1079  0.3068   0.4643   0.4437   0.6100     0.8035
```

These commands provide a quick overview of your time series, including its structure and some basic statistical insights.

5. Plot the time series object. A quick plot can help you visually inspect the trends, seasonality, or any anomalies in your data. Use R's `plot()` function for a basic line chart of your time series:

```
plot(my_time_series, main = "My First Time Series Plot", xlab = "Time",
ylab = "Observations")
```

This produces the following plot:

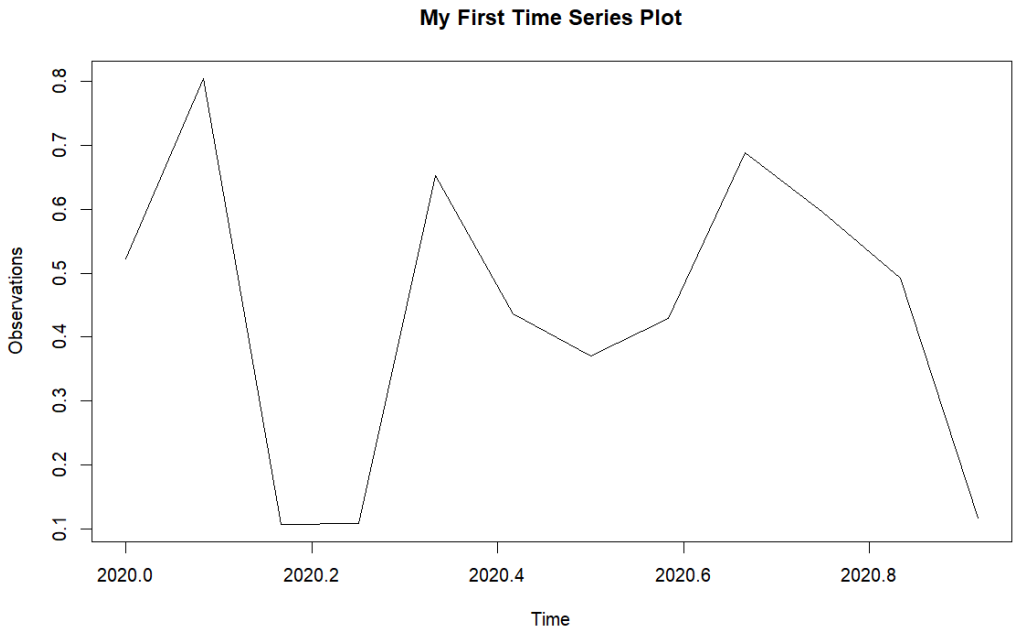


Figure 2.9: Time series plot example

This command generates a simple plot, with time on the x-axis and your observations on the y-axis, helping you get a visual feel for your data's behavior over time.

Creating a time series object in R is just the beginning. Once you have your data in the right format, a world of analytical possibilities opens up, from decomposing your series to uncover underlying patterns to forecasting future values. With practice, this process becomes a natural first step in your time series analysis workflow.

Using zoo and xts packages

The zoo and `xts` packages in R are particularly adept at handling time series data like stock prices. Here's how you can leverage these packages to transform your data into a more analyzable format and visually explore market trends.

Transforming with zoo

The zoo package offers a versatile way to manage time series data. Go through the following code block to convert irregular time series data into a zoo object:

1. First, check to see if the zoo package is already installed, and if it is not installed, install it with the following code:

```
if (!requireNamespace("zoo", quietly = TRUE)) {
  install.packages("zoo")
}
```

2. Next, load the zoo package:

```
library(zoo)
```

3. Create the zoo object and print it out, assuming your data is in `my_data` with `dates` as your date column:

```
zoo_object <- zoo(my_data$Value, order.by = my_data$Date)
print(zoo_object)
```

This creates a zoo object from your data, ordering by the date

```
2020-01-01 2020-02-01 2020-03-01 2020-04-01 2020-05-01 2020-06-01
2020-07-01 2020-08-01 2020-09-01 2020-10-01 2020-11-01
 0.5222043  0.8034903  0.1078941  0.1092760  0.6525748  0.4363834
0.3700710  0.4291788  0.6885271  0.5957612  0.4922222
2020-12-01
 0.1170296
```

This code snippet creates a zoo object, allowing for time-based indexing and manipulation. The `order.by` parameter ensures your data is aligned according to the dates, which is essential for time series analysis.

Transitioning to xts

The `xts` package extends zoo functionalities, optimized for financial time series data. Converting data into an `xts` object is straightforward:

1. Install the `xts` package if not yet done:

```
if (!requireNamespace("xts", quietly = TRUE)) {
  install.packages("xts")
}
```

2. Load the `xts` package:

```
library(xts)
```

3. Create the `xts` object and print it out:

```
xts_object <- xts(my_data$Value, order.by = as.POSIXct(my_data$Date))
print(xts_object)
               [,1]
2020-01-01 0.5222043
2020-02-01 0.8034903
2020-03-01 0.1078941
2020-04-01 0.1092760
2020-05-01 0.6525748
2020-06-01 0.4363834
2020-07-01 0.3700710
2020-08-01 0.4291788
2020-09-01 0.6885271
2020-10-01 0.5957612
2020-11-01 0.4922222
2020-12-01 0.1170296
Warning message:
object timezone ('UTC') is different from system timezone ('')
  NOTE: set 'options(xts_check_TZ = FALSE)' to disable this warning
    This note is displayed once per session
```

This transforms your dataset into an `xts` object, suitable for time-ordered operations and analysis. Here, `as.POSIXct` ensures that dates are correctly formatted for `xts`, which is particularly important for financial data with specific timestamp requirements.

Plotting with zoo

Visualizing your zoo object can reveal patterns and trends as follows:

```
plot(zoo_object, main = "Stock Price Over Time", xlab = "Date", ylab = "Price")
```

This produces the following output:

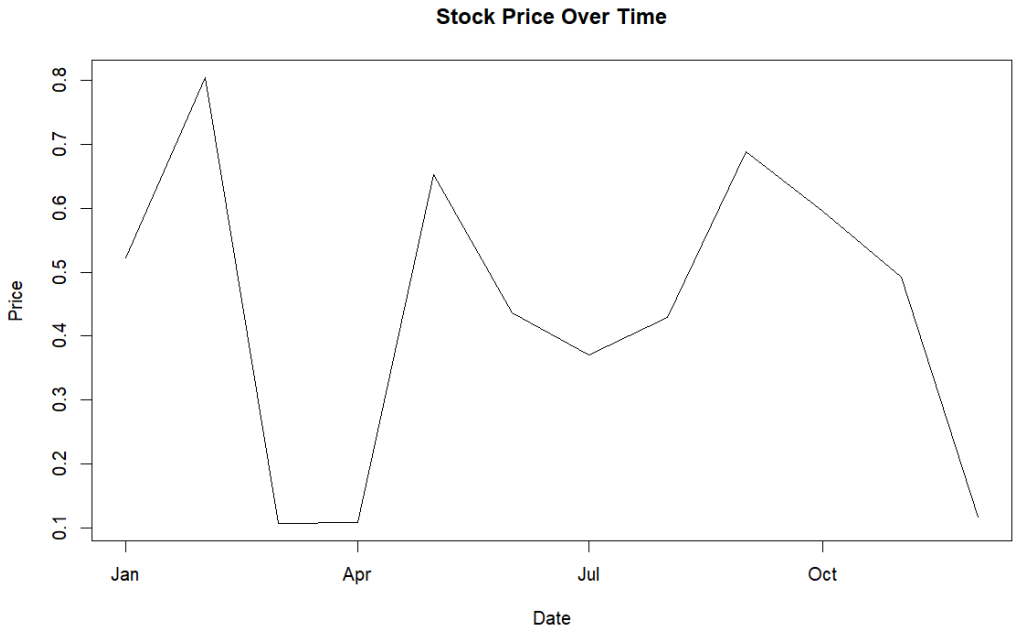


Figure 2.10: Simple plot in R on a zoo object

The code generates a simple line plot, ideal for spotting trends in stock prices over time.

Plotting with xts

Plotting with `xts` package is fairly similar to general plotting; its x-axis contains a time scale. You can just use `plot.xts()` if the object used in the function is an `xts` object:

```
plot.xts(xts_object, main = "Detailed Stock Price Analysis")
```

This produces the following output:

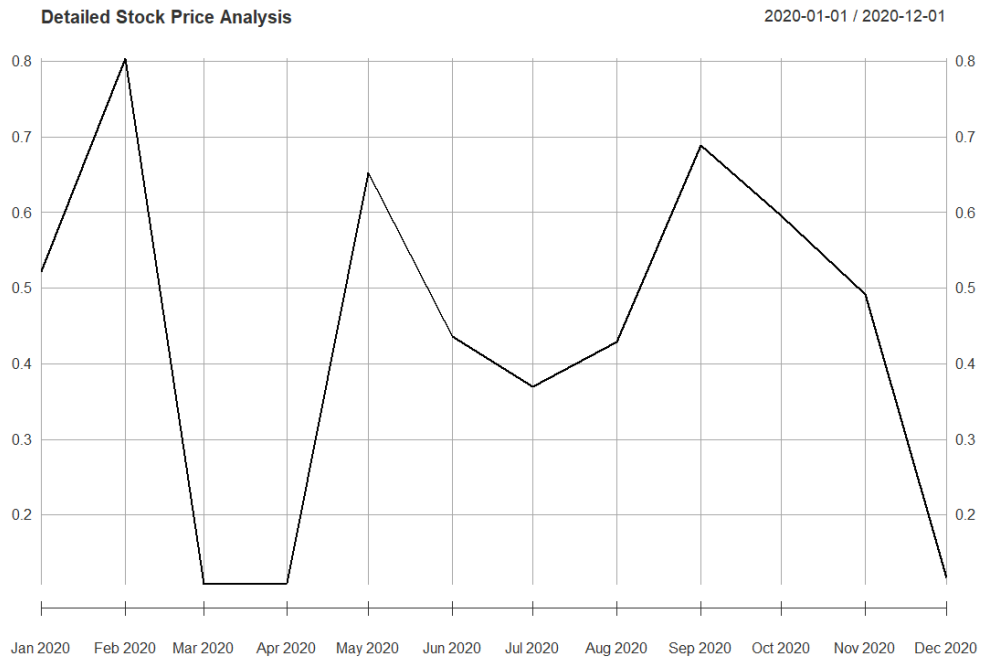


Figure 2.11: Detailed plot in R on an xts object

The code creates a detailed plot, leveraging `xts`'s specialized time series plotting capabilities.

Using the `zoo` and `xts` packages not only simplifies the task of structuring irregular time series data but also enables a deeper exploration of the data through visualization. By converting your dataset into these formats, you open up a wealth of analytical possibilities, from basic trend analysis to complex financial modeling, all while gaining visual insights into market dynamics.

So far you have seen some fundamental code snippets in R on time series object; next we will see operation examples specific to time series data.

Learning fundamental operations specific to time series data

Diving into time series data opens up a world of analytical possibilities. Mastering the basic operations is crucial for anyone looking to explore and analyze temporal data effectively. This section introduces fundamental operations that are essential for manipulating and understanding time series data.

Time indexing and slicing

Time indexing and slicing are fundamental techniques in time series analysis, allowing you to navigate through and analyze temporal data efficiently. These operations enable you to focus on specific segments of your dataset based on time, enhancing the precision of your analysis. Let's start with time indexing.

Time indexing

This operation involves selecting data points at specific time stamps or periods. Time indexing is essential when you're interested in analyzing data from particular moments. For example, in a dataset of daily stock prices, you might want to index and retrieve the price on a specific date to observe market reactions to an event.

Let's start by retrieving the Bitcoin price in 2023:

1. Install the `quantmod` package if that is not yet done:

```
if (!requireNamespace("quantmod", quietly = TRUE)) {
  install.packages("quantmod")
}
```

2. Load the `quantmod` package:

```
library(quantmod)
```

3. Retrieve BTC-USD from yahoo finance data source. Specify the symbol for Bitcoin from Yahoo Finance (`BTC-USD`):

```
getSymbols("BTC-USD", src = "yahoo", from = "2023-01-01", to =
"2023-12-31", auto.assign = TRUE)
```

4. Extract the closing prices for BTC-USD. `Cl()` extracts the closing prices. You can use `Hi()` and `Lo()` to extract high and low prices accordingly:

```
btc_data <- Cl(`BTC-USD`)
```

Then we can isolate a price via its time index.

5. Retrieve the closing price for 2023-03-15. `btc_data` is an `xts` time series object of daily stock prices:

```
specific_day_price <- btc_data['2023-03-15']
specific_day_price
           BTC-USD.Close
2023-03-15      24375.96
```

This retrieves the stock price from March 15, 2023.

Slicing data ranges

Slicing allows you to select and extract data within a specific range. This is particularly useful for conducting analyses over certain periods, such as quarterly financial performance or seasonal sales trends. Slicing helps in narrowing down the dataset to the relevant timeframe, making the analysis more manageable and focused.

Slice the data from 2023-03-01 to 2024-04-30:

```
march_to_april <- btc_data['2023-03-01/2023-04-30']
march_to_april
           BTC-USD.Close
2023-03-01      23646.55
2023-03-02      23475.47
2023-03-03      22362.68
2023-03-04      22353.35
2023-03-05      22435.51
2023-03-06      22429.76
2023-03-07      22219.77
2023-03-08      21718.08
2023-03-09      20363.02
2023-03-10      20187.24
2023-03-11      20632.41
…
```

This command extracts all daily prices in the specified date range. The output is truncated for display purposes.

When working with time series data in R, especially with `xts` or `zoo` objects, these operations are straightforward. The square brackets (`[ ]`) are used for indexing and slicing, where you can

specify the exact dates or range of interest. This approach allows for intuitive and efficient data manipulation, enabling deeper insights into temporal patterns and anomalies.

Frequency conversion and resampling

Adjusting the frequency of time series data, through techniques like frequency conversion and resampling, is essential for tailoring your dataset to specific analysis needs. These methods allow for altering the granularity of data, from broad, summarizing views to detailed, fine-grained observations.

Frequency conversion

Switching the observation frequency of your dataset can significantly impact your analysis, allowing you to zoom in or out on trends. If you're starting with daily stock market data but are more interested in overarching trends rather than daily volatility, converting to a monthly frequency by averaging daily prices provides a clearer view of long-term movements.

Calculate the monthly average and view the results with the following code:

```
average_monthly_prices <- apply.monthly(btc_data, FUN = colMeans)

print(average_monthly_prices)
           BTC-USD.Close
2023-01-31      20250.72
2023-02-28      23304.54
2023-03-31      25116.90
2023-04-30      28857.57
2023-05-31      27499.31
2023-06-30      27763.20
2023-07-31      30057.47
2023-08-31      27852.79
2023-09-30      26306.14
2023-10-31      29755.90
2023-11-30      36596.16
2023-12-31      42546.89
```

This averages daily prices into monthly, smoothing out short-term fluctuations.

Resampling

Resampling steps in when you need to alter the dataset's frequency, especially to a finer time scale, often requiring interpolation to fill in the gaps. This is particularly useful when historical

data isn't available at the desired frequency, and estimates are needed to fill those temporal gaps, such as estimating daily sales figures from monthly totals.

1. Create sample monthly data using `xts`:

```
dates <- seq(as.Date("2023-01-01"), as.Date("2023-12-01"), by = "month")
monthly_sales_data <- xts(runif(12, 1000, 2000), order.by = dates)
```

2. Resample the monthly data to daily and create a new object using the `to.daily()` function for the conversion:

```
daily_sales_data <- to.daily(monthly_sales_data)
```

3. Extract the close prices fields and create a new daily close object using `$FieldName`. Here, `monthly_sales_data.Close` is the field name to use:

```
daily_data_close <- daily_sales_data$monthly_sales_data.Close
```

4. Replace the NA value by interpolation, using `na.approx`:

```
filled_daily_data <- na.approx(daily_data_close)
```

5. View the new daily sales data:

```
print(filled_daily_data)

           monthly_sales_data.Close
2023-01-01                 1507.926
2023-02-01                 1360.102
2023-03-01                 1765.300
2023-04-01                 1279.414
2023-05-01                 1854.846
2023-06-01                 1960.487
2023-07-01                 1195.902
2023-08-01                 1416.758
2023-09-01                 1924.835
2023-10-01                 1947.496
2023-11-01                 1018.545
2023-12-01                 1656.608
```

The code creates daily estimates from monthly sales figures, providing a continuous daily series.

Engaging in frequency conversion and resampling transforms your time series data, making it adaptable to various analytical frameworks. Whether condensing data for a broad analysis or expanding it to capture detail, these techniques are indispensable for homing in on the insights that matter most, all achievable with the versatile tools provided by R.

Summary

In this chapter, you've taken a deep dive into understanding the core concepts of time series data. You learned how time series data differs from other types of data by focusing on its sequential nature and regular intervals, which allows us to track changes, spot patterns, and make predictions over time.

We explored key components of time series data such as trends, seasonality, cyclic changes, and noise. You now know how these elements influence the data and how to identify and work with them. Real-world examples, from stock prices to weather patterns, showed you how time series data plays an essential role in various fields.

Additionally, you discovered the different types of time series data, including univariate and multivariate, and the distinction between continuous and discrete data. You also learned about some common challenges in time series analysis, like dealing with missing data, managing evolving datasets, and filtering out noise, all of which are crucial for accurate forecasting and decision-making.

You got hands-on experience with R, learning to create time series objects, manipulate data using essential operations like indexing, slicing, frequency conversion, and resampling. Tools like the `zoo` and `xts` packages were introduced to help you handle irregular time series data and perform advanced analyses.

By mastering these fundamental techniques, you're now equipped to explore and analyze time series data with confidence. This foundation will be key as we move into the next chapter, where we'll focus on working with dates and times in R, deepening your skills in handling temporal data for real-world projects.

3

Working with Dates and Times

Welcome to *Chapter 3*, where you will focus on mastering the manipulation and management of dates and times in R—essential skills for any data analyst working with time series data. As you dive into this chapter, you'll uncover the robust functionalities that R offers for dealing with temporal data.

From formatting and parsing dates and times to performing intricate time-based calculations, this chapter equips you with the knowledge to handle real-world data analysis challenges effectively. You will walk through practical examples and experiment with hands-on techniques to manage and transform time series data, ensuring you can apply these skills directly to your projects.

We will cover the following topics in the chapter:

- Understanding R's date-time classes
- Performing operations specific to temporal data
- Practical examples of manipulating time series

Technical requirements

R and RStudio are required for this chapter. The code for this chapter can be found in the `Chapter03` folder of the code bundle.

You will need the following data file and packages for this chapter:

- `zoo`: This popular R package was introduced in the previous chapter; you will need this library for the code in this chapter.

- `dplyr`: This R package is a powerful tool designed to make data manipulation and cleaning of tabular data (data frames) easier and faster. You will use functions like `mutate()` and `filter()` in the code snippets.
- `bitcoin_price.csv`: This is the data file we will use to demonstrate data handling, processing, and applying analytics like `rollmean()` from the zoo package.

Understanding R's date-time classes

In this section, we will dive into the backbone of time series analysis in R: the **date-time classes**. Understanding these classes is fundamental for any analysis involving time-stamped data. We'll explore the different types of date-time objects available in R, their properties, and when to use each type for optimal performance in your data tasks.

Introducing date-time classes in R

In any analysis where time plays an integral role—be it economic forecasting, weather reporting, or trend analysis—accurate management of date and time data is paramount. Date-time classes allow analysts to store dates and times in a consistent format, perform time-based calculations, and maintain chronological order in datasets.

Without the proper use of date-time classes, data can become disorganized or misleading, leading to incorrect conclusions and ineffective decisions.

R provides a comprehensive system for date and time management through its built-in classes designed specifically for these types of data. The primary classes used are:

- `Date`: This class handles dates in R and is ideal for cases where time (hours, minutes, seconds) isn't needed. It stores dates in a standard format, making it easier to perform operations like date arithmetic and sorting.
- `POSIXct` and `POSIXlt`: These two classes manage times and dates. `POSIXct` represents the number of seconds since the beginning of 1970 (also known as the Unix epoch), making it useful for arithmetic operations and straightforward time calculations. On the other hand, `POSIXlt` is a list-like structure that provides detailed components of a date and time (like hours, minutes, seconds, day of the week, and so on), allowing for more complex manipulations.

R's system for managing these date and time classes ensures that data analysis tasks can be performed with high accuracy and efficiency, enabling users to focus more on extracting insights rather than worrying about data integrity issues related to time management. This system forms the backbone of temporal data handling in R, supporting a wide range of applications and making it a versatile tool for analysts across disciplines.

The Date class

In R, the `Date` class is specifically tailored for handling dates without time components, representing dates in the standard `YYYY-MM-DD` format. This class is crucial for tasks that require precise date information but not the specific time of day, such as tracking daily transactions, recording birthdates, or scheduling events that occur on specific days.

Common functions used with Date objects

The `Date` class interacts with several functions in R designed to simplify the manipulation and formatting of date objects:

- `as.Date()`: Converts different date-time formats into `Date` objects, suitable for consistent date handling in R
- `format()`: Adjusts the display format of `Date` objects for easier reading or specific reporting needs
- `seq()`: Generates sequences of dates, useful for creating schedules or time series
- `difftime()`: Calculates the time difference between two dates, presenting the duration in various units such as days or weeks

Now, let's showcase these functions in action.

Practical examples of creating and manipulating date objects

Let's explore some practical examples of how to create and work with `Date` objects in R:

- Here's how to convert a string into a `Date` object:

  ```
  my_date <- as.Date("2023-04-15")
  print(my_date)
  [1] "2023-04-15"
  ```

  This code transforms a string into a `Date` object representing April 15, 2023.

- The following example shows how to change the format of a date to `MM/DD/YYYY`:

  ```
  formatted_date <- format(my_date, format = "%m/%d/%Y")
  print(formatted_date)
  [1] "04/15/2023"
  ```

  This operation adjusts the display of the `Date` object, making it more familiar to those used to the `MM/DD/YYYY` format.

- To create a weekly sequence of dates starting from a specific day (in this case, a weekly sequence starting from April 15, 2023, for the next ten weeks), use the following code:

```
date_sequence <- seq(from = my_date, by = "week", length.out = 10)
print(date_sequence)
[1] "2023-04-15" "2023-04-22" "2023-04-29" "2023-05-06" "2023-05-13"
[6] "2023-05-20" "2023-05-27" "2023-06-03" "2023-06-10" "2023-06-17"
```

  This sequence can be particularly useful for planning and forecasting events on a weekly basis for the next ten weeks.

- Calculating the difference in days between two dates can be seen in the following example:

```
end_date <- as.Date("2023-05-15")
date_diff <- difftime(end_date, my_date, units = "days")
print(date_diff)
Time difference of 30 days
```

  This function returns the number of days between April 15, 2023, and May 15, 2023, which can be useful for duration or interval analysis.

These examples illustrate the flexibility and utility of the `Date` class in R for handling and analyzing date-specific data efficiently and accurately.

The POSIXct and POSIXlt classes

In R, handling date and time with precision often requires understanding the differences between two main classes: `POSIXct` and `POSIXlt`. Both are used for date and time computations but serve different purposes based on the specific needs of the data analysis.

Understanding when to use each can significantly enhance the efficiency and accuracy of your data analysis tasks.

POSIXct is suitable for arithmetic operations and large datasets

`POSIXct` is ideal for scenarios requiring fast and efficient date-time calculations. It's particularly useful in environments dealing with large data sets where performance is a concern:

```
# Current date and time in POSIXct
current_time <- Sys.time() # Automatically returns POSIXct
print(current_time)
 [1] "2024-09-18 13:30:06 CEST"
```

```
# Adding one day (86400 seconds)
tomorrow <- current_time + 86400
print(tomorrow)
[1] "2024-09-19 13:30:06 CEST"
```

This code demonstrates how to retrieve the current time and add one day (86400 seconds) to it. The operation is straightforward due to `POSIXct`'s numeric nature, facilitating quick calculations without parsing detailed components.

POSIXlt offers handling of hours, minutes, and seconds

`POSIXlt` is more flexible for component-wise manipulation, allowing access to any part of the date-time:

```
# Current date and time in POSIXlt
current_time_lt <- as.POSIXlt(Sys.time())

# Changing the hour and minute
current_time_lt$hour <- 20 # Set the hour to 8 PM
current_time_lt$min <- 30 # Set the minute to 30
print(current_time_lt)
[1] "2024-09-18 20:30:48 CEST"
```

This snippet changes the hour and minute of a `POSIXlt` object, showcasing its utility when specific time components need to be adjusted independently.

Converting between POSIXct and POSIXlt

Conversion between `POSIXct` and `POSIXlt` can be necessary depending on the analysis requirements. Here's how to convert between these two classes:

- Convert `POSIXct` to `POSIXlt`:

  ```
  posixct_time <- Sys.time()
  posixlt_time <- as.POSIXlt(posixct_time)
  ```

```
print(posixlt_time)
[1] "2024-09-18 13:32:06 CEST"
```

- Convert `POSIXlt` to `POSIXct`:

```
posixct_converted <- as.POSIXct(posixlt_time)
print(posixct_converted)
[1] "2024-09-18 13:32:06 CEST"
```

This code demonstrates converting a `POSIXct` object to `POSIXlt` for detailed manipulation, and then back to `POSIXct` for efficient storage or calculations. Such conversions are crucial when your data analysis tasks require both detailed component handling and efficient arithmetic operations across large datasets.

Any programmer will tell you that dates are some of the most horrible things to deal with. However, it is time zones where the devil really lies.

Handling time zones

Time zones are important in time series analysis, particularly when handling data that spans multiple geographical locations. Mismanagement of time zones can lead to errors in data interpretation, such as erroneously comparing figures from different regions without aligning their local times. Proper time zone management ensures that events are recorded and analyzed relative to their actual occurrence time, which is crucial for tasks like coordinating global transactions, scheduling across borders, or understanding user activities in various parts of the world.

R supports robust functionalities for managing time zones through the `POSIXct` and `POSIXlt` classes, allowing for accurate time zone specification and conversion. Here's how to efficiently handle and convert time zones:

- **Setting time zones**: When creating a date-time object, you can specify the time zone directly using the `tz` parameter. If not specified, R defaults to using the system's time zone.
- **Converting time zones**: Converting the time zone of a date-time object is straightforward in R, facilitating the analysis of data collected from different time zones.

Let's walk through some examples that demonstrate how to adjust for daylight saving changes and manage time zone differences effectively:

- *Creating date-time objects with specific time zone*: Here we create a `POSIXct` object for a specific date and time set to UTC:

```
time_utc <- as.POSIXct("2023-10-01 12:00:00", tz = "UTC")
print(time_utc)
 [1] "2023-10-01 12:00:00 UTC"
```

- Converting to Eastern Standard Time (EST):

```
time_est <- format(time_utc, format = "%Y-%m-%d %H:%M:%S", tz = "America/
New_York")
print(time_est)
[1] "2023-10-01 08:00:00"
```

  This code snippet demonstrates how to specify a time zone when creating a date-time object (in UTC) and how to convert it to another time zone (Eastern Standard Time), ensuring accurate representation of the same moment in different time zones.

- *Handling daylight saving time (DST)*: Consider these two examples set around the ending of daylight saving time in New York:
  - Date before daylight saving time ends:

```
dst_date_before <- as.POSIXct("2023-11-04 12:00:00", tz = "America/
New_York")
print(dst_date_before)
 [1] "2023-11-04 12:00:00 EDT"
```

  - Date after daylight saving time ends:

```
dst_date_after <- as.POSIXct("2023-11-06 12:00:00", tz = "America/
New_York")
print(dst_date_after)
[1] "2023-11-06 12:00:00 EST"
```

  This code correctly adjusts for the daylight saving change. R automatically handles the one-hour shift associated with daylight saving time, which is essential for maintaining accurate timing in analyses that span such adjustments.

These examples underscore the necessity of precise time zone handling in R to ensure that your analyses accurately reflect local times, particularly when dealing with international data. This careful management is vital for analyses that depend on precise timing and coordination across different geographical locations.

Formatting dates and times

Formatting date-time data consistently and accurately is crucial for clarity and precision in data analysis tasks. Proper formatting ensures that date-time information is interpreted correctly by different stakeholders, from analysts to business decision-makers, and aligns with regional or sector-specific reporting standards. This section discusses the foundational aspects of how date-time data should be presented to enhance understanding and utility across various applications.

Using the format() function

The `format()` function in R is a powerful tool for displaying date-time objects according to specific needs. It allows for customization of how date-time information is presented, making it adaptable to various regional or professional standards. This subsection will explore different ways to utilize the `format()` function to tailor the appearance of date-time data, ensuring it meets the requirements of diverse analytical tasks and reporting contexts. Formatting a `POSIXct` object to display only the date or only the time, depending on your needs:

- Display only the date in the format of Year-Month-Day:

```
date_only <- format(Sys.time(), "%Y-%m-%d")
print(date_only)
 [1] "2024-09-18"
```

- Display only the time in the format of Hour:Minute:Second:

```
time_only <- format(Sys.time(), "%H:%M:%S")
print(time_only)
[1] "13:38:20"
```

These examples show how to use the `format()` function to strip down a full date-time object to either just the date or the time component. This can be useful in reports or data visualizations where full timestamp information is not required.

Customizing format to include or exclude components

When formatting a date-time object, you can provide a customized format to the `format()` function to include or exclude components like time zone or seconds:

- Format including time zone with `%Z`:

```
datetime_tz <- format(Sys.time(), "%Y-%m-%d %H:%M:%S %Z")
print(datetime_tz)
 [1] "2024-09-18 13:39:07 CEST"
```

- Format excluding seconds and time zone:

```
datetime_no_sec_no_tz <- format(Sys.time(), "%Y-%m-%d %H:%M")
print(datetime_no_sec_no_tz)
[1] "2024-09-18 13:39"
```

These snippets demonstrate custom formatting options for including additional details like the time zone or for simplifying the format by excluding seconds. Adjusting the format this way can help tailor presentations of time data to the specific requirements of an analysis or audience.

Parsing date-time strings into R objects

Converting string representations of dates and times into R's structured date-time formats is important for any analysis requiring temporal accuracy. String data often comes from various external sources like databases, CSV files, or user input and may not initially adhere to a uniform structure. Parsing these strings into standardized R date-time objects like `Date` or `POSIXct` facilitates precise calculations and comparisons and ensures that time-based operations yield valid and consistent results.

Using as.Date() and as.POSIXct()

To transform string data into usable date-time formats in R, the functions `as.Date()` for dates and `as.POSIXct()` for date-time objects are indispensable. These functions interpret and

convert character strings into R's date-time classes, allowing further data manipulation and analysis. Let's see them in detail as follows:

- `as.Date()`: Parses strings into `Date` objects, which are ideal for tasks that require date information without the time component. This function is crucial when the analysis focuses on daily, monthly, or annual trends.
- `as.POSIXct()`: Converts date and time strings into `POSIXct` objects, which include both date and time components. This function is suited for analyses that require precise time stamping down to seconds, such as intraday trading analysis or time-sensitive scientific measurements.

This is a demonstration of how to parse a column of date strings from a CSV file into `Date` objects for subsequent analysis:

1. Read the CSV file from the same directory as the R code. Load the `bitcoin_price.csv` as a CSV:

   ```
   df = read.csv("bitcoin_price.csv")
   ```

   The `read.csv` will create a data frame object from the CSV file.

2. Format the Date column and assign it back to the Date column of the data frame. `df` is a data frame loaded with a `Date` column:

   ```
   df$Date <- as.Date(df$Date, format="%Y-%m-%d")
   print(df$Date)
     [1] "2014-09-17" "2014-09-18" "2014-09-19" "2014-09-20" "2014-09-21"
     [6] "2014-09-22" "2014-09-23" "2014-09-24" "2014-09-25" "2014-09-26"
     [11] "2014-09-27" "2014-09-28" "2014-09-29" "2014-09-30" "2014-10-01"
     [16] "2014-10-02" "2014-10-03" "2014-10-04" "2014-10-05" "2014-10-06"
     [21] "2014-10-07" "2014-10-08" "2014-10-09" "2014-10-10" "2014-10-11"
     [26] "2014-10-12" "2014-10-13" "2014-10-14" "2014-10-15" "2014-10-16"
   (output truncated)
   ```

This example shows how to convert a column of date strings in a standard `YYYY-MM-DD` format into `Date` objects. Proper conversion ensures that these dates can be accurately used in chronological analyses or visualizations.

Handling common parsing issues

Let's address frequent issues encountered when parsing date-time strings, including mismatches in format specifications and values that fall outside logical ranges (for example, February 30):

1. Create a function to handle the parsing exceptions using `tryCatch` to handle errors:

```
safe_parse_date <- function(date_string)
{

  tryCatch(

    as.Date(date_string, format="%Y-%m-%d"),
    error=function(e) NA # Return NA on error

  )
}
```

   This function will try to parse and format the date string using `"%Y-%m-%d"`. If the string uses an incorrect format, it will return `NA`.

2. Apply the preceding function to parse the Date column from the data frame object:

```
parsed_dates <- sapply(df$Date, safe_parse_date)
```

This example utilizes a `tryCatch` block to gracefully handle errors during the parsing process, such as incorrect date formats or impossible dates. It attempts to parse each date and returns `NA` if the parsing fails, thereby maintaining the integrity of the dataset without halting the execution of the script.

Advanced formatting and parsing techniques

While R's `format()`, `as.Date()`, and `as.POSIXct()` functions handle many basic formatting and parsing needs, `strftime()` and `strptime()` provide additional flexibility for more complex scenarios. These functions are essential when standard approaches fall short, offering precise control over both the input and output formats of date-time data:

- `strftime()`: This function is used to convert date-time objects into formatted strings, allowing for customized presentations of date and time information. It's particularly

useful when you need to generate reports or display dates and times in user-friendly ways that deviate from standard formats.

- `strptime()`: The counterpart to `strftime()`, this function parses formatted strings into date-time objects according to specified formats. This is especially useful when dealing with data inputs that come in various complex formats, ensuring they are correctly converted into R's date-time formats for analysis.

Using strftime() to create user-friendly date-time strings

Customize the output of date-time objects to produce formats that are intuitive and easy to read for report viewers.

Formatting date-time for a user-friendly report with the following code:

```
report_date <- Sys.time()
formatted_date <- strftime(report_date, format="%A, %B %d, %Y at %H:%M")
print(formatted_date)
[1] "Wednesday, September 18, 2024 at 13:53"
```

This code snippet uses `strftime()` to format the current date and time into a string that is easy to read and suitable for inclusion in reports or user interfaces. The format includes the full weekday and month name, making the output intuitive and engaging for readers.

Tip

Use the R built-in `?` symbol to retrieve R documentation. For example, `?strftime()`. In the R documentation, you will find common format specifiers in R; for example, `%Y` for year with century, `%A` for full weekday name, and so on.

Parsing date-time strings using strptime()

Handle date-time strings that could be in various international formats, ensuring accurate parsing into R date-time objects with the following code:

```
date_string <- "Monday, 21 March 2023 14:00"
parsed_date <- strptime(date_string, format="%A, %d %B %Y %H:%M", tz="Europe/
Berlin")
print(parsed_date)
[1] "2023-03-21 14:00:00 CET"
```

This example demonstrates how to parse a date-time string that includes non-English day and month names, typical in data from German sources. The `strptime()` function is configured

with the appropriate format and locale settings to correctly interpret the string as a date-time object.

Best practices in handling date-time data

Managing date and time data efficiently in R involves adhering to several best practices to ensure data integrity and prevent common errors.

Using the right R date-time classes

Picking the right class for the date and time is an important starting point. You want to take advantage of R's powerful date-time libraries. Use the `Date Class` class for dates without time components. Use `POSIXct` and `POSlXlt` classes to help you with your project. `POSlXlt` stores values as the number of seconds since the Unix epoch, which is efficient for storage and calculation.

Here are some tips for time zone management:

- *Always specify time zones*: When working with time-sensitive data across different regions, always specify the time zone when creating date-time objects. This practice avoids ambiguity and ensures consistency across your data analysis.
- *Use UTC for storage*: Store all date-time data in Coordinated Universal Time (UTC) to maintain a standard reference time. Convert to local times only when performing user-facing operations or specific analyses that require localization.
- *Keep date and time formats consistent*: Throughout your dataset and analysis processes, maintain consistent date and time formats. This uniformity helps prevent conversion errors and simplifies the process of data aggregation and comparison.

Handling historical data that spans multiple time zones can be particularly challenging due to changes in time zone definitions and daylight saving adjustments over the years.

Here are some tips to manage such data accurately:

- *Historical time zone data*: Use libraries that handle historical changes in time zones and daylight saving rules, such as the `lubridate` and `chron` packages. These tools account for past adjustments and can accurately convert historical times across different zones.
- *Validate time zone conversions*: Regularly validate time zone conversions, especially when dealing with historical data, to ensure that the conversions still adhere to contemporary or historical local time regulations.

Consistent format and documentation

Maintaining consistency in date-time formats throughout a data analysis project is essential for ensuring the integrity and reliability of your results. Consistent formats help prevent

confusion and errors during data processing and analysis, making the data easier to understand and manipulate.

Additionally, it's crucial to document any deviations from standard formats or specific formatting rules that are used. This documentation aids in the reproducibility of the analysis and provides clarity for other team members or external reviewers who might work with the data in the future.

Properly anticipating and managing common issues in date-time formatting and parsing can significantly enhance the robustness of your data handling processes. This includes being prepared for leap years, which add an extra day to February; time zone anomalies, which can affect the accuracy of time data depending on geographic and political changes; and adjustments for daylight saving time, which can shift time data forward or backward.

Adhering to best practices in date-time formatting and parsing is crucial. By staying consistent, documenting exceptions, and being prepared for common issues, you can greatly enhance the robustness and clarity of your data analysis workflows.

Establishing enterprise standards

It is always good practice to have an enterprise level standard on date and time. This includes, but is not limited to, using the same version and same class across various projects, and for date-time objects, establishing a format standard for coding, reporting, and documentation.

Let's use the following example to show a standard date-time format across multiple teams. We will create a uniform approach to handle date-time data:

1. Define a standard format:

```
standard_datetime_format <- "%Y-%m-%d %H:%M:%S"
```

2. Define a function to use the preceding standard format:

```
format_datetime <- function(datetime) {
  format(datetime, standard_datetime_format)
}
```

3. Apply the standard format:

```
current_time <- Sys.time()
formatted_time <- format_datetime(current_time)
print(formatted_time)
[1] "2024-09-18 13:57:20"
```

This example demonstrates how to define and apply a standard date-time format across all date-time handling in a project. By using a custom function to format date-time objects, the project ensures consistency and reduces the likelihood of format-related errors.

Common issues and error checks

Working with date-time objects in R can sometimes lead to errors or unexpected results. Here are some strategies to troubleshoot common issues:

- *Invalid date errors*: Ensure all date-time inputs are valid and in the correct format before attempting to convert them into `POSIXct` or `POSIXlt` objects. Invalid dates, such as `"2023-02-30"`, can lead to errors or unexpected NA values.
- *Unexpected NA values*: If you encounter NA values after a date-time conversion, check for time zone or formatting issues. Ensure that the original data does not contain any strings or formats unrecognized by R's date-time conversion functions.
- *Performance issues with large datasets*: When working with large datasets, performance can degrade if the date-time operations are not optimized. Consider simplifying your operations, using more efficient date-time classes like `POSIXct`, or performing operations on a subset of data whenever possible
- *Debugging time zone issues*: If your date-time data behaves unexpectedly across different time zones, use R's `Sys.timezone()` to verify the current system's time zone settings and adjust your R environment or date-time objects accordingly.

By following these best practices and troubleshooting tips, you can effectively manage and analyze date and time data in R, minimizing errors and enhancing the reliability of your time series analyses.

Beside the strategies listed already, it is also best practice to enhance the robustness of date-time parsing operations by incorporating error handling mechanisms that manage unexpected or incorrect date-time inputs. The following example demonstrates how to use `tryCatch` block to handle errors:

1. Define a function to safely parse date-time strings, with error handling. When a parsing error occurs, it will generate a warning message, and return `NA`:

```
safe_parse_datetime function(datetime_string)
{
  tryCatch(
  as.POSIXct(datetime_string, format = "%Y-%m-%d %H:%M:%S"),
  error = function(e) {
  warning("Invalid datetime format: ", datetime_string)
```

```
    return(NA)
    }
    )
}
```

2. Apply the preceding function to parse a bad date:

```
# Example usage
datetime_input <- "2023-02-30 12:00:00" # Incorrect date
parsed_datetime <- safe_parse_datetime(datetime_input)
print(parsed_datetime)
[1] NA
```

This example outlines a method for parsing date-time strings with built-in error handling. If parsing fails due to an incorrect format or impossible date (such as February 30), the function issues a warning and returns `NA`, preventing the error from halting script execution and allowing for further remediation if necessary.

By adhering to these best practices in date-time formatting and parsing, analysts can ensure that their data is handled accurately and consistently across various stages of a project, ultimately leading to more reliable and valid analytical outcomes.

Next, we'll look at the specific time-related data operations we can perform in R.

Performing operations specific to temporal data

In this section, you will learn about time-based data operations, focusing on techniques specific to manipulating and analyzing temporal data in R. It's crucial for data analysts to understand these operations to effectively handle the dynamic aspects of time within datasets, whether for predicting trends, understanding periodic patterns, or performing time-sensitive calculations.

Now, why bother with time-based data operations? Because when you work with data that changes over time, you need to understand its dynamic nature to draw meaningful insights.

Temporal data, characterized by its sequential order and time-stamped entries, requires specialized analytical approaches to fully exploit its potential.

The nature of this data often necessitates operations that take into account the chronological arrangement, ensuring that patterns and changes over time are accurately captured and analyzed:

- **Time series decomposition**: This operation involves breaking down a time series into its constituent components, such as trend, seasonality, and noise. Such decomposition allows analysts to understand complex data sets by isolating simpler patterns and irregularities that could influence their analyses. We will cover more on decomposition in *Chapter 5*.
- **Lag analysis**: This operation involves comparing a time series with a lagged version of itself, analysts can uncover patterns such as cycles and lead-lag relationships between different variables. Lag analysis is fundamental in domains like economics where understanding the time delay between variables can pinpoint causal relationships. You will see more details and examples in *Chapter 5*.
- **Rolling statistics**: These involve calculating statistics over a moving window of time, providing a dynamic view of changes. Rolling means, medians, and sums are commonly used to smooth out short-term fluctuations and highlight longer-term trends in data. This technique is invaluable in financial markets analysis, where it helps to stabilize volatile series and reveal underlying trends. We will cover more details (like `rollmean()`, `rollsum()`, `rollmedian()`, and so on) with examples in later sections of this chapter.

This introduction sets the stage for a deeper exploration of how these operations are applied in real-world scenarios.

Time series arithmetic

In this segment, we explore the fundamental arithmetic operations applied to time series data: addition, subtraction, and calculation of differences. These operations are critical for adjusting data, calculating changes over time, or preparing data for further analysis.

Mastering these basic tools is essential for any analyst working with temporal data, as they allow for manipulation of the data set to better understand trends, make comparisons, and predict future values. Let's check the fundamental arithmetic operations.

Calculating the daily change in stock price

Track daily fluctuations in stock prices to gauge market volatility and investor behavior. Apply `diff` to compute the period-over-period return:

```
daily_changes <- diff(df$Adj.Close)
print(daily_changes)
```

```
  [1] -3.289401e+01 -2.964401e+01  1.410800e+01 -1.008298e+01  3.330994e+00
3.363898e+01
   [7] -1.258600e+01 -1.163098e+01 -7.149017e+00 -4.904999e+00 -2.233899e+01
-1.713989e+00
  [13]  1.147699e+01 -3.329010e+00 -8.542999e+00 -1.556000e+01 -3.064600e+01
-8.355988e+00
  [19]  9.569000e+00  6.108002e+00  1.675299e+01  1.208600e+01 -3.463989e+00
7.369995e-01
  [25]  1.625000e+01  1.186499e+01  1.045599e+01 -6.096985e+00 -1.221701e+01
1.201996e+00
(output truncated)
```

This example demonstrates how to use the `diff()` function in R to calculate daily price changes. The result, `daily_changes`, represents the daily profit or loss, providing critical information for traders and investors regarding the stock's day-to-day volatility.

Adjusting time series data for inflation or other external factors

Ensure that economic data is analyzed in real terms to account for the effects of inflation, making comparisons across different times more valid. Follow these steps:

1. Create a flat 2% inflation rate. Use the `rep()` function to replicate the 2% (0.02) based on the row count of the `bitcoin_prices` data frame:

   ```
   inflation_rates <- rep(0.02, nrow(df)) # Assuming a constant 2% inflation
   for simplicity
   ```

2. Calculate adjusted close prices based on inflation rate using `Adj.Close`:

   ```
   real_data <- df$Adj.Close / (1 + inflation_rates)
   ```

3. Create a new column called `Real.Adj.Close` and assign the value:

   ```
   df$Real.Adj.Close <- real_data
   ```

4. Display the first few rows of data:

   ```
   head(df)
          Date    Open    High     Low   Close Adj.Close   Volume
   Real.Adj.Close
   1 2014-09-17 465.864 468.174 452.422 457.334   457.334 21056800
   448.3667
   ```

```
2 2014-09-18 456.860 456.860 413.104 424.440   424.440 34483200
416.1176
3 2014-09-19 424.103 427.835 384.532 394.796   394.796 37919700
387.0549
4 2014-09-20 394.673 423.296 389.883 408.904   408.904 36863600
400.8863
5 2014-09-21 408.085 412.426 393.181 398.821   398.821 26580100
391.0010
6 2014-09-22 399.100 406.916 397.130 402.152   402.152 24127600
394.2667
```

By dividing the adjusted closing stock prices by one plus the adjustment rate, this method accounts for external factors and provides a value that reflects the stock's true performance over time without distortions.

This example highlights how basic arithmetic operations are applied to time series data to facilitate deeper analysis and understanding. Such techniques are indispensable tools in the arsenal of an analyst dealing with time-sensitive data across various sectors.

Window functions and rolling calculations

Window functions and rolling calculations are pivotal in time series analysis for smoothing or accentuating trends in data. These techniques apply statistical functions to subsets of data within a moving window, allowing analysts to reduce noise and better discern underlying patterns in the data.

This is especially crucial in fields such as finance, meteorology, and market analysis, where recognizing trends and cyclical behavior over time can guide critical decisions.

Rolling calculations, involving rolling means, medians, and sums, are performed over a specified moving window that progresses through the dataset. This approach is integral for analyzing trends and patterns over defined intervals, providing a dynamic view of changes. There are three kinds of rolling calculations:

- **Rolling mean**: Helps in smoothing out short-term fluctuations and highlighting longer-term trends
- **Rolling median**: Offers a robust measure against data outliers, maintaining a central tendency that is not skewed by extreme values
- **Rolling sum**: Utilized for aggregating values over a period, useful in accumulating totals such as weekly sales or monthly expenses

The following list is a set of examples demonstrating how rolling calculations can be applied to stock data to analyze trends, aggregate performance, and smooth out daily fluctuations, providing deeper insights into market behavior and aiding in more informed decision-making:

- *Calculating a 7-Day moving average of stock prices*: Smooth out daily stock price variations to uncover broader market trends. Calculate a 7-day moving average for `Adj.Close` in the DataFrame:

```
library(zoo)
rolling_avg <- rollmean(df$Adj.Close, 7, fill = NA, align = "center")
print(rolling_avg)
[1]        NA        NA        NA  417.4626  412.5870  410.7490  412.1246
410.7840
[9]  407.6926  403.8804  396.9023  391.2466  386.0320  379.6159  369.5224
361.4266
[17]  354.9426  347.6916  343.3094  341.8743  342.1671  346.9433  355.2346
363.8539
```

  This example utilizes the `rollmean` function from the `zoo` package to compute a 7-day moving average for the adjusted closing prices, centering the window around each day to provide a balanced view of the stock price trend.

- *Using rolling sums to monitor weekly volume performance*: Aggregate the trading volume on a weekly basis to assess market activity and investor interest. Calculate a rolling sum for the `Volume` column over a 7-day window:

```
weekly_rolling_sum <- rollsum(df$Volume, 7, fill = NA, align = "right")
print(weekly_rolling_sum[1:15])
[1]        NA        NA        NA        NA        NA        NA 226130500
235701400
 [9] 228032600 211573700 189739400 186772600 195142700 184750500 180352200
```

By applying the `rollsum` function, this method computes the total trading volume for each week, aligning the window to end on the day of calculation and thereby summarizing the market activity for the prior seven days.

- *Determining weekly median of stock prices using rolling median*: Track the median adjusted closing price over a week to mitigate the impact of unusually high or low days. Calculate a rolling median for `Adj.Close` over a 7-day window:

```
weekly_median_prices <- rollmedian(df$Adj.Close, 7, fill = NA, align =
"center")
print(weekly_median_prices[1:5])
[1]      NA      NA      NA 408.904 408.904
```

The `rollmedian` function is employed here to calculate a rolling median across a 7-day window, centered around each day. This approach provides a more stable indicator of the typical stock price performance, smoothing out the effects of any anomalous spikes or dips in daily prices.

These examples illustrate how rolling calculations can be applied to financial data to provide smoother analysis and clearer insights into market trends. By using these methods, analysts gain a more nuanced understanding of stock behavior over time, facilitating informed investment decisions and strategic planning.

Time shifts and date adjustments

Time shift operations allow analysts to adjust the temporal alignment of datasets for various purposes. These adjustments can be forward or backward shifts in time, which are essential for synchronizing data from different sources, simulating future scenarios, or correcting timing errors in data entry.

Effective use of time shifts ensures that datasets are comparable, and that analyses are relevant and accurate.

Employ time shift techniques to realign time series data, which is fundamental for comprehensive analyses where temporal alignment affects outcomes such as financial analysis or operational planning.

Adjusting dates within time series data to account for non-standard working days, public holidays, or other calendar-specific effects is another critical aspect of handling temporal data. These adjustments are vital for operations planning, financial forecasting, and any analysis where business days or operational days differ from calendar days.

This practice is essential in ensuring that the data aligns with real-world conditions, avoiding misinterpretations that could arise from including non-operational days in the analysis.

In financial markets, weekends and public holidays often need to be accounted for since trading activity ceases during these periods, requiring adjustments to ensure accurate analysis:

- *Shifting stock prices to adjust for weekend non-trading days*. Ensure stock price analysis reflects actual trading days by excluding weekends when the market is closed:

```
#install.packages("dplyr")
if (!requireNamespace("dplyr", quietly = TRUE)) {
  # If not installed, install it
  install.packages("dplyr")
}
# load package dplyr
library(dplyr)

# mutate function is used to create or modify a data frame column
# if_else(condition, true, false), here if the date is either Sat or Sun,
set Close value to NA, otherwise use Adj.Close
# %>% is R forward pipe operator
# %in% is for value matching
adjusted_stock_data <- df %>%
  mutate(Adjusted.Close = if_else(weekdays(as.Date(Date)) %in%
c("Saturday", "Sunday"), NA, Adj.Close))
print(adjusted_stock_data)
```

This will print out the new `adjusted_stock_data` data frame object for the date `2014-09-20`, which is a Saturday. The `Adjusted.Close` column value is `NA`, as you can see in *Figure 3.1*.

Date	Open	High	Low	Close	Adj.Close	Volume	Real.Adj.Close	Adjusted.Close
2014-09-17	465.864	468.174	452.422	457.334	457.334	21056800	448.3667	457.334
2014-09-18	456.860	456.860	413.104	424.440	424.440	34483200	416.1176	424.440
2014-09-19	424.103	427.835	384.532	394.796	394.796	37919700	387.0549	394.796
2014-09-20	394.673	423.296	389.883	408.904	408.904	36863600	400.8863	*NA*
2014-09-21	408.085	412.426	393.181	398.821	398.821	26580100	391.0010	*NA*
2014-09-22	399.100	406.916	397.130	402.152	402.152	24127600	394.2667	402.152
2014-09-23	402.092	441.557	396.197	435.791	435.791	45099500	427.2461	435.791
2014-09-24	435.751	436.112	421.132	423.205	423.205	30627700	414.9068	423.205
2014-09-25	423.156	423.520	409.468	411.574	411.574	26814400	403.5039	411.574
2014-09-26	411.429	414.938	400.009	404.425	404.425	21460800	396.4951	404.425
2014-09-27	403.556	406.623	397.372	399.520	399.520	15029300	391.6863	*NA*
2014-09-28	399.471	401.017	374.332	377.181	377.181	23613300	369.7853	*NA*

Figure 3.1: The stock price adjusted for weekend, showing NA

This example demonstrates how to use the `dplyr` package to adjust stock price records, setting weekend data to `NA` since no trading occurs on these days. This adjustment helps maintain the integrity of financial metrics by ensuring that analysis only includes days when the market is open for trading.

- *Adjusting stock prices to accommodate public holidays.* Modify the dataset to exclude stock prices for public holidays when the market is closed, ensuring a more accurate reflection of trading activity:

```
# Assuming 'public_holidays' includes Christmas and other holidays
public_holidays <- c("2014-12-25") # Christmas in 2014

# Adjusting the DataFrame
adjusted_for_holidays <- df %>%
  mutate(Adjusted.Close = if_else(as.Date(Date) %in%
as.Date(public_holidays), NA, Adj.Close))

# Print the rows where the date matches public holidays
holidays_adjusted <- adjusted_for_holidays %>%
  filter(as.Date(Date) %in% as.Date(public_holidays))

print(holidays_adjusted)
       Date    Open    High     Low   Close Adj.Close  Volume Real.Adj.Close
1 2014-12-25 322.286 322.67 316.958 319.008  319.008  9883640  312.7529
```

```
  Adjusted.Close
1             NA
```

This code adjusts the `Adj.Close` column to `NA` for the specified public holidays, like Christmas, and then prints the rows in which these adjustments were made.

These examples of time shifts and date adjustments illustrate critical techniques for aligning and correcting time series data in contexts where timing and precision play significant roles in analysis and operational planning. By adjusting for weekends and public holidays, analysts can ensure that their evaluations accurately reflect the true nature of the data.

Next, you will see a practical example of how to manipulate time series data.

Practical examples of manipulating time series data

This section dives into the hands-on world of time series data wrangling, providing practical guidance and real-world examples to enhance your ability to manipulate and analyze time series data effectively. Time series data wrangling involves techniques that prepare raw data for analysis, from cleaning and normalization to more complex transformations that are crucial for making the data usable and meaningful.

Data cleaning and normalization

Data cleaning and normalization are crucial when working with time series data. In the context of financial markets, raw data often contains anomalies like non-trading days, holidays, and market outliers that must be handled before analysis. Failing to address these irregularities can lead to skewed results and misleading insights.

Therefore, it's essential to apply techniques that adjust the dataset to reflect the real-world conditions under which the data was generated, ensuring that subsequent analyses and models provide meaningful and accurate results.

This section focuses on the vital initial steps in time series data wrangling, which involve cleaning anomalies or errors from data and normalizing it to a standard format. These practices are essential to ensure that the dataset is accurate, consistent, and ready for advanced analysis and modeling.

Techniques covered for data cleaning and normalization are as follows:

- Handling missing values by imputation or removal: Tackle gaps in time series data that could skew analysis and lead to inaccurate results. This can be dealt with via:
    - **Imputation**: Utilize statistical methods to fill missing data points, thus preserving the integrity of the data sequence
    - **Removal**: Opt to remove data segments where significant data is missing, applicable when the absence of data does not impact the overall dataset integrity
- Normalizing data to a common scale: Convert data values to a uniform scale, facilitating unbiased comparisons and analyses across different datasets or variables. Our methods for this include:
    - **Min-Max scaling**: Adjust data to a specific range, typically 0 to 1, which helps eliminate scale discrepancies while maintaining the structure of the dataset
    - **Z-score normalization (standardization)**: Normalize the distribution of data so that it has a mean of zero and a standard deviation of one, effectively reducing bias from outliers

Imagine that you're dealing with daily stock prices, like in the `df` we have been working on; it will have data that has missing values for some non-trading days and outliers due to market anomalies. To handle this effectively, you need a strategy that addresses these issues head-on, ensuring the data remains reliable and ready for analysis.

Let's practice these concepts in the following exercise:

1. Replace NA with the interpolated value. This line of code fills in missing stock prices in the `Adj.Close` column by interpolating between available data points:

```
df$Adj.Close <- na.approx(df$Adj.Close)
```

2. Display the first few rows:

```
cat("Data after linear interpolation for missing values:\n")
Data after linear interpolation for missing values:
print(head(df))
       Date    Open    High     Low   Close Adj.Close   Volume
Real.Adj.Close
1 2014-09-17 465.864 468.174 452.422 457.334   457.334 21056800
448.3667
2 2014-09-18 456.860 456.860 413.104 424.440   424.440 34483200
416.1176
```

```
3 2014-09-19 424.103 427.835 384.532 394.796   394.796 37919700
387.0549
4 2014-09-20 394.673 423.296 389.883 408.904   408.904 36863600
400.8863
5 2014-09-21 408.085 412.426 393.181 398.821   398.821 26580100
391.0010
6 2014-09-22 399.100 406.916 397.130 402.152   402.152 24127600
394.2667
```

3. Get the z-score and create a new column, `Normalized.Adj.Close`:

```
df$Normalized.Adj.Close <- (df$Adj.Close - mean(df$Adj.Close, na.rm =
TRUE)) / sd(df$Adj.Close, na.rm = TRUE)
```

4. Display the first couple rows with the new column after normalization:

```
cat("\nData after Z-score normalization:\n")
Data after Z-score normalization:
print(head(df))

       Date    Open    High     Low   Close Adj.Close   Volume
Real.Adj.Close
1 2014-09-17 465.864 468.174 452.422 457.334   457.334 21056800
448.3667
2 2014-09-18 456.860 456.860 413.104 424.440   424.440 34483200
416.1176
3 2014-09-19 424.103 427.835 384.532 394.796   394.796 37919700
387.0549
4 2014-09-20 394.673 423.296 389.883 408.904   408.904 36863600
400.8863
5 2014-09-21 408.085 412.426 393.181 398.821   398.821 26580100
391.0010
6 2014-09-22 399.100 406.916 397.130 402.152   402.152 24127600
394.2667
  Normalized.Adj.Close
1           -0.8722892
2           -0.8743216
3           -0.8761533
4           -0.8752816
```

```
5           -0.8759046
6           -0.8756988
```

This approach helps ensure that the financial data is continuous and standardized, facilitating more accurate trend analysis and financial forecasting.

Feature engineering

This section focuses on enhancing the predictive power of models by creating new features from raw time series data. Feature engineering is a critical step in data analysis, especially in time series modeling, where properly designed features can significantly improve model performance by capturing important aspects of the data's underlying patterns and dynamics. We have the features as follows:

- *Deriving lagged features*: Utilize past values within the time series to predict future outcomes, leveraging the inherent temporal dependencies in the data.
  Lagged features involve shifting the time series data backward by one or more time steps to create new features that represent previous values. This technique is particularly useful in forecasting scenarios where past behavior is indicative of future trends.
- *Rolling window features*: Extract insights from the rolling calculations of basic statistics over a fixed window, which helps to understand local variations and overall trends. Rolling window features, such as moving averages or rolling standard deviations, provide a smoothed or dynamic view of the data's changes over time. These features are effective at capturing short-term fluctuations and long-term patterns, aiding in the detection of volatile behavior or stable trends in the data.
- *Implementing lagged features for stock price prediction*. Predict future stock prices using information about previous price movements, assuming `df` is a dataframe with daily adjusted closing prices in `Adj.Close`:

```
df$lag_1 <- lag(df$Adj.Close, 1)
df$lag_7 <- lag(df$Adj.Close, 7)
print(head(df,10))
```

This will generate two new columns, `lag_1` and `lag_7`, as displayed in the following figure:

	Date	Open	High	Low	Close	Adj.Close	Volume	Real.Adj.Close	Normalized.Adj.Close	lag_1	lag_7
1	2014-09-17	465.864	468.174	452.422	457.334	457.334	21056800	448.3667	-0.8722892	NA	NA
2	2014-09-18	456.860	456.860	413.104	424.440	424.440	34483200	416.1176	-0.8743216	457.334	NA
3	2014-09-19	424.103	427.835	384.532	394.796	394.796	37919700	387.0549	-0.8761533	424.440	NA
4	2014-09-20	394.673	423.296	389.883	408.904	408.904	36863600	400.8863	-0.8752816	394.796	NA
5	2014-09-21	408.085	412.426	393.181	398.821	398.821	26580100	391.0010	-0.8759046	408.904	NA
6	2014-09-22	399.100	406.916	397.130	402.152	402.152	24127600	394.2667	-0.8756988	398.821	NA
7	2014-09-23	402.092	441.557	396.197	435.791	435.791	45099500	427.2461	-0.8736203	402.152	NA
8	2014-09-24	435.751	436.112	421.132	423.205	423.205	30627700	414.9068	-0.8743980	435.791	457.334
9	2014-09-25	423.156	423.520	409.468	411.574	411.574	26814400	403.5039	-0.8751166	423.205	424.440
10	2014-09-26	411.429	414.938	400.009	404.425	404.425	21460800	396.4951	-0.8755583	411.574	394.796

Figure 3.2: Stock price data frame with new columns after applying the lag function

These lagged features can help a model learn the patterns from the previous days' prices, which are often predictive of the next day's price, especially in volatile markets.

- *Using rolling window features to analyze retail sales.* Understand trends and volatility in daily sales data:
    - Calculate a 7-day moving average for `Adj.Close`:

```
df$moving_average_7 <- rollmean(df$Adj.Close, 7, fill = NA)
```

    - Calculate a 30-day rolling standard deviation for `Adj.Close`:

```
df$rolling_std_30 <- rollapply(df$Adj.Close, 30, sd, fill = NA)
```

    - Display the first few rows after adding rolling window features:

```
print(head(df[10:13],10))
```

This will display the following four columns and the first 10 rows, as shown in *Figure 3.3*.

	lag_1	lag_7	moving_average_7	rolling_std_30
1	NA	NA	NA	NA
2	457.334	NA	NA	NA
3	424.440	NA	NA	NA
4	394.796	NA	417.4626	NA
5	408.904	NA	412.5870	NA
6	398.821	NA	410.7490	NA
7	402.152	NA	412.1246	NA
8	435.791	457.334	410.7840	NA
9	423.205	424.440	407.6926	NA
10	411.574	394.796	403.8804	NA

Figure 3.3: Data frame with news columns after rolling window feature is applied

The 7-day moving average smooths out daily fluctuations and shows a clearer trend, while the 30-day rolling standard deviation highlights periods of higher and lower volatility in sales figures.

By incorporating these feature engineering techniques into your data analysis workflow, you can create more robust and accurate predictive models. These features help capture essential dynamics in the data, making your models more sensitive to real-world phenomena and patterns.

Summary

In this chapter, you've gained a solid foundation in managing dates and times in R, an indispensable skill for any data analyst working with time series data. You delved into R's core date-time classes—`Date`, `POSIXct`, and `POSIXlt`—learning how to handle and manipulate time-stamped data efficiently. This understanding forms the backbone of any temporal data analysis, equipping you to manage the complexities of time-based datasets.

Through hands-on examples, you acquired practical skills for formatting, parsing, and performing calculations with date-time objects. These exercises have prepared you to apply these techniques directly in your data analysis tasks, enabling you to process and interpret time series data with greater accuracy. You also explored how to manage time zones, ensuring that your analysis is accurate and globally relevant, which is essential when working with international datasets.

The chapter also introduced you to advanced feature engineering techniques, such as creating lagged and rolling window features. These methods allow you to extract deeper insights from your data, particularly in predictive modeling and financial analysis, where understanding

historical trends is crucial for forecasting. Equipped with these strategies and best practices, you are now better prepared to maintain data integrity and tackle common challenges in date-time data analysis.

In the next chapter, you'll dive into the art of visualizing time series data using R. This journey will guide you through the various techniques and tools available to make your data tell a compelling story. Starting with an overview of R's built-in plotting functions, you'll learn the basics of creating insightful visualizations.

4

Visualizing Time Series Data

Visualizing time series data is crucial for uncovering trends, seasonal patterns, and anomalies, making it easier to interpret temporal changes. R offers an extensive toolkit for time series visualization, ranging from basic plotting functions to powerful, customizable packages like `ggplot2` and interactive frameworks like `shiny` and `plotly`.

This chapter guides you through the core techniques, including how to generate simple line plots, customize visuals for clarity, and enhance interactivity for deeper data exploration. We'll also address common pitfalls and best practices to ensure your visualizations are accurate and insightful.

In this chapter we'll cover the following topics:

- Overview of plotting in R
- Using ggplot2 for time series visualization
- Interactive visualizations with shiny and plotly
- Common pitfalls in visualizing time series

Technical requirements

R and RStudio are required for this chapter. The code for this chapter can be found in the `Chapter04` folder.

You will need the following data file and packages for this chapter:

- `ggplot2`: This popular R package is based on Leland Wilkinson's *Grammar of Graphics*. This package is an open-source and powerful tool for creating elegant and versatile visualization. You will see a lot of usage for the code in this chapter.

- `reshape2`: This R package is used for flexibly restructuring (melting and casting) data between "wide" and "long" formats.
- `plotly`: This R package is a common tool used for creating interactive visualization; you will see code snippets in this chapter.
- `Shiny`: This is the library that makes it easy to build interactive applications and dashboards directly from R code.

Overview of plotting in R

R is a versatile programming language that is highly regarded for its statistical computing and graphical capabilities. It has become a go-to tool for statisticians and data analysts to create custom statistical software and perform data analysis. The language supports a broad array of methods for data manipulation, computations, and visualization, while its open-source nature allows for the development of additional packages that enhance its functionality. With these packages, R can handle everything from linear and nonlinear modeling to statistical testing, time-series forecasting, classification, and clustering.

One of R's strengths is its vast array of built-in plotting commands and the flexibility to enhance these plots. Here's a brief overview of some fundamental plotting functions that are integral to visualizing datasets in R:

- `plot()`: This is the generic function for plotting in R. It can produce various forms of graphs, depending on the nature of the data and the desired display. For time series data, `plot()` can be used to create a simple line plot showing data points connected over time intervals.
- `hist()`: Useful for creating histograms, `hist()` helps in understanding the distribution of continuous variables. It is an excellent tool for quickly examining and visualizing the shape of data distributions in a dataset.
- `boxplot()`: This function displays a box and whisker plot which is useful for visually summarizing data from multiple categories side by side. It can be particularly informative for comparing distributions and spotting outliers.
- `barplot()`: As the name suggests, `barplot()` is used to produce bar charts, which are helpful for comparing quantities corresponding to different groups or categories.
- `pie()`: For displaying proportional data as slices of a pie, the `pie()` function can be used. It helps in showing the composition of something as parts of a whole.

These functions represent just the beginning of what's possible with plotting in R. Each function can be customized with additional parameters to modify aesthetics such as colors,

labels, titles, and much more, enabling users to create not only informative but also attractive visualizations.

Plotting time series data

First, we need to generate some dummy time series data. We'll simulate daily temperatures over a one-month period by going through the following code:

1. First, we will create a sequence of 31 days:

```
dates <- seq(as.Date("2024-01-01"), by="day", length.out=31)
```

2. Generate random temperature for 31 days:

```
temperatures <- rnorm(31, mean=22, sd=3) # Random data centered around 22
degrees Celsius
```

   The preceding code will use the `rnorm()` term to generate 31 temperatures, with mean value of 22, and a standard deviation of 3.

3. Combine the days and temperatures into a data frame:

```
temperature_data <- data.frame(Date=dates, Temperature=temperatures)
```

   The resulting data frame looks as follows, containing date and corresponding temperature values:

Date	Temperature
2024-01-01	28.72670
2024-01-02	21.16706
2024-01-03	21.82293
2024-01-04	17.19017
2024-01-05	16.66427
2024-01-06	17.21478
2024-01-07	18.08974
2024-01-08	20.32106
2024-01-09	22.31221
2024-01-10	21.16659

Figure 4.1: Example of temperature data over a 10-day period

Ensure the **Date** column is formatted as a Date type, although in this script, it's explicitly created as such. It's good practice to check.

4. Display the structure of the data frame object:

```
str(temperature_data)
'data.frame':       31 obs. of  2 variables:
 $ Date       : Date, format: "2024-01-01" "2024-01-02" ...
 $ Temperature: num  21.9 15.3 20.3 20.9 29.9...
```

5. Now, we will use R's base `plot()` function to create a line plot of the temperature data:

```
plot(temperature_data$Date, temperature_data$Temperature, type="l",
xlab="Date", ylab="Temperature (°C)",
 main="Simulated Daily Temperatures Over Time", col="blue", lwd=2)
```

Figure 4.2 shows the output. The plot visualizes the daily temperatures over a 10-day period, showcasing the fluctuations in temperature values.

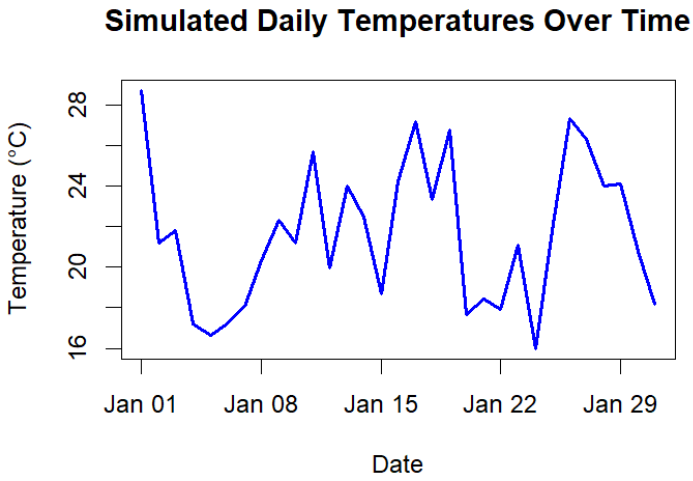


Figure 4.2: Simulated daily temperature fluctuations over time

The code uses the `plot()` function to visualize the temperature data. The following list is a breakdown of the parameters used:

- `type="l"` creates a line plot
- `xlab` and `ylab` provide labels for the x-axis and y-axis
- `main` adds a title to the plot
- `col="blue"` sets the color of the line
- `lwd=2` increases the line width for better visibility

Now, let's move beyond the basics.

Enhancing basic plots

Enhancing the visual appeal and readability of basic plots is crucial in effectively communicating the underlying patterns and insights of your data. Here are several techniques to enhance time series plots in R, including how to add titles, labels, legends, adjust colors and line types, and handle missing data points:

- *Adding titles, axes labels, and legends*: Adding titles and labels to your plots is essential for context. It helps the viewer quickly understand the plotted data without needing to dig into the raw data themselves:

  ```
  plot(temperature_data$Date, temperature_data$Temperature, type="l",
   main="Daily Temperatures Over Time",
   xlab="Date",
   ylab="Temperature (°C)",
   col="blue")
  ```

  The following plot shows the daily temperature variations over time, with appropriate titles and labels added for clarity:

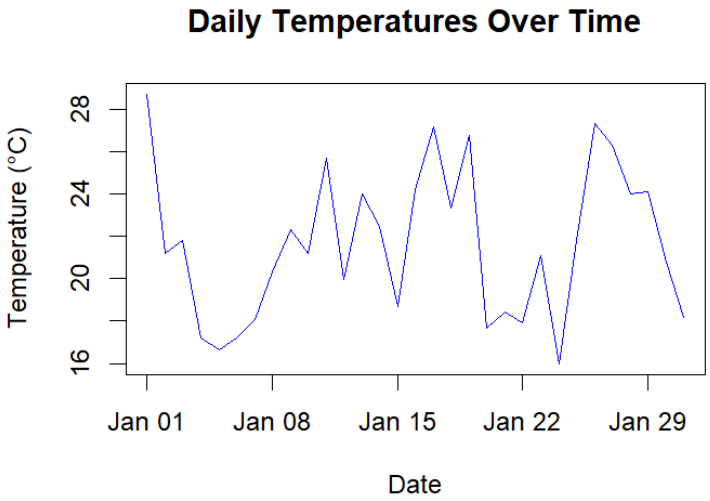


Figure 4.3: A line plot showing daily temperature fluctuations

Legends are crucial when multiple datasets or plot elements (like lines or points) are included in the same plot. They help distinguish between different series or categories. See the following code block for an example:

```
legend("topright", legend=c("Observed Temperature"), col=c("blue"), lty=1:
1, cex=0.8)
```

The updated plot, *Figure 4.4*, now includes a legend, now includes a legend for the observed temperature data line:

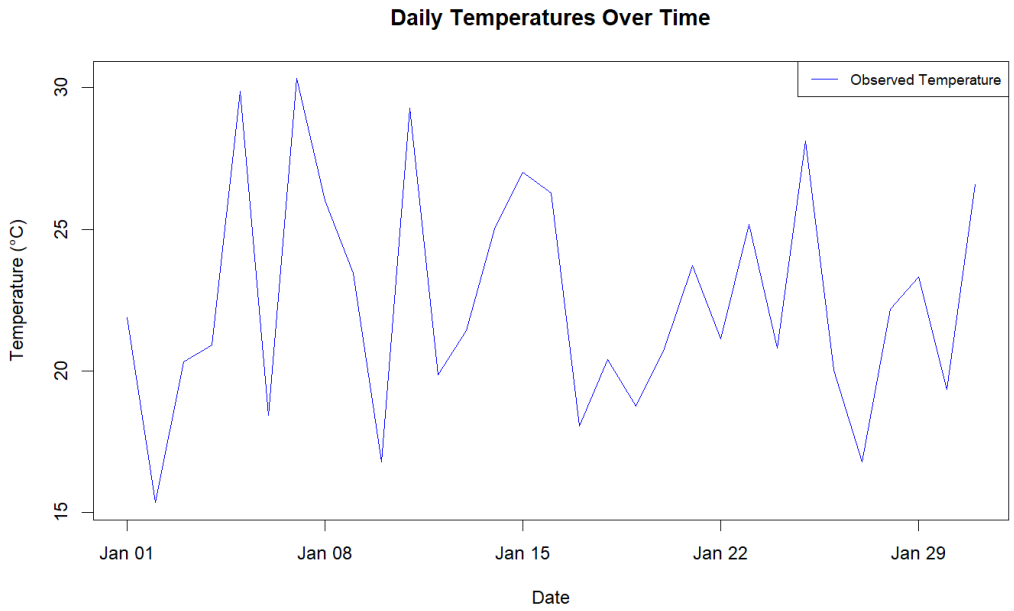


Figure 4.4: Line plot of daily temperatures with a legend

- *Adjusting colors and line types*: Changing the color of your plots can not only improve aesthetics but also enhance readability, especially when plots are dense or contain multiple categories:

