Advanced Forecasting with Python

Mastering Modern Forecasting Techniques with Machine Learning and Cloud Tools

—

Second Edition

—

Joos Korstanje

Apress®

Joos Korstanje

Advanced Forecasting with Python

Mastering Modern Forecasting Techniques with Machine Learning and Cloud Tools

Second Edition

Apress®

Joos Korstanje
Maisons Alfort, France

ISBN 979-8-8688-2027-4 e-ISBN 979-8-8688-2028-1
https://doi.org/10.1007/979-8-8688-2028-1

The first edition offers a practical guide to forecasting, blending classical and modern machine learning methods. It combines clear explanations, mathematical foundations, and Python examples to help readers choose and implement

© Joos Korstanje 2021, 2025

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Apress imprint is published by the registered company APress Media, LLC, part of Springer Nature.
The registered company address is: 1 New York Plaza, New York, NY 10004, U.S.A.

Introduction

Advanced Forecasting with Python covers all machine learning techniques relevant for forecasting problems, ranging from univariate and multivariate time series to supervised learning, state-of-the-art deep forecasting models like LSTMs, recurrent neural networks, and numerous cloud-based forecasting setups.

Rather than focus on a specific set of models, this book presents an exhaustive overview of all techniques relevant to practitioners of forecasting. It begins by explaining the different categories of models that are relevant for forecasting in a high-level language. Next, it covers univariate and multivariate time series models, followed by advanced machine learning and deep learning models such as recurrent neural networks, LSTMs, and cloud platforms. It concludes with reflections on model selection, like benchmark scores vs. understandability of models vs. compute time, and automated retraining and updating of models. Each of the models presented in this book is covered in-depth, with an intuitive, simple explanation of the model, a mathematical transcription of this idea, and Python code that applies the model to an example dataset.

This book is a great resource for those who want to add a competitive edge to their current forecasting skillset. The book is also adapted to those who wish to start working on forecasting tasks and are looking for an exhaustive book that allows them to start with traditional models and gradually move into more and more advanced models.

You can follow along with the code using the GitHub repository that contains a Jupyter Notebook per chapter. You are encouraged to use Jupyter notebooks for following along, but you can also run the code in any other Python environment of your choice. An environment.yml is provided for each notebook, which can be imported through Anaconda or other environment tools.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

Table of Contents

Part I: Machine Learning for Forecasting

Chapter 1: Models for Forecasting

Reading Guide for This Book

Machine Learning Landscape

Univariate Time Series Models

Supervised Machine Learning Models

Other Distinctions in Machine Learning Models

Supervised vs. Unsupervised Models

Classification vs. Regression Models

Univariate vs. Multivariate Models

Key Takeaways

Chapter 2: Model Evaluation for Forecasting

Evaluation with an Example Forecast

Error Metrics

Error Metric 1: MSE

Error Metric 2: RMSE

Error Metric 3: R2

Error Metric 4: MAE

Error Metric 5: MAPE

Model Evaluation Strategies

Overfit and the Out of Sample Error

Strategy 1: Train-Test Split

Strategy 2: Train-Validation-Test Split

Strategy 3: Cross-Validation for Forecasting

Backtesting

Which Strategy to Use for Safe Forecasts?
Final Considerations on Model Evaluation
Key Takeaways
Chapter 3: Model Management and Benchmarking Using MLflow
Introduction to MLflow
Local vs. Hosted MLflow
Setting Up MLflow Locally
Experiment Tracking Using MLflow
MLflow Data
Inspecting the Model Logs Through Python
The MLflow UI
MLflow Throughout This Book
Key Takeaways
Part II: Univariate Time Series Models
Chapter 4: The AR Model
Autocorrelation: The Past Influences the Present
Compute Autocorrelation in Earthquake Counts
Positive and Negative Autocorrelation
Stationarity and the ADF Test
Differencing a Time Series
Lags in Autocorrelation
Partial Autocorrelation
How Many Lags to Include?
AR Model Definition
Estimating the AR Using Yule–Walker Equations
The Yule–Walker Method

Train, Test, Evaluation, and Tuning
Saving the Final Model Using MLflow
Key Takeaways
Chapter 5: The MA Model
The Model Definition
Fitting the MA Model
Stationarity
Choosing Between an AR and an MA Model
Application of the MA Model
Multistep Forecasting with Model Retraining
Grid Search to Find the Best MA Order
Saving This Model in MLflow
Key Takeaways
Chapter 6: The ARMA Model
The Idea Behind the ARMA Model
The Mathematical Definition of the ARMA Model
An Example: Predicting Births Using ARMA
Fitting an ARMA(1,1) Model
More Model Evaluation KPIs
Automated Hyperparameter Tuning
Grid Search: Tuning for Predictive Performance
Saving This Model in MLflow
Key Takeaways
Chapter 7: The ARIMA Model
ARIMA Model Definition
Model Definition

ARIMA on the CO_2 Example
Key Takeaways
Chapter 8: The SARIMA Model
Univariate Time Series Model Breakdown
The SARIMA Model Definition
Example: SARIMA on Walmart Sales
Key Takeaways
Part III: Multivariate Time Series Models
Chapter 9: The SARIMAX Model
Time Series Building Blocks
Model Definition
Supervised Models vs. SARIMAX
Example of SARIMAX on the Walmart Dataset
Key Takeaways
Chapter 10: The VAR Model
The Model Definition
Order: Only One Hyperparameter
Stationarity
Estimation of the VAR Coefficients
One Multivariate Model vs. Multiple Univariate Models
An Example: VAR for Forecasting Walmart Sales
Key Takeaways
Chapter 11: The VARMAX Model
Model Definition
Multiple Time Series with Exogenous Variables
Key Takeaways

Part IV: Supervised Models

Chapter 12: The Linear Regression

Linear Regression

Model Definition

Example: Linear Model to Forecast CO_2 Levels

Key Takeaways

Chapter 13: The Decision Tree Model

Mathematics

Splitting

Pruning and Reducing Complexity

Example

Key Takeaways

Chapter 14: The kNN Model

Intuitive Explanation

Mathematical Definition of Nearest Neighbors

Combining k Neighbors into One Forecast

Deciding on the Number of Neighbors k

Predicting Traffic Using kNN

Grid Search on kNN

Random Search: An Alternative to Grid Search

Key Takeaways

Chapter 15: The Random Forest

Intuitive Idea Behind Random Forests

Random Forest Concept 1: Ensemble Learning

Bagging Concept 1: Bootstrap

Bagging Concept 2: Aggregation

Random Forest Concept 2: Variable Subsets
Predicting Births Using a Random Forest
Grid Search on the Two Main Hyperparameters of the Random Forest
Random Search CV Using Distributions
Distribution for max_features
Distribution for n_estimators
Fitting the RandomizedSearchCV
Interpretation of Random Forests: Feature Importance
Key Takeaways
Chapter 16: Gradient Boosting with XGBoost, LightGBM, and CatBoost
Boosting: A Different Way of Ensemble Learning
Gradient Boosting
The Difference Between XGBoost and LightGBM
Adding CatBoost to the Mix
Forecasting Traffic Volume with XGBoost
Forecasting Traffic Volume with LightGBM
Forecasting Traffic Volume with CatBoost
Inspecting the Differences in Predictions
Hyperparameter Tuning Using Bayesian Optimization
The Theory of Bayesian Optimization
Bayesian Optimization Using scikit-optimize
Conclusion
Key Takeaways
Chapter 17: Bayesian Models with pyBATS
Intuitive Idea Behind Bayesian Models

Bayesian vs. Frequentist Statistics
pyBATS As an Implementation of Bayesian Forecasting
Forecasting Sales Using pyBATS
Sales Forecast: Exploratory Data Analysis
Sales Forecast: Univariate Poisson DGLM
Sales Forecast: Normal DGLM with X Variable
Key Takeaways

Part V: Neural Networks

Chapter 18: Neural Networks
Fully Connected Neural Networks
Activation Functions
The Weights: Backpropagation
Optimizers
Learning Rate of the Optimizer
Hyperparameters at Play in Developing a NN
Introducing the Example Data
Specific Data Prep Needs for NN
Scaling and Standardization
Principal Component Analysis (PCA)
The Neural Network Using Keras
Conclusion
Key Takeaways
Chapter 19: RNNs Using SimpleRNN and GRU
What Are RNNs: Architecture
Inside the SimpleRNN Unit
The Example

Predicting a Sequence Rather Than a Value
Univariate Model Rather Than Multivariable
Preparing the Data
A Simple SimpleRNN
SimpleRNN with Hidden Layers
Simple GRU
GRU with Hidden Layers
Key Takeaways
Chapter 20: LSTM RNNs
What Is LSTM
The LSTM Cell
Example
LSTM with 1 Layer of 8
LSTM with 3 Layers of 64
Conclusion
Key Takeaways
Part VI: Black Box and Cloud-Based Models
Chapter 21: The N-BEATS Model with Darts
Intuition and Mathematics of N-BEATS
Interpretability
Stacking
Automatic Feature Engineering and Decomposition
The Darts Package and Its Implementation of N-BEATS
Forecasting Sales Using N-BEATS in Darts
Sales Forecast: Preparing the Data
Sales Forecast: Create Default N-BEATS Model

Sales Forecast: Create Multivariate N-BEATS Model
Sales Forecast: Tuning the Multivariate N-BEATS Model
Benchmark Results
Key Takeaways
Chapter 22: The Transformer Model with Darts
Intuition and Mathematics of Transformers
Attention Is All You Need
Transformers Move Away from Recurrent Units
The Darts Package and Its Implementation of Transformers
Forecasting Sales Using Transformer in Darts
Sales Forecast: Preparing the Data
Sales Forecast: Create a Default N-BEATS Model
Sales Forecast: Create the Multivariate Transformer Model
Sales Forecast: Tuning the Multivariate Transformer Model
Benchmark Results
Key Takeaways
Chapter 23: The NeuralProphet Model
The NeuralProphet Model
Predicting Heat Waves with NeuralProphet
Wikipedia Pageview Tool
Preparing the Data in Python
Time Series Decomposition Using NeuralProphet
Train-Test Split
Default Model
Tuned Model
MLflow

Key Takeaways

Chapter 24: The DeepAR Model and AWS SageMaker AI

About DeepAR

Forecasting Heat Wave Page Views Using gluonts DeepAR Locally

Tuning the DeepAR Model Using Ray HyperOptSearch

DeepAR vs. AWS SageMaker AI

Forecasting Heat Wave Page Views Using DeepAR Inside an AWS SageMaker AI Training Job

Step 1: Prepare the Data File

Step 2: Upload the Data to S3

Step 3: SageMaker AI

Key Takeaways

Chapter 25: Uber's Orbit Model

About Orbit

Forecasting Heat Wave Page Views Using Orbit's Local-Global Trend Model

Tuning Orbit's GLT

Forecasting Heat Wave Page Views Using Orbit's Damped Local Trend Model

Tuning Orbit's DLT

Key Takeaways

Chapter 26: AutoML with Microsoft Azure

Cloud Computing

From Cloud-Based Maths to AutoML

Microsoft Azure Cloud

Forecasting Heat Wave Page Views with Microsoft Azure Cloud AutoML

Step 1: Create an Azure Machine Learning Studio
Step 2: Create an Automated ML Run
Optional: Download MLflow Artifact of the Model
Key Takeaways
Chapter 27: AutoML with Vertex AI on Google Cloud Platform
GCP and BigQuery
Vertex AI
Forecasting Heat Wave Page Views with Google Cloud Platform's Vertex AI AutoML
Step 1: Data Preparation for GCP
Step 2: Create a Bucket
Step 3: Uploading the Data
Optional Next Steps
Key Takeaways
Chapter 28: Nixtla Suite and TimeGPT
The NixtlaVerse
Simple Use Case with the NixtlaVerse
The GPT in TimeGPT
Nixtla's TimeGPT API
Key Takeaways
Chapter 29: Model Selection
Model Selection Based on Metrics
Model Structure and Inputs
One-Step Forecasts vs. Multistep Forecasts
Model Complexity vs. Gain
Model Complexity vs. Interpretability

Model Stability and Variation

Open Source vs. Cloud Services Black Box

Conclusion

Key Takeaways

Index

About the Author

Joos Korstanje
is a data scientist and author specializing in machine learning and geospatial analysis. He has written *Advanced Forecasting with Python* and *Machine Learning on Geographical Data Using Python* (Apress), books praised for their practical approach to forecasting, real-time data processing, and spatial modeling. With extensive experience at major financial institutions in Paris, he combines academic expertise with hands-on industry knowledge to solve complex analytical challenges. His work bridges the gap between theory and application, making advanced techniques accessible to practitioners. Passionate regarding empowering people, he regularly provides insights through writing, speaking, and mentoring in the data science community.

About the Technical Reviewer

Olivia Petris

is a data engineer at a defense company based in France. Throughout her professional journey, she has consistently sought out challenging, engaging, and meaningful assignments that push her technical and analytical abilities. After earning her Engineering Diploma in Computer Science, she chose to specialize in data science and big data—fields that align with her passion for innovation, problem-solving, and working with cutting-edge technologies. She continuously hones her skills and stays current with the latest advancements in IT, data engineering, and the broader tech landscape. Outside of work, she enjoys traveling to new places, practicing karate, and spending quality time with her family and close friends.

Part I
Machine Learning for Forecasting

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_1

1. Models for Forecasting

Joos Korstanje[1]
(1) Maisons Alfort, France

Forecasting, grossly translated as the task of predicting the future, has been present in human society for ages. Whether it is through fortune tellers, weather forecasts, or algorithmic stock trading, man has always been interested in predicting what the future holds.

Yet, forecasting the future is not easy. Consider fortune tellers, stock market gurus, or weather forecasters: many try to predict the future, but few succeed. And for those who succeed, you will never know whether it was luck or skill.

In recent years, the computing power of computers has become much more commonly available than, say, 30 years ago. This has created a great boom in the use of Artificial Intelligence. Artificial Intelligence and especially Machine Learning can be used for a wide range of tasks, including robotics, self-driving cars, but also forecasting, that is, if you have a reasonable amount of data about the past that you can project into the future.

Throughout this book, you will learn the modern Machine Learning techniques that are relevant for forecasting. I will present a large number of Machine Learning models, together with an intuitive explanation of the model, its mathematics, and an applied use case.

The goal of this book is to give you a real insight into the application of those Machine Learning models. You will see worked examples applied to real datasets together with honest evaluations of the results: some successful, some less successful.

In this way, this book is different than many other resources, which often present perfectly fitting use cases on simulated data. To learn real-life

Machine Learning and forecasting, it is important to know how models work, but it is even more important to know how to evaluate a model honestly and objectively. This pragmatic point of view will be the guideline throughout the chapters.

Reading Guide for This Book

Before going further into the different models throughout this book, I will first present a general overview of the Machine Learning landscape: many types and families of models exist. Each of them has its applications. Before starting, it is important to have an overview of the types of models that exist in Machine Learning and which of them are relevant for forecasting.

After this, I will cover several strategies and metrics for evaluating forecasting models. It is important to understand objective evaluation before practicing: you need to understand your goal before starting to practice.

The remaining chapters of the book will each cover a specific model with an intuitive explanation of the model, its mathematical definitions, and an application on a real dataset. You will start with common but simple methods and work your way up to the most recent and state-of-the-art methods on the market.

Machine Learning Landscape

Having the bigger picture of Machine Learning models before getting into detail will help you to understand how the different models compare to each other and will help you to keep the big picture throughout the book. You will first see univariate time series and supervised regression models: the main categories of forecasting models. After that, you will see a shorter description of Machine Learning techniques that are less relevant for forecasting.

Univariate Time Series Models

The first category of Machine Learning models that I want to talk about is time series models. Even though univariate time series have been around for a long time, they are still used. They also form an important basis for

several state-of-the-art techniques. They are classical techniques that any forecaster should be familiar with.

Time series models are models that make a forecast of a variable by looking only at historical developments of the variable itself. This means that time series, as opposed to other model families, do not try to describe any “logical” relationships between variables. They do not try to explain the “why” of trends or seasonalities, but they simply put a mathematical formula on the past and try to project it to the future.

Time series modeling is sometimes criticized for this “lack of science.” But time series models have gained an important place in forecasting due to their performance, and they could not be ignored.

A Quick Example of the Time Series Approach

Let’s look at a super simple, purely hypothetical, example of forecasting the average price of the cup of coffee in an imaginary city called X. Imagine someone has made the effort of collecting the average price of coffee for 90 years in this town, with intervals of five years, and that this has yielded the data in Table 1-1.

Table 1-1 A hypothetical example: the price of a cup of coffee over the years

Year	Average Price
1960	0,80
1965	1,00
1970	1,20
1975	1,40
1980	1,60
1985	1,80
1990	2,00
1995	2,20
2000	2,40
2005	2,60
2010	2,80
2015	3,00
2020	3,20

This fictitious data clearly shows an increase of 20ct in the price every five years. This is a **linearly increasing trend**: linear because it increases by the same amount every year. Increasing because it becomes more rather than less.

Let's get this data into Python to see how to plot this linear increasing trend using Listing 1-1. The source code for this book is available on GitHub via the book's product page, located at https://github.com/Apress/Advanced-Forecasting-with-Python-2nd-ed. Please note that the library imports are done once per chapter.

```
import pandas as pd

years = [1965, 1970, 1975, 1980, 1985, 1990, 1995,
2000, 2005, 2010, 2015, 2020]
prices = [1.00, 1.20, 1.40, 1.60, 1.80, 2.00,
2.20, 2.40, 2.60, 2.80, 3.00, 3.20]

data = pd.DataFrame({
    'year' : years,
    'prices': prices
})
ax = data.plot.line(x='year')
ax.set_title('Coffee Price Over Time',
fontsize=16)
```

Listing 1-1 Getting the coffee example into Python and plotting the trend

You will obtain the graph displayed in Figure 1-1.

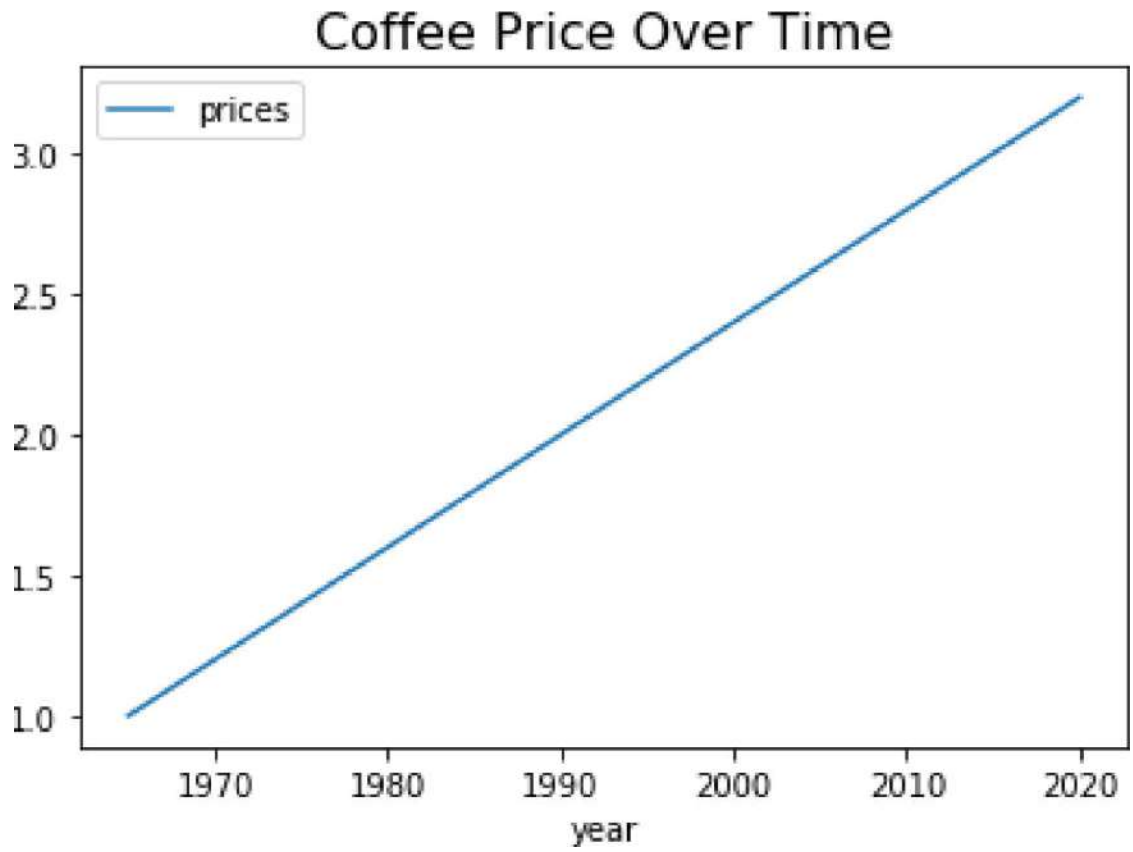


Figure 1-1 The plot of the coffee prices example

To make predictions for the price of coffee in this hypothetical town, you could just put your ruler next to the graph and continue the upward line: the prediction for this variable does not need any **explanatory variables** other than its past values. The historical data of this example allows you to forecast the future. This is a determining characteristic of **time series models**

Now, let's see a comparable example but with the prices of hot chocolate rather than the prices of a cup of coffee and quarterly data rather than 5-yearly data (Table 1-2).

Table 1-2 Hot chocolate prices over the years

Period	Average Price
Spring 2018	2,80
Summer 2018	2,60
Autumn 2018	3,00
Winter 2018	3,20

Period	Average Price
Spring 2019	2,80
Summer 2019	2,60
Autumn 2019	3,00
Winter 2019	3,20
Spring 2020	2,80
Summer 2020	2,60
Autumn 2020	3,00
Winter 2020	3,20

Do you see the trend? In the case of hot chocolate, you do not have a year-over-year increase in price, but you do detect **seasonality:** in the example, hot chocolate prices follow the temperatures of the seasons. Let's get this data into Python to see how to plot this seasonal trend (use Listing 1-2 to obtain the graph in Figure 1-2).

```
seasons = ["Spring 2018", "Summer 2018", "Autumn
2018", "Winter 2018",
           "Spring 2019", "Summer 2019", "Autumn
2019", "Winter 2019",
           "Spring 2020", "Summer 2020", "Autumn
2020", "Winter 2020"]
prices = [2.80, 2.60, 3.00, 3.20,
          2.80, 2.60, 3.00, 3.20,
          2.80, 2.60, 3.00, 3.20]

data = pd.DataFrame({
    'season': seasons,
    'price': prices
})

ax = data.plot.line(x='season')
ax.set_title('Hot Chocolate Price Over Time',
fontsize=16)
ax.set_xticklabels(ax.get_xticklabels(),
rotation=45, ha='right')
```

Listing 1-2 Getting the hot chocolate example into Python and plotting the trend

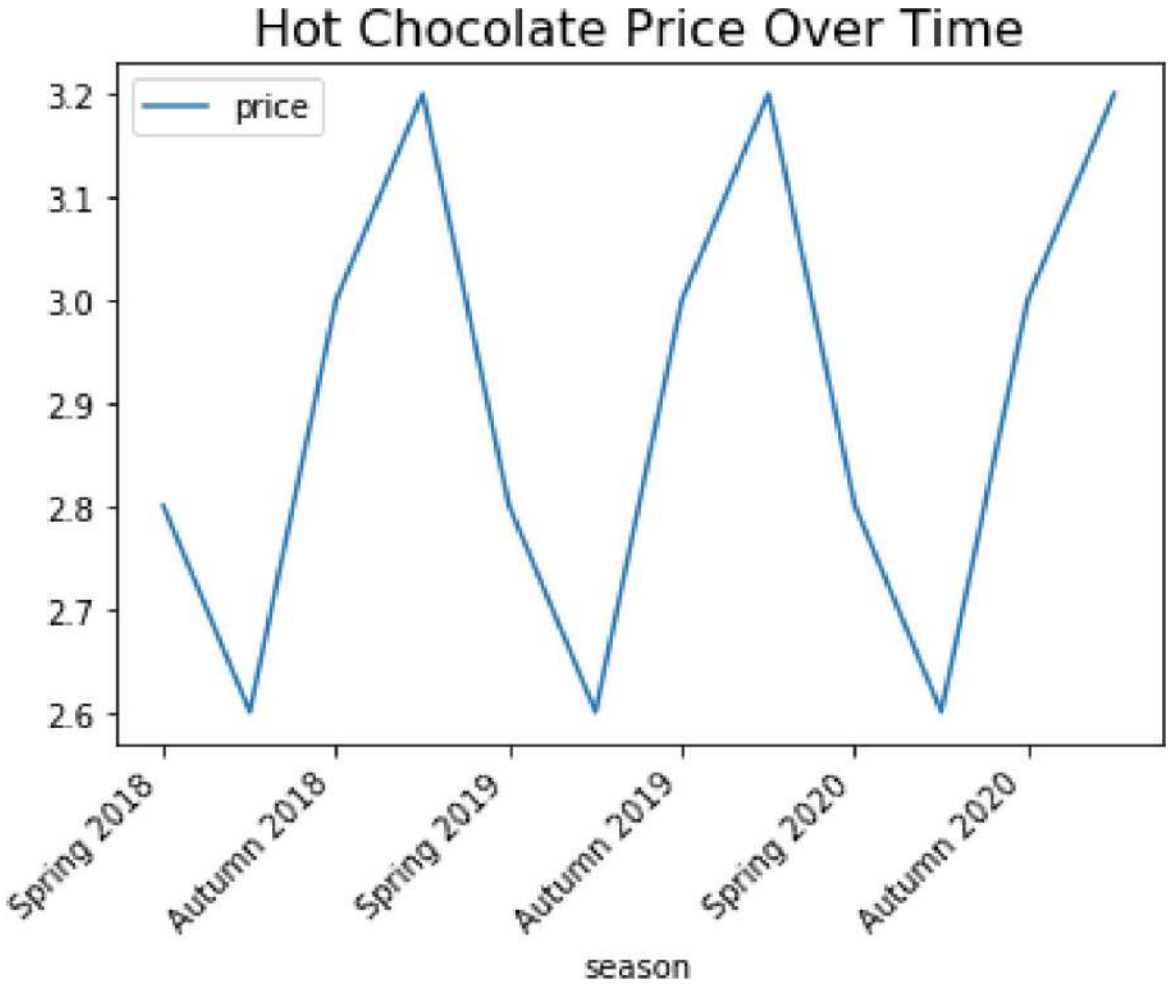


Figure 1-2 Plot of the hot chocolate prices

As in the previous example, you can predict the future prices of hot chocolate easily using the past data on hot chocolate prices: the prices depend only on the season and are not influenced by any explanatory variables.

> **Note** Univariate time series models make predictions based on trends and seasonality observed in their own past and do not use explanatory variables other than the **target variable: the variable that you want to forecast**.

You can imagine numerous types of combinations of those two processes, for example, having both a quarterly seasonality and a linear increasing

trend, etc. There are many types of processes that can be forecasted by modeling the historical values of the target variable. In Chapters 3 to 7, you will see numerous univariate time series models for forecasting.

Supervised Machine Learning Models

Now that you are familiar with the idea of using the past of one variable, you are going to discover a different approach to making models. You have just seen univariate time series models, which are models that use only the past of a variable itself to predict its future.

Sometimes, this approach is not logical: processes do not always follow trends and seasonality. Some predictions that you would want to make may be dependent on other, independent sources of information: **explanatory variables**.

In those cases, you can use a family of methods called **supervised machine learning** that allows you to model relationships between explanatory variables and a target variable.

A Quick Example of the Supervised Machine Learning Approach

To understand this case, you have the fictitious data in Table 1-3: a new example that contains the sales amount of a company per quarter, with three years of historical data.

Table 1-3 Quarterly sales

Period	Quarterly Sales
Q1 2018	48,000
Q2 2018	20,000
Q3 2018	35,000
Q4 2018	32,0000
Q1 2019	16,000
Q2 2019	58,000
Q3 2019	40,000
Q4 2019	30,000
Q1 2020	32,000
Q2 2020	31,000
Q3 2020	63,000

Period	Quarterly Sales
Q4 2020	57,000

To get this data into Python, you can use the following code (Listing 1-3).

```
quarters = ["Q1 2018", "Q2 2018", "Q3 2018", "Q4
2018",
            "Q1 2019", "Q2 2019", "Q3 2019", "Q4
2019",
            "Q1 2020", "Q2 2020", "Q3 2020", "Q4
2020"]

sales = [48, 20, 42, 32,
         16, 58, 40, 30,
         32, 31, 53, 40]

data = pd.DataFrame({
    'quarter': quarters,
    'sales': sales
})

ax = data.plot.line(x='quarter')
ax.set_title('Sales Per Quarter', fontsize=16)
ax.set_xticklabels(ax.get_xticklabels(),
rotation=45, ha='right')
```

Listing 1-3 Getting the quarterly sales example into Python and plotting the trend

The graph that you obtain is a line graph that shows the sales over time (Figure 1-3).

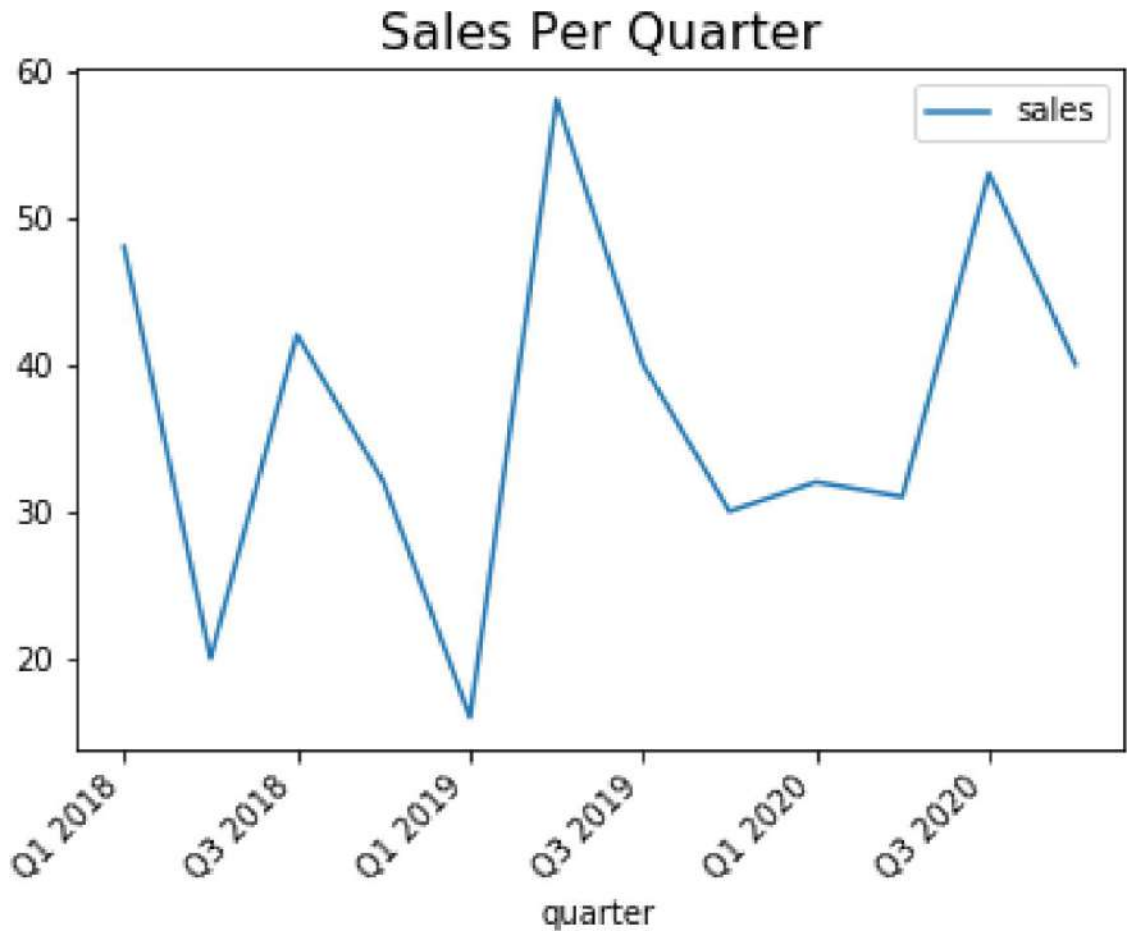


Figure 1-3 Plot of the quarterly sales

What you can see in this graph does not resemble the previous examples: there is no clear linear trend (neither increasing nor decreasing), and there is no clear quarterly seasonality either.

But as the data is about sales, you could imagine many factors that influence the sales that you'll realize. Let's look for explanatory variables that could help in explaining sales. In Table 1-4, the data have been updated with two explanatory variables: discount and advertising budget. Both are potential variables that could influence sales numbers.

Table 1-4 Quarterly sales, discount, and advertising budget

Period	Quarterly Sales	Avg Discount	Advertising Budget
Q1 2018	48,000	4%	500
Q2 2018	20,000	2%	150
Q3 2018	35,000	3%	400

Period	Quarterly Sales	Avg Discount	Advertising Budget
Q4 2018	32,0000	3%	300
Q1 2019	16,000	2%	100
Q2 2019	58,000	6%	500
Q3 2019	40,000	4%	380
Q4 2019	30,000	3%	280
Q1 2020	32,000	3%	290
Q2 2020	31,000	3%	315
Q3 2020	63,000	6%	625
Q4 2020	57,000	6%	585

Let's have a look at whether it would be possible to use those variables for a prediction of sales using Listing 1-4.

```
quarters = ["Q1 2018", "Q2 2018", "Q3 2018", "Q4
2018",
            "Q1 2019", "Q2 2019", "Q3 2019", "Q4
2019",
            "Q1 2020", "Q2 2020", "Q3 2020", "Q4
2020"]

sales = [48, 20, 42, 32,
         16, 58, 40, 30,
         32, 31, 53, 40]

discounts = [4,2,3,
             3,2,6,
             4,3,3,
             3,6,6]

advertising = [500,150,400,
               300,100,500,
               380,280,290,
               315,625,585]

data = pd.DataFrame({
    'quarter': quarters,
```

```
    'sales': sales,
    'discount': discounts,
    'advertising': advertising
})

ax = data.plot.line(x='quarter')
ax.set_title('Sales Per Quarter', fontsize=16)
ax.set_xticklabels(ax.get_xticklabels(),
rotation=45, ha='right')
```

Listing 1-4 Getting the quarterly sales example into Python and plotting the trend

This gives you the graph that is displayed in Figure 1-4: a graph displaying the development of the three variables over time.