```
plot(temperature_data$Date, temperature_data$Temperature, type="l",
col="darkgreen")
```

The following plot illustrates the same temperature data, but with the line color adjusted to dark green for better visibility:

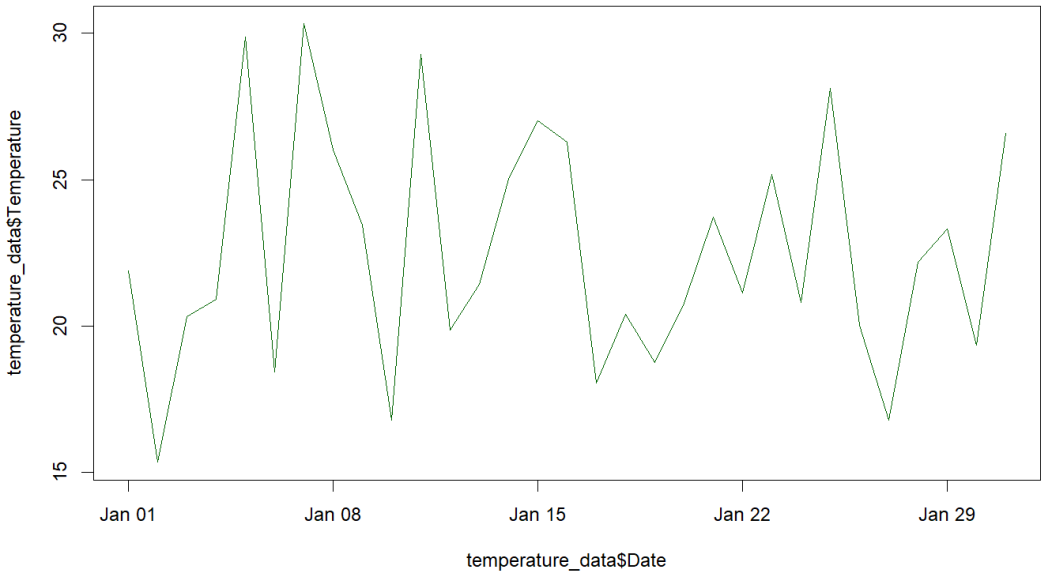


Figure 4.5: A line plot of daily temperatures with the line color changed

Adjusting the line type (solid, dashed, dotted, and so on) can help differentiate between different data series on the same plot:

```
plot(temperature_data$Date, temperature_data$Temperature, type="l", lty=2,
col="darkgreen")
```

The following plot demonstrates how adjusting the line type to a dashed style can change the appearance of the data and make the visualization more distinct.

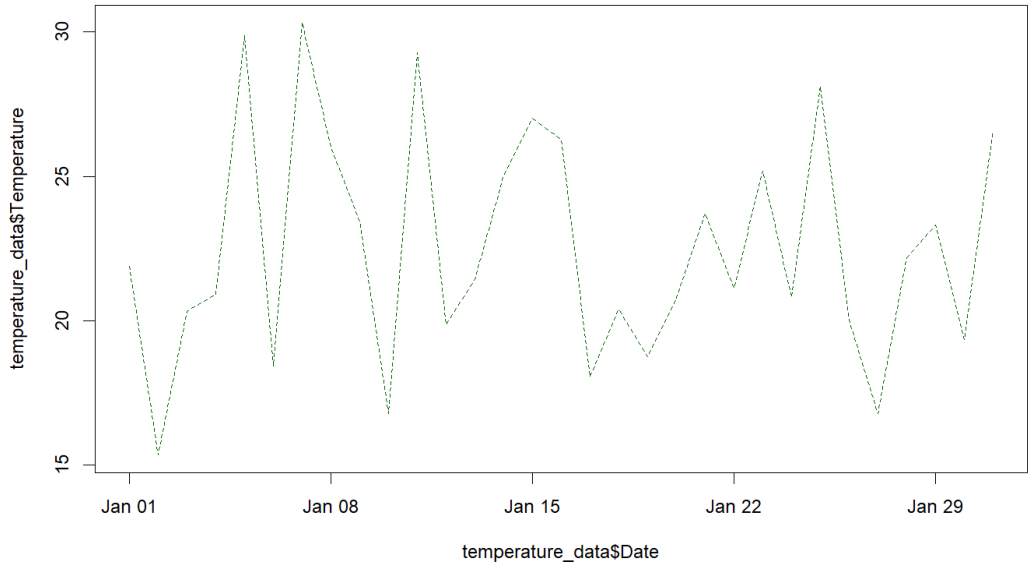


Figure 4.6: A line plot of daily temperatures with a dashed line type for better distinction

Enhancing your plots not only makes them more visually appealing but also more effective in communicating the story behind the data. With these techniques, you can turn basic plots into insightful visual narratives that clearly and attractively convey important information.

Of course, this is just the start. Wait until you try specialized libraries for data visualization.

Using ggplot2 for time series visualization

This section of the chapter focuses on utilizing the **ggplot2** package in R, a powerful tool for creating aesthetically pleasing and complex visualizations. This section will guide readers through the basics of `ggplot2`, how to create advanced time series plots, and ways to customize these plots for more effective data presentation.

`ggplot2`, one of the most powerful and flexible visualization libraries available in R, is based on *The Grammar of Graphics*—a unique approach to structuring graphics that breaks down plots into semantic components such as scales and layers. This framework allows users to think about graphics in a layered manner, where each component of a plot is added in a step-by-step process. It not only aids in building complex visualizations systematically but also encourages thoughtful design and coherent storytelling through data.

Here are some advantages of using ggplot2 instead of base R graphics:

- **Layering principle**: Unlike base R plots, ggplot2 allows you to build plots layer by layer. You can start with the data, add layers of annotations and statistical summaries, and adjust with scales, coordinates, and themes. This method offers unmatched flexibility in plot customization and complexity.
- **Extensive customization**: ggplot2 provides an extensive range of options for fine-tuning the appearance of plots, from adjusting legends, axes, and plot backgrounds to using different color palettes and typography. This capability makes it easier to match the visual output to the stylistic guidelines of different publications or presentations.
- **Consistent coding syntax**: Using ggplot2 involves a consistent and predictable coding syntax, making it easier to learn and use, especially for complex visualizations. This consistency also aids in readability and maintenance of code, particularly in larger projects involving data visualization.

Now, we will walk through the setup of ggplot2 and demonstrate how to create different types of visualizations using it.

Installation and setup of ggplot2

To start using ggplot2, you must first install it from CRAN. If you haven't installed ggplot2 yet, you can do so using the following command in your R console:

```
# Check if the ggplot2 package is already installed
if (!requireNamespace("ggplot2", quietly = TRUE)) {
  #If not installed, install it
  install.packages("ggplot2")
}
```

Once installed, load ggplot2 into your R session to access its functions:

```
library(ggplot2)
```

There are, however, some dependencies and recommended packages you should have too:

- dplyr: Often used alongside ggplot2 for its powerful data manipulation capabilities. It helps in preparing and restructuring data for visualization.

- `scales`: Useful for tweaking the scales used by `ggplot2`, such as customizing date-time scales or creating your own palette for color scales.
- `ggthemes`: Offers additional themes and scaling functions for `ggplot2` to further customize the look and feel of your plots.

By installing `ggplot2` and familiarizing yourself with its philosophy and capabilities, you can significantly enhance the quality and effectiveness of your data visualizations. This setup not only prepares you for basic plotting tasks but also sets the stage for creating advanced, publication-quality graphics.

Creating advanced time series plots

To begin visualizing time series data using `ggplot2`, you first need to ensure that your data is in the correct format. `ggplot2` works best with data in a data frame where each column is a variable and each row is an observation.

For time series, you typically need a date or time column and one or more columns representing the measurement over time.

Suppose you have a data frame named `temp_data` with columns `Date` (in `Date` format) and `Temperature` (numeric values). Here's how you can create a simple line plot:

1. Create a data frame of 30 rows of dates and temperatures:

```
# Example data frame
temp_data <- data.frame(
 Date = seq(as.Date("2024-01-01"), by = "day", length.out = 30),
 Temperature = rnorm(30, mean = 20, sd = 5)
)
```

2. Generate a line plot using the `ggplot` function in the `ggplot2` package. The `+` operator adds layers to the plot:

```
ggplot(data = temp_data, aes(x = Date, y = Temperature)) +
 geom_line() +
 labs(title = "Daily Temperature Over Time",
 x = "Date",
 y = "Temperature (°C)") +
 theme_minimal()
```

   The following plot is generated using `ggplot2` and illustrates daily temperature fluctuations over a specified time period, with minimal styling for a clean look.

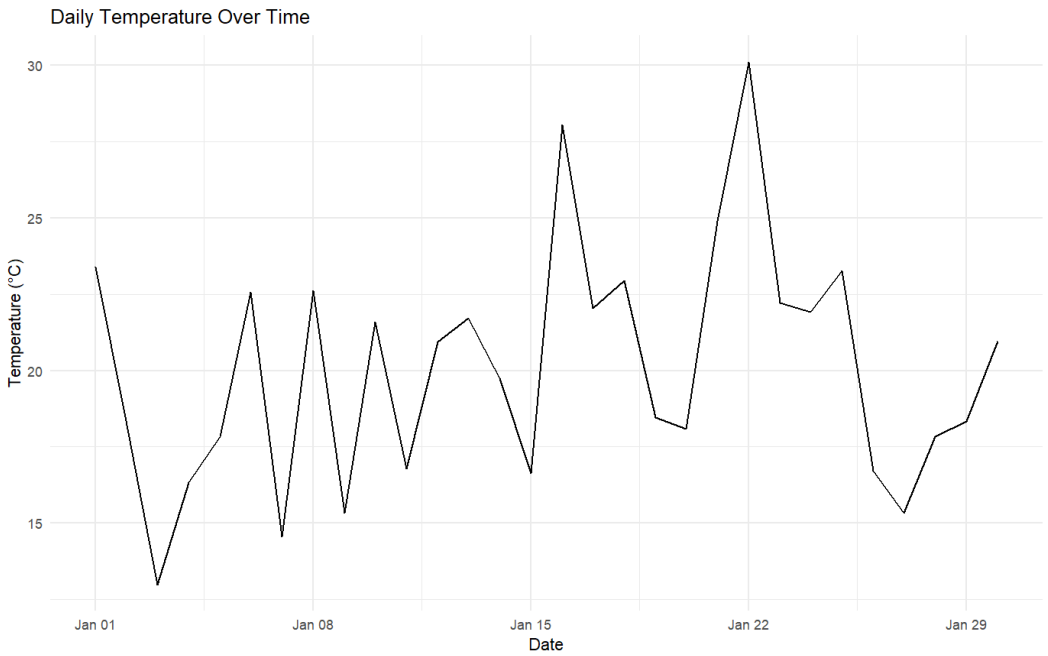


Figure 4.7: A line plot of daily temperature variations over time, created using ggplot2

In *Figure 4.7*:

- `aes(x = Date, y = Temperature)` sets the aesthetics of the plot, mapping `Date` to the x-axis and `Temperature` to the y-axis
- `geom_line()` adds a line plot layer
- `labs()` and `theme_minimal()` are used to add labels and apply a minimalistic theme

Let's look at some more types of plots you can generate.

Advanced plot types

To compare several time series on a single graph, you can add another categorical variable to your data frame that identifies each time series. Here's how you can plot multiple time series:

1. Create a new column called `Temperature2`, simulating another time series:

```
temp_data$Temperature2 = rnorm(30, mean = 15, sd = 5)
```

2. Install and load the `reshape2` package:

```
# Check if the reshape2 package is already installed
if (!requireNamespace("reshape2", quietly = TRUE)) {
  #If not installed, install it
  install.packages("reshape2")
}

# Load reshape2 package
library(reshape2)
```

3. Melt the data for `ggplot2`:

```
temp_data_melted <- melt(temp_data, id.vars = "Date")
```

4. Plot the two temperature columns in a line plot:

```
ggplot(data = temp_data_melted, aes(x = Date, y = value, color = variable))
+
 geom_line() +
 labs(title = "Comparison of Two Temperature Time Series",
 x = "Date",
 y = "Temperature (°C)")
```

The following plot shows a comparison of two temperature time series, where the blue and red lines represent different datasets. This allows us to observe how the temperatures vary between the two series over the same time period.

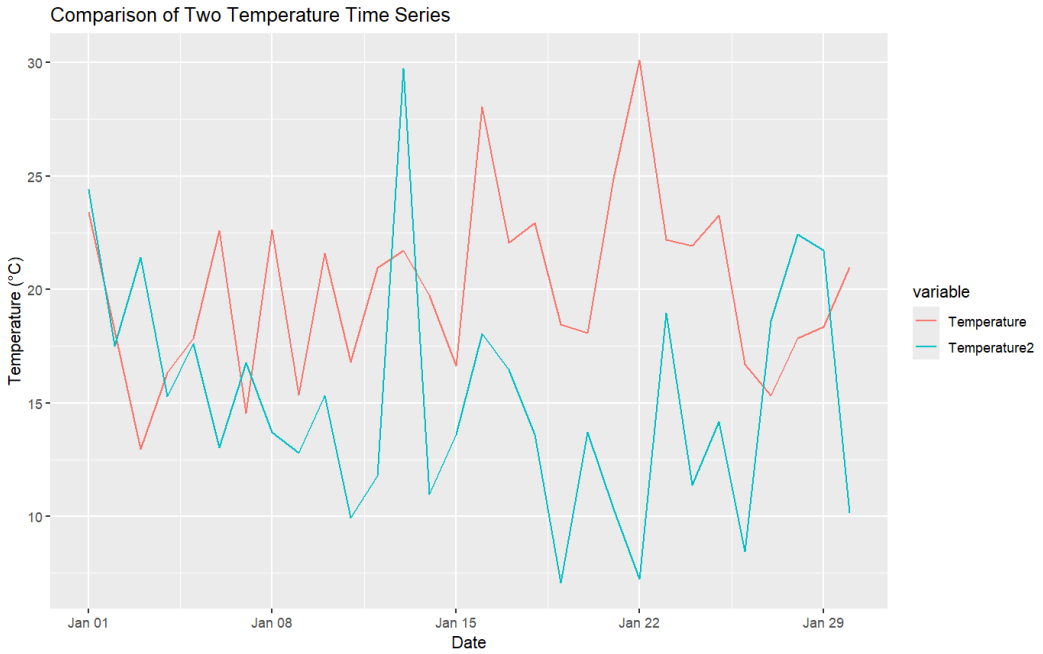


Figure 4.8: A comparison of two temperature time series

Let's cover plot types next:

- **Area plots**: Area plots are useful for displaying cumulative data points over time. They can be particularly effective for showing volume or quantity changes where the accumulation of values is significant:

```
ggplot(data = temp_data, aes(x = Date, y = Temperature)) +
 geom_area(fill = "lightblue") +
 labs(title = "Cumulative Daily Temperature",
 x = "Date",
 y = "Cumulative Temperature (°C)")
```

  The following plot illustrates the cumulative daily temperature using an area plot, which helps to visualize the overall trends and the magnitude of temperature changes over time:

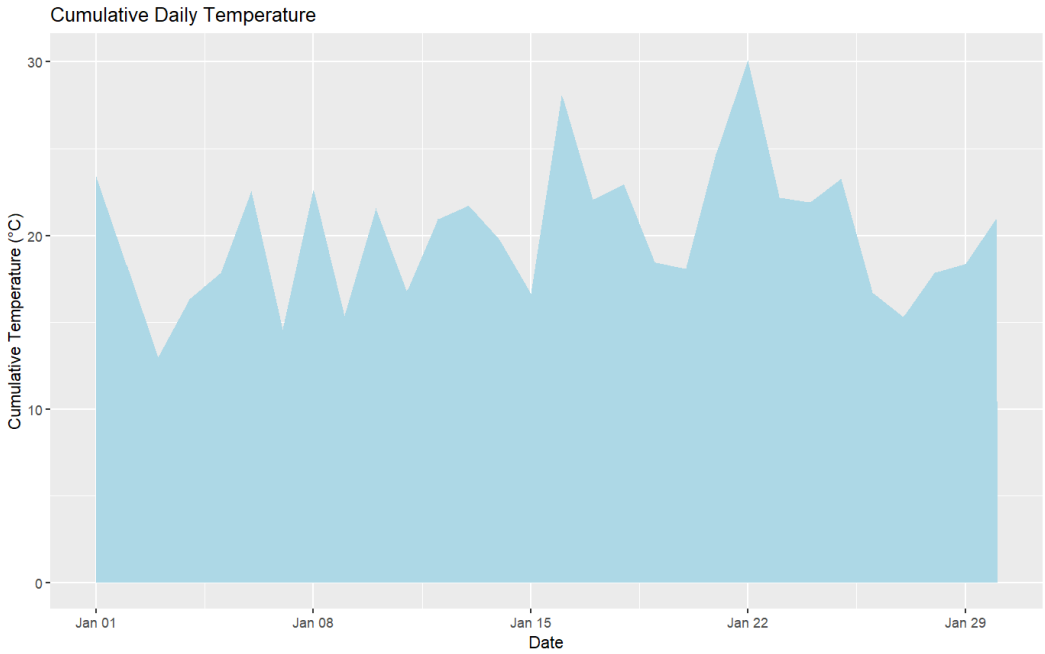


Figure 4.9: Cumulative daily temperature represented by an area plot

- **Step plots**: Step plots are useful for time series data where changes occur at discrete intervals, making them ideal for digital signals or inventory level data:

```
ggplot(data = temp_data, aes(x = Date, y = Temperature)) +
 geom_step() +
 labs(title = "Step Plot of Temperature Changes",
 x = "Date",
 y = "Temperature (°C)")
```

The following step plot visualizes the temperature changes over time, emphasizing the discrete intervals at which the temperature values shift:

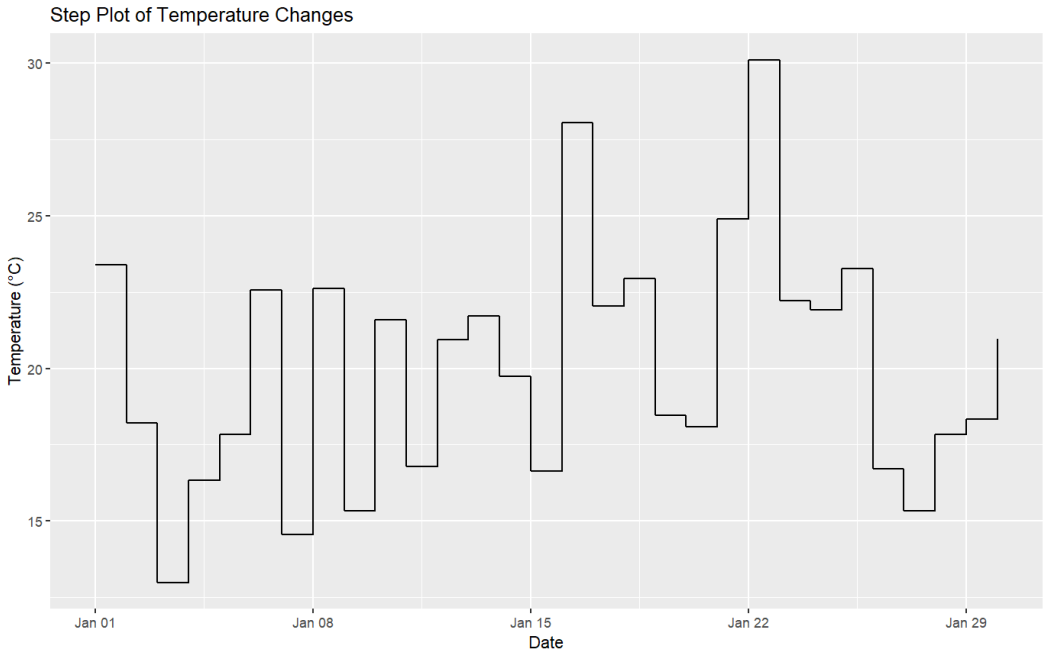


Figure 4.10: Step plot showing the discrete changes in temperature over time

- **Stacked charts**: Stacked charts are great for comparing multiple categories of data stacked over time, showing the part-to-whole relationships and trends in composition. We use `geom_area(position = "stack")` to stack charts:

```
temp_data$Temperature3 = rnorm(30, mean = 10, sd = 5)

temp_data_melted <- melt(temp_data, id.vars = "Date")

ggplot(data = temp_data_melted, aes(x = Date, y = value, fill = variable))
+
 geom_area(position = "stack") +
 labs(title = "Stacked Area Chart of Temperature Data",
 x = "Date",
 y = "Temperature (°C)")
```

The following stacked area chart shows the cumulative temperature data from three different datasets. This visualization highlights how different variables contribute to the total temperature distribution over time:

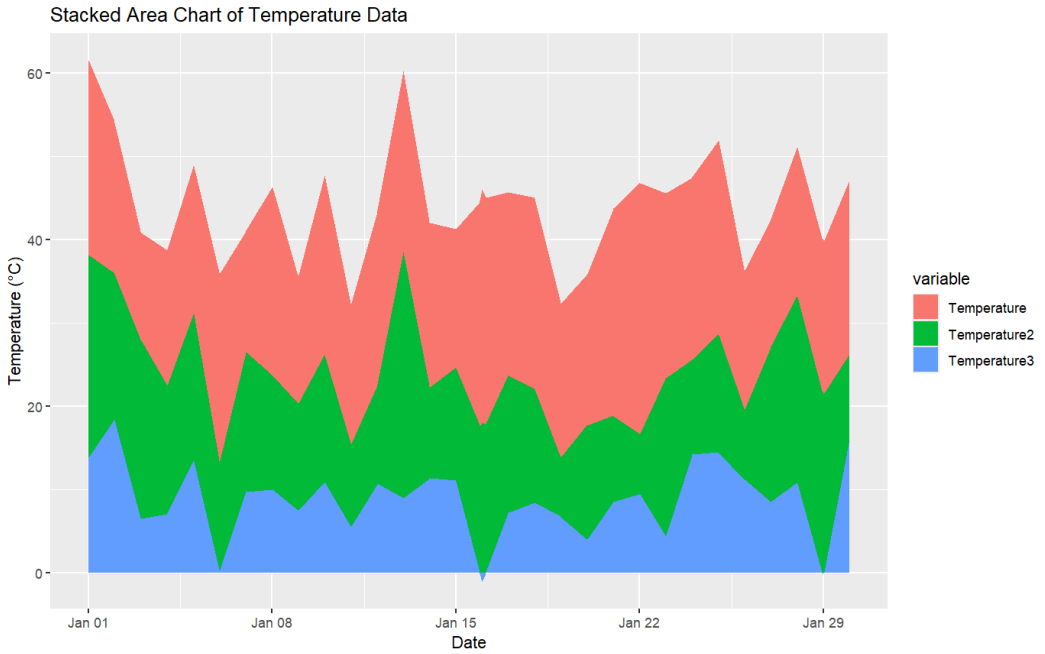


Figure 4.11: Stacked area chart comparing three temperature datasets over time

These examples illustrate how to use `ggplot2` to create advanced time series plots, from simple line charts to more complex visualizations like multi-line comparisons and stacked area charts. These tools allow you to effectively convey trends, comparisons, and distributions in time series data.

Customizing plots with ggplot2

Themes in `ggplot2` are powerful tools that help you control the overall aesthetic of your plots. The `theme()` function in `ggplot2` allows extensive customization, from tweaking text elements to adjusting plot background styles.

Now that we've introduced the concept of themes in `ggplot2`, let's explore how to apply a predefined theme. Using built-in themes can significantly enhance the visual appeal of your plots, making them more readable and professional. For example, the `theme_minimal()` function provides a clean and minimalistic style.

Applying a predefined theme

ggplot2 includes several built-in themes that can be easily applied to any plot. For instance, `theme_minimal()` provides a clean, minimalistic look to your plot:

```
ggplot(temp_data, aes(x = Date, y = Temperature)) +
  geom_line() +
  theme_minimal()  # Applies a minimal theme
```

The following plot illustrates how the `theme_minimal()` changes the plot's appearance, removing the background gridlines and delivering a sleek, minimalist aesthetic:

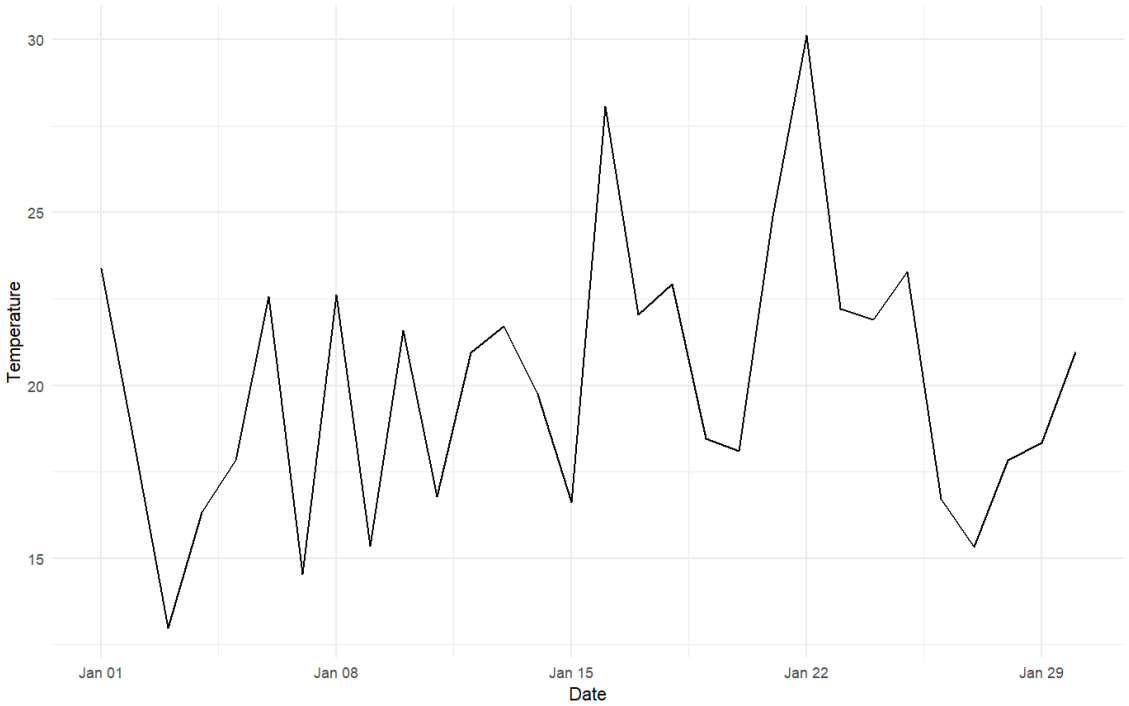


Figure 4.12: A line plot of daily temperatures with the minimal theme applied

You can modify specific components of the theme such as text elements and axis text. This customization is done by adding `theme()` elements to your plot code:

```
ggplot(temp_data, aes(x = Date, y = Temperature)) +
    geom_line() +
    theme_minimal() +
    theme(title = element_text(face = "bold", color = "#333333"),  # Bold title
```

```
with specific color
          axis.title.x = element_text(size = 12, color = "blue"),  # Blue color
and size for x-axis title
          axis.title.y = element_text(size = 12, color = "blue"),  # Blue color
and size for y-axis title
          legend.title = element_text(face = "italic"))            # Italicized
legend title
```

The following plot demonstrates how the axis labels are customized with bold and colored text, enhancing the readability and visual appeal of the plot:

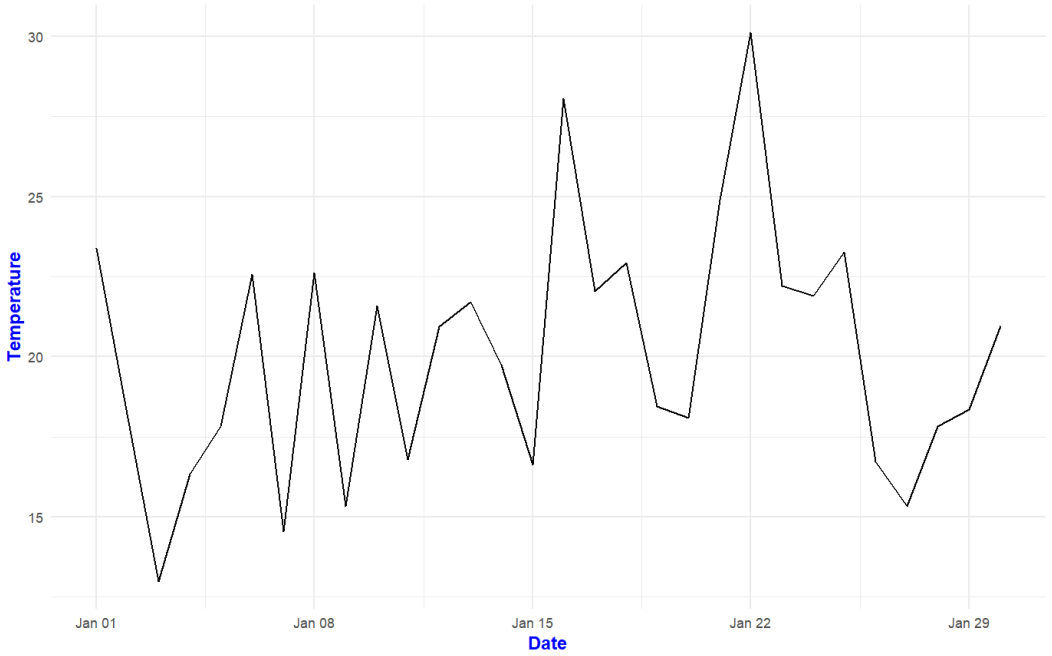


Figure 4.13: Line plot with customized axis labels, using bold and blue text

Text elements like titles, axis labels, and legends can significantly enhance the readability and effectiveness of a plot.

Customizing titles and labels

Use the `labs()` function to add and customize titles and labels as follows:

```
ggplot(temp_data, aes(x = Date, y = Temperature)) +
    geom_line() +
    labs(
```

```
        title = "Daily Temperature Overview",  # Main plot title
        subtitle = "January 2024",  # Subtitle
        x = "Date",  # X-axis label
        y = "Temperature (°C)",  # Y-axis label
        caption = "Source: Weather Data"  # Caption at the bottom
    )
```

The following plot includes a caption at the bottom to indicate the source of the data, making the visualization more informative and providing proper attribution:

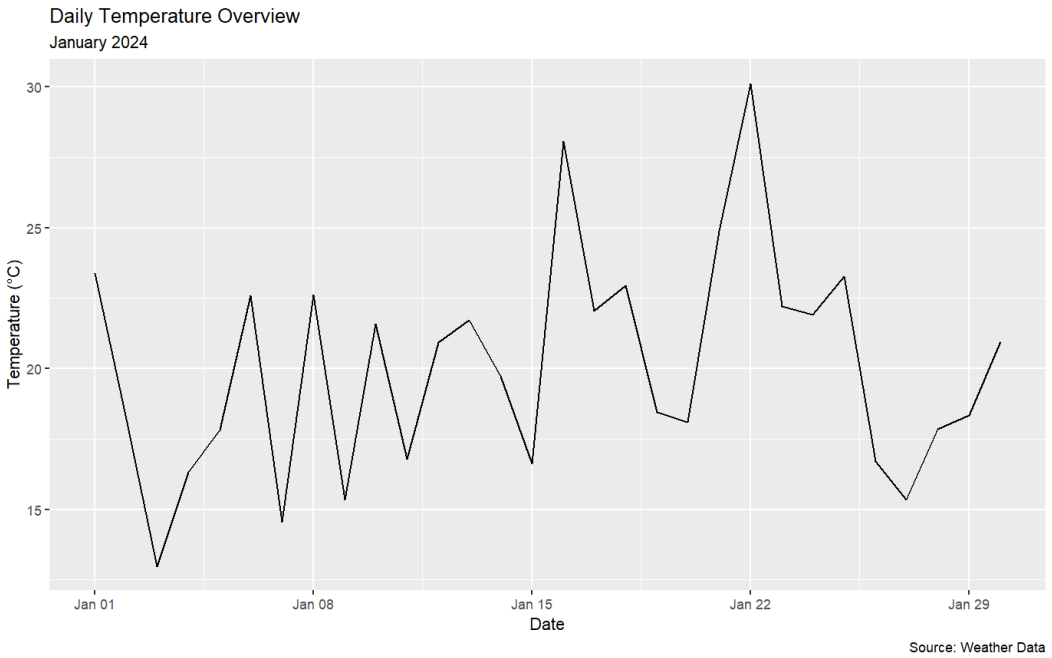


Figure 4.14: Line plot showing daily temperature data with a caption indicating the data source

Transforming axes scales

For data that spans several orders of magnitude, a logarithmic scale might be appropriate. Use the `scale_y_log10()` function to apply logarithmic transformation, as in the following code block:

```
ggplot(temp_data, aes(x = Date, y = Temperature)) +
    geom_line() +
    scale_y_log10()  # Logarithmic transformation for the y-axis
```

The next plot demonstrates how the y-axis is transformed using a logarithmic scale, making the temperature variations across different ranges more interpretable:

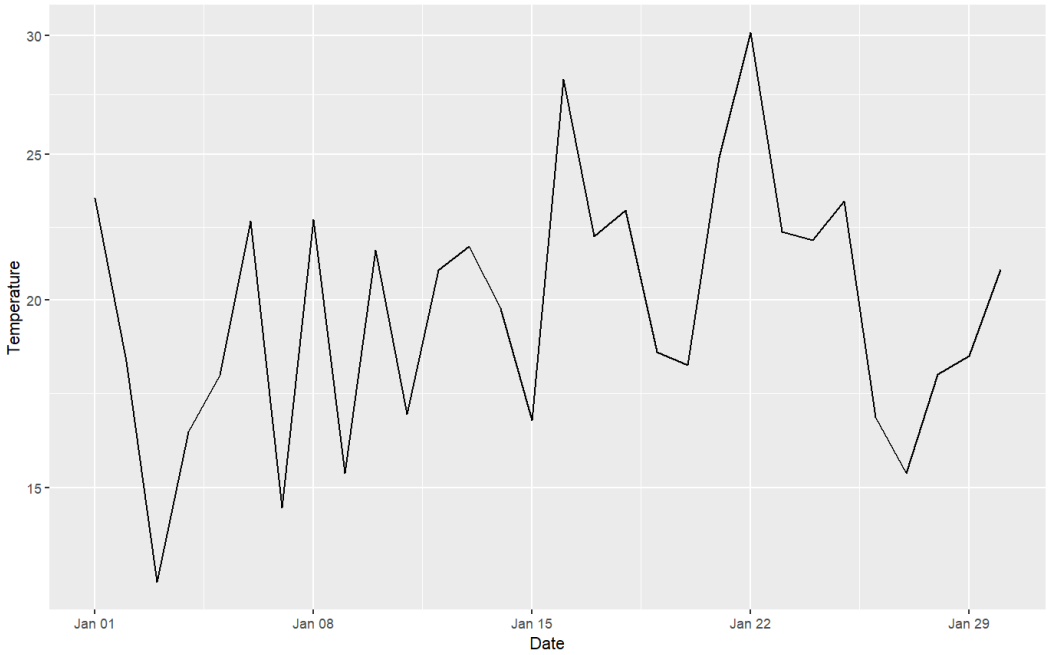


Figure 4.15: Line plot of temperature data with a logarithmic scale applied to the y-axis

Adjusting axis text and formatting date labels

Customizing the text and format of date labels can help make plots easier to interpret. The example in the following code tilts the labels by 45 degrees to improve readability:

```
ggplot(temp_data, aes(x = Date, y = Temperature)) +
    geom_line() +
    scale_x_date(date_labels = "%b %d", date_breaks = "1 week") +  # Display
labels in "Month Day" format
    theme(axis.text.x = element_text(angle = 45, hjust = 1))  # Tilt text at 45
degrees for readability
```

The following plot shows the date labels on the x-axis tilted at 45 degrees, making the dates easier to read, especially when data points are closely spaced:

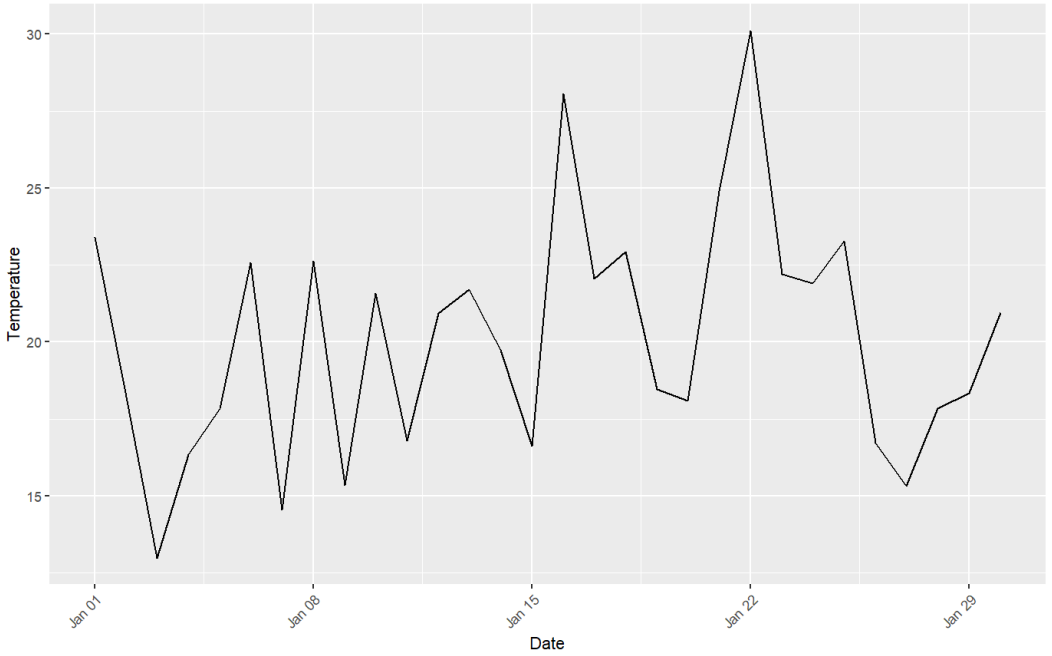


Figure 4.16 :Line plot of daily temperatures with x-axis labels tilted at 45 degrees

Adding annotations

To emphasize particular observations or events in your time series, you can add annotations directly to your plots. You can add a vertical line and text annotation to indicate an important event:

```
ggplot(temp_data, aes(x = Date, y = Temperature)) +
    geom_line() +
    geom_vline(xintercept = as.Date("2024-01-15"), color = "red", linetype =
"dashed") +  # Vertical line at a specific date
    annotate("text", x = as.Date("2024-01-15"), y = 25, label = "Important
Event", color = "red")  # Text annotation
```

The following plot uses an annotation to highlight an important event that occurred on January 15, 2024. The event is marked by red text and a dashed vertical line to draw attention to the specific date:

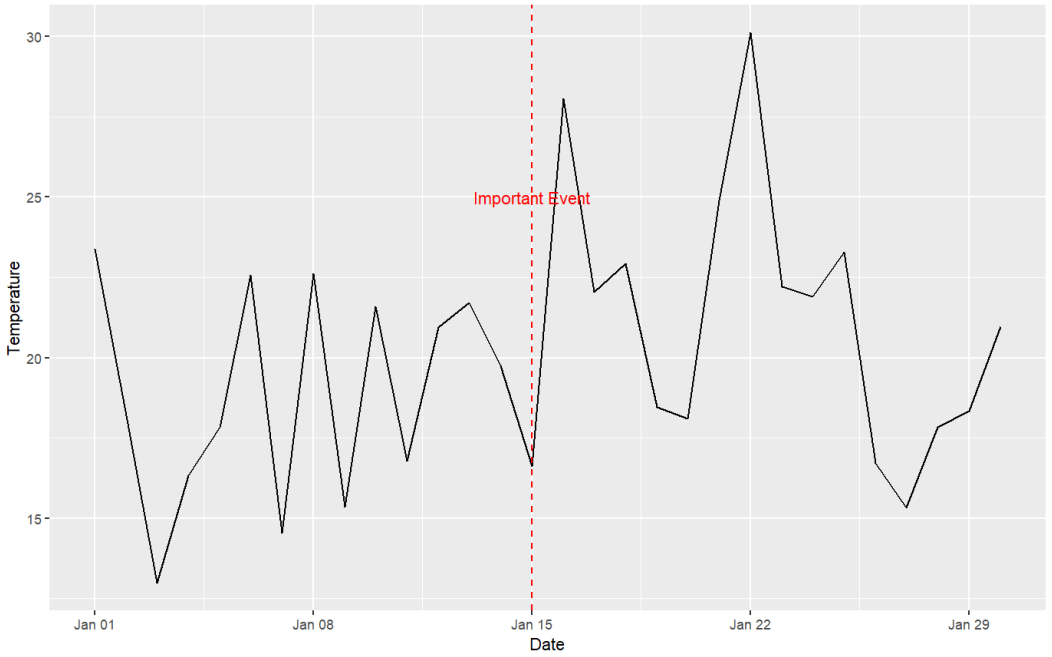


Figure 4.17: Line plot with an annotation highlighting an Important Event on January 15, 2024

Using color, shapes, and lines

Different colors, shapes, and line types can be used to draw attention to key periods or points in your data. The following code shows how you can highlight points where temperature exceeds a threshold (25 °C) using red:

```
 ggplot(temp_data, aes(x = Date, y = Temperature)) +
    geom_line() +
    geom_point(data = subset(temp_data, Temperature > 25), color = 'red')  # Red
points for temperatures > 25°C
```

The following plot displays red points for any temperature values that exceed 25°C, making it easier to identify extreme values in the dataset:

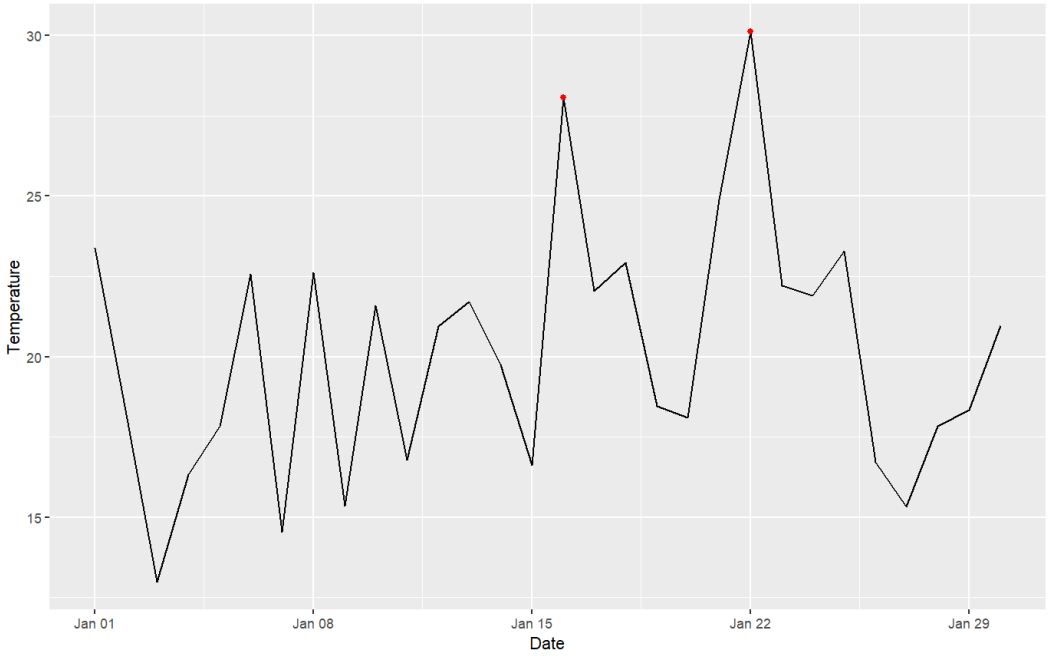


Figure 4.18: Line plot of daily temperatures with red points indicating temperatures above 25°C

Combining ggplot2 with plotly for interactive visualizations

Convert `ggplot2` plots into interactive Plotly graphs using the `ggplotly()` function from the `plotly` library:

```
# Check if the plotly package is already installed
if (!requireNamespace("plotly", quietly = TRUE)) {
  # If not installed, install it
  install.packages("plotly")
}
#load plotly package
library(plotly)

p <- ggplot(temp_data, aes(x = Date, y = Temperature)) + geom_line()
# Creating a basic ggplot2 plot
ggplotly(p)  # Convert ggplot2 plot to a plotly object
```

The following plot is an interactive version of the temperature graph. By hovering over any point, the user can view the exact date and temperature value at that point:

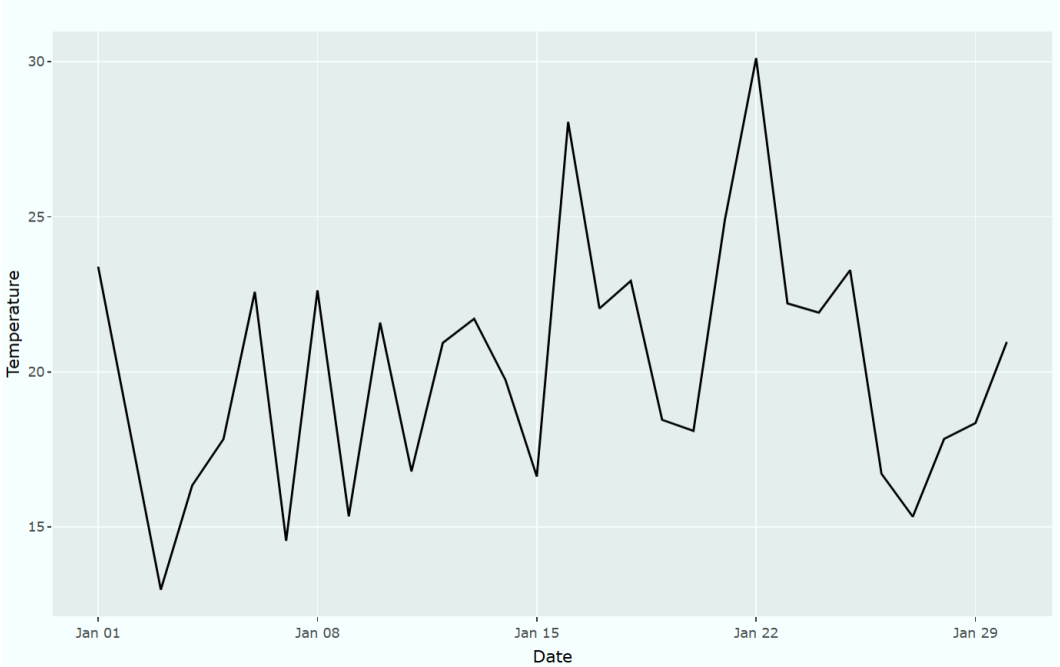


Figure 4.19: Interactive plotly graph generated from a ggplot2 plot

Hovering over each data point will show the x and y values. Make sure you take note of these compatibility tips:

- Ensure all `ggplot2` layers are supported by `plotly`, as some custom `ggplot2` functions might not convert perfectly
- Test interactive plots thoroughly to make sure interactivity does not obscure data interpretation

By mastering these customization techniques in `ggplot2`, you can elevate the clarity, impact, and professionalism of your time series visualizations, making your data not only more accessible but also more compelling to your audience.

Next, you will see how to plot interactive visualizations with the R shiny package.

Interactive visualizations with shiny and plotly

This section explores the use of `shiny` and `plotly` in R to create interactive visualizations. You will cover the basics of interactive visualization and get some guidance on utilizing these tools to make dynamic and user-interactive plots.

Interactive visualizations involve the viewer by allowing them to manipulate what they see. This interaction not only maintains interest but also encourages deeper exploration of the data. By interacting with data, users can uncover relationships and trends that might not be visible in static images. Interactivity can lead to a better grasp of the data's scope, scale, and nuances.

Interactive tools enable users to adjust parameters, filter views, and drill down into specifics. Such actions can reveal hidden patterns, outliers, or anomalies that static graphs might obscure. Users can personalize their data exploration journey, focusing on aspects most relevant to their questions or interests. This tailored approach can lead to more effective and focused analyses.

Let's go through the tools present for interactive visualization.

Tools for interactive visualization

Various tools within the R ecosystem enable you to create interactive visualizations. Among these, `shiny` and `plotly` stand out due to their robust features and flexibility. Let's compare when to use `plotly` and when to use `shiny`.

`shiny` is ideal for building full-fledged web applications. These can range from simple dashboards to complex apps with multiple user inputs and outputs. `Plotly`, meanwhile, focuses more on the graphical aspects, enabling users to enhance plots with interactive features like hover effects, zooming, and panning capabilities.

`shiny` requires a steeper learning curve due to its vast capabilities and the need to manage both UI and server components. However, it offers unparalleled flexibility in app development. `Plotly` is generally easier to start with, especially for those already familiar with `ggplot2`, as it integrates seamlessly with `ggplot2`'s coding style.

When it comes to interactivity components, `shiny` allows for the creation of wide-ranging interactive components beyond visualizations, including data input by users, dynamic reporting, and other responsive feedback mechanisms.

On the other hand, `plotly` provides interactivity directly within the plots, making it better suited for presentations and reports where the focus is on visual data exploration.

Choosing between `shiny` and `plotly` often depends on the specific needs of the project—whether the priority is creating a standalone interactive application or enhancing visualizations within a broader analysis or presentation framework. Each tool offers unique advantages that can be leveraged to build powerful, engaging, and insightful interactive visualizations.

Building dynamic plots with shiny

Here's how to get started with `shiny`, from installation to understanding its core architectural components.

Getting started with shiny

To begin using `shiny`, you first need to install the package and then load it in your R session. Follow these steps to get set up:

1. Open your R console and use the following command to install `shiny` from CRAN:

```
# Check if the shiny package is already installed
if (!requireNamespace("shiny", quietly = TRUE)) {
  # If not installed, install it
  install.packages("shiny")
}
```

2. Once installed, load the package into your R environment to start using its functions:

```
library(shiny)
```

With `shiny` installed and loaded, you're ready to begin creating interactive web applications.

Introducing the architecture of a shiny application

`shiny` applications have a unique structure that separates the UI from the server-side processing logic, allowing for dynamic and responsive web applications.

UI

The UI defines how the application looks and is structured. It is created using R commands that specify the layout and appearance of the application. `shiny` uses familiar R syntax but introduces functions designed to create web-friendly interfaces. Here's a simple example of defining a UI in `shiny`:

```
ui <- fluidPage(
    titlePanel("Simple Shiny Application"),
    sidebarLayout(
        sidebarPanel(
            sliderInput("num", "Choose a number:", 1, 100, 50)
        ),
        mainPanel(
            textOutput("numberText")
```

```
        )
    )
)
```

The following figure lists the components of the `ui` object in a `shiny` application:

Name	Type	Value
ui	list [4] (S3: shiny.tag.list, list)	List of length 4
[[1]]	list [10] (S3: html_dependency)	List of length 10
[[2]]	NULL	Pairlist of length 0
[[3]]	NULL	Pairlist of length 0
[[4]]	list [1]	List of length 1

Figure 4.20: Breakdown of the ui object in a shiny app

We have covered the following in this example:

- `fluidPage` is used to create a fluid and responsive layout
- `titlePanel` adds a title to the page
- `sidebarLayout` organizes content into a sidebar and main panel, where controls and outputs can be placed respectively

We also need to discuss server logic.

Server logic

The server part of a `shiny` app handles the backend computations, reacts to user inputs, and sends output back to the UI. This separation of concerns makes `shiny` powerful for building interactive tools without needing to manually manage the web server details. Here's how you might define server logic corresponding to the above UI:

```
server <- function(input, output) {
 output$numberText <- renderText({

 paste("You selected:", input$num)
 })
```

In this server function we have:

- `input` and `output` are list-like objects provided by `shiny`. `input` contains values from the UI elements, and `output` is used to send data back to the UI.
- `renderText` is used to create a text output element. The actual text is dynamically generated based on the user's selection from the `sliderInput`.

With these components, a `shiny` app can dynamically update based on user interaction, making it ideal for creating interactive visualizations that need to adapt to changing data or user preferences. This architecture allows data scientists and statisticians to build powerful web applications tailored to specific datasets or analytical requirements, all within R.

Creating a basic shiny app

Let's walk through building a simple Shiny app that dynamically displays time series data. This example will demonstrate how to utilize sliders and other UI elements to control plot displays:

1. First, create a UI that includes a plot output area and a slider input to control the visualization:

```
ui <- fluidPage(
 titlePanel("Dynamic Time Series Plot"),
 sidebarLayout(
   sidebarPanel(
     sliderInput("slider", "Select Number of Points:",      min = 1, max =
100, value = 50)
   ),
   mainPanel(
     plotOutput("timePlot")
   )
 )
)
```

2. Then, set up server logic to react to user inputs and update the plot accordingly:

```
server <- function(input, output) {

 output$timePlot <- renderPlot({
 # Generate sample time series data
 time_data <- data.frame(
 Time = seq(as.Date("2020-01-01"), by = "days", length.out = input$slider),
 Value = cumsum(rnorm(input$slider))
```

```
  )

  # Plot the time series data
  plot(time_data$Time, time_data$Value, type = 'l',
  xlab = 'Time', ylab = 'Value', main = 'Dynamic Time Series')
  })
}
```

3. And, boom, we are done. Execute the app by calling the `shinyApp()` function:

```
shinyApp(ui = ui, server = server)
```

The slider in the following image allows the user to select the number of data points to be shown in the dynamic time series plot, ranging from 1 to 100.

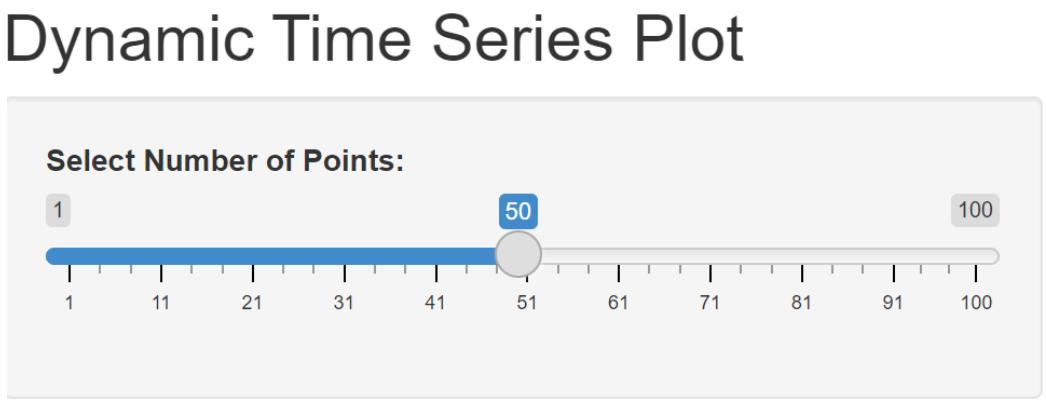


Figure 4.21: Shiny app interface with a slider

The following time series plot updates dynamically as the user interacts with the slider in the `shiny` app, providing a real-time view of the selected data points.

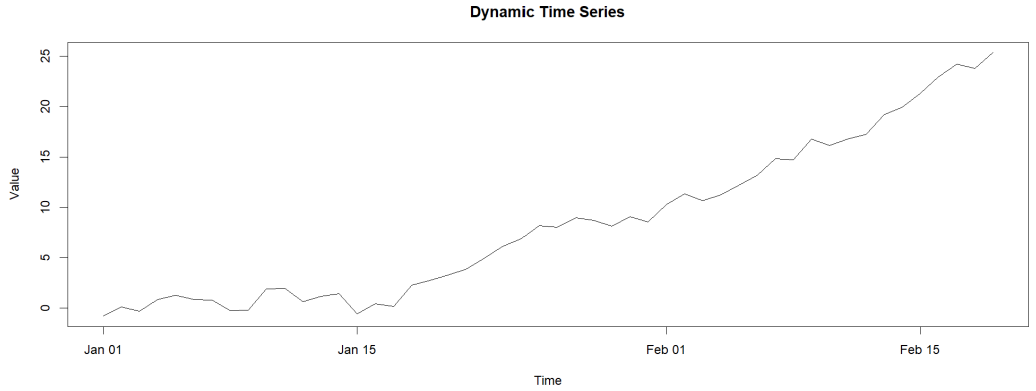


Figure 4.22: Dynamic time series plot generated by the Shiny app

This app displays a dynamic time series plot where the number of data points can be adjusted using a slider. The plot updates automatically as the slider is adjusted.

Advanced features in shiny

To further enhance `shiny` applications, consider incorporating more complex interactions using reactive expressions and observer functions. Integrating `shiny` with other R packages can also significantly expand its functionality. Let's quickly go through the three main options: reactive expressions, observer functions, and integration with data manipulation packages.

Reactive expressions

Reactive expressions are used for creating reactive objects that automatically update when the inputs change:

1. Create the `ui` object including a slider and a plot output:

```
ui <- fluidPage(
  # Slider input allows the user to select the number of days for time
series data
  sliderInput("slider", "Select number of days:",
              min = 1, max = 365, value = 30),

  # Output area for the time series plot
  plotOutput("timePlot")
)
```

2. Setup server logic:

```
server <- function(input, output) {

  # Reactive expression to generate time series data based on user input
  reactive_data <- reactive({
    # Create a data frame with two columns:

# Time: Sequence of dates starting from 2020-01-01, with a length based on
the slider input
    # Value: Cumulative sum of random numbers generated using rnorm()
    data.frame(
      Time = seq(as.Date("2020-01-01"), by = "days", length.out =
input$slider),
      Value = cumsum(rnorm(input$slider))  # Cumulative sum of random
normal values
    )
  })

  # Render a plot using the reactive time series data
  output$timePlot <- renderPlot({
    # Get the data from the reactive expression
    data <- reactive_data()

    # Create a line plot of Time vs Value
    plot(data$Time, data$Value, type = 'l', xlab = 'Time', ylab = 'Value')
  })
}
```

3. Run the `shiny` application. We will combine the UI and server to create the `shiny` app:

```
shinyApp(ui = ui, server = server)
```

The slider shown in the following figure allows the user to choose the number of days, ranging from 1 to 365, to be displayed in the dynamic time series plot.

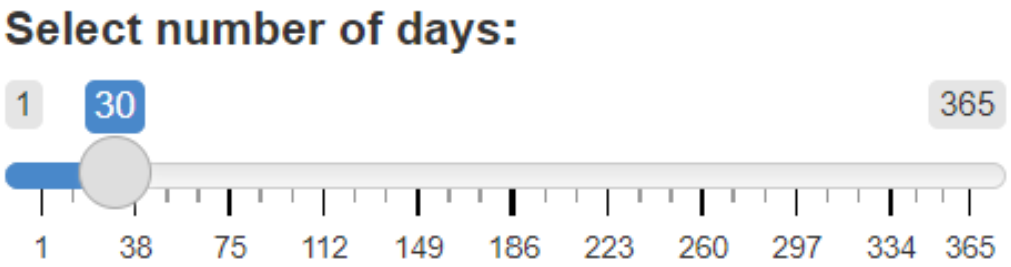


Figure 4.23: Shiny app interface with a slider that enables users to select the number of days for display in the time series plot

The following time series plot visualizes the data for the selected number of days, allowing users to observe trends over time as they interact with the app.

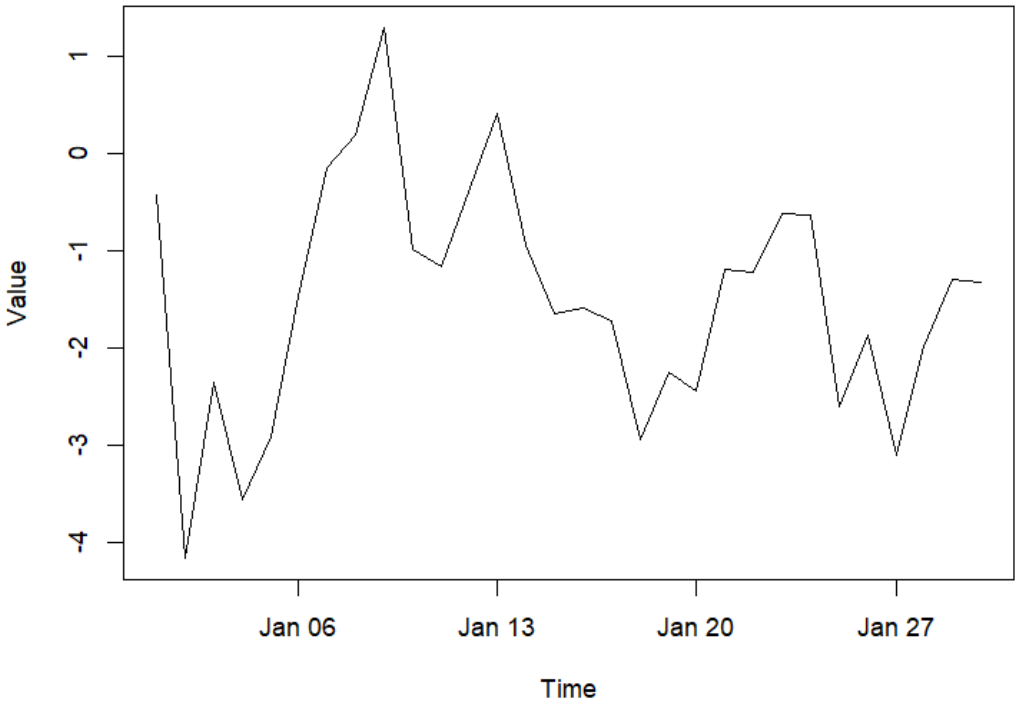


Figure 4.24: Dynamic time series plot generated by the Shiny app

Observer functions

Observers are useful for side effects such as writing to databases or other system operations in response to user input:

```
server <- function(input, output, session) {
 observe({
 # Log when slider value changes
 cat("Slider value is now", input$slider, "\n")
 })
}
Listening on http://127.0.0.1:4165
Slider value is now 30
```

These advanced features allow `shiny` apps to handle complex data processing and provide interactive, real-time visualizations, significantly enhancing the user's ability to engage with and understand the underlying data.

Enhancing interactivity with plotly

`plotly` for R is a powerful tool that enhances your plots with interactivity, making them more engaging and informative. Here's how to get started and make the most of `plotly`'s capabilities.

Overview of plotly syntax and functions

`plotly` in R uses a syntax that can be familiar if you are already accustomed to `ggplot2` or base R graphics. Here's a quick overview of essential functions:

- `plot_ly()`: The primary function to initiate a `plotly` graph. It is highly customizable and can take various inputs like data frames and plot attributes
- `add_trace()`: Adds layers to the plot, similar to layers in `ggplot2`, but each trace can represent parts of a graph such as lines, bars, markers, and so on
- `layout()`: Used to modify aspects of the plot layout, such as titles, axes properties, and annotations

Creating a time series plot with plotly

Let's use the above syntax to plot a time series plot for Google (ticker symbol GOOG). We will use the `quantmod` package introduced in *Chapter 2* to retrieve close price from January 1, 2023 to December 31, 2023:

1. Load the `quantmod` package:

```
library(quantmod)
```

2. Get the google prices from yahoo finance (from date between 2023-01-01 to 2023-12-31), then create a data frame:

```
getSymbols("GOOG", src = "yahoo", from = "2023-01-01", to = "2023-12-31",
auto.assign = TRUE)

df <- data.frame(GOOG$GOOG.Close)
head(df)
           GOOG.Close
2023-01-03      89.70
2023-01-04      88.71
2023-01-05      86.77
2023-01-06      88.16
2023-01-09      88.80
2023-01-10      89.24

df <- data.frame(rownames(df),df)
# append the column names accordingly
colnames(df) <- append('Date','Close')
head(df)
                Date Close
2023-01-03 2023-01-03 89.70
2023-01-04 2023-01-04 88.71
2023-01-05 2023-01-05 86.77
2023-01-06 2023-01-06 88.16
```

```
2023-01-09 2023-01-09 88.80
2023-01-10 2023-01-10 89.24
```

3. Create the `plotly` figure based on the dataframe and display it using `add_trace`:

```
goog_plotly <- plot_ly(df, type='scatter', mode='lines')%>%
add_trace(x=df$Date, y=df$Close, name ='GOOG')%>% layout(showlegend = TRUE)
goog_plotly
```

This code will display the following chart for Google's close price in the year 2023 using the `plotly` package.

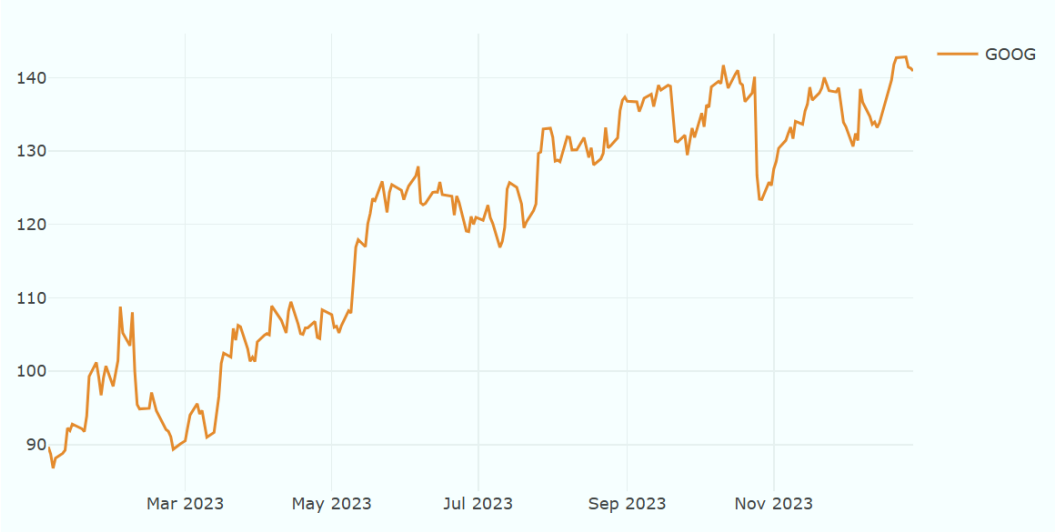


Figure 4.25: A time series plot displaying Google close price for 2023 using plotly

Creating interactive time series plots with plotly

To convert a `ggplot2` time series plot into an interactive `plotly` visualization, you can use the `ggplotly()` function. This function automatically translates your `ggplot2` syntax into a `plotly` object.

Here's how to create a basic `ggplot2` time series plot and convert it to an interactive `plotly` plot:

```
p <- ggplot(data.frame(x = 1:100, y = cumsum(rnorm(100))), aes(x, y)) +
geom_line()
```

```
ggplotly(p)
```

The time series plot in *Figure 4.26* highlights a dataset's general trend, showing significant fluctuations followed by a rapid upward spike. Such plots are commonly used to analyze temporal data.

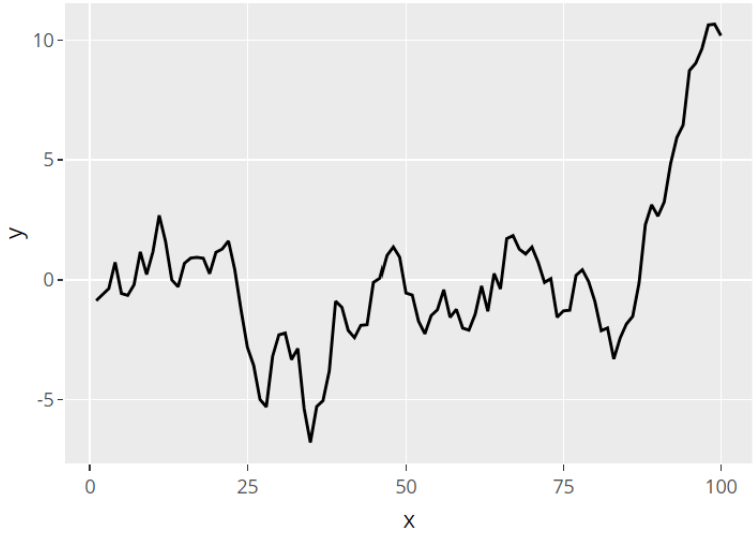


Figure 4.26: A time series plot displaying fluctuations and a rapid increase over time

Thanks to the previous examples, you should now know how to utilize `shiny` and `plotly` for advanced visualization techniques. Let's explore the common issues and concerns in visualizing time series.

Common pitfalls in visualizing time series

This section addresses some of the common errors that can occur when visualizing time series data and provides best practices to ensure accurate and effective visualizations.

Identifying common errors

When visualizing time series data, certain errors can significantly affect the accuracy and clarity of insights. Here are some common issues and suggestions to mitigate them.

Overplotting

Overplotting occurs when too many data points are plotted in close proximity, resulting in an unreadable graph. The data points may overlap, making it hard to identify patterns or outliers.

Examples of overplotting in time series include:

- Daily temperature readings over several years
- Stock prices for hundreds of companies displayed simultaneously

Techniques to mitigate overplotting include:

- **Transparency**: Adjust the opacity of data points so overlapping areas become more evident
- **Data aggregation**: Summarize the data over time intervals (for example, by week or month) to reduce the number of points displayed
- **Jittering**: Apply a small random offset to data points to spread them out visually and reveal clusters

Misleading scales

The use of inappropriate scales can distort data trends and lead to misinterpretation. Here are the most common scale issues:

- **Linear versus logarithmic scales**: A linear scale may understate growth trends in exponential data. Use a logarithmic scale when the data spans multiple orders of magnitude to more accurately represent growth patterns.
- **Inconsistent time intervals**: Using irregular intervals can create a misleading appearance of uniformity. Ensure time intervals are consistent throughout the plot to prevent visual misinterpretation.

Ignoring seasonality

Overlooking seasonal patterns in time series data can lead to incorrect interpretations of trends. Seasonal patterns (for example, quarterly sales cycles, annual temperature changes) may obscure long-term trends if not accounted for.

Failing to highlight anomalies

Anomalies are important in time series analysis as they can reveal significant, unexpected events or data quality issues. Failing to highlight these in visualizations can result in their oversight. Anomalies can indicate data integrity issues or reflect significant events that require special attention. Correctly identifying anomalies helps refine predictive models or respond effectively to unusual occurrences.

The following are strategies to highlight Anomalies:

- **Color coding**: Use distinct colors or symbols to identify outliers or significant events
- **Statistical detection**: Employ statistical methods to detect anomalies and label them directly on the graph

By understanding and addressing these common pitfalls, you can significantly improve the clarity and accuracy of your time series visualizations, leading to more actionable insights.

Summary

In this chapter, we explored the essential techniques and tools for visualizing time series data in R, covering everything from basic plotting functions to advanced and interactive visualizations. We highlighted the power of R's built-in tools like `plot()` for creating simple time series plots and delved into `ggplot2` for more sophisticated, customizable graphics. We also examined how to enhance time series visualizations with interactive elements using `shiny` and `plotly`, bringing data to life through dynamic user interaction.

Key takeaways include understanding how to avoid common pitfalls such as overplotting, misleading scales, and ignoring seasonality, which can distort the true meaning behind the data. We also stressed the importance of adhering to best practices like using consistent time scales, appropriate colors, and clear annotations to make visualizations more accurate, readable, and accessible.

By following the techniques and best practices covered in this chapter, you'll be able to create compelling, insightful time series visualizations that effectively communicate the underlying trends and patterns in your data, ultimately driving better decisions and deeper analysis.

In the next chapter, we'll dive deep into the world of time series analysis. You'll learn how to identify trends, apply smoothing techniques, and break down data into its key components.

5

Exploratory Data Analysis for Time Series

Exploratory Data Analysis (**EDA**) for time series is an essential discipline in statistics and data science, where the main objective is to analyze and forecast time-dependent data effectively. This chapter serves as a gateway to understanding the dynamic behaviors exhibited by time series data across various fields, including economics, environmental science, and market analysis. You will explore topics like trend analysis, smoothing techniques, and anomaly detection.

This chapter combines theoretical knowledge with practical examples using R. The practical component includes step-by-step instructions and R code snippets that you can apply directly to your datasets, ensuring an interactive and application-focused learning experience. By the end of this chapter, you will be equipped to perform detailed exploratory analyses and apply these insights to real-world scenarios effectively.

We will cover the following topics in the chapter:

- Trend identification and analysis
- Understanding smoothing techniques
- Understanding seasonal decomposition
- Methods for decomposing time series
- Auto-correlation analysis
- Detecting anomalies

Technical requirements

R and RStudio are required for this chapter. Github scripts can be found in the repository.

You will need the following data file and packages for this chapter:

- `Kendall`: An R package used to compute Kendall's rank correlation and perform Mann-Kendall trend tests.
- `TTR`: The `TTR` (Technical Trading Rules) package in R is often used for technical analysis of financial market data.
- `stats`: The R `stats` package is a foundational packaged in R that provides a wide array of essential functions for statistical computing and data analysis.
- `Forecast`: The `forecast` package in R is a very popular set of tools designed for analyzing and forecasting univariate time series data. You will see some seasonal subseries plots using this package.
- `AnomalyDetection`: Originally developed by Twitter, is an open-source tool for detecting statistically significant anomalies in seasonal time series data.

Trend identification and analysis

In time series analysis, a trend represents a consistent, long-term upward or downward movement in the data over time, independent of any short-term fluctuations. Trends can be linear, showing a steady rate of increase or decrease, or non-linear, where the rate of change accelerates or decelerates.

Identifying trends in time series data is crucial across various fields:

- **Economics**: Trends can indicate general economic conditions such as inflation rates, unemployment levels, or stock market activities, guiding policy decisions and investment strategies
- **Environmental science**: In environmental monitoring, trend analysis helps in detecting gradual changes in climate patterns, such as global warming or shifts in rainfall patterns, which are vital for long-term ecological assessments and planning
- **Market analysis**: Understanding trends in market data enables businesses to forecast future conditions, plan inventory levels, adjust pricing strategies, and more effectively manage supply chain operations

Recognizing and understanding trends not only aids in making informed decisions based on historical data but also in predicting future occurrences, allowing stakeholders to prepare and respond proactively.

Methods for detecting trends

In time series analysis, identifying and analyzing trends is crucial for forecasting and understanding underlying patterns. In this section, we will explore methods for trend detection and interpretation, emphasizing statistical tests and regression analysis, along with actionable examples in R.

Visual inspection

Visual inspection of time series data is a foundational method for identifying trends. This approach involves plotting the data to visually assess its behavior over time. In this section, we'll explore how to use line graphs to detect trends and interpret various types of trends such as upward, downward, and stable.

Using line graphs to visually assess trends

Line graphs are particularly useful for visualizing time series data because they clearly show how data points are connected over time intervals:

1. Let's start by importing the data. Ensure you have the correct date format:

```
df = read.csv("choco_monthly_revenue.csv")
df$Month.with.Year <- as.Date(df$Month.with.Year)
```

2. Load the `ggplot2` package:

```
library(ggplot2)
```

3. Create a line graph in R:

```
ggplot(df, aes(x = Month.with.Year, y = revenue)) +
  geom_line(color = "blue", linewidth= 1) + # Line color and thickness
  labs(title = "Time Series Trend Analysis",
       x = "Date",
       y = "Value",
       caption = "Data simulated for trend analysis.") +
  theme_minimal() # Apply a minimalistic theme for clarity
```

This script generates a line graph using the `ggplot2` package. The `geom_line` function connects these points in a line, providing a visual display of how the values change over time.

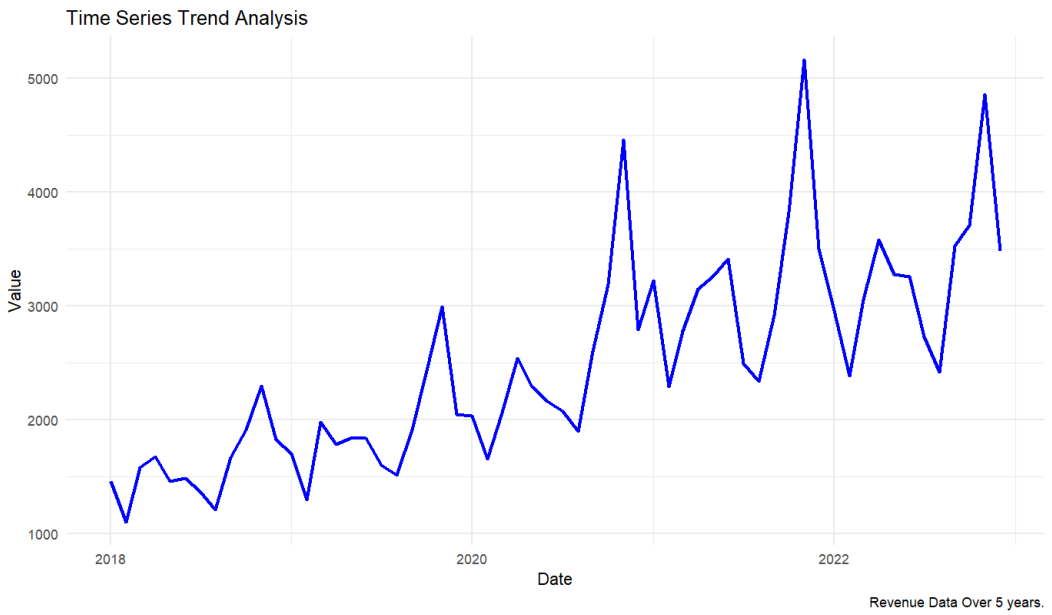


Figure 5.1: Time series trend analysis of revenue data over five years

The graph shows a time series trend analysis of revenue data from 2018 to 2023, with noticeable seasonal fluctuations and an overall upward trend. The revenue peaks sharply around 2021 and 2023, followed by slight declines, but maintains a general growth trajectory over the 5-year period.

Interpreting trends in line graphs

The following is a quick, obvious assessment of trends:

- An upward trend in a time series graph shows a consistent increase in the values over time. Look for a line that generally slopes upwards from left to right.
- A downward trend is indicated by a consistent decrease in the values. The line will slope downwards from left to right.
- A stable trend, or no trend, is characterized by relatively constant values over time. The line will appear relatively flat or horizontal.

The more interesting part is how to transform the interpretation into code. Here's how you might enhance the graph to better interpret these trends:

```
ggplot(df, aes(x = Month.with.Year, y = revenue)) +
 geom_line(color = "blue", linewidth= 1) +
 geom_smooth(method = "lm", color = "red", se = FALSE) + # Add a linear trend line
```

```
  labs(title = "Analysis of Trend Types",
  subtitle = "Red line represents a linear fit to the data",
  x = "Date",
  y = "Value") +
  theme_minimal()
```

This produces the following output:

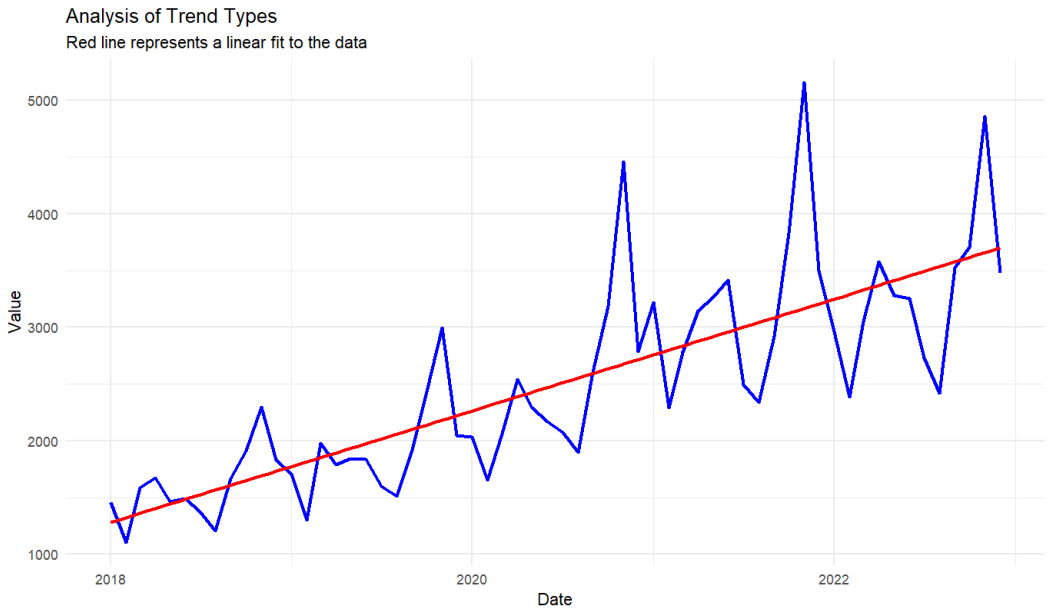


Figure 5.2: Time series plot with a red linear trend line representing the overall direction of the data

In this enhanced version, a red line represents a linear model fit to the data. The `geom_smooth()` adds smoothed conditional mean lines (trends) to plots; `method="lm"`, which is an explicit instruction to linear model for smoothing and a straight line should be fit to the data. This line helps clarify the trend direction by averaging fluctuations and emphasizing the overall movement.

By visually inspecting the line graph with these enhancements, you can quickly identify and interpret trends within your time series data. This method, while subjective, provides an immediate, intuitive understanding of data trends, serving as a precursor to more detailed statistical analysis.

Statistical tests

Statistical tests provide a more formal method of detecting trends in time series data. Among these, the **Mann-Kendall test** is widely used for its robustness in handling non-parametric data, which makes it suitable for datasets that don't necessarily adhere to normal distribution assumptions. This section will guide you through using the Mann-Kendall test in R and how to interpret its results.

Employing the Mann-Kendall test to detect trends

The Mann-Kendall test is a non-parametric test that assesses whether there is a monotonic upward or downward trend of the variable of interest over time. Here's how you can apply this test in R:

1. Install and load Kendall package:

```
# Check if the Kendall package is already installed
if (!requireNamespace("Kendall", quietly = TRUE)) {
# If not installed, install it
  install.packages("Kendall")
}
library(Kendall)
```

2. Run the Mann-Kendall test. We will assume that `time_series_data` is already defined; if not, load the `choco_monthly_revenue.csv` file:

```
mk_test <- MannKendall(df$revenue)
```

3. Print the test results:

```
print(mk_test)

tau = 0.669, 2-sided pvalue =< 2.22e-16
```

This script uses the `Kendall` package, which must be installed and loaded into your R session. The `MannKendall` function is then applied to the `Value` column of our `time_series_data` data frame.

Understanding the outputs and significance of the Mann-Kendall test

The output of the `MannKendall` function includes several components, but the key elements to focus on are:

- `tau_value`: This is the Kendall tau statistic, a measure of correlation that indicates the strength and direction of the association between two variables. A positive tau suggests an increasing trend, while a negative tau suggests a decreasing trend.
- `p-value`: The p-value (the probability of observing a trend as extreme as the one calculated, assuming the null hypothesis is true) tests the null hypothesis that there is no trend. A small p-value (typically ≤ 0.05) can be interpreted as strong evidence against the null hypothesis, thus suggesting a significant trend in the time series.

Here is an example of interpreting these results in your analysis:

1. Extract tau and p-value from Mann-Kendall test:

```
p_value <- mk_test$sl
tau_value <- mk_test$tau
```

2. Interpret and print the results extracted:

```
if (p_value < 0.05) {
  if (tau_value > 0) {
    cat("There is a statistically significant increasing trend in the data.
\n")
  } else {
    cat("There is a statistically significant decreasing trend in the data.
\n")
  }
} else {
  cat("No significant trend detected in the data.\n")
}
```

   There is a statistically significant increasing trend in the data.

This script uses conditional statements to interpret and print a message based on the results of the Mann-Kendall test. The `cat()` function is used to print the interpretation directly to the console.

By employing the Mann-Kendall test and understanding its outputs, you can provide robust statistical evidence of trends in your time series data, which is crucial for making informed decisions in various applications such as environmental science, economics, and market

analysis. This method not only detects trends but also quantifies the significance of these trends, offering a deeper insight than visual inspection alone.

Next, we will introduce some smoothing techniques, which are widely used in time series analysis.

Understanding smoothing techniques

Smoothing techniques are essential in time series analysis, serving to refine the observational data by reducing noise and randomness. This process is vital for revealing clearer patterns and trends that might be obscured in raw data. By applying smoothing methods, analysts can better understand the underlying behaviors and structures within the time series.

First, we need to understand the issues that come with the trend:

- **Noise reduction**: Time series data often contain a significant amount of variability or "noise" that can obscure real trends. Smoothing helps to mitigate this noise, making the true underlying patterns in the data more discernible.
- **Trend visualization**: Smoothing facilitates a clearer visualization of trends. It simplifies the data, allowing analysts to observe long-term tendencies without getting distracted by short-term fluctuations.

There are several benefits of reducing noise to clarify underlying trends, including:

- **Enhanced forecast accuracy**: Smoother data typically leads to more accurate predictions because the models are less likely to overfit meaningless fluctuations and more likely to capture significant trends
- **Improved decision-making**: By providing a cleaner overview of the data, smoothing techniques enable decision-makers to formulate strategies based on the significant movements observed in the time series, rather than reacting to random variations
- **Efficient data analysis**: Smoothing can make complex data sets more manageable and easier to analyze by reducing the amount of detail to a level that retains essential information while discarding redundant noise

There are various smoothing methods, each suited to different types of data and analytical needs. We will explore them in more detail shortly.

Moving averages

Moving averages are crucial tools in time series analysis, used to smooth out data to better identify underlying trends. They work by averaging the data points over a specified number of periods, which helps to reduce the noise and reveal clearer trends. This section will cover three

primary types of moving averages: Simple, Exponential, and Weighted, each with unique characteristics and applications.

Simple moving average

The **Simple Moving Average** (**SMA**) calculates the average of a selected range of values, in a stock market example we typically choose closing prices, by the number of periods in that range. It gives equal weight to all values, which smooths out the data but can also lag significant price movements.

Here is how you can calculate and plot a Simple Moving Average in R:

1. Define the filter coefficients. We will set a 6-point moving average with weights summed as 1:

```
# rep(1/6, 6) creates c(0.1666..., 0.1666..., 0.1666...,
0.1666...,0.1666..., 0.1666...,)
filter_coeff <- rep(1/6, 6)
print(filter_coeff)

[1] 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667
```

2. Apply the filter and calculate the SMA with a window size of 6. Using `sides = 1` means only previous and current values are used for calculation:

```
sma_results <- stats::filter(df$revenue, filter_coeff, sides = 1)
```

3. Plot the original data and the SMA:

```
plot(df$revenue, type = "l", col = "blue", lwd = 1, ylab = "Revenue", xlab
= "Time", main = "Simple Moving Average")
lines(sma_results, col = "red", lwd = 2)
legend("topright", legend = c("Original Data", "SMA"), col = c("blue",
"red"), lty = 1, lwd = 2)
```

This method is straightforward and uses only base R functions, making it easy to implement without additional libraries. Here's the plot output:

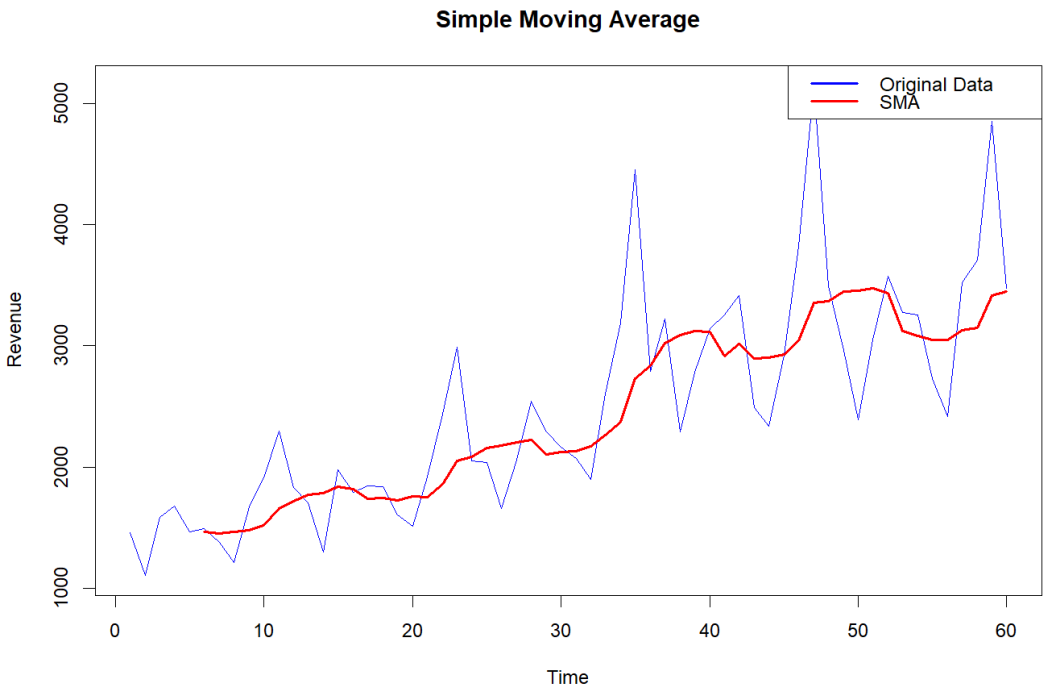


Figure 5.3: Time series plot with a simple moving average (red line) applied to smooth the data

The outcome is a line that removes the spikes, allowing for clearer trend detection.

Exponential moving average (EMA)

The **Exponential Moving Average** (**EMA**) gives more weight to recent data points, making it more sensitive to recent value changes compared to the SMA. This can be particularly useful for traders who need to react quickly to price changes:

1. Install and load the TTR package:

```
# Check if the TTR package is already installed
if (!requireNamespace("TTR", quietly = TRUE)) {
# If not installed, install it
  install.packages("TTR")
}
# Load the TTR package
library(TTR)
```

2. Calculate EMA for a window of 10 periods:

```
ema_results <- EMA(df$revenue, n = 10)
```

3. Plot the EMA and original data:

```
plot(df$revenue, type = "l", col = "blue", lwd = 1, ylab = "Value", xlab =
"Time", main = "Exponential Moving Average")
lines(ema_results, col = "green", lwd = 2)
legend("topright", legend = c("Original Data", "EMA"), col = c("blue",
"green"), lty = 1, lwd = 2)
```

In this script, the `EMA` function is used to calculate the exponential moving average for the same data set. The EMA is then plotted in green, providing a visual comparison against the original blue line of the data points. The preceding code generated a smoother curve compared to the simple moving average shown in the following figure:

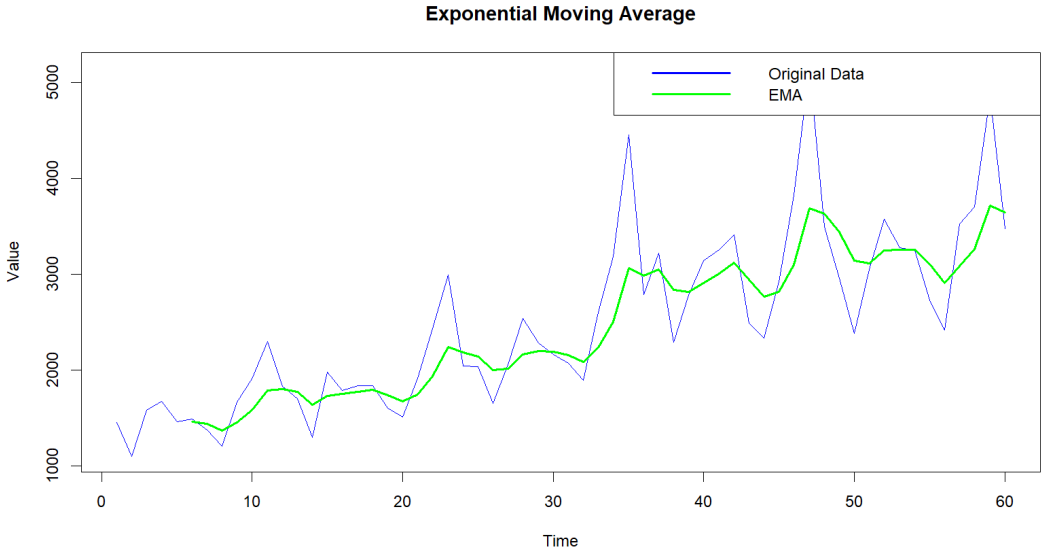


Figure 5.4: Time series plot with an exponential moving average (EMA) in green

Weighted moving average (WMA)

The **Weighted Moving Average** (**WMA**) assigns a heavier weighting to more recent data points. Unlike the EMA, the weights decrease linearly and can be customized, which offers flexibility in emphasizing the importance of recent data:

1. Calculate the WMA for a window of 10 periods:

```
wma_results <- WMA(df$revenue, n = 10)
```

2. Plot the original data and WMA:

```
plot(df$revenue, type = "l", col = "blue", lwd = 1, ylab = "Value", xlab =
"Time", main = "Weighted Moving Average")
lines(wma_results, col = "orange", lwd = 2)
legend("topleft",
       legend = c("Original Data",
                  "WMA"),
       col = c("blue", "orange"),
       lty = 1,
       lwd = 2)
```

The preceding code utilizes the default linear weights (`1:n`) and generates the following plot, which is another way of smoothing techniques compared to the `SMA` and `EMA`.

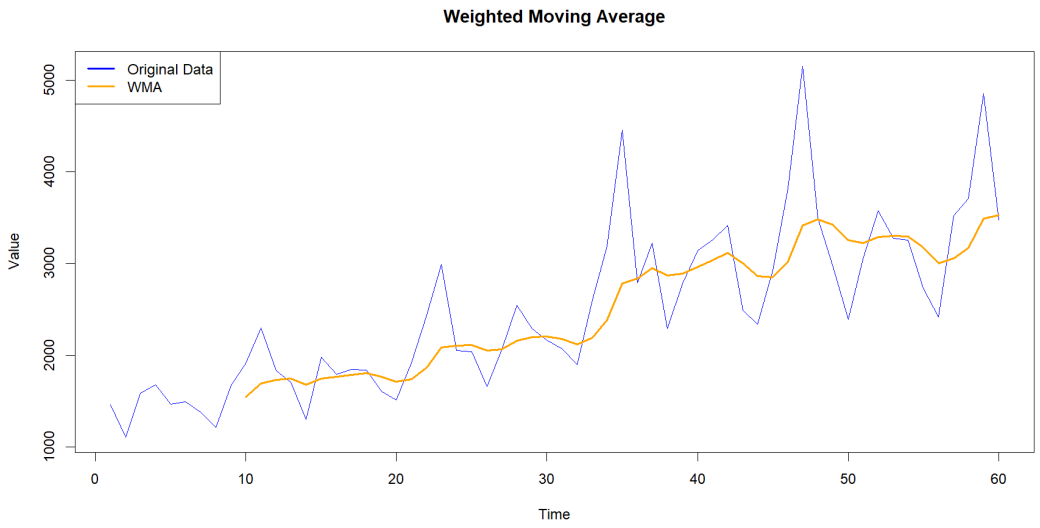


Figure 5.5: Time series plot with a weighted moving average (WMA) in orange

Let's move beyond the basics.

Advanced smoothing techniques

As we explore time series analysis, we encounter more sophisticated smoothing techniques that offer greater flexibility and accuracy in dealing with data irregularities.

One such method, **Locally Estimated Scatterplot Smoothing (LOESS)**, stands out for its ability to produce highly refined visual representations of complex data sets. This section introduces LOESS and explores how it compares with other smoothing techniques, focusing on their applications and effectiveness in different scenarios.

LOESS is a non-parametric (which is a technique that makes minimal or no assumptions about the underlying data probability distribution) method that uses local polynomial regression (in LOESS, this means that within a specific moving window of data points, a low-degree polynomial which often is just linear, is used to fit the local relationship between the variables) to smooth out data. Unlike moving averages or exponential smoothing which apply a global model to the data, LOESS fits multiple simple models in a localized fashion. This approach makes LOESS particularly powerful for modeling data with non-linear trends or varying variance within the data set.

Applying LOESS in R

R provides built-in support for LOESS, making it accessible for anyone involved in data analysis. Here's how to apply LOESS smoothing to a time series data set:

1. Convert the date to numeric and assign a new column:

```
df$Month.with.Year.numeric <- as.numeric(df$Month.with.Year)
head(df)
```

```
   Month.with.Year revenue Month.with.Year.numeric
1      2018-01-01    1458                   17532
2      2018-02-01    1102                   17563
3      2018-03-01    1584                   17591
4      2018-04-01    1676                   17622
5      2018-05-01    1461                   17652
6      2018-06-01    1491                   17683
```

2. Apply the LOESS smoothing:

```
loess_smoothed <- loess(revenue ~ Month.with.Year.numeric,
                        data = df,
                        span = 0.3)
```

The preceding code snippets fit a LOESS regression model; the ~ (tilde) operator is often shown in a model code format, for example, `model(y ~ x, data = df)`, where y is the response variable and x is the predictor variable. Span controls the degree of smoothing; it determines the proportion of data points used for each local regression.

In R, the `predict()` function for a LOESS object is used to generate predictions for new data points based on the fitted model:

1. Predict the smoothed values:

```
smoothed_values <- predict(loess_smoothed)
```

After applying LOESS, visualizing the smoothed data alongside the original data can provide valuable insights into the effectiveness of the smoothing:

2. Plot the original data with LOESS values:

```
plot(df$Month.with.Year, df$revenue, type = "l", col = "blue", lwd = 1,
     ylab = "Revenue", xlab = "Date", main = "LOESS Smoothing")
lines(df$Month.with.Year, smoothed_values, col = "red", lwd = 2)
legend("topright", legend = c("Original Data", "LOESS"), col = c("blue",
"red"), lty = 1, lwd = 2)
```

This is the output:

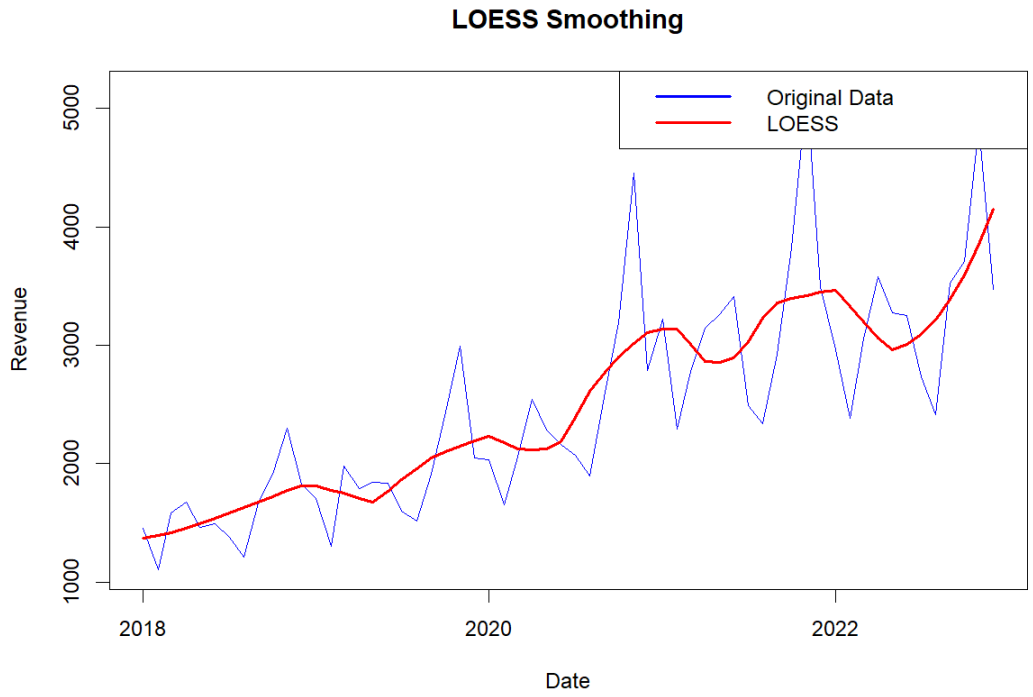


Figure 5.6: Time series plot with LOESS smoothed data in red

This plot shows the original data in blue and the LOESS smoothed data in red, illustrating how LOESS adapts to local changes in the data's trend and volatility without assuming a global form.

Comparing smoothing techniques

While LOESS offers remarkable flexibility and local adaptation, it is not always the best choice for every type of data. Comparing LOESS with other smoothing techniques like SMA and EMA can help determine the most appropriate method based on specific requirements:

- **Responsiveness versus smoothness**: LOESS can be adjusted to be more or less responsive to changes in data through its span parameter. A smaller span makes the model more sensitive to local fluctuations, while a larger span produces smoother results.

- **Handling non-linear trends**: LOESS excels in scenarios where the data exhibits non-linear trends that simpler methods like SMA or EMA might oversimplify or fail to capture accurately.
- **Computational intensity**: LOESS is computationally more intensive than SMA or EMA, which may be a consideration for very large datasets or real-time analysis needs.

By understanding the strengths and limitations of advanced smoothing techniques like LOESS, you can better tailor your approach to the specific characteristics of your data, ensuring more accurate and actionable insights.

Analyzing smoothed data

Once time series data has been smoothed, the next critical step is to analyze and interpret the results effectively. This section explores techniques for extracting meaningful insights from smoothed data and focuses on how to identify significant trends that can guide decision-making.

Techniques for analyzing smoothed time series

Smoothed data reduces noise and clarifies the underlying patterns, making it easier to conduct a more accurate analysis. Here are some key techniques used to analyze smoothed time series data:

- **Trend identification**: With the noise reduced, identifying long-term trends becomes more straightforward. Analysts look for consistent upward or downward movements in the smoothed data as indicators of underlying trends.
- **Seasonal adjustments**: In many time series, underlying trends can be obscured by seasonal variations. Once the data is smoothed, it's essential to adjust for seasonality to reveal the true trend.
- **Change point detection**: Smoothed data can help in detecting change points—times when the statistical properties of the series change significantly. This can be crucial for diagnosing shifts in the process generating the data.
- **Visual inspection**: Although quantitative analysis is powerful, visual inspection of smoothed data remains a valuable tool. Plots of smoothed data can reveal dynamics that are not apparent from raw data alone.

Steps to analyze smoothed data in R

Here's how you might implement these techniques in R to analyze smoothed data, particularly focusing on identifying significant trends:

The resulting plot highlights the **smoothed time series**, showing a clear upward trend over time, with peaks and troughs becoming more visible as short-term fluctuations are smoothed out:

1. Create a data frame from the previous example:

```
smoothed_data <- data.frame(
  Date = df$Month.with.Year,
  Smoothed_Price = predict(loess_smoothed)
)
```

2. Plot the smoothed curve:

```
ggplot(smoothed_data, aes(x = Date, y = Smoothed_Price)) +
  geom_line(color = "blue") +
  labs(title = "Smoothed Time Series Analysis",
       x = "Date",
       y = "Smoothed Price") +
  theme_minimal()
```

The **Smoothed Time Series Analysis** graph in *Figure 5.7* visually represents the long-term trend of revenue data after applying LOESS smoothing:

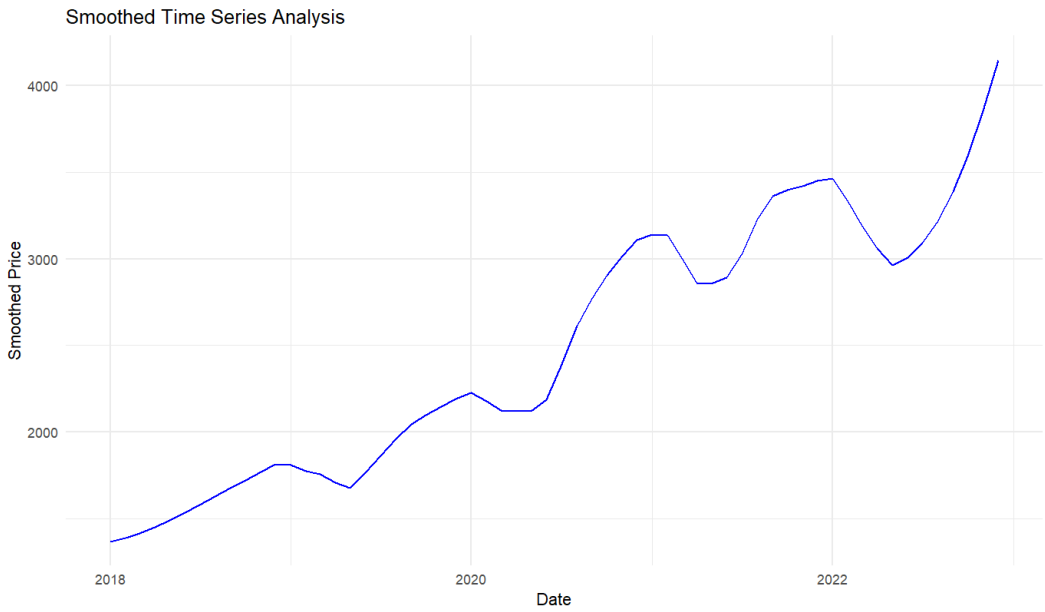


Figure 5.7: Smoothed time series curve

From *Figure 5.7*, we observe an upward trend in revenue, with a few key fluctuations around the years 2020 and 2022, suggesting periods of growth and dips.

3. Calculate the price changes compared to the previous period. Use the lag function to get the shifted price:

```
smoothed_data$Shifted_Price <- dplyr::lag(smoothed_data$Smoothed_Price, 1)
smoothed_data$Price_Change <- smoothed_data$Smoothed_Price -
smoothed_data$Shifted_Price
```

4. Remove the NA Value caused by lag shift and display the first few rows of data:

```
smoothed_data <- smoothed_data[!is.na(smoothed_data$Price_Change), ]

Head(smoothed_data)

       Date Smoothed_Price Shifted_Price Price_Change
2 2018-02-01       1390.306      1367.211     23.09497
3 2018-03-01       1417.499      1390.306     27.19325
4 2018-04-01       1451.361      1417.499     33.86152
5 2018-05-01       1491.463      1451.361     40.10238
```

```
6 2018-06-01       1535.252      1491.463     43.78853
7 2018-07-01       1580.752      1535.252     45.50073
```

5. Plot the rate of change:

```
ggplot(smoothed_data, aes(x = Date, y = Price_Change)) +
  geom_line(color = "red") +
  labs(title = "Rate of Change in Smoothed Price",
       x = "Date",
       y = "Rate of Change") +
  theme_minimal()
```

The preceding code snippet generates *Figure 5.9*, which shows the price changes after smoothing in red.

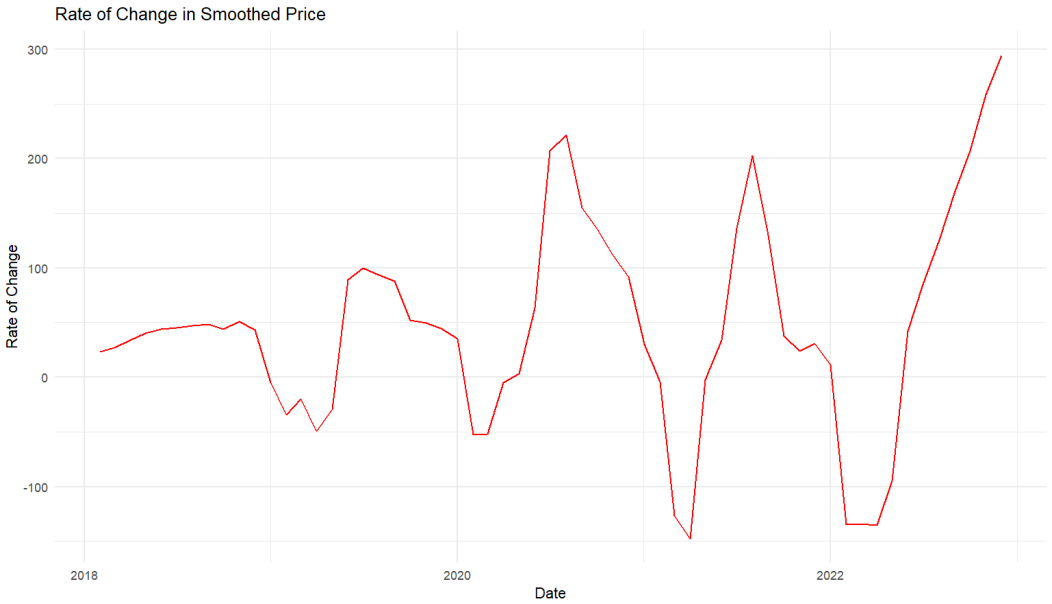


Figure 5.8: Time series on rate of change in smoothed price between 2018 and 2022

This script first plots the smoothed time series for visual inspection. It then calculates the rate of change between consecutive data points, providing insights into the dynamics of the trend, such as acceleration or deceleration.

Next, you will dive into more details on seasonal variations in time series.

Understanding seasonal decomposition

This section focuses on understanding the impact of seasonal variations in time series data, emphasizing the decomposition of a series into its fundamental components: trend, seasonality, and residuals. This analytical process is crucial for accurate forecasting and understanding underlying patterns influenced by seasonal changes.

Introducing seasonal decomposition

Seasonal decomposition plays a pivotal role in time series analysis, especially in fields where data exhibit regular and predictable patterns over time, such as economics, climatology, and retail analytics. This technique separates a time series into several distinct components, each representing underlying patterns that are often obscured by noise or other complex interactions in raw data. This process is crucial because it allows analysts to:

- **Identify underlying trends**: By isolating the trend component from seasonal fluctuations and irregularities, analysts can see the long-term direction of the data, be it rising, falling, or stable.
- **Understand seasonality**: Many time series exhibit seasonal patterns, which are critical for planning and decision-making. Recognizing these patterns helps businesses and organizations prepare for future conditions by adjusting operations, budgets, and strategies accordingly.
- **Assess cyclic and irregular influences**: Apart from the main trend and seasonal effects, decomposition also helps in identifying cyclical elements not tied to a fixed calendar schedule and irregular components, which include random or one-off variations.

Seasonal decomposition of time series typically breaks down into three main components:

- **Trend**: This component reflects the long-term progression of the data, showing how the central tendency moves over a significant period of time. It is the smoothed, consistently upward or downward path the series follows through years or decades.
- **Seasonality**: These are intra-year variations which typically occur at fixed periods and are associated with weather, social customs, and so on. Understanding seasonality is crucial for any business or activity that is impacted by seasonal factors.
- **Residuals**: The residuals of a time series decomposition are the components not explained by the trend or the seasonality. Studying residuals can help in detecting outliers, unexpected spikes or drops, and other anomalies that could suggest opportunities for further analysis or model improvements.

Now that you have a good idea of what seasonal decomposition is, let's look at how to implement it in R.

Implementing seasonal decomposition in R

Seasonal decomposition in R can be accomplished using several functions, with the most common being `decompose()` for classical decomposition and `stl()` for **Seasonal and Trend decomposition using LOESS (STL)** decomposition. Each function caters to different types of time series data and offers unique advantages depending on the nature and granularity of the dataset.

Using decompose() and stl() functions

The `decompose()` function is ideal for time series with a strong seasonal pattern and a stable seasonality that does not change over time. It requires the time series to be a complete cycle, meaning that you must have data for a full year if your data has an annual seasonality:

1. Load the stats package:

```
library(stats)
```

2. Creating a sample time series data with monthly observations:

```
time_series <- ts(df$revenue, frequency = 12)
```

3. Decompose the time series and plot it:

```
decomposed_data <- decompose(time_series)
plot(decomposed_data)
```

   The code snippet will plot the following diagram:

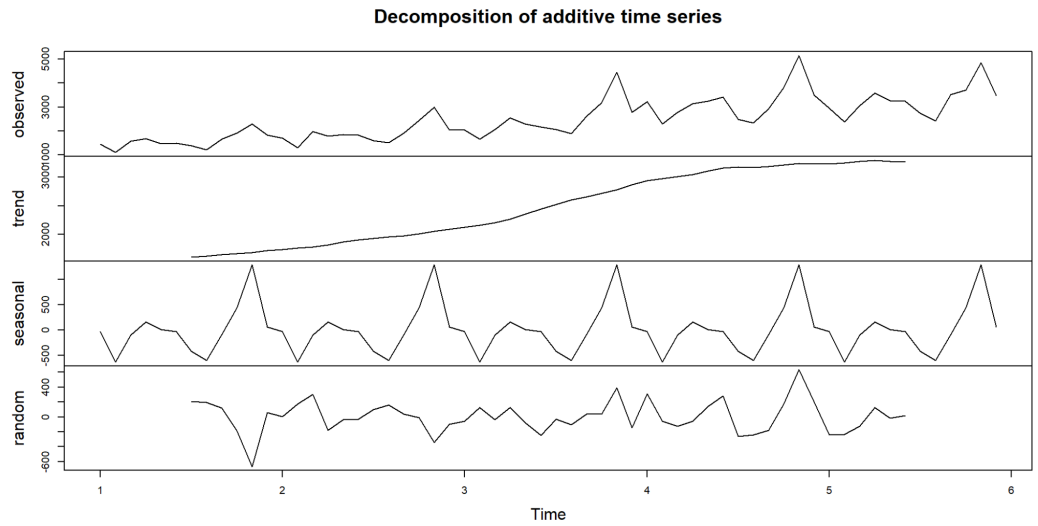


Figure 5.9: Decomposition plot of time series data showing the trend

This script generates a time series and applies the classical decomposition method to separate it into trend, seasonal, and other components. The observed curve contains raw data, the trend shows the underlying upward or downward slope; the seasonal represents the periodic signals, and the random is the residual fluctuations commonly referred as noise. STL decomposition is more flexible than classical decomposition as it allows handling of any type of seasonality, not necessarily a fixed one, and can be applied to both univariate and multiple time series.

4. Apply `stl()` to the previous time series data and plot it:

```
stl_data <- stl(time_series, s.window = "periodic")
plot(stl_data)
```

This script demonstrates the application of the STL method on the same time series dataset, showing how flexible the method can be with the `s.window` parameter set to "periodic," which controls the smoothness of the seasonal component.

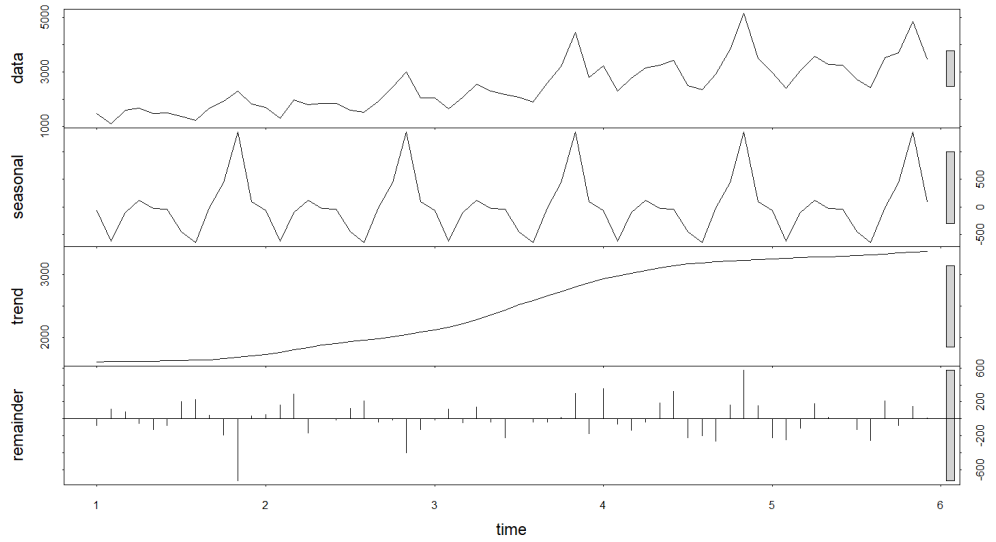


Figure 5.10: STL decomposition on a time series dataset

This decomposition plot breaks down the time series data into **seasonal**, **trend**, and **remainder** components. The first panel shows the raw data, followed by the seasonal component, which highlights recurring patterns. The trend component shows a steady upward trend over time, and the remainder captures the residuals or random noise after accounting for the trend and seasonal effects.

Evaluating the significance of the seasonal component

Understanding the significance of the seasonal component in time series data is crucial for accurate forecasting and effective model application across various industries. This section explores the statistical tests used to confirm seasonality, discusses the impact of significant seasonal components on forecasting models, and examines real-world case studies demonstrating the effectiveness of seasonal decomposition.

To confirm the presence of a seasonal pattern in your data, several statistical tests can be employed. These tests help validate whether the seasonal fluctuations are statistically significant or merely random variations.

- **Seasonal subseries plot**: A visual tool that can help in identifying seasonality by plotting the data's values within each season against the season itself. If consistent patterns are visible across similar seasons, seasonality is likely present:

    1. Install and load forecast package:

    ```
    # Check if the forecast package is already installed
    if (!requireNamespace("forecast", quietly = TRUE)) {
      # If not installed, install it
      install.packages("forecast")
    }
    # Load necessary libraries
    library(forecast)
    ```

    2. Generate seasonal plots:

    ```
    ggseasonplot(time_series, year.labels = TRUE, col = 1:12) +
      labs(title = "Seasonal Subseries Plot", x = "Month", y = "Value")
    ```

    We will get the following output:

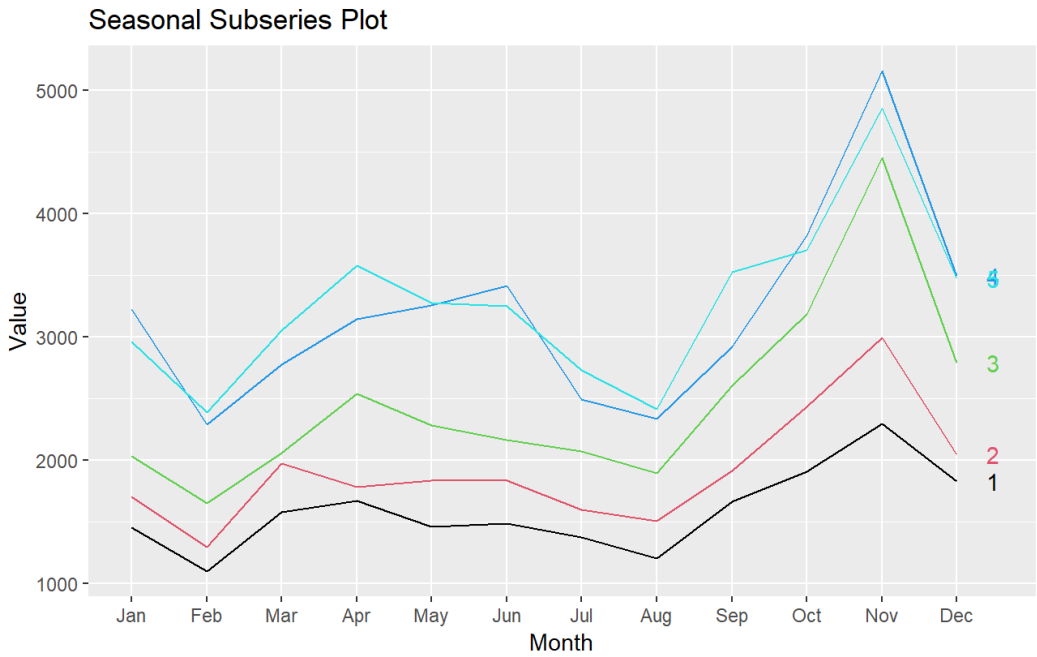


Figure 5.11: Seasonal plots of a time series data display 5 years subseries

The **Seasonal subseries plot** displays the monthly values for multiple years, with each line representing a different year (1 to 5 denoted by different colors). This plot reveals clear seasonal patterns, with peaks in November and December, while the values dip in February and July, suggesting recurring trends over the years.

- **Fourier analysis**: Converts time series data into its frequency components, allowing identification of dominant cycles which correspond to seasonal patterns:

```
spectrum(time_series, main = "Spectral Density", span = 12)
```

This produces the following graph:

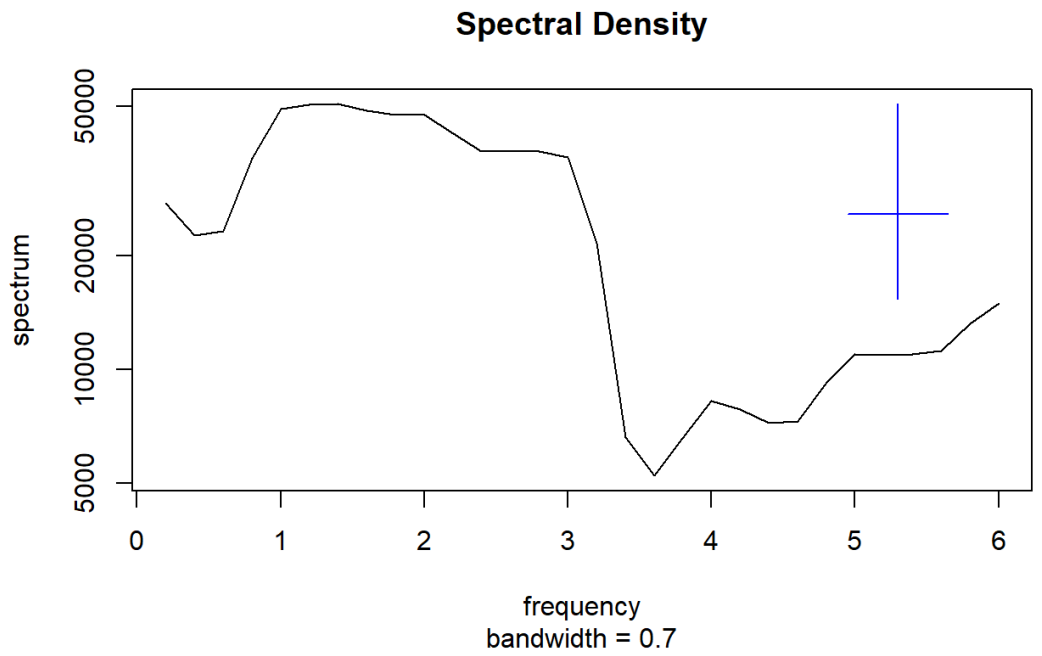


Figure 5.12: Spectral dentistry diagram for a time series dataset

The plot shows the **Spectral density** of a time series, where the x-axis represents the frequency and the y-axis represents the spectrum (intensity, power, or amplitude of the signal at each frequency). The large blue cross near the right side suggests a highlighted data point or uncertainty estimate, and the plot shows a general trend of decreasing power around frequencies 2 to 4 (This indicates that the time series has more variation (power) at 2 and less variation at 4), followed by a slight increase near frequency 6.

Next, you will touch on the tools and techniques for autocorrelation analysis.

Autocorrelation analysis

Auto-correlation, often referred to as serial correlation, is a measure of the correlation of a time series with its own past and future values. It quantifies the degree to which current values of a time series are related to its historical values. This concept is fundamental in time series analysis because it can reveal hidden patterns within the data, which are crucial for accurate forecasting and understanding the underlying mechanisms of the dataset.

Auto-correlation is particularly significant in the fields of economics, finance, meteorology, and various other disciplines where time series data are used. In these contexts, recognizing

and understanding auto-correlation can lead to more effective predictions and insights into dynamic processes.

Impact of autocorrelation on statistical inference and model assumptions

Auto-correlation affects the assumptions underlying many statistical modeling techniques, particularly those that assume independence among observations. For instance, in regression analysis, the presence of auto-correlation violates the assumption of independent errors, leading to biased and inefficient parameter estimates and affecting the validity of hypothesis tests:

- **Biased statistical inferences**: When positive auto-correlation is present in regression residuals, standard errors of estimate tend to be underestimated, which can falsely increase the statistical significance of predictors in a regression model. This leads to confidence intervals that are too narrow and p-values underestimate the true probability of observing the estimated effects.
- **Model assumptions and corrections**: Understanding the impact of auto-correlation necessitates adjustments in the modeling approach, such as using time-series-specific methods like **AutoRegressive Integrated Moving Average** (**ARIMA**) models, which explicitly account for auto-correlation. Additionally, techniques such as pre-whitening (transforming it into a "white noise" series with no discernible patterns or autocorrelation) are used to remove auto-correlation from the series before applying traditional statistical tests or regression models.

By identifying and correcting for auto-correlation, analysts can enhance the accuracy of their models and ensure more reliable conclusions from their data analyses. This section will further explore how to detect auto-correlation using graphical and statistical methods and discuss strategies to adjust models to account for this phenomenon, thereby improving the robustness of time series analyses.

Detecting autocorrelation

Detecting auto-correlation in time series data is important for effective modeling and analysis. Various methods and techniques are employed to identify the presence and extent of auto-correlation, ensuring that any models built reflect the true dynamics of the data. Two such methods are presented here:

Lag plots

Lag plots are simple scatter plots of a time series against its own lagged values. Observing a systematic pattern (such as a linear arrangement) in these plots suggests the presence of auto-correlation:

1. Load `dplyr` package:

```
library(dplyr)
```

2. Create lag data from time series:

```
lag_data <- data.frame(
  actual = df$revenue[-1],  # Remove the first observation
  lag1 = lag(df$revenue, 1)[-1]  # Lag by 1, remove the first NA
)
```

3. Load `ggplot2` and plot the `lag` plots:

```
library(ggplot2)

ggplot(lag_data, aes(x = lag1, y = actual)) +
  geom_point(color = "blue") +
  labs(title = "Lag Plot of Time Series (Revenue)",
       x = "Lag 1",
       y = "Actual Values") +
  theme_minimal()
```

`ggplot2` will display the following lag plot on the time series data.

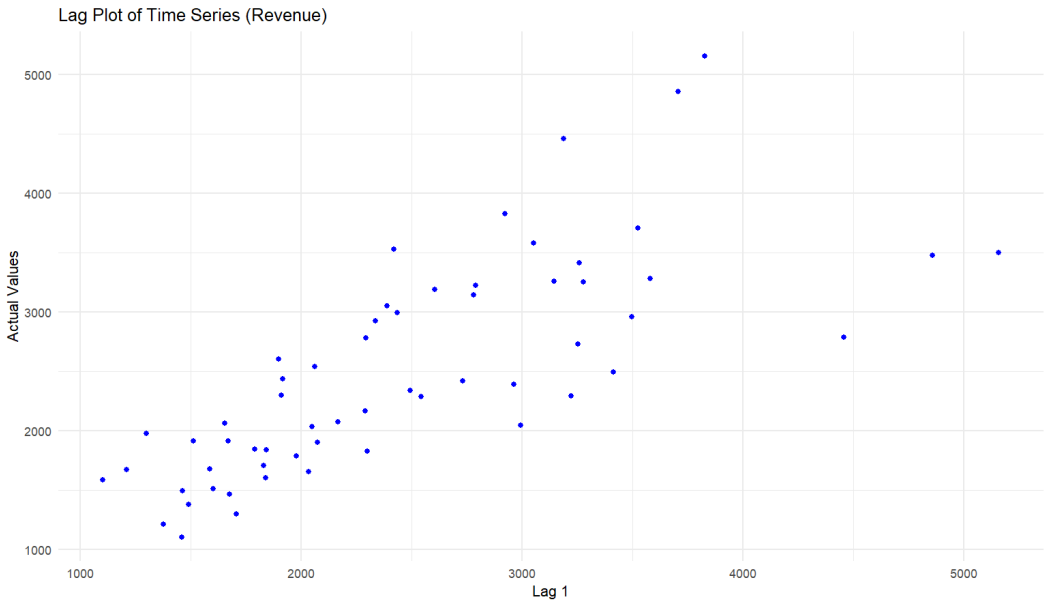


Figure 5.13: Lag plot diagram of time series data on revenue

The **Lag Plot of Time Series (Revenue)** shows the relationship between the revenue at a given time point (**Lag 1** on the x-axis) and the revenue at the subsequent time point (**Actual Values** on the y-axis). The plot reveals a positive relationship, suggesting that the revenue values from one period are moderately correlated with the values from the next period. This indicates some degree of persistence or autocorrelation in the data.

Autocorrelation function (ACF)

The ACF measures the correlation between a time series and its lagged values across different lag lengths. It helps to identify the degree of persistence or correlation in a time series, where each value is correlated with its past values at varying lags. In the context of time series analysis, the ACF is a useful tool for detecting repeating patterns, seasonality, or trends. High values at certain lags indicate a strong correlation between values separated by that lag:

```
acf(df$revenue, main = "ACF of Time Series (Revenue)")
```

The code script will calculate and plot the ACF of the revenue time series data.

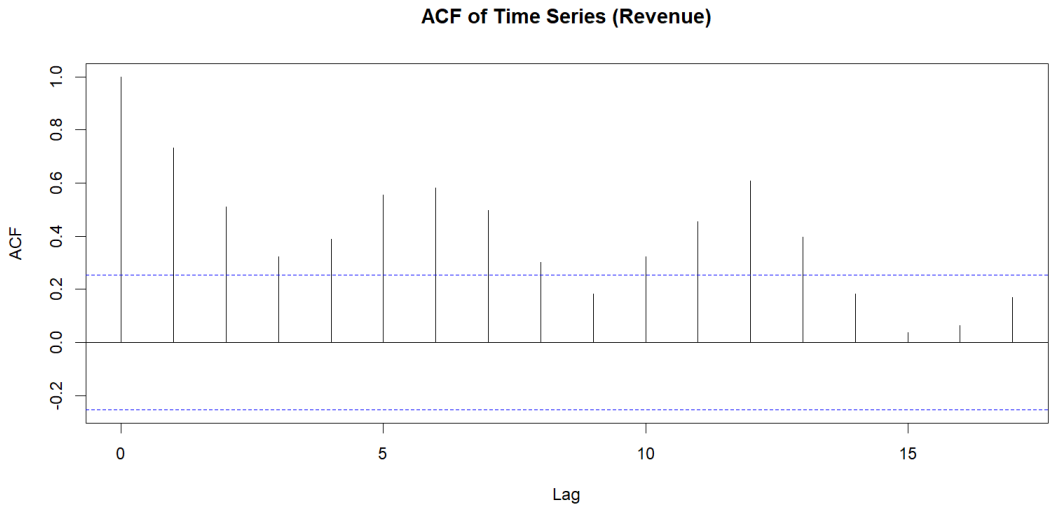


Figure 5.14: ACF of time series on revenue data

The **ACF of Time Series (Revenue)** plot displays the autocorrelation of the time series at various lags. Significant spikes at lags 1, 2, 3, and several others indicate that the revenue values are strongly correlated with their past values up to those lag lengths. The dashed blue lines represent the 95% confidence interval, and any spike that goes beyond this boundary is statistically significant.

Partial Autocorrelation Function (PACF)

The PACF differs from the ACF in that it measures the correlation of the residuals of a time series with the next lagged value, removing the influence of the intermediate lags. Essentially, the PACF shows the direct relationship between a time series and its lag after accounting for the influence of shorter lags. This is particularly helpful in identifying the number of AR terms to include in an ARIMA model, as the PACF cuts off after the number of significant lags.

Plotting the PACF:

```
pacf(df$revenue, main = "PACF of Time Series (Revenue)")
```

The code snippet will display a PACF of revenue time series data.

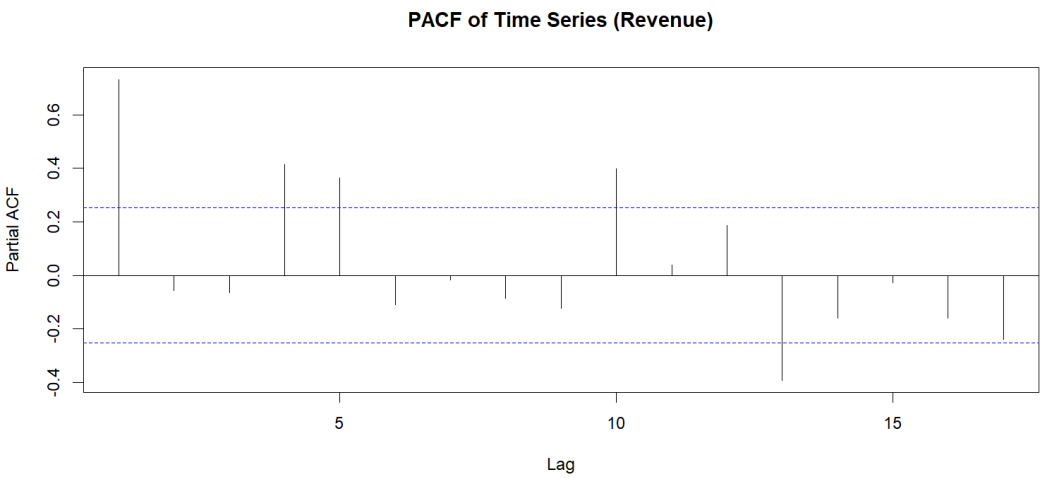


Figure 5.15: PACF of time series on revenue data

The **PACF of Time Series (Revenue)** plot displays partial autocorrelation for different lags. The first lag shows a significant spike, indicating that there is a strong direct relationship between the current revenue value and the value one period before. Other lags show smaller correlations, suggesting weaker or indirect relationships with the current values once the immediate lag effects are accounted for. The dashed blue lines indicate the 95% confidence interval, and values outside this boundary are considered statistically significant.

These plots help to visually assess the presence and strength of periodic structures in the time series data, which are crucial for specifying the terms of ARIMA models effectively. The interpretation of these plots directly influences the accuracy and efficiency of subsequent forecasts, making them essential tools in the arsenal of any analyst working with time series data.

In summary, ACF is useful for identifying overall patterns and seasonality across various lags. PACF helps isolate the direct effect of a given lag, filtering out the influence of all previous lags.

Next, you will learn the techniques for dealing with anomalies in time series, which is important for the analytic process.

Detecting anomalies

Anomaly detection in time series data is a critical analytical process aimed at identifying patterns that do not conform to expected behavior. The primary purpose of detecting these anomalies is to ensure the integrity and reliability of data analysis. Anomalies, often referred to as outliers, can significantly skew the results of statistical analyses and predictive modeling, leading to misleading conclusions if not properly managed.

Anomalies can have profound effects on time series forecasting and trend analysis. For example, Anomalies can distort predictive models, which typically assume that historical patterns will continue unless accounted for. Unexpected spikes or dips in the data may lead these models to generate inaccurate forecasts. Outliers can lead to false perceptions of trends. For example, a single anomalous spike might suggest a trend where none exists, or mask a decline if unusually high.

Robust statistical models depend on clean, outlier-free data. The presence of anomalies can affect the stability and reliability of these models, impacting their utility in real-world scenarios.

Recognizing and appropriately handling these anomalies is paramount for maintaining the quality and accuracy of insights derived from time series data.

There are two primary approaches for detecting anomalies: visual techniques and statistical techniques. Each provides unique insights and complements the other, allowing for a thorough examination of the time series.

Visual techniques

Utilizing visual techniques is one of the most intuitive ways to spot anomalies in time series data. By plotting the data, analysts can quickly identify unusual spikes, drops, or irregular patterns that deviate from typical trends. Key methods include:

- **Time series plots**: Plotting the data points in a time sequence can highlight unexpected deviations against the backdrop of regular patterns
- **Scatter plots**: Useful for identifying outliers in datasets that compare multiple variables, where anomalies may not follow the trend observed in the majority of data points

These visual methods are particularly effective for initial anomaly detection and can provide immediate insights before deeper statistical analysis. The following figure is an example display on an anomaly detection plot:

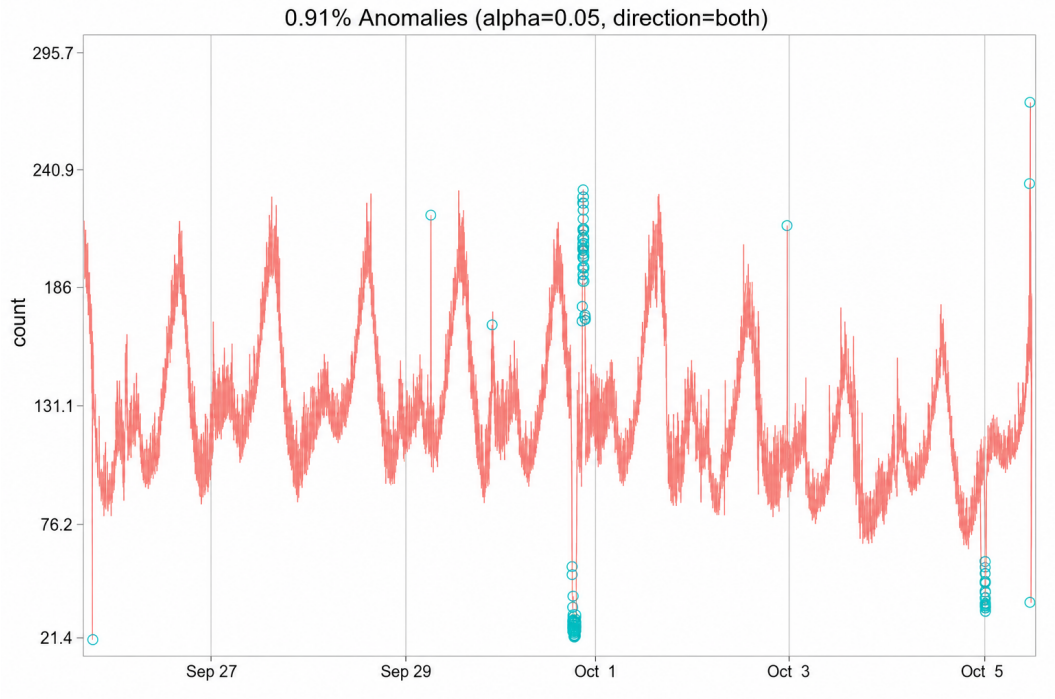


Fig 5.16: A sample anomaly detection visualization plot showing the color circled anomalies (Source: `https://www.r-bloggers.com/2016/01/a-graphical-extension-of-twitters-anomalydetection-package/`*)*

The diagram shows both positive and negative anomalies. You will see sample code of how to use Twitter's `AnomalyDetectionTs` later in this chapter.

Statistical techniques

Statistical techniques provide a systematic approach to identifying outliers in time series data. Here are examples of how to implement z-scores and **Interquartile Range** (**IQR**) methods in R to detect anomalies:

Using z-scores

Z-scores are a popular method for identifying outliers, as they measure how far an observation is from the mean in terms of standard deviations. Here's how to calculate and identify outliers using z-scores in R:

1. Calculate z-scores:

```
z_scores <- scale(time_series) # scale() function standardizes the data
```

2. Define threshold and identify outliers:

```
threshold <- 2

outliers <- which(abs(z_scores) > threshold)
```

3. Print outliers:

```
cat("Outlier indices:", outliers, "\n")
Outlier indices: 35 47 59
```

This code generates a sample dataset, computes z-scores, and identifies any data points where the z-score exceeds 2, which is typically considered a significant deviation from the mean. We can see three months with values that would be considered outliers.

Using the interquartile range (IQR)

The IQR method is effective in identifying outliers by focusing on the statistical range between the 25th and 75th percentiles. Here's how to apply the IQR method to detect outliers in R:

1. Calculate the IQR values:

```
Q1 <- quantile(time_series, 0.25)
Q3 <- quantile(time_series, 0.75)
IQR_value <- Q3 - Q1
```

2. Define the boundary for outliers:

```
lower_bound <- Q1 - 1.5 * IQR_value
upper_bound <- Q3 + 1.5 * IQR_value
```

3. Identify outliers and print them out:

```
outliers <- which(time_series < lower_bound | time_series > upper_bound)

cat("Outlier indices:", outliers, "\n")
Outlier indices:
```

This script calculates the first and third quartiles and the IQR, then defines the lower and upper bounds for non-outlier data points. It identifies as outliers any points that fall outside these bounds.

No outliers were detected via the IQR method. Compared to the Z-score method, which leverages mean and standard deviation for z-score, and IQR focuses on median and quartiles, it's not strange that they yield different results.

By utilizing these statistical techniques, you can effectively identify and handle outliers in your time series data, ensuring that your analysis is based on robust and reliable data.

Automated methods – Twitter anomaly detection libraries

The Twitter/X anomaly detection library provides a powerful tool for automatically detecting anomalies in time series data. This library is particularly useful in large datasets where manual anomaly detection would be impractical. It uses sophisticated statistical algorithms to detect unexpected changes in data trends, which can be critical for timely decision-making in various industries.

Here's a step-by-step guide to using the Twitter Anomaly Detection library in R:

1. First, you need to install `devtools` if it is not installed yet:

```
if(!require(devtools)) install.packages("devtools")
```

2. Next, you need to install the library from GitHub, as it might not be available directly through CRAN:

```
devtools::install_github("twitter/AnomalyDetection")
```

3. Once installed, you can load the library into your R session:

```
library(AnomalyDetection)
```

4. With the data prepared, you can now run the anomaly detection:

```
anomaly_data <- data.frame(
timestamp =  df$Month.with.Year,
revenue = df$revenue  )
```

5. Now run the anomaly detection:

```
result <- AnomalyDetectionTs(anomaly_data, max_anoms = 0.05, direction =
'both', plot = TRUE)
```

```
anomalies <- result$anoms
plot <- result$plot
```

6. The output of the function includes the anomalies detected and optionally a plot showing these anomalies. You can further explore or export the results:

```
print(anomalies)

 data frame with 0 columns and 0 rows

ggsave("anomaly_detection_plot.png", plot=plot)
```

   There are no anomalies detected, so the output is empty.

This example demonstrates the use of the Twitter Anomaly Detection library to automate the detection of anomalies in time series data. By automating this process, you can efficiently monitor large datasets for unusual activities or changes without the need for manual inspection. This capability is particularly valuable in areas like financial monitoring, infrastructure maintenance, and operational surveillance, where real-time data analysis can provide crucial insights for immediate actions.

It seems that the `AnomalyDetectionTs()` function did not detect any anomalies in your data, as indicated by the "0 obs. of 0 variables" for `anomalies`. This means that the function didn't identify any data points that significantly deviated from the expected patterns within the given threshold (`max_anoms = 0.05`).

Summary

In this chapter, you got a comprehensive overview of **Exploratory Data Analysis** (**EDA**), specifically for time series data, a key aspect of data science and statistics used to uncover trends, seasonal patterns, and anomalies in data. The chapter blends theoretical concepts with practical applications in R, offering step-by-step guidance and hands-on code examples to help users apply these techniques to real-world datasets.

You began by discovering the significance of trend analysis, explaining how to identify and interpret upward, downward, and stable trends. You explored various methods for detecting trends, from basic visual inspection to more advanced techniques like statistical regression and Mann-Kendall testing. You learned how to plot time series data, apply trend lines, and statistically validate these trends using R.

Smoothing techniques, such as **simple moving averages** (**SMA**), **exponential moving averages** (**EMA**), and **Locally Estimated Scatterplot Smoothing** (**LOESS**), were discussed as

essential methods to reduce noise in time series data, providing a clearer view of underlying trends. Comparisons between these techniques help analysts choose the right approach depending on the data's characteristics.

You also covered seasonal decomposition, emphasizing the importance of separating time series data into trend, seasonal, and irregular components. Using R functions like `decompose()` and `stl()`, readers learn how to apply seasonal decomposition to uncover hidden patterns and better understand the cyclical nature of time series data.

Auto-correlation and partial auto-correlation are introduced and methods for detecting auto-correlation, including **Auto-Correlation Function** (**ACF**) and **Partial Auto-Correlation Function** (**PACF**) plots, are explained, providing insight into the persistence of patterns in time series data.

Finally, the chapter addresses anomaly detection, teaching you how to spot outliers that may distort analysis and predictions. Various techniques, including visual methods, z-scores, and the Twitter Anomaly Detection library, are presented to help you efficiently identify and handle anomalies in large datasets.

In the next chapter, we will dive into **Exponential Smoothing** techniques, including **Simple**, **Double**, and **Triple Exponential Smoothing** (Holt-Winters).

6

Exponential Smoothing and Holt-Winters

It's time to explore the fascinating world of exponential smoothing and Holt-Winters. Using R, you will get hands-on experience turning data into insights. We will explore simple, double, and triple exponential smoothing, with focused coding exercises to connect theoretical concepts with practice.

Error measurement is another exciting part of this chapter. You'll learn how to evaluate forecast accuracy using key error metrics, helping you understand how well your models perform. We'll work with daily data, discuss the strengths and limitations of each method, and build the judgment needed to choose the right approach in different situations. We will cover the following sections in the chapter:

- Customer support case study
- Simple exponential smoothing
- Double exponential smoothing
- Triple exponential smoothing (also known as Holt-Winters)

Technical requirements

R and RStudio are required for this chapter. Github scripts can be found in the repository.

You will need the following data file and packages for this chapter:

- `TTR`: The **TTR** (Technical Trading Rules) package in R is often used for technical analysis of financial market data.

- `Forecast`: The **forecast** package in R is a very popular set of tools designed for analyzing and forecasting univariate time series data. You will see some seasonal subseries plots using this package.
- `tsibble`: The **tsibble** package in R provides data class and functions for handling tidy temporal data frames.
- `feasts`: The **feasts** (Feature Extraction And Statistics for Time Series) package in R provides tools for analyzing, visualizing and extracting features from time series data.
- `lubridate`: The **lubridate** package in R provides tools that make it easier to parse and manipulate dates.
- `weekly_customer_complaints.csv`: The weekly customer complaints data file we will use to process, and demonstrate data handling, simple, double, triple exponential smoothing modeling and perform model assessment.
- `bitcoin_price.csv`: The daily data file we will use to demonstrate the Holt-Winters model.

Customer support case study

Let's go through the case study covering everything from the situation, the problem, the strategy, and so on:

- *The situation* is that TelecoWave, a major player in the telecom industry, is struggling with fluctuating complaints. Customer complaints are unpredictable – some weeks they're minimal, but in others, they spike without warning, creating a lot of confusion and operational strain.
  They've brought us in to make sense of this. Our task? Predict these swings in complaints and bring some order to the chaos.
- *What's at stake* is if they get the staffing wrong, it either leads to wasted resources (too many reps during low periods) or unhappy customers (not enough reps when complaints surge). It's about ensuring their customer service runs smoothly and efficiently.
- *Why it matters* is that if we can crack this, TelecoWave can go from reactive to proactive, forecasting demand based on analytics and aligning their workforce with customer needs. Fewer complaints slipping through the cracks will lead to more satisfied customers and better business health overall.

Now, let's get to work:

1. Install and load the necessary packages:

```
packages <- c("readr","tsibble","feasts", "ggplot2", "forecast", "TTR",
"dplyr" )

for (p in packages) {
  if (!requireNamespace(p, quietly = TRUE)) {
    install.packages(p)

  }
  library(p, character.only = TRUE)
}
```

   The preceding code will loop through the package list. If a package is not installed, it will be installed. Also, load the package accordingly. Please note, `character.only` needs to be set to `TRUE` to let R treat the argument as a character string or a variable that contains the name of the package.

2. Load the data as data frame from the csv file:

```
df <- read_csv("weekly_customer_complaints.csv")
```

3. Check and display the first couple of rows:

```
head(df)
```

   To begin the exploratory data analysis, we first take a look at the initial few rows of the dataset to understand its structure and content:

```
# A tibble: 6 × 6
  week      complaints discount_rate small_commercial_event
  <chr>          <dbl> <chr>                           <dbl>
1 1/1/2018        1750 37.26%                              1
2 1/8/2018        1701 35.38%                              0
3 1/15/2018       1711 36.22%                              1
4 1/22/2018       1443 38.33%                              1
5 1/29/2018       1397 36.34%                              0
6 2/5/2018        1477 36.29%                              1
# ℹ 2 more variables: medium_commercial_event <dbl>,
#   big_commercial_event <dbl>
```

Figure 6.1: Initial rows of the dataset

In this step, we import libraries for data manipulation, visualization, and time series analysis. We then read our dataset into a data frame and display its first few rows.

4. You then convert the data to a `tsibble` and fill any gaps to ensure it's ready for time series analysis; the `tsibble` object in R is a "tidy" time series data structure that manages temporal data, containing a time index and measured variables:

```
df <- df %>%
  mutate(week = yearweek(week)) %>%
  as_tsibble(index = week) %>%
  fill_gaps()

head(df)
```

This gives us the following output:

```
# A tsibble: 6 x 6 [1W]
      week complaints discount_rate small_commercial_event
     <week>      <dbl> <chr>                          <dbl>
1 2018 W01        1750 37.26%                             1
2 2018 W02        1701 35.38%                             0
3 2018 W03        1711 36.22%                             1
4 2018 W04        1443 38.33%                             1
5 2018 W05        1397 36.34%                             0
6 2018 W06        1477 36.29%                             1
# ℹ 2 more variables: medium_commercial_event <dbl>,
#   big_commercial_event <dbl>
```

Figure 6.2: Initial rows of data set after converting and filling in gap

This step pipes the data frame to a mutate function, which updates the week column with the Year-Week value using the `yearweek()` function. It then transforms our data frame into a `tsibble`. The `fill_gaps()` function will convert implicit missing values into NA, which fills any missing weeks to ensure continuity in the time series.

5. Visualize the weekly customer complaints. We use the `ggplot()` function from the `ggplot2` package to create a line plot of weekly customer complaints, helping us identify any obvious trends or patterns:

```
ggplot(df, aes(x = week, y = complaints)) +
  geom_line() +
  labs(title = "Weekly Customer Complaints")
```

We get *Figure 6.3* as the output:

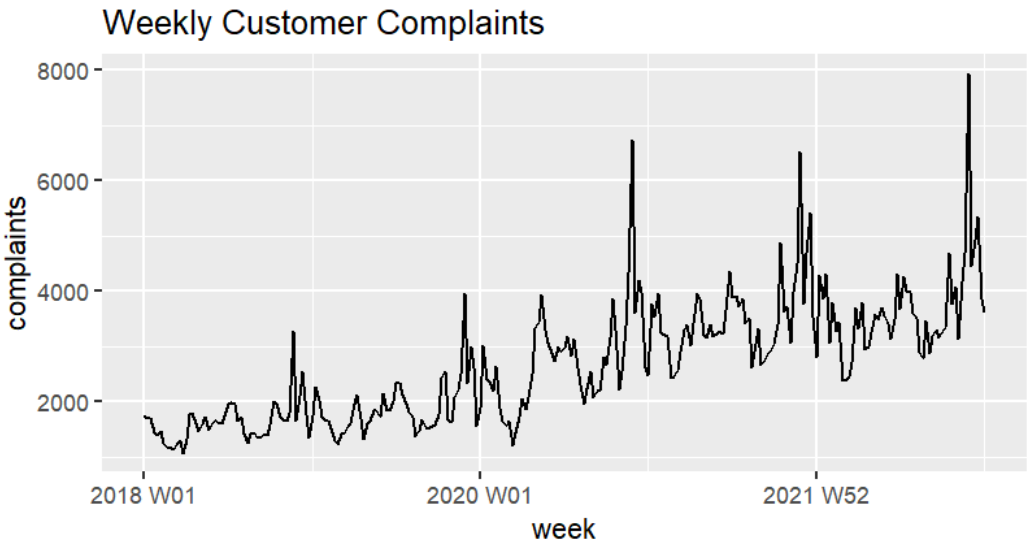


Figure 6.3: A time series plot of weekly customer complaints

6. Next, we create a new month column.
   Using a lambda expression, apply the `yearmonth()` function to the existing index variable of the `tsibble`. The `.` refers to the index column of the data. Function `index_by()` will add a new month column and then group the data using the month index:

```
month_df <- df %>%
  index_by(month = ~ yearmonth(.))
head(month_df)
```

   This will display the data structure as follows:

```
# A tsibble: 6 x 7 [1W]
# Groups:    @ month [2]
      week complaints discount_rate small_commercial_event medium_commercial_event big_commercial_event    month
     <week>      <dbl> <chr>                          <dbl>                   <dbl>                 <dbl>    <mth>
1 2018 W01       1750 37.26%                             1                       0                     0 2018 Jan
2 2018 W02       1701 35.38%                             0                       0                     0 2018 Jan
3 2018 W03       1711 36.22%                             1                       0                     0 2018 Jan
4 2018 W04       1443 38.33%                             1                       0                     0 2018 Jan
5 2018 W05       1397 36.34%                             0                       0                     0 2018 Jan
6 2018 W06       1477 36.29%                             1                       0                     0 2018 Feb
```

Figure 6.4: The data structure after adding the month column

Summarize each month to one row. The `summarize()` function will create a new data frame. It returns one row for each month, calculating the mean value of that month's complaints:

```
month_df <- month_df %>%
  summarize(complaints = mean(complaints))

head(month_df)

# A tsibble: 6 x 2 [1M]

     month complaints
     <mth>      <dbl>

1 2018 Jan      1600.
2 2018 Feb      1270.
3 2018 Mar      1187.
4 2018 Apr      1599.
5 2018 May      1602.
6 2018 Jun      1657.
```

Tip

You can use either `summarize()` or `summarise()`. Either will work!

7. Then we use the `gg_season()` to produce a time series seasonal plot. The x-axis shows the data from within each month:

```
month_df %>%
  gg_season(complaints, labels = NULL) +  # remove or adjust labels
  labs(title = "Monthly Seasonality of Customer Complaints")
```

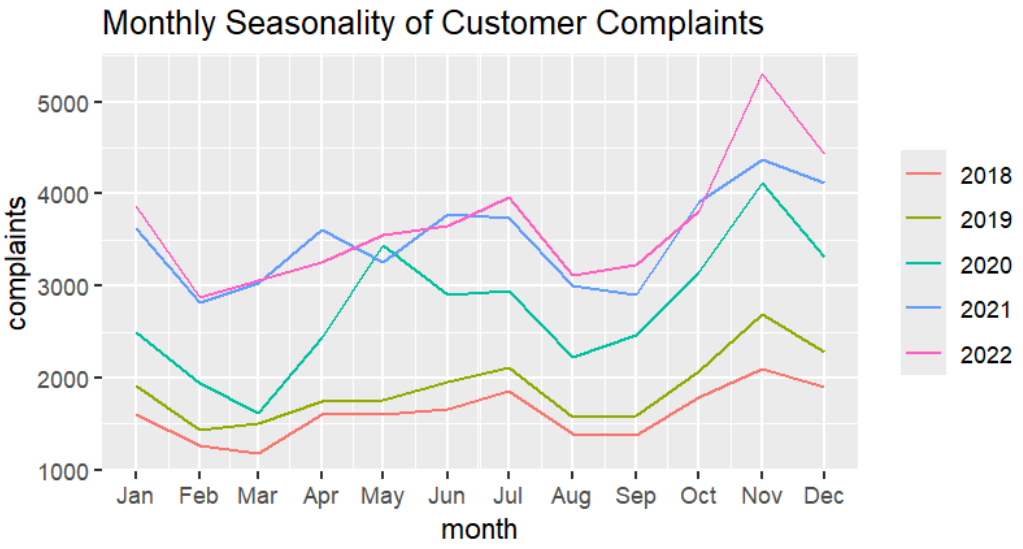


Figure 6.5: A plot showing the monthly seasonality of customer complaints from 2018 to 2022

There is a noticeable peak in complaints around October and November across multiple years, with 2022 experiencing the highest spike in complaints.

8. Plot the quarterly seasonality of complaints. Following a similar code pattern, the following code shows the way to use the `yearquater()` function and eventually plot the quarterly seasonality for complaints:

```
df %>%
  index_by(quarter = ~ yearquarter(.)) %>%
  summarize(complaints = mean(complaints)) %>%
  gg_season(complaints, labels = NULL) +
  labs(title = "Quarterly Seasonality of Customer Complaints")
```

The next visualization summarizes the quarterly seasonality of customer complaints, revealing how complaint patterns fluctuate across different quarters of the year.

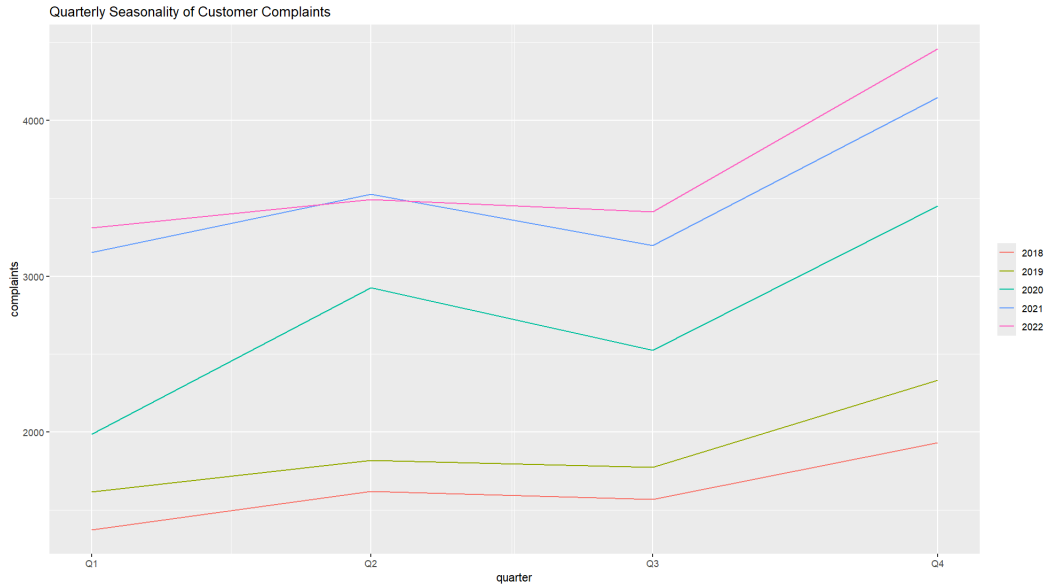


Figure 6.6: This chart shows the quarterly seasonality of customer complaints from 2018 to 2022

Similarly, the chart displays the quarterly seasonality of customer complaints from 2018 to 2022. Complaints show a general increase in Q4 across most years, with 2022 showing the most significant rise in complaints during the final quarter.

Training and test in time series

Let me introduce you to the concept of splitting a data set into training and test sets. It is simple and straightforward, yet it is powerful. Imagine you have a dataset that is represented by the top rectangle in *Figure 6.7*.

What you do is split the data set, leaving around 80% for training and 20% for testing purposes (also known as the **Pareto Principle**, frequently used in machine learning as a rule of thumb for splitting data into training and testing sets). The key idea is to create a model using the training data and assess it with the test data. The goal is to have an unbiased way of evaluating your model. Simple right?

This image illustrates the typical division of a dataset into training and test subsets, with around 80% of the data used for training and 20% reserved for testing. The test data helps to evaluate the model's performance on unseen data.



Figure 6.7: Visualizing the dataset versus the training and test data split

Time series data is different since information on a given day is meaningless if it does not have the context of the surrounding days, and we usually want to predict the future.

Imagine *Figure 6.8* is our time series data. In time series analysis, the structure of the data is different, as each point is connected to its preceding and following points. This figure demonstrates the flow of time series data, where each observation depends on the previous ones.



Figure 6.8: Visualizing a time series dataset

The key principle in splitting time series data is to preserve the historical order of the data to maintain the context. In practice, this often ends up removing the last observation. Let's imagine the last one is taken out. The yellow balls become the Training set and the light blue ball is the test set, shown as follows:

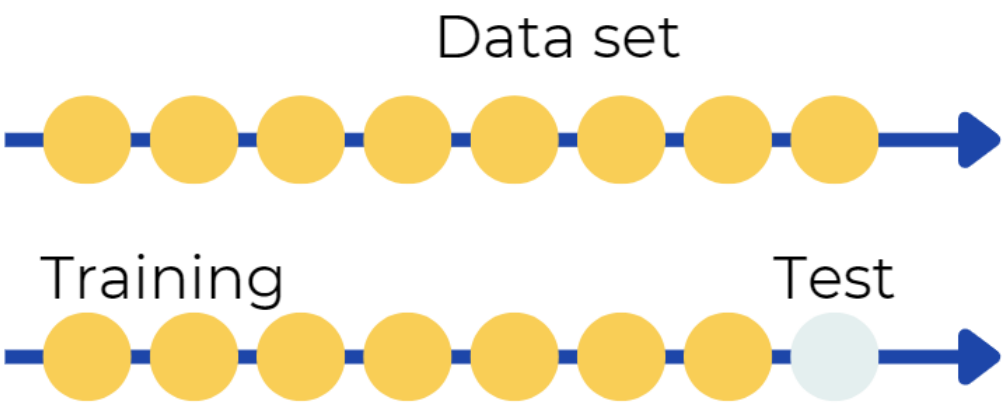


Figure 6.9: Representation of the time series dataset, split into training and test data

Another essential piece of information is that the test set should be the number of periods you expect the model to predict in practice. To explain it differently, if you are creating a model to predict the next four weeks, you should also assess how it performs forecasting in periods of 4 weeks.

The last thing I wanted to mention is that for the training set, you need to have at least one whole period of information, ideally, 2 or 3, to achieve robust forecasts.

Evaluating your model

To evaluate your models, we will split the data into training and test sets:

1. Set the test data period (in weeks):

```
periods <- 13
```

2. Convert the previous loaded data frame to `tsibble`:

```
df$week <- as.Date(df$week)
df <- df %>%
  as_tsibble(index = week)
```

3. Load library and retrieve the last week from data frame:

```
library(lubridate)
max_week <- max(df$week)
```

4. Split the data into training and test sets:

```
train <- df %>%
  filter(week < (max_week - weeks(periods)))  # Training set: before last
13 weeks

test <- df %>%
  filter(week >= (max_week +1 - weeks(periods)))  # Test set: last 13 weeks
```

5. Display the first couple rows of test data:

```
head(test)
```

This gives us the following output:

```
# A tsibble: 6 x 6 [7D]
  week       complaints discount_rate small_commercial_event
  <date>          <dbl> <chr>                           <dbl>
1 2022-10-03       3350 29.78%                              0
2 2022-10-10       4679 31.62%                              0
3 2022-10-17       3773 32.37%                              0
4 2022-10-24       4065 32.95%                              0
5 2022-10-31       3142 32.64%                              0
6 2022-11-07       4164 30.20%                              0
# ℹ 2 more variables: medium_commercial_event <dbl>,
#   big_commercial_event <dbl>
```

Figure 6.10: Display of the first couple rows of test data set

We split the data such that the last 13 weeks are used as the test set while the remaining data forms the training set.

Now the data is split into training and test sets, let's chat about simple exponential smoothing and how we can assess the trends in our data!

Simple exponential smoothing

Simple exponential smoothing (**SES**) is a method used in time series forecasting, particularly useful when there are no clear trends or seasonality in the data. It applies exponentially decreasing weights to older data points and gives more emphasis to recent observations, meaning recent data has a greater influence on the forecast.

We'll cover the basics of simple exponential smoothing and how it fits into the broader category of exponential smoothing methods, which include **double exponential smoothing** and **triple exponential smoothing**. By the end of this section, you will understand how to apply this technique and its variations to real-world datasets.

So, what's this all about? Imagine you're checking out our weekly complaint figures. Some weeks are super busy, others not so much. We're trying to find a steady rhythm in these numbers, and that's exactly where simple exponential smoothing steps in.

Let's talk about the formula without getting too deep into math:

$$Next\ Forecast = Current\ Level + \alpha \times (Recent\ Actual - Current\ Level)$$

Here's the breakdown:

- **Next forecast**: This is the forecasted value for the upcoming period.
- **Current level**: Think of this as our baseline, our starting point. It's like saying, "Based on everything up to now, this is where we stand."
- **Recent actual**: This is our latest observation actual value, the most recent piece of real data we've got, for example, last week's customer complaint numbers.
- **α (Alpha)**: This is our smoothing constant, a value between 0 and 1. When it's close to 1, it means we're putting a lot of stock in what just happened. If it is closer to 0, it means that the past matters more.

Let's put it into context. Say our current week's level is 1,000 complaints (that's our baseline), but in the previous week, we actually had 1,200 complaints.

If α is 0.2, our forecast for next week doesn't just jump to 1200. Instead, it adjusts up a bit, acknowledging that we did better than our baseline but not going all-in on one week's spike.

If we apply the formula, the current level being 1000, recent actual to 1200, and an alpha chosen of 0.2, then the prediction for next week is 1040. So, how do we apply it to our data?

Applying simple exponential smoothing to our data

When we look at our weekly complaints data and apply this simple exponential smoothing, we're trying to get past the spikes in *Figure 6.3* – like that week when we had a surprise rush. It's great for a quick, broad-stroke idea of where we're headed.

We start with simple exponential smoothing to make our first set of predictions.

1. Convert the complaint data frame to a time series object:

```
complaints_ts <- ts(train$complaints,
                    start = c(year(min(train$week)),
```

```
                                week(min(train$week))),
                    frequency = 52)
```

The code script created a time series object from the complaints column of the training data set, which was created in previous code snippets. The training data set contains the data before the last 13 weeks. The frequency was set to 52, which is the value of observations per unit of time; here it refers to the data being weekly (do not confuse this as the total data points count).

2. Fit the simple exponential smoothing model:

```
model_simple <- forecast::ses(complaints_ts, h = periods)
```

We fit the simple exponential smoothing (SES) model using `ses()` function, and set `h` (the number of periods for forecasting into the future) to the value of 13 weeks.

3. Forecast the future period using the `forecast()` function to forecast the next 13 weeks:

```
predictions_simple <- forecast::forecast(model_simple, h = periods)
```

4. We then convert the test data set to a time series object, use similar code as for the training data set:

```
complaints_ts_test <- ts(test$complaints,
                    start = c(year(min(test$week)),
                              week(min(test$week))),
                    frequency = 52)
```

5. Plot the training, test, and forecast on one chart:

```
autoplot(complaints_ts, series = "Train") +
  autolayer(complaints_ts_test, series = "Test") +
  autolayer(predictions_simple, series = "Forecast", alpha=0.5) +
  labs(title = "Train, Test, and Predictions with Simple Exponential
Smoothing") +
  theme_minimal()
```

This creates the following plot:

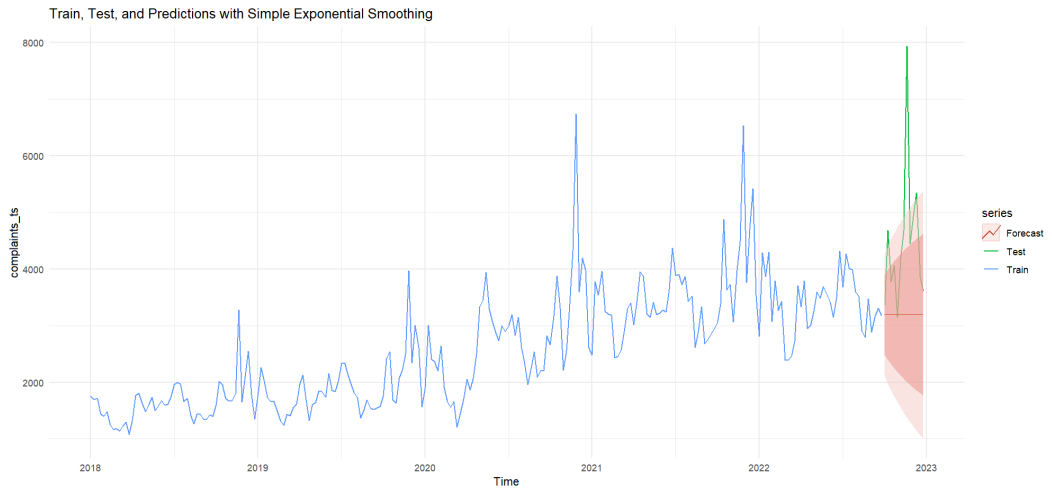


Figure 6.11: Simple Exponential Smoothing Predictions

Here, we plot (using the `autoplot()` function) the training data, test data, and the SES forecasts data into one chart. We set the alpha value to 0.5 for the prediction layer so that the testing data set can be viewed. As shown on the chart, for the forecasts it has a line (point forecast), with the darker shaded area typically the 80% prediction interval, and the lighter shade represents the 95% prediction interval.

While the SES method is a solid tool in our kit, it's not a fortune-teller. It won't catch big trends or seasonal rushes. To capture more detail, we need to proceed to the next level: *double* exponential smoothing.

Double exponential smoothing

What makes it "double"? In SES, we focused on smoothing out the data. Double exponential smoothing adds another layer – it tackles trends in our data too. So, we're not just averaging out the highs and lows; we're also catching if our weekly complaints are generally going up or down over time.

Double exponential smoothing essentially uses two equations:

- **Smoothing the level**: Like we did before:

$$Smoothed\ Level = \alpha \times Recent\ Actual + (1 - \alpha) \times (Previous\ Level + Previous\ Trend)$$

- **Smoothing the trend:** This is the new part:
$Smoothed\ Trend = \beta \times (Smoothed\ Level - Previous\ Level) + (1 - \beta) \times Previous\ Trend$

- **Smoothed level**: This is our updated baseline, taking into account the recent actual data (a lot of time this is actual data at time t).
- **Smoothed trend**: Here, we're looking at how much the level has changed from one period to the next – it's our trend factor.
- α and β: These are our smoothing parameters again. α adjusts how much we consider the recent complaints versus our previous level and trend. β adjusts how much weight we give to the change in level – basically, how quickly we think the trend is happening.

Putting double exponential into practice

Let's say our customer complaints have been increasing overall, but with some weekly ups and downs. Double exponential smoothing helps us see that upward trajectory, not just the weekly fluctuations.

> **Tip**
>
> **A few words of caution**
>
> While it's great for catching trends, remember that it still doesn't handle seasonality.

So double exponential smoothing is about smoothing out the noise and catching the wave – understanding not just where our complaints are but where they're heading.

Next, we apply double exponential smoothing to account for trends in the data:

1. Fit the model to data:

```
model_double <- holt(complaints_ts, damped = FALSE, h = periods)
```

The double exponential smoothing is often known as **Holt's method**; we use the `holt()` function to fit the model with the time series converted from the training set in previous code. If damped is `TRUE`, then it will use a damped trend; `h` is the number of periods of forecasting. Although in our model, we didn't pass in the alpha and beta value, it could be passed in as `holt(alpha= 0.452, beta= 0.0201)`.

You can also print out the model `p`; it will display both the model that we have and the alpha, beta parameter values:

```
print(model_double$model)
Holt's method

Call:
```

```
holt(y = complaints_ts, h = periods, damped = FALSE)

  Smoothing parameters:
    alpha = 0.5188
    beta  = 1e-04

  Initial states:
    l = 1833.8322
    b = -11.8717

  sigma:  554.1752

     AIC     AICc      BIC
4487.622 4487.871 4505.169
```

2. Forecast the future period:

```
predictions_double <- forecast::forecast(model_double,
                                         h = periods)
```

We fit a Holt model to the training data and forecast the next 13 weeks.

3. Plot the training, test data set together with the forecasts:

```
autoplot(complaints_ts, series = "Train") +
  autolayer(complaints_ts_test, series = "Test") +
  autolayer(predictions_double, series = "Forecast", alpha=0.5) +
  labs(title = "Train, Test, and Predictions with Double Exponential
Smoothing") +
  theme_minimal()
```

The following image illustrates the training, testing, and forecasted data using double exponential smoothing. This method is particularly useful for datasets with trends but no seasonality, as it accounts for level and trend components in the forecast:

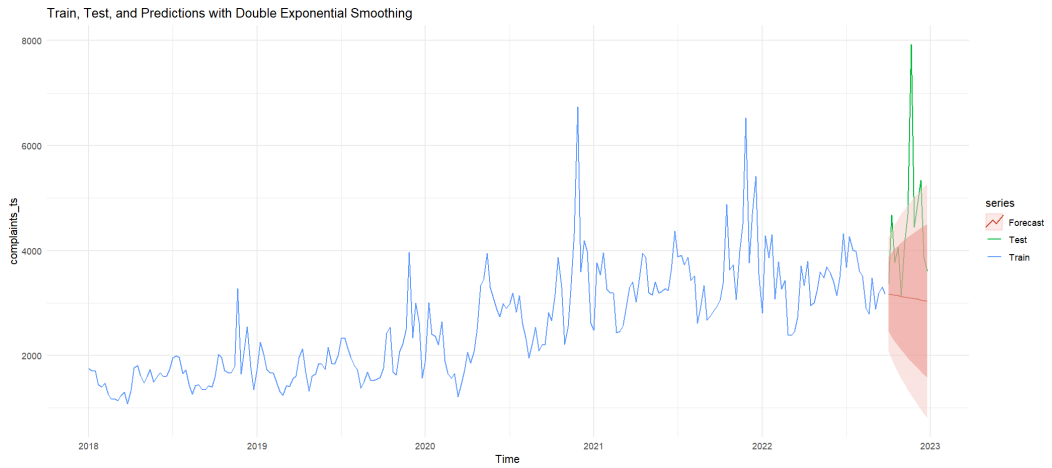


Figure 6.12: Double Exponential Smoothing: Train, Test, Forecast

Observe that the predictions, when compared to the test data, show a high number of errors. As displayed in the plot, the test data (real data) showed an upward trend, while predictions showed a downward trend. This code plots the training, test, and forecast data for double exponential smoothing, helping us see the model's effectiveness.

Let's go even further to *triple* exponential smoothing.

Triple exponential smoothing (also known as Holt-Winters)

Let's talk about triple exponential smoothing, often referred to as the **Holt-Winters method** (it was developed by Charles Holt and Peter Winters in 1960). This goes even further than the simple and double exponential smoothing methods we've discussed before. It's designed for data that has not just trends, but also seasonality—think of things like seasonal customer complaints patterns or temperature changes over the year.

What is "triple" about triple exponential smoothing? Triple exponential smoothing incorporates three components:

- **Level**: This is similar to what we saw in simple exponential smoothing. It's about the baseline value of our data.
- **Trend**: Here, we're looking at whether our data shows an increasing or decreasing trend over time.
- **Seasonality**: This is the new addition. We're considering the repeating patterns or cycles in our data over a specific period, like increased ice cream sales in summer.

Alpha, Beta, Gamma

Just like in our previous methods, we use coefficients (α, β, γ) to control how much weight we give to each component:

- α (Alpha): Controls the level; high alpha (closer to 1) gives more weight to recent observations, making the model react faster to changes
- β (Beta): Controls the trend; high beta makes the trend line more responsive to recent trend changes
- γ (Gamma): Controls the seasonality; high gamma gives higher weight to the most recent seasonal cycles, suitable if seasonal patterns shift quickly

Practical applications

Holt-Winters is fantastic for forecasting where you have seasonality. For instance, predicting electricity demand, which typically has daily and weekly seasonal patterns, or forecasting retail sales that spike during holidays.

We then use triple exponential smoothing to include seasonality in our model.

1. Fit the Holt-Winters model:

   ```
   model_triple <- hw(complaints_ts, seasonal = "multiplicative", h = periods)
   ```

   We used the multiplicative parameter for the complaints data. The primary difference between `additive` and `multiplicative` in Holt-Winters (`hw()`) is how seasonal fluctuations behave relative to the trend. When set to `additive`, seasonality adds a constant value to the trend, regardless of the trend level. When `multiplicative` is passed in, seasonality scales (multiplies) with the trend, meaning if the trend grows, the seasonal spikes also grow.

2. Here, we fit a Holt-Winters model to the training data and forecast the next 13 weeks:

   ```
   Error in ets(x, "AAA", alpha = alpha, beta = beta, gamma = gamma, phi =
   phi,  :
     Frequency too high
   ```

> Unfortunately, Holt-Winters has issues with weekly seasonality. The error seen here from the `ets()` function (which is called by `hw()`) occurs because the seasonal exponential smoothing models struggle to estimate the large number of initial seasonal states required for the algorithm.

The Holt-Winters method is a robust tool for dealing with complex patterns in time series data. However, it also has limitations: the `ets()` function has a limit of 24 for the seasonal frequency, so when we try to estimate the seasonal components, if the seasonal period in the time series data exceeds the maximum limit the `ets()` function can handle, it will error out.

The ways to address this error in R include adjusting the time series object frequency, using `tbats()` (we have coverage on `tbats()` in *Chapter 8*) or `auto.arima()` (covered in *Chapter 7*), or simply disregarding seasonality. We will switch to `holt()` to handle the complaints data set in the following code base and will have examples to use `hw()` to handle daily data later in this chapter.

Predicting the future period

Because of the error in the previous subsection, we switch to the `holt()` method to make future predictions using the entire dataset.

1. Fit the Holt's linear model (without seasonality):

```
model_holt <- holt(complaints_ts , h = periods)
```

2. Forecast the future period (13 weeks):

```
forecast_holt <- forecast(model_holt, h = periods)
```

3. Plot the forecast:

```
plot(forecast_holt)
```

   We fit the model to the complete dataset and forecast the next 13 weeks.

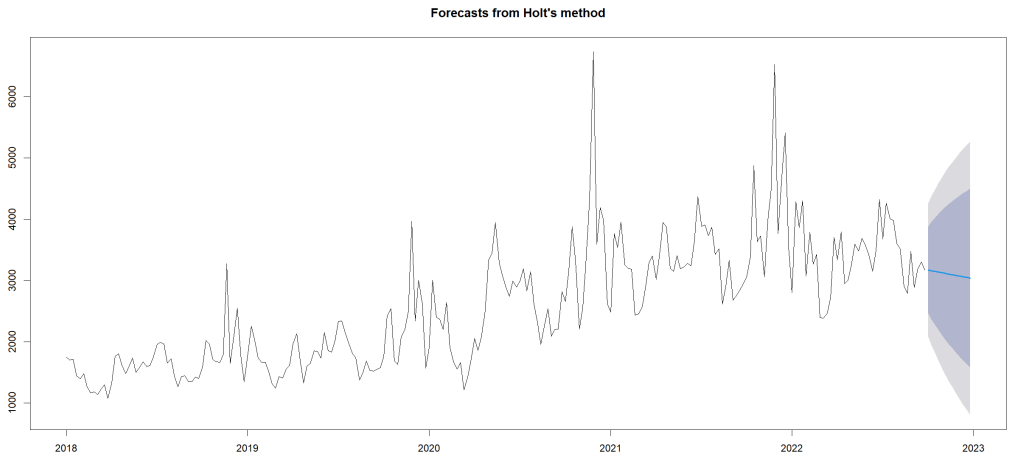


Figure 6.13: Forecasts from Holt method

Figure 6.13 shows the forecast on a graph, with clear spikes.

Model Assessment

We need to evaluate our model's accuracy using various error metrics. We will use the outcome from double exponential smoothing.

1. Retrieve model error metrics:

```
accuracy(forecast_holt)
                 ME     RMSE      MAE       MPE     MAPE     MASE      ACF1
Training set 32.6073 549.6696 363.1893 -1.031575 13.60411 0.609441
0.0649584
```

2. Retrieve test data set error metrics:

```
accuracy(forecast_holt, test$complaints)
                   ME      RMSE       MAE       MPE     MAPE      MASE
ACF1
Training set   32.6073  549.6696  363.1893 -1.031575 13.60411 0.9436677
0.0649584
Test set     1353.4809 1797.7280 1353.4809 26.548551 26.54855
3.5167231        NA
```

By calling `accuracy(forecast_holt)`, the function is designed to default to calculating metrics on the data that was used to train the model, treating the model's fitted values as the "forecast" and the original training data as the "actual" values. This is useful for comparing in-sample fit across models but is not a reliable way to assess the performance on new data not seen by the model. When providing the `test$complaints` to the function, it then measures "out-of-sample" test set accuracy. This will be more meaningful for the assessment of the model.

From the error metrics, there are several key metrics displayed:

- **ME**: Mean Error, which is the average of the forecast errors, the closer to 0, the better.
- **RMSE**: Root Mean Squared Error, this measures the average magnitude of the error by squaring difference, which penalizes larger errors more heavily. Lower values are better.
- **MAE**: Mean Absolute Error, which represents the average of absolute differences between actual and forecasted values.
- **MPE**: Mean Percentage Error, the average of the percentage errors, useful for understanding bias in percentage. Value close to 0 indicates low bias.
- **MAPE**: Mean Absolute Percentage Error, which measures the average percentage of the absolute percentage error. Close to 0 is a better value.
- **MASE**: Mean Absolute Scaled Error, which is a scale-independent measure that compares MAE of forecast against MAE of a simple naive forecast. Value less than 1 indicates that your forecast value is better than the naive forecast.

We have now calculated the metrics like MAE, RMSE, and MAPE to quantify the accuracy of our forecasts. These metrics help us understand how well our model performs. The error metric values shown here, specifically for the test set, indicate that the model does not perform well on the complaints forecast.

Handling daily data

In the previous example, `hw()` didn't handle the weekly complaints seasonality well. We will pick a daily data set (Bitcoin prices) and apply the same process to daily data.

1. Load the data file to data frame, then preprocess the daily data, converting the date column to a Date type and transforming the data frame into a `tsibble`:

```
df_daily <- read_csv("bitcoin_price.csv")

df_daily$Date <- as.Date(df_daily$Date)
df_daily <- df_daily %>% as_tsibble(index = Date)
```

2. We load and define the number of periods and set the max date:

```
periods <- 30

max_date <- max(df_daily$Date)
```

3. Prepare training and test data set:

```
train_daily <- df_daily %>%
  filter(Date < (max_date - days(periods)))

test_daily <- df_daily %>%
  filter(Date >= (max_date+1 - days(periods)))
```

4. View the first couple of rows of test data:

```
head(test_daily)
# A tibble: 6 × 7
  Date         Open   High    Low  Close `Adj Close`      Volume
  <date>      <dbl>  <dbl>  <dbl>  <dbl>       <dbl>       <dbl>

1 2023-12-01 37718. 38954. 37629. 38689.      38689. 23512784002
2 2023-12-02 38689. 39679. 38653. 39476.      39476. 15534035612
3 2023-12-03 39472. 40136. 39298. 39978.      39978. 15769696322
4 2023-12-04 39979. 42372. 39979. 41980.      41980. 39856129827
5 2023-12-05 41986. 44409. 41421. 44081.      44081. 36312154535
6 2023-12-06 44080. 44266. 43478. 43746.      43746. 29909761586
```

We split the daily data into training and test sets, with the last 30 days used for testing.

5. Convert close price column to a time series:

```
min_date <- min(train_daily$Date)
year_start <- year(min_date)
day_start <- yday(min_date)
ts_train_close <- ts(train_daily$Close,
                     # Weekly seasonality
                     frequency = 7,
                     start = c(year_start,day_start))
```

6. Fit the Holt-Winters model with multiplicative seasonality:

```
model_daily_triple <- hw(ts_train_close, seasonal = "multiplicative", h =
periods)
```

7. Forecast the future period:

```
forecast_daily_triple <- forecast(model_daily_triple, h = periods)
```

We fit a triple exponential smoothing model to the daily training data and forecast the next 30 days.

8. Plot the chart:

```
plot(forecast_daily_triple)
```

In this image, we plot the forecast obtained from applying a triple exponential smoothing model to the daily training data for the next 30 days. The graph shows a smooth forecast with prediction intervals based on the Holt-Winters multiplicative method, which is useful for capturing seasonality and trend patterns in the dataset.

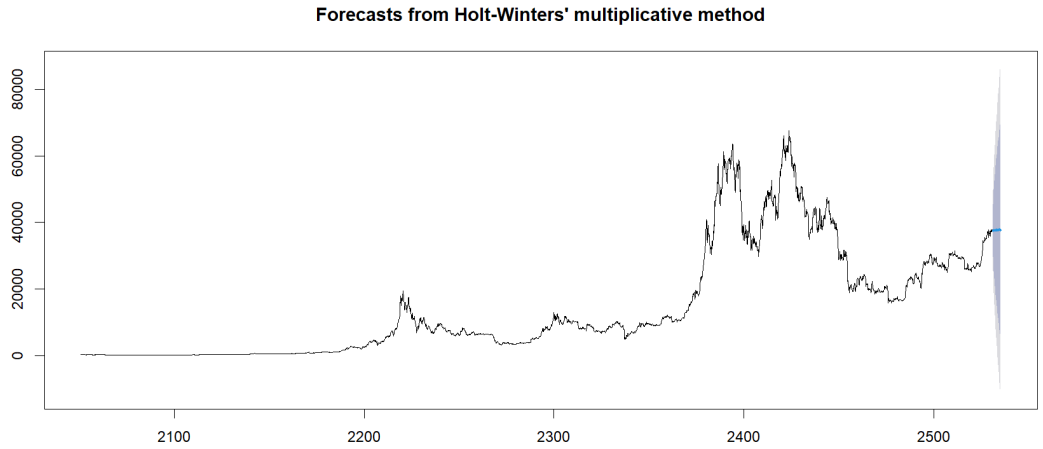


Figure 6.14: Triple Exponential Smoothing forecast for next 30 days

Let's summarize what we learned in this chapter.

Summary

In this chapter, we explored the concept of exponential smoothing, focusing on its three main variations: simple, double, and triple (also known as Holt-Winters). Using a real-world case study of customer complaints from a telecom company, we applied these smoothing techniques to help forecast fluctuations and manage operational challenges. Starting with a deep dive into the data, we visualized patterns and trends in customer complaints, setting the stage for accurate predictions. By understanding how complaints varied weekly, monthly, and quarterly, we identified critical patterns that would inform our forecasting models.

We began with simple exponential smoothing, which smooths data by giving more weight to recent observations. This method worked well for data without strong trends or seasonality. We then moved on to double exponential smoothing, which took into account both the level and the trend, making it more suited for data with a consistent upward or downward movement. Finally, we applied triple exponential smoothing, or Holt-Winters, which is ideal for data with both trends and seasonality. This method allowed us to model more complex patterns in the customer complaints, incorporating seasonal fluctuations that appeared at different times of the year.

After applying each method, we evaluated their performance using error metrics such as MAE, RMSE, and MAPE to measure the accuracy of the forecasts. We also discussed how these models could be adapted for daily data, providing a glimpse into more granular, daily forecasting. The chapter offered a hands-on experience with smoothing techniques, showing how they can be applied to real business challenges, and concluded by preparing us for the next chapter, where we will explore ARIMA models for time series forecasting.

Looking ahead to *Chapter 7*, we'll dive into the ARIMA family of models. We will also explore SARIMA and SARIMAX. Using a case study involving forecasting chocolate sales, we'll prepare data, visualize trends, and break down time series components.

7

The ARIMA Model Family

You are going to explore some powerful tools in this chapter. **ARIMA** stands for **Autoregressive Integrated Moving Average**. ARIMA has some interesting relatives: **SARIMA** and **SARIMAX**. Each one has its own special touch for handling different kinds of time series data. ARIMA is a classic model used in forecasting when data points are recorded over time. However, ARIMA has a limit: it does not really account for seasons or external factors that could affect your data.

SARIMA is ARIMA's seasonal option. It understands patterns that repeat over time, such as increased chocolate sales in the winter holiday season. SARIMAX goes even further. It brings in external factors, for example, a big event in town that boosts chocolate sales.

In this chapter, you will set up R for your time series analysis with ARIMA, dive into each model, and use some real-world data to see these models in action.

You will also learn how to choose the best model using AIC and BIC, which are scorecards for how well your model is doing. By the end, you will not only know how to predict the future (data-wise, of course) but also understand which model to pick for your specific scenario. You will also look at fine-tuning your models to get the best predictions possible.

You are going to explore the following topics:

- Introducing the case study
- Introducing ARIMA
- Exploring seasonality with SARIMA
- Going beyond seasonality with SARIMAX

Let's get started.

Technical requirements

R and RStudio are required for this chapter. GitHub scripts can be found in the repo.

You will need the following data file and packages for this chapter:

- `TSA`: The `TSA` (Time Series Analysis) package in R provides functions and data sets for time series analysis. It is an essential package for ARIMA modeling.
- `forecast`: The `forecast` package in R is a very popular set of tools designed for analyzing and forecasting univariate time series data. You will see some seasonal subseries plots using this package.
- `tseries`: The `tseries` package in R is for time series analysis and computational finance. Function like ADF test are from this package.
- `tidyverse`: The `tidyverse` package is a cohesive collection of R packages designed specifically for data science.
- `lubridate`: The `lubridate` package in R provides tools that make it easier to parse and manipulate dates.
- `daily_revenue.csv`: The daily data file you will use to demonstrate ARIMA, SARIMA, and SARIMAX models.

Introducing the case study

Imagine managing a busy chocolate retail shop where accurately forecasting daily revenue is essential for business success. Reliable sales forecasts help the shop optimize inventory, schedule staff efficiently, reduce product waste, and, more importantly, meet customer demand. This is especially important during seasonal events and holidays, such as Valentine's Day and Thanksgiving, when chocolate sales increase significantly.

In this case study, you will investigate the historical sales data and develop forecasting models based on them. You will explore trends, seasonal patterns, and other factors that influence sales, including weekends, holidays, and special events. Through this analysis, you will apply forecasting techniques to generate predictions that support appropriate business decisions.

More importantly, this case study demonstrates how data analytics creates tangible business value. Effective forecasting extends beyond building statistical models; it also requires understanding customers' purchasing behavior, identifying seasonal demand patterns, and balancing supply with anticipated demand. By the end of this case study, you will gain practical experience in transforming historical data into actionable business insights. You will

appreciate how accurate forecasting can contribute to improved operational performance and customer satisfaction.

First, you need to load the necessary libraries for data manipulation, visualization, and time series analysis:

1. Install and load the required packages. You will use the `tidyverse` for data manipulation, `ggplot2` for visualization, and `forecast`, `lubridate`, and `TSA` for time series analysis:

```
packages <- c("tidyverse", "ggplot2", "forecast", "lubridate", "TSA")

for (p in packages){
 if (!requireNamespace(p, quietly = TRUE)) {

    install.packages(p)
 }

 library(p, character.only = TRUE)
}
```

2. Next, you load the data from a comma-separated values (CSV) file and set the date as the index. This ensures that the date column is correctly parsed, allowing you to perform time series analysis:

```
df <- read_csv("daily_revenue.csv", col_types = cols(date =
col_date(format = "%d/%m/%Y")))
head(df)
```

This produces the following output:

```
# A tibble: 6 × 4
  date       revenue discount_rate coupon_rate
  <date>       <dbl> <chr>         <chr>
1 2018-01-01 6270839 34.27%        1.09%
2 2018-01-02 8922076 30.87%        1.08%
3 2018-01-03 8446101 28.11%        1.01%
4 2018-01-04 7785798 27.32%        0.96%
5 2018-01-05 6375303 25.70%        0.90%
6 2018-01-06 6095418 25.02%        0.87%
```

Figure 7.1: The first few rows of the dataset

You should see the first few rows of the dataset with the date column parsed correctly.

3. To understand the structure and types of each column, you use the `str` function. Get information about the data frame:

```
str(df)
```

The following figure displays the structure of the data frame, showing the types of each column and a preview of the data:

```
spc_tbl_ [1,795 × 4] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ date         : Date[1:1795], format: "2018-01-01" "2018-01-02" "2018-01-03" ...
 $ revenue      : num [1:1795] 6270839 8922076 8446101 7785798 6375303 ...
 $ discount_rate: chr [1:1795] "34.27%" "30.87%" "28.11%" "27.32%" ...
 $ coupon_rate  : chr [1:1795] "1.09%" "1.08%" "1.01%" "0.96%" ...
 - attr(*, "spec")=
  .. cols(
  ..   date = col_date(format = "%d/%m/%Y"),
  ..   revenue = col_number(),
  ..   discount_rate = col_character(),
  ..   coupon_rate = col_character()
  .. )
 - attr(*, "problems")=<externalptr>
```

Figure 7.2: Structure of the data frame showing column types and sample data

4. Set the frequency to daily. By setting the frequency of the time series to daily, you ensure that there are no gaps in the date sequence. The `complete()` function will ensure that every day between `min(date)` and `max(date)` exists in the data set. It will add a new row with NA if a date is missing, while the `fill()` function will fill NA values in the revenue column:

```
df <- df %>%
  complete(date = seq.Date(min(date), max(date), by = "day")) %>%
  fill(revenue)
```

5. To simplify your analysis, you rename the revenue column to `y`. This makes it easier to reference in subsequent steps:

```
df <- df %>%
  rename(y = revenue)
head(df)
```

You should see the updated data frame with the `y` column:

```
# A tibble: 6 × 4
  date              y discount_rate coupon_rate
  <date>        <dbl> <chr>         <chr>
1 2018-01-01 6270839 34.27%        1.09%
2 2018-01-02 8922076 30.87%        1.08%
3 2018-01-03 8446101 28.11%        1.01%
4 2018-01-04 7785798 27.32%        0.96%
5 2018-01-05 6375303 25.70%        0.90%
6 2018-01-06 6095418 25.02%        0.87%
```

Figure 7.3: Data frame showing renamed revenue column to y

6. Now, you plot the daily revenues to get a first look at the data. This visualization helps you understand the overall trend and identify any obvious patterns:

```
ggplot(df, aes(x = date, y = y)) +
  geom_line() +
  ggtitle("Daily Revenues") +
  xlab("Date") +
  ylab("Revenue")
```

You get the following plot:

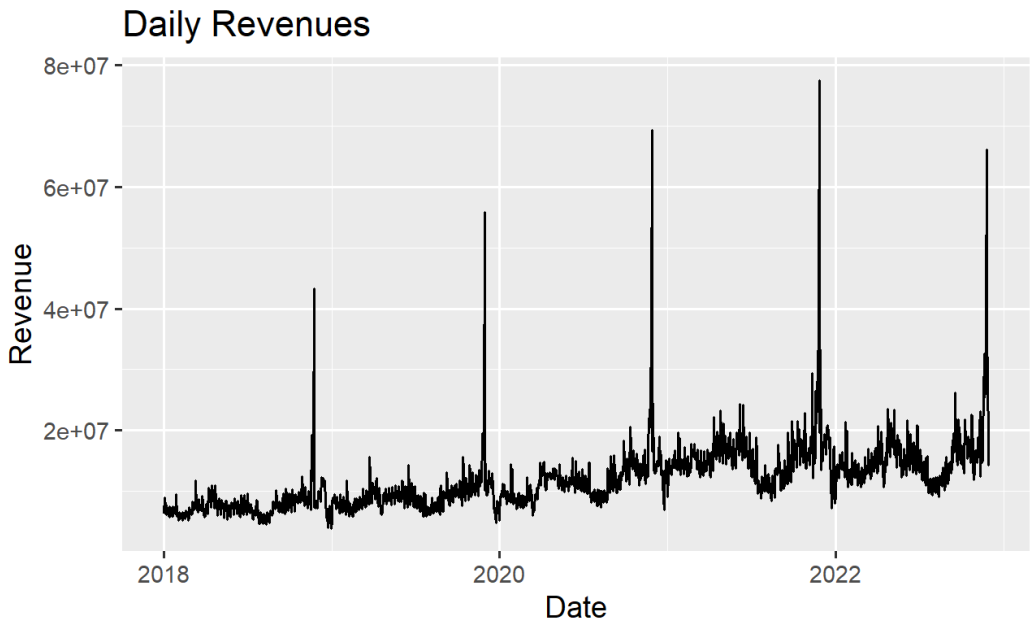


Figure 7.4: Line chart showing daily revenue trends over time

7. Plot the monthly seasonality. By plotting the average revenue per month, you can observe monthly seasonality. This highlights any seasonal patterns that occur regularly each month:

```
df %>%
  group_by(month = floor_date(date, "month")) %>%
  summarise(mean_revenue = mean(y, na.rm = TRUE)) %>%
  ggplot(aes(x = month, y = mean_revenue)) +
  geom_line() +
  ggtitle("Monthly Seasonality") +
  xlab("Month") +
  ylab("Revenue")
```

The monthly seasonality plot is shown here:

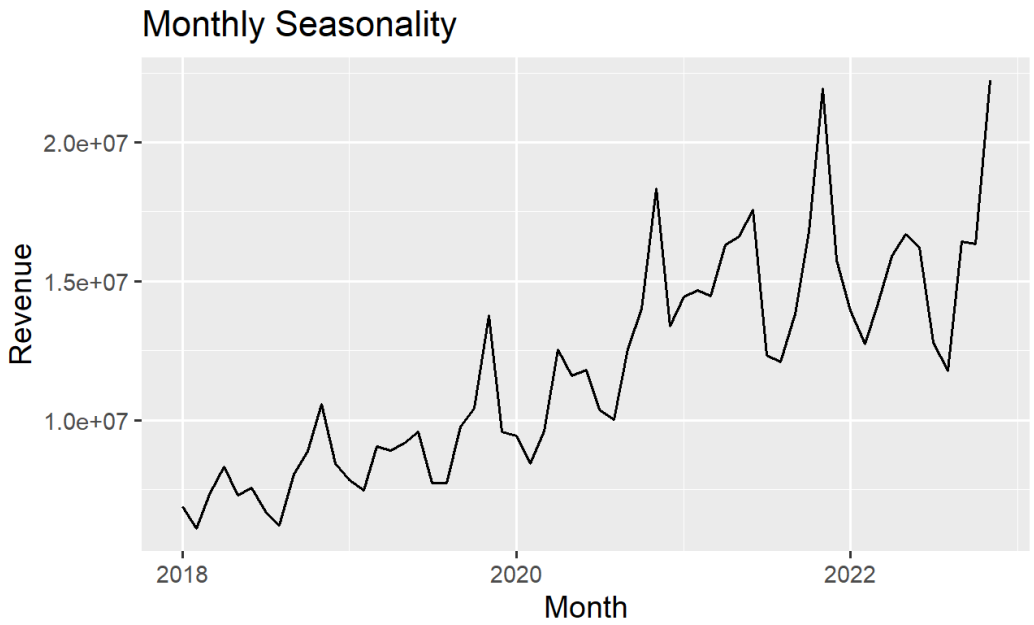


Figure 7.5: Line chart showing monthly revenue seasonality patterns

8. Similarly, plotting the average revenue per quarter reveals quarterly trends in the revenue data:

```
df %>%
  group_by(quarter = floor_date(date, "quarter")) %>%
  summarise(mean_revenue = mean(y, na.rm = TRUE)) %>%
  ggplot(aes(x = quarter, y = mean_revenue)) +
  geom_line() +
  ggtitle("Quarterly Seasonality") +
  xlab("Quarter") +
  ylab("Revenue")
```

You get the following plot:

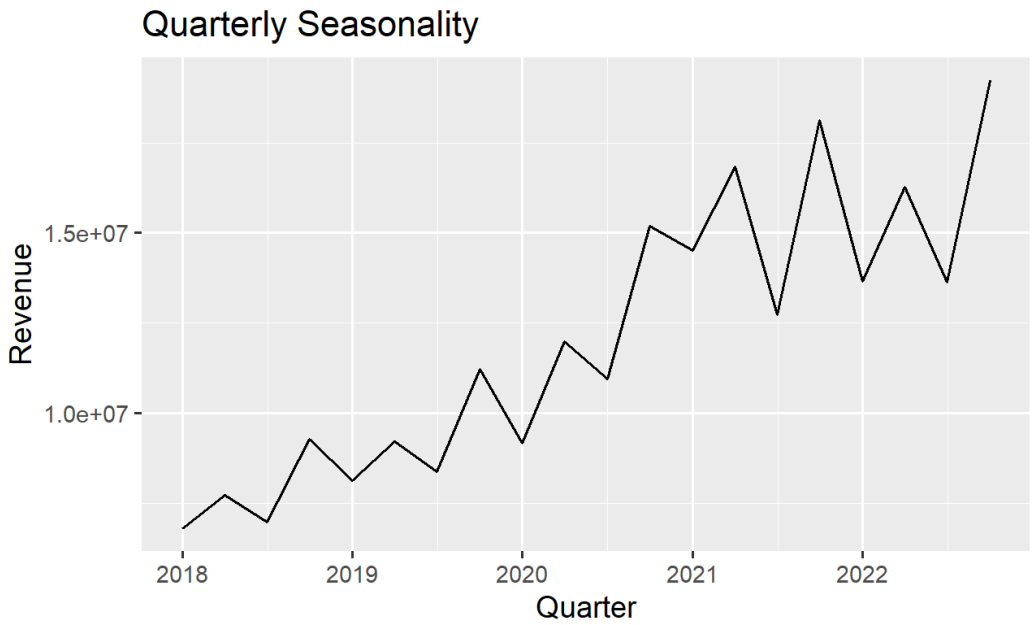


Figure 7.6: Line chart showing quarterly revenue seasonality patterns

9. Decomposing the time series breaks down the underlying components into trend, seasonal, and residual components:

```
decomposed <- decompose(ts(df$y, frequency = 365), type = "multiplicative")
autoplot(decomposed)
```

The decomposed plot is shown here:

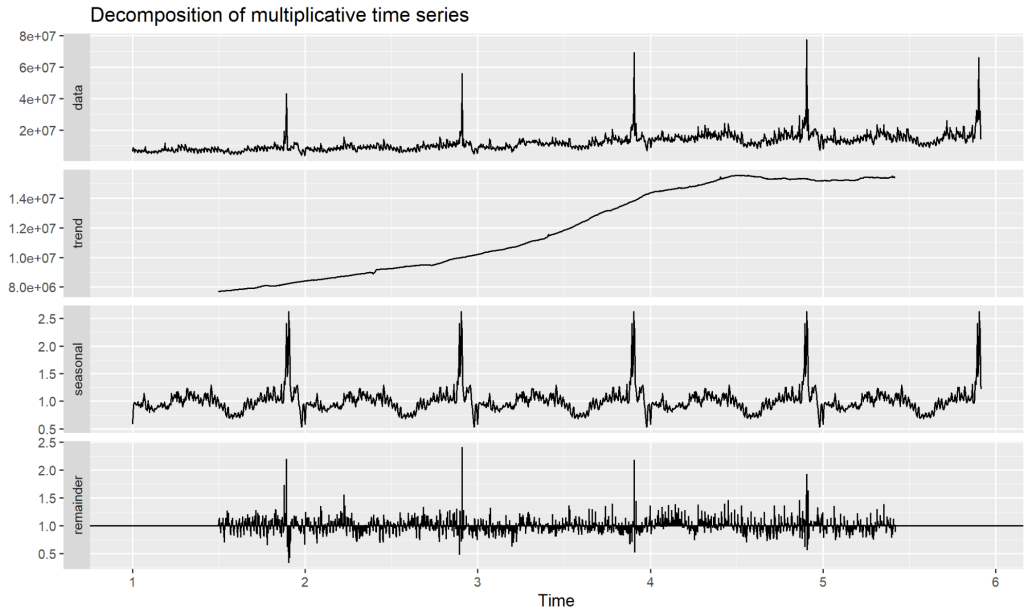


Figure 7.7: Decomposition of time series into trend, seasonal, and residual

The plot contains four parts:

- **Data:** This is the top chart, which displays the original time series data.
- **Trend:** This is the overall direction of the data. The revenue time series shows an upward trend, ignoring seasonality and noise.
- **Seasonal:** This is the repeating pattern that occurs within a specific time period. In this case, it is a one-year window.
- **Remainder:** This is the noise or irregular component not explained by trend or seasonality.

10. To identify patterns over time, you plot the **autocorrelation function (ACF)**, which shows the correlation of the time series with its past values:

```
acf(df$y, lag.max = 100, main = "Autocorrelation Function")
```

You get the following plot:

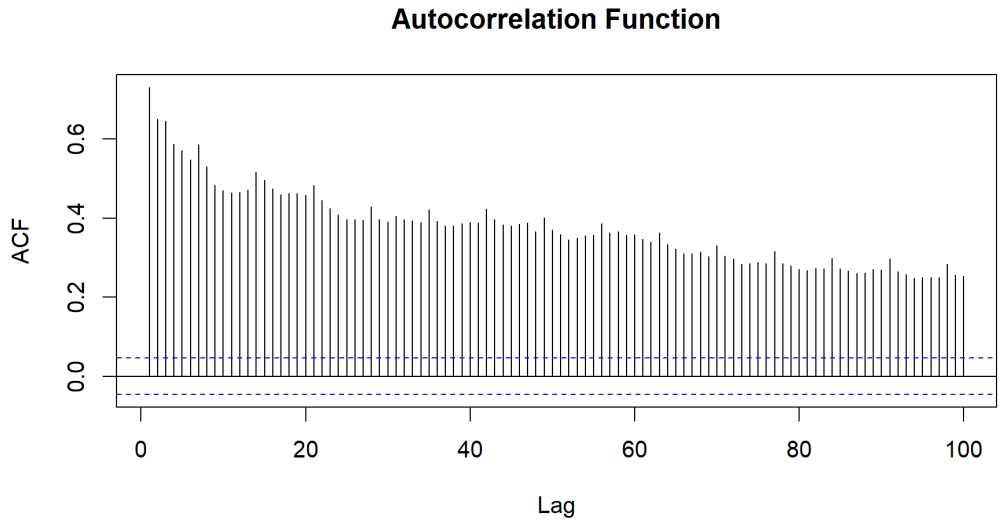


Figure 7.8: ACF plot showing time series correlation

Key components of the preceding ACF plot:

- **Lag:** On the x-axis, this represents the time difference (lag) between the observation and its past value. For example, lag 1 is the correlation with the immediately previous data point, lag 3 is two observations prior, and so on.
- **ACF**: On the y-axis, the height of each vertical bar indicates the correlation coefficient at that specific lag. The value ranges from –1 to +1.
- **Dashed lines:** These two lines represent the statistical significance threshold, in this case, the 95% confidence interval. Any bar that extends beyond the dashed lines is considered statistically significant (also referred as non-zero autocorrelation).

11. Finally, the **partial autocorrelation function (PACF)** measures the relationship between the current observation and a specific past observation (lag), after removing the influence of all shorter lags. This helps identify which previous observations have a direct effect on the current value:

```
pacf(df$y, lag.max = 100, main = "Partial Autocorrelation Function")
```

This outputs the following chart:

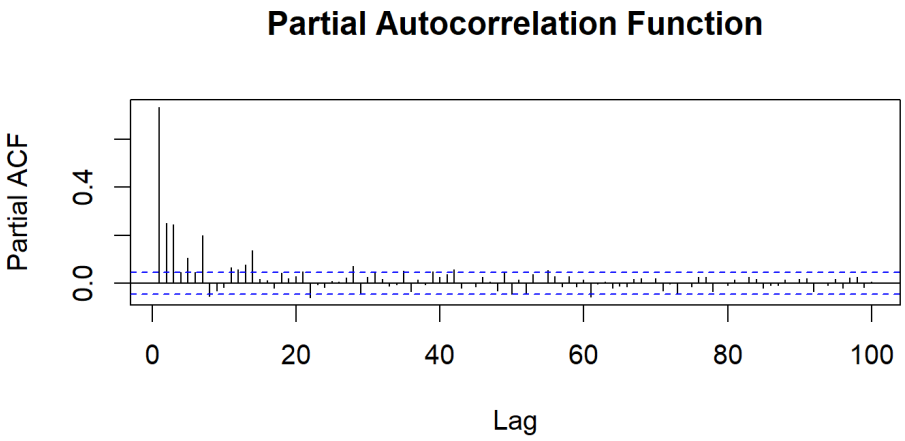


Figure 7.9: PACF plot showing lag correlations

The PACF plot also displays two blue dashed lines, which help detect any vertical spikes that cross the blue dashed lines (95% confidence interval). These spikes are statistically significant correlations. The PACF also will help to determine the number of **autoregressive (AR)** terms in the time series model by showing the direct correlation between an observation and a specific lag. The PACF plot in *Figure 7.9* showed a cutoff after lag 7 (this suggests an AR(7) model), however due to the insignificant lags at 4 and 6, you should also consider whether the data has seasonal patterns (at lag 7) or if an AR(3) is a more practical and simpler fit.

By following these steps, you perform an initial exploratory data analysis and time series decomposition, providing insights into the seasonal and autocorrelation patterns of the revenue data.

Introducing ARIMA

In this section, you are going to explore a really cool topic in forecasting. **ARIMA** stands for **Autoregressive Integrated Moving Average**.

It is a favorite in the forecasting world, often used across various departments, because it is straightforward to use, not too tough to understand, and accurate. However, ARIMA does not handle seasonality on its own, leading to the creation of **Seasonal ARIMA** (**SARIMA**). **Seasonal ARIMA with eXogenous variables** (**SARIMAX**) not only handles seasonality such as **SARIMA**, but also lets you include external factors, called exogenous regressors, in your forecasts.

ARIMA has three main components. The notation ARIMA(p, d, q) defines the core components of the model, where each letter represents a specific parameter used to capture trends and remove noise from the data:

- **Autoregressive (AR) - p**: This part is like looking at the past to predict the future. You use previous values of your time series to forecast what is coming next. In the ARIMA notation, this is represented by the **p** in ARIMA(p,d,q).
- **Integrated (I) - d:** This component uses differencing of raw observations (subtracting data points) to make the time series stationary. This is represented by the **d**.
- **Moving Average (MA) - q**: Models the dependency between an observation and residual errors from a moving average model. It is represented by the **q**.

ARIMA(p, d, q) is typically represented by this equation:

$$Y_t = \alpha + \sum_{i=1}^{p} \phi_i Y_{t-i} + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j} + \varepsilon_t$$

Here is what it means:

- Y_t: This is what you are trying to forecast, the value of your time series at time *t*.
- α: A constant term, kind of like the starting point.
- ϕ_i: These are the coefficients for the autoregressive (AR) part. They show how much previous values (Y_{t-i}) influence the current value.
- θ_j: These are the coefficients for the moving average (MA) part. They measure the impact of past errors (ε_{t-j}) on the current value.
- ε_t: The error term for the current time. It is the part of Y_t that the model could not predict.

That is ARIMA in a nutshell.

Autoregressive (AR)

The autoregressive (AR) component of an ARIMA model is based on the principle that past observations influence future values. Consider this chapter's use case: you want to predict the chocolate shop's revenue tomorrow. A logical approach is to examine the shop's revenue over the previous few days, as recent revenue data often provide useful information about future behavior. This is the fundamental idea behind the AR component. It uses historical observations to predict future values. To accomplish this, the AR component uses *lags*, which are simply previous observations in a time series. A lag represents a step backward in time. For example, a *lag of 1* in your chocolate use case, you are looking at yesterday's revenue to predict

today's, while a *lag of 2* refers to the day-before-yesterday's revenue influencing today's number.

Let's dive a bit deeper. The AR component is represented by the parameter p in the ARIMA(p, d, q) notation. The value of p specifies the number of lagged observations of the series included in the model. For example, an ARIMA(2, d, q) model uses the previous two days' observations on revenue to predict the current value. This relationship can be expressed as:

- Today's chocolate revenue=α+ ϕ_1 $*$ Yesterday's revenue + ϕ_2 $*$ Day Before Yesterday's revenue + ...
- Here, α is a constant (like a baseline amount of the chocolate shop's revenue), and ϕ_1, ϕ_2, ... are coefficients. These are numbers that tell you how much each previous day's chocolate revenue is influencing (correlated to) today's. Larger coefficient values indicate a stronger relationship between past and present observations.

The AR part allows your model to capture a really important aspect of data, namely, that what happened in the past has an impact on what is happening now. This is especially powerful in time series in which patterns and trends over time play a huge role, such as in stock prices, weather forecasting, sales figures, consumer behavior, and, in your use case, the chocolate shop's revenue predictions!

Understanding the AR component provides a strong foundation for time series forecasting. By systematically incorporating information from past observations, the model can identify recurring patterns and use them to generate more accurate forecasts for future values.

Integrated (I)

The **integrated (I)** component of an ARIMA model is responsible for transforming a non-stationary time series into a stationary one. **Stationarity** is a foundational assumption of many time series forecasting methods because it implies that the statistical properties of the series, such as its mean, variance, and autocovariance, all remain constant over time. When these properties are stable, the underlying patterns are more consistent, enabling the model to produce more reliable forecasts.

In practice, many real-world time series are **non-stationary**. They may exhibit long-term trends, seasonal effects, or changing levels of variability. Examples include stock prices, cryptocurrency values, sales revenue, and economic indicators, all of which often display evolving trends over time. Since the statistical characteristics of these series change, directly applying forecasting models can result in inaccurate predictions.

To address this issue, ARIMA employs a process known as **differencing**. Differencing involves subtracting each observation from its preceding observation, thereby creating a new series that represents the changes between consecutive data points rather than the original value.

The integrated component is represented by the parameter d in ARIMA(p, d, q). The value of d indicates the number of times the time series must be differenced before it becomes stationary. For example, if a single round of differencing is sufficient, then d = 1, resulting in an ARIMA(p, 1, q) model. If two rounds of differencing are required, then d = 2. In practice, the objective is to apply the minimum amount of differencing necessary to achieve stationarity, as excessive differencing may remove meaningful information and reduce the model's forecasting performance.

By converting a non-stationary series into a stationary one, the integrated component provides a stable foundation for the AR and MA components to model temporal relationships effectively. As a result, the ARIMA model can better capture the underlying data and generate more accurate forecasts.

Before you move into the details on the MA components, let's take a deeper dive on stationarity and how to assess whether your data is stationary in the following sections.

Stationarity and non-stationary visualization

Let's look at the three different graphs shown as follows:

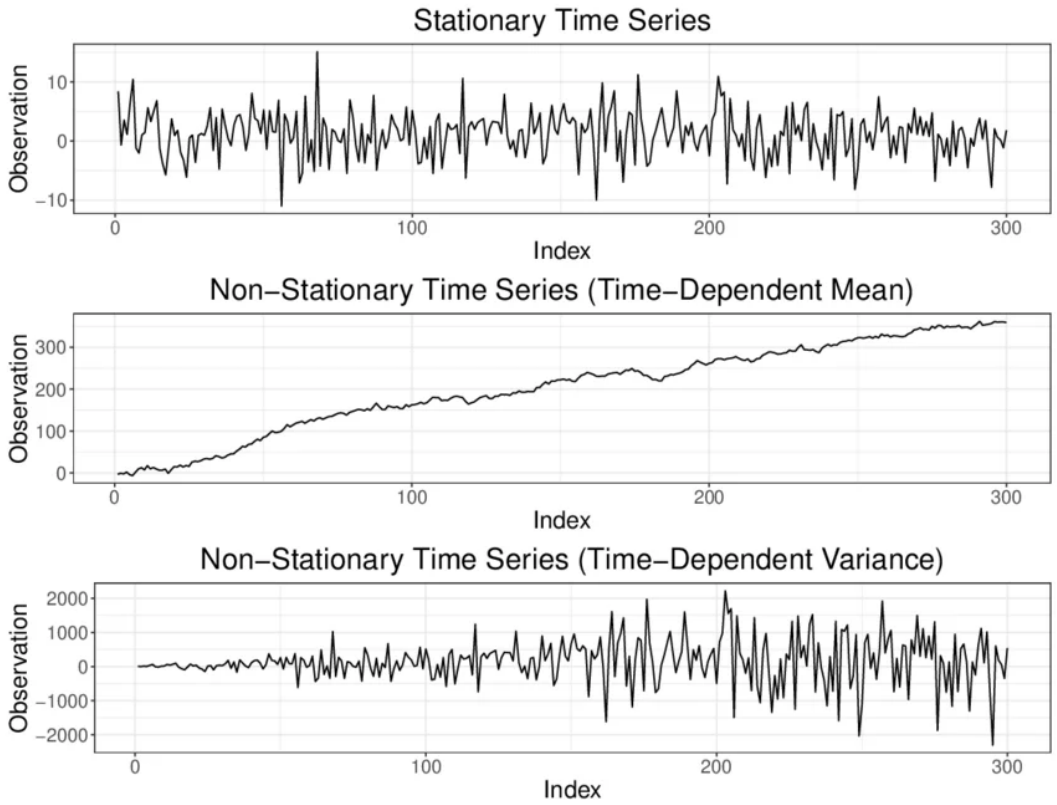


Figure 7.10: Visualizing stationary and non-stationary time series (Source: `https://www.researchgate.net/figure/Examples-for-stationary-and-non-stationary-time-series_fig3_348592737`*)*

Let's analyze each of the three graphs in the previous image:

- The first one is steady. It has a stable pattern over time. This one is stationary (the mean does not change over time).
- The second graph grows over time. This one is not stationary because its mean is changing.
- The third has varying amplitudes. Think of seasonal sales, peaking in December and dipping in January. Its variance changes, so it is not stationary.

The graphs display the difference between stationary and non-stationary time series; obviously, in stationary data, the statistical properties do not change over time. You may wonder: how can you test whether the time series data is stationary?

ADF test

Before fitting an ARIMA model, it is important to determine whether the time series is stationary. While visual inspection of a time series plot can provide useful insights, a statistical test offers a more objective assessment.

The **Augmented Dickey-Fuller (ADF)** test is one of the most widely used methods for testing stationarity. It determines whether a time series contains a **unit root**, which is a characteristic of non-stationary data. A unit root indicates that shocks or changes to the series have a persistent effect, causing the mean and other statistical properties to change over time. In contrast, a stationary series tends to fluctuate around a constant mean and returns to its long-term equilibrium after temporary disturbances. The ADF test extends the original Dickey-Fuller test by incorporating lagged differences of the series, allowing it to account for autocorrelation and produce more reliable results.

The ADF test evaluates the following hypotheses:

- **Null hypothesis (H_0)**: The time series contains a unit root and is *non-stationary*
- **Alternative hypothesis (H_1)**: The time series is *stationary*

The test returns a test statistic and a **p-value**. If the p-value is less than 0.05, the null hypothesis is rejected, indicating that the series is stationary. If the p-value is greater than 0.05, the series is considered non-stationary and typically requires differencing.

In practice, many real-world time series exhibit trends or changing statistical properties, making them non-stationary. As a result, the ADF test is commonly used to determine whether differencing is necessary before fitting an ARIMA model. Once the series becomes stationary, the ARIMA model can more effectively identify temporal patterns and produce reliable forecasts.

Let's have some code examples for the ADF test:

1. Load the `tseries` package:

```
library(tseries)
```

2. Perform the Augmented Dickey-Fuller (ADF) test to check the stationarity of your time series data. Use the chocolate shop revenue `df$y` variable as your data:

```
adf_test<- adf.test(df$y, alternative="stationary")

# Print p-value
```

```
print(paste("p-value:", adf_test$p.value))
[1] "p-value: 0.01"
```

This code performs the ADF test on the time series data in `df$y`, with the null hypothesis that the time series is non-stationary. The p-value helps determine whether the time series is stationary. A `p-value` less than 0.05 suggests that the time series is stationary.

3. Next, interpret the results of the ADF test. If the `p-value` is less than 0.05, the time series is accepted as stationary. Otherwise, it is not:

```
if (adf_test$p.value < 0.05) {
  print("Evidence suggests that the time series is stationary.")
} else {
  print("Evidence suggests that the time series is not stationary.")
}
[1] "Evidence suggests that the time series is stationary."
```

4. To handle non-stationarity (which is not the case), we can difference the series. Differencing helps remove trends and seasonality, making the time series stationary:

```
df <- df %>% mutate(y_diff = c(NA, diff(y)))
head(df)
```

This code creates a new column `y_diff` that contains the differenced series:

	date	y	discount_rate	coupon_rate	y_diff
	<date>	<dbl>	<chr>	<chr>	<dbl>
1	2018-01-01	6270839	34.27%	1.09%	NA
2	2018-01-02	8922076	30.87%	1.08%	2651237
3	2018-01-03	8446101	28.11%	1.01%	-475975
4	2018-01-04	7785798	27.32%	0.96%	-660303
5	2018-01-05	6375303	25.70%	0.90%	-1410495
6	2018-01-06	6095418	25.02%	0.87%	-279885

Figure 7.11: Difference the time series data and create a new column

5. Next, plot the original and differenced series to visualize the effect of differencing. First, plot the original data:

```
plot(df$date, df$y, type = "l", col = "blue",lwd = 2, main = "Original and
Differenced Time Series", xlab = "Date", ylab = "Revenue")
```

6. Then, overlay the differenced series:

```
par(new = TRUE)
plot(df$date[-1],  df$y_diff[-1],  type = "l",  col = "orange",  lwd = 2,
axes = FALSE,  xlab = "",  ylab = "")
```

7. Add the right y-axis:

```
axis(side = 4, col.axis = "orange", col = "orange")
```

8. Add the right y-axis label:

```
mtext("Differenced Revenue", side = 4,  line = 3,  col = "orange")
```

9. Add the legend:

```
legend("topright",  legend = c("Original Series", "Differenced Series"),
col = c("blue", "orange"),  lty = 1,  lwd = 2)
```

These code steps plot the original series and the differenced one on the same graph. These plots allow you to compare them visually, as shown in *Figure 7.12*. The differenced series should have reduced trends and seasonality.

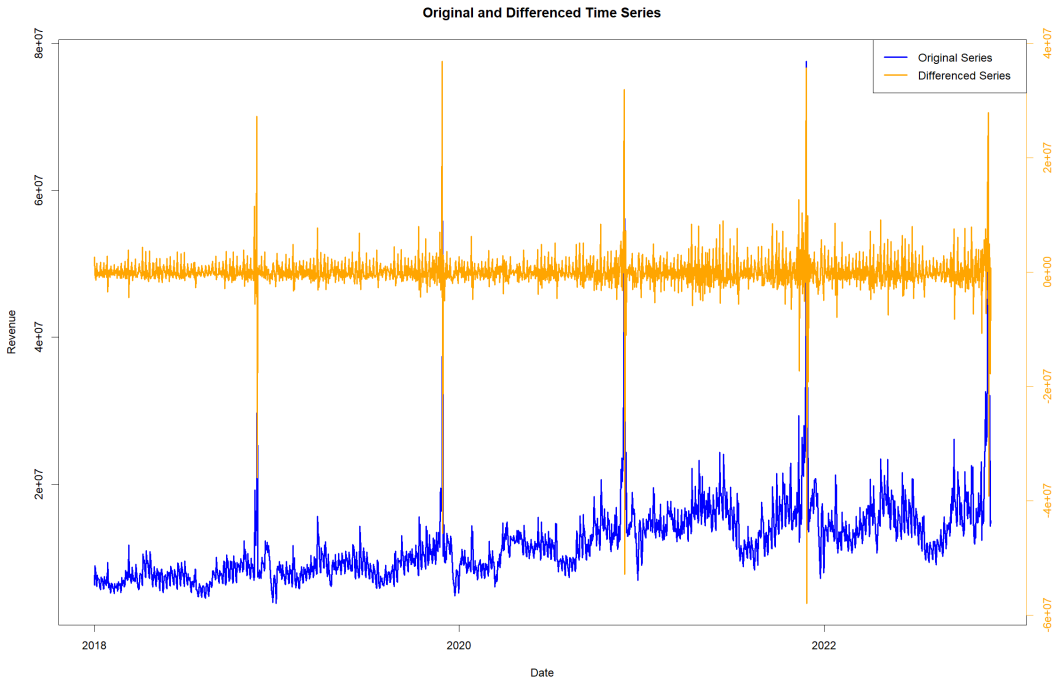


Figure 7.12: Plot of original and differenced time series

10. Verify the ADF test on the differenced series:

```
adf_test_diff <- adf.test(df$y_diff[-1], alternative = "stationary")

print(paste("p-value:", adf_test_diff$p.value))
[1] "p-value: 0.01"
```

By following these steps, you perform an ADF test to check for stationarity, difference the time series to achieve stationarity if necessary, and then verify the stationarity of the differenced series. This process is crucial for preparing the time series data for further analysis and modeling.

Moving average (MA)

The **Moving Average (MA)** component is the final building block of the ARIMA model. Despite its name, the **MA** component does not compute the average of past observations. Instead, it models the relationship between the current observation and the **forecast errors** (also known as **residuals**) from previous time periods.

To illustrate this concept, consider your chocolate shop revenue use case. Suppose you predicted that you would have $6 million in revenue yesterday, but you actually had only $4 million. The difference between the predicted value and the observed value is the forecast error. Rather than disregarding this error, the MA component uses it to improve future forecasts. If previous predictions consistently overestimated or underestimated the actual values, the model incorporates these errors to make more accurate predictions.

The MA component is represented by the parameter q in the ARIMA(p, d, q) notation. The value of q specifies the number of past forecast errors included in the model. For example, an ARIMA(p, d, 2) model uses the forecast errors from the previous two time periods when generating the current forecast.

By incorporating information from past forecasting errors, the MA component captures short-term patterns that may not be explained by the autoregressive component alone. This enables the model to continuously adjust its forecasts based on recent prediction performance, resulting in improved forecasting accuracy.

With all the components in ARIMA explained, let's look at the chocolate shop use case.

ARIMA in R

In this section, you will walk through the process of building an ARIMA model using R. By following these steps, you will learn how to prepare your data, configure the ARIMA model, and evaluate its performance to make accurate predictions:

1. First, split the data into training and test sets. This step is crucial to evaluate the model's performance on unseen data:

   ```
   test_days <- 30
   train <- head(df$y, -test_days)
   test <- tail(df$y, test_days)
   ```

   The preceding code uses `head(df$y, -30)`, which returns all parts of `df$y` except for the last 30 elements. This is the training data. It uses `tail(df$y, 30)` to set the last 30 data points as the test set.

2. Find the best ARIMA model. Use the `auto.arima` function from the forecast package to find the optimal ARIMA model (in this case, `seasonal= FALSEmeans` that the model fits a non-seasonal ARIMA model) for your training data. This function automatically selects the best parameters for the ARIMA model:

   ```
   model <- auto.arima(train, seasonal = FALSE)
   summary(model)
   ```

The `auto.arima` function fits an ARIMA model to the training data and selects the best parameters based on AIC. The `summary` function provides detailed information about the model.