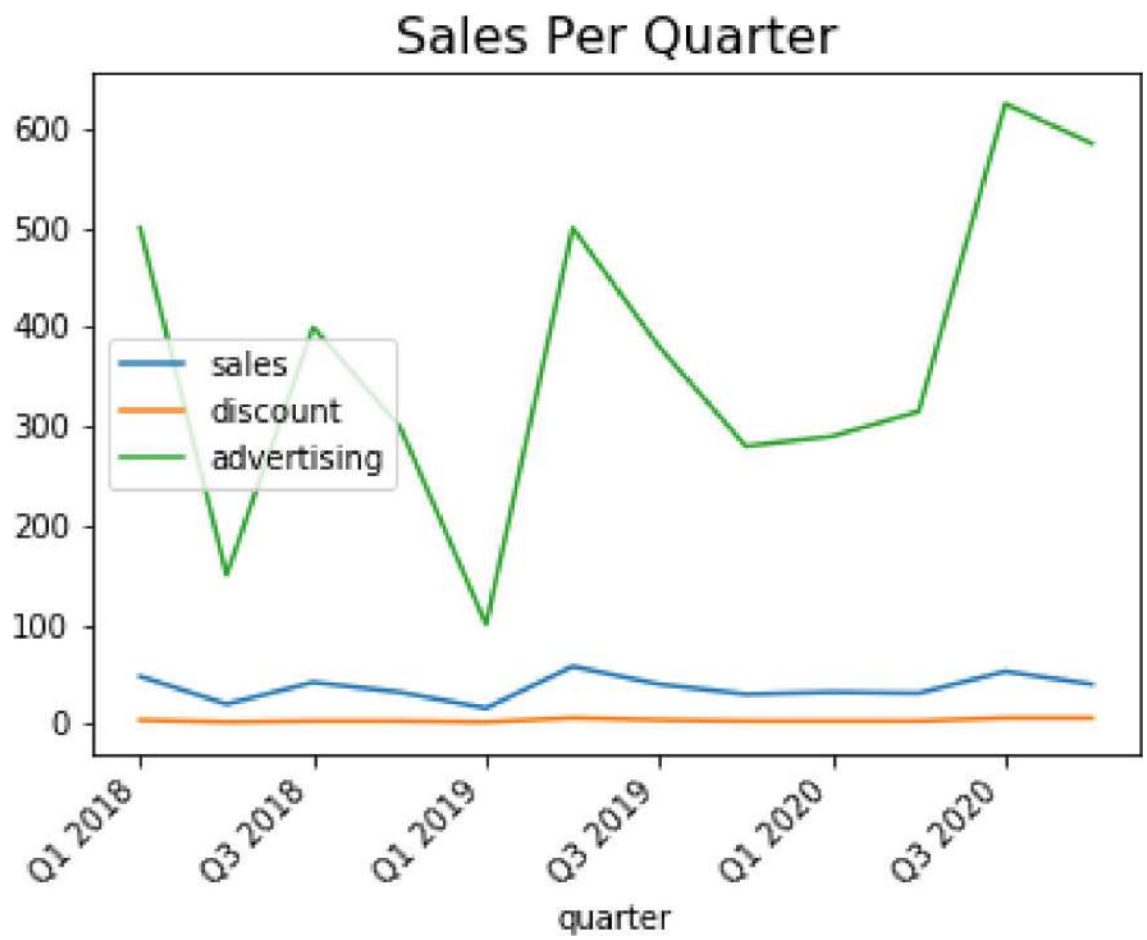


Figure 1-4 Plot of the sales per quarter with correlated variables

At this point, visually, you'd probably say that there is not a very important relationship between the three variables. But let's have a more zoomed-in look at the same graph (Listing 1-5).

```
quarters = ["Q1 2018", "Q2 2018", "Q3 2018", "Q4
2018",
            "Q1 2019", "Q2 2019", "Q3 2019", "Q4
2019",
            "Q1 2020", "Q2 2020", "Q3 2020", "Q4
2020"]

sales = [48, 20, 42, 32,
         16, 58, 40, 30,
         32, 31, 53, 40]

discounts = [4,2,3,
             3,2,6,
             4,3,3,
             3,6,6]

discounts_scale_adjusted = [x * 10 for x in
discounts]

advertising = [500,150,400,
               300,100,500,
               380,280,290,
               315,625,585]

advertising_scale_adjusted = [x / 10 for x in
advertising]

data = pd.DataFrame({
    'quarter': quarters,
    'sales': sales,
    'discount': discounts_scale_adjusted,
    'advertising': advertising_scale_adjusted
})
```

```
ax = data.plot.line(x='quarter')
ax.set_title('Sales Per Quarter', fontsize=16)
ax.set_xticklabels(ax.get_xticklabels(),
rotation=45, ha='right')
```

Listing 1-5 Zooming in on the correlated variables of the quarterly sales example

This gives the graph displayed in Figure 1-5: you can suddenly observe a very clear relationship between the three variables! The relationship was already there in the previous graph (Figure 1-4), but it was just not visually obvious due to the difference in scale of the curves.

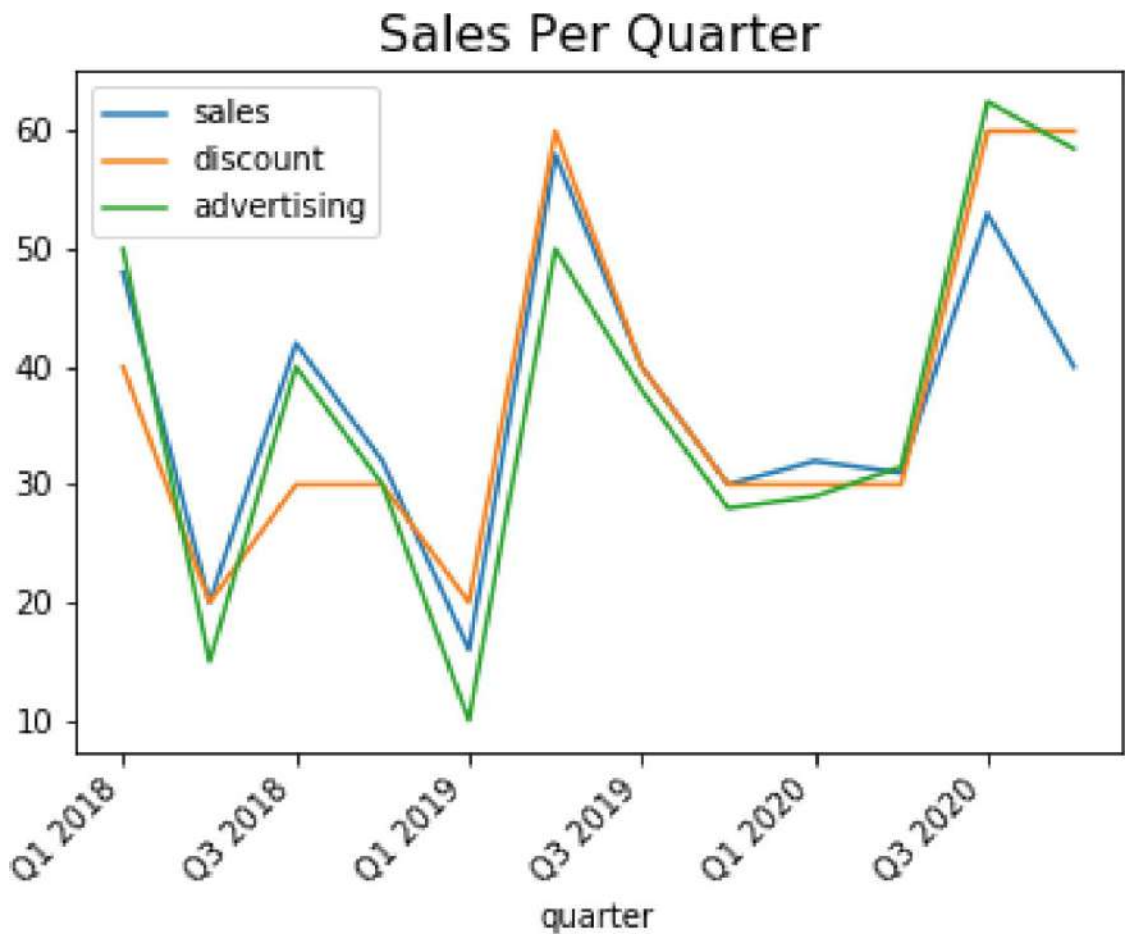


Figure 1-5 Zoomed view of the correlated variables of the quarterly sales example

Imagine you observe a correlation as strong as in Figure 1-5. If you had to do this sales forecast for next month, you could simply ask your colleagues what the average discount is going to be next month and what

next month's advertising budget is, and you would be able to come up with a reasonable guess of the future sales.

This type of relationships is what you are generally looking at when doing supervised Machine Learning. Intelligent use of those relations is the fundamental idea behind the different techniques that you will see throughout this book.

Correlation Coefficient

The visual way to detect correlation is great. Yet, there is a more exact way to investigate relationships between variables: the correlation coefficient. The **correlation coefficient** is a very important measure in statistics and Machine Learning as it determines how much two variables are correlated.

The correlation coefficient between two variables x and y can be computed as

$$r = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 + \sum(y_i - \bar{y})^2}}$$

A **correlation matrix** is a matrix that contains the correlations between each pair of variables in a dataset. Use Listing 1-6 to obtain a correlation matrix.

```
data.drop('quarter', axis=1).corr()
```

Listing 1-6 Getting the quarterly sales example into Python and plotting the trend

It will give you the correlations between each pair of variables in the dataset as shown in Figure 1-6.

	sales	discount	advertising
sales	1.000000	0.848135	0.902568
discount	0.848135	1.000000	0.920958
advertising	0.902568	0.920958	1.000000

Figure 1-6 Correlation table of the quarterly sales example

A correlation coefficient is always **between -1 and 1**. A positive value for the correlation coefficient means that two variables are positively correlated: if one is higher, then the other is generally also higher. If the correlation coefficient is negative, there is a negative correlation: if one value is higher, then the other is generally lower. This is the **direction of the correlation**.

There is also a notion of the **strength of the correlation**. A correlation that is close to 1 or close to -1 is strong. A correlation coefficient that is close to 0 is a weak correlation. Strong correlations are generally more interesting, as an explanatory variable that strongly correlates to your variable can be used for forecasting it.

In the variables in the example, you see a strong positive correlation between sales and discount (0.848) and a strong positive correlation between sales and advertising (0.90). As the correlations are strong, they could be useful in predicting future sales.

You can also observe a strong correlation between the two explanatory variables, discount and advertising (0.92). This is important to notice, because if two explanatory variables are very correlated, it may be of little added value to use them both: they contain almost the same “information,” just measured on a different scale.

Later, you’ll discover numerous mathematical models that will allow you to make models of the relationships between a target variable and explanatory variables. This will help you choose which correlated variables to use in a predictive model.

Other Distinctions in Machine Learning Models

To complete the big picture overview of the Machine Learning landscape, there are a few more groups that need to be mentioned.

Supervised vs. Unsupervised Models

You have just seen what supervised models are all about. But there are also unsupervised models. Unsupervised models differ from all approaches before, as there is no target variable in unsupervised models.

Unsupervised models are great for making segmentations. A classic example is regrouping customers of a store, based on the similarity of those

customers. There are many great use cases for such segmentations, but the approach is not generally useful for forecasting.

Classification vs. Regression Models

Inside the group of supervised models, there is an important split between **classification** and **regression**. Regression is supervised modeling in which the target variable is **numeric**. In the examples that you have seen in the previous sections, there were only numeric target variables (e.g., amount of sales). They would therefore be considered regressions.

In classification, the target variable is **categorical**. An example of classification is predicting whether a specific customer will buy a product (yes or no), based on their customer behavior.

There are many important use cases for classification, but when talking about forecasting the future, it is generally numerical problems and therefore regression models. Think about the weather forecast: rather than trying to predict either hot or cold weather (which are categories of weather), you can try to predict the temperature (numeric). And instead of predicting rainy vs. dry weather (categories), you would predict the percentage of rain.

Univariate vs. Multivariate Models

A last important split between different approaches in Machine Learning is the split between **univariate models** and **multivariate models**.

In the examples you have seen until now, there was always one target variable (sales). But in some cases, you want to predict multiple related variables at the same time. For example, on social media, you may want to forecast how many likes you will receive, but also how many comments.

The first possibility for treating this is to build two models: one model for forecasting likes and a second one for forecasting the number of comments. But some models allow benefiting from a correlation between target variables. These are called multivariate models, and they use the correlation between target variables in such a way that the forecast accuracy improves from forecasting the two at the same time.

You will see multivariate models in Chapters 8 and 9. Multivariate models can be great scientific descriptions of reality, but caution needs to be paid to their predictive performance. Multiple models are sometimes more performant than one model that makes multiple predictions. Data-driven

model evaluation strategies will help you make the best choice. This will be the scope of the next chapter.

Key Takeaways

- Univariate time series models use historical data of a target variable to make predictions.
- Seasonality and trend are important effects that can be used in time series models.
- Supervised Machine Learning uses correlations between variables to forecast a target variable.
- Supervised Machine Learning can be split into classification and regression. In classification, the target variable is categorical; in regression, the target variable is numeric. Regression is most relevant for forecasting.
- The correlation coefficient is a KPI of the relationship between two variables. If the value is close to 1 or -1, the correlation is strong; if it is close to 0, it is weak. A strong correlation between an explanatory variable and the target variable is useful in supervised Machine Learning.
- Univariate models predict one variable; multivariate models predict multiple variables. Most forecasting models are univariate, but some multivariate models exist.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_2

2. Model Evaluation for Forecasting

Joos Korstanje[1]
(1) Maisons Alfort, France

When developing machine learning models, you generally benchmark multiple models during the build phase. Then you estimate the performances of those models and select the model that you consider most likely to perform well. You need objective measures of performance to decide which forecast to retain as your actual forecast.

In this chapter, you'll discover numerous tools for model evaluation. You are going to see different strategies for evaluating Machine Learning models in general and specific adaptations and considerations to take into account for forecasting. You are also going to see different metrics for scoring model performances.

Evaluation with an Example Forecast

Let's look at the purely hypothetical example with stock prices per month of the year 2020 and the forecast that someone has made for this (Table 2-1). Assume that this forecast was made in December 2019 for the complete year and that it has not been updated since.

Table 2-1 Stock prices data

Period	Stock Price	Forecasted
January	35	30
February	35	31
March	10	30
April	5	10

Period	Stock Price	Forecasted
May	8	12
June	10	17
July	15	18
August	20	27
September	23	29
October	21	24
November	22	23
December	25	22

You can already see that there is quite a difference between the actual values and the forecasted values. But that happens. Let's start with getting the data into Python and plotting the two lines using Listing 2-1.

```
import pandas as pd

period = ['Januray', 'February', 'March',
          'April', 'May', 'June',
          'July', 'August', 'September',
          'October', 'November', 'December']

actual = [35, 35, 10,
           5, 8, 10,
           15, 20, 23,
           21, 22, 25]

forecast = [30, 31, 30,
               10, 12, 17,
               18, 27, 29,
               24, 23, 22]

data = pd.DataFrame({
    'period': period,
    'actual': actual,
    'forecast': forecast
})
```

```
ax = data.plot.line(x='period')
ax.set_title('Forecast vs Actual', fontsize=16)
ax.set_xticklabels(ax.get_xticklabels(),
rotation=45, ha='right')
```

Listing 2-1 Getting the stock data example into Python

This should give you the graph in Figure 2-1, which displays the actual values against the forecasted values over time.

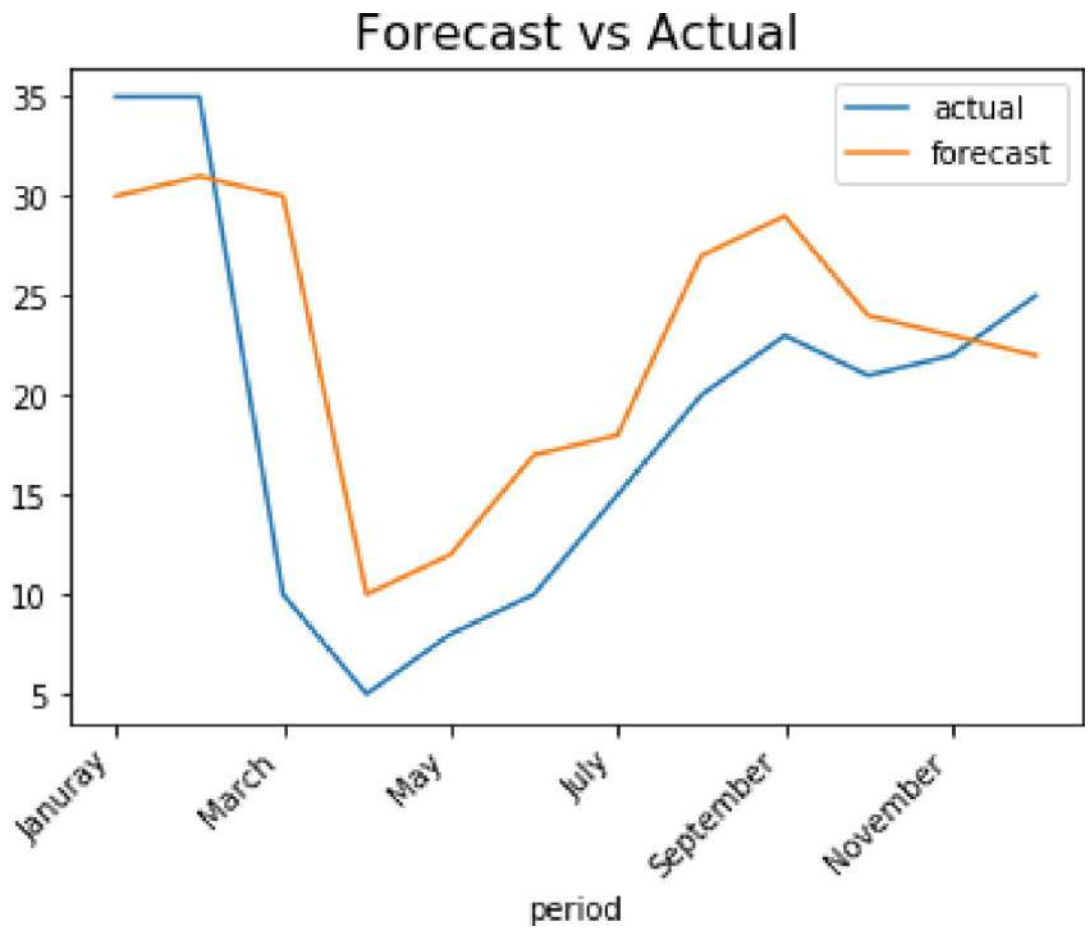


Figure 2-1 Stock prices vs. forecasted stock prices over time

Now the next simple step is to compute the differences between each forecasted value and each actual value, as shown in Table 2-2.

Table 2-2 Adding the errors of each forecasted value to the data

Period	Stock Price	Forecasted	Error

Period	Stock Price	Forecasted	Error
January	35	30	-5
February	35	31	-4
March	10	30	20
April	5	10	5
May	8	12	4
June	10	17	7
July	15	18	3
August	20	27	7
September	23	29	6
October	21	24	3
November	22	23	1
December	25	22	-3

Error Metrics

This month-by-month error is useful information for model improvement. Yet, it remains too detailed to do a model comparison. To do a model comparison faster and easier, you ideally want one or a few KPIs per model. You will now see five common metrics that do exactly this.

Error Metric 1: MSE

The Mean Squared Error is one of the most used metrics in Machine Learning. It is computed as the average of the squared errors. To compute the MSE, you take the errors per row of data, square those errors, and then take the average of them.

$$MSE = \frac{1}{n}\sum(y_i - \hat{y}_i)^2$$

You can compute it in Python using the scikit-learn library, as is done in Listing 2-2.

```
from sklearn.metrics import mean_squared_error
mean_squared_error(data['actual'],
data['forecast'])
```

Listing 2-2 Computing the Mean Squared Error in Python

For the current example, this gives an MSE of **53.6**.

The MSE error metric is great for comparing different models on the same dataset. The scale of the MSE will be the same, and the smaller the error, the better. However, the scale of the metric is not very intuitive, which makes it difficult to interpret outside of benchmarking multiple models.

Error Metric 2: RMSE

The RMSE, or Root Mean Squared Error, is the square root of the Mean Squared Error. The reason for taking the square root of the MSE is that this will get the scale of the error metric back to the scale of your actual values.

$$RMSE = \sqrt{MSE}$$

If you compute the RMSE using Listing 2-3, you will see that the RMSE is **7.3**. This is much more in line with the actual stock prices that you are working with. This can have an advantage for explanation and communication purposes. As the RMSE is an error measure, a lower RMSE indicates a better model.

```
from sklearn.metrics import mean_squared_error
from math import sqrt
sqrt(mean_squared_error(data['actual'],
data['forecast']))
```

Listing 2-3 Computing the Root Mean Squared Error in Python

Although the RMSE is more intuitively understandable, its scale is still dependent on the actual values. This makes it impossible to compare the RMSE values of different datasets with one another, just like the MSE.

Error Metric 3: R2

The R2 (R squared) metric is even more intuitive for communication. The R2 is a value that tends to be between 0 and 1, with 0 being bad and 1 being perfect. It can therefore be easily used as a percentage by multiplying it by 100. This works great for communicating your results. The only case where the R2 can be negative is if your forecast is more than 100% wrong.

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

In Python, you can do this using the code listed in Listing 2-3.

```
from sklearn.metrics import r2_score
r2_score(data['actual'], data['forecast'])
```

Listing 2-4 Computing the R2 in Python

This gives you a value (rounded) of **0.4**. As the R squared is measured on a scale between 0 and 1, you could translate this as 40% accuracy, which, although not technically the best interpretation, will be very clear for your business partners and managers.

Error Metric 4: MAE

The Mean Absolute Error is calculated by taking the absolute differences between the predicted and actual values per row. The average of those absolute errors is the Mean Absolute Error.

$$MAE = \frac{1}{n} \sum |\, y_i - \hat{y}_i|$$

In some way, the MAE is comparable to the RMSE, as they both yield scores that are in the same range of values as the actual values. Therefore, they both have the same interpretation.

The MAE has a more intuitive formula: it is the error metric that most people intuitively come up with. Yet the RMSE is generally favored over the MAE. Since the RMSE uses squares rather than absolute values, the RMSE is easier to use in mathematical computations that demand to take derivatives: the derivative of squared errors is much easier to compute than the derivative of absolute errors.

You can compute the Mean Absolute Error in Python using the code in Listing 2-5. You should obtain an MAE of **5.67**.

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(data['actual'],
data['forecast'])
```

Listing 2-5 Computing the Mean Absolute Error in Python

Error Metric 5: MAPE

The MAPE, short for Mean Absolute Percent Error, is to the R2 what MAE is to the RMSE. The MAPE is computed by taking the error for each prediction, divided by the actual value. Take the absolute values of those percentages per row and compute their average to obtain the MAPE. For the example in Listing 2-6, you'll obtain **0.46**.

$$MAPE = \frac{1}{n} \sum \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

The MAPE has the opposite interpretation from the R2. Where the R2 measures a percentage of goodness of fit, the MAPE measures a percentage error. You can convert the MAPE to a goodness-of-fit measure by computing 1 – MAPE. Although more intuitive, the MAPE has a serious drawback: when the actual value is 0, the formula will lead to division by zero. This makes it problematic to use.

```
from sklearn.metrics import
mean_absolute_percentage_error
mean_absolute_percentage_error(data['actual'],
data['forecast'])
```

Listing 2-6 Computing the Mean Absolute Percent Error in Python

---

Model Evaluation Strategies

Now that you have seen five important metrics, let's look at how to set up tests for model comparison. When you're doing advanced forecasting, you will often be working with a lot of models at the same time. There are a lot of models that you can use, and the remainder of this book is going to present them.

But when working with all those different models, you generally need to make a final forecast: *How to decide which model is most accurate?*

In theory, you could, of course, use multiple models to predict the (short) future and then wait and see which one works best. This does happen in practice, and this is a great way to make sure you deliver quality

forecasts. However, you can do more. Rather than waiting, it is much more interesting to try and use past data to estimate errors.

In practice, the question is: *How to decide which model is most accurate, without waiting for the future to confirm your model*?

Overfit and the Out of Sample Error

When doing forecasting, you are building models on historical data and projecting a forecast into the future. When doing this, it is important to avoid certain biases. A very common bias is to fit a model on historical data, compute errors on this historical data, and if the error is good, to use it for a forecast.

Yet, this will not work due to overfitting. Overfitting a model means that your model has learned the past data too specifically. With the advanced methods that are available on the market, models could get to fit almost any historical trend to 100%. However, this is not at all a guarantee for performance in out-of-sample prediction.

When models overfit, they obtain very high scores on historical data and poor performances on future data. Rather than learning general, true trends, an overfitted model remembers a lot of “useless” and “noisy” variations in the past and will project this noise into the future.

Machine Learning models are powerful learning models: they will learn anything that you give them. But when they overfit, they learn too much. This happens often, and many strategies exist to avoid it.

Strategy 1: Train-Test Split

A first split that is often deployed is the train-test split. When applying a train-test split, you split the rows of data into two. You would generally keep 20% or 30% of data in a test set.

You can then proceed to fit models on the rest of the data: the training data. You can apply many models to the train data and predict on the test data. You compare the performance of the Machine Learning models on the test set, and you will notice that the models with good performance on the train data do not necessarily have a good performance on the test data.

A model with a higher score on the training data and a lower score on the test data is a model with an overfit. In this case, a model with a slightly lower error on the training data may well outperform the first model on the test set.

The test performances are those that matter, as they best replicate the future case: they try to predict on data that is “unknown” to the model: this replicates the situation of your forecast in the future.

Now, a question that you could ask is how to choose the test set. In most Machine Learning problems, you do a random selection of data for the test set. But forecasting is quite particular in this case. As you generally have an order in the data, it makes more sense to keep the test set as the 20% last observations: this will replicate the future situation of application of the forecast.

As an example of applying the train-test split, let’s make a very simple forecasting model: the mean model. It consists of taking the average of the train data and using that as a forecast. Of course, it will not be very performant, but it is often used as a “minimum” benchmark in forecasting models. Any model that is worse than this is not worth being considered at all.

So, let’s see an example with the stock prices data from before using Listing 2-7.

```
y = data['actual']
train, test = train_test_split(y, test_size=0.3,
shuffle=False, random_state=12345)
forecast = train.mean() # forecast is 17.25
train = pd.DataFrame(train)
train['forecast'] = forecast
train_error = mean_squared_error(train['actual'],
train['forecast'])

test = pd.DataFrame(test)
test['forecast'] = forecast
test_error = mean_squared_error(test['actual'],
test['forecast'])
print(train_error, test_error)
```

Listing 2-7 Train-test split in Python

This will give a train error of 122.9375 and a test error of 32.4375. Attention here: with one year of data, a train-test split will cause a problem. The training data will not have a full year of data, which means the model

cannot fully learn any seasonality. In addition, you should strive for having at least three observations per period (in this case, three years of training data).

It is generally a good idea to take seasonality into account when selecting the testing period. In non-forecasting situations, you will often see test sets of 20% or 30% of the data. In forecasting, I advise using a full seasonal period. If you have yearly seasonality, you should use a full year as a test period, to avoid having, for example, a model that works very well in summer, but badly in winter (potentially due to different dynamics in what you are forecasting in different seasons).

Strategy 2: Train-Validation-Test Split

When doing a **model comparison**, you benchmark the performances of many models. As said before, you can avoid overfitting by training the models on the train set and testing the models on the test set.

Adding to this approach, you can add an extra split: the **validation split**. When using the train, validation, and test split, you will train models on the training data, then you benchmark the models on the validation data, and this will be the basis for your model selection. Then, finally, you use your selected model to compute an error on the test set: this should confirm the estimated error of your model.

In case you have an error on the test set that is significantly worse than the error on the validation set, this will alert you that the model is not as good as you expected: the validation error is underestimated and should be reinvestigated.

When using the train-validation-test split, there is a serious amount of data dedicated fully to model comparison and testing. Therefore, this approach should be avoided when working with little data.

As said before, it is important to consider which periods you leave out of your validation and test data. If, for example, you have five years of monthly data and you decide to use three years for training, one year for validation, and one year for testing, you are missing out on the two most recent years of data: not a good idea.

In this case, you could select the best model (i.e., the **model type** and its optimal **hyperparameters**), and once the model is selected and benchmarked, you retrain the optimal model including the most recent data.

Listing 2-8 shows a very basic example in which two models are compared: the mean and the median. Of course, those models should not be expected to be performant: the goal here is to show how to use the validation data for model comparison.

```
# Splitting into 70% train, 15% validation and 15%
test
train, test = train_test_split(data['actual'],
test_size = 0.3, shuffle = False,
random_state=12345)
val, test = train_test_split(test, test_size =
0.5, shuffle = False, random_state=12345)

# Fit (estimate) the two models on the train data
forecast_mean = train.mean() # 17.25
forecast_median = train.median() # 12.5

# Compute MSE on validation data for both models
val = pd.DataFrame(val)

val['forecast_mean'] = forecast_mean
val['forecast_median'] = forecast_median

mean_val_mse = mean_squared_error(val['actual'],
val['forecast_mean'])
median_val_mse = mean_squared_error(val['actual'],
val['forecast_median'])

# You observe the following validation mse: mean
mse: 23.56, median mse: 91.25
print(mean_val_mse, median_val_mse)

# The best performance is the mean model, so
verify its error on test data
test = pd.DataFrame(test)
test['forecast_mean'] = forecast_mean
```

```
mean_test_mse = mean_squared_error(test['actual'],
test['forecast_mean'])

# You observe a test mse of 41.3125, almost double
the validation mse
print(mean_test_mse)
```

Listing 2-8 Train-validation-test split in Python

If you follow the example, you observe a validation MSE for the mean model of **23.56** and **91.25** for the median model. This would be a reason to retain the mean model in favor of the median model. As a final evaluation, you verify whether the validation error is not influenced by a selection of the validation set that is (randomly) favorable to the mean model.

This is done by taking a final error measure on the test set. As both the validation and the test set were unseen data for the mean model, their errors should be close. However, you observe a test MSE of **41.3125**, almost double the validation error. The fact that the test error is far off from the validation error tells you that there is a bias in your error estimate. In this particular case, the bias is rather easy to find: there are too little data points in the validation set (only two data points), which means that the error estimate is not reliable.

The train-validation-test set would have given you an early warning on the error metrics of your model. It would therefore have prevented you from relying on this forecast for future estimates. Although an evaluation strategy cannot improve a model, it can definitely improve your choice of forecasting models! It therefore has an indirect impact on your model's accuracy.

Strategy 3: Cross-Validation for Forecasting

A problem with the train-test set may occur when you have very little data. In some cases, you cannot "afford" to keep 20% of the data apart from the model. In this case, cross-validation is a great solution.

K-Fold Cross-Validation

The most common type of cross-validation is K-fold cross-validation. K-fold cross-validation is like an addition to the train-test split. In the train-test split, you compute the error of each model one time: on the test data. In k-

fold cross-validation, you fit the same model k times, and you evaluate the error each time. You then obtain k error measures of which you take the average. This is your cross-validation error.

To do this, a K-fold cross-validation makes a number of different train-test splits. If you choose a k of 10, you will split the data into 10. Then, each of the ten parts will serve as test data one time. This means that the remaining nine parts are used as training data.

As values for k, you are generally looking between 3 and 10. If you go much higher, the test data becomes small, which can lead to biased error estimates. If you go too low, you have very little folds, and you lose the added value of cross-validation. Note that when k is 1, you are simply applying a train-test split.

Figure 2-2 K-fold cross-validation in Python

As a hypothetical example, imagine a case with 100 data points and a five-fold cross-validation. You will make five different train-test datasets, as you can see in the schematic illustration in Figure 2-2.

On those five train-test splits, you train the model on the observations that have been selected as training data, and you compute the error on the observations that have been selected as the test data. The cross-validation error is then the average of the five test errors (Listing 2-9).

```
import numpy as np
from sklearn.model_selection import KFold

kf = KFold(n_splits=5)

errors = []
for train_index, test_index in kf.split(data):
    train = data.iloc[train_index,:]
    test = data.iloc[test_index,:]

    pred = train['actual'].mean()
    test['forecast'] = pred
    error = mean_squared_error(test['actual'],
test['forecast'])
    errors.append(error)

print(np.mean(errors))
```

Listing 2-9 K-fold cross-validation in Python

This example will give you a cross-validation error of **106.1**. This is an MSE, so it should only be compared with errors of other models on the same dataset.

Time Series Cross-Validation

The fact that you have multiple estimates of the error will improve the error estimate. As you can imagine, when you have only one estimate of the error, your test data may be is very favorable: all difficult to predict events may have fallen in the training data! Cross-validation reduces this risk, by having an evaluation applied to all the data. This makes the error estimate generally more reliable.

Yet, attention must be paid to the specific case of forecasting. As you can see in the image, K-fold cross-validation creates test splits equally throughout the data. In the example, you can see many cases where the test set is temporally before the train set. This means that you are measuring the error of forecasting the past rather than the future using these methods.

For the category of supervised models, in which you use relations between your target variable and a set of explanatory variables, this is not

necessarily a problem: the relationships between variables can be assumed to be the same in the past and the present.

However, for the category of time series models, this is a serious problem. Time series models are generally based on making forecasts based on trends and/or seasonality: they use the past of the target variable to forecast the future.

A first problem you'll encounter is that many time series models will not work with missing data: if a month is missing in the middle, the methods can simply not be estimated.

A second problem is that when the models can be estimated, they are often not realistically accurate: estimating a period in between data points is much easier than estimating a period that is totally in the future.

A solution that can be used is called **the time series split**. In this approach, you take only data that is before the test period for model training. In this approach, you take only data that is before the test period for model training, as shown schematically in Figure 2-3.

Figure 2-3 Time series cross-validation in Python

You can apply time series cross-validation using the code in Listing 2-10.

```
from sklearn.model_selection import
TimeSeriesSplit
tscv = TimeSeriesSplit()
```

```
errors = []
for train_index, test_index in tscv.split(data):
    train = data.iloc[train_index,:]
    test = data.iloc[test_index,:]
    pred = train['actual'].mean()
    test['forecast'] = pred
    error = mean_squared_error(test['actual'],
test['forecast'])
    errors.append(error)

print(np.mean(errors))
```

Listing 2-10 Time series cross-validation in Python

This method of cross-validation estimates the error to be **194.7**. Again, this error should not be used as a stand-alone, but as a way to compare performances of multiple methods on the same data.

This method can at least be used on time series modes, as there are no gaps in the data. However, with this method, you have a bias. For the first fold, you will have much less historical data than for the last fold. This means that the errors of the folds will not be comparable: if they cannot use enough historical data for fitting the model, the errors of the first folds could be considered unfair. This is an inherent weakness of this method that you should keep in mind if you use it.

Rolling Time Series Cross-Validation

An alternative that has been proposed is the **rolling time series split**. It uses the same period for each fold in the cross-validation. You can see the schematic overview of this method in Figure 2-4. This will solve the problem of having unequal errors: they are all trained with the same amount of historical data. Yet a problem remains that you cannot base your model on data very far into the past using this method.

But it should only be used in situations where you have a lot of data or where you don’t have to look very far in the past for predicting the future.

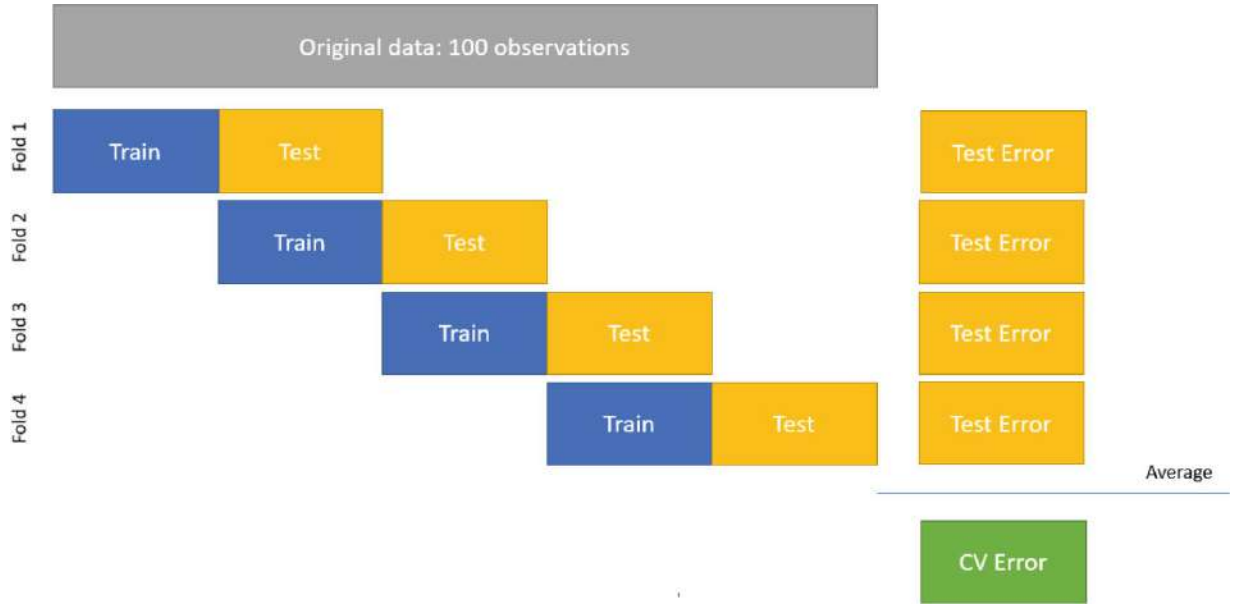


Figure 2-4 Rolling time series cross-validation in Python

You can use the code in Listing 2-11 to execute the rolling time series cross-validation on the stock prices example.