```
Series: train
ARIMA(5,1,4)

Coefficients:
          ar1      ar2      ar3     ar4     ar5      ma1     ma2      ma3
      -0.4932  -0.4999  -0.2130  0.3096  0.0780  -0.0822  0.0261  -0.0754
s.e.   0.1166   0.0817   0.0917  0.0507  0.0364   0.1136  0.0995   0.0960
          ma4
      -0.6235
s.e.   0.0629

sigma^2 = 8.986e+12:  log likelihood = -28806.27
AIC=57632.55   AICc=57632.67   BIC=57687.3

Training set error measures:
                    ME    RMSE     MAE       MPE     MAPE      MASE        ACF1
Training set 36416.2 2989200 1460878 -2.501121 11.93404 0.9314165 0.00193239
```

Figure 7.13: ARIMA model output showing coefficients, diagnostics, and error measures

This output shows the results from fitting an ARIMA(5, 1, 4) model to a time series dataset. The coefficients for the AR terms (AR1 to AR5) and MA terms (MA1 to MA4) are provided along with their **standard errors** (**SE**). These coefficients quantify the relationship between the time series' current value and its past values (AR terms) and past errors (MA terms). Additionally, model diagnostics such as log likelihood, **Akaike Information Criterion (AIC)**, **corrected AIC** (**AICc**), and **Bayesian Information Criterion** (**BIC**) (You will see more coverage on **AIC** and **BIC** in the next section) are provided to assess model fit, while training set error measures (like RMSE and MAE) reflect the prediction accuracy on the training data.

Based on the training data set, the ARIMA(5,1,4) model is the optimal model from `auto.arima()`:

- **p = 5 (AR)**: The model uses 5 lagged observations to predict the current value.
- **d = 1 (I)**: The data is differenced once to make the time series stationary.
- **q = 4 (MA): The** model uses four lagged forecast.

3. You then generate predictions for the test period using the fitted ARIMA model:

```
predictions_arima <- forecast(model, h = test_days)$mean
predictions_arima
```

The `forecast` function produces predictions for the specified number of periods (`h`), in this case, the length of the test set. The `$mean` component extracts the predicted values:

```
Time Series:
Start = 1766
End = 1795
Frequency = 1
 [1] 14708826 14929106 15217369 15554565 15676590 15612101 15617504 15747942
 [9] 15758705 15676593 15680576 15758165 15748895 15689254 15701610 15751632
[17] 15736666 15697226 15712681 15744409 15728702 15703921 15718947 15738295
[25] 15724130 15709349 15722318 15733488 15721768 15713523
```

Figure 7.14: ARIMA forecasted values layout

Note the start and end with the wrong years because you did not set it as time series object. It's not an issue, though, since it does not affect the modeling.

4. Then, you create a function to assess the model and visualize the output. This function will plot the training data, test data, and predictions, as well as calculate the **Mean Absolute Error (MAE)**, **Root Mean Squared Error (RMSE)**, and **Mean Absolute Percentage Error (MAPE)**:

```
model_assessment <- function(train, test, predictions, chart_title)
{
```

5. Combine all data points (train, test, and predictions) with a sequence index:

```
  combined_length <- length(train) + length(test) + length(predictions)
```

6. Generate the index for the Date column based on the lengths of the train, test, and predictions:

```
  index <- seq(1, combined_length)
```

7. Create a data frame for plotting:

```
  plot_df <- data.frame(
    Date = c(index[1:length(train)], index[(length(train)
+1):(length(train) + length(test))],
             index[(length(train) + length(test) + 1):combined_length]),
    Value = c(as.numeric(train), as.numeric(test),
as.numeric(predictions)),
    Type = rep(c("Train", "Test", "Forecast"), c(length(train),
```

```
length(test), length(predictions)))
  )
```

8. Plot the data using `ggplot`:

```
  library(ggplot2)
  p <- ggplot(plot_df, aes(x = Date, y = Value, color = Type)) +
    geom_line() +
    labs(title = chart_title, x = "Index", y = "Value") +
    theme_minimal()

  print(p)  # Force the plot to display
```

9. Calculate the MAE, RMSE, and MAPE:

```
  mae <- mean(abs(test - predictions))
  rmse <- sqrt(mean((test - predictions)^2))
  mape <- mean(abs((test - predictions) / test)) * 100

  cat(sprintf("The MAE is %.2f\n", mae))
  cat(sprintf("The RMSE is %.2f\n", rmse))
  cat(sprintf("The MAPE is %.2f %%\n", mape))
}
```

This function takes the training and test datasets, along with the predictions and the chart title, to produce visual and numerical assessments of the model's performance.

10. Finally, you assess the model using the previously defined `model_assessment` function. This step involves visualizing the train, test, and forecast values and calculating error metrics:

```
dev.off()  # Reset the graphics device
plot.new()  # Open a new graphics device

model_assessment(train, test, predictions_arima, "ARIMA")

The MAE is 7269502.07
The RMSE is 12376399.66
The MAPE is 24.15 %
```

The ARIMA model's performance shows an average error of about 7.27 million units (MAE), with larger errors reflected in the RMSE of 12.38 million, suggesting some significant prediction deviations. The MAPE of 24.15% indicates that, on average, the model's predictions deviate by about 24% from the actual values.

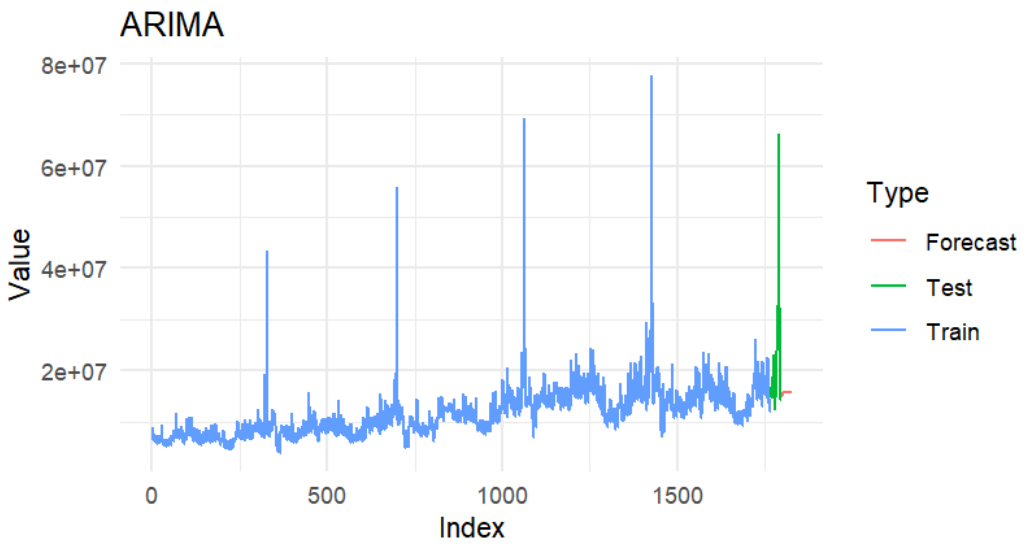


Figure 7.15: Plot for ARIMA training, test, and forecast data

In the plot, you can see that the training data has some sharp spikes, while the test data and forecasts are relatively small in comparison, which might indicate the model is struggling with some extreme values in the data.

By following these steps, you split the data, fit an ARIMA model, make predictions, and assess the model's performance. This process ensures that you evaluate the model on both training and unseen test data, providing a comprehensive understanding of its accuracy and reliability.

AIC and BIC

Let's dive into two really important concepts in time series analysis that was mentioned earlier in the chapter: **Akaike Information Criterion** (**AIC**) and **Bayesian Information Criterion** (**BIC**). These tools help you pick the best model for your data, scoring each model based on how well it performs while keeping it as simple as possible.

They take into account two things: how well the model fits the data (goodness of fit) and the complexity of the model (number of parameters used):

- AIC wants a model that fits well but does not go overboard with too many parameters
- BIC is similar to AIC but gives a harsher penalty for models with more parameters

When you use `auto.arima` , it automatically tests out a bunch of different ARIMA models for your data. For each model, it calculates the AIC and BIC scores. The model with the lowest AIC or BIC is usually considered the optimal choice.

Here are some pros for using this approach:

- **Objective comparison**: AIC and BIC provide a straightforward way to compare different models, making the selection process more objective and data-driven.
- **Penalty for complexity**: They discourage overfitting by penalizing models that are too complex. AIC penalizes based on the number of parameters, while BIC applies a stronger penalty based on parameters and sample size, favoring simpler models.
- **Flexibility**: AIC and BIC can be used with a wide range of models, not just ARIMA, making them versatile tools in the modeler's toolkit.

And some cons:

- **Relative measures**: They do not give an absolute "good" or "bad" score. A model's AIC or BIC is only meaningful when compared to other models' scores.
- **Business focus**: A company would mostly focus on the error (like the MAE, or MAPE) and not really care about simplicity or goodness of fit.
- **Information loss**: In penalizing complexity, there is a chance of dismissing models that capture important nuances in the data.

While they are not perfect and should be used alongside other considerations (like domain knowledge and data understanding), AIC and BIC provide a solid starting point in model selection.

Later on, you will use error metrics, such as the MAE, for model selection.

Exploring seasonality with SARIMA

SARIMA stands for **Seasonal Autoregressive Integrated Moving Average**. SARIMA extends ARIMA by adding a seasonal component to the mix. Most real-world scenarios have seasonal patterns. In retail, companies often face fluctuating sales patterns throughout the year, particularly during holiday seasons. For example, a retailer might use SARIMA to predict December sales by analyzing data from past Decembers. This helps them prepare for the holiday rush, ensuring they have enough stock and staff to meet demand, thus avoiding understocking or overstocking.

Another common use case is in weather forecasting. Meteorologists rely on SARIMA to predict weather conditions by leveraging historical seasonal data. For instance, forecasting this winter's snowfall by analyzing snowfall patterns from previous winters allows for better planning in infrastructure, resource allocation, and emergency responses.

SARIMA is usually noted as SARIMA$(p, d, q)(P, D, Q)_m$:

- p, d, and q are the non-seasonal components (just like in ARIMA).
- P, D, and Q are the seasonal components. Notice that they use the same letters in uppercase because they represent the same types of model components, but operate at different time scales:
    - P is the seasonal autoregressive order
    - D is the seasonal differencing order
    - Q is the seasonal moving average order
- m is the number of periods in each season

Let's learn more about how SARIMA works:

- **Seasonal differencing order (D)**: Just as you use differencing in ARIMA to stabilize the data, SARIMA applies seasonal differencing to iron out those seasonal patterns.
- **Seasonal AR order (P)**: This is about looking back at previous seasons (season is a repeating pattern or cycle in time series data that occurs at a consistent interval and could be daily, monthly, or quarterly fluctuations). Say you're forecasting this December's sales. SARIMA will use data from last December to help make that prediction.
- **Seasonal MA order (Q)**: Similar to the MA part in ARIMA, but here, it's focused on the errors made in past seasonal periods.

Using SARIMA can be a balancing act—you've got to find the right combination of seasonal and non-seasonal parameters. By recognizing these seasonal patterns, SARIMA can guide decisions in industries where cyclical trends are prevalent, ensuring smoother operations and more accurate forecasts.

SARIMA in R

First, you use the `auto.arima` function from the `forecast` package to find the best SARIMA model for your training data. This function automatically selects the best parameters for the SARIMA model, considering seasonality with a specified period.

1. Create the time series object by converting the training data to a time series with weekly seasonality:

    ```
    train_ts <- ts(train, frequency = 7)
    ```

    The code creates a time series from the training data with a seasonal frequency of 7 (daily data with a weekly cycle). This will model a weekly pattern in the data.

2. Fit the SARIMA model using `auto.arima`:

```
model_sarima <- auto.arima(train_ts, seasonal = TRUE)
summary(model_sarima)
```

The `auto.arima` function fits a SARIMA model to the training data and selects the best parameters based on AIC, considering weekly seasonality (`period = 7`). The `summary` function provides detailed information about the model.

```
Series: train_ts
ARIMA(2,1,4)(0,0,2)[7]

Coefficients:
         ar1     ar2      ma1      ma2     ma3      ma4    sma1    sma2
      0.3088  0.3447  -0.8907  -0.3533  0.3325  -0.0756  0.1712  0.0938
s.e.  0.1470  0.1425   0.1475   0.2184  0.0608   0.0437  0.0278  0.0227

sigma^2 = 8.76e+12:  log likelihood = -28784.74
AIC=57587.48   AICc=57587.58   BIC=57636.75

Training set error measures:
                    ME    RMSE     MAE       MPE     MAPE      MASE          ACF1
Training set 98118.49 2952245 1395756 -2.010136 11.29394 0.7697489 -0.001823686
```

Figure 7.16: Summary of the optimal SARIMA model

The `ARIMA(2,1,4)(0,0,2)[7]` model was fitted to the `train_ts` dataset, capturing both AR and MA components, with seasonality of 7 periods. The model's performance metrics show an RMSE of 2.95 million and a MAPE of 11.29%, indicating that the average prediction error is about 11.29% of the actual values.

3. You then generate predictions for the test period using the fitted SARIMA model:

```
predictions_sarima <- forecast(model_sarima, h = test_days)$mean
predictions_sarima[1:5]
[1] 14547172 13839977 13697350 13734062 13871322
```

The `forecast` function produces predictions for the specified number of periods (`h`), which in this case is the length of the test set. The `$mean` extracts the predicted values.

4. Finally, you assess the model using the previously defined `model_assessment` function. This involves visualizing the train, test, and forecast values and calculating error metrics:

```
model_assessment(train, test, predictions_sarima, "SARIMA")
The MAE is 7694807.38
```

```
The RMSE is 12761064.83
The MAPE is 25.79 %
```

The SARIMA model's performance is indicated by a Mean Absolute Error (MAE) of 7.69 million, a Root Mean Squared Error (RMSE) of 12.76 million, and a Mean Absolute Percentage Error (MAPE) of 25.79%, meaning the model's predictions are off by an average of 25.79% compared to the actual values.

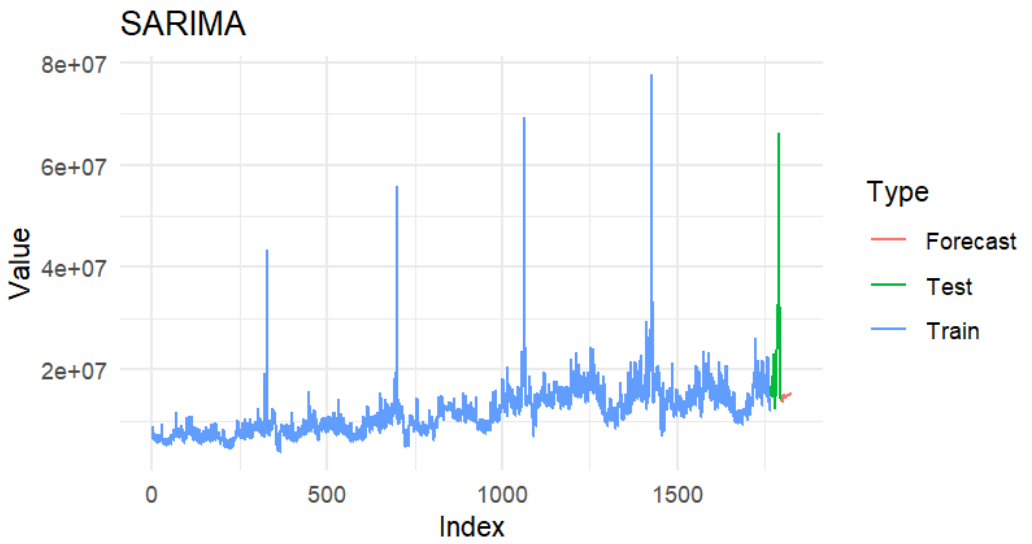


Figure 7.17: SARIMA model plot showing train, test, and forecast data

In the plot, the training data (blue) shows some significant spikes, while the test data (green) and forecast (red) portions align closely at the end, indicating that the model captures the general pattern but struggles with some volatility in the data.

To sum it up, this is how you split the data, fit a SARIMA model, make predictions, and assess the model's performance. This process ensures that you evaluate the model on both training and unseen test data, providing a comprehensive understanding of its accuracy and reliability.

Going beyond seasonality with SARIMAX

SARIMAX stands for **Seasonal Autoregressive Integrated Moving Average** with **eXogenous** variables. Essentially, SARIMAX extends the SARIMA model by including external factors. These are the *X* or *exogenous* variables in SARIMAX, that may influence the target time series. By including these additional predictors, SARIMAX can capture the impact of factors beyond the historical values and seasonal patterns of the series itself.

In your chocolate shop use case, you are forecasting chocolate revenue. A SARIMA model would rely on historical revenue data and recurring seasonal patterns, such as increased revenue during holidays or special occasions. However, chocolate sales and revenue may also be affected by external factors. For instance, promotional campaigns, product coupons, changes in product pricing, or advertising efforts can significantly influence customers' purchasing behavior. A SARIMAX model can incorporate these exogenous variables into the forecasting process, allowing it to account for their impact and potentially produce more accurate and reliable forecasts.

The X factor — Exogenous variables

Exogenous variables are factors outside your main time series that might influence it. In your chocolate shop use case, these could be promotional campaigns, holidays, or even nearby events. SARIMAX takes these into account, giving you a more holistic view of what affects your data.

SARIMAX builds upon the seasonal (S), autoregressive (AR), integrated (I), and moving average (MA) components of SARIMA, and adds **exogenous variables (X)**. The model analyzes how external variables have historically affected your main series and uses this to make more accurate predictions.

Using SARIMAX effectively means carefully selecting and understanding your exogenous variables. It is crucial to choose factors that genuinely influence your main series and for which you have reliable future data for forecasting.

SARIMAX in R

Let's put SARIMAX into action:

1. First, you need to transform the string columns into numeric types. This step is essential because percentage signs need to be removed, and the values should be converted to numeric types for analysis. You use the `gsub()` function to remove all the `%` symbols (replace `%` with an empty string) on `discount_rate` and `coupon_rate` columns and use `as.numeric()` to convert them to numeric columns. Once converted, assign the numeric vector back to the `discount_rate` and `coupon_rate` column:

   ```
   df <- df %>%
     mutate(discount_rate = as.numeric(gsub("%", "", discount_rate)),
            coupon_rate = as.numeric(gsub("%", "", coupon_rate)))

   str(df)
   ```

   The following is the display on the data structure of the data frame object:

```
tibble [1,795 × 5] (S3: tbl_df/tbl/data.frame)
 $ date         : Date[1:1795], format: "2018-01-01" "2018-01-02" ...
 $ y            : num [1:1795] 6270839 8922076 8446101 7785798 6375303 ...
 $ discount_rate: num [1:1795] 34.3 30.9 28.1 27.3 25.7 ...
 $ coupon_rate  : num [1:1795] 1.09 1.08 1.01 0.96 0.9 0.87 1.06 1.03 1 0.91 ...
 $ y_diff       : num [1:1795] NA 2651237 -475975 -660303 -1410495 ...
```

Figure 7.18: Dataframe structure after applying transformations using the mutate function

2. Next, you split the `discount_rate` and `coupon_rate` data into training and test sets. This step is crucial to evaluate the model's performance on unseen data while also using the additional regressor variables:

```
test_days <- 30
exog_train <- head(df[, 3:4], -test_days)
exog_test <- tail(df[, 3:4], test_days)
head(exog_test)
```

You get the following output:

```
# A tibble: 6 × 2
  discount_rate coupon_rate
          <dbl>       <dbl>
1          23.8        1.5
2          20.8        0.94
3          21.8        0.92
4          20.2        0.88
5          17.8        0.41
6          16.4        0.31
```

Figure 7.19: Sample data showing discount_rate and coupon_rate variables

This allows you to include these variables in your SARIMAX model.

3. You use the `auto.arima` function from the `forecast` package to find the best SARIMAX model for your training data, including the exogenous variables:

```
model_sarimax <- auto.arima(train_ts, seasonal = TRUE, period = 7, xreg =
as.matrix(exog_train))
summary(model_sarimax)
```

The `auto.arima` function fits a SARIMAX model to the training data and selects the best parameters, considering weekly seasonality (`period = 7`) and including the exogenous variables (`xreg`). You get the following output:

```
Series: train_ts
Regression with ARIMA(1,1,1)(0,0,2)[7] errors

Coefficients:
         ar1      ma1     sma1     sma2  discount_rate  coupon_rate
      0.3000  -0.8679  0.1472  0.0825       363626.0    731503.09
s.e.  0.0314   0.0176  0.0251  0.0230        31526.6     64182.37

sigma^2 = 7.5e+12:  log likelihood = -28648.15
AIC=57310.31   AICc=57310.37   BIC=57348.63

Training set error measures:
                    ME    RMSE     MAE       MPE     MAPE      MASE         ACF1
Training set 28398.79 2733101 1341705 -1.499182 11.65821 0.7399401 -0.00223584
```

Figure 7.20: SARIMAX model output showing coefficients, diagnostics, and error measures

The output shows a SARIMAX model fitted with an `ARIMA(1,1,1)(0,0,2)[7]` structure and two external regressors, `discount_rate` and `coupon_rate`. The `ARIMA(1,1,1)(0,0,2)[7]` coefficients for the autoregressive (AR1), moving average (MA1), seasonal moving averages (SMA1 and SMA2), and the external regressors indicate their respective impacts on the model.

4. You then generate predictions for the test period using the fitted SARIMAX model, also providing the exogenous variables for the test set:

```
predictions_sarimax <- forecast(model_sarimax, xreg =
as.matrix(exog_test), h = test_days)$mean
```

The `forecast` function produces predictions for the specified number of periods (`h`), which, in this case, is the length of the test set, using the exogenous variables from the test set.

5. Finally, you assess the model using the previously defined `model_assessment` function. This involves visualizing the train, test, and forecasted values and calculating error metrics:

```
model_assessment(train, test, predictions_sarimax, "SARIMAX")
The MAE is 6657928.84
The RMSE is 11125124.98
The MAPE is 23.31 %
```

The SARIMAX model's performance metrics show an **MAE** of 6.66 million, an **RMSE** of 11.13 million (in this case, this is not a big deal – the scale is relative to the scale of the y variable), and an **MAPE** of 23.31%. This means the model's average prediction error is around 23.31% of the actual values. This is another minor improvement compared to SARIMA.

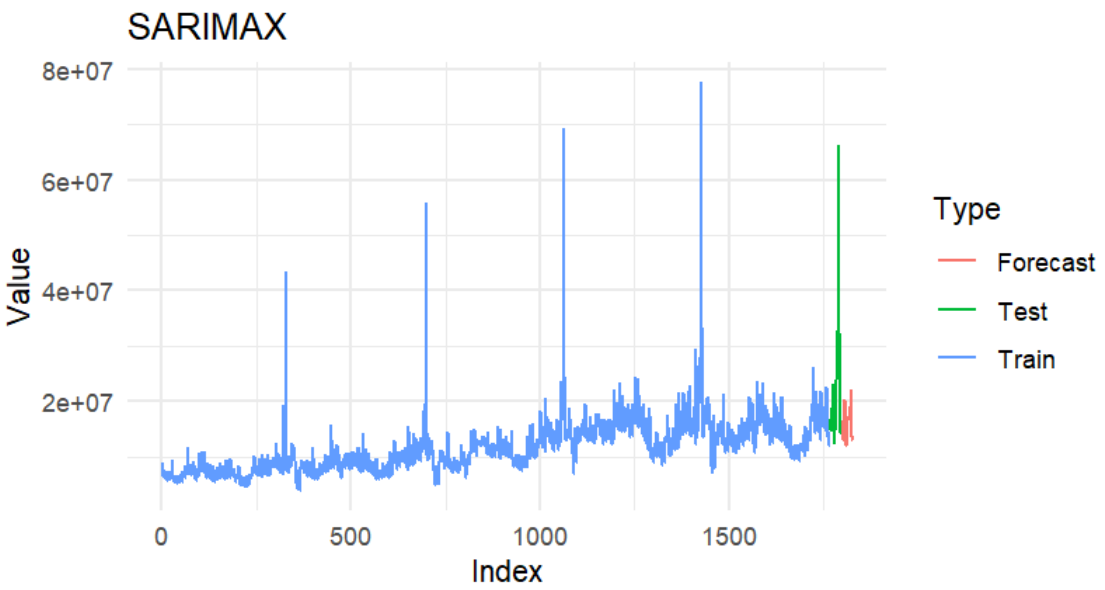


Figure 7.21: SARIMAX model plot showing train, test, and forecast data

The plot shows that the training data (blue) exhibits notable spikes, while the forecast (red) and test data (green) portions at the end align more closely, though the forecast seems to slightly underestimate the test values, indicating some room for improvement in accuracy.

By following these steps, you split the data, fit a SARIMAX model including exogenous variables, make predictions, and assess the model's performance. This process ensures that you evaluate the model on both training and unseen test data, providing a comprehensive understanding of its accuracy and reliability.

Cross-validation

Cross-validation is essential in forecasting because it helps build confidence in how well a model truly performs. Instead of relying on a single test, you repeatedly evaluate the model under different conditions to see whether its results hold up. This involves creating multiple training and test sets, allowing the model to be tested again and again as new data is introduced. For example, you can gradually expand the training set by incorporating previous test data and rerunning the model several times. In time series forecasting, this process becomes especially important to test the model in different seasonalities.

There are two types of cross-validation.

Rolling forecast is a time series evaluation technique in which the model is tested on a series of often single future observations. The following image provides a visualization of this method:

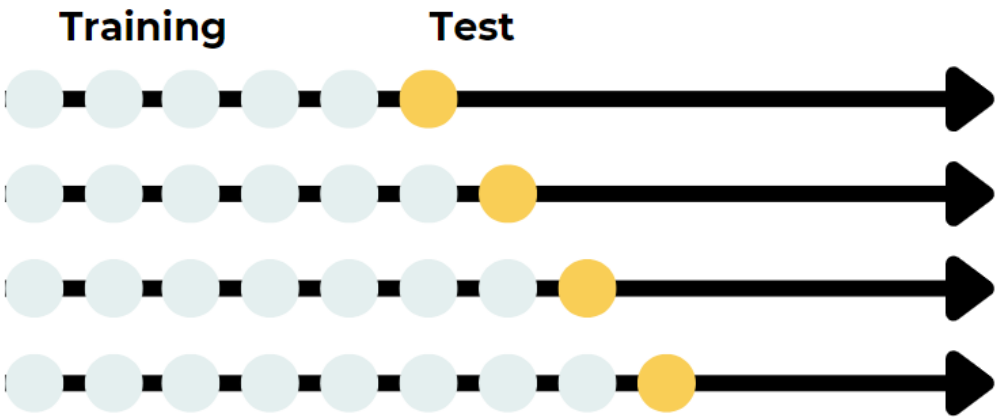


Figure 7.22: Rolling cross-validation

The other type is **sliding forecast**, which involves trimming older data from the training set during each round of cross-validation while keeping its overall size constant. *Figure 7.23* shows a visualization:

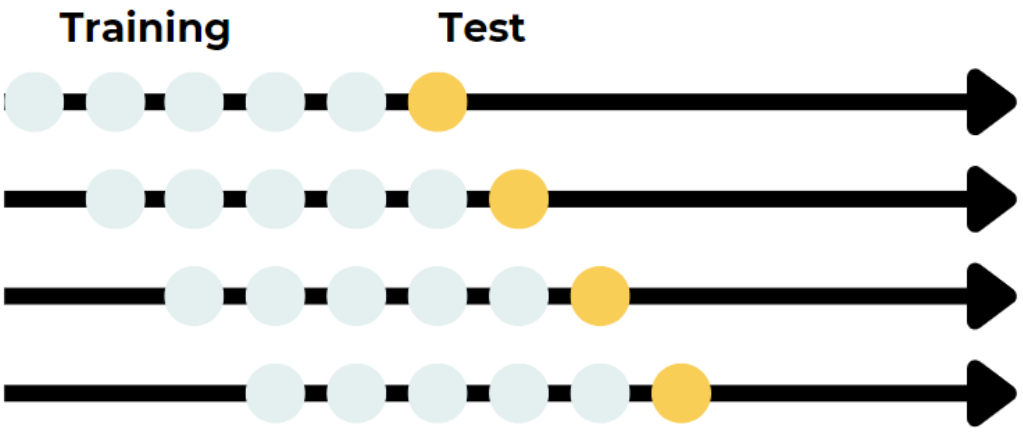


Figure 7.23: Sliding cross-validation

Next, you will apply these cross-validation techniques in R.

Cross-validation in R

Cross-validation is an essential technique for evaluating the performance of a time series model. It helps ensure that the model generalizes well to new data by testing it on multiple subsets of the dataset.

In this section, you will learn the steps for setting up and performing cross-validation using an ARIMA model in R:

1. First, you set the cross-validation rules. You use a rolling forecast approach, which involves making predictions on a rolling basis and updating the training set with each step:

```
rolling_forecast_cv <- function(data, initial, h, step, arima_order,
seasonal_order, xreg_cols = NULL, period = NULL) {
  n <- nrow(data)
  print(n)  # Check the number of rows in the dataset
  print(initial)  # Check initial size
  print(h)  # Check the horizon size
  print(step)  # Check the step size

  if (n <= initial + h) {
    stop("Initial window size plus forecast horizon is larger than the
dataset")
  }

  results <- list()

  for (start in seq(1, n - initial - h, by = step)) {
    # Split the data into training and testing sets
    train <- data[start:(start + initial - 1), ]
    test <- data[(start + initial):(start + initial + h - 1), ]

    if (!is.null(xreg_cols)) {
      train_xreg <- as.matrix(train[, xreg_cols, drop = FALSE])
      test_xreg <- as.matrix(test[, xreg_cols, drop = FALSE])

      fit <- Arima(train$y, order = arima_order,
                   seasonal = list(order = seasonal_order, period =
period),
                   xreg = train_xreg)
```

```
      forecast_vals <- forecast(fit, h = h, xreg = test_xreg)$mean
    } else {
      fit <- Arima(train$y, order = arima_order,
                   seasonal = list(order = seasonal_order))
      forecast_vals <- forecast(fit, h = h)$mean
    }

    # Calculate RMSE for each forecast
    rmse <- sqrt(mean((test$y - forecast_vals)^2))
    results[[length(results) + 1]] <- rmse
  }

  return(results)
}
```

The `rolling_forecast_cv` function takes the data, initial training size, forecast horizon (h), and step size. It fits the `ARIMA` model on the training data, makes predictions, and calculates the `RMSE` for each rolling forecast step.

Once the parameters are passed in, the function will print out the parameters and ensure that the number of rows is larger than the initial training size plus the forecast horizon to move forward. It will then step into a loop, each time split the data into training and test data set based on the parameters. It will also ensure that the exogenous variables (`xreg`) are valid and create the training and test `xreg` accordingly.

Once the data and components are ready, it will fit the ARIMA or SARIMAX model, get the prediction, and finally calculate the `RMSE` for each forecast and put each `RMSE` into the results list and return back.

2. Next, you configure the model parameters and perform rolling-forecast cross-validation using the SARIMAX model. The Initial training data set to 1,400 observations, and the forecasting horizon is defined as 30 days. The `xreg_cols` parameter is assigned the value 3:4, specifying that the third and fourth columns of the dataset (`discount_rate` and `coupon_rate`) are included as exogenous variables in the models:

```
results <- rolling_forecast_cv(
  data = df,
  initial = 1400,
  h = test_days,
  step = floor(test_days / 2),  # Ensure step is a valid integer
```

```
  arima_order = c(1, 1, 1),
  seasonal_order = c(0, 0, 2),
  xreg_cols = 3:4
)
```

You then apply the cross-validation function to your dataset. `SARIMAX` model uses the previous optimal one `ARIMA(1,1,1)(0,0,2)`.

3. Calculate the mean RMSE from the results list:

```
rmse <- mean(unlist(results))

cat(sprintf("The RMSE is %.2f\n", rmse))
The RMSE is 4284417.85
```

This code applies the `rolling_forecast_cv` function to your dataset, using the specified initial training size, forecast horizon (`h`), and step size. It calculates the RMSE for each step and computes the average RMSE.

Parameter tuning

Parameter tuning is what takes a forecasting product from good to great. The purpose of parameter tuning is to move away from a one-size-fits-all approach and instead customize models for each specific situation, ultimately improving accuracy. This begins by defining a set of possible parameter values that the model can test.

Once the parameters are set, the model is run multiple times, each time with a different combination of values. After every run, the model's accuracy is measured, and the error is recorded. For example, you might test autoregressive values of 1, 2, and 3, running the model separately for each and comparing the results. The final step is to select the parameter combination that produces the lowest error, as this represents the most accurate model for your data.

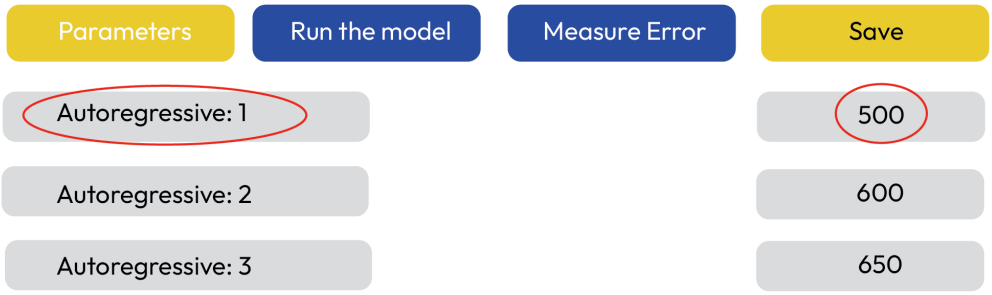


Figure 7.24: Parameter selection showing autoregressive values and error scores

The preceding diagram showing **Autoregressive: 1** has the lowest error (**500**), so its parameters should be the optimal set for the model.

Let's see how to put this into practice.

Parameter tuning in R

To achieve optimal model accuracy, it is essential to fine-tune the ARIMA parameters. By systematically adjusting these settings, you can identify the best combination for your time series data. In this section, you will learn how to define and search through different parameter configurations in R:

1. First, you define the grid of parameters to search through. This includes various values for the ARIMA and seasonal components. You will use an expanded `param_grid` with more possible combinations of p, d, q, P, D, and Q:

```
param_grid <- expand.grid(
  p = c(0, 1, 2, 3),  # Non-seasonal AR order (p)
  d = c(0, 1),        # Non-seasonal differencing (d)
  q = c(0, 1, 2),     # Non-seasonal MA order (q)
  P = c(0, 1),        # Seasonal AR order (P)
  D = c(0, 1),        # Seasonal differencing (D)
  Q = c(0, 1)         # Seasonal MA order (Q)
)

nrow(param_grid)
[1] 192

head(param_grid)
  p d q P D Q
```

```
1 0 0 0 0 0 0
2 1 0 0 0 0 0
3 2 0 0 0 0 0
4 3 0 0 0 0 0
5 0 1 0 0 0 0
6 1 1 0 0 0 0
```

The `expand.grid` function creates a data frame with all combinations of the specified parameters. You then count the number of parameter combinations.

2. Next, you define a function to perform cross-validation for a given set of parameters. This function will build an ARIMA model with the specified parameters, perform cross-validation, and return the RMSE. It handles the setup for tuning, iterates over the parameter grid, and calls the rolling forecast evaluation function (Since the function is relatively long, we will break the function into several parts for explanation purposes. When you run the code, please utilize the GitHub version, or run the whole function together to ensure the code executed in the right way).
Inside the function, it creates the placeholder for `best_param`, which is used to store the best parameters. `best_rmse` is used to store the lowest RMSE in the tuning:

```
rolling_forecast_cv_tuning <- function(data, initial, h, step, param_grid,
xreg_cols = NULL) {
  n <- nrow(data)

  best_params <- NULL
  best_rmse <- Inf  # Start with a high RMSE to track the best one
  error_count <- 0  # Track how many times an error occurs
```

3. Next, you start the `for` loop and walk through the parameter grid. You get the ARIMA parameters p, d, and q, and the seasonal parameters P, D, and Q from the grid. These will be used to compose the model:

```
  for (i in 1:nrow(param_grid)) {
    # Extract ARIMA and seasonal orders from param_grid
    arima_order <- c(param_grid$p[i], param_grid$d[i], param_grid$q[i])
    seasonal_order <- c(param_grid$P[i], param_grid$D[i], param_grid$Q[i])

    cat(sprintf("Testing params: p=%d, d=%d, q=%d, P=%d, D=%d, Q=%d\n",
                param_grid$p[i], param_grid$d[i], param_grid$q[i],
                param_grid$P[i], param_grid$D[i], param_grid$Q[i]))
```

```
results <- list()
```

4. You then create the train and test datasets. The training data is used to fit the ARIMA model, and the test data is used to calculate the RMSE for each forecast. When `xreg` parameters exist, it will fit a SARIMAX model with external regressors:

```
for (start in seq(1, n - initial - h, by = step)) {
 train <- data[start:(start + initial - 1), ]
 test <- data[(start + initial):(start + initial + h - 1), ]

 if (!is.null(xreg_cols)) {
   train_xreg <- as.matrix(train[, xreg_cols, drop = FALSE])
   test_xreg <- as.matrix(test[, xreg_cols, drop = FALSE])
 }

 result <- tryCatch({
   if (!is.null(xreg_cols)) {
     # Fit the ARIMA or SARIMAX model with external regressors
     fit <- Arima(train$y, order = arima_order,
                  seasonal = list(order = seasonal_order),
                  xreg = train_xreg)
     forecast_vals <- forecast(fit, h = h, xreg = test_xreg)$mean
   } else {
     # Fit the ARIMA model without external regressors
     fit <- Arima(train$y, order = arima_order,
                  seasonal = list(order = seasonal_order))
     forecast_vals <- forecast(fit, h = h)$mean
   }

   rmse <- sqrt(mean((test$y - forecast_vals)^2))
   rmse  # Return RMSE

 }, error = function(e) {
   # Log the error and skip this iteration
   cat("Error: ", conditionMessage(e), "\n")
   error_count <<- error_count + 1
   return(NULL)  # Return NULL if there's an error
 })

 if (!is.null(result)) {
```

```
        results[[length(results) + 1]] <- result
      }
    }
```

5. The RMSEs will be stored and average RMSE will be calculated; the value will be compared to the best RMSE (initially set to positive infinity). If a lower average RMSE is found, it will be stored as the best RMSE. Together with the best RMSE, the corresponding parameters used will be stored in the `best_params` variable:

```
    if (length(results) > 0) {
      avg_rmse <- mean(unlist(results))
      cat(sprintf("Average RMSE for current params: %.2f\n", avg_rmse))

      # Track the best parameters
      if (avg_rmse < best_rmse) {
        best_rmse <- avg_rmse
        best_params <- param_grid[i, ]
      }
    }
  }
```

You also want the function to report the number of errors:

```
  cat(sprintf("Number of errors encountered: %d\n", error_count))
```

Return the best parameter combination and its corresponding RMSE with this line:

```
  if (!is.null(best_params)) {
    cat(sprintf("Best parameters: p=%d, d=%d, q=%d, P=%d, D=%d, Q=%d with
RMSE: %.2f\n",
                best_params$p, best_params$d, best_params$q,
                best_params$P, best_params$D, best_params$Q, best_rmse))
  } else {
    cat("No valid model found.\n")
  }

  return(list(best_params = best_params, best_rmse = best_rmse))
}
```

The `rolling_forecast_cv` function focuses solely on rolling-forecast cross-validation, fitting models, and calculating RMSEs. You then loop through each combination of parameters, perform cross-validation, and store the RMSE for each set of parameters:

```
tuning_results <- rolling_forecast_cv_tuning(
  data = df,
  initial = 1400,
  h = 30,
  step = 15,
  param_grid = param_grid,  # Your param_grid data frame
  xreg_cols = 3:4  # Assuming columns 3 and 4 are external regressors
)
```

6. Finally, print the best parameters and RMSE:

```
cat(sprintf("Best RMSE is %.2f with parameters: p=%d, d=%d, q=%d, P=%d,
D=%d, Q=%d\n",
            tuning_results$best_rmse, tuning_results$best_params$p,
tuning_results$best_params$d,
            tuning_results$best_params$q, tuning_results$best_params$P,
tuning_results$best_params$D,
            tuning_results$best_params$Q))
Best RMSE is 1324700.87 with parameters: p=1, d=0, q=2, P=0, D=0, Q=0
```

You create an empty list to store the RMSE values. For each combination of parameters, you call the `Arima()` function, store the RMSE, and then combine the results into a data frame.

Please note that this tuning step might be time-consuming, and it is normal to see some parameter combinations with errors in the console output (hence the `tryCatch` block). For the tuning in this example, the lowest RMSE is 1.3 million, with the parameters (p, d, q, P, D, Q) printed out.

Summary

In this chapter, you explored the ARIMA family of models, starting with the foundational ARIMA model, which helps forecast future values based on past data points.

You saw how ARIMA, despite being robust and widely used, does not account for seasonality or external factors. This led you to its enhanced versions: SARIMA, which adds a seasonal component, and SARIMAX, which includes exogenous variables such as promotional events or holidays, giving a more comprehensive view of the data.

Using the case study of a chocolate shop forecasting daily revenues, you explored data preparation and exploratory data analysis (EDA). This involved loading and cleaning data, visualizing trends and seasonal patterns, and decomposing the time series into trend, seasonal, and residual components.

You also performed autocorrelation and partial autocorrelation analyses to guide model selection, ensuring that your approach was data-driven and thorough.

You wrapped up with a deep dive into model assessment and parameter tuning, emphasizing the importance of cross-validation for ensuring model reliability.

By fitting models, generating predictions, and evaluating performance using metrics such as MAE, RMSE, and MAPE, you learned to refine your forecasts for accuracy.

In the next chapter, you will learn how the TBATS model handles complex time series data with multiple seasonal patterns through its core components: trigonometric seasonality, Box-Cox transformation, ARMA errors, trend, and seasonal components. You will explore how these features make TBATS more effective than traditional models and go through practical steps to implement it in R, from data setup to interpreting the output and forecasting. The chapter also covers fine-tuning model parameters for optimal performance, equipping you to use TBATS for accurate time series forecasting.

8

Forecasting with TBATS Modeling

You'll start this chapter with an introduction to the TBATS forecasting model, providing a solid foundation for understanding its underlying concepts and applications. TBATS is an advanced statistical forecasting framework specifically designed to model time series data exhibiting multiple, complex or non-integer seasonal patterns. Unlike traditional forecasting methods, TBATS effectively captures complex seasonal structures through a combination of trigonometric seasonality, Box-Cox transformation, ARMA errors, trend, and seasonal components. You will learn how they all work together to tackle multiple seasonalities in data. Next, we focus on the practical implementation of TBATS models in R. You'll learn how to prepare time series data, fit a TBATS model, generate forecasts, and interpret the resulting outputs. Through step-by-step guides, you will gain the skills needed to apply TBATS models to your own datasets and improve forecasting performance for complex seasonal time series.

Finally, we break down the model itself. This section focuses on analyzing and selecting appropriate model components and parameters. You'll get to grips with how each part of the TBATS model contributes to its overall performance and learn how to tweak these components for optimal results.

We'll cover the following headings in the chapter:

- Overview of TBATS models
- Implementing TBATS in R
- Applying TBATS to multiple seasonal time series

Soon, you'll not only understand TBATS inside out but also be able to use it effectively to forecast time series data with multiple seasonal patterns. Let's dive in!

Technical requirements

R and RStudio are required for this chapter. Github scripts can be found in the repository.

You will need the following data file and packages for this chapter:

- `forecast`: The `forecast` package in R is a very popular set of tools designed for analyzing and forecasting univariate time series data. You will see some seasonal subseries plots using this package. Our TBATS model and functions are from this package.
- `AirPassengers`: The `AirPassengers` dataset is a classic built-in dataset in R used for time series analysis. It contains monthly totals of international airline passengers (in thousands) from 1949 to 1960, with 144 observations in total.
- `taylor`: The `taylor` electricity demand dataset is a widely used time series dataset for evaluating forecasting models with multiple seasonal patterns. It contains 4,032 half-hourly electricity demand observations, representing approximately 12 weeks of electricity consumption data. Because the data is recorded at 30-minute intervals, it contains two important seasonal cycles: a daily pattern with 48 observations per day and a weekly pattern with 336 observations per week.

Overview of TBATS models

TBATS is an advanced statistical forecasting model designed to tackle complex time series forecasting challenges. It is particularly well suited for time series data containing multiple, complex and even non-integer seasonal patterns that are difficult to model using traditional forecasting methods. The model was introduced in 2011 by Alysha M. De Livera, Rob J. Hyndman, and Ralph D. Snyder in their paper, *Forecasting Time Series with Complex Seasonal Patterns Using Exponential Smoothing*, published in the Journal of the American Statistical Association. Their work addressed a significant limitation of existing forecasting methods by providing a unified framework capable of handling multiple and non-integer seasonal patterns within a single model.

The name TBATS is derived from the five key components that make up the model: **trigonometric seasonality**, **Box-Cox transformation**, **ARMA errors**, **trend**, and **seasonal components**. Each component is responsible for modeling a different characteristic of the time series. Together, they form a unified forecasting framework capable of capturing complex seasonal behavior, long-term trends, changing variance, and residual autocorrelation.

One of the primary strengths of TBATS is its ability to model multiple seasonalities within the same dataset. Many real-world time series contain several overlapping seasonal cycles. For

instance, retail sales might have weekly seasonality due to customer shopping habits, yearly seasonal patterns driven by holidays, and even longer-term recurring cycles. Traditional forecasting models often struggle with these situations because they typically assume only a single seasonal period. TBATS overcomes this limitation by modeling multiple seasonal patterns concurrently, resulting in more accurate and flexible forecasts.

Throughout this chapter, we will use the `AirPassengers` dataset to illustrate the key concepts behind each component of the TBATS model. Although this dataset contains only a single yearly seasonal cycle, it clearly demonstrates how TBATS models trend, seasonality, changing variance, and residual autocorrelation. Later in the chapter, we will examine examples containing multiple seasonalities using the `taylor` electricity demand dataset, where the full capabilities of TBATS become even more apparent.

Let's take a look at the components first to understand their capabilities.

Trigonometric seasonality

The **trigonometric seasonality** component is one of the defining features of the TBATS model. Instead of representing seasonal effects using fixed seasonal indices, TBATS models seasonality with Fourier series, which express periodic patterns as combinations of sine and cosine functions. This mathematical representation provides greater flexibility than traditional seasonal models and allows TBATS to capture complex repeating patterns that may vary in shape and duration.

One of the major advantages of this approach is its ability to model multiple seasonalities simultaneously. Many real-world datasets exhibit several recurring cycles at different frequencies. For example, electricity demand often exhibits daily, weekly, and yearly seasonal patterns, while retail sales may contain weekly purchasing behavior together with annual vacation effects. TBATS models these seasonal cycles concurrently without requiring separate forecasting models for each seasonal period.

The `AirPassengers` dataset provides a simple illustration of trigonometric seasonality. The dataset records the monthly number of international airline passengers from 1949 to 1960 and exhibits a clear yearly seasonal pattern. Passenger numbers consistently peak during the summer months and decline during the winter months. TBATS represents this repeating annual pattern using Fourier series, producing a smooth and flexible seasonal model. Although `AirPassengers` contains only a single seasonal cycle, the same trigonometric representation naturally extends to datasets containing multiple seasonal patterns.

The trigonometric approach is particularly beneficial when dealing with non-integer seasonality (the seasonal cycles that do not repeat over a precise integer number of time steps using sine and cosine pairs). Such irregular seasonal behavior frequency occurs in high-

frequency data, making Fourier series one of the key reasons why TBATS performs well on complex forecasting use cases.

Box-Cox transformation

The **Box-Cox transformation** is a statistical technique used to stabilize the variance of a time series before the forecasting model is estimated. Many real-world datasets exhibit increasing variability as the magnitude of the observations increases. This phenomenon, known as **heteroscedasticity**, can violate the assumptions of many statistical forecasting models and reduce forecasting accuracy.

The `AirPassengers` dataset provides an excellent example of this behavior. Passenger numbers increase steadily over time, but the size of the seasonal fluctuations also grows. During the early years of the dataset, the difference between seasonal peaks and troughs is relatively small. By the end of the series, however, these fluctuations become substantially larger. Because the variance changes over time, fitting a forecasting model directly to the original data may lead to less reliable parameter estimates.

The Box-Cox transformation addresses this issue by transforming the data into a scale where the variance remains more consistent over time, producing a more homoscedastic dataset. A more stable variance allows the forecasting model to estimate its parameters more accurately and often leads to improved forecast performance. Once forecasting has been completed, the inverse Box-Cox transformation converts the predictions back to the original scale of the data.

The transformation is controlled by a parameter known as **lambda** (**λ**). TBATS can estimate the optimal value of λ automatically, although users may also specify the value manually when appropriate. A value of $\lambda = 1$ leaves the data unchanged, $\lambda = 0$ corresponds to a logarithmic transformation, while other values apply varying degrees of power transformation.

Although the Box-Cox transformation is incorporated into TBATS, it is not unique to this model. It is widely used in statistical forecasting and regression techniques, including multiple linear regression, time-series analysis, and even machine learning. Whenever a dataset exhibits changing variance, applying a Box-Cox transformation often improves the performance of the underlying statistical model.

ARMA errors

The **ARMA errors** component models any remaining autocorrelation in the residuals after the trend and seasonal patterns have been captured. Ideally, the residuals of a forecasting model should resemble random noise, containing no systematic information that could further improve the forecasts. However, many real-world time series retain short-term dependencies even after the primary patterns have been modeled. The **ARMA** errors component captures

these remaining relationships, improving both the accuracy and statistical validity of the forecasting model.

ARMA stands for **AutoRegressive Moving Average** and consists of two complementary components:

- The Autoregressive (AR) component models the relationship between the current observation and its previous observations
- The Moving Average (MA) component models the relationship between the current forecast error and previous forecast errors

Together, these components capture the short-term temporal dependencies that remain after the trend and seasonal effects have been modeled.

In *Chapter 7*, we explored ARIMA in detail; you may wonder why TBATS uses ARMA rather than ARIMA. The difference lies in the purpose of each model. An ARIMA model includes an additional *Integrated* (I) component, which applies differencing to transform a non-stationary time series into a stationary one before modeling the data. In contrast, TBATS explicitly models the trend and seasonal structure through its trend and seasonal components. Once these systematic patterns have been removed, the remaining residuals are expected to be approximately stationary. Therefore, additional differencing is generally unnecessary, allowing TBATS to model the residuals using an ARMA process instead of an ARIMA model.

Using the `AirPassengers` dataset as an example, the long-term upward trend and yearly seasonal pattern explain most of the variation in passenger numbers. However, small correlations may still exist between consecutive forecast errors. The ARMA errors component models these remaining dependencies, reducing autocorrelation in the residuals and improving the overall forecasting performance of the TBATS model.

Residual diagnostics are typically performed after fitting the model to verify that the remaining errors resemble white noise, meaning they contain no significant autocorrelation or systematic patterns. When this condition is satisfied, it indicates that the TBATS model has successfully captured the underlying structure of the time series and that the remaining residuals consist primarily of random variation.

Trend

The **trend components** model the long-term direction of a time series. Many datasets exhibit sustained increases or decreases over time due to factors such as population growth, technological advances, inflation, or changing consumer demand. Separating long-term trend from seasonal variation is essential for producing reliable forecasts.

The `AirPassengers` dataset clearly illustrates this concept. Between 1949 and 1960, the number of international airline passengers increased steadily, reflecting the rapid expansion of

commercial air travel during that period. Although seasonal peaks occur every year, the overall level of passenger traffic continues to rise. The trend component models this long-term growth independently of the seasonal patterns, allowing TBATS to distinguish structural growth from recurring seasonal fluctuations.

Estimating the trend simultaneously with the remaining TBATS components enables the model to generate forecasts that accurately reflect both long-term changes and shorter-term seasonal behavior.

Seasonal components

The **seasonal components** model recurring patterns that occur at regular intervals. These patterns may arise from weather conditions, holidays, business cycles, consumer behavior, or other predictable events. Seasonal effects are often among the most influential characteristics of time series data because they create systematic fluctuations that repeat over time.

The `AirPassengers` dataset demonstrates a clear annual seasonal pattern. Passenger numbers consistently increase during the summer travel season and decline during the winter months. The seasonal component captures this recurring yearly behavior, allowing the forecasting model to anticipate future seasonal fluctuations with greater accuracy.

Although `AirPassengers` contains only a single seasonal cycle, many real-world datasets exhibit multiple seasonalities. For example, hourly electricity demand often contains daily, weekly, and yearly seasonal patterns, while call center data may display hourly, daily, and weekly cycles simultaneously. TBATS models all of these seasonal patterns within a single forecasting framework, making it considerably more flexible than many traditional forecasting models that assume only one seasonal period. In a later section, you will see the `taylor` examples, which demonstrate even more details on how TBATS handles multiple seasonalities in the time series.

The seasonal components work closely with the trigonometric representation provided by the Fourier series. Together, they enable TBATS to model both simple and highly complex seasonal structures, making it one of the most powerful statistical forecasting models for analyzing real-world time series.

Together, these elements form a comprehensive model that excels in capturing the nuances of complex time series data. Next, let's utilize the classic `AirPassengers` dataset to see how to implement TBATS in R.

Implementing TBATS in R

Implementing TBATS models in R opens up a powerful toolkit for handling complex time series data with multiple seasonal patterns. R, with its rich ecosystem of packages, makes it

straightforward to apply TBATS and extract meaningful insights from your data. Let's walk through the process of implementing TBATS in R, from setting up your environment to interpreting the results.

Before we start the R coding, let's have an overview of the dataset we will use in this section.

Introducing the AirPassengers dataset

As we discussed in the *Overview of TBATS models* section, the `AirPassengers` dataset is one of the best-known benchmark datasets in time series analysis and has been widely used for decades to illustrate forecasting techniques and compare the performance of different forecasting models.

The dataset contains the monthly totals of international airline passengers from January 1949 through December 1960, providing 144 consecutive monthly observations. The data was originally compiled from international airline traffic statistics and has become a standard example in the forecasting literature because it exhibits several characteristics that are frequently encountered in real-world time series data.

The dataset was selected for this chapter because it exhibits several important characteristics commonly found in time series data. It shows a clear upward trend, a strong yearly seasonal pattern, and increasing seasonal variation over time. These characteristics make it well suited for illustrating how the different components of the TBATS model work together to capture trend, seasonality, changing variance, and residual autocorrelation.

Although the `AirPassengers` dataset contains only a single yearly seasonal cycle, it provides an excellent foundation for learning the TBATS modeling process. After understanding the workflow with this dataset, you can apply the same techniques to more complex datasets containing multiple seasonal patterns, which is where TBATS demonstrates its greatest advantage.

In the following exercise, you will use the `AirPassengers` dataset to build your first TBATS forecasting model, explore its components, and generate forecasts in R.

TBATS forecasting in R

Let's get started:

1. First, ensure you have the necessary packages installed and loaded into your R environment. You'll need the `forecast` package, which includes the TBATS function. If you don't have it installed yet, you can do so with the following commands:

```
if (!requireNamespace("forecast", quietly = TRUE)) {
  install.packages("forecast")
```

```
}
library(forecast)
```

2. Next, we'll load a sample dataset to apply the TBATS model. We'll use the `AirPassengers` dataset, a classic time series dataset that contains, on a monthly level across several years, international airline passengers. This dataset is built into R, making it easy to access and use:

```
data("AirPassengers")
```

3. Let's print out the `AirPassenger` dataset in the console. This will be helpful for you to see the raw data directly:

```
print(AirPassengers)

     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1949 112 118 132 129 121 135 148 148 136 119 104 118
1950 115 126 141 135 125 149 170 170 158 133 114 140
1951 145 150 178 163 172 178 199 199 184 162 146 166
1952 171 180 193 181 183 218 230 242 209 191 172 194
1953 196 196 236 235 229 243 264 272 237 211 180 201
1954 204 188 235 227 234 264 302 293 259 229 203 229
1955 242 233 267 269 270 315 364 347 312 274 237 278
1956 284 277 317 313 318 374 413 405 355 306 271 306
1957 315 301 356 348 355 422 465 467 404 347 305 336
1958 340 318 362 348 363 435 491 505 404 359 310 337
1959 360 342 406 396 420 472 548 559 463 407 362 405
1960 417 391 419 461 472 535 622 606 508 461 390 432
```

4. Plot the dataset to visualize the time series:

```
plot(AirPassengers, main="AirPassengers Data", ylab="Number of
Passengers", xlab="Time")
```

This produces the following output:

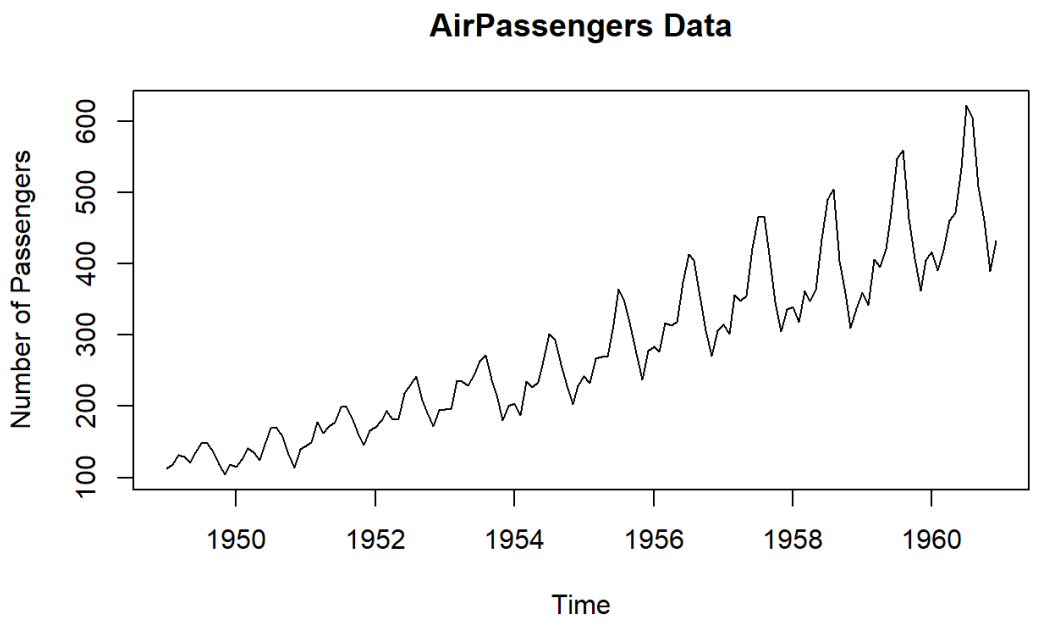


Figure 8.1: Plot of AirPassengers time series data

5. With our data loaded and visualized, we can now fit a TBATS model to the `AirPassengers` data. The `tbats()` function in the `forecast` package allows us to create a TBATS model, which will automatically determine the optimized model components for our data:

```
tbats_model <- tbats(AirPassengers)
```

6. Print the fitted TBATS model to see the components and parameters:

```
tbats_model
```

We get the following:

```
TBATS(0, {0,0}, 1, {<12,5>})

Call: tbats(y = AirPassengers, seasonal.periods = 12)

Parameters
  Lambda: 0
  Alpha: 0.7539964
  Beta: 0.03574778
  Damping Parameter: 1
  Gamma-1 Values: 0.003469021
  Gamma-2 Values: 0.005754791

Seed States:
               [,1]
 [1,]  4.810033172
 [2,] -0.008583295
 [3,] -0.148378447
 [4,]  0.056357573
 [5,] -0.009050016
 [6,]  0.010821683
 [7,]  0.005632253
 [8,]  0.027288737
 [9,]  0.058710056
[10,] -0.027537728
[11,] -0.032132832
[12,] -0.021334438
attr(,"lambda")
[1] 1.565565e-08

Sigma: 0.03709061
AIC: 1397.015
```

Figure 8.2: TBATS fitted model summary output showing components and parameters

The summary output provides detailed information about the fitted model, including the estimated parameters for the Box-Cox transformation, ARMA errors, trend, and seasonal components. This insight helps us understand how the model captures the underlying patterns in the data. The first line summarizes the overall model configuration: `TBATS(0,{0,0},1,{<12,5>}).s`

The following table captures how each part of the model output relates to the TBATS framework:

TBATS component	**Model output**	**AirPassengers result**	**Interpretation**
T – Trigonometric seasonality	{<12,5>}	Seasonal period = 12 months Fourier harmonics = 5	The model detected a yearly seasonal cycle and represented it using five pairs of sine and cosine functions (Fourier harmonics). This flexible representation captures the recurring seasonal pattern.
B – Box-Cox transformation	Lambda	0	A Lambda value of 0 corresponds to a logarithmic transformation. The log transformation stabilizes the increasing variance observed in the `AirPassengers` dataset before fitting the model.

TBATS component	**Model output**	**AirPassengers result**	**Interpretation**
A – ARMA errors	{0,0}	AR order (p) = 0 MA order (q) = 0	No autoregressive or moving average error terms were required. After modeling the trend and seasonality, the remaining residuals contained little autocorrelation, so an ARMA model was unnecessary.
T – Trend	Alpha Beta Damping Parameter	Alpha = 0.754 Beta = 0.036 Damping = 1	Alpha controls how quickly the model updates the level of the series. Beta controls how quickly the trend is updated. A damping parameter of 1 indicates that the model uses a non-damped trend.

TBATS component	Model output	AirPassengers result	Interpretation
S – Seasonal components	Gamma-1 Gamma-2	Gamma-1 = 0.00347 Gamma-2 = 0.00575	These parameters update the trigonometric seasonal states represented by the Fourier series. Their relatively small values indicate that the yearly seasonal pattern remains stable over time.

Table 8.1: TBATS fitted model core components details for AirPassengers dataset

The remaining output provides additional information about the quality of the fitted model:

Additional output	AirPassengers result	Interpretation
Sigma	0.0371	The estimated standard deviation of the residual errors. Smaller values indicate that the model explains a greater proportion of the variability in the data.
AIC	1397.015	The Akaike Information Criterion (AIC) measures the trade-off between model fit and complexity. Lower AIC values generally indicate a better model when comparing multiple TBATS models fitted to the same dataset.

Additional output	AirPassengers result	Interpretation
Seed States	12 estimated values	Initial values of the level, trend, and seasonal states used to initialize the state space model. These values are estimated automatically and are primarily used internally by the algorithm. They are generally not interpreted individually.

Table 8.2: TBATS fitted model additional information details for AirPassengers dataset

7. Next, we'll use the fitted TBATS model to make forecasts. The `forecast()` function generates future values based on the model, allowing us to see how well it predicts future trends for the next 24 months:

```
tbats_forecast <- forecast(tbats_model, h=24)
```

8. Plot the forecast along with the original data:

```
plot(tbats_forecast, main="TBATS Forecast for AirPassengers", ylab="Number
of Passengers", xlab="Time")
```

The preceding code will forecast the next 24 months (`h=24` specifies the forecast horizon) and plot the forecast out. The two greyed areas in the plot represent the 80% (darker gray) and 95% (lighter gray) prediction intervals. These areas quantify the uncertainty of the forecast, indicating the range based on confidence intervals:

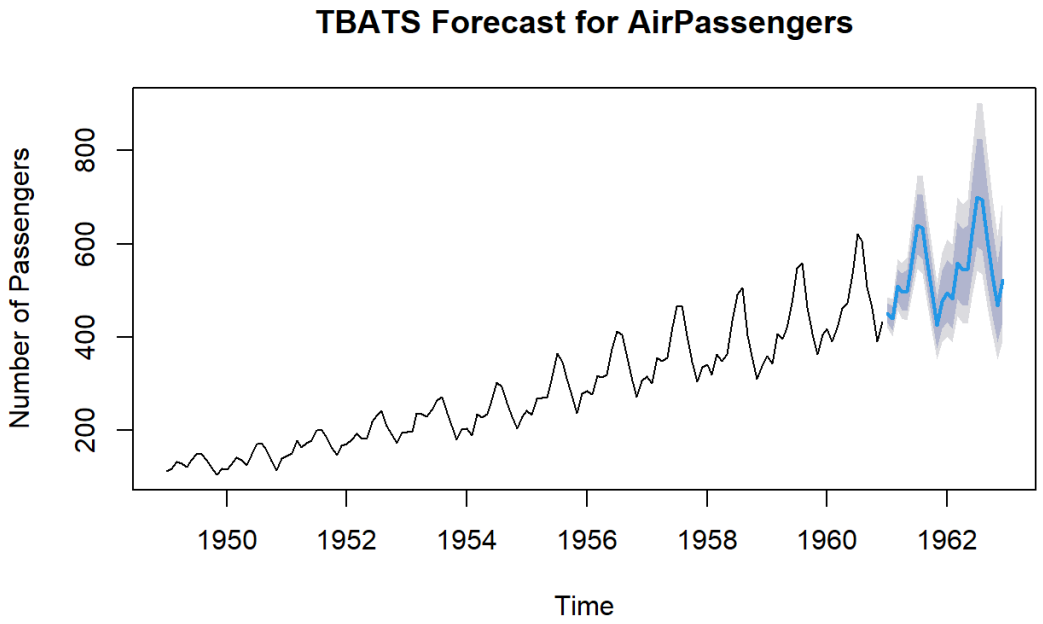


Figure 8.3: TBAST forecast for AirPassengers on the next 24 months

The forecast plot shows the predicted values for the next 24 months, along with confidence intervals. This visualization helps us assess the model's performance and reliability in forecasting future data points.

9. Finally, let's evaluate the accuracy of the TBATS model. We can compare the forecasted values to the actual data to measure the model's accuracy and make any necessary adjustments:

```
accuracy(tbats_forecast)
```

The code will calculate the accuracy metrics of the TBATS model. As the values displayed, ME, RMSE, MAE, MPE, and MAPE are all relatively low. Lower values typically indicate a good fitted model. MASE value is <1, indicating that the model is much better than a naive model. Overall, the model is performing very well.