```
from sklearn.model_selection import
TimeSeriesSplit
tscv = TimeSeriesSplit(max_train_size = 2)

errors = []
for train_index, test_index in tscv.split(data):
    train = data.iloc[train_index,:]
    test = data.iloc[test_index,:]

    pred = train['actual'].mean()
    test['forecast'] = pred
    error = mean_squared_error(test['actual'],
test['forecast'])
    errors.append(error)

print(np.mean(errors))
```

Listing 2-11 Rolling time series cross-validation in Python

This method of model evaluation estimates the error to be **174.0.** You have now seen three cross-validation errors that are all different. Yet, each of them uses the same model. This tells you two things: firstly, it insists on only comparing error estimates using the same metric, using the same dataset. Secondly, you have observed a significantly lower error in the rolling time series split which may be a hint that using the model with very little historical data (rolling) could be the more performant model in this case. Yet, to be confident of that conclusion, you would need to study the results a bit further. A great starting point would be to add multiple models into the benchmark. This is left for later, as you will discover numerous models throughout this book.

Backtesting

As a last strategy, I want to mention backtesting. Backtesting is a term that is not much practiced in data science and Machine Learning. Yet, it is the go-to model validation technique in stock trading and finance. Backtesting builds on the same principles as model evaluation.

There is a fundamental difference between backtesting and model evaluation using metrics. In backtesting, rather than measuring the accuracy of a forecasting model, you measure the result of a (stock trading) strategy. Rather than trying to forecast the future prices of stocks, you define at which event or trigger you want to buy or sell. For example, you could define a strategy stating that you will buy a given stock at price x and sell at price y.

To evaluate the performance of this strategy, you run the strategy on historical data of stock prices: what would have happened if you had sold and bought stocks at the prices you indicated. Then you evaluate how much profit you would have obtained.

Although backtesting is applied to trading strategies rather than to forecasts, I found it interesting to list it in the chapter on model evaluation, as an alternative to model evaluation in forecasting.

There are several Python libraries for finance, who propose backtesting solutions. A very simple to use example is **fastquant**, available on GitHub over here: `https://github.com/enzoampil/fastquant`.

Which Strategy to Use for Safe Forecasts?

After we've seen multiple approaches for forecasting model evaluation, you may wonder which solution you should choose. I'd argue that you do not have to choose. As a forecaster or modeler, your most important task is being confident in the evaluation of your model. A reliable evaluation strategy is the key to success! So rather than choosing one strategy, I'd advise you to use a combination of strategies that makes you feel the most confident about your predictions.

A **combined strategy** that you can use is to train and tune your models using cross-validation on the training data. As a second step, you can measure predictive errors on the validation data: the model that has the best error on the validation data will be your preferred model. As a last verification (you can never be sure enough), you make a prediction on the test data with the selected model, and you make sure that the error is the same as the error observed on the validation data. If this is the case, you can be confident that your model should be delivering the same range of errors on future data.

As you do more predictive forecasts, you will develop a feel for which evaluation metrics are important to consider. Just remember that the goal of model evaluation is twofold: on one hand, you use it to improve model performance, and on the other hand, you use it for objective performance measures.

Be aware that, sometimes, predictive modelers are focused so much on the performance improvement part that they start tweaking their evaluation methods to improve their performance scores. This is to be avoided: you want to be as confident as possible about the future performance of your model. Metrics are one important part of this; putting in place an objective model evaluation strategy is even more important.

Final Considerations on Model Evaluation

Besides looking at metrics, you also need to understand what your model is fitting: Which explanatory variables are taken into account by the model? Do those variables make sense intuitively, and do they fit the business knowledge about your data?

This scientific, or explanatory, understanding of your model will give you additional confidence, which is an important complement to the numerical minimization of forecasting errors.

For example, if you predict the stock market, you may obtain a 90% accuracy based on the past. But by reflecting on what you are doing, you may realize that in past data, there has not been a simple market crash present. And you may understand from this that your model will go completely wrong in the case of a market crash, while that is maybe the moment when you are the most at risk ever!

A great book that talks about this is *The Black Swan*, in which author Nicholas Taleb insists on the importance of taking into account very rare events when working on stock market trading. A takeaway here is that extreme crises in the history of stock trading have been very influential on trading success, while methods used for day-to-day trading do not (always) take into account such risk management. You may even prefer a model that is worse on a day-to-day basis, but better in predicting huge crashes.

In short, you do not always have to obtain the lowest possible Mean Squared Error! Sometimes, stable performances are the most important. Other times, you may want to avoid any overestimation, while underestimations are not a problem (e.g., in a restaurant, it may be worse to buy too much food due to an overestimated demand forecast, rather than too little). To get to this level of understanding, it is important to understand your models deeply. A model is not a black box that makes predictions for you: it is a mathematical formula that you are building. For good results, you need to evaluate the performances of the formula using evaluation criteria that fit your use case: many strategies and metrics are available, but no one size fits all. Only you can know which evaluation method fits the success criteria of your particular problem statement.

Key Takeaways

- Metrics
  - The most suitable metrics for regression problems are R Squared (R2), Root Mean Squared Error, and Mean Squared Error.
  - The R2 gives a percentage-like value. The RMSE gives a value on the scale of the actuals. The MSE gives a value on a scale that is difficult

to interpret.
    - Metrics should be used for benchmarking different models on one and the same dataset.
- Model evaluation strategies
    - Cross-validation gives you a very reliable error estimate. Adaptations are necessary to make it work for time series.
    - Train, test, and validation split can be used for benchmarking.
    - A combined strategy will give you the safest estimate: Use cross-validation on the training data, validation data for model selection, and test data for a last estimate of the error.
- Overfitting models
    - If your model learns too much from the training data and will not generalize into the future, it is overfitting.
    - Overfit is identified by a good performance on the train data, but a bad performance on the test data.
- Underfitting models
    - If your model does not learn enough from the training data, it is underfitting.
    - Underfit is determined by a bad performance on the training data.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_3

3. Model Management and Benchmarking Using MLflow

Joos Korstanje[1]
(1) Maisons Alfort, France

As you have seen in the previous chapter, an essential part of building machine learning models is model evaluation. Once your machine learning use case is clearly identified, you will generally test multiple machine learning models using the same evaluation criteria. This will allow you to have a fair comparison between the models.

During this development phase, you are likely going to be playing around with a lot of parameters of your machine learning pipeline. In order to get to the best performing model, you may be testing the performance impact of using different datasets, different data preparation and feature engineering approaches, different models, etc.

Throughout this process, it is easy to lose track of some of your settings. Imagine how frustrating it would be to know that you have built a model with an impressive performance, only to be unable to reproduce the results!

In this chapter, you will discover MLflow. MLflow is built especially for tracking machine learning experimentations. You will learn how to set up MLflow, track your machine learning experiments with it, save your work safely, and obtain an easy-to-use interface for it.

Introduction to MLflow

Over the past few years, MLflow has become the industry standard tool for experiment tracking and model management. The first main reason for this is of course that MLflow solves the problem of experiment tracking and model management very efficiently.

Yet, there are other reasons as well. For example, the fact that MLflow is an open source tool means that it is free to use. It also integrates easily with Python, and it can therefore integrate with a large number of software and cloud vendors.

Since MLflow has decided to focus on the problem of model evaluation and management, it is compatible with all sorts of machine learning technology. In this book, we focus on forecasting, but MLflow can be used for any machine learning use

case, ranging from simple statistical models to advanced deep learning and LLM use cases.

In short, MLflow is an important tool to have in your machine learning toolkit. This is why you'll be seeing MLflow model tracking being used throughout the remainder of this book. In this chapter, we are going to explore one of its main functionalities: experiment tracking.

Local vs. Hosted MLflow

If you're using MLflow in a company, you may encounter hosted versions of MLflow: maybe there is a shared MLflow for your team or even for the whole company.

The easiest method for getting started is the local setup. Although it will not allow for collaborating with others on your models, the local setup will allow you to use all the basic functionalities of MLflow, as well as allow you to learn on your own. Once you master the use of MLflow in your local environment, you will be able to use the documentation on mlflow.org to move on to other environments.

Setting Up MLflow Locally

Let's start by having a look at the setup and installation of MLflow. To follow along, you can use the notebook in the git repository. Installing and importing MLflow in Python is shown in Listing 3-1.

```
# If you do not have mlflow installed, use pip to install
it
!pip install mlflow

# Simply import the mlflow package
import mlflow
```

Listing 3-1 Installing MLflow

At the time of writing, an environment running Python 3.9 is required for MLflow to work, but make sure to check the environment requirements in their documentation. If you're unsure about creating Python environments, the easiest tool for getting started is Anaconda Navigator.

You can download and install Anaconda from anaconda.org. To create an environment, you open Anaconda Navigator and go to the Environments tab. Click Create, type in a new (e.g., chapter3-env), and select a Python version (e.g., 3.9.21).

As Python versions are coming out regularly, Python packages are maintained independently from Python version releases, packages not always compatible with every Python version. Although it is a pain to manage those package versions and Python version incompatibilities, tools like Anaconda and other environment managers can make this a lot easier for you.

Experiment Tracking Using MLflow

Let's do a simple example of training a model using MLflow. As we will go in depth into different machine learning algorithms throughout this book, let's ignore the exact model training code for now and focus on the MLflow code.

In Listing 3-2, you'll see a function that trains a machine learning model (we'll ignore this detail for now).

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression

def train_and_predict_model():
    # get the famous diabetes dataset through sklearn
    X, y = load_diabetes(return_X_y=True)

    # instantiate a linear regression model
    linear_model = LinearRegression()

    # fit the model on the diabetes data
    linear_model.fit(X, y)

    # predict on the training data
    preds = linear_model.predict(X)
```

Listing 3-2 The model code (ignore details for now)

In Listing 3-3, you see how to start an MLflow tracking: it is as simple as executing the line mlflow.autolog(). After that, anything you do is tracked automatically by MLflow!

```
# Start mlflow auto logging
mlflow.autolog()

# Call the model training function
train_and_predict_model()
```

Listing 3-3 Training a model using MLflow autologging

The notebook will tell you the run ID of your MLflow run. This is important for finding your logged metrics. This looks like the message shown in Listing 3-4.

```
2025/05/13 16:46:56 INFO mlflow.utils.autologging_utils:
Created MLflow autologging run with ID
'39bc21bad4df490abcfd03ba28155479', which will track
```

```
hyperparameters, performance metrics, model artifacts, and
lineage information for the current sklearn workflow
```

Listing 3-4 The INFO message containing the run ID

MLflow Data

You have now trained a model using MLflow autologging. Let's figure out how to find our logs and make some sense of them.

When using MLflow autologging using the basic setup that we used here, MLflow will automatically create data on your local file system. You will see in your local file system that a new folder has been created called "mlruns". Inside, you'll be able to find a folder that has the run ID from Listing 3-4 as the folder name. In the example case, this is 39bc21bad4df490abcfd03ba28155479, but it will be a different ID when you run the notebook.

The folder 39bc21bad4df490abcfd03ba28155479 contains the following contents:

```
├── artifacts
│   ├── estimator.html
│   └── model
│       ├── MLmodel
│       ├── conda.yaml
│       ├── model.pkl
│       ├── python_env.yaml
│       └── requirements.txt
├── inputs
│   ├── cf3623947cb69772f970788c5c749443
│   │   └── meta.yaml
│   └── e736aaed654d29e199bee1101fb917ca
│       └── meta.yaml
├── meta.yaml
├── metrics
│   ├── training_mean_absolute_error
│   ├── training_mean_squared_error
│   ├── training_r2_score
│   ├── training_root_mean_squared_error
│   └── training_score
├── params
│   ├── copy_X
│   ├── fit_intercept
│   ├── n_jobs
│   └── positive
└── tags
    ├── estimator_class
```

```
├── estimator_name
├── mlflow.autologging
├── mlflow.log-model.history
├── mlflow.runName
├── mlflow.source.name
├── mlflow.source.type
└── mlflow.user
```

Let's go over the most important data inside these folders to gain an understanding of their contents.

Artifacts

The first directory that we are going to look at is the artifacts directory. A model training loop can generate multiple stored objects, of which the most important is a stored version of the model.

It is extremely important to store your model after training, as the training process is often long and hardware-intensive, and if your resulting model's performance is satisfactory, you'd want to be able to reuse it without going through the training phase. Also, model training can sometimes contain randomness, and it may be impossible to obtain the same exact performance, even when rerunning the exact same Python code that you used the first time.

The model artifactory is stored as a Pickle file: model.pkl. Pickle is a Python package for serializing Python objects. It is commonly used for saving machine learning models as well.

The other files that are present in the artifacts directory are related to the Python environment required to use this model. The model object can only be used as long as it remains compatible with the Python packages. For example, if you want to open a model that was generated many versions of scikit-learn ago, the model may not work. In that case, you can simply install the version of scikit-learn that is listed in the requirements file, and all will work fine.

Metrics

The metrics directory will contain all metrics that MLflow has logged automatically. MLflow will automatically select a number of useful metrics. This choice is based on the Python package used for the model training, as well as the specific model used.

In the current example, you can see that five files have been created: "training_mean_absolute_error", "training_mean_squared_error", "training_r2_score", "training_root_mean_squared_error", and "training_score". These files do not have a file extension, but they are plain text files and can be read with any text editor of your choice.

As we have covered metrics extensively in the previous chapter, we will not go deeper into the exact metric definitions. Just know that you can find the metric values of the current run ID in this directory.

Params

Params are the hyperparameters that were set for the training. Based on the model and library used, the hyperparameters will be logged automatically by MLflow autologging. In the current example, you can see the following hyperparameters being logged: “copy_X”, “fit_intercept”, “n_jobs”, and “positive”. Each of them is a plain text file containing the value for this parameter.

Tags

Finally, tags is also worth mentioning. As said earlier, you can refer to the run IDs to find models and refer to them. However, the run ID gives no information at all on the run that you did. If you want to help yourself or your teammates understand the specifics of each run, it is a good practice to set tags for the autologging, as shown in Listing 3-5.

```
# Start mlflow auto logging
mlflow.autolog(
    extra_tags={"code_listing": "3-5"},
)

# Call the model training function
train_and_predict_model()
```

Listing 3-5 Giving the run an additional tag to remember which code listing it belongs to

Inspecting the Model Logs Through Python

Now that you have seen where the model data are being stored on the local file system, you understand how MLflow works in the back. However, you will not have to go browsing through these files each time you’re looking for this data.

A first method for accessing this data is provided in the MLflow package. You can use the code in Listing 3-6 to load a specific run into the Python session and access the model and its metrics.

```
# Specify the path to the model and run that you want to
load
model_path =
'mlruns/0/39bc21bad4df490abcfd03ba28155479/artifacts/model'

# Load the model
loaded_model = mlflow.sklearn.load_model(model_path)

# Use the model, in this case for example for making a
prediction on the diabetes data
X, y = load_diabetes(return_X_y=True)
loaded_model.predict(X)
```

Listing 3-6 Retrieving the model through Python

The MLflow UI

Although going through the file system or going through Python are acceptable methods to retrieve your training logs, the real added value of using MLflow is the MLflow UI. You can start the MLflow UI using the following command (Listing 3-7).

```
!mlflow ui --port 8080
```

Listing 3-7 Start the MLflow UI

After running this command, go to your internet browser and go to the following URL: http://localhost:8080 (Figure 3-1).

Once you go here, you'll see an extensive page giving you all the runs that you have executed (there are two runs in the example image).

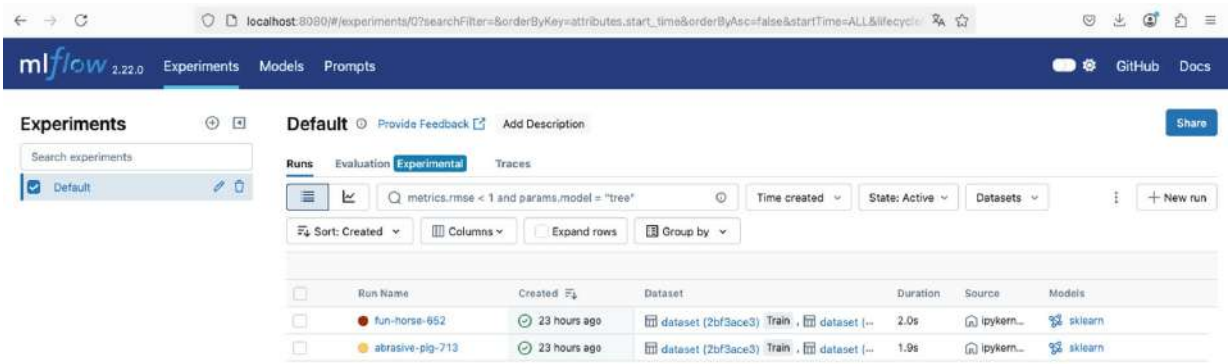


Figure 3-1 MLflow Experiments interface

When you click on your run name (the name is randomly generated by MLflow), you'll get to the page shown in Figure 3-2, which shows all data that you have seen before when browsing with the file system, including metrics, parameters, artifacts, and more.

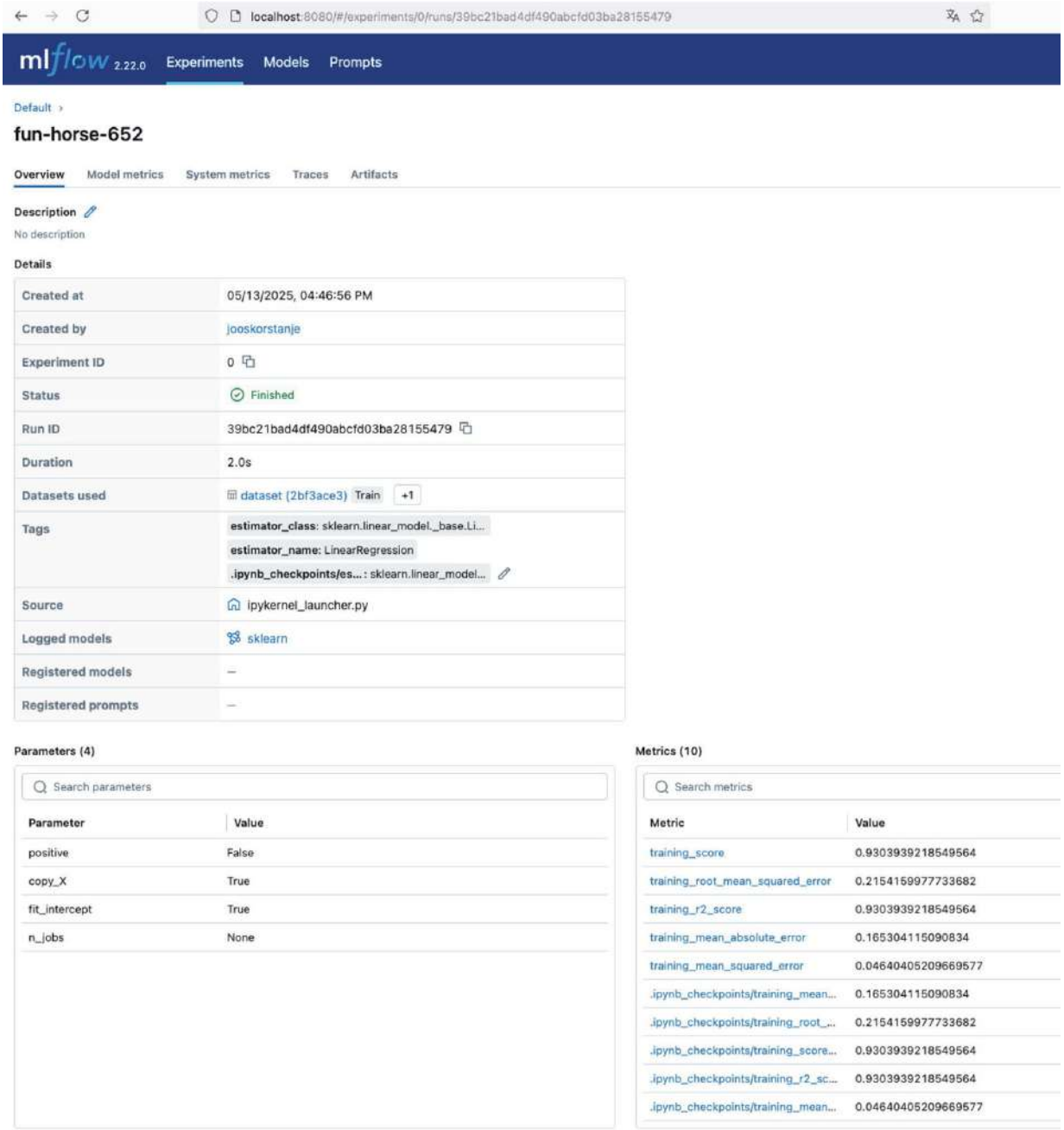


Figure 3-2 MLflow run details page

MLflow Throughout This Book

In the remainder of this book, MLflow tracking will be used extensively. As you use the code and Jupyter Notebooks provided, MLflow tracking will store the results on your local file system. Mastering the MLflow UI will improve the learning experience, as you'll be able to easily compare models from different chapters, without having to rerun any code. If you are motivated to go beyond, you could even try applying

different models to the use cases that will be presented and set up your own benchmarks throughout the book.

Key Takeaways

- MLflow is an industry-standard tool for machine learning experiment tracking.
- It is open source and easily usable as a Python package.
- The MLflow UI is a very powerful tool for model benchmarking and allows for keeping track of your experiments.
- MLflow has many more use cases beyond experiment tracking. Feel free to check out the MLflow documentation or other detailed resources.

Part II
Univariate Time Series Models

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_4

4. The AR Model

Joos Korstanje[1] ✉
(1) Maisons Alfort, France

In this chapter, you will discover the AR model: the autoregressive model. The AR model is the most basic building block of univariate time series. As you have seen before, univariate time series are a family of models that use only information about the past of the *target variable* to forecast its future, and they do not rely on other *explanatory variables*.

Univariate time series models' work can be intuitively understood as building blocks: they add up from the simplest model to complex models that combine the different effects described by individual models. The AR model is the simplest model of the univariate time series models.

Throughout the following chapters, you will discover more building blocks to add to this basis. This builds up from the AR model to the SARIMA model. You can see a schematic overview of the building blocks in univariate time series in Table 4-1.

Table 4-1 The building blocks of univariate time series models

Name	Explanation	Chapter
AR	Autoregression	4
MA	Moving Average	5
ARMA	Combination of AR and MA models	6
ARIMA	Adding differencing (i) to the ARMA model	7
SARIMA	Adding seasonality (S) to the ARIMA model	8
SARIMAX	Adding external variables (X) to the SARIMA model *(note that external variables mean that it is not univariate anymore)*	9

I must give you an alert here. As I stated in the first chapter, the strong point of this book is that it contains real-life examples and honest and objective model evaluations. I cannot insist enough on the importance of model evaluation, as the goal of any forecasting model should be to obtain the best predictive performance.

With the AR model being the simplest building block, it would be unrealistic to expect great predictive performance on most real-life datasets. Although I could have tweaked the example to fit perfectly or work with a simulated dataset, I prefer to show you the weaknesses of certain models as well.

The AR model is great to start learning about univariate time series, but it is unlikely that you will use an AR model without its brothers and sisters in practice. You would generally work with one of the combined models (SARIMA or SARIMAX) and test which of these building blocks improves predictive performance on your forecast.

Autocorrelation: The Past Influences the Present

The autoregressive model describes a relationship between the present of a variable and its past. It is therefore suitable for variables in which the past and present values correlate.

As an intuitive example, consider the waiting line at the doctor's. Imagine that the doctor has a plan in which each patient has 20 minutes with the doctor. If every patient takes exactly 20 minutes, this works fine. But what if a patient takes a bit more time? An **autocorrelation** could be present if the duration of a consultation has an impact on the duration of the next consultation. So, if the doctor needs to speed up a consultation because the previous consultation took too long, you observe a correlation between past and present. Past values influence future values.

Compute Autocorrelation in Earthquake Counts

Throughout this chapter, you will see examples applied to the well-known **USGS Significant Earthquake dataset (*Data by USGS, license CC0: Public Domain*)**. This dataset is collected by the US National Earthquake Information Center. You can find a copy of the dataset in the GitHub repository of this book.

Note To get a very quick description of a dataset, you can use **df.describe()** to get quick statistics for each column of your dataframe: count, mean, standard deviation, minimum, maximum, and quantiles. For a more thorough description, you can use the **pandas profiling package.** Examples of both are given in the GitHub **repository** of this book.

If you import the data using pandas, you can retrieve a description of the dataframe using the **describe** method, as is shown in Listing 4-1.

```
import pandas as pd

# Import the dataframe
eq = pd.read_csv('Earthquake_database.csv')

# Describe the dataframe
eq.describe()
```

Listing 4-1 Describing a dataframe

Feel free to scroll through this descriptive table to get a better understanding of what the data is about. If you want a more detailed description of the data, you can use the **ydata-profiling package**, which automatically creates a very detailed description of the variables of a dataframe. You can use Listing 4-2 to get a **profile report**.

```
# Import the pandas profiling package
from ydata_profiling import ProfileReport

# Get the pandas profiling report
profile = ProfileReport(eq, title="Profiling
Report")

# Show the profile report
profile
```

Listing 4-2 Profiling a dataframe

The task that you'll be performing is forecasting the **number of strong earthquakes per year**. The dataset is currently not in the right format to do

this, as it has one line per earthquake and not one line per year. To prepare the data for further analysis, you will need to aggregate the data using Listing 4-3.

```
%matplotlib inline
import matplotlib.pyplot as plt

# Convert years to dates
eq['year'] = pd.to_datetime(eq['Date'],
format="mixed", utc=True).dt.year

# Filter on earthquakes with magnitude of 7 or
higher
eq = eq[eq['Magnitude'] >= 7]

# Compute a count of earthquakes per year
earthquakes_per_year = eq.groupby('year').count()

# Remove erroneous values for year
earthquakes_per_year =
earthquakes_per_year.iloc[1:-2, 0]

# Make a plot of earthquakes per year
ax = earthquakes_per_year.plot()
ax.set_ylabel("Number of Earthquakes")
plt.show()
```

Listing 4-3 Convert the earthquake data to the yearly number of earthquakes

This code should give you the graph displayed in Figure 4-1. You can see that there might be some relationship between years. One year it's higher and one year it's lower. This could be due to a negative autocorrelation. Remember that a negative correlation implies that a higher value in one variable corresponds to a lower value in the second variable. For autocorrelation, this means that a higher value this year corresponds with a lower value next year.

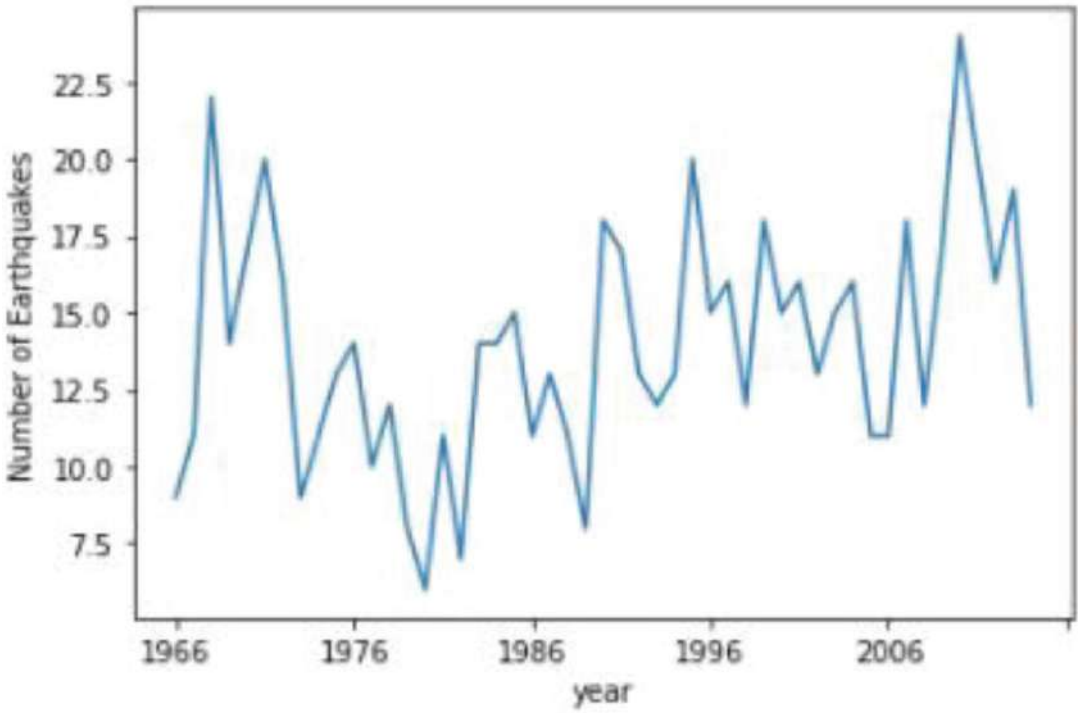


Figure 4-1 Evaluation of the number of earthquakes per year

Now that you have obtained the number of earthquakes per year, you should **check for autocorrelation numerically**. You need to find out whether there is a correlation between the number of earthquakes in any given year and the year before it.

To do this, you can use the same method for computing correlation as seen in Chapter 2. Correlation is computed pairwise, on two columns. But for now, you just have one column (the data per year). You'll need to add a column in the data containing the data for the previous year. This can be obtained by applying a **shift** to the original data and concatenating the shifted data with the original data. Listing 4-4 shows you how to do this. Figure 4-2 shows you how the data has shifted one year back.

```
shifts = pd.DataFrame(
    {
        'this year': earthquakes_per_year,
        'past year': earthquakes_per_year.shift(1)
    }
)
```

```
ax = shifts.plot()
ax.set_ylabel('Number of Earthquakes')
plt.show()
```

Listing 4-4 Convert the earthquake data to the yearly number of earthquakes

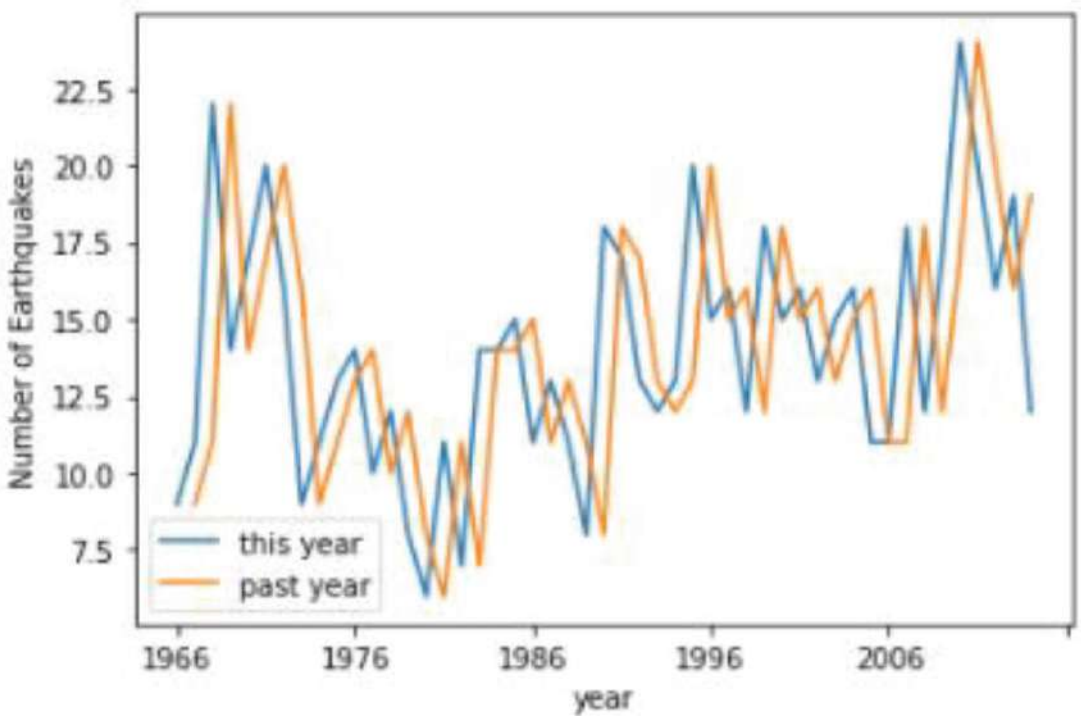


Figure 4-2 Shifted data

This works well, except for one thing. As you can see in the data, there are observations from 1966 to 2014. This means that you don't have a value for the past year of 1966: this introduces a **nan value** (a **missing value**). This type of border effect is very common in time series analysis, and the best solution, in this case, is to leave the year 1966 out of consideration. You can delete all rows with missing data using Listing 4-5.

```
shifts = shifts.dropna()
```

Listing 4-5 Drop missing data

The next step is to compute the Pearson correlation coefficient using the code that you've seen in Chapter 2. In case you forgot, you can find it in Listing 4-6. And you can see the result in Figure 4-3.

```
shifts.corr()
```

Listing 4-6 Compute a correlation matrix for the shifts dataframe

	this year	past year
this year	1.000000	0.313667
past year	0.313667	1.000000

Figure 4-3 Correlation matrix

In the following paragraph, you will discover how to interpret this autocorrelation.

Positive and Negative Autocorrelation

Just like “regular” correlation, autocorrelation can be positive or negative. Positive autocorrelation means that a high value now will likely give a high value in the next period. This can, for example, be observed in stock trading: as soon as many people want to buy a stock, its price goes up. This positive trend makes people want to buy this stock even more, as it has positive results. The more people buy the stock, the more it goes up, and the more people may want to buy it.

A positive correlation also works on downtrends. If today’s stock value is low, then it is likely that tomorrow’s value will be even lower, as people start selling. When a lot of people sell, the value drops, and even more people will want to sell. This is also a case of positive autocorrelation as the past and the present go in the same direction. If the past is low and the present is low, as well as if the past is high and the present is high.

Negative autocorrelation exists if two trends are opposite. This is the case in the doctor’s consultation duration example. If one consultation takes longer, the next one will take less time. If one consultation takes less time, the doctor may take a bit more time on the next one.

If you have applied the code in Listing 4-6, you have observed a correlation of around **0.31** between the current year and the past year. This correlation is positive rather than negative, as was expected based on the graph. This correlation is of medium strength. It seems that the correlation has captured the trend in the data. If you look at the graph, you can see that there is a trend in the number of earthquakes: a bit higher in the earliest

years, then a bit lower for a certain period, and in the last years, it becomes a bit higher again. This is a problem, as the trend is not the relationship that you want to capture here.

Stationarity and the ADF Test

The problem of having a trend in your data is general in univariate time series modeling. The **stationarity** of a time series means that a time series does not have a (long-term) trend: it is stable around the same average. If not, you say that a time series is **non-stationary**.

AR models can theoretically have a trend coefficient in the model, yet since stationarity is an important concept in the general theory of time series, it is better to learn how to deal with it right away. A lot of models can only work on stationary time series.

A time series that is strongly growing or diminishing over time is obvious to spot. But sometimes it is difficult to tell whether a time series is stationary. This is where the **Augmented Dicky–Fuller test** comes in useful. The Augmented Dicky–Fuller test is a hypothesis test that allows you to test whether a time series is stationary. It is applied as shown in Listing 4-7.

```
from statsmodels.tsa.stattools import adfuller
result = adfuller(earthquakes_per_year.dropna())
print(result)

pvalue = result[1]
if pvalue < 0.05:
    print('stationary')
else:
    print('not stationary')
```

Listing 4-7 Augmented Dicky–Fuller test

In the earthquake data case, you will see a **p-value** that is smaller than **0.05** (the reference value), and this means that the series is theoretically stationary: you don’t have to change anything to apply the AR model.

Note If you're not familiar with hypothesis testing, you should know that the p-value is the conclusive value for a hypothesis test. In a hypothesis test, you try to prove an alternative hypothesis against a null hypothesis. The p-value indicates the probability that you would observe the data that you have observed if the null hypothesis were true. If this probability is low, you conclude that the null hypothesis must be wrong and therefore the alternative must be true. The reference value for the p-value is generally 0.05, but may differ for certain applications.

Differencing a Time Series

Even though the ADF test tells you that the data is stationary, you have seen in the autocorrelation that the autocorrelation is positive where you would expect a negative one. The hypothesis was that this could be caused by a trend.

You now have two opposing indicators. One tells you that the data is stationary, and the other tells you that there is a trend. To be safe, it is better to remove the trend anyway. This can be done by **differencing**.

Differencing means that, rather than modeling the original values of the time series, you model the differences between each value and the next. Even though there is a trend in the actual data, there will probably not be a trend in the differenced data. To confirm, you can do an ADF test again. If it is still not good, you can difference the differenced time series again, until you obtain a correct result.

Listing 4-8 shows you how you can easily difference your data in pandas.

```
# Difference the data
differenced_data =
earthquakes_per_year.diff().dropna()

# Plot the differenced data
ax = differenced_data.plot()
ax.set_ylabel('Differenced number of Earthquakes')
```

Listing 4-8 Differencing in pandas

You can see what the differenced data looks like in Figure 4-4.