```
                      ME      RMSE       MAE        MPE      MAPE       MASE
Training set 0.4655263 10.66144 7.720635 0.2467939 2.851916 0.2410416
                    ACF1
Training set 0.02145759
```

Figure 8.4: Accuracy metrics on the TBATS model

The accuracy metrics provide a quantitative assessment of the model's performance, including measures such as Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE).

Note

We did not split the dataset into training and test sets in this exercise, as our primary goal at this stage is to understand how TBATS works, examine the fitted model structure, and interpret how its components capture the trend, seasonality, and changing variance. Later in the chapter you will see examples of splitting the dataset into train and test sets, and using the test set to evaluate the performance of the model!

Implementing TBATS in R is a powerful way to handle complex seasonal data and make accurate forecasts. By following these steps, you can leverage the capabilities of TBATS to gain deeper insights and make more informed decisions based on your time series data. Whether you're dealing with retail sales, energy demand, or financial indicators, TBATS provides a robust framework for forecasting with precision and reliability.

So far you've implemented the TBATS in R, got familiar with the concepts and R code base. Let's move further with a more complex time series which has multiple seasonal patterns.

Applying TBATS to multiple seasonal time series

The `AirPassengers` dataset introduced the fundamental concepts of TBATS, including trend modeling, Box-Cox transformation, and seasonal representation. However, its single seasonal cycle does not fully demonstrate the full advantages of TBATS. To explore TBATS's ability to handle complex seasonal structures, this section uses the `taylor` electricity demand dataset, which contains multiple overlapping seasonal patterns.

Introducing the Taylor electricity demand dataset

The `taylor` dataset is a well-known time series dataset included in the R `forecast` package. It contains half-hourly electricity demand observations and is commonly used to demonstrate forecasting methods designed for data with multiple seasonal patterns.

Because observations are recorded every 30 minutes, the dataset contains 48 observations per day. It exhibits two important seasonal cycles: a daily seasonal period of 48 observations and a weekly seasonal period of 336 observations (48 × 7). Electricity demand typically follows recurring patterns throughout the day as consumer and business activity changes, while demand also differs systematically across the days of the week. There are 4,032 observations in the dataset.

These overlapping daily and weekly seasonal patterns make the `taylor` dataset particularly suitable for demonstrating the strengths of the TBATS model. Unlike the `AirPassengers` dataset, which contains only one primary seasonal cycle, the `taylor` dataset allows TBATS to model multiple seasonalities simultaneously using its trigonometric seasonal representation.

In this section, we will use the `taylor` dataset to explore TBATS model parameters and evaluate different model configurations. The data will be divided chronologically into training and test sets, and several TBATS specifications will be compared based on their forecasting accuracy. This provides a practical example of how model configuration and parameter selection can influence forecasting performance.

Exploring multiple seasonal patterns

With the introduction of `taylor` dataset described in the previous section, you will practice loading the dataset into R and explore more about the dataset and plot it so that you can see the data directly:

1. Load the dataset. The `taylor` dataset is built in with R; you can directly load the dataset as long as you have loaded the `forecast` package already:

   ```
   data("taylor")
   ```

2. Examine the `taylor` dataset. You can simply type the name of the `taylor` dataset in R to explore the dataset:

   ```
   taylor
   ```

   As we described in the introduction, the dataset has 48 (daily) and 336 (weekly) seasonalities, which are printed in the following console printout:

```
Multi-Seasonal Time Series:
Start: 1 1
Seasonal Periods: 48 336
Data:
   [1] 22262 21756 22247 22759 22549 22313 22128 21860 21751 21336
  [11] 21363 22176 24649 27466 31020 33529 35428 36034 36834 37296
  [21] 37338 37608 37692 37944 37880 37493 37065 36855 36871 36837
  [31] 36603 36982 37369 37650 37413 36475 35486 34452 33704 32883
  [41] 31940 32215 32572 32850 32251 30895 28460 26572 25093 24320
  [51] 24684 25338 24943 24637 24424 24163 23943 23324 23412 24065
  [61] 26035 28632 32023 34315 35927 36367 37087 37184 37009 37212
  [71] 37531 37890 37982 37540 37160 36920 37060 36930 36640 36910
  [81] 37220 37530 37320 36331 35206 34203 33273 32376 31838 31211
  [91] 31505 31838 32185 30881 28800 26833 25095 24437 24697 25259
 [101] 25029 24806 24437 24279 23902 23418 23424 24094 26039 28636
```

Figure 8.5: Layout of Taylor data set

3. Plot the `taylor` data to visualize the dataset:

```
library(ggplot2)

autoplot(taylor) +
  labs(
    title = "Half-Hourly Electricity Demand",
    x = "Time",
    y = "Electricity Demand"
  )
```

This will generate the following diagram of the dataset:

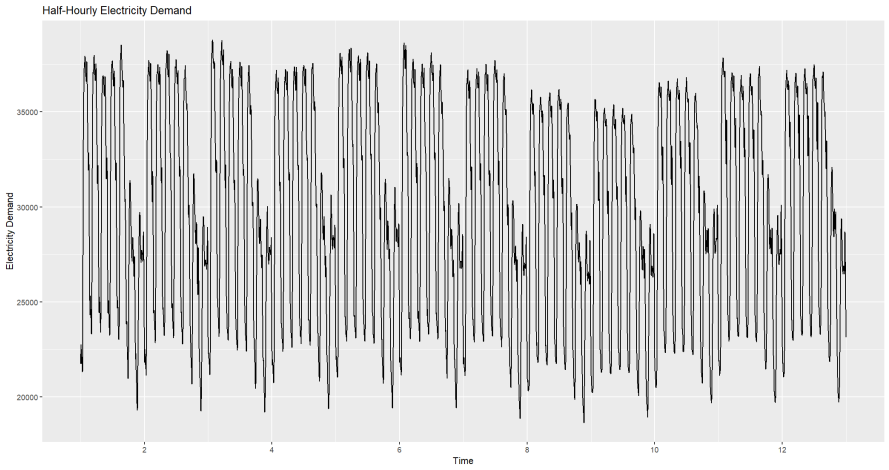


Figure 8.6: Plot of Taylor electricity demand time series data

We can see the seasonal patterns represented by the regular peaks and troughs in the graph.

Fitting the baseline TBATS model

Now the dataset is loaded, it's time for you to prepare the dataset, split it into training and test subsets, and fit it into the TBATS model. We pick 336 (one week) to be reserved as the test dataset for validation purposes:

1. Prepare the `taylor` dataset:

```
# Define the forecast horizon
h <- 336
# Split into training and test sets
train <- head(taylor, length(taylor) - h)
test <- tail(taylor, h)
```

2. Fit the default TBATS model:

```
default_model <- tbats(train)
```

3. Print the default fitted TBATS model

```
default_model
```

This produces the following output:

```
TBATS(0, {4,3}, -, {<48,11>, <336,6>})

Call: tbats(y = train)

Parameters
  Lambda: 0.000102
  Alpha: 0.01413051
  Gamma-1 Values: 5.1607e-05 -3.843489e-05
  Gamma-2 Values: -4.537189e-05 4.104274e-06
  AR coefficients: 0.794018 0.510489 -0.434233 -0.057371
  MA coefficients: 0.56897 -0.009836 -0.090937

Seed States:
               [,1]
 [1,] 10.2877017106
 [2,] -0.1765574979
 [3,]  0.0249609322
 [4,]  0.0203946459
 [5,] -0.0204189524
 [6,] -0.0252263523
 [7,] -0.0026245388
 [8,] -0.0024378595
 [9,] -0.0055915421
[10,] -0.0037674246
[11,]  0.0010770563
[12,] -0.0007372167
[13,] -0.1282774454
[14,] -0.0846922399
[15,]  0.0113026992
[16,] -0.0034828972
[17,] -0.0157817962
[18,] -0.0025977773
[19,]  0.0052614424
[20,]  0.0032465035
[21,]  0.0005379585
[22,] -0.0019129967
[23,] -0.0015110329
[24,] -0.0683107865
[25,]  0.0002167950
[26,]  0.0059714331
[27,] -0.0082989651
[28,] -0.0066927156
[29,]  0.0092288788
[30,]  0.0751121615
[31,]  0.0597963321
[32,]  0.0192828551
[33,]  0.0127814711
[34,]  0.0225499316
[35,]  0.0210083798
[36,]  0.0000000000
[37,]  0.0000000000
[38,]  0.0000000000
[39,]  0.0000000000
[40,]  0.0000000000
[41,]  0.0000000000
[42,]  0.0000000000
attr(,"lambda")
[1] 0.0001021968

Sigma: 0.008852752
AIC: 71488.12
```

Figure 8.7: TBATS Taylor dataset fitted model summary output

After fitting the default TBATS model to the training portion of the Taylor electricity demand dataset, the following model structure was selected: `TBATS(0, {4,3}, -, {<48,11>, <336,6>})`.

The preceding output is a strong example because it demonstrates TBATS's key strengths on multiple seasonalities and ARMA errors. This compact notation summarizes the major components selected by the model. Unlike the `AirPassengers` example, the `taylor` dataset contains two overlapping seasonal patterns. The fitted model therefore provides a clearer demonstration of how TBATS handles multiple seasonalities.

Let's take a detailed look at the default model and understand the components.

Understanding the fitted model

Let's dig a bit deeper into the fitted model and see how the components we saw earlier function in this particular case.

T — Trigonometric seasonalities (multiple)

The seasonal structure is represented as: `{<48,11>, <336,6>}`.

The first seasonal period, `<48,11>`, represents the daily cycle. Because the data are recorded every 30 minutes, 48 observations correspond to one day. The model selected 11 Fourier harmonics to represent this daily pattern.

The second seasonal period, `<336,6>`, represents the weekly cycle, since 336 half-hourly observations correspond to seven days. The model selected six Fourier harmonics for this seasonal pattern.

This is one of the main advantages of TBATS. Rather than representing seasonality using a large number of conventional seasonal states, TBATS uses pairs of sine and cosine terms to model each seasonal cycle efficiently and flexibly.

B — Box-Cox transformation

The estimated Box-Cox parameter is: `Lambda: 0.000102`.

Because this value is extremely close to zero, the transformation is approximately equivalent to a logarithmic transformation. This is also reflected in the compact model notation, where the value is rounded to 0. The transformation can help stabilize the variance of the electricity demand series and allow the model to represent changes more consistently across different demand levels.

A — ARMA errors

The notation: `{4,3}` indicates that the model selected an `ARMA(4,3)` error structure. The four autoregressive coefficients capture relationships between current residual errors and previous

residual errors, while the three moving-average coefficients capture relationships involving previous forecast errors.

The inclusion of an ARMA error component indicates that the trend and seasonal components alone did not capture all of the temporal dependence in the series. The ARMA structure helps model the remaining short-term autocorrelation.

T — Trend

The dash (`-`) in the model notation indicates that no explicit trend component was included. Consequently, the output does not contain Beta or a damping parameter. This suggests that, over the training period, the model did not require a separate long-term increasing or decreasing trend after accounting for the strong seasonal patterns and ARMA error structure.

S — Seasonal components

The model estimates two sets of seasonal smoothing parameters:

```
Gamma-1 Values:  5.1607e-05  -3.843489e-05
Gamma-2 Values: -4.537189e-05  4.104274e-06
```

There is one Gamma-1 and one Gamma-2 value for each seasonal period. These parameters control how the trigonometric seasonal states evolve as new observations become available. Their small magnitudes indicate that the estimated seasonal patterns change only gradually over time.

Other parameters

The 42 seed-state values are the initial values required to initialize the TBATS state-space model. For this fitted model, they include the initial level, the trigonometric states associated with the daily and weekly seasonal components, and the states required for the ARMA error structure.

These values are estimated automatically during model fitting and are primarily used internally by the algorithm. They generally do not need to be interpreted individually.

The estimated residual standard deviation is `Sigma: 0.008852752`. Because the model applies an approximately logarithmic Box-Cox transformation, Sigma is measured on the transformed modeling scale rather than directly in the original electricity-demand units.

The model also reports `AIC: 71488.12`. The **Akaike Information Criterion (AIC)** provides a measure that balances model fit against model complexity. It is primarily useful for comparing candidate models fitted to the same training data under comparable conditions. A lower AIC generally indicates a better balance between goodness of fit and model complexity.

Forecasting with the default TBATS model

Now we have a good understanding of the default TBATS model, let's use it to forecast.

1. Generate forecasts, still using the `taylor` dataset:

```
default_forecast <- forecast( default_model, h = h )
```

2. Evaluate forecast accuracy:

```
accuracy(default_forecast, test)
```

   We get the following output:

```
                     ME      RMSE      MAE           MPE      MAPE
Training set   1.410054 247.8219 180.6580 -0.001803784 0.6347864
Test set     -68.758362 881.3876 657.4424 -0.262703898 2.3313689
                  MASE        ACF1 Theil's U
Training set 0.3078625 -0.02481645        NA
Test set     1.1203591  0.88945181   1.01241
```

Figure 8.8: Accuracy metrics on the TBATS model fitted on the taylor dataset

The default TBATS model performs well on the training data, with an RMSE of 247.82, an MAE of 180.66, and a MAPE of only 0.63%. On the test set, the RMSE increases to 881.39, the MAE to 657.44, and the MAPE to 2.33%. Although forecast errors increase for unseen data, a MAPE of 2.33% indicates that the model's forecasts differ from the actual values by approximately 2.33% on average, demonstrating strong out-of-sample forecasting accuracy.

While the model achieves a low percentage error, there might still be room to improve the model through configuration tuning.

Evaluating alternative model configurations

The `tbats()` function automatically estimates many model parameters, including the Box-Cox transformation parameter, ARMA error orders and coefficients, smoothing parameters, and the number of Fourier harmonics used to represent seasonal patterns. It also determines whether certain model components are appropriate based on its automatic model-selection procedure.

However, the automatically selected model is not guaranteed to produce the highest forecasting accuracy on unseen data. Automatic selection focuses on finding a well-fitting model based on the training data and the model's internal selection criteria. Configuration tuning provides an additional opportunity to test different model configurations, such as enabling or disabling the Box-Cox transformation, ARMA errors, trend, and damped trend, and then compare their performance on a separate test set.

The default TBATS model provides a strong baseline, while the configuration tuning process helps determine whether a different combination of components is more suitable for the specific forecasting task.

Let's look at the code example for configuration tuning:

1. First, we must create a configuration tuning grid, which contains the options to turn on/off based on the TRUE/FALSE value passed in for the related components. The names used here are straightforward and correspond to their related component. TBATS still estimates the numerical parameters such as alpha, lambda, ARMA coefficients, and Fourier harmonics internally:

```
tuning_grid <- expand.grid(
  use.box.cox = c(TRUE, FALSE),
  use.arma.errors = c(TRUE, FALSE),
  use.trend = c(TRUE, FALSE),
  use.damped.trend = c(TRUE, FALSE)
)
```

2. Next, remove invalid combinations. A damped trend is a modification of an existing trend. We will need to remove this invalid combination from the grid:

```
tuning_grid <- subset(
  tuning_grid,
  !(use.trend == FALSE & use.damped.trend == TRUE)
)
```

3. Create the tuning coding. The configuration tuning process will loop through the components tuning grid, create the TBATS model with the related components on/off based on the grid's TRUE/FALSE value. The model will fit the train set and perform forecasts:

Note

This tuning code might take a while (for example, 30 minutes) to complete, depending on your compute resources.

```
results <- data.frame()

for (i in 1:nrow(tuning_grid)) {
```

```
  # Fit the TBATS model
  model <- tbats(
    train,
    use.box.cox = tuning_grid$use.box.cox[i],
    use.arma.errors = tuning_grid$use.arma.errors[i],
    use.trend = tuning_grid$use.trend[i],
    use.damped.trend = tuning_grid$use.damped.trend[i]
  )

  # Generate forecasts
  fc <- forecast(
    model,
    h = h
  )
```

4. The related TBATS models will be evaluated with the test set, and `accuracy()` will be used to calculate the metric values described in the previous section. The **RMSE**, **MAE** and **MAPE** from the `accuracy()` metrics will be collected and stored for evaluation:

```
  acc <- accuracy(
    fc,
    test
  )

  # Store the results
  results <- rbind(
    results,
    data.frame(
      use.box.cox = tuning_grid$use.box.cox[i],
      use.arma.errors = tuning_grid$use.arma.errors[i],
      use.trend = tuning_grid$use.trend[i],
      use.damped.trend = tuning_grid$use.damped.trend[i],
      RMSE = acc["Test set", "RMSE"],
      MAE = acc["Test set", "MAE"],
      MAPE = acc["Test set", "MAPE"]
    )
  )
}
```

5. Print the tuning results. The stored results will be used here. They will be sorted according to RMSE value; the top one, which has the lowest RMSE, will be the one we are looking for. The configuration set will be the optimal configuration for the TBATS model:

```
results <- results[ order(results$RMSE), ]

print(results)
```

We get the following output:

```
   use.box.cox use.arma.errors use.trend use.damped.trend      RMSE       MAE      MAPE
9         TRUE            TRUE     FALSE            FALSE  881.3876  657.4424 2.331369
3         TRUE           FALSE      TRUE             TRUE  882.9501  648.1648 2.308825
1         TRUE            TRUE      TRUE             TRUE  884.2438  653.0987 2.325391
11        TRUE           FALSE     FALSE            FALSE  884.4034  665.2067 2.356307
10       FALSE            TRUE     FALSE            FALSE  993.5426  730.0348 2.676482
12       FALSE           FALSE     FALSE            FALSE 1075.3027  853.2162 3.084129
7         TRUE           FALSE      TRUE            FALSE 1106.6903  845.9647 2.975423
2        FALSE            TRUE      TRUE             TRUE 1163.1899  875.1539 3.238129
5         TRUE            TRUE      TRUE            FALSE 1238.1634 1038.7114 3.535371
4        FALSE           FALSE      TRUE             TRUE 1499.0560 1273.0474 4.593691
8        FALSE           FALSE      TRUE            FALSE 1540.0688 1315.0819 4.628768
6        FALSE            TRUE      TRUE            FALSE 1876.7791 1594.7351 5.640654
```

Figure 8.9: Accuracy metrics on the TBATS model tuning

The configuration tuning process did not produce a significant improvement over the default TBATS model. The best tuned model achieved the same RMSE as the automatically fitted model, indicating that the default TBATS procedure had already identified an effective model configuration for the `taylor` dataset.

Although tuning did not improve forecast accuracy in this case, the exercise showed how alternative TBATS configurations can be systematically evaluated. It also confirms that adding or removing model components does not necessarily improve performance and that the automatically selected TBATS model can provide a strong baseline. In practice, configuration tuning is most valuable when it produces measurable improvements in out-of-sample accuracy or when users need to test specific modeling assumptions.

6. We then use the optimal model configuration and fit the final TBATS model with the whole `taylor` dataset:

```
final_model <- tbats(
  taylor,
  use.box.cox = TRUE,
  use.arma.errors = TRUE,
  use.trend = FALSE,
  use.damped.trend = FALSE
)
```

7. Finally, we use the fitted model to forecast the future one week's data:

```
final_forecast <- forecast(
 final_model,
 h = 336
)
```

8. Then plot the result on the diagram:

```
autoplot(final_forecast) +
  labs(
   title = "TBATS Forecast of Electricity Demand",
   x = "Time",
   y = "Electricity Demand"
  )
```

As you can see from the plot, the forecasts also display the multiple seasonal patterns that the original dataset has:

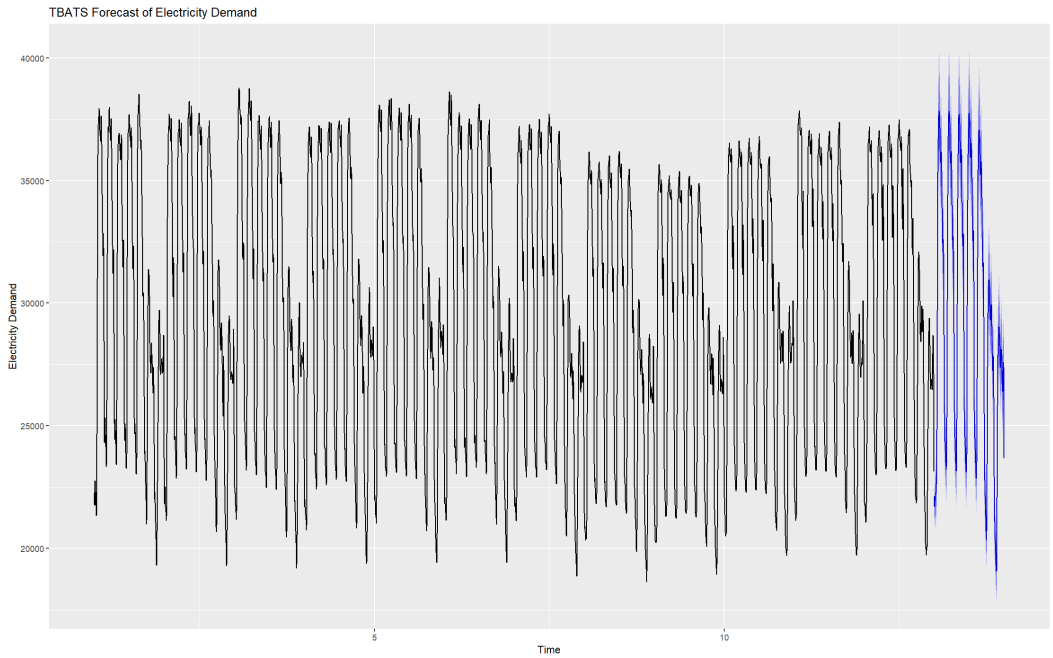


Figure 8.10: Forecast plot of the TBATS model

Understanding and optimizing the components of the TBATS model is essential for harnessing its full potential. By mastering these elements, you can significantly improve your time series forecasts, making them more accurate and reliable for decision-making. Whether you're working with sales data, energy demand, or other time series, TBATS provides a flexible and powerful framework for capturing the complexities of your data.

Summary

This chapter offers an overview of the intricacies of TBATS models, equipping you with the tools and knowledge to handle complex time series data with multiple seasonal patterns effectively.

You started by exploring the five core components of TBATS: Trigonometric seasonality, Box-Cox transformation, ARMA errors, Trend, and Seasonal components. Using `AirPassengers` dataset, you learned how these components work together and how to fit, interpret, and generate forecasts from a TBATS model in R.

The chapter then introduced the `taylor` electricity demand dataset, which contains multiple seasonal patterns corresponding to daily and weekly cycles. This example demonstrated one of the key strengths of TBATS: its ability to model multiple seasonalities simultaneously using

flexible trigonometric representations. You also learned how to evaluate the default TBATS model using a chronological training and test split and common forecast accuracy measures.

Finally, you explored model configuration tuning by testing different combinations of the Box-Cox transformation, ARMA errors, trend, and damped trend components. Although the best tuned configuration produced the same forecast accuracy as the default TBATS model, this exercise demonstrated how alternative model structures can be systematically evaluated. It also showed that configuration tuning does not always improve performance, as the automatic TBATS fitting process may already identify an effective model structure.

By the end of this chapter, you've gained a practical understanding of TBATS models and how to implement, interpret, evaluate, and optimize them in R. You are now equipped to leverage TBATS for various time series forecasting tasks, transforming your approach to analyzing complex seasonal data and making informed, data-driven decisions.

In the next chapter, you'll explore how Bayesian methods provide a flexible framework for time series forecasting by incorporating prior knowledge and updating predictions as new data arrives. By the end, you'll be equipped to apply Bayesian techniques to enhance your time series forecasting accuracy.

9

Bayesian Approaches in Time Series Analysis

Welcome to *Chapter 9*, where you will venture into Bayesian approaches in time series analysis. If you are accustomed to traditional methods, Bayesian techniques will offer you a fresh, insightful perspective. Bayesian analysis allows you to incorporate prior knowledge into your models and update your beliefs as new data comes in. This flexibility makes Bayesian methods particularly powerful for time series forecasting, where the ability to adapt to new information is crucial.

In this chapter, you will discover the principles of Bayesian inference and how these can be applied to time series data. You will cover everything from the foundational concepts to practical implementation using R, ensuring you can harness the full potential of Bayesian methods in your analysis.

By the end of this chapter, you will:

- Understand the core principles of Bayesian analysis in time series
- Apply Bayesian methods to time series data using R
- Evaluate and interpret results from Bayesian forecasting models

This chapter is designed to equip you with the knowledge and skills to apply Bayesian methods effectively in your time series analysis. Let's dive in and explore the powerful world of Bayesian time series modeling!

Technical requirements

R and RStudio are required for this chapter. GitHub scripts can be found in the folder.

You will need the following packages for this chapter:

- `brms`: The `brms` (**Bayesian Regression Models using Stan**) package in R allows users to fit complex Bayesian multilevel models using an intuitive formula syntax. It acts as an interface for Stan, which is a powerful probabilistic programming language that performs full Bayesian inference.
- `loo`: The `loo` package in R is an essential tool for Bayesian model evaluation, comparison and checking. You use the **Watanabe-Akaike information criterion** (**WAIC**) in our examples to assess the model.

Foundations of Bayesian time series analysis

Bayesian inference provides a framework for updating beliefs in light of new data. The process incorporates prior knowledge or beliefs about a parameter and updates this with the data evidence to form a posterior distribution.

Here are some key concepts related to Bayesian inference:

- **Prior distribution**: Represents your initial belief about a parameter before seeing the data
- **Likelihood**: The probability of the observed data given a parameter value
- **Posterior distribution**: Updated belief about the parameter after incorporating the data

For example, imagine you are forecasting the number of sales for a product. Based on historical data, you might have an initial belief (also called a **prior**) that the average sales per day follow a normal distribution with a mean of 100 and a standard deviation of 10. As you collect new sales data, you update this belief, resulting in a posterior distribution that better reflects the current trends.

Prior distributions

The **prior distribution** is crucial in Bayesian analysis as it encapsulates any existing knowledge about the parameter. In time series analysis, priors can be particularly useful when data is scarce or noisy.

When it comes to choosing priors, consider the following:

- **Informative priors**: Use when you have substantial prior knowledge. For example, if you know that the sales data from past years typically peaks around a certain season, you can incorporate this into your prior.
- **Non-informative priors**: Use when you have little prior knowledge, allowing the data to dominate the posterior distribution. A common non-informative prior is the uniform distribution, indicating that all values are equally likely a priori.

For example, in a seasonal time series model, you might use an informative prior for the seasonal component based on historical patterns. If past data shows a strong seasonal effect in December due to holidays, this prior knowledge can be built into your model.

Likelihood and posterior distributions

The likelihood function measures how well a statistical model explains the observed data for a given set of parameter values. It quantifies the probability of the observed data occurring under specific assumptions about the model parameters.

To move from a simple likelihood to a comprehensive understanding of your parameter estimates, you incorporate prior beliefs through **Bayes' theorem.** The theorem combines the likelihood of the observed data with prior knowledge or assumptions (the **prior distribution**) to form the **posterior distribution**. This posterior distribution reflects the updated beliefs about the parameters after observing the data.

For instance:

- The **likelihood** tells you how probable the observed data is, given a set of parameters
- The **prior distribution** encodes your initial beliefs or knowledge about the parameters before seeing the data
- The **posterior distribution** combines these two sources of information, providing a refined understanding of the parameters based on both prior knowledge and the observed evidence

The relationship between the likelihood, prior distribution, and posterior distribution is central to Bayesian inference, making Bayes' theorem an essential tool in this process.

Bayes' theorem

Bayes' theorem is the cornerstone of Bayesian inference. It provides a mathematical framework to update existing beliefs about a parameter in light of new data. The theorem combines our prior beliefs with the likelihood of the observed data to form a posterior distribution, which reflects the newly updated beliefs.

The theorem can be written as:

$$Posterior \propto Likelihood \times Prior$$

Let's break down each component to understand how they interact:

- **Prior distribution** ($P(\theta)$): The prior distribution represents your initial beliefs about the parameter (θ) before seeing any data. It encapsulates any existing knowledge or assumptions about the parameter.
  Example: If you are predicting sales and you believe that the average sales have typically been around 100 units, you might set a prior distribution around 100.
- **Likelihood** ($P(D|\theta)$): The likelihood is the probability of observing the data (D) given a specific value of the parameter (θ). It tells you how well your parameter value explains the observed data.
  Example: If your observed sales data closely matches a predicted average of 100 units, the likelihood of the data given that average will be high.
- **Posterior distribution** ($P(\theta|D)$): The posterior distribution is the updated belief about the parameter after incorporating the observed data. It combines the prior and the likelihood to give a more accurate estimate of the parameter.
  Example: After observing new sales data, your belief about the average sales might shift slightly from your initial guess, reflecting the new information.

Now that you have covered the basics of Bayes' theorem, it is time to dive deeper into how it actually impacts your beliefs in the real world. This is where the **Posterior distribution** comes in. It represents the updated belief about a parameter after accounting for the observed data, blending prior knowledge with new information to give you a more accurate estimate.

When constructing likelihood functions in time series analysis, the likelihood function depends on the model and the data. For example, if the assumption is that the data follows a normal distribution, the likelihood function can be written as:s

$$L(\theta|y) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i-\mu)^2}{2\sigma^2}\right)$$

Where y_i are the observed data points, and $\theta = (\mu, \sigma^2)$ are the parameters (mean and variance).

Suppose you are modeling monthly sales data with a normal distribution. Your prior suggests a mean sales of 100 units with a variance of 25. As you collect data over several months, you observe sales data points.

You use these observations to construct the likelihood function. By applying Bayes' Theorem, you combine the likelihood with your prior to get the posterior distribution, giving you an updated estimate of the mean and variance of sales.

Practical example in R

Let's implement a simple Bayesian analysis for a time series dataset using R. Assume you have monthly sales data and want to estimate the mean sales.

The following code fits a simple Bayesian regression model to a set of time series data using the `brms` package in R, which provides a user-friendly interface to Bayesian modeling. First, it ensures the necessary package is installed and loads it. Then, it generates some example time series data, representing monthly sales over a 12-month period. The code uses the `brm()` function to fit a Bayesian regression model where sales depend on time, assuming a Gaussian distribution for the data. The model is run with four chains of 2000 iterations each, including a warm-up period of 500 iterations. Finally, it summarizes the model, providing insights into the posterior distributions and other key parameters.

The following code demonstrates how to implement Bayesian regression using the `brms` package in R. It walks through generating a sample time series dataset, fitting a Bayesian regression model with a Gaussian distribution, and summarizing the posterior distributions.

1. First, install and load the `brms` package:

```
if (!requireNamespace("brms", quietly = TRUE)) {
  install.packages("brms")
}
library(brms)
```

2. Generate the data frame:

```
data <- data.frame(
  time = 1:12,
  sales = rnorm(12, 100, 10)  # Random example sales data
)
```

   The preceding code uses `rnorm()` to generate 12 random numbers following a normal distribution, with a mean of 100 and a standard deviation of 10. Then, assemble this into a data frame.

3. Fit a simple Bayesian regression model:

```
fit <- brm(
  sales ~ time,  # Formula for the time series
  data = data,
  family = gaussian(),  # You can choose different distributions if needed
  chains = 4, iter = 2000, warmup = 500  # Bayesian MCMC parameters
)
```

The preceding code block fits a simple Bayesian model. The formula (`sales ~ time`) is used to define a relationship between `sales` and `time`, with `time` as the independent variable and `sales` as the dependent variable. The family(`gaussian()`) assumes the dependent variable `sales` is continuous and normally distributed. The `chains/iter/warmup` specify the parameters for the model run. In this case, total iterations (2000) minus warmup (500) equals 1500 samples per chain. With 4 chains, the total draws available for inference are 4 x 1500 = 6000.

4. Summarize the model:

```
summary(fit)
```

The output of the Bayesian regression model provides detailed insights into the posterior distributions and key parameters. The summary includes information on the regression coefficients, their credible intervals, and diagnostic metrics such as Rhat and effective sample sizes, which help assess model convergence and reliability.

Note

Rhat is a convergence diagnostic used to assess whether multiple chains have converged to the same posterior distribution. Values close to 1.00 indicate good convergence, whereas higher values may suggest the chains have not fully converged and additional sampling or model adjustments maybe needed.

Figure 9.1 shows the summary output. Here, there is an Rhat value of 1.00:

```
 Family: gaussian
  Links: mu = identity; sigma = identity
Formula: sales ~ time
   Data: data (Number of observations: 12)
  Draws: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
         total post-warmup draws = 6000

Regression Coefficients:
          Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
Intercept   105.66      6.27    93.14   118.08 1.00     4077     3367
time         -1.20      0.86    -2.89     0.50 1.00     4049     3681

Further Distributional Parameters:
      Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
sigma    10.05      2.45     6.56    15.89 1.00     4109     4173

Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS
and Tail_ESS are effective sample size measures, and Rhat is the potential
scale reduction factor on split chains (at convergence, Rhat = 1).
```

Figure 9.1: Output summary of a Bayesian regression model

What you are seeing is the output summary of a Bayesian regression model using the `brms` package. Here is a breakdown:

- **Family**: Gaussian, meaning the model assumes normally distributed residuals.
- **Formula**: `sales ~ time`, indicating the model is predicting sales based on time.
- **Draws**: The model ran 4 chains with 2000 iterations each, totaling 6000 post-warm-up draws.
- The **intercept:** In a regression model always indicates the expected outcome (mean) when all predictors are zero. In your case:

$$sales \sim time \rightarrow intercept = expected\ sales\ when\ time = 0$$

  The credible interval says that, based on the data and prior, you are 95% confident that the TRUE baseline sales (when time = 0) range between 93.14 and 118.08.

- The coefficient for **time** is -1.20, meaning sales decrease by 1.20 units for each time unit, with a CI of [-2.89, 0.50].

- **sigma** represents the residual standard deviation, measuring the typical difference between the observed values and the model's predicted values. In this model, it is 10.05, indicating that the observed sales typically vary by about 10.05 units from the predicted sales. The **Rhat** values are 1.00, indicating good convergence of the model across chains.

Next, let's analyze the posterior distribution.

Analyzing posterior

In this code, you extract posterior samples from the fitted Bayesian model to visualize the distribution of the estimated mean sales. The posterior distribution represents your updated belief about the mean sales after considering the observed data and prior knowledge.

To better understand the posterior distribution generated by the Bayesian regression model, you can extract the posterior samples and visualize their distribution. The following code demonstrates how to extract the posterior samples of the mean sales and plot a histogram of the posterior distribution. This helps you understand how the model has updated your belief about the mean sales based on the observed data and prior information.

1. Extract the posterior samples:

```
posterior_samples <- as.data.frame(fit)
```

2. Plot the posterior distribution of the mean sales:

```
hist(posterior_samples$mu, breaks = 30, main = "Posterior Distribution of
Mean Sales", xlab = "Mean Sales")
abline(v = mean(posterior_samples$mu), col = "red", lwd = 2)
```

   The resulting plot illustrates the posterior distribution of the mean sales, derived from the posterior samples. The red vertical line marks the mean of the posterior samples, representing the updated estimate of the mean sales after applying the Bayesian model.

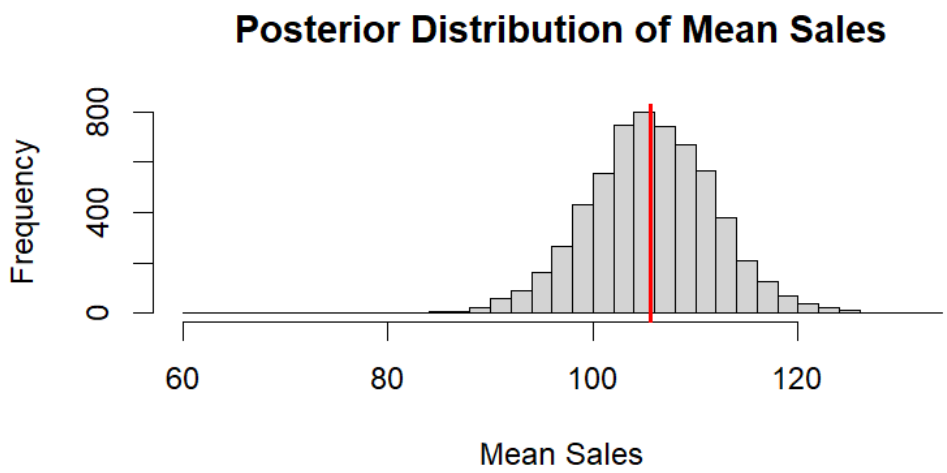


Figure 9.2: Histogram of the posterior distribution of mean sales.

You then plot a histogram of the posterior samples for the mean (Intercept) to observe its spread and central tendency. The red line in the plot marks the mean of these posterior samples, giving you a visual indication of the most likely value for the mean sales based on the model.

Bayesian inference in a time series context

When applying Bayesian inference to time series data, you need to account for the temporal dependencies inherent in such data. Bayesian methods offer a robust framework for incorporating these dependencies, allowing you to continuously update your models as new data becomes available.

Here are the key steps to do so:

1. **Define the model**: Specify a model that captures the temporal structure of the data, such as an ARIMA model, a state-space model, or a dynamic linear model.
2. **Choose priors**: Select prior distributions for the model parameters. These priors can be informed by historical data or expert knowledge.

3. **Construct the likelihood**: The likelihood function reflects how likely the observed data are given the parameters of the model.
4. **Compute the posterior**: Using Bayes' Theorem, combine the prior and the likelihood to form the posterior distribution, which provides updated estimates of the model parameters.

Here is an example. Consider a simple **autoregressive** (**AR**) model for time series data. The AR(1) model can be expressed as:

$$y_t = \phi y_{t-1} + \epsilon_t$$

Where y_t is the value at time t, ϕ is the autoregressive coefficient, and ϵ_t is the error term.

Using Bayesian methods, you define priors for ϕ and σ^2(the variance of the error term), construct the likelihood based on the observed data, and then compute the posterior distribution to update your estimates of ϕ and σ^2.

Second practical example in R

This code extends the Bayesian model by incorporating new observed data points to update the analysis. You first simulate 12 new data points, representing additional monthly sales data with a slightly higher mean than the original dataset. The model's data list is updated with these new observations, and the total number of data points (`N`) is adjusted accordingly.

Finally, the Bayesian regression model is refitted to include both the original and new data, and the updated results are printed. This process illustrates how Bayesian models can be dynamically updated as new data becomes available.

1. Generate new observed data:

```
new_y <- rnorm(12, mean = 115, sd = 10)  # New data points
```

2. Refit the model with the new data:

```
fit_updated <- brm(
  sales ~ time,
  data = data.frame(time = 1:length(new_y), sales = new_y),
  family = gaussian(),
```

```
  chains = 4, iter = 2000, warmup = 500
)
```

3. Print the updated results:

```
summary(fit_updated)
```

This gives you the following output:

```
 Family: gaussian
  Links: mu = identity
Formula: sales ~ time
   Data: data.frame(time = 1:length(new_y), sales = new_y) (Number of observations: 12)
  Draws: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
         total post-warmup draws = 6000

Regression Coefficients:
          Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
Intercept   111.15      4.84   101.44   120.82 1.00     5147     3618
time          0.23      0.67    -1.12     1.58 1.00     5341     3900

Further Distributional Parameters:
      Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
sigma     7.97      1.82     5.32    12.36 1.00     3973     3471

Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS
and Tail_ESS are effective sample size measures, and Rhat is the potential
scale reduction factor on split chains (at convergence, Rhat = 1).
```

Figure 9.3: Summary output of the Bayesian regression model

The updated model shows that the Intercept (mean sales when time is zero) is estimated at 111.15 with a credible interval of [101.44, 120.82]. The time coefficient is positive (0.23), indicating that sales slightly increase over time, with a 95% credible interval of [-1.12, 1.58].

The standard deviation of the residuals (sigma) is 7.97. The model has converged well, as indicated by Rhat values of 1.00 for all parameters, and the effective sample sizes (`Bulk_ESS` and `Tail_ESS`) are sufficiently large.

This process demonstrates how Bayesian inference allows you to refine your models continuously, ensuring they remain relevant and accurate as new data is incorporated.

Bayesian hierarchical models

Bayesian hierarchical models are particularly useful in time series analysis when dealing with multiple related time series or data with nested structures. These models allow you to share

information across different levels of the hierarchy, improving parameter estimates and forecasts.

- **Level 1**: Individual time series data, modeled with specific parameters.
- **Level 2**: Hyperparameters that govern the distribution of parameters at Level 1, allowing for information sharing and pooling.

Third practical example in R

This code fits a hierarchical Bayesian model using the `brms` package to analyze data from multiple groups (in this case, data from different stores). In hierarchical models, each group (store) has its own individual mean, but all group means are modeled as coming from a shared global distribution. This structure allows the model to "borrow strength" across groups, making predictions more robust, especially when there are fewer data points in some groups.

The `brms` function `brm()` is used to specify a random-intercept model (`y ~ (1 |store)`), where each store gets its own intercept (mean), but these intercepts are drawn from a common global distribution. This is a powerful approach for understanding variations within and between groups while incorporating uncertainty from all levels.

Let's go through the following example:

1. Generate the example data and combine it into a data frame:

```
set.seed(123)
N <- 50
J <- 5
store <- factor(sample(1:J, N, replace = TRUE))
mu_global <- 100
sigma_global <- 10
mu <- rnorm(J, mean = mu_global, sd = sigma_global)
y <- rnorm(N, mean = mu[as.numeric(store)], sd = 10)

data_hierarchical <- data.frame(store = store, y = y)
```

2. Next, fit the model with data. This will be a hierarchical Bayesian model with `brms`:

```
fit_hierarchical <- brm(
  y ~ (1 | store),  # Random intercept for each store
  data = data_hierarchical,
  family = gaussian(),
```

```
  chains = 4, iter = 2000, warmup = 500
)
```

3. Print the results:

```
summary(fit_hierarchical)
```

This hierarchical model allows you to estimate both store-specific parameters and global parameters, providing a comprehensive understanding of the sales patterns across different stores while sharing information between them.

```
 Family: gaussian
  Links: mu = identity
Formula: y ~ (1 | store)
   Data: data_hierarchical (Number of observations: 50)
  Draws: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
         total post-warmup draws = 6000

Multilevel Hyperparameters:
~store (Number of levels: 5)
              Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
sd(Intercept)     6.91      3.34     2.34    15.44 1.00     1751     2633

Regression Coefficients:
          Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
Intercept    99.44      3.27    92.55   105.76 1.00     1968     2404

Further Distributional Parameters:
      Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
sigma     9.14      0.99     7.41    11.29 1.00     4033     3492

Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS
and Tail_ESS are effective sample size measures, and Rhat is the potential
scale reduction factor on split chains (at convergence, Rhat = 1).
```

Figure 9.4: Summary output of a hierarchical Bayesian model

The hierarchical model estimates a global intercept of 99.44, indicating the overall average outcome across all stores, with credible intervals [92.55, 105.76]. The standard deviation of store-specific intercepts is 6.91, suggesting that the individual stores deviate moderately from the global mean, and the model has converged well with Rhat values of 1.00 for all parameters.

In summary, Bayesian inference in the context of time series offers a flexible and powerful approach to modeling and forecasting. By incorporating prior knowledge, continuously updating beliefs with new data, and using hierarchical models to share information across related time series, you can achieve more accurate and insightful analysis.

Now that you have a clear understanding of the benefits of hierarchical Bayesian models, particularly in time series, let's move on to constructing a Bayesian time series model from scratch. In the following section, you will walk through the essential steps, including defining the model structure, choosing priors, and estimating the posterior distribution.

Constructing a Bayesian model

Constructing a Bayesian time series model involves defining the structure and components of the model to appropriately capture the temporal dynamics of the data. The process includes specifying the model form, selecting priors, constructing the likelihood function, and using computational methods to estimate the posterior distribution.

Here are the steps to define a Bayesian time series model:

1. Choose a model that fits the nature of the time series data. Common choices include ARIMA, state-space models, or dynamic linear models.
   Example: For an ARIMA model, specify the order (p, d, q) that indicates the autoregressive order, differencing order, and moving average order, respectively.
2. Choose appropriate prior distributions for the model parameters based on prior knowledge or assumptions:
   Example: For the AR(1) model $y_t = \phi y_{t-1} + \epsilon_t$, you might use a normal prior for ϕ centered around 0 with a small variance, indicating a belief that the parameter is close to zero but allowing for some uncertainty.
3. Define the likelihood function based on the assumed distribution of the data given the parameters.
   Example: For normally distributed errors in an AR(1) model, the likelihood function would be the product of normal density functions for each observation.
4. Use computational methods such as Markov Chain Monte Carlo (MCMC) to sample from the posterior distribution.
   Example: Use software such as Stan or JAGS to perform MCMC sampling and obtain posterior estimates for the parameters.

Let's look at a practical example using R to illustrate this concept.

Fourth practical example in R

This code fits an autoregressive model of order 1 (`AR(1)`) using the `brms` package in R. In an AR(1) process, the value at each time step is dependent on the previous time step, plus some random noise. The code first generates 100 data points following this AR(1) process with a coefficient (`phi`) of 0.6 and a standard deviation of 1 for the noise. Using the `brm()` function,

you specify an `AR(1)` model and fit it to the data, allowing you to estimate the AR coefficient and error term via Bayesian inference. The results summarize the posterior distributions of the estimated parameters.

In practice, that unfolds in the following steps:

1. Set up the parameters:

```
set.seed(123)
N <- 100
phi <- 0.6
sigma <- 1
y <- numeric(N)
y[1] <- rnorm(1)
for (t in 2:N) {
  y[t] <- phi * y[t-1] + rnorm(1, sd = sigma)
}
```

2. Generate the data frame:

```
data_ar1 <- data.frame(time = 1:N, y = y)
```

3. Fit the AR(1) model with `brms`:

```
fit_ar1 <- brm(
  bf(y ~ ar(time = time, p = 1)),  # AR(1) structure with lag 1
  data = data_ar1,
  family = gaussian(),
  chains = 4, iter = 2000, warmup = 500
)
```

4. Print the results:

```
summary(fit_ar1)
```

Figure 9.5 is a summary output of a hierarchical Bayesian model. This output includes global and store-specific parameters, credible intervals, diagnostic metrics such as Rhat, and effective sample sizes, providing a detailed view of the model's performance and parameter estimates.

```
 Family: gaussian
  Links: mu = identity
Formula: y ~ ar(time = time, p = 1)
   Data: data_ar1 (Number of observations: 100)
  Draws: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
         total post-warmup draws = 6000

Correlation Structures:
      Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
ar[1]     0.57      0.09     0.40     0.75 1.00     4096     3604

Regression Coefficients:
          Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
Intercept     0.20      0.23    -0.25     0.64 1.00     3685     2764

Further Distributional Parameters:
      Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
sigma     0.93      0.07     0.81     1.07 1.00     4051     4114

Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS
and Tail_ESS are effective sample size measures, and Rhat is the potential
scale reduction factor on split chains (at convergence, Rhat = 1).
```

Figure 9.5: Summary output of an AR(1) Bayesian regression model

The AR(1) coefficient is estimated at 0.57, with a 95% credible interval of [0.40, 0.75], indicating a moderate level of autocorrelation in the time series data. The standard deviation of the residuals (sigma) is estimated at 0.93, suggesting some variation around the AR(1) process. The model has converged well, with Rhat values of 1.00 for all parameters, ensuring reliable estimates.

Choosing priors for time series models

Choosing the right priors is crucial in Bayesian modeling. Priors can significantly influence the posterior distribution, especially when data are limited. The choice of priors should reflect any prior knowledge or reasonable assumptions about the parameters.

Guidance on selecting priors:

- **Informative priors**: Use when substantial prior knowledge exists.
  Example: If you have strong evidence that the seasonal effect in your time series is pronounced, you can use an informative prior for the seasonal component.
- **Non-informative priors**: Use when little prior knowledge is available.
  Example: For a new product with no historical sales data, you might use a uniform prior, indicating all parameter values are equally likely.

- **Weakly informative priors**: Balance between informative and non-informative priors. *Example*: A normal prior centered at zero with a large variance can be used for regression coefficients when you believe the coefficients could be large but are unsure of their exact values.

Choosing priors in R

This code fits an AR(1) model with weakly informative priors using the `brms` package. Weakly informative priors are used to provide some structure to the model without being too restrictive, ensuring the model can still learn from the data. In this case, the AR(1) coefficient (`phi`) is given a normal prior with a mean of 0 and standard deviation of 1, while the standard deviation of the residuals (`sigma`) has a more flexible Cauchy prior with a mean of 0 and scale of 5. The model is then fit using the data generated from an AR(1) process, and the results summarize the posterior distributions of the estimated parameters.

Please note that this example follows a similar workflow to the previous AR(1) model implementation, including generating an AR(1) time series, fitting the model using brms, and examining the posterior estimates. The key difference is that this example explicitly introduces a weakly informative prior for the residual standard deviation (σ), by specifying prior assumptions, it demonstrates an important aspect of Bayesian modeling: prior knowledge or reasonable constraints can be incorporated into the model before observing the data, and these assumptions are then updated through the posterior distribution.

1. Set up the parameters:

```
set.seed(123)
N <- 100
phi <- 0.6
sigma <- 1
y <- numeric(N)
y[1] <- rnorm(1)
for (t in 2:N) {
  y[t] <- phi * y[t-1] + rnorm(1, sd = sigma)
}
```

2. Create the data frame:

```
data_ar1 <- data.frame(time = 1:N, y = y)
```

3. Define a weakly informative prior for sigma. This is the part that is different from previous examples:

```
priors <- c(
  prior(cauchy(0, 5), class = "sigma")  # Weakly informative prior for
sigma
)
```

4. Fit the AR(1) model with the prior using `brms`:

```
fit_weakly_informative <- brm(
  bf(y ~ ar(time = time, p = 1)),  # AR(1) structure with lag 1
  data = data_ar1,
  family = gaussian(),
  prior = priors,  # Using the weakly informative prior for sigma
  chains = 4, iter = 2000, warmup = 500
)
```

5. Print the results:

```
summary(fit_weakly_informative)
```

This output presents the results of fitting an AR(1) model with a weakly informative prior on the residual standard deviation (σ). The weakly informative prior provides a balance between incorporating prior beliefs and allowing the data to influence the results, ensuring robust parameter estimates.

```
 Family: gaussian
  Links: mu = identity
Formula: y ~ ar(time = time, p = 1)
   Data: data_ar1 (Number of observations: 100)
  Draws: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
         total post-warmup draws = 6000

Correlation Structures:
      Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
ar[1]     0.57      0.09     0.41     0.74 1.00     4527     4139

Regression Coefficients:
          Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
Intercept     0.20      0.23    -0.26     0.64 1.00     4969     3575

Further Distributional Parameters:
      Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
sigma     0.93      0.07     0.81     1.07 1.00     4518     3779

Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS
and Tail_ESS are effective sample size measures, and Rhat is the potential
scale reduction factor on split chains (at convergence, Rhat = 1).
```

Figure 9.6: Summary of an AR(1) Bayesian regression model with a weakly informative prior

The AR(1) coefficient is estimated at 0.57 with a 95% credible interval of [0.41, 0.74], indicating a moderate degree of autocorrelation in the data. The standard deviation of the residuals (sigma) is 0.93, suggesting the noise around the AR(1) process is relatively small. All parameters have converged well, as indicated by the Rhat values of 1.00, and the effective sample sizes (`Bulk_ESS` and `Tail_ESS`) are large, ensuring the model's reliability.

Next, I will cover how to evaluate your Bayesian model.

Model specification and diagnostics

Correctly specifying the model and performing diagnostics are crucial to ensuring the model's validity and reliability. Mis-specification can lead to biased estimates and poor predictive performance.

Steps for model specification:

1. Clearly define the model structure, including the priors, likelihood, and parameters.
2. Ensure the model captures the key characteristics of the time series data, such as seasonality or trend.

3. Use diagnostic tools to assess the model fit and identify any issues, such as residual plots, posterior predictive checks, and convergence diagnostics.
4. Compare model performance using metrics such as the **deviance information criterion** (**DIC**) or the **Watanabe-Akaike information criterion** (**WAIC**), sometimes known as the **widely applicable information criterion**. For example, assess out-of-sample predictive accuracy to ensure the model generalizes well to new data.

Let's explore this further with a practical example.

Practical example of model specifications and diagnostics in R

This code performs essential diagnostic checks and model evaluation for a fitted AR(1) model using Bayesian methods in `brms`. First, it conducts **posterior predictive checks** by drawing predictions from the posterior distribution and plotting them against the observed data, which helps assess how well the model fits the data. The plot overlays multiple predicted time series on the observed values, showing the model's uncertainty. Next, **convergence diagnostics** are performed using trace plots to ensure the Markov chains mixed well and reached convergence during sampling.

1. Install and load packages:

```
packages <- c("loo")

for (p in packages) {
 if (!requireNamespace(p, quietly = TRUE)) {
 install.packages(p)
}
library(p, character.only = TRUE)
}
```

2. Posterior predictive checks:

```
posterior_predictive <- posterior_predict(fit_ar1)
```

3. Plot the posterior distribution:

```
plot(y, type = "l", col = "blue", main = "Posterior Predictive
Check", xlab = "Time", ylab = "Observed vs Predicted")
for (i in 1:100) {
  lines(1:N, posterior_predictive[i, ], col = adjustcolor("red", alpha.f =
0.1))
```

```
}
lines(1:N, y, col = "blue", lwd = 2)
```

Posterior predictive checks are an essential diagnostic tool for Bayesian models. They compare the observed data with data simulated from the posterior predictive distribution to assess how well the model captures the underlying structure of the data. The following plot visualizes this comparison for the AR(1) model:

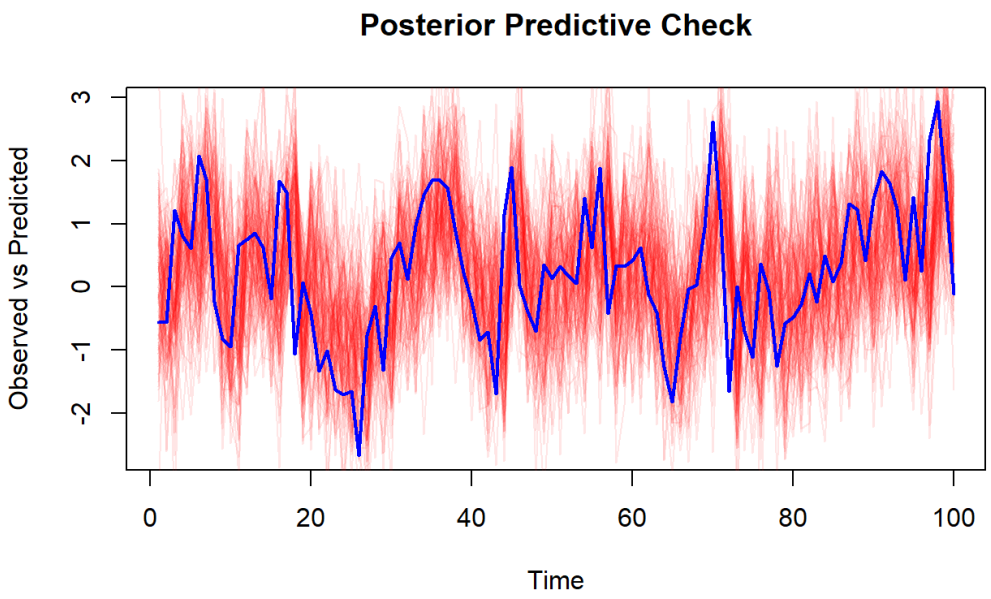


Figure 9.7: Posterior predictive check for the AR(1) model

This chart shows a **posterior predictive check**, where the observed data (blue line) is compared with 100 posterior predictive samples (red lines), indicating that the model captures the overall trend and variability in the time series, with some uncertainty around individual predictions.

4. Finally, let's calculate the WAIC, a measure used to compare the predictive accuracy of the model, helping evaluate how well it generalizes to new data:

```
# Model comparison using WAIC
waic_fit <- waic(fit_ar1)
print(waic_fit)
```

```
Computed from 6000 by 100 log-likelihood matrix.

          Estimate   SE
elpd_waic   -135.3  6.8
p_waic         2.9  0.5
waic         270.7 13.6
```

The WAIC is a commonly used metric in Bayesian modeling to evaluate the predictive accuracy of a model. It balances model complexity and goodness of fit, helping determine how well the model generalizes to new data.

The WAIC value is 270.7, with an effective number of parameters (`p_waic`) of 2.9, indicating the model's goodness of fit and complexity, computed from a 6000 by 100 log-likelihood matrix.

Defining, constructing, and diagnosing Bayesian time series models involves careful specification of the model form, thoughtful selection of priors, and rigorous model evaluation. By following these steps, you can build robust Bayesian models that accurately capture the temporal dynamics of your data and provide reliable forecasts.

Following this introduction, you will delve into model selection for time series data, covering common models such as ARIMA and their Bayesian adaptations to help you choose the right structure for your forecasting tasks.

Implementing Bayesian models in time series

As you dive into the practical aspects of Bayesian time series analysis, it is crucial to understand how to effectively implement these models to harness their full potential. Bayesian models offer a flexible framework that incorporates prior knowledge and updates beliefs with new data, making them particularly powerful for time series forecasting. The implementation process involves defining the model structure, selecting appropriate priors, constructing the likelihood function, and using computational methods to estimate the posterior distribution.

Model selection for time series data

You will start with defining and constructing Bayesian models. Along the way, you will provide practical examples and tips to ensure you can confidently apply these techniques to your data. Let's get started on this journey into the world of Bayesian time series modeling.

AutoRegressive Integrated Moving Average (**ARIMA**) models, which you also covered in *Chapter 7*, are widely used for time series forecasting. In a Bayesian framework, ARIMA models can be extended by treating model parameters as probability distributions rather than fixed values. Instead of obtaining single point estimates, Bayesian ARIMA models estimate posterior distributions for parameters by combining prior assumptions with the observed data. For

example, consider monthly sales data that follows an ARIMA(1,1,1) structure. The model structure is defined by the ARIMA components, while Bayesian inference allows prior distributions to be assigned to parameters such as the autoregressive coefficient, moving average coefficient, and error variance. The observed data is then used to update these priors, producing posterior distributions that capture both prior knowledge and uncertainty in the parameter estimates.

State-space models provide a flexible structure for time series data by incorporating latent (unobserved) variables that evolve over time. These models are particularly useful for handling missing data, non-stationary processes, and complex dynamics.

- A state-space model consists of two equations: the observation equation, which relates the observed data to the latent state, and the state equation, which describes the evolution of the latent state over time
- Bayesian state-space models specify priors for the parameters of both equations and use data to update these priors

In tracking the performance of a stock price, you might use a state-space model where the observed stock price depends on an underlying latent value influenced by market trends and external factors. Priors on these latent variables and their evolution rules can be updated as new price data comes in, providing a dynamic and robust forecast.

Dynamic linear models (DLM) are a subclass of state-space models that are particularly well-suited for time series with complex patterns that change over time. DLMs are used to model various components such as trend, seasonality, and regression effects in a unified framework.

- DLMs represent the observed time series as a linear combination of different components that evolve over time according to specified dynamics
- In a Bayesian DLM, priors are set for the parameters governing these components, and the posterior distributions are updated with observed data

For a time series with a strong seasonal component, such as monthly sales data that peaks during the holiday season, a DLM can be used to model both the overall trend and the seasonal effects. Priors on the seasonal component can be informed by historical data, and Bayesian updating allows these priors to be refined as new sales data is observed.

Criteria for model selection

Selecting the appropriate Bayesian time series model starts with assessing the characteristics of your data. Consider aspects such as stationarity, seasonality, and trend.

- **Stationarity**: Determine whether the mean and variance of your time series are constant over time. Non-stationary data may require differencing or more complex models such as state-space models to capture the underlying dynamics.

- **Seasonality**: Identify any regular, repeating patterns within your data. Seasonal components can be explicitly modeled using DLMs or state-space models.
- **Trend**: Assess whether there is a long-term upward or downward movement in your data. Trends can be modeled within both Bayesian ARIMA and state-space frameworks.

Balance the complexity of the model with the amount of data available. More complex models, while potentially more accurate, require more data to estimate parameters reliably.

- **Avoiding overfitting**: Overfitting occurs when a model becomes too tailored to the training data, capturing noise rather than the underlying process. Bayesian models can mitigate overfitting through regularization via priors, but it is still important to choose a model complexity appropriate for your data size.
- **Parsimony principle**: Choose the simplest model that adequately captures the dynamics of your time series. This principle helps in making your model more interpretable and less prone to overfitting.

If you have a short time series with a clear seasonal pattern but no strong trend, a simpler Bayesian seasonal ARIMA model might be more appropriate than a complex state-space model with multiple latent states.

By understanding these types of Bayesian time series models and the criteria for their selection, you can better match your model to the characteristics and needs of your specific data, leading to more accurate and reliable forecasts.

Specifying model components

The foundation of any Bayesian time series model lies in its parameters. These parameters define the behavior and characteristics of the model and are crucial for capturing the underlying dynamics of the time series data.

- **Autoregressive coefficients (AR)**: In models like ARIMA, the autoregressive coefficients represent the relationship between an observation and its previous values. For example, in an AR(1) model, the parameter ϕ determines how much of the current value is influenced by the immediately preceding value.
- **Moving average coefficients (MA)**: These coefficients capture the relationship between an observation and past error terms. In an MA(1) model, the parameter θ indicates how past forecast errors affect the current observation.
- **Variance components**: Variance parameters, such as σ^2, are essential for modeling the uncertainty in the data. They define the variance of the error terms in the model, indicating the degree of variability not explained by the model.

Hyperparameters are higher-level parameters that control the behavior of other model parameters or components. They are not directly estimated from the data but are often set based on prior knowledge or through a tuning process.

Practical example in R

Here is how you might specify these components in a Bayesian time series model using R and **Stan**. Stan is a probabilistic programming language designed for Bayesian statistical modeling. It serves as the computational engine behind R packages such as `brms`, performing the calculations required to estimate posterior distributions from the specified model.

The previous examples introduced Bayesian time series modeling using an AR(1) structure and demonstrated how weakly informative priors can be incorporated into the model. This example extends the previous workflow by introducing additional model inputs: lagged observations, lagged error terms, and a time trend.

The key difference is that, instead of modeling the response using only its previous value, this example includes multiple components to capture different patterns in the time series. The lagged observation (`lag_y`) represents autoregressive behavior, the lagged error term (`lag_error`) captures short-term unexplained variation, and the time variable (`time`) models an underlying trend.

By including these additional predictors, the Bayesian model can estimate the contribution of each component through posterior distributions while still incorporating prior assumptions about the parameters.

1. Set up the parameters for an ARMA(1,1) model with a trend:

```
set.seed(123)
N <- 100
phi <- 0.6
theta <- 0.4
sigma <- 1
trend <- 0.1
y <- numeric(N)
y[1] <- rnorm(1)
for (t in 2:N) {
  y[t] <- phi * y[t-1] + theta * (y[t-1] - trend) + rnorm(1, sd = sigma)
}
```

2. Create the data frame:

```
data_arma <- data.frame(time = 1:N, y = y)
```

3. Create lagged variables and remove the first row (NA). These are new components in the example:

```
data_arma$lag_y <- c(NA, head(data_arma$y, -1))  # AR(1) lagged term
data_arma$lag_error <- c(NA, diff(data_arma$y))  # MA(1) lagged error term

data_arma <- data_arma[-1,]
```

4. Define weakly informative priors:

```
priors <- c(
  prior(normal(0, 0.5), class = "b", coef = "lag_y"),     # Prior for
AR(1) coefficient
  prior(normal(0, 0.5), class = "b", coef = "lag_error"), # Prior for
MA(1) coefficient
  prior(normal(0, 1), class = "sigma"),                   # Prior for
standard deviation
  prior(normal(0, 1), class = "b", coef = "time")         # Prior for trend
)
```

5. Fit the model using `brms`:

```
fit_arma <- brm(
  bf(y ~ time + lag_y + lag_error),  # ARMA(1,1) with trend modeled using
lagged terms
  data = data_arma,
  family = gaussian(),
  prior = priors,
  chains = 4, iter = 2000, warmup = 500
)
```

6. Print the model summary:

```
summary(fit_arma)
```

This code demonstrates how to fit a Bayesian ARMA(1,1) model with a trend component using the `brms` package. `brms` simplifies the process of building Bayesian models by allowing you to specify the model formula and priors directly in R, with the heavy lifting done in the background by Stan. In this model, you include autoregressive (AR) and moving-average (MA) components to capture the time-dependent structure and a trend component to account for

any linear drift over time. You will generate synthetic data from an ARMA(1,1) process with a trend and then fit the model using `brms`.

```
 Family: gaussian
  Links: mu = identity; sigma = identity
Formula: y ~ time + lag_y + lag_error
   Data: data_arma (Number of observations: 99)
  Draws: 4 chains, each with iter = 2000; warmup = 500; thin = 1;
         total post-warmup draws = 6000

Regression Coefficients:
          Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
Intercept     0.00      0.00    -0.00     0.00 1.02     2769     1288
time         -0.00      0.00    -0.00     0.00 1.03     5132     2293
lag_y         1.00      0.00     1.00     1.00 1.03      823      733
lag_error     1.00      0.00     1.00     1.00 1.03      693      960

Further Distributional Parameters:
      Estimate Est.Error l-95% CI u-95% CI Rhat Bulk_ESS Tail_ESS
sigma     0.00      0.00     0.00     0.00 2.02        6       29

Draws were sampled using sampling(NUTS). For each parameter, Bulk_ESS
and Tail_ESS are effective sample size measures, and Rhat is the potential
scale reduction factor on split chains (at convergence, Rhat = 1).
Warning messages:
1: Parts of the model have not converged (some Rhats are > 1.05). Be careful when analysing the results!
We recommend running more iterations and/or setting stronger priors.
2: There were 1783 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help. See ht
tp://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
```

Figure 9.8: Summary output of a Bayesian ARIMA model

The model estimates for the AR(1) coefficient (`lag_y`) and the MA(1) coefficient (`lag_error`) are both exactly 1, which suggests a problem with model convergence. Several warnings indicate that some parameters, including `sigma`, have not converged properly, with Rhat values exceeding 1.05 and very low **effective sample sizes** (**ESS**). The presence of 1783 divergent transitions after warmup suggests that adjustments such as increasing `adapt_delta` or setting stronger priors are needed to stabilize the model.

This example specifies a simple Bayesian ARIMA model with an autoregressive term, a moving average term, a trend component, and a variance parameter. The priors are defined based on reasonable assumptions, and the model is fitted to the data using Stan. This process can be adapted to more complex models by incorporating additional components and dependencies as needed.

Model validation and performance evaluation

To validate your Bayesian time series model, generating and analyzing the posterior predictive distribution is essential. This distribution provides a way to compare the model's predictions

with the observed data, helping to assess the model's fit. Here is how to generate the posterior predictive distribution:

1. You will reuse the setup from the previous example, using the same AR(1) lagged terms and data set. The following code snippets will skip the parameter setup, data frame creation, and lagged variable creation.
2. Fit the model with lagged terms:

```
fit_arma <- brm(
  bf(y ~ time + lag_y + lag_error),  # ARMA(1,1) with trend modeled using
lagged terms
  data = data_arma,
  family = gaussian(),
  chains = 4, iter = 2000, warmup = 500
)
```

3. Generate posterior predictive distribution:

```
posterior_predictive <- posterior_predict(fit_arma)
```

4. Plot the posterior predictive distribution:

```
plot(data_arma$y, type = "l", col = "blue", main = "Posterior Predictive
Check", xlab = "Time", ylab = "Observed vs Predicted")
for (i in 1:100) {
  lines(1:nrow(data_arma), posterior_predictive[i, ], col =
adjustcolor("red", alpha.f = 0.1))
}
lines(1:nrow(data_arma), data_arma$y, col = "blue", lwd = 2)
```

Visual checks, such as residual plots and predictive intervals, provide intuitive ways to evaluate model performance.

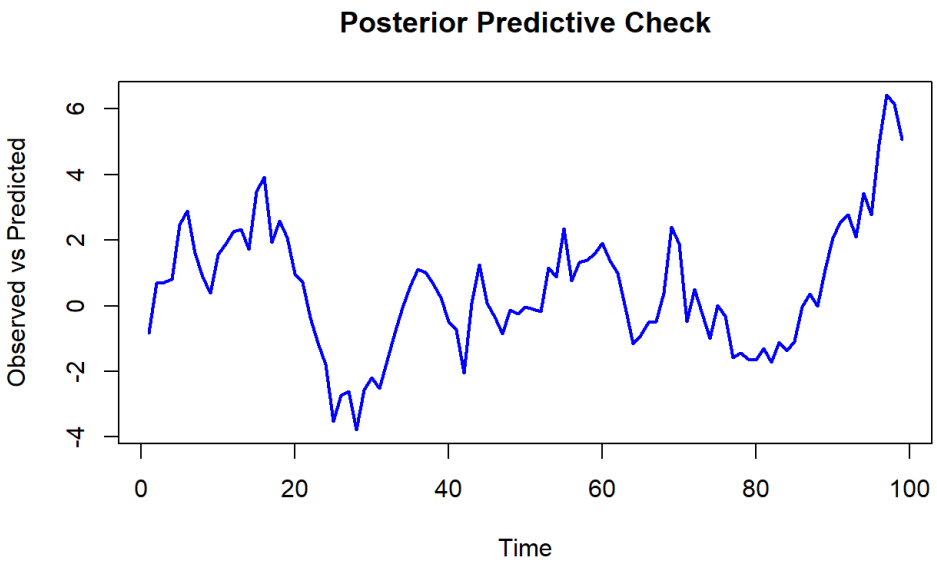


Figure 9.9: Posterior predictive accuracy check

The plot displays a **posterior predictive check**, showing how the model's predictions (in red) align with the observed data (in blue). The model seems to capture the overall trend of the observed data, but the lack of red lines suggests that the posterior predictive samples are overlaid with the observed data due to the scale of the diagram. You will use the residual plots to display the difference between them.

Residual plots

Plot the residuals (differences between observed and predicted values) to check for patterns. Ideally, residuals should be randomly distributed with no discernible pattern.

```
residuals <- y - apply(posterior_predictive, 2, mean)
plot(residuals, type = 'h', main = 'Residuals', xlab = 'Time', ylab = 'Residuals')
abline(h = 0, col = 'red')
```

Residual plots are a crucial diagnostic tool for assessing the performance of a model. They help identify patterns, biases, or deviations in the residuals (differences between observed and predicted values) that might indicate model misfit or underlying issues.

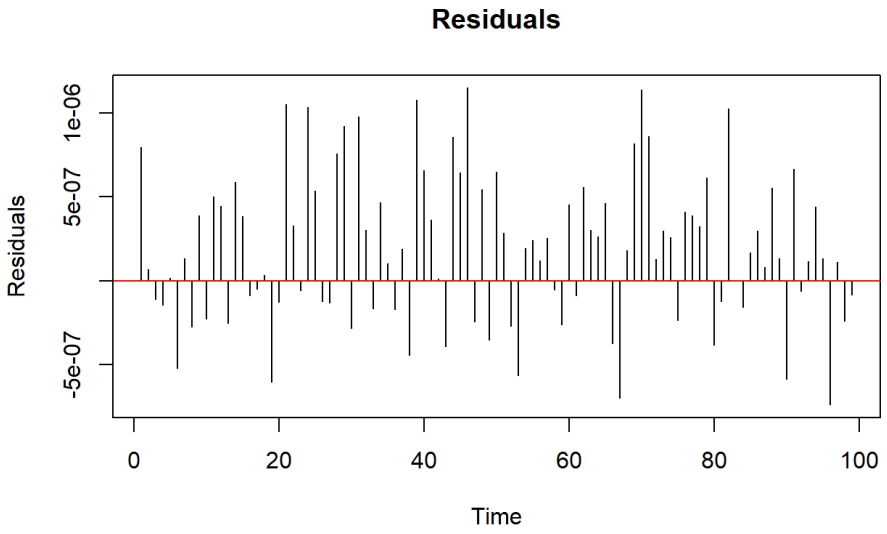


Figure 9.10: Residual plot for the fitted model

This residual plot shows the difference between the observed data and the model's predicted values over time. The vertical lines represent the residuals at each time point, with positive values above the red line (indicating underestimation by the model) and negative values below the red line (indicating overestimation). The residuals are quite small (in the range of 1e-07), suggesting that the model is closely fitting the data, though some variability remains.

Sensitivity analysis

When building Bayesian models, it is essential to evaluate how sensitive your model is to various assumptions and conditions. Sensitivity analysis and robustness checks help ensure that your model provides reliable and stable predictions, even when faced with changes in priors or data subsets.

Sensitivity analysis focuses on assessing the influence of different prior choices on the model's posterior estimates and overall performance. By comparing models with weak versus strong priors, you can evaluate how robust the model is to prior assumptions. This step is critical in Bayesian analysis because the choice of priors can significantly impact the results, especially when data is sparse or uncertain.

In this example, you use the same time series data, model structure, and lagged variables from the previous example. The key difference is that you modify the prior distribution assigned to the trend coefficient while keeping all other aspects of the model unchanged. By fitting the

same Bayesian model with different prior specifications, we can examine whether the posterior estimates and predictive performance are affected by the choice of priors.

The first model uses a weakly informative prior, which allows the trend coefficient to vary within a wider range. The second model uses a more informative prior with a narrower distribution, placing stronger assumptions on the possible values of the coefficient. Comparing these models helps demonstrate how prior assumptions influence Bayesian inference, especially when limited data are available.

The following code focuses only on fitting the models with different prior specifications and comparing their predictive performance. The data generation, parameter setup, and lagged variable creation steps are omitted because they are identical to those in the previous example. If you are running this example independently, complete those preparation steps before executing the following code:

1. Fit the model with a weak prior:

```
fit_weak_prior <- brm(
  bf(y ~ time + lag_y + lag_error),
  data = data_arma,
  family = gaussian(),
  prior = prior(normal(0, 1), class = "b", coef = "time"),  # Weak prior
  chains = 4, iter = 2000, warmup = 500
)
```

2. Fit the model with a strong prior:

```
fit_strong_prior <- brm(
  bf(y ~ time + lag_y + lag_error),
  data = data_arma,
  family = gaussian(),
  prior = prior(normal(0, 0.1), class = "b", coef = "time"),  # Strong
prior
  chains = 4, iter = 2000, warmup = 500
)
```

3. Compute LOO for each model:

```
loo_weak_prior <- loo(fit_weak_prior)
loo_strong_prior <- loo(fit_strong_prior)
```

4. Compare the models using LOO:

```
loo_compare(loo_weak_prior, loo_strong_prior)

                elpd_diff se_diff
fit_strong_prior  0.0       0.0
fit_weak_prior   -1.4       0.0
```

The **leave-one-out (LOO) cross-validation** method is used here to compare the performance of models fitted with different priors. LOO provides an estimate of the model's predictive accuracy by computing expected log predictive density (**elpd**). The comparison helps evaluate how the choice of priors affects the model's predictive performance.

The comparison shows that the model with the *strong prior* performed slightly better (higher expected log pointwise predictive density, or `elpd_diff` = 0) than the model with the *weak prior* (`elpd_diff` = -1.4), although the difference is minimal given the small `se_diff` (standard error).

By conducting thorough model validation and performance evaluation, you can ensure that your Bayesian time series models are accurate, reliable, and robust. These steps help in identifying the best model for your data, providing more trustworthy and actionable forecasts.

Summary

By now, you should have a solid understanding of how Bayesian methods offer a fresh perspective compared to traditional approaches. These techniques allow you to incorporate prior knowledge and continuously update your models as new data becomes available, making them especially powerful for time series forecasting.

You started with the foundations of Bayesian inference, discussing the key concepts of prior, likelihood, and posterior distributions. You then dove into practical implementation, showing you how to set up Bayesian models using R and Stan, select appropriate priors, and construct likelihood functions tailored to time series data. You have seen how to handle missing data and how to apply Bayesian inference in the context of time series, emphasizing the importance of updating beliefs with new data and leveraging hierarchical models.

To ensure that your models are robust and reliable, you covered essential steps for model validation and performance evaluation. From generating posterior predictive distributions and conducting visual checks to comparing models using information criteria and cross-validation, you have learned how to assess and refine your Bayesian time series models. Sensitivity analysis and robustness checks further highlighted the importance of testing your models under various conditions.

By mastering these techniques, you can now confidently apply Bayesian methods to your time series data, enhancing your ability to make accurate and insightful forecasts. Remember, the flexibility and power of Bayesian analysis lie in its capacity to adapt to new information and to incorporate both prior knowledge and observed data.

In the next chapter, you will learn about multi-series forecasting, a crucial technique when managing interconnected time series such as sales data from multiple retail stores or financial portfolios. Multi-series forecasting allows you to capture the nuances of each series while leveraging their relationships for better predictions. This chapter will prepare you to navigate the intricacies of forecasting multiple time series with greater accuracy and insight, providing valuable tools for both strategic and operational planning.

10

Multi-Series Forecasting Techniques

Imagine you are managing forecasts for several retail stores, each with its own unique trends and seasonality. Unlike single-series forecasting, where you focus on one data stream at a time, multi-series forecasting helps you handle multiple interconnected series. This approach provides deeper insights, especially in complex scenarios such as regional sales analysis, supply chain management, or financial portfolios.

In this chapter, you will explore different techniques for managing and forecasting multiple time series. You will start by understanding what multi-series forecasting is and why it is important. Then, you will look at some practical methods such as hierarchical and grouped forecasting. To make these concepts more relatable, you will examine real-world case studies. Finally, you will touch on some advanced challenges you might encounter and discuss how to tackle them.

You will explore the following headings in the chapter:

- Introducing multi-series forecasting
- Hierarchical forecasting methods
- Grouped time series analysis

Let's dive in and see how mastering these techniques can give you a powerful toolset for analyzing and forecasting complex datasets.

Technical requirements

R and RStudio are required for this chapter. GitHub scripts can be found in the repository.

You will need the following data file and packages for this chapter:

- `forecast`: The `forecast` package in R is a very popular set of tools designed for analyzing and forecasting time series data. You will see code examples in this chapter that show how to use this package to forecast hierarchical time series.
- `hts`: The `hts` (Hierarchical and Grouped Time Series) package provides methods for analyzing and forecasting hierarchical and grouped time series.
- `infantgts`: The `infantgts` dataset is a built-in dataset available in the `hts` package. It contains regional infant mortality counts across Australia from 1933 to 2003. The data is broken down by state (NSW, VIC, and so on) and sex (male and female).

Introducing multi-series forecasting

In this section, you will discover what multi-series forecasting is and why it is a crucial skill in the data analyst's toolkit. Unlike forecasting a single time series, multi-series forecasting involves predicting several interrelated data series, such as managing forecasts for different stores in a retail chain, analyzing product sales across various regions, or tracking multiple financial assets.

Multi-series forecasting often leverages the relationships between different series to enhance the accuracy and depth of insights. This involves analyzing multiple time series that are related to or influenced by each other. For instance, in a retail context, the sales data for various products may be interconnected due to shared factors such as seasonal trends or promotional activities. Recognizing these relationships allows for a more comprehensive understanding of the data.

Data can often be aggregated at different levels, such as individual store sales versus regional or national sales. Multi-series forecasting techniques help in understanding and predicting data at various aggregation levels, providing both detailed and overarching views of the trends.

By considering the interactions between multiple series, forecasts can be more accurate than those made for individual series in isolation. This holistic approach can capture more nuanced patterns and provide better insights, especially in complex scenarios where different factors are interrelated.

As organizations grow, the complexity and volume of data increase. Multi-series forecasting techniques are essential for scaling forecasting processes, enabling analysts to manage and interpret large datasets efficiently.

The principles of multi-series forecasting are applicable across a wide range of industries. In finance, it can be used to predict stock prices or economic indicators. In retail, it helps in managing stock levels and understanding customer demand across different locations. The healthcare sector uses it to predict patient numbers and resource needs.

Understanding these key concepts and appreciating the importance of multi-series forecasting sets the foundation for the specific methodologies and applications discussed in the following sections.

Use cases of multi-series forecasting across industries

This section explores practical example use cases that highlight the value of forecasting multiple interconnected time series:

- **Retail: Inventory and sales forecasts**: In the retail industry, accurately forecasting inventory and sales is crucial for optimizing stock levels, reducing waste, and maximizing profits. Retailers often deal with multiple products, each with unique demand patterns influenced by factors such as seasonality, promotions, and regional preferences. Multi-series forecasting helps retailers when:
    - **Predicting demand across product lines**: By analyzing sales data across various products and regions, retailers can identify trends and anticipate demand shifts, ensuring they stock the right products in the right quantities.
    - **Optimizing inventory levels**: Forecasting at different aggregation levels (for example, store, regional, and national) helps in maintaining optimal inventory, reducing overstock and stockouts.
    - **Planning promotions and discounts**: Understanding how different products' sales are interrelated allows retailers to design effective promotional strategies that drive overall sales.
- **Finance: Predicting asset prices and economic indicators**: The finance industry uses multi-series forecasting to analyze and predict the behavior of various financial instruments and economic indicators. The use cases include:
    - **Stock market analysis**: Analysts use multi-series forecasting to understand the relationships between different stocks, sectors, or indices. This helps in identifying trends, correlations, and potential investment opportunities.
    - **Economic forecasting**: Governments and financial institutions rely on multi-series forecasting to predict economic indicators such as GDP growth,

unemployment rates, and inflation. By analyzing multiple related time series, they can make more accurate economic forecasts and policy decisions.

- **Risk management**: Recognizing how financial assets are interconnected (for example, they may belong to the same asset type, the same industry sector, and so on) improves the ability to assess and manage risk, which is essential for effective portfolio optimization and stress testing.

- **Supply chain and logistics**: In supply chain management, forecasting is essential for efficient resource allocation and demand planning. Multi-series forecasting aids in:
  - **Demand planning**: By forecasting demand for different products across multiple locations, companies can better plan their manufacturing and distribution processes, ensuring timely delivery and reducing costs.
  - **Resource allocation**: Accurate forecasting helps in allocating resources such as labor, raw materials, and transportation more effectively, aligning supply chain operations with market demand.
  - **Seasonal and promotional adjustments**: Supply chains can adapt to seasonal changes and promotional activities by predicting their impact on demand, thus maintaining service levels and minimizing disruptions.
- **Healthcare patient flow and resource management**: In healthcare, forecasting helps in managing patient flow and optimizing resource allocation. Multi-series forecasting is particularly useful for:
  - **Predicting patient admissions**: Hospitals use forecasting to anticipate patient admissions based on historical data, allowing them to allocate resources like beds, staff, and medical supplies efficiently.
  - **Managing seasonal variability**: Understanding patient admission patterns related to seasons or outbreaks (such as flu season) helps in better preparedness and resource management.
  - **Budget and staffing planning**: Accurate forecasts support budgeting and staffing decisions, ensuring that healthcare facilities can meet patient needs without overextending resources.

These examples illustrate the broad applicability and value of multi-series forecasting across different sectors. By leveraging these techniques, organizations can enhance their decision-making processes, optimize operations, and ultimately achieve better outcomes in their respective fields.

Challenges in multi-series forecasting

In this section, you will assess some common challenges faced when implementing multi-series forecasting. Topics will include data management complexities, handling different data qualities across series, and the computational demands of processing large datasets. Understanding these challenges is crucial for effectively applying the techniques discussed later in the chapter.

While multi-series forecasting offers significant benefits, it also presents several challenges that can complicate its implementation and effectiveness. Understanding these challenges is crucial for successfully applying multi-series forecasting techniques. In this section, you will explore some of the most common issues practitioners face.

Data management complexities

Multi-series forecasting often involves dealing with large volumes of data. Managing, storing, and processing this data requires robust data infrastructure and efficient data-handling practices. The complexity increases with the number of series and the frequency of data points, necessitating scalable data management solutions.

Integrating data from different sources or systems can lead to inconsistencies, such as varying data formats, missing values, or time mismatches. Ensuring data consistency across all series is essential for accurate forecasting. This challenge requires careful data cleaning, normalization, and synchronization (the process of ensuring data across multiple systems, devices, or databases remains consistent, accurate and up to date) processes.

Varying data quality across series

Different time series may have varying levels of data quality. Some series may have missing data points, irregular intervals, or outliers. Inconsistent data quality can lead to biased forecasts if not properly addressed. Techniques such as data imputation, outlier detection, and smoothing are often necessary to standardize data quality across series.

In some cases, particularly in emerging markets or niche sectors, data may be sparse or limited. Sparse data poses challenges for model training and can lead to less reliable forecasts. Advanced techniques, such as transfer learning (machine learning technique where a model developed for one task is reused as the starting point for a different and related task) or pooling (a technique used to aggregate, summarize, and reduce the dimension of feature maps that are largely populated by zero or near-zero values. It enables the network to extract the most prominent feature while disregarding redundant or empty background information) information across similar series, can help mitigate this issue.

Computational demands

Forecasting multiple time series simultaneously often requires more complex models, which can be computationally intensive. The complexity arises from the need to account for interactions between series and to model hierarchical or grouped structures. This increases the computational load, especially when using sophisticated models such as hierarchical forecasting or machine learning-based approaches.

Choosing the right model for multi-series forecasting is challenging due to the diverse nature of the data. Models must be validated not only for accuracy but also for their ability to generalize across different series. Cross-validation techniques, model selection criteria, and ensemble methods (machine learning techniques that combine predictions from multiple models, often called "weak learners" to produce a more accurate model) can help address this challenge, but they also add to the computational burden.

Understanding these challenges is the first step in developing strategies to overcome them. As you proceed with implementing multi-series forecasting, being aware of these potential hurdles will help you prepare better solutions and make more informed decisions. The following sections of the chapter will provide detailed methodologies and tools to address these challenges effectively.

Hierarchical forecasting methods

Hierarchical forecasting methods are essential in scenarios where data can be naturally organized into hierarchical structures, such as products within categories, stores within regions, or departments within a company. These methods allow for more accurate forecasting at different levels of aggregation by leveraging the relationships between these levels.

Introducing hierarchical forecasting

Hierarchical forecasting is a method used to predict data organized into a natural hierarchy, such as products grouped by categories, stores within regions, or various departments within an organization. This structure allows data to be analyzed and forecasted at different levels, from the most detailed to the most aggregated.

Hierarchical data structures are systems where data is organized into levels, with each level representing a different degree of aggregation. This layered structure helps businesses understand and forecast data not just at the detailed level (for example, individual product sales) but also at higher levels (such as total sales for a category or region).

Hierarchical forecasting is crucial in industries where decisions depend on both granular and aggregated data. For instance, in retail, companies need to forecast at the product level for inventory management, while also predicting overall category sales for marketing strategies.

Similarly, in finance, hierarchical forecasting can help in analyzing performance at the asset, portfolio, and sector levels. This method ensures that forecasts are consistent across different levels of aggregation, improving accuracy and aiding better decision-making.

Traditional forecasting methods often focus on a single series or aggregated data without considering the relationships between different levels of data aggregation. Hierarchical forecasting, on the other hand, leverages the hierarchical structure of data by making forecasts at each level of the hierarchy and ensuring that they are aligned. This approach provides a more comprehensive view and improves forecast accuracy by capturing the nuances at each level of the hierarchy, from the detailed to the most aggregated.

- **Top-down approach**: The top-down approach in hierarchical forecasting involves starting with a forecast at the highest level of the data hierarchy and then disaggregating the forecast down to lower levels. This method assumes that the overall forecast can be proportionally divided among the lower levels based on historical data or other relevant factors.
- **Concept and methodology**: In the top-down approach, a forecast is first generated for the aggregate level (such as total sales across all regions). This forecast is then broken down into more detailed forecasts (for example, sales by region, then by store) using historical proportions or other criteria. This method is relatively straightforward and ensures consistency across the hierarchy since all lower-level forecasts sum to the aggregate forecast.
- **Benefits and limitations**: The primary benefit of the top-down approach is its simplicity and the ease of ensuring consistency across different levels of the hierarchy. It is particularly useful when detailed data is sparse or when there is a need for a quick, high-level forecast.

However, the top-down approach has limitations. It can oversimplify the underlying data relationships and may not capture local variations or specific trends at lower levels. This can lead to less accurate forecasts at the granular level, particularly if the proportional disaggregation does not account for changing trends or unique characteristics in different segments.

Practical example – Aggregating and disaggregating data

The following is an example of using the top-down approach in R to forecast sales for a set of regions based on an overall forecast:

1. Generate sales data by region:

```
historical_sales <- data.frame(
   Region = c("North", "South", "East", "West"),
```

```
  Sales = c(100000, 150000, 120000, 130000)
)
```

2. Calculate total sales:

```
total_sales <- sum(historical_sales$Sales)
```

3. Set the forecasted total future sales:

```
total_forecast <- 550000  # Example total forecast for the next period
```

4. Calculate the proportion of sales by region:

```
historical_sales$Proportion <- historical_sales$Sales / total_sales
```

5. Disaggregate the total forecast into region:

```
historical_sales$Forecast <- historical_sales$Proportion * total_forecast
```

6. Print the forecast sales:

```
print(historical_sales)
```

In this example, you first calculate the total historical sales and the proportion of sales for each region. You then apply these proportions to a new forecast for total sales, distributing it proportionally across regions. This approach provides a quick way to derive regional forecasts from an overall forecast, ensuring consistency across the hierarchy.

	Region	Sales	Proportion	Forecast
1	North	100000	0.20	110000
2	South	150000	0.30	165000
3	East	120000	0.24	132000
4	West	130000	0.26	143000

Figure 10.1: Proportional region sales forecast

The top-down method is a useful tool in hierarchical forecasting, especially when time or data constraints limit the ability to develop more granular forecasts. However, it is important to be aware of its limitations and consider complementary methods, such as the bottom-up approach, for more detailed and accurate forecasts.

Bottom-up approach

The bottom-up approach in hierarchical forecasting involves forecasting at the most granular level of data first and then aggregating these forecasts to obtain higher-level forecasts. This method provides detailed insights into each component of the hierarchy, allowing for a more nuanced and accurate overall forecast.

In the bottom-up approach, forecasts are created for the smallest units within the hierarchy, such as individual products, stores, or regions. These individual forecasts are then summed or otherwise aggregated to produce forecasts at higher levels, such as total sales for a category or the entire company. This method is particularly useful in situations where detailed, granular data is available and where local trends or variations are significant.

The advantages of this method are as follows:

- **Detail-oriented**: Provides forecasts at the most detailed level, which can be very informative for decision-making.
- **Captures local variations**: More accurately reflects local trends, seasonality, and other factors affecting specific units within the hierarchy.
- **Flexibility**: Can easily accommodate additional granular data as it becomes available.

The challenges are as follows:

- **Data requirements**: Requires detailed and high-quality data at the most granular level, which may not always be available.
- **Complexity**: Managing and processing a large number of detailed forecasts can be computationally intensive.
- **Consistency issues**: Aggregating individual forecasts might lead to inconsistencies at higher levels if not carefully managed.

Practical example – Summing individual forecasts to create aggregated forecasts

The following is an example of how to implement a bottom-up forecasting approach in R:

1. Generate regional sales data for different products:

```
product_forecasts <- data.frame(
   Product = c("A", "A", "B", "B", "C", "C"),
   Region = c("North", "South", "North", "South", "North", "South"),
   Forecast = c(5000, 6000, 7000, 8000, 4500, 5500)
 )
```

2. Aggregate sales by product:

```
total_forecast_product <- aggregate(Forecast ~ Product, data =
product_forecasts, sum)
```

3. Aggregate sales by region:

```
total_forecast_region <- aggregate(Forecast ~ Region, data =
product_forecasts, sum)
```

4. Aggregate all forecasts to get the forecast for all sales:

```
total_forecast_overall <- sum(product_forecasts$Forecast)
```

5. Print aggregated forecasts at different levels:

```
  print(total_forecast_product)
 Product Forecast
1       A    11000
2       B    15000
3       C    10000

 print(total_forecast_region)
 Region Forecast
1  North    16500
2  South    19500

 print(total_forecast_overall)
[1] 36000
```

In this example, you start with individual forecasts for different products in various regions. You then aggregate these forecasts to obtain the total forecast for each product and region, and finally sum these to get an overall forecast. This bottom-up method provides detailed insights and allows for the analysis of forecasts at multiple levels of aggregation.

The bottom-up approach is highly effective when detailed data is available and there is a need to understand the dynamics at the most granular levels. However, it requires robust data management and careful aggregation to ensure accuracy and consistency across different levels of the hierarchy.

Middle-out approach

The middle-out approach is a hybrid forecasting technique that combines elements of both top-down and bottom-up methods. This approach starts with forecasting at an intermediate level of the data hierarchy, then disaggregates to lower levels or aggregates to higher levels as needed. It strikes a balance between the detail-oriented nature of the bottom-up approach and the broader view of the top-down approach.

The middle-out approach is particularly useful in complex organizations or systems where data can be more reliably forecast at an intermediate level due to better data quality or more stable trends. It is also effective in scenarios where forecasting at the highest or lowest levels alone may be challenging or less accurate. This approach works well when:

- **Intermediate data quality is higher**: The data at the middle levels is more reliable or consistent, providing a stable basis for forecasts.
- **Organizational complexity**: The organization has multiple layers that can obscure clear patterns when viewed only from the top or bottom.
- **Resource constraints**: The approach is useful when there are limited resources for detailed bottom-up forecasting but a need for more than a broad top-down perspective.

The following is an example of how to implement the middle-out approach in R:

1. Generate intermediate-level data (such as data at the category level):

```
category_forecasts <- data.frame(
   Category = c("Electronics", "Furniture", "Clothing"),
   Forecast = c(150000, 100000, 120000)
 )
```

2. Generate proportions for products to disaggregate category forecasts into products:

```
product_proportions <- data.frame(
   Category = c("Electronics", "Electronics", "Furniture", "Furniture",
"Clothing", "Clothing"),
   Product = c("Phones", "Computers", "Chairs", "Tables", "Shirts",
"Pants"),
   Proportion = c(0.6, 0.4, 0.5, 0.5, 0.55, 0.45)
 )
```

3. Disaggregate the product-level forecast into product-level forecasts:

```
product_forecasts <- merge(category_forecasts, product_proportions, by =
"Category")
product_forecasts$Product_Forecast <- product_forecasts$Forecast *
product_forecasts$Proportion
```

4. Aggregate product forecasts to confirm consistency with category forecasts:

```
aggregated_category_forecasts <- aggregate(Product_Forecast ~ Category,
data = product_forecasts, sum)
```

5. Print disaggregated and aggregated forecasts and aggregated category-level checks:

```
print(product_forecasts)
```

The preceding command will print the disaggregated product-level forecasts based on the proportion ratio. In this case, **Shirts'** proportion is 0.55 of the overall **Clothing** forecast (120000), which will result in a forecast sales number of 66000.

	Category	Forecast	Product	Proportion	Product_Forecast
1	Clothing	120000	Shirts	0.55	66000
2	Clothing	120000	Pants	0.45	54000
3	Electronics	150000	Phones	0.60	90000
4	Electronics	150000	Computers	0.40	60000
5	Furniture	100000	Chairs	0.50	50000
6	Furniture	100000	Tables	0.50	50000

Figure 10.2: Disaggregated product-level forecasts

6. The following command will print out the aggregated category forecasts, which should be equal to the original category forecasts data frame value:

```
print(aggregated_category_forecasts)
```

You will get the following:

```
     Category Product_Forecast
1    Clothing           120000
2 Electronics           150000
3   Furniture           100000
```

Figure 10.3: Aggregated forecasts

In this example, you start with forecasts at the category level, an intermediate point in the hierarchy. You then use predefined proportions to disaggregate these forecasts into more granular product-level forecasts. Finally, you aggregate the product-level forecasts to ensure that they align with the original category forecasts, maintaining consistency.

The middle-out approach provides flexibility and adaptability, making it a valuable method in environments where data quality varies across levels or where organizational complexity necessitates a nuanced forecasting approach. This method ensures a coherent forecasting strategy that can be adjusted according to data availability and accuracy at different hierarchical levels.

Reconciliation and consistency in hierarchical forecasting

In hierarchical forecasting, ensuring consistency across different levels of the data hierarchy is crucial. This involves reconciling forecasts made at various levels, such as individual products, categories, and overall totals, so that they align and accurately reflect the underlying data structure. Reconciliation helps maintain the coherence of forecasts, ensuring that they add up correctly from the bottom of the hierarchy to the top.

Common reconciliation methods include:

- **Top-down reconciliation**: In this method, the forecast for the highest level is disaggregated into lower levels. Adjustments are made at the lower levels to ensure that they sum up to the top-level forecast. This method is simple but may not capture local variations effectively.
- **Bottom-up reconciliation**: This involves aggregating forecasts from the lowest levels to form the higher-level forecasts. While this method captures local details well, it may not ensure alignment with broader trends unless adjustments are made.
- **Optimal combination**: This approach uses statistical methods to combine forecasts from different levels in an optimal way. It aims to minimize errors and discrepancies across the hierarchy by using weighted averages or other statistical techniques. This method often yields more accurate and consistent forecasts.

- **Forecast reconciliation**: This technique involves using specific algorithms or software tools to adjust forecasts across the hierarchy. For example, the "forecast reconciliation" method in R uses matrix algebra to adjust forecasts in a way that balances errors and aligns different levels.

Maintaining consistency and coherence in hierarchical forecasts is crucial for several reasons. Consistent forecasts provide a reliable basis for decision-making. Consistent forecasting also builds credibility with stakeholders, including management, investors, and customers. It instills confidence that the organization understands its data and can predict future trends accurately.

On the other hand, *inconsistent* forecasts can result in suboptimal resource allocation, whether it is inventory management, budget planning, or staffing. Discrepancies can lead to confusion and potentially flawed decisions, particularly when different levels of an organization rely on data with different levels of granularity for planning and strategy.

By employing reconciliation techniques, organizations can achieve coordinated alignment of forecasts across all levels, enhancing the accuracy and reliability of their predictions. In the next section, you will look at grouping data by various categories.

Grouped time series analysis

Grouped time series analysis involves forecasting and analyzing time series data that can be categorized into groups based on certain characteristics, such as geographical regions, product types, or customer segments. This approach allows for a deeper understanding of patterns within each group and can lead to more accurate forecasting and better decision-making.

For example, sales data might be grouped by product type (electronics, clothing, and so on) or by region (such as North, South, East, or West). Within each group, the time series share certain similarities that can be exploited to improve forecasting accuracy. Grouping helps in identifying patterns and trends that are specific to each group, providing more granular insights than treating the entire dataset as a homogeneous series.

Grouping time series is crucial for several reasons:

- **Enhanced forecasting accuracy**: By analyzing groups with similar characteristics, forecasters can identify and leverage common patterns and trends, leading to more accurate predictions.
- **Targeted insights**: Grouped analysis allows organizations to tailor strategies for specific segments, whether it's marketing efforts, resource allocation, or inventory management.

- **Efficiency in model building**: Models can be designed for each group rather than for each individual series, simplifying the forecasting process while still capturing the essential characteristics of the data.

Comparison with hierarchical forecasting

While both grouped and hierarchical forecasting involve organizing data into categories, they differ in structure and focus:

- **Hierarchical forecasting**: This approach deals with data that is naturally arranged in a hierarchy, such as a company's sales data divided into regions, then cities, and then stores. The focus is on maintaining consistency across different levels of aggregation.
- **Grouped forecasting**: Here, the primary concern is analyzing and forecasting data within distinct groups, which might not have a hierarchical relationship. The emphasis is on understanding the similarities and differences within and between groups, rather than on consistency across levels.

Grouped time series analysis is a powerful tool for uncovering deeper insights and trends in data that can inform more effective decision-making.

By focusing on specific segments, organizations can achieve a more nuanced understanding of their data, leading to better-targeted actions and strategies. This method complements other forecasting approaches, providing a more comprehensive toolkit for tackling complex time series data.

Methods for grouping time series data

Grouping time series data involves categorizing individual series into coherent groups based on shared characteristics. This process enhances the ability to detect and leverage patterns within each group, leading to more precise and meaningful forecasts. The grouping of time series can be guided by various criteria, which are crucial for ensuring that the resulting groups are both relevant and useful for analysis.

Grouping criteria

Grouping criteria are the defining characteristics used to categorize time series. Common criteria include:

- **Geographic**: Time series data can be grouped based on geographic locations, such as countries, regions, cities, or even store locations. This is useful for analyzing and forecasting data influenced by local factors, such as weather, economic conditions, or cultural differences.

- **Demographic**: In cases where time series data involves human behavior, such as customer purchases or service usage, demographic criteria such as age, gender, income level, or educational background can be used. This allows for targeted analysis and forecasting that considers specific demographic trends.
- **Product-based**: When dealing with sales or inventory data, products can be grouped by type, category, or brand. This helps in understanding and forecasting demand patterns specific to different product lines or categories.
- **Temporal**: Grouping based on time-related factors, such as seasons, holidays, or business cycles, helps capture periodic patterns and trends that are time-dependent.

Techniques for identifying natural groupings in data

To effectively group time series data, it is often necessary to use techniques that help identify natural clusters or segments within the data. Some common methods include:

- **Clustering algorithms**: Algorithms such as k-means, hierarchical clustering (both from the built-in `stats` package), or DBSCAN (the `dbscan` package) can be used to identify groups based on similarities in the data. These techniques are particularly useful when the appropriate grouping criteria are not obvious.
- **Principal Component Analysis (PCA)**: PCA (from the built-in `stats` package) can reduce the dimensionality of time series data, highlighting the most significant features that differentiate series. This helps identify natural groupings based on these key features.
- **Correlation analysis**: Analyzing correlations between time series can reveal groups of series that behave similarly over time. Highly correlated series are often grouped together because they share common trends or influences.
- **Domain knowledge**: Leveraging expert knowledge about the data can help identify meaningful groups that statistical methods might miss. For example, an industry expert might know that certain products are often purchased together, which statistical analysis alone might not reveal.

Data preprocessing steps for grouped time series analysis

Before performing grouped time series analysis, it is essential to preprocess the data to ensure accuracy and consistency. Key preprocessing steps include:

- **Data cleaning**: Remove or correct any anomalies, such as missing values, outliers, or data entry errors. Consistent data quality is critical for accurate grouping and analysis.

- **Normalization**: Normalize the data to standardize scales across different series, making it easier to compare and analyze the series within each group.
- **Time alignment**: Ensure that all time series are aligned on the same time scale, especially when dealing with data from different sources or with different reporting frequencies.
- **Feature engineering**: Create additional features or metrics that might help identify groups, such as average sales per month, growth rates, or seasonality indices.

By carefully selecting grouping criteria and employing robust techniques for identifying natural groupings, analysts can effectively segment time series data for more targeted and accurate forecasting. Proper data preprocessing further ensures that the analysis is based on high-quality, consistent data, leading to more reliable and actionable insights.

Forecasting techniques for grouped time series

Forecasting grouped time series involves predicting future values for time series data that have been categorized into meaningful groups. This approach leverages both the individual characteristics of each series and the shared attributes within groups, allowing for more precise and relevant forecasts. Various forecasting methods can be applied, each with unique strengths and applications.

- **Univariate time series models**: Traditional methods such as ARIMA (Autoregressive Integrated Moving Average) or exponential smoothing are often used for individual series. When applied within groups, these models can be adjusted to account for group-specific patterns.
- **Multivariate time series models**: Models such as **vector autoregression** (**VAR**) (from the R package `vars`) handle multiple time series simultaneously, capturing the interactions between series within a group. This is particularly useful when group members influence each other's behavior.
- **Hierarchical and grouped forecasting models**: These models, including hierarchical ARIMA (from the R package `forecast`) or hierarchical exponential smoothing (from the `hts` package), explicitly account for the grouping structure in the data. They forecast at multiple levels, ensuring consistency across forecasts within and between groups.

- **Machine learning models**: Techniques such as random forests (from the R package `randomForest`), neural networks (from packages like `nnet`, `keras`), and gradient boosting (using packages such as `xgboost` or `lightgbm`) can be adapted for grouped time series by incorporating group identifiers as features. These models are flexible and can capture complex, nonlinear relationships in the data.

Specialized models for grouped time series

There are also several specialized models:

- **Multilevel models**: These models, also known as mixed-effects models, account for variations at different levels (for example, within-group and between-group variations). They are useful for handling hierarchical data structures and can provide more accurate forecasts by accounting for group-specific effects and correlations.
- **Panel data approaches**: Panel data models, including fixed-effects and random-effects models, are designed for datasets with multiple entities (such as countries or companies) observed over time. They effectively handle both cross-sectional and temporal dimensions, making them suitable for grouped time series analysis.

Grouped forecasting practical example

Let's walk through an example of implementing grouped time series forecasting using the `forecast` and `hts` packages in R, focusing on hierarchical forecasting methods:

1. Install and load the `hts` package:

```
if (!requireNamespace("hts", quietly = TRUE)) {
  # If not installed, install it
  install.packages("hts")
}
# Load the hts package
library(hts)
```

2. Load the `infantgts` dataset:

```
data(infantgts)
str(infantgts)

List of 3
 $ bts    : Time-Series [1:71, 1:16] from 1933 to 2003: 738 886 760 908 851
807 833 873 1000 912 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:16] "NSW female" "VIC female" "QLD female" "SA
female" ...
 $ groups: 'gmatrix' int [1:4, 1:16] 1 1 1 1 1 1 2 2 1 1 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:4] "Total" "Sex" "State" "Bottom"
  .. ..$ : chr [1:16] "NSW female" "VIC female" "QLD female" "SA
female" ...
 $ labels:List of 2
  ..$ Sex  : chr [1:2] "Sex/female" "Sex/male"
  ..$ State: chr [1:8] "State/NSW" "State/VIC" "State/QLD" "State/SA" ...
 - attr(*, "class")= chr "gts"
```

The `infantgts` dataset is an example dataset from the `hts` package. The dataset contains regional infant mortality counts across Australia from 1933 to 2003 and can first be disaggregated by sex and then by state, or vice versa.

3. Plot the dataset:

```
plot(infantgts)
```

The command will generate the plot from the infant mortality data. The Total diagram will display the aggregated information, the second one (Sex) will display the aggregated diagram by sex (male and female), the third one displays the aggregation by State, and the Bottom displays the 27 detailed series on one chart.

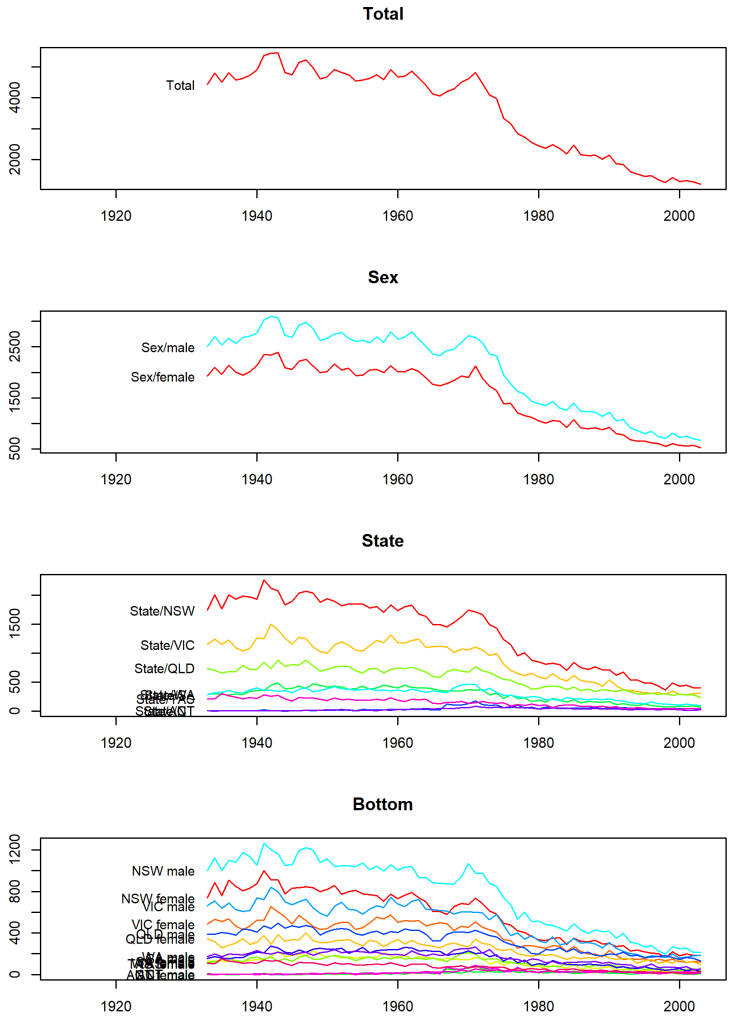


Figure 10.4: Plot of infantgts dataset showing grouped time series

4. Plot the grouped-level dataset. Since the `infantgts` set contains grouped labels, the following command can pass the levels parameter to display aggregated information:

```
# Set levels = 2 for State aggregation
# Set levels = 1 to aggregate at Sex (male/female)
plot(infantgts, levels = 1)
```

This produces the following graphic:

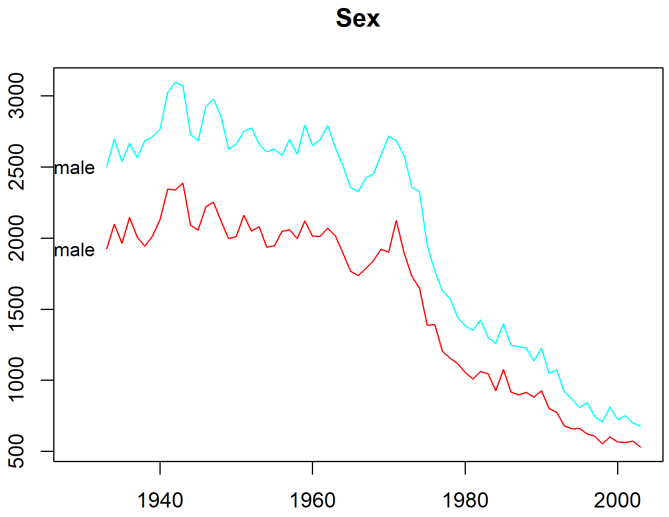


Figure 10.5: Aggregated group plot at level 1

You then use an ARIMA model to forecast infant mortality for each group. This approach illustrates how traditional models can be applied in a grouped forecasting context, taking advantage of group-level insights to improve forecast accuracy.

5. Forecast the future period using ARIMA:

```
h <- 10  # Forecast horizon
forecast_infant <- forecast(infantgts, h = h, method = "comb", fmethod =
"arima")
plot(forecast_infant)
```

The preceding command will set the forecast period to 10, "`comb`" tells R to use a combination-based reconciliation approach (in hierarchical time series, you first generate forecasts at each level, then reconcile them so they add up correctly across the hierarchy), and the `fmethod` parameter specifies the forecasting model used at each node of the hierarchy before reconciliation.

You can set the levels parameter in the plot to display aggregated information for States or Sex, as in the previous code snippets.

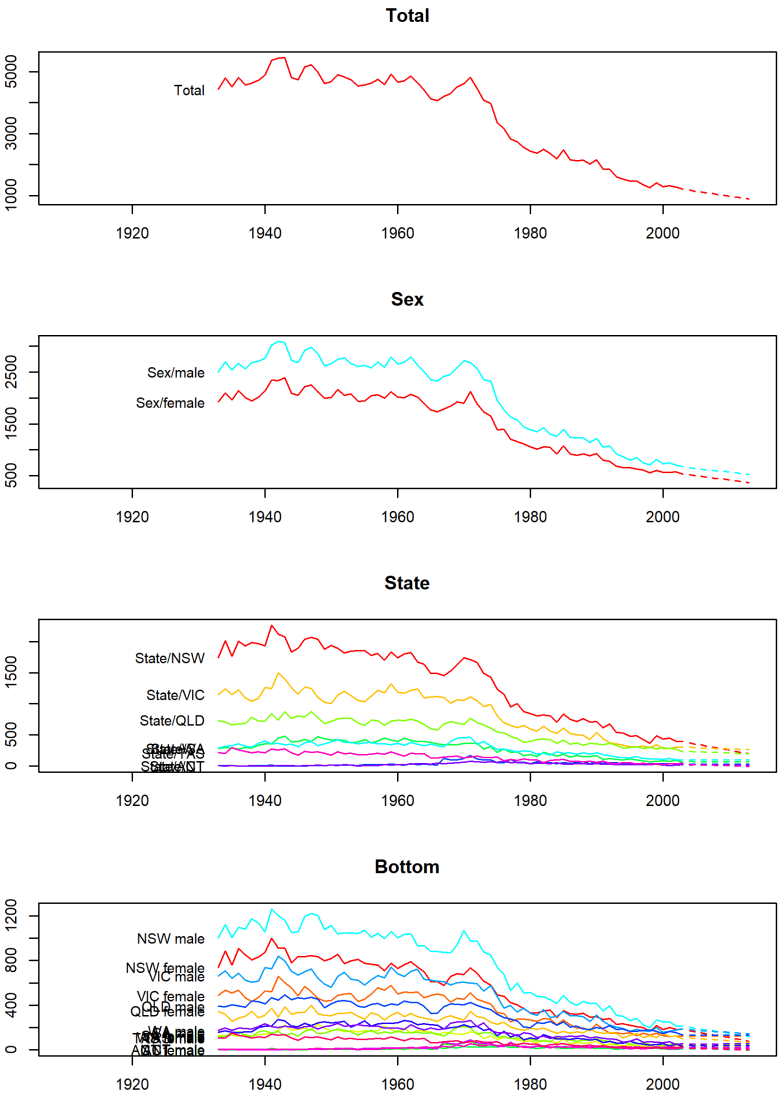


Figure 10.6: Forecast plot on infant mortality

By using specialized models tailored for grouped time series, analysts can more effectively capture and utilize the unique dynamics within each group, leading to more robust and actionable forecasts. These techniques are essential in industries where data is naturally segmented, providing critical insights for decision-making and strategic planning.

Handling cross-sectional dependencies

In grouped time series analysis, cross-sectional dependencies refer to the relationships and interactions between different groups within the data. These dependencies can significantly impact the accuracy of forecasts if not properly accounted for. Understanding and managing these dependencies is crucial for creating reliable and accurate predictive models.

Cross-sectional dependencies arise when the behavior of one group influences or is correlated with the behavior of another group. For example, in a retail context, the sales of different product categories may be interrelated due to factors such as consumer preferences or economic conditions. Similarly, in financial markets, the performance of different sectors or asset classes can be interconnected.

To manage these dependencies, it's important to use statistical methods, such as correlation analysis or cross-correlation functions, to identify and quantify the relationships between different groups. Understanding these relationships helps in selecting appropriate models and adjusting forecasts accordingly.

You must also consider external factors that might affect multiple groups simultaneously, such as economic indicators, weather conditions, or market trends. These factors can introduce dependencies that need to be incorporated into the forecasting model.

Techniques for modeling interactions between different groups

Several techniques are available to model the relationships and interactions between different groups in time series data. These techniques are particularly useful when analyzing complex, interrelated datasets, and they include the following:

- **Multivariate models**: Multivariate time series models, such as VAR, are designed to handle multiple interrelated time series simultaneously. These models can capture the dynamic relationships between groups and allow for the forecasting of each series while considering the influence of others.
- **Cointegration analysis**: When groups are affected by common long-term trends, cointegration analysis can be used to model these shared trends and adjust the forecasts accordingly. This technique is particularly useful when groups exhibit a long-term equilibrium relationship despite short-term fluctuations.
- **Hierarchical models**: Hierarchical models can incorporate cross-sectional dependencies by modeling data at multiple levels of aggregation, accounting for both within-group and between-group variations. These models are flexible and can be tailored to specific hierarchical structures within the data.
- **Panel data models**: Panel data models are effective for handling data with both cross-sectional (between groups) and time-series (within group) dimensions. Fixed-effects

or random-effects models can be used depending on the nature of the data and the assumptions about the cross-sectional dependencies.

Impact of cross-sectional dependencies on forecasting accuracy

Ignoring cross-sectional dependencies can lead to significant forecasting errors. If the interactions between groups are not accounted for, models may either overestimate or underestimate the forecasted values. This is because independent forecasts for each group fail to consider the mutual influences that can drive actual outcomes.

When groups are influenced by common factors, failing to model these shared trends can result in biased forecasts. For instance, an economic downturn might simultaneously depress sales across multiple product categories, a relationship that simple independent models might miss.

Conversely, without considering cross-sectional dependencies, models might overfit to noise specific to each group, leading to less generalizable and robust forecasts.

Properly addressing cross-sectional dependencies not only enhances the accuracy of forecasts but also provides a deeper understanding of the underlying factors driving group behaviors. This, in turn, can lead to more informed strategic decisions, whether in inventory management, financial planning, or market analysis.

Summary

In this chapter, you learned about the complexities and methodologies associated with multi-series forecasting. This approach is indispensable in scenarios where managing multiple interconnected data streams is crucial, such as retail chains, financial portfolios, or large-scale supply chains.

You have covered several key aspects, starting with the basic concepts and importance of multi-series forecasting. This technique not only improves forecast accuracy but also provides richer insights by leveraging the interrelationships between different data series. Understanding these relationships helps in achieving a more comprehensive view of the data, essential for informed decision-making.

The chapter also detailed various methods such as hierarchical, grouped, and middle-out forecasting approaches, each with its unique applications and benefits. Hierarchical forecasting, whether top-down, bottom-up, or middle-out, helps in structuring and analyzing data across different levels of aggregation, ensuring consistency and coherence. Grouped time series analysis, on the other hand, allows for more nuanced forecasting by focusing on similarities within categorized groups.

Addressing challenges such as data management complexities, varying data quality, and computational demands is critical in implementing these forecasting techniques effectively. Techniques for reconciling forecasts and handling cross-sectional dependencies further enhance the accuracy and reliability of forecasts, ensuring that they align across different levels and groups.

In *Chapter 11*, you will dive into Prophet, Facebook/Meta's powerful tool for time series forecasting. You will learn the basics, implement it in R, and explore advanced techniques such as custom seasonalities, handling holidays, and parameter tuning.

11

Prophet for Time Series

Welcome to *Chapter 11*, where you will explore **Prophet**, a powerful tool from Meta designed for time series forecasting. In this chapter, you will discover the essentials alongside some advanced techniques to help you master this tool.

You will begin by learning the basics of Prophet, its origins, and why it stands out from traditional forecasting methods. Next, you will learn how to install Prophet, fit it to your data, make future predictions, and visualize your forecasts, all within R.

You will cover the following topics in the chapter:

- Introducing Prophet
- Implementing Prophet in R
- Tuning and customizing models
- Advanced Prophet techniques

By the end of this chapter, you will be able to use Prophet confidently, tailor it to your specific needs, and make accurate forecasts for complex time series data. Let's get started and see what Prophet can do for you!

Technical requirements

R and RStudio are required for this chapter. GitHub scripts can be found in the repository.

You will need the following data file and packages for this chapter:

- `Prophet`: The `Prophet` package in R is a very popular package developed by Meta specifically for time series forecasting. You will see code examples in this chapter showing how to use this package to forecast and tune the model.

- `dplyr`: This R package is a powerful tool designed to make data manipulation and cleaning of tabular data (data frames) easier and faster.
- `Daily_Bike_Sharing_training.csv`: This is the data file that you will use in the chapter to demonstrate how to use the `Prophet` model for prediction. A detailed description will be provided in the content.
- `Daily_Bike_Sharing_future.csv`: This is the data file that you will use as a future regressor for the preceding dataset. It will be used by Prophet for prediction.

Introducing Prophet

Prophet is a powerful tool developed by Facebook (now Meta) designed specifically for time series forecasting. Unlike many traditional forecasting methods, Prophet excels at handling complex, real-world data. It originated from the need to make forecasting more accessible and accurate, especially for business applications where seasonal patterns and irregular data points are common.

The goal was to develop a model that could handle a wide range of data issues, from missing values to outliers, without requiring extensive tuning.

Prophet sets itself apart from traditional methods in several ways:

- **Additive model**: Prophet uses an additive model in which nonlinear trends are fitted with yearly, weekly, and daily seasonality, plus holiday effects. This approach allows it to adapt to various data patterns effectively.
- **Flexibility**: It automatically detects and adapts to changes in trends, making it suitable for datasets with abrupt shifts or missing values.
- **Ease of use**: Prophet's design focuses on simplicity and speed. It is developed in Stan with vectorized operations and optimized for daily-to-sub-hourly data at a large scale. While traditional models such as ARIMA and ETS are fast for small series, they can struggle or necessitate manual modifications for long series with multiple seasonalities.
- **Robustness**: The model is robust to missing data and outliers, making it ideal for business environments where data quality can be inconsistent.

Prophet combines ease of use with the ability to handle complex data scenarios, making it a go-to tool for time series forecasting in various industries.

Key components of Prophet

Prophet's strength lies in its additive model, which breaks down a time series into three main components: *trend*, *seasonality*, and *holidays*. Each of these components plays a critical role in accurately modeling and forecasting time series data:

- **Trend**: The trend component captures the long-term direction of the time series. It identifies whether the data is increasing, decreasing, or remaining constant over time. Prophet handles trends using a piecewise linear or logistic growth model, which allows it to adapt to changes in the direction and rate of growth. This flexibility is crucial for datasets that exhibit non-linear growth patterns or sudden shifts.
- **Seasonality**: Seasonality refers to periodic fluctuations that occur at regular intervals, such as daily, weekly, or yearly patterns. Prophet includes multiple seasonal components to account for these regular variations. For instance, daily seasonality might capture patterns that recur every 24 hours, while yearly seasonality might capture annual cycles such as holiday sales or weather patterns. By incorporating these seasonal effects, Prophet can accurately model and predict recurring patterns in the data.
- **Holidays**: The holidays component adjusts for the impact of holidays and special events on the time series. These events often cause significant deviations from the regular trend and seasonality patterns. Prophet allows users to specify a list of holidays and special events that might affect the data, and it adjusts the model accordingly. This feature is particularly useful for business applications, where holidays and promotional events can have a substantial impact on sales and other key metrics.

By breaking down the time series into these fundamental components, Prophet offers a flexible and robust approach to forecasting that can handle a wide variety of real-world scenarios.

When to use Prophet

For datasets with strong seasonal trends, such as daily, weekly, or yearly cycles, Prophet models multiple seasonalities simultaneously, providing accurate predictions. It also captures long-term growth or decline trends through its piecewise linear and logistic growth models, which can adapt to changes in the direction and rate of growth.

Prophet accounts for the impact of holidays, promotions, or special events on the data through its holidays component, improving forecast accuracy. It is user-friendly and allows quick generation of reliable forecasts without requiring deep knowledge of complex time series modeling techniques.

In summary, Prophet is a robust and flexible forecasting tool suitable for handling complex time series data with seasonal patterns, long-term trends, and event impacts. It is ideal for

business metrics, irregular data, and long-term projections, providing accuracy and ease of use.

Implementing Prophet in R

To start using Prophet in R, you need to install and load the package. Begin by opening your R environment:

1. Install and load the Prophet package:

```
if (!requireNamespace("prophet", quietly = TRUE)) {
  install.packages("prophet")
}
library(prophet)
```

2. Load your data into R. Ensure your data frame contains a date column named `ds` and a value column named `y`. Here is how to read your data from a CSV file:

```
data <- read.csv('Daily_Bike_Sharing_training.csv')
head(data)
```

   The output shows the first few rows of the dataset, giving an overview of the variables and their values:

```
  instant   dteday season yr mnth holiday weekday workingday weathersit     temp
1       1 1/1/2011      1  0    1       0       6          0          2 0.344167
2       2 1/2/2011      1  0    1       0       0          0          2 0.363478
3       3 1/3/2011      1  0    1       0       1          1          1 0.196364
4       4 1/4/2011      1  0    1       0       2          1          1 0.200000
5       5 1/5/2011      1  0    1       0       3          1          1 0.226957
6       6 1/6/2011      1  0    1       0       4          1          1 0.204348
     atemp      hum windspeed casual registered  cnt
1 0.363625 0.805833 0.1604460    331        654  985
2 0.353739 0.696087 0.2485390    131        670  801
3 0.189405 0.437273 0.2483090    120       1229 1349
4 0.212122 0.590435 0.1602960    108       1454 1562
5 0.229270 0.436957 0.1869000     82       1518 1600
6 0.233209 0.518261 0.0895652     88       1518 1606
```

Figure 11.1: A snapshot of the Daily Bike Sharing dataset

   The dataset is quite extensive, and, therefore, you may need to choose which variables to use. Here is a description of the data:

```
========================================
Dataset characteristics
========================================
Both hour.csv and day.csv have the following fields, except hr which is not available in day.csv

        - instant: record index
        - dteday : date
        - season : season (1:springer, 2:summer, 3:fall, 4:winter)
        - yr : year (0: 2011, 1:2012)
        - mnth : month ( 1 to 12)
        - hr : hour (0 to 23)
        - holiday : weather day is holiday or not (extracted from http://dchr.dc.gov/page/holiday-schedule)
        - weekday : day of the week
        - workingday : if day is neither weekend nor holiday is 1, otherwise is 0.
        + weathersit :
                - 1: Clear, Few clouds, Partly cloudy, Partly cloudy
                - 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
                - 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
                - 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
        - temp : Normalized temperature in Celsius. The values are divided to 41 (max)
        - atemp: Normalized feeling temperature in Celsius. The values are divided to 50 (max)
        - hum: Normalized humidity. The values are divided to 100 (max)
        - windspeed: Normalized wind speed. The values are divided to 67 (max)
        - casual: count of casual users
        - registered: count of registered users
        - cnt: count of total rental bikes including both casual and registered
```

Figure 11.2: Description of the variables available in the dataset

Drop the variables not needed. Based on relevance, you can, for instance, exclude "`casual`" and "`registered`" because they are subsets of the "`cnt`" variable. These are the variables that you will exclude:

```
library(dplyr)
data <- data %>% select(-instant, -season, -yr, -mnth, -weekday, -casual, -
registered)
```

The `select()` function keeps all columns except the ones preceded by `-`.

3. Then, Prophet is very specific. The date variable must be called `'ds'` for `datestamp`, and the time series is to be named `'y'`:

```
data <- data %>% rename(y = cnt, ds = dteday)
head(data)
```

That produces the following output:

```
        ds holiday workingday weathersit     temp    atemp      hum windspeed    y
1 1/1/2011       0          0          2 0.344167 0.363625 0.805833 0.1604460  985
2 1/2/2011       0          0          2 0.363478 0.353739 0.696087 0.2485390  801
3 1/3/2011       0          1          1 0.196364 0.189405 0.437273 0.2483090 1349
4 1/4/2011       0          1          1 0.200000 0.212122 0.590435 0.1602960 1562
5 1/5/2011       0          1          1 0.226957 0.229270 0.436957 0.1869000 1600
6 1/6/2011       0          1          1 0.204348 0.233209 0.518261 0.0895652 1606
```

Figure 11.3: First few rows of the prepared dataset after loading and cleaning

4. Format the date variable. The last prep step is that the date variable must be in the `YYYY-MM-DD` format:

```
data$ds <- as.Date(data$ds, format = "%m/%d/%Y")
```

You will get the following outcome:

```
          ds holiday workingday weathersit     temp    atemp      hum
1 2011-01-01       0          0          2 0.344167 0.363625 0.805833
2 2011-01-02       0          0          2 0.363478 0.353739 0.696087
3 2011-01-03       0          1          1 0.196364 0.189405 0.437273
4 2011-01-04       0          1          1 0.200000 0.212122 0.590435
5 2011-01-05       0          1          1 0.226957 0.229270 0.436957
6 2011-01-06       0          1          1 0.204348 0.233209 0.518261
  windspeed    y
1 0.1604460  985
2 0.2485390  801
3 0.2483090 1349
4 0.1602960 1562
5 0.1869000 1600
6 0.0895652 1606
```

Figure 11.4: The prepared dataset showing key columns

5. Next, fit the Prophet model to your data. The `prophet` function initializes and fits the model to the provided data frame:

```
model <- prophet(data)
```

6. To make future predictions, create a data frame with future dates using `make_future_dataframe`. This function generates a sequence of dates extending from the end of your data. You specify the number of periods (for example, days, months) that you want to forecast:

```
future <- make_future_dataframe(model, periods = 30)
```

7. Use the predict function to generate forecasts for the future dates. This function returns a data frame with the predicted values and additional information about the forecast components:

```
forecast <- predict(model, future)
```

This produces the following:

Filter

	ds	trend	additive_terms	additive_terms_lower	additive_terms_upper	weekly
1	2011-01-01	993.9687	75.61331	75.61331	75.61331	75.61331
2	2011-01-02	1014.9852	-223.22852	-223.22852	-223.22852	-223.22852
3	2011-01-03	1036.0016	-159.75476	-159.75476	-159.75476	-159.75476
4	2011-01-04	1057.0181	-21.50200	-21.50200	-21.50200	-21.50200
5	2011-01-05	1078.0346	29.76709	29.76709	29.76709	29.76709
6	2011-01-06	1099.0510	142.07513	142.07513	142.07513	142.07513
7	2011-01-07	1120.0675	157.02975	157.02975	157.02975	157.02975
8	2011-01-08	1141.0840	75.61331	75.61331	75.61331	75.61331
9	2011-01-09	1162.1004	-223.22852	-223.22852	-223.22852	-223.22852
10	2011-01-10	1183.1169	-159.75476	-159.75476	-159.75476	-159.75476

Showing 1 to 10 of 731 entries, 16 total columns

Figure 11.5: Snippet of the forecasted data generated by Prophet

This is just a snippet of the forecast. There are 16 variables in this data frame that build to the forecast `yhat` (not shown on the screen). This is, however, still a first model, and the complexity will increase. First, let's visualize it.

8. Prophet includes built-in functions to visualize forecasts and their components. To plot the forecast, use the `plot` function, which shows the actual data points and the predicted values:

```
plot(model, forecast)
```

Figure 11.6 is the output:

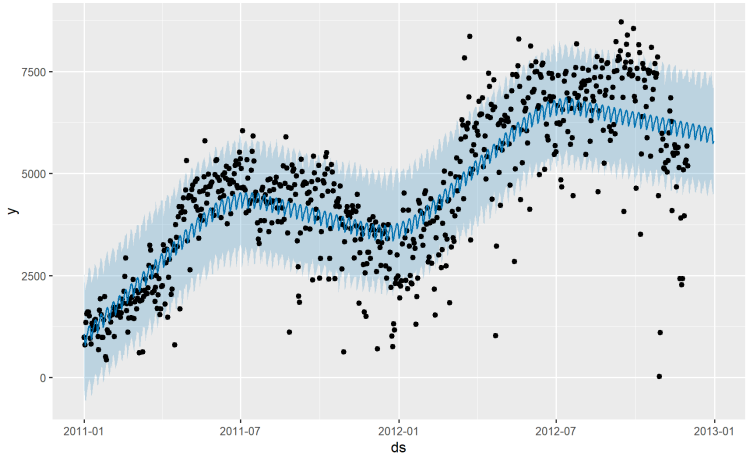


Figure 11.6: Visualization of the forecast generated by Prophet

The plot shows a clear upward trend in the data, with some seasonal fluctuations around the trend. The blue line represents the fitted trend using Prophet, and the shaded area around it indicates the confidence interval. The fluctuations, especially the repeating peaks and troughs, suggest a recurring pattern, possibly annual. The black dots represent the actual historical data points, and at the end, you can see that Prophet added data after 12-01-2012, which are future forecasts.

There seems to be a strong possibility of *yearly seasonality* given the repeating pattern within each year, particularly in the rises and falls from mid-2011 to mid-2012. This cyclic behavior is a key indicator of yearly seasonality, which could be tied to external factors such as weather, holidays, or business cycles that follow a yearly pattern.

9. To visualize the components of the forecast, such as the trend, seasonality, and holidays, use the `prophet_plot_components` function. This provides a breakdown of the different elements that contribute to the forecast:

```
prophet_plot_components(model, forecast)
```

The Prophet library allows you to break down the forecast into its individual components, such as overall trend, weekly seasonality, and holiday effects, providing deeper insights into the factors driving the predictions.

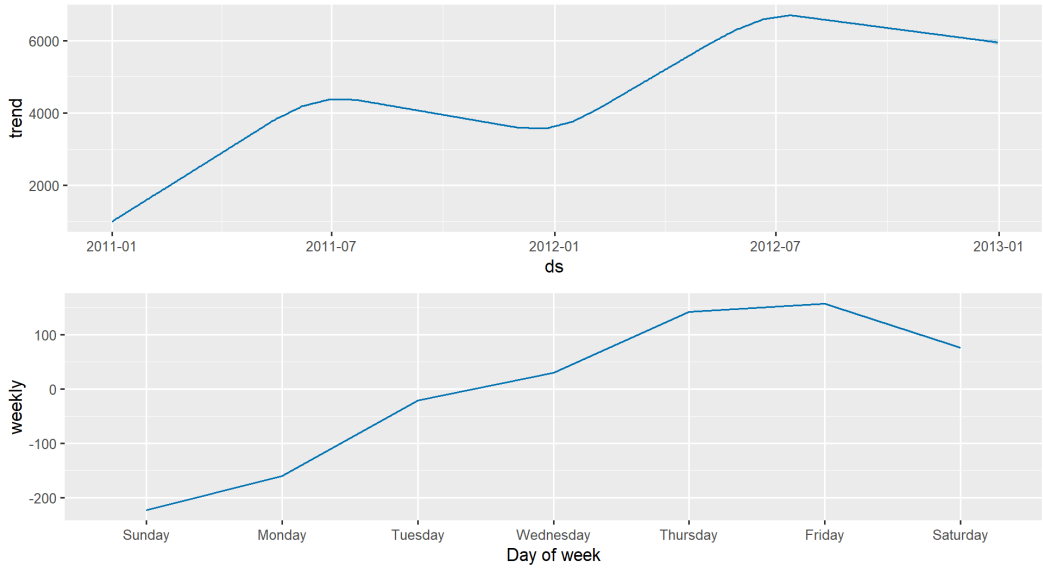


Figure 11.7: Visualization of the individual components

The top plot shows the *overall trend* over time, with a rise in activity from early 2011, peaking around mid-2012, and then tapering off slightly toward the end of 2012. The bottom plot illustrates *weekly seasonality*, showing that activity is lowest on Sundays, gradually increasing throughout the week, peaking on Fridays, and then decreasing on Saturdays. This indicates both a long-term upward trend and clear weekly patterns in the data.

By following these steps, you can implement Prophet in R, fit a model to your data, make forecasts, and visualize the results effectively.

Tuning and customizing models

Once you have a basic Prophet model up and running, the next step is to fine-tune and customize it to better fit your specific time series data. Prophet offers several features that allow you to adjust for trends, seasonality, additional regressors, and even handle missing data. These adjustments can significantly enhance the accuracy and relevance of your forecasts.

Seasonality modes

Prophet can model multiple seasonal patterns, including yearly, weekly, and daily seasonality. By default, Prophet includes yearly and weekly seasonalities. You can enable daily seasonality or customize the existing ones to better fit your data.

To initiate the Prophet model with the desired daily seasonality parameters, use the following code:

```
m <- prophet(
  yearly.seasonality = TRUE,        # Include yearly seasonality
  weekly.seasonality = TRUE,        # Include weekly seasonality
  seasonality.mode = 'multiplicative', # Use multiplicative seasonality
  seasonality.prior.scale = 10,     # Increase flexibility of seasonality
  holidays.prior.scale = 10,        # Increase impact of holidays
  changepoint.prior.scale = 0.05    # Control flexibility in detecting trend
changes
)
```

The `yearly.seasonality=TRUE` will add repeating yearly patterns to the model, while `weekly.seasonality = TRUE` adds weekly patterns. This will improve the model's accuracy because removing a known cycle (such as weekly or yearly) forces the model to try to absorb it into the trend or noise, which usually hurts out-of-sample fit. The `seasonality.mode='multiplicative'` means that seasonal effects change proportionally with the trend. The `seasonality.prior.scale` controls how flexible the seasonal components are, with larger values allowing more complex seasonal patterns. `holidays.prior.scale` increases the influence of holiday effects on forecasts, and finally, `changepoint.prior.scale` controls how sensitive the model is to trend changes. Larger values allow more trend flexibility.

By tuning these seasonal components, you can capture more intricate patterns in your data, leading to more accurate forecasts.

Adding Regressors

In some cases, additional external variables, known as **regressors**, can improve your model's accuracy. Prophet allows you to incorporate these regressors, which can be any time series that might influence your primary series. Adding regressors to Prophet is a simple process.

1. Let's add the regressors to your data frame and inform Prophet about this additional variable:

```
# Add additional regressors (external factors) to the model
m <- add_regressor(m, 'workingday')   # Whether it's a working day
m <- add_regressor(m, 'temp')         # Temperature
m <- add_regressor(m, 'atemp')        # Apparent temperature
m <- add_regressor(m, 'hum')          # Humidity
```

```
m <- add_regressor(m, 'windspeed')    # Wind speed
m <- add_regressor(m, 'weathersit')   # Weather situation category 2
```

2. Fit the Prophet model:

```
m <- fit.prophet(m, data)
```

When you fit the model and make future predictions, ensure your future data frame also includes the regressor.

3. To begin forecasting, you will extract the necessary regressors from the past data, ensuring that all the columns needed for the forecast are included. This will help you maintain a consistent dataset for both past and future dates:

```
regressors <- data %>% select(ds, holiday, workingday, weathersit, temp,
atemp, hum, windspeed)
```

4. Next, you will import the future regressors from a CSV file. This step brings in the future data, which contains the same columns needed for prediction. By doing this, you ensure continuity between past and future datasets:

```
future_regressors <- read.csv("Daily_Bike_Sharing_future.csv")
```

5. After importing the future data, select only the necessary columns that are required for forecasting. This reduces your data to only the important variables, making it more manageable:

```
future_regressors_selected <- future_regressors %>%
  select(dteday, holiday, workingday, weathersit, temp, atemp, hum,
windspeed)
```

6. You now need to rename the `dteday` column to `ds` so that the future regressors match the format Prophet expects for dates. This step aligns the data structure for future forecasting:

```
future_regressors_selected <- future_regressors_selected %>% rename(ds =
dteday)
```

7. Since the date format in the future data might not be compatible with Prophet, you will convert the `ds` column to `YYYY-MM-DD` format. This ensures that both past and future dates are in the same format:

```
future_regressors_selected$ds <- as.Date(future_regressors_selected$ds,
format = "%m/%d/%Y")
tail(future_regressors_selected)
```

This gives you the following:

```
           ds holiday workingday weathersit     temp    atemp
25 2012-12-26       0          1          3 0.243333 0.220333
26 2012-12-27       0          1          2 0.254167 0.226642
27 2012-12-28       0          1          2 0.253333 0.255046
28 2012-12-29       0          0          2 0.253333 0.242400
29 2012-12-30       0          0          1 0.255833 0.231700
30 2012-12-31       0          1          2 0.215833 0.223487
        hum windspeed
25 0.823333  0.316546
26 0.652917  0.350133
27 0.590000  0.155471
28 0.752917  0.124383
29 0.483333  0.350754
30 0.577500  0.154846
```

Figure 11.8: The future regressors dataset

8. Now that the future regressors are ready, you will combine them with the past data into one `combined_data` data frame. This combined dataset will be used to generate predictions for both past and future dates:

```
combined_data <- bind_rows(regressors, future_regressors_selected)
tail(combined_data)
```

The output is shown in the following figure:

```
           ds holiday workingday weathersit     temp
726 2012-12-26       0          1          3 0.243333
727 2012-12-27       0          1          2 0.254167
728 2012-12-28       0          1          2 0.253333
729 2012-12-29       0          0          2 0.253333
730 2012-12-30       0          0          1 0.255833
731 2012-12-31       0          1          2 0.215833
      atemp      hum windspeed
726 0.220333 0.823333  0.316546
727 0.226642 0.652917  0.350133
728 0.255046 0.590000  0.155471
729 0.242400 0.752917  0.124383
730 0.231700 0.483333  0.350754
731 0.223487 0.577500  0.154846
```

Figure 11.9: The combined dataset containing both past and future data

Notice the difference in the index.

9. Finally, use the `combined_data` to generate forecasts for the future dates. This step allows the model to predict based on both the past and future data you have prepared:

Filter

	ds	trend	atemp	atemp_lower	atemp_upper	extra_regressors_multiplicative
1	2011-01-01	1898.414	-0.037192483	-0.037192483	-0.037192483	-0.302309579
2	2011-01-02	1906.720	-0.040329128	-0.040329128	-0.040329128	-0.307526675
3	2011-01-03	1915.025	-0.092469266	-0.092469266	-0.092469266	-0.100161854
4	2011-01-04	1923.331	-0.085261582	-0.085261582	-0.085261582	-0.092546002
5	2011-01-05	1931.636	-0.079820839	-0.079820839	-0.079820839	-0.018767346
6	2011-01-06	1939.942	-0.078571067	-0.078571067	-0.078571067	0.001372676
7	2011-01-07	1948.247	-0.086303217	-0.086303217	-0.086303217	-0.204390537
8	2011-01-08	1956.553	-0.101083776	-0.101083776	-0.101083776	-0.462414942
9	2011-01-09	1964.858	-0.115703790	-0.115703790	-0.115703790	-0.380618308
10	2011-01-10	1973.164	-0.104689997	-0.104689997	-0.104689997	-0.148620373

Showing 1 to 10 of 731 entries, 40 total columns

Figure 11.10: Preview of the forecast dataset

By following these steps, you have prepared both the past and future data and used them to generate a seamless forecast. With the added complexity, the `forecast` now has 40 columns. But you can go on.

Advanced Prophet techniques

Prophet offers a range of advanced features that allow you to fine-tune your forecasts further. By incorporating holiday effects, implementing cross-validation, and tuning parameters, you can make your Prophet models more precise and tailored to your specific needs.

Holiday effects

Holidays and special events can significantly impact time series data. Prophet allows you to incorporate these effects to improve forecast accuracy:

1. First, create a data frame containing the dates and names of the holidays:

```
holidays <- data.frame(
  holiday = c('New Year', 'Christmas'),
  ds = as.Date(c('2012-01-01', '2012-12-25')),
  lower_window = 0,
  upper_window = 1
)
```

2. Next, include the holidays in your Prophet model:

```
m2 <- prophet(
  yearly.seasonality = TRUE,        # Include yearly seasonality
  weekly.seasonality = TRUE,        # Include weekly seasonality
  seasonality.mode = 'multiplicative', # Use multiplicative seasonality
  holidays = holidays,              # Adding the holidays to the model
  seasonality.prior.scale = 10,     # Increase flexibility of seasonality
  holidays.prior.scale = 10,        # Increase impact of holidays
  changepoint.prior.scale = 0.05    # Control flexibility in detecting
trend changes
)
```

By specifying the `lower_window` and `upper_window`, you can adjust the impact window of each holiday, capturing effects that span multiple days.

Cross-validation

To ensure the reliability of your forecasts, it is essential to implement cross-validation. Cross-validation tests your model's performance over different periods, providing a robust assessment of its accuracy.

Prophet includes built-in functions for cross-validation. First, use the `cross_validation` function to split your data into training and testing sets:

```
df_cv <- cross_validation(model, initial = 365, period = 15, horizon = 30, units
= 'days')
```

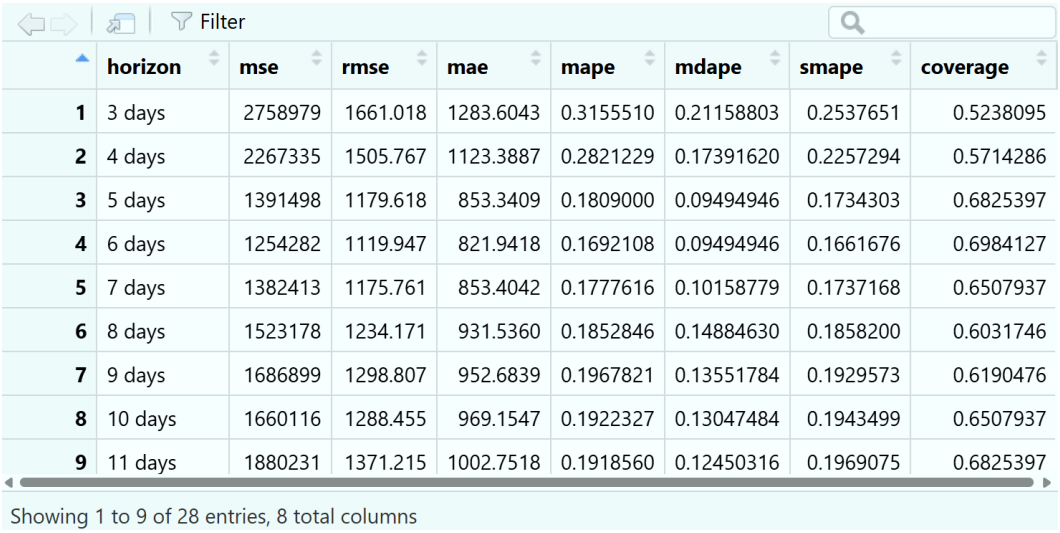
Let's break down the parameters used in the cross-validation setup for your Prophet model.

- `initial = 365`: This sets the size of the initial training period, meaning the model will first be trained using 365 days of data. In this case, it represents one year of historical data before validation begins.
- `period = 15`: This parameter controls how frequently the model refits and evaluates on new data. With a period of 15, the model will be evaluated every 15 days. It determines how often new forecasts are made within the cross-validation window.
- `horizon = 30`: The horizon sets the forecast period for each cross-validation step. Here, it means the model will predict 30 days into the future after each training period.
- `units = 'days'`: This specifies the units of the parameters. Since you are working with daily data, the units are set to `'days'`. This ensures that the `initial`, `period`, and `horizon` values are interpreted in terms of days.

Next, use the `performance_metrics` function to evaluate the model's accuracy:

```
df_p <- performance_metrics(df_cv)
```

You get the following output:



Filter

	horizon	mse	rmse	mae	mape	mdape	smape	coverage
1	3 days	2758979	1661.018	1283.6043	0.3155510	0.21158803	0.2537651	0.5238095
2	4 days	2267335	1505.767	1123.3887	0.2821229	0.17391620	0.2257294	0.5714286
3	5 days	1391498	1179.618	853.3409	0.1809000	0.09494946	0.1734303	0.6825397
4	6 days	1254282	1119.947	821.9418	0.1692108	0.09494946	0.1661676	0.6984127
5	7 days	1382413	1175.761	853.4042	0.1777616	0.10158779	0.1737168	0.6507937
6	8 days	1523178	1234.171	931.5360	0.1852846	0.14884630	0.1858200	0.6031746
7	9 days	1686899	1298.807	952.6839	0.1967821	0.13551784	0.1929573	0.6190476
8	10 days	1660116	1288.455	969.1547	0.1922327	0.13047484	0.1943499	0.6507937
9	11 days	1880231	1371.215	1002.7518	0.1918560	0.12450316	0.1969075	0.6825397

Showing 1 to 9 of 28 entries, 8 total columns

Figure 11.11: Performance metrics for the forecasting model

This approach splits your data into an initial training period, with subsequent periods used for testing, providing metrics such as MAE and RMSE to gauge forecast accuracy.

Parameter tuning

Tuning model parameters can significantly enhance forecast performance. Important parameters include `changepoint_prior_scale`, `seasonality_prior_scale`, and `holidays_prior_scale`. These parameters control the flexibility of the trend, seasonality, and holiday components, respectively. By experimenting with different values for these parameters, you can find the optimal balance between underfitting and overfitting.

To systematically explore multiple combinations of parameters, you can create a list of parameter combinations and use a simple loop to iterate through them. This approach helps identify the best parameter settings for your specific data.

Here is how you can implement parameter tuning with Prophet in R:

1. Define the range of parameter values to explore:

```
changepoint_priors <- c(0.01, 0.05, 0.1)
seasonality_priors <- c(1, 10, 20)
holiday_priors <- c(1, 10, 20)
```

2. Next, create a list to store the parameter combinations and initialize a list to store the performance metrics:

```
param_combinations <- expand.grid(changepoint_priors, seasonality_priors,
holiday_priors)

results <- list()
```

In this script, `expand.grid` creates a data frame with all combinations of the specified parameter values. The loop iterates through each combination, fits a Prophet model, performs cross-validation, and evaluates the performance using metrics such as RMSE. Finally, the best-performing set of parameters is identified and printed.

3. Now, loop through each combination of parameters, fit the Prophet model, perform cross-validation, and evaluate the performance metrics:

```
for (i in 1:nrow(param_combinations)) {
  # Extract the current combination of parameters
  cp <- param_combinations[i, 1]
  sp <- param_combinations[i, 2]
  hp <- param_combinations[i, 3]

seasonality if needed)
  m <- prophet(
    yearly.seasonality = TRUE,
    weekly.seasonality = TRUE,
    daily.seasonality = FALSE,
# Disable daily seasonality, or set to TRUE to enable
    seasonality.mode = 'multiplicative',
    holidays = holidays,
    changepoint.prior.scale = cp,
    seasonality.prior.scale = sp,
    holidays.prior.scale = hp)

  m <- add_regressor(m, 'workingday')   # Whether it's a working day
  m <- add_regressor(m, 'temp')         # Temperature
  m <- add_regressor(m, 'atemp')        # Apparent temperature
  m <- add_regressor(m, 'hum')          # Humidity
  m <- add_regressor(m, 'windspeed')    # Wind speed
  m <- add_regressor(m, 'weathersit')   # Weather situation category
```

4. Now, fit the model with the data:

```
  m <- fit.prophet(m, data)
```

5. Perform cross-validation:

```
  df_cv <- cross_validation(m,
                            initial = 365,
                            period = 15,
                            horizon = 30,
                            units = 'days')
```

6. Evaluate the performance metrics:

```
  df_p <- performance_metrics(df_cv)
```

7. Store the results with corresponding parameters:

```
  param_key <- paste0("cp=", cp, "_sp=", sp, "_hp=", hp)
  results[[param_key]] <- df_p
}
```

When tuning Prophet models, multiple forecasts are generated across different cutoffs to evaluate performance over varying time frames. The following output indicates that the tuning process is successfully running:

```
Making 21 forecasts with cutoffs between 2012-01-06 and 2012-11-01
Making 21 forecasts with cutoffs between 2012-01-06 and 2012-11-01
Making 21 forecasts with cutoffs between 2012-01-06 and 2012-11-01
Making 21 forecasts with cutoffs between 2012-01-06 and 2012-11-01
Making 21 forecasts with cutoffs between 2012-01-06 and 2012-11-01
```

Figure 11.12: Output from the tuning process

Tip

Do not be scared by the warning text! It means that it is working.

8. After running the loop, identify the best-performing set of parameters based on the performance metrics:

```
best_params <- NULL
best_performance <- Inf
for (param_set in names(results)) {
  current_performance <- mean(results[[param_set]]$rmse)
  if (current_performance < best_performance) {
    best_performance <- current_performance
    best_params <- param_set
  }
}
```

9. Print the best-performing parameters:

```
print(paste("Best parameters:", best_params))
print(paste("Best RMSE:", best_performance))