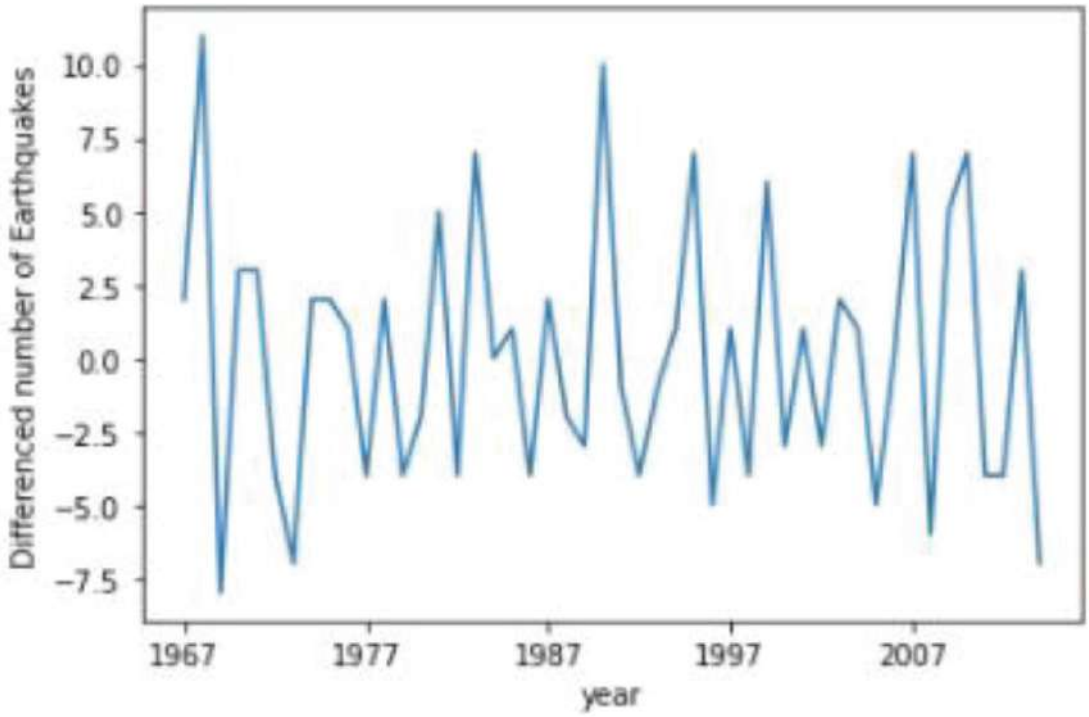


Figure 4-4 Differenced data

Now, to see whether this has switched your autocorrelation to the right direction, you can compute the correlation coefficient as you did before. The code for this can be seen in Listing 4-9. You can see what this shifted and differenced data looks like in Figure 4-5. The correlation matrix (Figure 4-6) shows the correlation coefficient between the shifted differenced data and the non-shifted differenced data.

```
shifts_diff = pd.DataFrame(
    {
        'this year': differenced_data,
        'past year': differenced_data.shift(1)
    }
)

ax = shifts_diff.plot()
ax.set_ylabel('Differenced number of Earthquakes')

shifts_diff.corr()
```

Listing 4-9 Autocorrelation of the differenced data

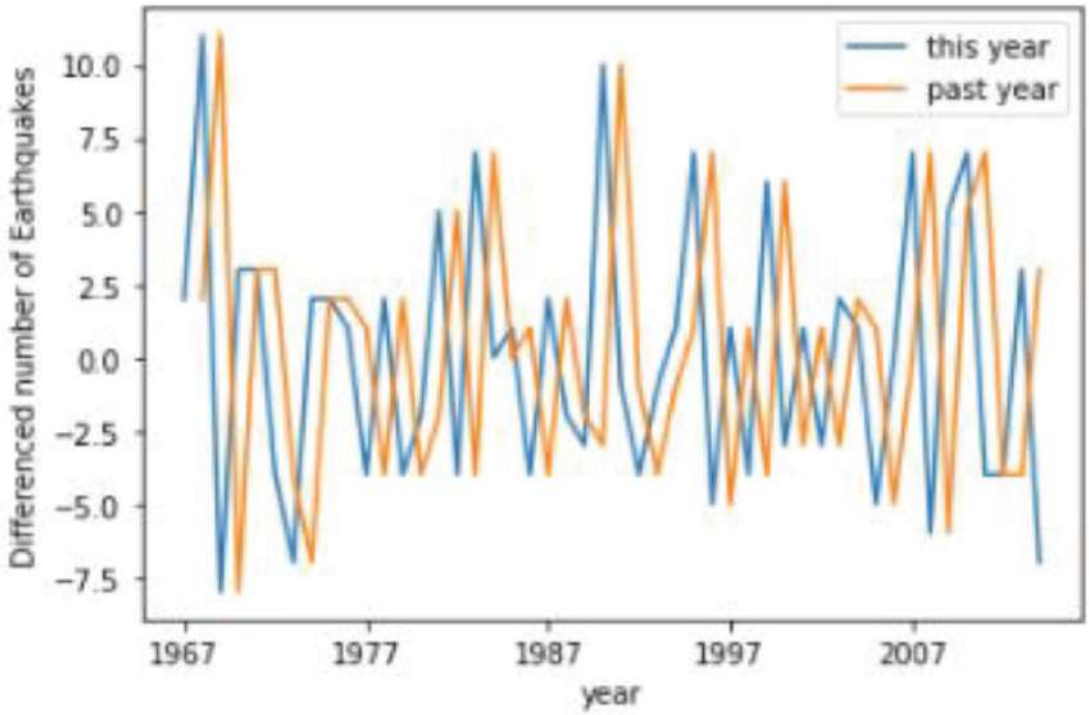


Figure 4-5 Shifted and differenced data

You now obtain a correlation coefficient of **-0.37**, as can be seen in Figure 4-6: you have successfully removed the trend from the data. Your correlation is now the negative correlation that you expected. If there are more earthquakes in one year, then you generally have fewer earthquakes the next year.

	this year	past year
this year	1.000000	-0.376859
past year	-0.376859	1.000000

Figure 4-6 Autocorrelation on differenced data

Lags in Autocorrelation

Besides the direction of the autocorrelation, the notion of **lag** is important. This notion is applicable only in autocorrelation and not in “regular” correlation.

The number of lags of an autocorrelation means the number of steps back in time that have an impact on the present value. In many cases, if autocorrelation is present, it is not just the previous timestep that has an impact, but it can be many timesteps.

Applied to the doctors' example, you could imagine that a doctor has a very long delay in one consultation and that he needs to speed up for the next three consultations. This would mean that the third "speedy" consultation is still impacted by the delayed one. So, the lag of autocorrelation would be at least three: three steps back in time.

The doctor's example is very intuitive, as it is very logical that a doctor has a limited amount of time. In the example of earthquakes per year, there is no such clear logic that explains the relationships between past values and current values. This makes it difficult to intuitively identify the number of lags that should be included.

A great tool to investigate autocorrelation on multiple lags at the same time is to use the autocorrelation plot from the **statsmodels package**. This code is shown in Listing 4-10. **ACF** is short for **Autocorrelation Function**: the autocorrelation as a function of lag.

Note that the choice for 20 lags is arbitrary: it is for plotting purposes only. Twenty lags means 20 years back, and this should be enough to show relevant autocorrelations, but 40 or even more would be acceptable options too.

```
from statsmodels.graphics.tsaplots import plot_acf
import matplotlib.pyplot as plt

plot_acf(differenced_data, lags=20)
plt.show()
```

Listing 4-10 Autocorrelation of the differenced data

You will obtain the plot in Figure 4-7. This plot shows which autocorrelations are important to retain. On the y axis, you see the **correlation coefficient** between the non-lagged data (original data) and the lagged data. The correlation coefficient is between -1 and 1. On the x axis, you see the **number of lags** applied, in this case, from 0 to 20. The first correlation is 1, as this is the data with 0 lag: the original data.

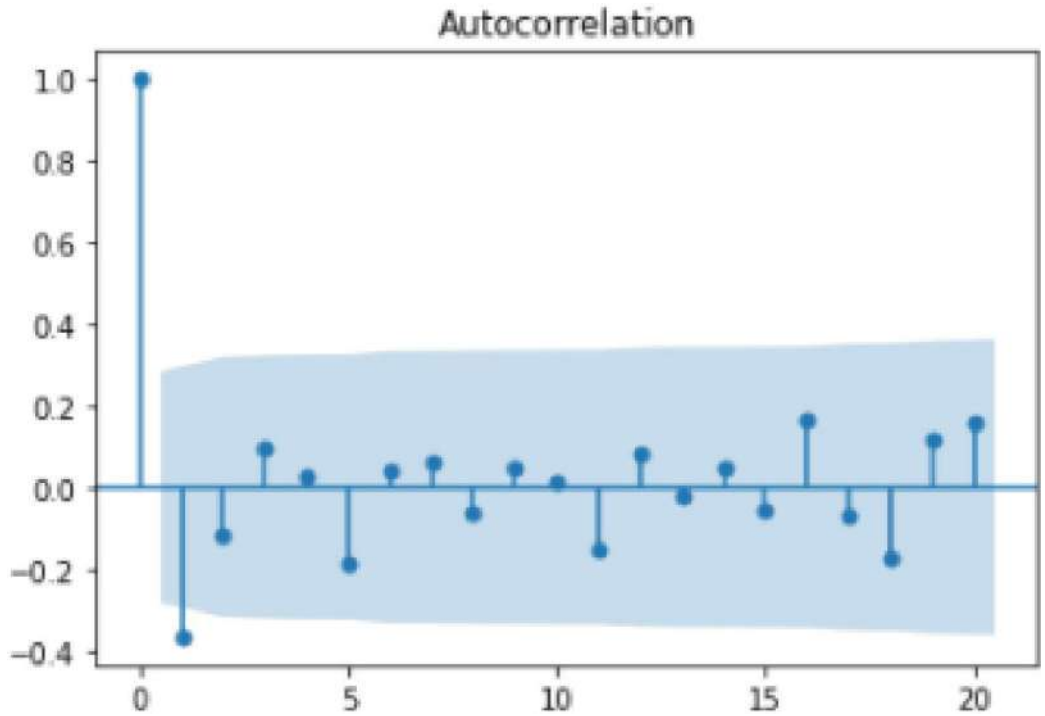


Figure 4-7 Autocorrelation plot

In this plot, you should look for **autocorrelation** values that are a least higher than 0.2 or lower than -0.2. Anything below that can be considered noise. Using 0.3 or 0.4 as a minimum value is also possible: there is no strict guideline in interpreting the correlation coefficient, as in some domains, data are noisier by nature. For example, in social studies, when working with measurements from questionnaire studies, you'll often see much more noise than when working with precise physics measurements.

The blue area in the graph allows you to detect visually which autocorrelations are just noise: if the spike goes outside of the blue area, there is a significant autocorrelation, but if the spike stays within the blue area, the observed autocorrelation is insignificant. You can generally expect the autocorrelation to be higher in lags that are closer to the present and diminish toward further away moments.

In the graph in Figure 4-7, you observe a first spike (at lag 0) which is very strong. This is normal, as it is the original data. The correlation between a variable and itself is always 1 (100%). The second spike is the correlation with lag 1. This is the autocorrelation that you have computed before using the correlation matrix. The correlation coefficient was -0.37, and you see this again in this graph.

The remaining autocorrelations are less strong. This means that to know the number of earthquakes today, we should mainly look at the values of 1 lag back; in the current example, this means the number of earthquakes last year. If there were many earthquakes last year, we can expect fewer earthquakes this year: this is indicated by the negative autocorrelation for lag 1.

For the sake of understanding how to interpret further lags, imagine a different (hypothetical) ACF plot with only one positive spike at lag 5 and the other lags were all 0. This would mean that to understand this year's number of earthquakes, you need to look at what happened five years ago: if there were many earthquakes five years ago, there will be many earthquakes this year. What happened in between is not relevant: since there is no autocorrelation with the previous four years, the number of earthquakes in the previous four years will not help you determine the number of earthquakes this year. This could, for example, be a case of seasonality over five-year periods.

Partial Autocorrelation

The **partial autocorrelation plot** is a second plot that you should also look at. The difference between autocorrelation and partial autocorrelation is that partial autocorrelation makes sure that any correlation is not counted multiple times for multiple lags. Therefore, the partial autocorrelation for each lag is the **additional autocorrelation** to each inferior lag. More mathematically correct is to say that partial autocorrelation is **autocorrelation conditional on earlier lags.**

$$PAC(y_i, y_{i-h}) = \frac{cov(y_i, y_{i-h} | y_{i-1}, \ldots, y_{i-h})}{\sqrt{var(y_i | y_{i-1}, \ldots, y_{i-h+1}) * var(y_{i-h} | y_{i-1}, \ldots, y_{i-h+1})}}$$

This has an added value, as the correlations in the autocorrelation plot may be redundant with one another.

The idea is easier to understand in a hypothetical example. Imagine a hypothetical case in which the values for the present, lag 1, and lag 5 are exactly the same. An autocorrelation would tell you that there is perfect autocorrelation with both lag 1 and lag 5. Partial autocorrelation would tell you that there is perfect autocorrelation with lag 1, but it would not show

autocorrelation for lag 5. As you should always try to make models with the lowest number of variables necessary (often referred to as **Occam's razor** or the **parsimony principle**), it would be better to include only lag 1 and not lag 5 in this case.

You can compute the partial autocorrelation plot using statsmodels, as is shown in Listing 4-11. **PACF** is short for **Partial Autocorrelation Function**: the autocorrelation as a function of lag.

```
from statsmodels.graphics.tsaplots import
plot_pacf
plot_pacf(differenced_data, lags = 20)
plt.show()
```

Listing 4-11 Autocorrelation of the differenced data

This code will give you the plot shown in Figure 4-8. As you can see, based on the blue background shading in the graph, the PACF shows the first and the second lag outside of the shaded area. This means that it would be interesting to also include the second lag in the AR model.

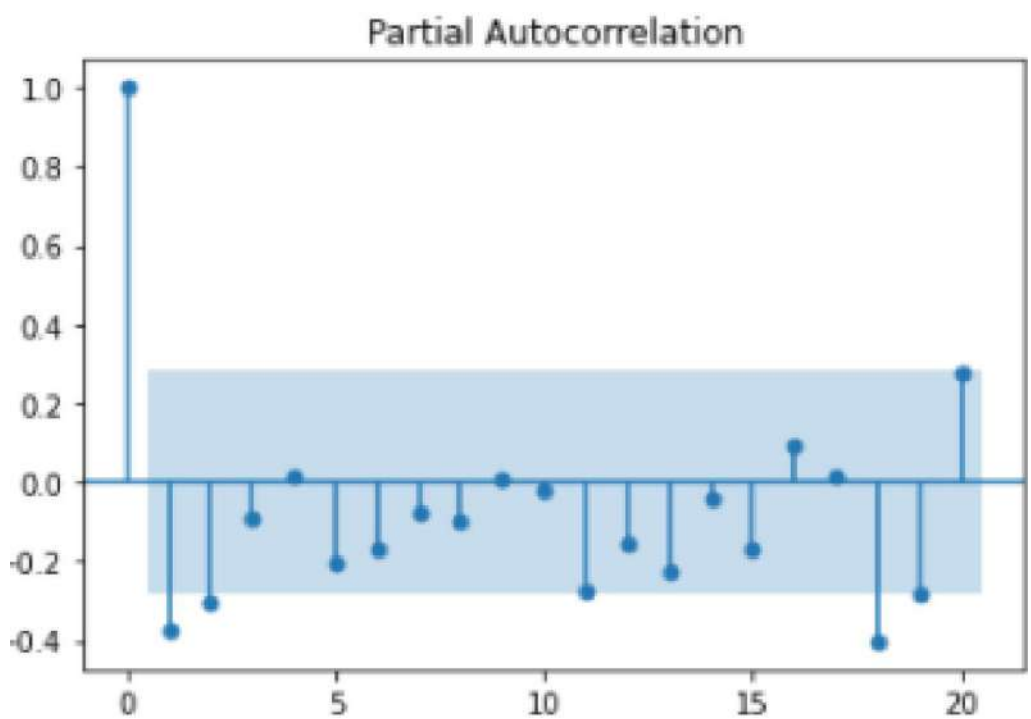


Figure 4-8 Partial autocorrelation

How Many Lags to Include?

Now the big question in time series analysis is always how many lags to include. This is called the **order of the time series**. The notation is **AR(1)** for **order one** and **AR(p)** for **order p**.

The order is up to you to decide. Theoretically speaking, you can base your order on the PACF graph. The theory tells you to take the number of lags before obtaining an autocorrelation of 0. All the other lags should be 0.

In the theory, you often see great graphs where the first spike is very high and the rest is equal to zero. In those cases, the choice is easy: you are working with a very "pure" example of AR(1). Another common case is when your autocorrelation starts high and slowly diminishes to zero. In this case, you should use all of the lags where the PACF is not yet zero.

Yet, in practice, it is not always this simple. Remember the famous saying "all models are wrong, but some are useful." It is very rare to find cases that perfectly fit an AR model. In general, the autoregression process can help to explain part of the variation in a variable, but not all of it.

In the earthquake example, there is still a bit of partial autocorrelation on further lags, and this goes on until very far lags. There is even a spike of PACF on lag 18. This could mean many things: maybe there really is autocorrelation with lags far away. Or maybe a hidden process is underlying that the AR model is not capturing well.

In practice, you will try to select the number of lags that gives your model the **best predictive performance**. Best predictive performance is often not defined by looking at autocorrelation graphs: those graphs give you a theoretical estimate. Yet, predictive performance is best defined by doing **model evaluation and benchmarking**, using the techniques that you have seen in Chapter 2. Later in this chapter, you will see how to use model evaluation to choose a performant order for the AR model. But before getting into that, it is time to go deeper into the exact model definition of the AR model.

AR Model Definition

Until now, you have seen many aspects of the autoregressive model intuitively. As seen before, autocorrelations show for each lag whether there

is a correlation with the present. The AR model uses those correlations to make a predictive model.