[1] "Best parameters: cp=0.01_sp=20_hp=1"
[1] "Best RMSE: 1253.72719297287"
```

By systematically tuning parameters, incorporating custom seasonalities, handling holiday effects, and implementing cross-validation, you can significantly improve your Prophet model's accuracy. These advanced techniques enable you to create highly customized and precise forecasts tailored to the unique characteristics of your time series data.

Summary

From understanding Prophet's origins and unique capabilities to implementing and fine-tuning models, you've covered a lot of ground. You now understand what makes Prophet unique and why it is a powerful tool for time series forecasting, especially with its ability to handle complex data patterns and outliers with ease.

You've learned how to install Prophet, fit it to your data, make future predictions, and visualize those forecasts. You can now confidently apply Prophet to your time series data. You also explored how to adjust changepoints, configure seasonal patterns, incorporate external regressors, and handle missing data to improve your forecasts' accuracy and relevance.

You dived into advanced features such as custom seasonalities, holiday effects, cross-validation, and parameter tuning. These skills enable you to create highly precise and tailored forecasts.

Next, you will dive into the exciting world of neural networks and their application to time series forecasting. You will learn how to build a simple neural network model in R, uncovering the structure of neurons and layers, and discover how activation functions such as sigmoid and ReLU help the network learn patterns.

12

Introducing Neural Networks and Time Series

Neural networks are a class of machine learning models inspired by the structure and function of the human brain. At their core are interconnected units called **neurons**, which process information by applying weights to inputs, summing them, and passing the result through an activation function. By stacking layers of these neurons—typically organized into input, hidden, and output layers—neural networks can learn complex, non-linear relationships within data. This makes them particularly powerful for tasks where traditional statistical models may struggle, especially when patterns are subtle or highly dynamic.

In this chapter, you will explore the fundamentals of neural networks and their application to time series forecasting. You will gain an understanding of the basic concepts, learn how to build and train a simple neural network model using R, and discover techniques to evaluate and optimize the model for better performance.

The following headings are covered in this chapter:

- Understanding the basics of neural networks
- Introducing simple neural networks
- Building a simple neural network model for time series in R
- Evaluating and optimizing simple neural network models

Technical requirements

R and RStudio are required for this chapter. GitHub scripts can be found in the repository.

You will need the following data file and packages for this chapter:

- `forecast`: The `forecast` package in R is a very popular package. It contains methods and tools for analyzing univariate time series forecasts. In this chapter, you will use the `nnetar` function from this package to fit a neural network autoregression model.
- `dplyr`: This R package is a powerful tool designed to make data manipulation and cleaning of tabular data (data frames) easier and faster.
- `DHS_Daily_Reports_2020.csv`: The CSV data file used in this chapter to demonstrate the simple neural network and parameter tuning for the neural network model.

Understanding the basics of neural networks

Neural networks (**NN**), inspired by the brain's architecture, are a class of machine learning models composed of interconnected units known as neurons. Each neuron receives input data, processes it through an activation function, and passes the result to subsequent neurons. The following diagram illustrates the human brain, the human neural network, and how human neurons interact with each other. It also shows the neuron structure from which the architecture was derived and enriched with layers and functions:

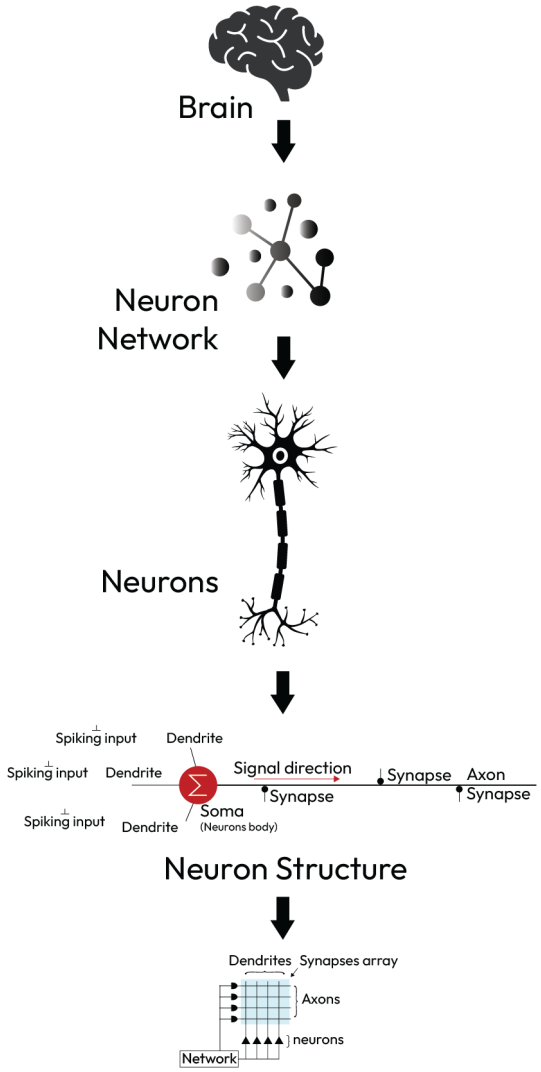


Figure 12.1: Illustration of human brain neurons and artificial neural network (Adapted from: `https://www.researchgate.net/figure/Neuron-networks-a-brain-b-neural-network-c-neuron-connecting-structure-d-neuron_fig1_320567646`*)*

This structure enables the network to learn and recognize complex patterns through iterative adjustments of connection weights, ultimately refining predictions based on training data.

A neuron in a neural network mimics the function of a biological neuron. It consists of **inputs**, **weights**, a **bias**, an **activation function**, and an **output**. Each input is multiplied by a corresponding weight, and the weighted inputs are summed together. A bias term is added to this sum to allow the model to better fit the data. The resulting value is then passed through an activation function, which determines the neuron's output. *Figure 12.2* is a diagram showing the components of a sample neuron.

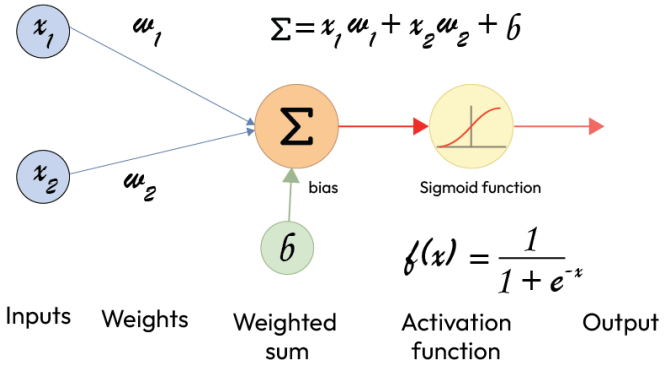


Figure 12.2: Components of neuron with inputs, weights, bias, activation function and output (Adapted from: `https://muneebsa.medium.com/deep-learning-101-lesson-7-perceptron-f6a698d81be8`*)*

The types of layers are as follows:

- **Input layer**: The input layer is the first layer of the neural network that receives the raw data. Each neuron in this layer represents a feature from the input data.
- **Hidden layers**: Hidden layers are the intermediate layers between the input and output layers. These layers perform complex computations and transformations on the input data through multiple neurons. The number and structure of hidden layers can vary, allowing the network to learn intricate patterns in the data.
- **Output layer**: The output layer is the final layer that produces the network's prediction or classification. The number of neurons in this layer typically corresponds to the number of desired output values.

The following image showcases the neural network layers:

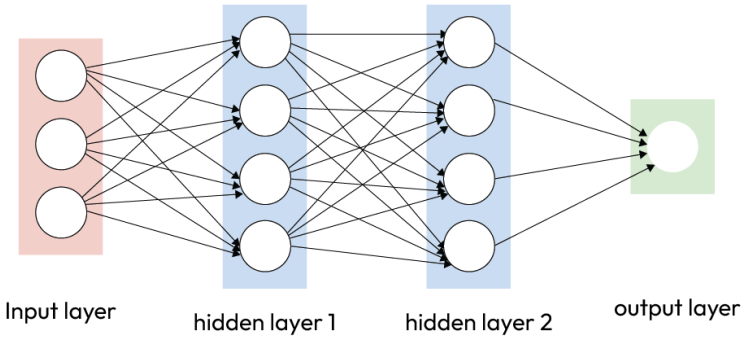


Figure 12.3: Illustration of neural network layers (Adapted from: `https://cs231n.github.io/neural-networks-1/`*)*

Activation functions

Activation functions are tools that allow neural networks to learn complex patterns. They determine whether a neuron should be activated by introducing nonlinearity into the network, enabling it to learn from data that is not linearly separable. Popular activation functions include **sigmoid**, **tanh**, and **ReLU**, each with unique properties that influence the network's learning behavior. These functions play a critical role in helping the network transform inputs into meaningful outputs, ensuring that it can generalize well to unseen data. Let's look at each of them in detail:

- **Sigmoid**: Also known as the logistic function, it maps input values to a range between 0 and 1, making it ideal for scenarios where you need to model probabilities. This smooth, S-shaped curve helps in deciding the likelihood of an output belonging to a particular class, especially in binary classification tasks. The following diagram illustrates the S-shaped curve of the Sigmoid function, showing how inputs are squashed into a range between 0 and 1. As input values increase positively or negatively, the function approaches 1 or 0 asymptotically, highlighting its "squashing" behavior.

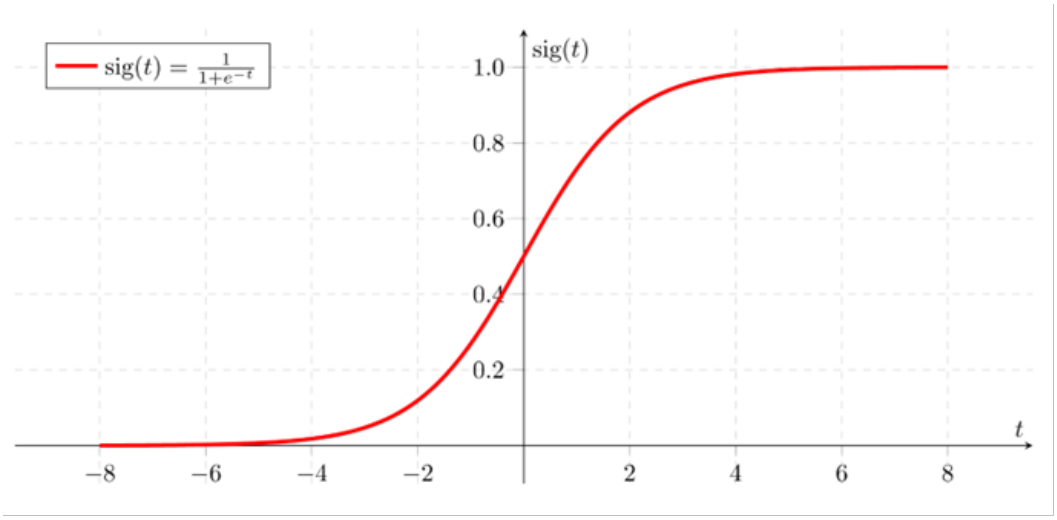


Figure 12.4: Illustration of Sigmoid function

- **Tanh**: The tanh (hyperbolic tangent) function is similar to **sigmoid** but outputs values in a range between -1 and 1. This zero-centered function is often preferred in hidden layers of neural networks because it tends to converge faster by centering data, which makes the optimization process more efficient.

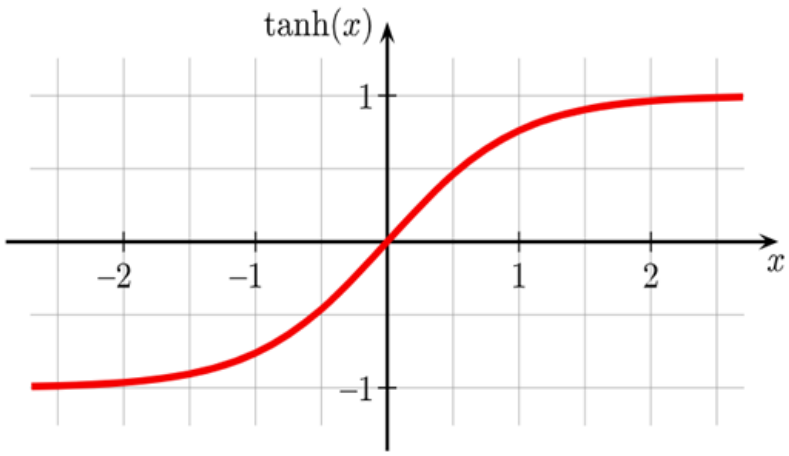


Figure 12.5: Illustration of tanh function, output between –1 and 1

- **ReLU**: The rectified linear unit (**ReLU**) is the most popular activation function in deep learning today. It outputs the input directly if it is positive. Otherwise, it returns zero.

This simple, piecewise-linear function is computationally efficient and helps models converge faster by keeping gradients from vanishing.

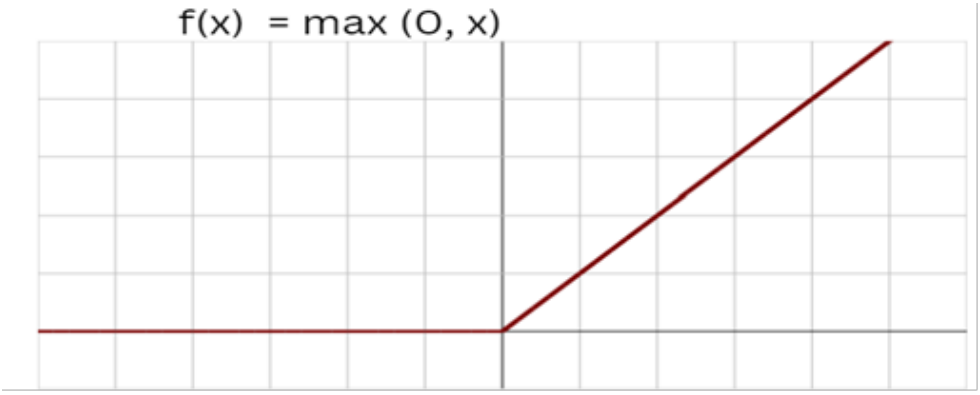


Figure 12.6: Illustration of ReLU function

By understanding the basic architecture and the role of activation functions in neural networks, you are well prepared to explore more about neural networks. Let's explore the simple neural network in the following section and do some practice.

Introducing simple neural networks

Feedforward neural networks are the foundational type of artificial neural network. In this structure, data flows in a single, forward direction, from input nodes through any hidden layers and ultimately to the output layer. This straightforward design is ideal for tasks where data is static and does not require recurrent processing, such as classification and regression tasks.

These networks lack cycles or loops, ensuring that information passes through only once. Primarily used for static data analysis, feedforward networks are well-suited for tasks where the relationships in the data do not change over time.

Let's consider the differences between simple and complex neural network structures.

Simple neural networks typically consist of an input layer, one or more hidden layers, and an output layer. They use straightforward architectures without intricate mechanisms. Examples include basic feedforward neural networks.

Complex neural networks have more sophisticated architectures, such as **convolutional neural networks** (**CNNs**) for image processing, **recurrent neural networks** (**RNNs**) for sequential data, and **long short-term memory networks** (**LSTMs**) for handling long-term dependencies in data. They incorporate advanced features such as weight sharing, gating

mechanisms, and multiple types of layers, which enable them to handle more complex tasks and larger datasets effectively.

Now that you have explored the differences between simple and complex neural network structures, let's move on to building a basic neural network model for time series forecasting.

Building a simple neural network model for time series in R

In this section, you will build a simple neural network model for time series forecasting using R. You will start with data preparation, move on to model construction using the `forecast` R library, and finally train and adjust your model to improve performance.

Data preparation

Before constructing and training the neural network, it is crucial to prepare the time series data properly. You will need to load the required libraries, including `dplyr` and `forecast`. These libraries have been installed and utilized in previous chapters.

1. Load the required libraries:

   ```
   library(dplyr)
   library(forecast)
   ```

   The `dplyr` package will help you with data manipulation, and the `forecast` packagewill be used later for time series operations.

2. Next, you import your dataset, named `DHS_Daily_Report_2020.csv`. This file contains daily records, including shelter population and certain holidays, which will be useful for your analysis:

   ```
   df <- read.csv("DHS_Daily_Report_2020.csv")
   ```

   You load the dataset from the csv file into a data frame. For a time series, you will convert the `Date` variable from a character format into a date format. This transformation ensures that the date data can be interpreted correctly in any time-based analysis or model training.

3. Transform the `Date` column to Date format:

   ```
   df$Date <- strptime(x = df$Date, format = "%m/%d/%Y")
   df$Date <- as.Date(df$Date)
   ```

4. After transforming the date, you select only the columns relevant to your model: `Date`, `Total.Individuals.in.Shelter`, major holidays, and `Temperature`. This step simplifies your dataset by focusing on variables that are likely to influence shelter usage:

```
df <- df %>% select(Date,
                    Total.Individuals.in.Shelter,
                    Easter,
                    Thanksgiving,
                    Christmas,
                    Temperature)
```

5. Finally, you rename the `Total.Individuals.in.Shelter` column to `y`, providing a simplified notation for the target variable in your model:

```
colnames(df)[2] <- "y"
```

With these steps complete, your data is now prepared and ready for further analysis and model training.

Model preparation

With the data prepared, the next step is to set up your model, train it on historical data, and evaluate its performance. You will start by creating training and testing datasets, transforming the time series data, and specifying the model architecture. Finally, you will generate predictions and measure the model's accuracy.

1. Split the data into training and test sets:

```
training <- df %>% filter(Date < '2020-12-01')
test <- df %>% filter(Date >= '2020-12-01')
```

You begin by dividing the dataset into training and test sets. The training set includes data up to December 1, 2020, while the test set includes all subsequent dates. This separation allows us to train the model on historical data and evaluate it on unseen data.

2. Next, you transform the target variable into a time series object with a `frequency` of `7`, indicating daily data. Adjusting this frequency is essential to ensure the model aligns with the data's periodicity. `52` would be weekly, `12` would be for monthly, `4` for quarterly, and `5` for weekdays only. You are choosing daily here:

```
training_y <- ts(training$y, frequency = 7)
```

3. Here, you extract additional variables (regressors) from the training and test sets, converting them into matrices. These regressors provide supplementary information, such as holiday and temperature data, which will help improve the model's predictive accuracy:

```
training_reg <- as.matrix(training[, 3:ncol(training)])
test_reg <- as.matrix(test[, 3:ncol(test)])
```

4. Now, you set up and train the neural network model using the `nnetar` function from the forecast package. You specify the parameters, including the target time series you want to forecast (`y=training_y`), the number of non-seasonal lagged observations used as input (`p=1` means the model uses only the most recent observation to help predict the next value), seasonal terms (`P=1` means the model includes one observation from the previous seasonal cycle), hidden layer size (`size=3`), regularization factor (`decay=0.1`), and the external regressors (additional predictors, `xreg=training_reg`). These parameters help shape the model's structure and control overfitting:

```
model <- nnetar(y = training_y,
                p = 1,
                P = 1,
                size = 3,
                xreg = training_reg,
                decay = 0.1)
```

5. With the model trained, you use it to make predictions on the test set. The `forecast` function generates the predicted values, leveraging the trained model and test set regressors:

```
predictions <- forecast(model, xreg = test_reg)
```

6. Finally, you assess the model's performance by comparing the predictions against the actual values in the test set. The `accuracy` function provides metrics to evaluate how well the model forecasts the time series data:

```
accuracy(predictions$mean, test$y)

                ME     RMSE      MAE       MPE      MAPE
Test set 458.2667 522.2198 470.5185 0.8557917 0.8786671
```

   The accuracy results will print out the `ME` (mean error), `RMSE` (root mean squared error), `MAE` (mean absolute error), `MPE` (mean percentage error), and `MAPE` (mean absolute percentage error). Although `ME`, `RMSE`, and `MAE` seem to have large values, the `MPE` (0.86%) and `MAPE` (0.88%) are less than 0.9%. Overall, your neural network model appears to perform very well on the test set.

By following these steps, you complete the model preparation, training, and evaluation processes, establishing a neural network model that is ready for time series forecasting. Building a simple neural network model for time series forecasting in R involves careful data preparation, setting up an appropriate model architecture using the `forecast` library, and training the model effectively.

Evaluating and optimizing simple neural network models

After constructing and training a simple neural network model for time series forecasting, the next crucial steps are to evaluate its performance and optimize it for better accuracy and reliability. This section covers optimization techniques, methods to prevent overfitting, and touches on advanced topics for further enhancements.

Optimization is essential to improve the performance of your neural network model. Here, you will focus on basic hyperparameter tuning and techniques to enhance model performance. Hyperparameter tuning within the `nnetar` function involves adjusting parameters to find the configuration that yields optimal accuracy.

The following steps implement cross-validation and tune the hyperparameters for your neural network model:

1. Generate dates:

```
last_date <- tail(df$Date, 1) - 30
first_date <- tail(df$Date, 1) - 200
dates <- seq.Date(from = first_date,
                  to = last_date,
                  by = "week")
```

   This code will define the date range and weekly date sequence.

2. The results vector will be created and used to store the performance of the models:

```
results_cv <- vector(mode = 'numeric',
                     length = length(dates))
```

3. Create the parameter grid:

```
Grid <- expand.grid(p = seq(from = 1, to = 6, by = 1),
                    P = c(1,2,3),
                    decay = c(0.05, 0.1, 0.15, 0.2),
                    size = c(2,3,4))
```

   In practice, the `p`, `P`, `decay`, and `size` are defined. You will try to find the optimal combination on the parameters combination from them (total combination count will be 6x3x4x3 = 216).

4. Create a results grid to store the mean value of the performance measure during the model iterations:

```
results_grid <- vector(mode = 'numeric', length = nrow(Grid))
```

5. Now it's the time to perform parameter tuning. The code block will loop through the defined parameters (`p`, `P`, `decay`, `size`), and the inner loop will iterate through the dates (weekly), create the training and test datasets on the fly based on the date, and create the target time series object.
   The following code will loop through each forecast origin, create a new training and test dataset, convert the target variable into a time series object, and prepare the

regressors. The regressors (`training_reg` and `test_reg`) are similar to those in the previous code for a simple neural network:

```
for (i in seq_len(nrow(Grid))) {
  parameters <- Grid[i, ]

  #Cross Validation
  for (j in 1:length(dates)) {

    #Training and test set
    training <- df %>% filter(Date <= dates[j])
    test <- df %>% filter(Date > dates[j] & Date <= (dates[j] + 31))

    #Time Series Object
    training_y  <- ts(training$y, frequency = 7)

    #Regressors
    training_reg <- as.matrix(training[, 3:ncol(training)])
    test_reg <- as.matrix(test[, 3:ncol(test)])
```

6. The parameters in the loop will be passed to `nnetar` to create and train the neural network model. The model will be used to predict the test set. The `p` parameter specifies the number of non-seasonal lagged observations, `P` specifies the number of seasonal lagged observations, `size` controls the number of neurons in the hidden layer, and `decay` applies weight regularization during training. Once the model has been fitted, forecasts are generated for the test dataset using the corresponding regressors:

```
    #Model
    model <- nnetar(y = training_y,
                    p = parameters$p,
                    P = parameters$P,
                    decay = parameters$decay,
                    size = parameters$size,
                    xreg = training_reg)

    #Predictions
    predictions <- forecast(model, xreg = test_reg)
```

7. The evaluation on the model (with parameters in the code loop) will be done by `accuracy` function like what you did in early of this chapter. The **RMSE** (`accuracy[1,2]`) will be stored in the `results_cv` vector and averaged after all validation windows have been evaluated. This average RMSE represents the overall forecasting performance for the current hyperparameter combination.
The process is then repeated until every combination in the tuning grid has been evaluated, allowing the model with the lowest average RMSE to be selected:

```
    #Accuracy
    results_cv[j] <- accuracy(predictions$mean, test$y)[1,2]
    print(dates[j])

  }

  #Store Results Grid
  results_grid[i] <- mean(results_cv)
  print(i)
}
```

Note

Please note that this loop process might take longer than expected, depending on your computer's computing power. Sometimes, it could take more than two hours.

8. Fetch the optimal parameter combination:

```
Grid <- cbind(Grid, results_grid)
best_params <- Grid[Grid$results_grid == min(results_grid), ]
print(best_params)


    p P decay   size   results_grid
147 3 1  0.05    4     427.1894
```

The code will fetch the results from the `results_grid`, finding the optimal parameters based on the lowest `RMSE` value. The optimal parameters found are `p=3`, `P=1`, `decay=0.05`, and `size=4`. The optimal `RMSE` value is `427.19`.

Evaluating and optimizing a simple neural network model involves a combination of hyperparameter tuning, performance improvement techniques, and methods to prevent overfitting. By implementing these strategies, you can significantly enhance the accuracy and reliability of your time series forecasts. As you advance, exploring more sophisticated techniques and architectures will further refine your modeling capabilities.

Summary

In this chapter, you explored the fundamentals of neural networks and their application to time series forecasting. You began by understanding the basic concepts and structure of neural networks, including neurons, layers, and activation functions. You then explored the construction and training of a simple neural network model for time series forecasting using the `forecast` R library. This involved preparing the data, setting up the model architecture, and training the model effectively.

You also covered essential techniques for evaluating and optimizing the model. This included hyperparameter tuning, methods to improve model performance, and strategies to prevent overfitting. These optimization techniques are crucial for enhancing the accuracy and reliability of the neural network model.

Lastly, you touched on advanced topics that can further refine your modeling capabilities. By exploring ensemble methods, advanced neural network architectures such as LSTMs, and transfer learning, you can tackle more complex time series forecasting challenges and achieve even greater predictive accuracy.

By mastering these concepts and techniques, you are well equipped to leverage the power of neural networks for time series analysis. This foundational knowledge provides a strong basis for further exploration and application of more advanced neural network models and techniques in your data science endeavors.

In the next chapter, you will explore advanced techniques for evaluating and improving the performance of LSTM models in real-world scenarios. Building on your foundational understanding of LSTM networks, the chapter will focus on practical methods for optimizing model performance, including hyperparameter tuning, advanced regularization techniques, and innovative data preprocessing approaches.

13

Deep Learning with LSTM Models

In this chapter, you will dive deeper into a topic touched on previously in *Chapter 12*: long short-term memory (LSTM) networks, a powerful and specialized type of recurrent neural network (RNN) designed to excel in handling sequential and time-dependent data. LSTM networks are particularly effective in overcoming the limitations of traditional RNNs, making them invaluable tools for tasks that involve long-term dependencies, such as time series forecasting.

You will learn about the core architecture of LSTM networks, including the cell state, input gate, forget gate, and output gate, and discover how these components work together to preserve long-term dependencies and improve forecasting accuracy. The chapter will also examine the strengths and limitations of LSTM models, helping you understand why they often outperform traditional Recurrent Neural Networks (RNNs) when working with complex, nonlinear time series data.

Building on this theoretical foundation, the chapter will guide you through the practical implementation of LSTM models in R for time series forecasting applications. You will learn how to build, train, and optimize LSTM networks by selecting suitable parameters and applying advanced modeling techniques. In addition, the chapter will introduce key strategies for improving model performance in real-world scenarios, including tuning epochs, hidden dimensions, and input chunk lengths, as well as applying regularization methods such as dropout and early stopping to reduce overfitting and improve model generalization.

These are the headings covered in the chapter:

- Introducing RNN
- Introducing LSTM networks

- Implementing LSTM in R
- Tuning LSTM models for optimal performance

Technical requirements

R and RStudio are required for this chapter. GitHub scripts can be found on the repo.

You will need the following data file and packages for this chapter:

- `torch`: The `torch` package in R is an open-source machine learning framework that brings the power of PyTorch natively to the R ecosystem. You will use the package to learn about LSTM in this chapter.
- `dplyr`: This R package is a powerful tool designed to make data manipulation and cleaning of tabular data (data frames) easier and faster. You will use functions such as `mutate()` in the code snippets.
- `electricity.csv`: You will use the electricity demand dataset to demonstrate how to build LSTM forecasting models using the torch package in R. The dataset contains 1,680 hourly observations collected over approximately 70 days. The objective is to forecast future electricity demand based on historical demand and additional external information.

Introducing RNNs

As you learned in *Chapter 12*, the simple neural network (feedforward network) treats inputs separately, which provides a mechanism for quick classification or regression tasks where each data point is independent. However, in a situation where context matters (for example, stock market prediction or electricity consumption), the limitations of simple neural networks become obvious.

Recurrent neural networks (**RNNs**) are designed to handle sequential data, making them particularly powerful for tasks where context and order matter, such as time series forecasting, natural language processing, and speech recognition. Unlike traditional neural networks, where the output depends only on the current input, RNNs maintain a connection between previous inputs and the current output. This allows the network to capture dependencies over time, where each output is influenced not just by the current input but by a sequence of prior inputs as well. Let's take a look at the RNN structure first.

Typical structure of an RNN

An RNN typically consists of an input layer, one or more hidden layers, and an output layer. The key innovation of RNNs is the hidden state, which serves as a memory that captures

information from previous time steps. As the network processes each input in the sequence, it updates the hidden state, which then influences the subsequent outputs. This recursive loop within the hidden layer enables the network to carry information forward, allowing it to model temporal dynamics effectively.

The following diagram illustrates the structure of an RNN:

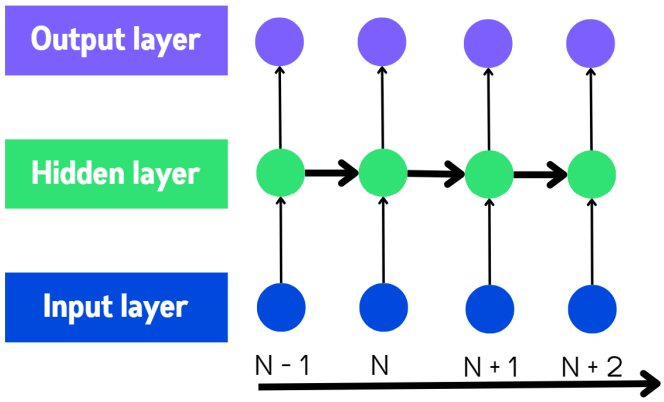


Figure 13.1: Illustration of an RNN structure

In this diagram, each node in the input layer feeds into the hidden layer for a particular time step (**N-1**, **N**, **N+1**, **N+2**), and the hidden state from the previous time step is passed forward. Notice how the hidden layer nodes are connected horizontally, representing the flow of information across time steps. The output layer generates the final output for each time step, influenced by both the current input and the hidden state.

By understanding this structure, you can appreciate how RNNs capture sequential dependencies, making them indispensable for analyzing data that unfolds over time.

Limitations of traditional RNNs

Despite their utility, traditional RNNs have some critical limitations that can hinder their performance, especially when it comes to learning from long sequences of data. Two of the most significant challenges are the vanishing gradient problem and the issue of short-term memory. These challenges prevent RNNs from effectively capturing long-term dependencies, which are crucial for many real-world applications. Let us understand these challenges in the following sections.

The vanishing gradient problem

The vanishing gradient problem is a fundamental issue that arises during the training of RNNs, particularly when dealing with long sequences of data.

During backpropagation, the gradients used to update the network's weights are passed backward through the network, not only through the layers but also across the time steps. As these gradients are propagated backward, they can shrink exponentially, especially if the network's weights are small. This shrinking of the gradients means that the earlier layers, which are critical for learning long-term dependencies, receive only minimal updates. As a result, the network struggles to learn from inputs that are far back in the sequence.

This vanishing gradient effect severely impacts the network's ability to learn effectively from long sequences. Since the gradients diminish as they are propagated backward, the influence of earlier inputs in the sequence is lost. This means the RNN becomes biased toward more recent inputs, ignoring crucial information from earlier in the sequence.

For example, in time series forecasting, this can lead to poor predictions because the network fails to account for important trends or patterns that occurred earlier in the sequence. This is shown in *Figure 13.2*:

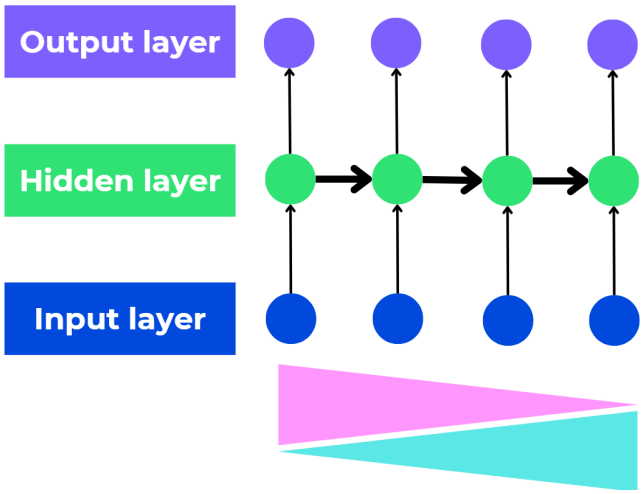


Figure 13.2: Diagram of an RNN structure

In the preceding figure, the arrows represent the flow of gradients during backpropagation in an RNN. As the gradient moves backward through the time steps, it diminishes, which is visually depicted by the tapering size of the arrows.

The lower part of the image shows how this diminishing gradient leads to weaker updates to the earlier layers, resulting in the vanishing gradient problem.

Short-term memory issue

Due to the vanishing gradient problem, traditional RNNs are prone to a short-term memory bias. This means that they tend to focus more on recent inputs, giving them disproportionate

importance while neglecting earlier information in the sequence. This short-term memory issue significantly undermines the RNN's ability to capture long-term dependencies in the data.

Consider a time series forecasting scenario where the network needs to predict future values based on historical data. If the RNN gives too much weight to recent inputs and disregards earlier trends or cycles, the resulting predictions are likely to be inaccurate. For instance, if a financial model only considers the most recent market fluctuations while ignoring broader economic trends from months or years ago, its forecasts could be seriously flawed.

With these limitations in mind, let's find out how LSTMs can help overcome them.

Introducing LSTM networks

Long Short-Term Memory (**LSTM**) networks are a special kind of RNN that are explicitly designed to address the limitations of traditional RNNs, particularly the vanishing gradient problem and short-term memory issue. LSTMs introduce a more complex architecture that allows them to capture and maintain long-term dependencies effectively.

The key components of an LSTM are as follows:

- **Cell state**: The cell state is the core of the LSTM, acting as a conveyor belt that carries information through the network. It is the memory of the network, which allows it to remember long-term dependencies.
- **Input gate**: The input gate controls how much of the new input should influence the cell state. It selectively adds new information to the cell state based on the relevance of the input.
- **Forget gate**: The forget gate decides what portion of the past information stored in the cell state should be discarded. This gate is crucial for eliminating irrelevant data that might otherwise clutter the network's memory.
- **Output gate**: The output gate determines what information from the cell state should be output at each time step. It essentially controls what the network uses to make predictions or pass to the next layer.

These gates work in unison to regulate the flow of information through the network. The input and forget gates work together to update the cell state by adding relevant new information and removing outdated or irrelevant data. The output gate then determines what part of the cell state should influence the current output, ensuring that the network maintains important long-term dependencies while being flexible enough to adapt to new inputs. Let's take a look at how the LSTM handles the problem RNNs have in the next section.

How LSTM overcomes RNN limitations

LSTM networks are specifically designed to tackle the issues that traditional RNNs face:

- **Preventing the vanishing gradient problem**: The architecture of the LSTM, with its gates and cell state, allows the network to retain gradients over many time steps, effectively mitigating the vanishing gradient problem. The constant flow of the cell state, modulated by the forget and input gates, ensures that gradients do not diminish over time, enabling the network to learn from long-term dependencies.
- **Addressing the short-term memory issue**: The forget gate plays a critical role in selectively discarding information that is no longer relevant, which helps the network focus on important data without being overwhelmed by irrelevant details. This selective forgetting ensures that the network maintains a balance between short-term and long-term memory, effectively overcoming the short-term memory bias of traditional RNNs.

You can see how the LSTM differs from traditional RNNs in the following diagram:

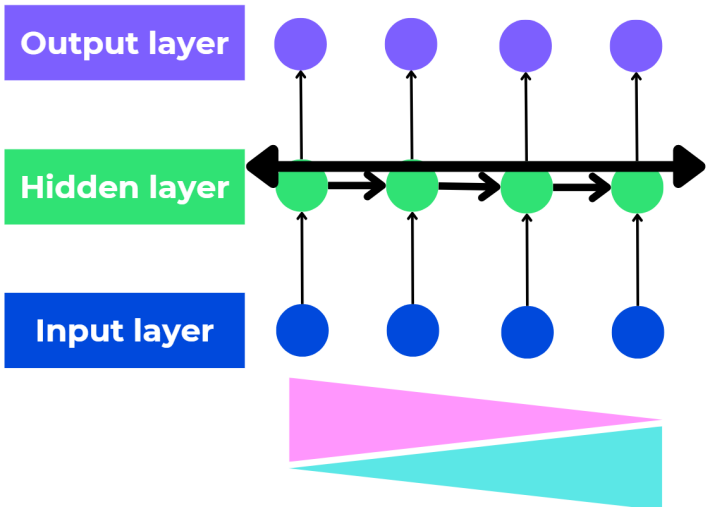


Figure 13.3: Illustration of a LSTM network information flow

The diagram illustrates the flow of information through an LSTM network. The thick black arrow represents the cell state, which flows through the network, carrying important information across time steps. The shaded layers show the various gates (input, forget, and output) that regulate the flow of information, ensuring that the network maintains long-term dependencies while adapting to new inputs.

LSTM excels in time series forecasting

One of the standout features of LSTM networks is their ability to retain gradients over long sequences, which makes them exceptionally good at learning from time-dependent data. In traditional RNNs, gradients often diminish as they are propagated backward through time steps, leading to the vanishing gradient problem.

This limitation prevents RNNs from learning long-term dependencies effectively. LSTM networks, however, with their gated structure, maintain gradients more effectively, allowing them to learn from data points far back in the sequence without losing information.

This ability to maintain long-term dependencies is crucial for time series forecasting, where the network must understand patterns that evolve over time.

For example, when forecasting financial markets, weather patterns, or product demand, the LSTM network can leverage its memory to capture trends, cycles, and seasonality that occur over extended periods. This gives LSTM a significant advantage over traditional RNNs and other models, particularly in scenarios where the data is nonlinear and complex. Let's walk through how LSTM addresses common challenges in time series data and better understand the feature LSTM introduces.

Addressing common challenges in time series data

Time series data often presents unique challenges, such as the presence of outliers and nonlinear patterns that can distort predictions. LSTM networks are robust in handling these challenges due to their architecture.

LSTM networks, with their ability to selectively forget irrelevant information via the forget gate, can effectively manage outliers. This prevents extreme values from disproportionately influencing the network's predictions, ensuring more stable and accurate forecasting.

Time series data is often nonlinear, with patterns that cannot be easily captured by simpler models. LSTM networks, through their dynamic gating mechanisms, are capable of learning these nonlinear relationships, making them powerful tools for real-world forecasting tasks where patterns and dependencies are complex and not immediately obvious.

LSTM networks are exceptionally well suited for time series forecasting due to their sophisticated architecture, which allows them to manage information flow, retain long-term dependencies, and handle the challenges inherent in sequential data. Whether dealing with non-linear patterns, outliers, or complex dependencies, LSTM provides the tools necessary for accurate and reliable forecasting.

Next, you will use LSTM in R.

Implementing LSTM in R

The following is a step-by-step guide to constructing and training an LSTM model in R. You will use the `torch` package.

Note

There are other packages available to achieve similar results. For example, the `keras` package provides an interface to an LSTM model.

1. You will use the `torch` and `dplyr` packages in the following scripts, so first let's install and load them:

```
if (!requireNamespace("torch", quietly = TRUE)) {
  # If not installed, install it
  install.packages("torch")
}
library(torch)

library(dplyr)
```

2. Load the data file. You will use the `electricity.csv` file. Make sure you set up your RStudio working directory so that it can locate the file. The file is an electricity time series from 10/22/2016 to 12/30/2016. You will use the data for the following code:

```
data <- read.csv("electricity.csv")
head(data)

                ds     y Exogenous1 Exogenous2
1 10/22/2016 0:00 70.00      49593      57253
2 10/22/2016 1:00 37.10      46073      51887
3 10/22/2016 2:00 37.10      44927      51896
4 10/22/2016 3:00 44.75      44483      48428
5 10/22/2016 4:00 37.10      44338      46721
6 10/22/2016 5:00 35.61      44504      46303
```

3. The data frame contains the following columns: `y`, `Exogenous1`, `Exogenous2`, the values range for `y` are between 10 to 700, while `Exogenous1` and `Exogenous2` are between

41000 to 82300. They are not on the same scale. In machine learning, a common technique is to normalize the data so that the columns are on the same scale.
The following code uses the `dplyr` package and the `across()` function to apply the same `scale` function to all three columns: `y`, `Exogenous1`, and`Exogenous2`. The `scale()` function transforms (normalizes) each column to have the value of z score (which is the value minus the mean divided by the standard deviation). After the transformation, the three columns will be on the same scale:

```
data <- data %>% mutate(across(c(y, Exogenous1, Exogenous2), scale))
head(data)

               ds          y  Exogenous1 Exogenous2
1 10/22/2016 0:00  0.2681882  -1.244896 -0.6137915
2 10/22/2016 1:00 -0.5827830  -1.814493 -1.2255551
3 10/22/2016 2:00 -0.5827830  -1.999936 -1.2245290
4 10/22/2016 3:00 -0.3849128  -2.071783 -1.6199066
5 10/22/2016 4:00 -0.5827830  -2.095246 -1.8145172
6 10/22/2016 5:00 -0.6213224  -2.068385 -1.8621722
```

4. The next step is to split the normalized data into training and test sets. You will set it to 80% training and 20% testing:

```
set.seed(123)  # for reproducibility
train_size <- floor(0.8 * nrow(data))
train_data <- data[1:train_size, ]
test_data <- data[(train_size + 1):nrow(data), ]
```

5. Define a function to create a sequence for LSTM input:

```
create_sequences <- function(data, time_steps) {
  X <- list()
  Y <- list()

  for (i in 1:(nrow(data) - time_steps)) {
    # Collect sequences of length `time_steps`
    X[[i]] <- as.matrix(data[i:(i + time_steps - 1), c("y", "Exogenous1",
"Exogenous2")])
    Y[[i]] <- data$y[i + time_steps]
  }
```

```
  # Convert lists to torch tensors
  X <- torch_tensor(unlist(X), dtype = torch_float32())$view(c(length(X),
time_steps, 3))
  Y <- torch_tensor(unlist(Y), dtype = torch_float32())$view(c(length(Y),
1))

  list(X = X, Y = Y)
}
```

Unlike traditional machine learning algorithms, **LSTM** models learn from sequences rather than individual observations. This step converts the time series into overlapping sequences using a sliding-window approach. Each input sequence consists of the specified number of consecutive time steps containing the target variable and both exogenous variables, while the corresponding output is the target value immediately following the sequence.

The resulting sequences are then converted into torch tensors, the data structure required by the `torch` package for model training. You will use the function shortly.

6. Use the previous train and test data set, utilize the function `create_sequences` created in the preceding code blocks, and prepare the data input for the `LSTM` model. The `time_steps` parameter is set to `5` so that each input sequence consists of five consecutive time steps:

```
time_steps <- 5
train <- create_sequences(train_data, time_steps)
test <- create_sequences(test_data, time_steps)
```

7. In the `torch` package, you use the `nn_module()` functionto define the entire neural network model and use the `nn_lstm()` function to create the LSTM layer. As shown in the code scripts listed below, the LSTM layer will take the following parameters:
   - `input_size`: The number of input features
   - `hidden_size`: The number of hidden units (memory cells) in the LSTM layer
   - `num_layers`: The number of stacked LSTM layers
   - `output_size`: Size of output

   Let's create it now:

```
lstm_model <- nn_module(
  # Initialize the model's layers and structure
```

```
  initialize = function(input_size, hidden_size,num_layers,output_size) {
    # LSTM layer: takes input sequences and produces hidden states
    self$lstm <- nn_lstm(input_size = input_size, hidden_size =
hidden_size, num_layers = num_layers, batch_first = TRUE)
    # Fully connected layer to produce final output
    self$fc <- nn_linear(hidden_size, output_size)
  },

  # Define the forward pass of the model
  forward = function(x) {
    # Pass the input `x` through the LSTM layer and retrieve only the
output tensor
    lstm_out <- self$lstm(x)[[1]]
    # Take the last hidden state and pass it through the fully connected
layer
    output <- self$fc(lstm_out[ , -1, ])
    output  # Return the output
  }
)
```

8. Define the model parameters. These will be used to specify the model:

```
input_size <- 3      # Number of features (`y`, `Exogenous1`, `Exogenous2`)
hidden_size <- 50    # Number of hidden units in the LSTM layer
num_layers <- 1 # Number of stacked LSTM layer
output_size <- 1     # Size of output (predicting a single value `y`)
```

9. Using the preceding function, you initialize the `LSTM` model, passing the parameters you just defined:

```
model <- lstm_model(input_size, hidden_size, num_layers, output_size)
```

10. Meanwhile, you will need a loss function for the model. The loss function is used to quantify the difference between the predicted output and the actual value. Here, you use the mean squared error (`MSE`), which is a widely used loss function.
You also specify the optimizer, which is an algorithm that adjusts an LSTM's internal weights to minimize prediction errors, and the `Adam` (**Adaptive Moment Estimation**)

optimizer is the most widely used and effective optimizer for this task, it will adjust learning rates during training:

```
loss_fn <- nnf_mse_loss # Mean Squared Error loss for regression
optimizer <- optim_adam(model$parameters, lr = 0.01)
```

11. Next, you have to train the model. You first set up the epoch value, which is how many rounds you want the model to be trained on the data. Since this is still a very simple data set, it will run the training loop very quickly and console output will show up quickly:

```
epochs <- 200

for (epoch in 1:epochs) {

model$train() # Set the model to training mode
optimizer$zero_grad() # Reset gradients for each epoch
#Forward pass: make predictions on training data
predictions <- model(train$X)
```

12. Compute the loss between predictions and actual target values:

```
loss <- loss_fn(predictions, train$Y)
```

13. Next, perform a backward pass to compute the gradients:

```
loss$backward()
```

14. Update model parameters:

```
optimizer$step()
```

15. Print loss every 10 epochs for monitoring:

```
if (epoch %% 10 == 0) { cat("Epoch:", epoch, " Loss:", loss$item(), "\n")
  }
 }
```

During model training, the loss value was monitored every 10 epochs to evaluate learning progress. In the following layout, the loss steadily decreased from 0.9496 to

0.048 at epoch 200, indicating that the LSTM model successfully learned the underlying temporal patterns in the training data.

```
Epoch: 10 Loss: 0.9496278
Epoch: 20 Loss: 0.768261
Epoch: 30 Loss: 0.6810883
Epoch: 40 Loss: 0.5714133
Epoch: 50 Loss: 0.5008591
Epoch: 60 Loss: 0.4406632
Epoch: 70 Loss: 0.3728359
Epoch: 80 Loss: 0.2852298
Epoch: 90 Loss: 0.1983767
Epoch: 100 Loss: 0.146183
Epoch: 110 Loss: 0.1425372
Epoch: 120 Loss: 0.09438557
Epoch: 130 Loss: 0.07661864
Epoch: 140 Loss: 0.06764557
Epoch: 150 Loss: 0.06221233
Epoch: 160 Loss: 0.0578345
Epoch: 170 Loss: 0.05461572
Epoch: 180 Loss: 0.05194827
Epoch: 190 Loss: 0.04970806
Epoch: 200 Loss: 0.04768897
```

16. After the model is trained, you set it to evaluation mode and use it to predict the test set:

```
model$eval()

predictions <- model(test$X)
predicted_values <- as.numeric(predictions)
actual_values <- as.numeric(test$Y)
```

17. With 200 rounds of training and the parameters you configured, let's assess the model performance using `MSE` (you can use others, such as mean absolute error `MAE`, mean absolute percentage error `MAPE`, and so on.) You will see the MSE printed in the console.

From the training loop, we get `MSE` at 0.0477, and the test `MSE` is 0.0245. They are very close, and it seems that the model works consistently for the training and test sets:

```
mse <- mean((predicted_values - actual_values)^2)
print(paste("Test MSE:", mse))

[1] "Test MSE: 0.0244781808551984"
```

18. You then plot the predicted values against the actual values and visualize the data on a chart:

```
plot(actual_values, type = "l", col = "blue", main = "LSTM Predictions vs
Actual", ylab = "y")
lines(predicted_values, col = "red")
legend("topright", legend = c("Actual", "Predicted"), col = c("blue",
"red"), lty = 1)
```

This outputs the following graphic:

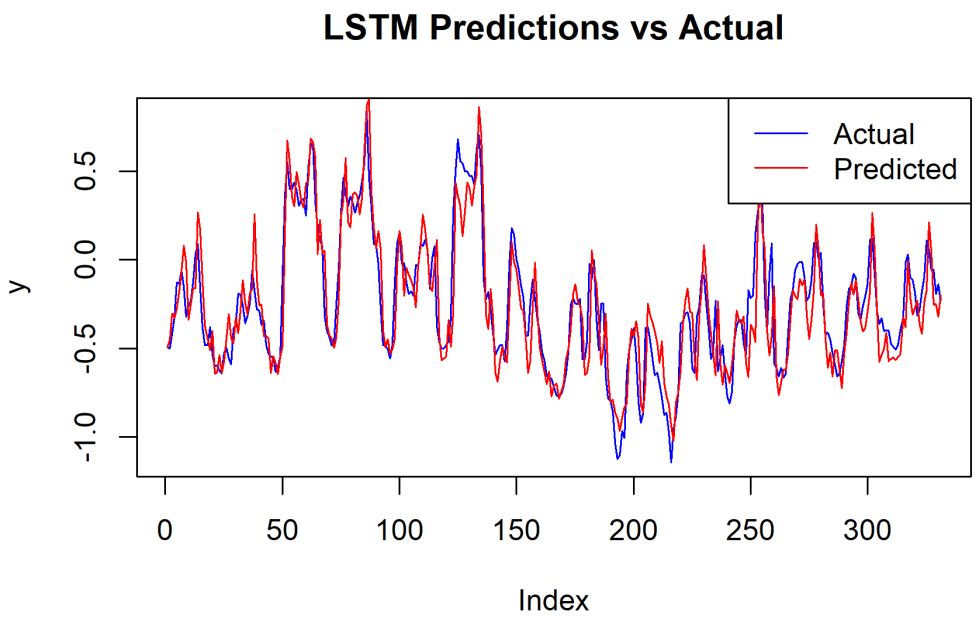


Figure 13.4: Visualization of the LSTM prediction vs actual

The LSTM models provide a powerful approach for time series forecasting by learning complex temporal patterns and long-term dependencies from sequential data. Compared with traditional forecasting methods that rely on predefined assumptions, LSTM networks can capture nonlinear relationships and interactions among variables, making them especially useful for complex forecasting problems.

However, model performance depends heavily on the selection of architecture and training parameters, such as the number of hidden units, layers, learning rate, and dropout. In the next section, you will explore how to tune LSTM model parameters systematically to improve forecasting accuracy and identify the optimal model configuration.

Tuning LSTM models for optimal performance

To achieve the best results from an LSTM model, fine-tuning is essential. This process involves adjusting key parameters to ensure that the model not only fits the training data well but also generalizes effectively to new, unseen data.

You will explore how to tune critical parameters such as hidden dimensions, the number of layers, learning rate, and dropout percentage. You will also discuss strategies to avoid overfitting and ensure robust model performance:

- **Hidden dimension:** The number of LSTM units (or neurons) in each hidden layer determines the model's capacity to learn. More units can capture more complex patterns but also increase the risk of overfitting.
  Start with a smaller number of units (for example, 30 to 50) and gradually increase if the model is underfitting (meaning that it fails to capture the underlying patterns). Use cross-validation to determine the optimal number of units.
- **Number of layers:** The number of stacked LSTM layers. The first LSTM processes the raw sequence data and passes its hidden sequence output as input to the next layer. The default value is set to 1.
  In practice, begin with `num_layers` = 1 or 2. Most sequential tasks can be handled by 1–2 layers.
- **Learning rate:** The goal of LSTM training is to find the optimal weights to minimize theloss function. During backpropagation, the gradients indicate the direction of the errors, and the learning rate decides how far the model moves in that direction.
  A high learning rate lets the model make large changes to its weights. While it may train faster initially, it risks overshooting the optimal minimum, leading to unstable training or oscillating predictions. A lower learning rate makes tiny adjustments. Training will be very stable, but the process will be slow and might get stuck in

suboptimal areas (saddle points). Most practitioners begin training LSTMs using the Adam optimizer with the default learning rate of 0.001.

- **Dropout:** Dropout in an LSTM is a regularization technique that randomly ignores a fraction of neurons during training. It prevents the model from relying on specific features and therefore reduces overfitting. A common value used is 0.2.

Let's put that knowledge into practice.

Finding the optimal parameters

The following code scripts show how to find the optimal parameters to get the minimum loss function output. Let's start the tuning process:

1. You will define the hyperparameter grid for the tuning process, including `hiddensize`, `num_layers`, `learning_rate` and `dropout` values:

```
hyperparameter_grid <- expand.grid(
  hidden_size = c(30,40,50,60,80),
  num_layers = c(1, 2),
  learning_rate = c(0.01, 0.001),
  dropout = c(0, 0.2)
)
```

2. Define variables for the tuning process. The loss value starts with `Inf` (infinitively large), and the best model and params will be the variable store the optimal one during tuning:

```
best_val_loss <- Inf
best_model <- NULL
best_params <- NULL
```

3. The tuning code will loop through the parameter combinations one by one. For each combination, it will create an LSTM model, use the same MSE loss function, and use the Adam optimizer (with learning rate from the grid).
   The training and validation code bases are similar to those in the previous code snippets.

   At the end of training, the code will evaluate whether the current `MSE` is less than the previously stored value. If so, it will replace it. Eventually, once the tuning loop completes, the optimal parameters will be stored, and the best `MSE` will be identified.

> **Note**
>
> Depending on your computer's computing power, this might take a long time (sometimes more than 1 hour) to complete.

4. Start by setting the hyperparameter grid size:

```
for(i in 1:nrow(hyperparameter_grid)) {
  params <- hyperparameter_grid[i, ]
  print(params)
```

5. Then initialize and train the model:

```
  model <- lstm_model(input_size, params$hidden_size, params$num_layers,
output_size, params$dropout)
```

6. You define the loss function and optimizer:

```
  loss_fn <- nnf_mse_loss  # Mean Squared Error loss for regression
  optimizer <- optim_adam(model$parameters, lr = params$learning_rate)  #
Adam optimizer for training
```

7. Next, set the number of epochs and the training loop:

```
  epochs <- 200

  for (epoch in 1:epochs) {
    model$train()  # Set the model to training mode

    optimizer$zero_grad()  # Reset gradients for each epoch
```

8. Then perform the forward pass to make predictions on the training data:

```
    predictions <- model(train$X)
```

9. Compute the loss between predictions and actual target values:

```
    loss <- loss_fn(predictions, train$Y)
```

10. The backward pass will compute the gradients:

```
    loss$backward()
```

11. Then you update the model parameters:

```
    optimizer$step()

    if (epoch %% 10 == 0) {
      cat("Epoch:", epoch, " Loss:", loss$item(), "\n")
    }
```

12. Set the model to evaluation mode, make predictions on the test set, and evaluate the outcome:

```
    model$eval()

    predictions <- model(test$X)
    predicted_values <- as.numeric(predictions)
    actual_values <- as.numeric(test$Y)

    mse <- mean((predicted_values - actual_values)^2)
    if (mse < best_val_loss) {
      best_val_loss <- mse
      best_model <- model
      best_params <- params
    }
  }
}
```

This gives you the following output:

```
hidden_size num_layers learning_rate dropout
1              30          1           0.01       0
Epoch: 10  Loss: 1.030354
Epoch: 20  Loss: 0.8101709
Epoch: 30  Loss: 0.7154472
Epoch: 40  Loss: 0.6369403
Epoch: 50  Loss: 0.5513749
Epoch: 60  Loss: 0.5086712
Epoch: 70  Loss: 0.4678951
```

```
Epoch: 80  Loss: 0.4122273
Epoch: 90  Loss: 0.3442233
Epoch: 100  Loss: 0.2655451
Epoch: 110  Loss: 0.2123822
Epoch: 120  Loss: 0.1747952
Epoch: 130  Loss: 0.1522827
Epoch: 140  Loss: 0.1096473
Epoch: 150  Loss: 0.08901455
Epoch: 160  Loss: 0.07384141
Epoch: 170  Loss: 0.06475461
Epoch: 180  Loss: 0.05859255
Epoch: 190  Loss: 0.0541386
Epoch: 200  Loss: 0.05088537
```

13. Print out the optimal parameters. The best loss (lowest) is 0.0227, with the `hidden_size` set to 50, `num_layer` set to 1, `learning_rate` set to 0.01, and `dropout` set to 0.2. Compared with the previous code without tuning, the current loss of 0.0227 is lower than 0.0245:

```
print(best_val_loss)
[1] 0.02274397

printC(best_params)
hidden_size num_layers  learning_rate  dropout
23            50          1            0.01     0.2
```

Tuning LSTM models is an important step in improving forecasting performance by identifying a model configuration that balances accuracy, complexity, and computational efficiency. By adjusting key hyperparameters such as the number of hidden units, layers, learning rate, and dropout, you can optimize the model's ability to learn meaningful patterns from time series data.

However, improving training performance alone does not guarantee that a model will generalize well to new data. A model that is too complex may learn noise from the training data, while a model that is too simple may fail to capture important patterns. In the next section, you will explore overfitting and underfitting, including how these issues affect LSTM forecasting performance and strategies to address them.

Overfitting and underfitting

Overfitting and **underfitting** are common issues during any machine learning process, not issues specific to LSTM.

The easiest way to determine whether you are overfitting or underfitting is to monitor your training and validation loss per epoch. In overfitting, training loss continues to drop while validation loss remains at a high level. This indicates that the model cannot handle new data well but handles training data well. You will see either flat highs or upward spikes. In underfitting, both training and validation loss typically remain high. Sometimes they are parallel; the model cannot handle both training and new data well. It needs more training or parameter tuning.

If your model performs perfectly in training but fails on test or validation data (new unseen data), you can try the following techniques:

- **Apply dropout:** Randomly ignore a fraction of neurons during the training process. Use the previous tuning process as a reference to find an optimal dropout value. A common one is 0.2 (drop out 20% randomly).
- **Implement early stopping:** Monitor your validation loss and stop the training process at the exact moment it starts increasing, rather than waiting for all epochs to finish.
- **Reduce model capacity:** Large networks are prone to memorizing data. Try reducing the number of hidden sizes (for example, from 80 to 40) or using the tuning process to find a good value.
- **Collect more data:** Expose the model to a wider variety of representative sequences. This is a very reliable way to prevent overfitting.

If your model performs poorly in prediction from the start or shows a high error rate across both training and validation data, you can try the following techniques:

- **Increase model complexity:** Add more hidden units, add an extra stacked LSTM layer, and so on.
- **Reduce dropout:** If you have heavy dropout applied, lower it or even set it to 0. Over-constraining a simple model prevents it from learning.
- **Add more epochs:** Your model may simply need more time to learn. If training loss is high, try increasing the maximum number of epochs to see whether it can help reduce the error.
- **Tune learning rate:** A high learning rate sometimes causes the model to skip over optimal patterns, while one that is too low will cause training to stall. Use an optimizer with an adaptive learning rate such as Adam. Use a tuning process to find a good learning rate.

By using these techniques, you ensure that your LSTM model is tested in a manner that closely mirrors how it would perform in real-world forecasting, where future data is predicted based

on past data. This method provides a more accurate assessment of the model's ability to generalize and handle sequential dependencies in time series data.

Summary

In this chapter, you learned about the intricacies of Long Short-Term Memory (LSTM) networks, a specialized type of Recurrent Neural Network (RNN) that excels in handling sequential and time-dependent data.

Starting with the foundational architecture of LSTM, you explored its key components, such as the cell state, input gate, forget gate, and output gate, which work together to overcome the limitations of traditional RNNs, particularly the vanishing gradient problem and short-term memory bias.

You also explored the practical applications of LSTM networks for time series forecasting, understanding how their unique structure makes them ideal for tasks that require capturing long-term dependencies and managing complex, nonlinear patterns. By identifying when and how to apply LSTM models, you have seen their effectiveness in handling real-world challenges, such as outliers and nonlinearity in time series data.

You then walked through a step-by-step guide to implementing and optimizing LSTM models in R. You learned how to build and train these models, configure essential parameters such as the number of epochs, hidden dimensions, and input chunk lengths, and employ techniques such as dropout regularization and early stopping to prevent overfitting. The rolling forecast cross-validation method was introduced to ensure your models are robust and generalize well to future data.

Whether you are dealing with seasonal patterns, long-term dependencies, or highly nonlinear data, LSTM models will be a powerful tool in your time series analysis toolkit, enabling you to make accurate and reliable predictions in a wide range of real-world scenarios.

Final words

Congratulations! You made it across the finish line! Across the thirteen chapters of this book, you followed a structured progression from foundational programming skills in R to advanced methods for time series forecasting and analysis. You began with essential R programming concepts, including syntax, data types, and data manipulation techniques, establishing the computational base required for later analytical tasks. Building on this foundation, you were introduced to time series data, its structure, and its unique characteristics, helping you to understand how temporal ordering, trend, and seasonality distinguish time series from other forms of data. Subsequent chapters further developed essential skills in handling dates and

times, as well as visualizing temporal data using both static and interactive tools, enabling clearer interpretation and communication of patterns.

The middle sections of the book focused on exploratory and diagnostic time series analysis. You were guided through techniques for identifying trends, smoothing fluctuations, decomposing time series into core components, and conducting autocorrelation analysis. These methods are complemented by anomaly detection techniques that help uncover unusual patterns or structural breaks in data. Together, these chapters built a strong analytical foundation for understanding the internal structure of time series and preparing data for robust forecasting models.

The later chapters transitioned into a wide range of forecasting methodologies, starting with classical statistical approaches such as exponential smoothing and extending to `ARIMA`, `SARIMA`, and `SARIMAX` models. More advanced techniques were then introduced, including `TBATS` for complex seasonal patterns, Bayesian forecasting for probabilistic inference, and hierarchical and grouped forecasting for multilevel data structures. The inclusion of practical case studies, such as customer support forecasting, reinforces the application of these methods in real-world contexts.

Finally, the book expanded into modern machine learning and deep learning approaches, introducing `Prophet` for flexible trend and seasonality modeling, as well as neural network-based methods, including basic neural networks, `RNNs`, and `LSTM` models. These approaches emphasize adaptability and predictive power in complex systems. Overall, the chapters collectively provide a comprehensive toolkit for time series analysis and forecasting in R, progressing from foundational programming to state-of-the-art predictive modeling techniques.

The main thing is to practice, practice, and practice. Handling time series data requires patience, persistence, and curiosity. Each use case is unique, and there is no single approach to solving the problem. Learn from your peers, community, and industry, and learn from your data! Keep on your journey and move forward!



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

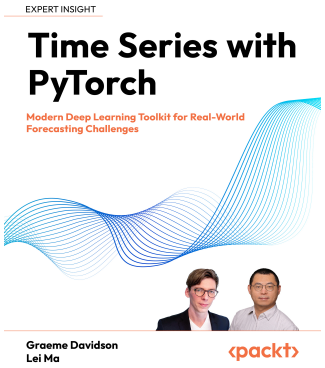
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

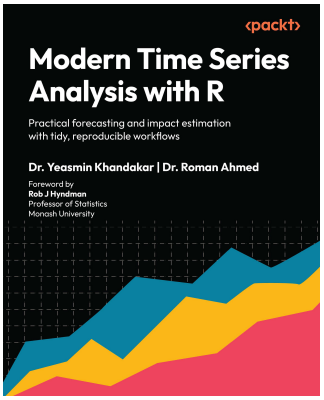


Time Series with PyTorch

Graeme Davidson, Lei Ma

ISBN: 978-1-80512-042-1

- Build, train, and evaluate neural networks for time series using PyTorch and PyTorch Lightning. Tune models with Bayesian optimisation and validate them with suitable metrics and strategies.
- Progress from feedforward and recurrent networks to transformers and models such as N-BEATS, N-HiTS, and TFT.
- Learn how global models use cross- and transfer learning across many series.
- Generate synthetic series and representations with diffusion and self-supervised methods.
- Apply modern approaches to classification, clustering, and anomaly detection.



Modern Time Series Analysis with R

Dr. Yeasmin Khandakar, Dr. Roman Ahmed

ISBN: 978-1-80512-430-6

- Understand the core concepts and structure of time series data
- Wrangle and visualize time series effectively
- Apply transformations and decomposition techniques
- Build and compare univariate forecasting models
- Apply statistical, ML, and DL models strategically based on context
- Forecast hierarchical and grouped time series
- Measure causal impact using interrupted time series analysis
- Detect anomalies, structural changes, and handle missing data

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Hands-on Time Series Analysis with R*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



`https://packt.link/r/180323878X`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

Adaptive Moment Estimation (Adam) **355**

Akaike Information Criterion (AIC) **197, 200, 240**
- cons 201
- pros 201

Augmented Dickey-Fuller (ADF) test **192 – 195**
- hypotheses 192

AutoRegressive Integrated Moving Average (ARIMA) **139, 187, 188, 270**

AutoRegressive Moving Average (ARMA)
- components 223
- errors 222, 223

Autoregressive (AR) component models **223**

Autoregressive Integrated Moving Average (ARIMA)
- components 188
- in R programming language 196 – 200

activation functions **333**
- rectified linear unit (ReLU) 334
- sigmoid 333
- tanh 334

advanced smoothing techniques **125**
- comparing 127

advanced techniques, Prophet **322**
- cross-validation 323, 324
- holiday effects 322
- parameter tuning 324 – 327

advanced time series plots
- creating 83, 84
- customizing, with ggplot2 package 89
- types 84 – 89

alternative model configurations
- evaluating 241 – 246

anomalies
- detecting 143, 144
- visual techniques 144, 145

arithmetic operations **11, 12**

as.Date() **52**

as.POSIXct() **52**

autocorrelation **27**
- autocorrelation function (ACF) 141, 142
- detecting 139
- lag plots 140, 141
- Partial Autocorrelation Function (PACF) 142, 143

autocorrelation analysis **138**
- impact, on statistical inference and model assumptions 139

autocorrelation function (ACF) **141, 142, 185**
- components 186

autoregressive (AR) **187 – 189**

autoregressive coefficients (AR) **272**

B

Bayes' theorem **251, 252**
- posterior, analyzing 256
- practical example, in R 253 – 256

Bayesian Information Criterion (BIC) **197, 200**
- cons 201
- pros 201

Bayesian Regression Models using Stan (brms) **250**

Bayesian hierarchical models
- practical example, in R 260 – 262

Bayesian inference **250**
- in time series context 257, 258
- likelihood 250, 251
- posterior distribution 250

posterior distributions 251
practical example, in R 258, 259
prior distribution 250
Bayesian models
implementing, in time series 270
Bayesian time series model
defining 262
example, in R programming language 275
model components, specifying 272
model specification and diagnostics 267
model specification and diagnostics, example 268 – 270
model validation and performance evaluation 275 – 277
practical example, in R 262 – 264
practical example, in R programming language 273, 274
priors, choosing 264
priors, choosing in R 265 – 267
Bayesian time series model analysis
Bayes' theorem 251 – 253
foundations 250
Box-Cox transformation **222**
bottom-up approach **291**
advantages 291
challenges 291
implementing, in R programming language 291, 292

C

Comprehensive R Archive Network (CRAN) **4**
cell state **349**
character vector **10**
complex neural networks **335**
continuous time series **28**
convergence diagnostics **268**
convolutional neural networks (CNNs) **335**
corrected AIC (AICc) **197**
cross-sectional dependencies
handling 305
impact, on forecasting accuracy 306
modeling interactions 305
techniques, between groups
cross-validation **208**
in R programming language 210 – 212
rolling forecast 209
sliding forecast 209
customer support case study **152 – 159**
model, evaluating 161, 162
test datasets 159 – 161
training datasets 159 – 161
cyclical changes, time series data **25, 26**

D

data aggregating **289, 290**
data analysis function **13**
data cleaning and normalization **66 – 69**
techniques 67
data disaggregating **289, 290**
data drift **30**
data frame vector **10**
data manipulation
arithmetic operations 11, 12
basic operations and functions, mastering 11
data analysis function 13
data preparation **336, 337**
data ranges
slicing 39
data types
exploring, in R programming language 8
date class **45**
date objects, creating 45, 46
date objects, manipulating 45, 46
functions, with date objects 45
date-time classes **44**
format() function, using 50
format, customizing to include or exclude components 51
formatting 50

in R programming language 44
date-time data, best practices **55**
enterprise standards 56
format and documentation 55
issues and error checks 57, 58
right class, using 55
date-time strings
advanced formatting 53
as.Date(), using 51, 52
as.POSIXct(), using 51, 52
parsing issues, handling 53
parsing techniques 53
parsing, into R objects 51
parsing, with strftime() 54
decompose() function
using 133
default TBATS model
alternative model 241 – 246
configurations, evaluating components 239, 240
parameters 240
used, for forecasting 241
deviance information criterion (DIC) **268**
differencing **190**
discrete time series **29**
double exponential smoothing **163 – 165**
using 166 – 168
dynamic data **30**
dynamic linear models (DLM) **271**
dynamic plots
building, with shiny application 99

E

Exponential Moving Average (EMA) **122, 123**
effective sample sizes (ESS) **275**
evolving data **30**
exogenous variables (X) **205**
expected log predictive density (elpd) **280**

F

feasts package **152**
feature engineering **69 – 71**
forecast errors **195**
forecast package **152**
forecasting techniques, grouped time series analysis
univariate time series models 299
forget gate **349**
format() function
using 50
frequency **20**
frequency conversion **40**

G

ggplot2 package **81**
advanced time series plots, creating 83, 84
advanced time series plots, customizing 89
advantages 82
annotations, adding 94
axes scales, transforming 92, 93
color, shape, and line types, using 95
combining, with interactive visualization 96, 97
date labels text and format, customizing 93
installing 82
predefined theme, applying 90, 91
setup 82
titles and labels, customizing 91, 92
using, for time series visualization 81
grouped time series analysis **296**
comparing, with hierarchical forecasting 297
cross-sectional dependencies, handling 305
data preprocessing steps 298
example 300 – 304
forecasting techniques 299

grouping criteria 297
methods 297
techniques 298

grouping criteria 297

H

Holt's method 166

Holt-Winters method 168

heteroscedasticity 222

hidden layer 332

hierarchical and grouped forecasting models 299

hierarchical forecasting 288, 289
benefits and limitations 289
concept and methodology 289
methods 288
reconciliation and consistency 295
top-down approach 289

I

Integrated Development Environment (IDE) 3

Interquartile Range (IQR) methods 145

informative priors 251, 264

input gate 349

input layer 332

integrated (I) 189, 190
Augmented Dickey-Fuller (ADF) test 192 – 195
non-stationarity visualization 190, 191
stationarity visualization 190, 191

interactive time series plot
creating, with plotly 108, 109

interactive visualization
dynamic plots, building with shiny application 99
tools 98
with shiny and plotly 97

interquartile range (IQR)
using 146, 147

K

Kalman filter 31

L

LSTM components
cell state 349
forget gate 349
input gate 349
output gate 349

LSTM models
optimal parameters, finding 360 – 363
overfitting 363, 364
tuning, for optimal performance 359, 360
underfitting 364

Locally Estimated Scatterplot Smoothing (LOESS) 125
applying, in R programming language 125 – 127

Long Short-Term Memory (LSTM) 349
implementing, in R programming language 352 – 359
RNN limitations, overcoming 350
time series data, challenges 351
time series forecasting 351

lag plots 140, 141

lambda (λ) 222

leave-one-out (LOO) cross-validation method 280

likelihood 251

list vector 11

long short-term memory networks (LSTMs) 335

lubridate package 152

M

Mann-Kendall test
outputs 119
significance 119
used, for trend detection 118

Mean Absolute Error (MAE) 172, 198, 234

Mean Absolute Percentage Error (MAPE) 172, 198

Mean Absolute Scaled Error (MASE) **172**
Mean Error (ME) **172**
Mean Percentage Error (MPE) **172**
Moving Average (MA) component models **223**
machine learning models **300**
matrix vector **10**
middle-out approach **293**
implementing, in R programming language 293, 294
missing data
dealing with 29
mixed-effects models **300**
model preparation **337 – 339**
model selection, criteria
seasonality 272
stationarity 271
trend 272
model validation and performance evaluation
residual plots 277, 278
sensitivity analysis 278 – 280
moving average (MA) **31, 120, 195**
Exponential Moving Average (EMA) 122, 123
Simple Moving Average (SMA) 121, 122
Weighted Moving Average (WMA) 124, 125
moving average coefficients (MA) **272**
multi-series forecasting **284**
challenges 287
use cases 285, 286
multi-series forecasting, challenges
ata management 287
complexities
computational demands 288
data quality 287
multilevel models **300**
multiple seasonal patterns
exploring 235 – 237
multiple seasonal time series
TBATS model, applying to 234
multivariate time series models **28, 299**

N

neural networks (NN) **330 – 332**
activation functions 333
noise **26, 27, 134**
managing 31
non-informative priors **251, 264**
non-stationarity **189**
non-stationarity visualization **190**
numeric vector **9**

O

observer function **106**
outliers **143**
output gate **349**
output layer **332**
overfitting **363**
overplotting **109**
examples 110
techniques 110

P

POSIXct classes **46**
for arithmetic operations 47
for arithmetic operations and datasets 46
POSIXlt classes **46**
date-time access 47, 48
Pareto Principle **159**
Partial Autocorrelation Function (PACF) **142, 143, 186**
Prophet **309, 310**
advanced techniques 322
implementing, in R programming language 312 – 317
key components 311
model, customizing 317
model, tuning 317
regressors, adding 318 – 322
seasonality modes 317
use cases 311
packages **1**

panel data models **300**
parameter tuning **212**
in R programming language 213 – 217
plotly
interactive time series plot, creating 108, 109
syntax and functions 106
time series plot, creating 107, 108
posterior distribution **251, 252**
posterior predictive check **268, 269, 277**
prior **250**
prior distribution **250, 251**

Q

quantmod package **18**

R

R programming language **2**
Autoregressive Integrated Moving Average (ARIMA) 196 – 200
code examples 9
cons 3
cross-validation 210 – 212
data structures 9
data types, exploring 8
date-time classes 44
functions 8
installing 4, 5
LOESS, applying 125 – 127
LSTM, implementing 352 – 356, 359
operations 8
overview 74
parameter tuning 213 – 217
plots, enhancing 77 – 81
Prophet implementation 312 – 317
pros 2
Seasonal Autoregressive Integrated Moving Average (SARIMA) 202 – 204
Seasonal Autoregressive Integrated Moving Average with eXogenous (SARIMAX) 205 – 208
simple neural networks, building for time series 336
smoothed data, analyzing 129 – 131
success tips 7
syntax 7
syntax, basics 7
time series data, plotting 75 – 77
time series object, creating 31 – 33
workspace, setting up 6
RStudio **3, 4**
installing 4 – 6
Rhat **254**
Root Mean Squared Error (RMSE) **172, 198, 234**
reactive expressions **103, 104**
rectified linear unit (ReLU) **334**
recurrent neural networks (RNNs) **335, 346**
structure 346, 347
traditional RNNs 347
regressors **318**
resampling **40 – 42**
residuals **196**
rolling calculations **61**
rolling mean 61
rolling median 61
rolling sum 61
rolling forecast **209**

S

Seasonal Autoregressive Integrated Moving Average (SARIMA) **187**
exploring 201, 202
in R programming language 202 – 204
Seasonal Autoregressive Integrated Moving Average with eXogenous (SARIMAX) **187, 204**
in R programming language 205 – 208
Seasonal and Trend decomposition using LOESS (STL) **133**
Simple Moving Average (SMA) **121, 122**
Stan **273**
sales forecasts
case study 178 – 187
seasonal components model **224**

seasonal decomposition **132**
decompose() function, using 133 – 135
implementing, in R programming language 133
importance 132
residuals 132
seasonal component 135 – 138
significance, evaluating seasonality 132
stl() functions, using 133 – 135
trend 132
seasonality **24, 25**
sensitivity analysis **278**
serial correlation **138**
shiny application
advanced features 103
architecture 99
creating 101 – 103
dynamic plots, building 99
interactivity, enhancing with plotly 106
observer function 106
plotly syntax and functions, overview 106
reactive expressions 103, 104
server logic 100, 101
setting up 99
UI 99, 100
short-term memory issue **348, 349**
sigmoid **333**
simple exponential smoothing (SES) **162**
applying, to data 163, 164
simple neural network models
evaluating 339 – 343
optimizing 339 – 343
simple neural networks **335**
building, for time series in R programming language 336
data preparation 336, 337
model preparation 337 – 339
slicing **39**
sliding forecast **209**
smoothed data
analyzing 128
analyzing, in R programming language 129 – 131
smoothed time series **129**
analyzing, techniques 128
smoothing techniques **120**
advanced smoothing techniques 125
moving average (MA) 120
specialized models **300**
multilevel models 300
panel data models 300
stacked charts **88**
standard errors (SE) **197**
state-space models **271**
stationarity **189**
stationarity visualization **190**
statistical techniques **145**
automated methods 147, 148
interquartile range (IQR) 146, 147
Z-scores, using 145
statistical tests
Mann-Kendall test 118
step plots **87**
stl() function
using 134, 135
strftime()
date-time strings, parsing 54
using, to create user-friendly date-time strings 54

T

TBATS models
applying, to multiple seasonal time series 234
ARMA errors 222, 223
Box-Cox transformation 222
dataset, fitting 237 – 239
multiple seasonal patterns, exploring 235 – 237
overview 220, 221
seasonal components model 224
Taylor electricity demand dataset 234, 235
trend components model 223
trigonometric seasonality 221

TBATS models, in R programming language
AirPassengers dataset 225
forecasting 225 – 234
implementing 224
TTR package **151**
Twitter anomaly detection libraries
using 147, 148
tanh **334**
taylor dataset **234, 235**
temporal data operations
performing 58
time series arithmetic 59
time shifts date adjustments 63 – 66
Window functions and rolling calculations 61 – 63
time indexing **38, 39**
time series arithmetic **59**
stock price change, calculating 59
time series data, adjusting for inflation or external factors 60
time series data
autocorrelation 27
challenges 351
components 20 – 23
continuous time series 28
cyclical changes 25, 26
data cleaning and normalization 66 – 69
defining 18, 19
dscrete time series 29
feature engineering 69 – 71
manipulating 66
model selection 270, 271
model selection, criteria 271, 272
multivariate time series models 28
noise 26, 27
seasonality 24, 25
trend 23, 24
univariate time series 27
time series data analysis
challenges 29 – 31
time series object
creating, in R programming language 31 – 33
time series plot
creating, with plotly 107, 108
time series visualization
errors, identifying 109
pitfalls 109
time series visualization, errors
anomalies 110
overplotting 109
scales issues 110
seasonal patterns 110
time zones
handling 48, 49
traditional RNNs **347**
short-term memory issue 348, 349
vanishing gradient problem 347, 348
trend **23, 24**
detecting 115
identification and analysis 114
identification, in time series data 114
inspecting, in graphs 116, 117
trend components model **223**
trend detection
statistical tests 118
visual inspection 115
trigonometric seasonality **221**
triple exponential smoothing **163, 168**
Alpha 169
Beta 169
daily data, handling 172 – 174
future period, predicting 170, 171
Gamma 169
model assessment 171, 172
practical applications 169, 170
tsibble package **152**

U

underfitting **363**
unit root **192**
univariate time series **27**
univariate time series models **299**
hierarchical and grouped forecasting models 299
machine learning models 300

multivariate time series models 299
specialized models 300

V

vanishing gradient problem **348**
variance components **272**
vector autoregression (VAR) **299**
visual inspection
line graphs, using to visually assess trends 115, 116

W

Watanabe-Akaike information criterion (WAIC) **268**
Weighted Moving Average (WMA) **124**
Window functions **61**
wavelet transforms **31**
weakly informative priors **265**
widely applicable information criterion **268**

X

xts package **18, 35, 36**
data, plotting 36, 37

Z

Z-scores
using 145
zoo package **18**
data, plotting 36
data, transforming 34, 35