But the most important part is still missing: the mathematical definition of the AR model. To make a prediction based on autocorrelation, you need to express the future values of the target variable as a function of the lagged variables. For the AR model, this gives the following model definition:

$$X_t = \sum_{i=1}^{p} \varphi_i X_{t-i} + \varepsilon_t$$

In this formula, X_t is the current value, and it is computed as the sum of each lagged value X_{t-i} multiplied by a coefficient φ_i for this specific lag. The error ε_t at the end is random noise that you cannot predict, but you can estimate.

Estimating the AR Using Yule–Walker Equations

You have now seen **the definition of the AR model**. That is, you have defined its form. However, you can't use it as long as you don't have the values to plug into the formula. The next step is to **fit the model**: you need to find the optimal values for each coefficient to be able to use this. This is also called **estimating the coefficients**.

Once you have values for the parameters, it is relatively easy to use the model: just plug in the lagged values. But as in any Machine Learning theory, the difficult part is always in estimating the model.

In the formula, you see that the past values (X_{t-i}) are multiplied by a coefficient (φ_i). The sum of this will give you the current value X_t. When starting to fit the model, you already know the past values and the current value. The only thing you don't know is the phis (φ_i). The phis are what needs to be found mathematically to define the model. Once you estimate those phis, you will be able to compute tomorrow's value of $X_t$using the estimated phis and the known values of today and before.

For the AR model, different methods have been found to estimate the coefficients (the phis), but the Yule–Walker method is generally accepted as the most appropriate method.

The Yule–Walker Method

The **Yule–Walker method** consists of a set of equations:

$$\gamma_m = \sum_{k=1}^{p} \varphi_k \gamma_{m-k} + \sigma_\varepsilon^2 \delta_{m,0}$$

In this formula, γ_m is the **autocovariance function** of X_t. m is the number of lags from 0 to p (p is the maximum lag). There are therefore p + 1 equations. $\sigma_\varepsilon^2$is the standard deviation of the errors.

$\delta_{m,\,0}$ is the **Kronecker delta function**: it returns 1 if both values are equal (so if m is equal to 0) and 0 if m is not 0. You can see that this means that everything after the + sign is equal to σ_ε^2 if m is 0 and the whole is equal to 0 otherwise. Therefore, there is only one equation that has the part after the +, while the others (from m = 1 to m = p) do not have this part.

The clue to solving this is to start with only the equations from m = 1 to m = p (p equations) written in **matrix format**:

$$\begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \dots \\ \gamma_p \end{pmatrix} = \begin{pmatrix} \gamma_0 & \gamma_{-1} & \gamma_{-2} & \dots \\ \gamma_1 & \gamma_0 & \gamma_{-1} & \dots \\ \gamma_2 & \gamma_1 & \gamma_0 & \dots \\ \dots & \dots & \dots & \dots \\ \gamma_{p-1} & \gamma_{p-2} & \gamma_{p-3} & \dots \end{pmatrix} \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \dots \\ \varphi_p \end{pmatrix}$$

This can be written in **matrix notation** as

$$\gamma = \Gamma^{-1} \widehat{\Phi}$$

To obtain the values for phi, you can use the **Ordinary Least Squares** method. The Ordinary Least Squares method is a matrix computation that is applied widely throughout statistics to solve problems like this one. It allows estimating coefficients in cases where you have a matrix with historical data and a vector with historical outcomes. The solution to finding the best estimates for the phis is the following:

$$\widehat{\Phi} = \left(\Gamma^{\mathrm{T}} \Gamma\right)^{-1} \Gamma^{\mathrm{T}} \gamma$$

You will see the OLS method also in the chapter on **Linear Regression** and other chapters throughout this book.

Once you've applied OLS, you will have obtained the estimates for Phi and you can then compute the sigma by using the estimated phis for solving:

$$\gamma_0 = \sum_{k=1}^{p} \varphi_k \gamma_{m-k} + \sigma_\varepsilon^2$$

Now you'll start fitting the AR model in Python, using Listing 4-12. Fitting the model in this case means to identify the best possible values for the phis. As you're working with real data, and not with a simulated sample, note that you'll be working toward the best fit possible of the AR model throughout this example.

```
from statsmodels.regression.linear_model import
yule_walker
coefficients, sigma =
yule_walker(differenced_data, order = 3)
print('coefficients: ', -coefficients)
print('sigma: ', sigma)
```

Listing 4-12 Estimate Yule–Walker AR coefficients with order 3

The **coefficients** that you obtain with this are the coefficients for each of the lagged variables. In this case, the order is 3, so that will need to be applied for the last three values to compute the new value. You can make a forecast by computing the next **steps**, based on the current coefficients. The code for this is shown in Listing 4-13. Attention: the more steps forward you want to forecast, the more difficult it becomes: your error will likely increase with the number of steps.

```
coefficients, sigma =
yule_walker(differenced_data, order = 3)

# Make a list of differenced values
val_list = list(differenced_data)
# Reverse the list so that the order corresponds
with the order of the coefficients
val_list.reverse()
```

```
# Define the number of years to predict
n_steps = 10

# For each year to predict
for i in range(n_steps):

    # Compute the new value as the sum of lagged
values multiplied by their corresponding
coefficient
    new_val = 0
    for j in range(len(coefficients)):

        new_val += coefficients[j] * val_list[j]

    # Insert the new value at the beginning of the
list
    val_list.insert(0, new_val)

# Redo the reverse to have the order of time
val_list.reverse()

# Add the original first value back into the list
and do a cumulative sum to undo the differencing
val_list = [earthquakes_per_year[0]] + val_list
new_val_list = pd.Series(val_list).cumsum()

# Plot the newly obtained list
plt.plot(range(1966, 2025), new_val_list)
plt.ylabel('Number of earthquakes')
plt.show()
```

Listing 4-13 Make a forecast with the AR coefficients

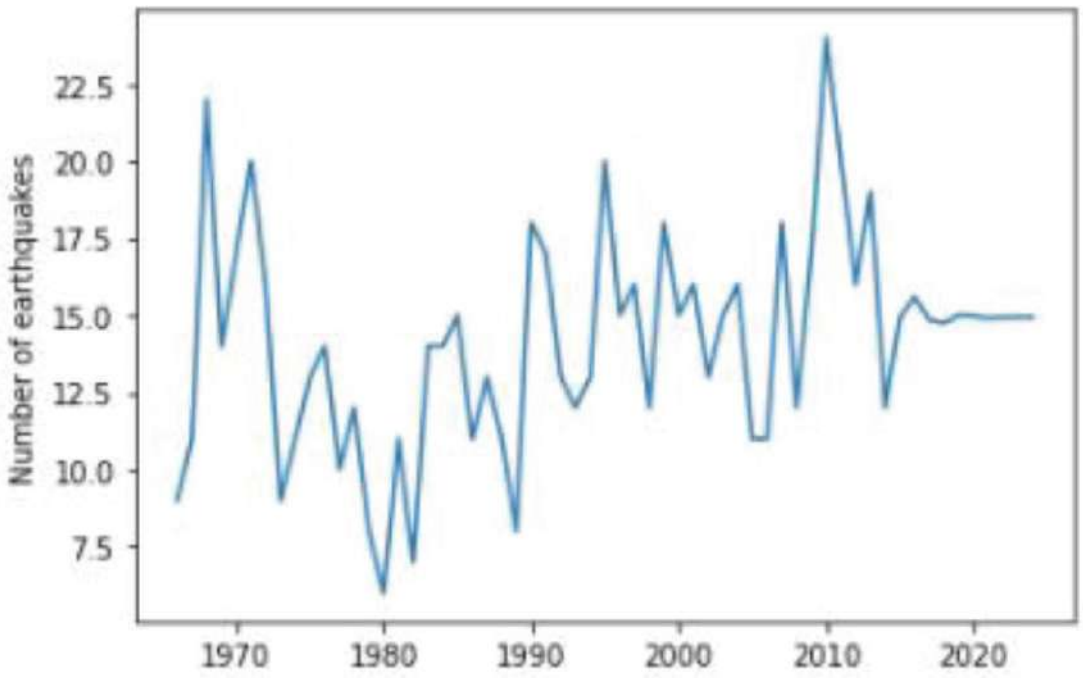


Figure 4-9 The AR(3) model has not captured anything: flat line forecast

You will obtain the plot in Figure 4-9. The first values are actual data, and the last ten steps are forecasted data. Unfortunately, you can see that the model has not captured anything interesting: it is just a flat line! But you haven't finished yet; let's try with an order of 20 this time and see whether anything improves.

Just change the order in the first line and you will obtain the plot in Figure 4-10. It obtains a very different forecast than the one in Figure 4-9. Your model seems to be learning something much more interesting when you add an order 20: it has captured some variation.

Although this seems to fit well with the data visually, you still need to verify whether the model was correct: Did it forecast something close to reality? To do this, you will use some strategies and metrics from Chapter 2 and apply them to this forecast.

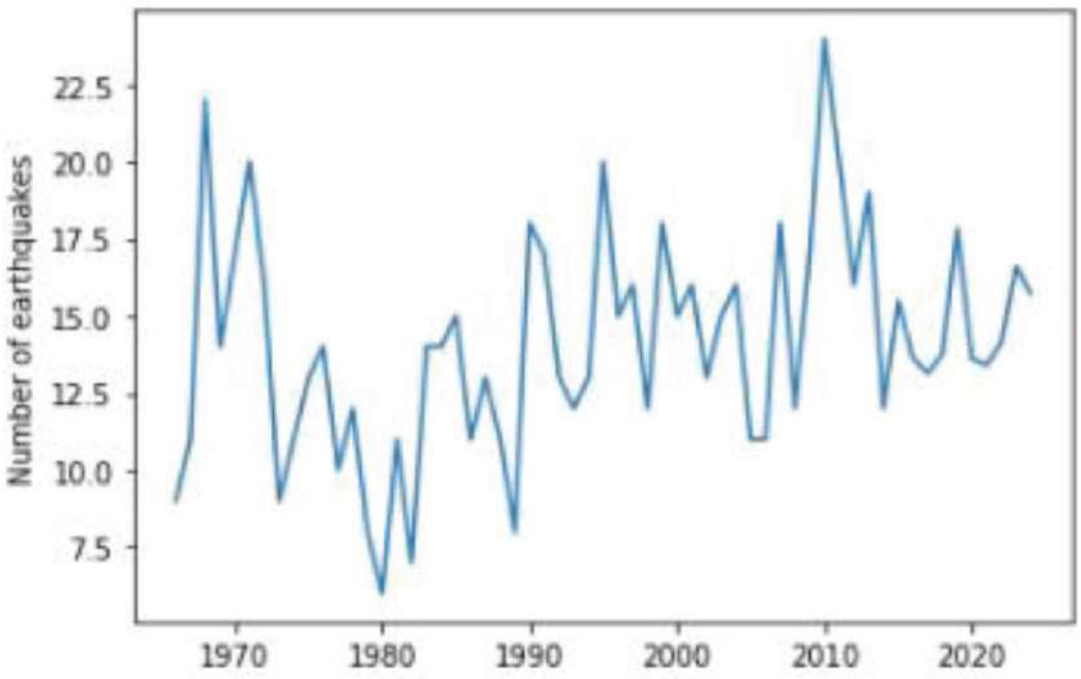


Figure 4-10 The forecast with order 20 has captured something, but is it correct?

Train, Test, Evaluation, and Tuning

In this case, let's apply a train-test split. You will cut off the last ten years of the dataset and use them as a test set. This allows you to fit your model using the rest of the data (the train set). Once you have fitted the train set, you can compute the error between the test set and your prediction. The code in Listing 4-14 does just that.

```
from sklearn.metrics import r2_score

train = list(differenced_data)[:-10]
test = list(earthquakes_per_year)[-10:]

coefficients, sigma = yule_walker(train, order =
3)

# Make a list of differenced values
val_list = list(train)
# Reverse the list so that the order corresponds
with the order of the coefficients
```

```
val_list.reverse()
# Define the number of years to predict
n_steps = 10

# For each year to predict
for i in range(n_steps):

    # Compute the new value as the sum of lagged
values multiplied by their corresponding
coefficient
    new_val = 0
    for j in range(len(coefficients)):

        new_val += coefficients[j] * val_list[j]

    # Insert the new value at the beginning of the
list
    val_list.insert(0, new_val)

# Redo the reverso to have the order of time
val_list.reverse()

# Add the original first value back into the list
and do a cumulative sum to undo the differencing
val_list = [list(earthquakes_per_year)[0]] +
val_list
new_val_list = pd.Series(val_list).cumsum()

# Plot the newly obtained list
validation = pd.DataFrame({
    'original':
earthquakes_per_year.reset_index(drop=True),
    'pred': new_val_list })

print('Test R2:', r2_score(validation.iloc[-10:,
0], validation.iloc[-10:, 1]))

# Plot the newly obtained list
```

```
plt.plot(range(1966, 2015), validation)
plt.legend(validation.columns)
plt.ylabel('Number of earthquakes')
plt.show()
```

Listing 4-14 Fit the model on a train set and evaluate it on a test set

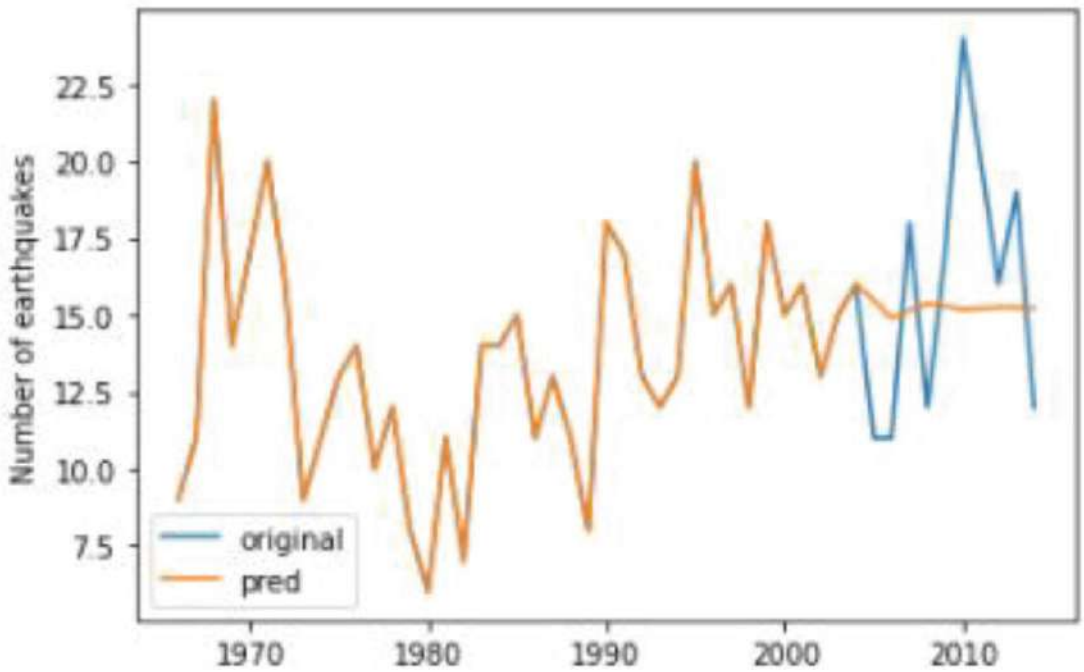


Figure 4-11 Evaluation plot of the AR model with order 3

Unfortunately, the model with order 3 is quite bad. So bad that the R2 is negative (**-0.04**). You can see from the graph (Figure 4-11) that the model suffers from **underfitting**: it did not capture anything and predicts only an average value into the future. This confirms what you've seen before with order 3.

In the previous example, you have applied the order 20 to see whether that gives a better result. Yet, the choice for order 20 is quite random. To find an order that works well, you can apply a **grid search**.

A grid search consists of doing a model evaluation for each value of a **hyperparameter**. Hyperparameters are parameters that are not estimated by the model but chosen by the modeler. The order of the model is an example of this. Other models often have more hyperparameters, which makes choosing hyperparameters a nontrivial decision.

A grid search for one parameter is very simple to code: you simply fit the model for each possible value for the hyperparameter, and you select the best-performing one. You can use the example in Listing 4-15.

```
def evaluate(order):
    train = list(differenced_data)[:-10]
    test = list(earthquakes_per_year)[-10:]

    coefficients, sigma = yule_walker(train, order
= order)

    # Make a list of differenced values
    val_list = list(train)
    # Reverse the list to corresponds with the
order of coefs
    val_list.reverse()
    # Define the number of years to predict
    n_steps = 10

    # For each year to predict
    for i in range(n_steps):

        # Compute the new value
        new_val = 0
        for j in range(len(coefficients)):
            new_val += coefficients[j] *
val_list[j]

        # Insert the new value at the beginning of
the list
        val_list.insert(0, new_val)

    # Redo the reverse to have the order of time
    val_list.reverse()

    # Undo the differencing with a cumsum
    val_list = [list(earthquakes_per_year)[0]] +
val_list
```

```
    new_val_list = pd.Series(val_list).cumsum()

    # Plot the newly obtained list
    validation = pd.DataFrame({
        'original':
earthquakes_per_year.reset_index(drop=True),
        'pred': new_val_list })

    return r2_score(validation.iloc[-10:, 0],
validation.iloc[-10:, 1])

# For each order between 1 and 30, fit and
evaluate the model
orders = []
r2scores = []
for order in range(1, 31):
    orders.append(order)
    r2scores.append(evaluate(order))

# Create a results data frame
results =pd.DataFrame({'orders': orders,
                       'scores': r2scores})

# Show the order with best R2 score
results[results['scores'] == results.max()
['scores']]10:, 0], validation.iloc[-10:, 1])
```

Listing 4-15 Apply a grid search to find the order that gives the best R2 score on the test data

This gives the highest R2 score for order 19, with an R2 score of **0.13**. This means that the AR(19) model explains about 13% of the variation in the test data.

You can reuse the code in Listing 4-13 to recreate the graph with the forecast using order 19. Just change the order in the line that fits the Yule–Walker coefficients. The graph is shown in Figure 4-12.

This code will also make a plot for the model with order 19 and observe that, although not perfect, the model indeed fits a bit better than the model with order 3.

The AR model is one of the basic building blocks of univariate time series. It can be used as a **standalone** model only in very rare and specific cases. On this real-life example of predicting earthquakes, the R2 score of 0.13 is not very good: you can surely do much better by including other building blocks of univariate time series. You will discover these building blocks throughout the next chapters.

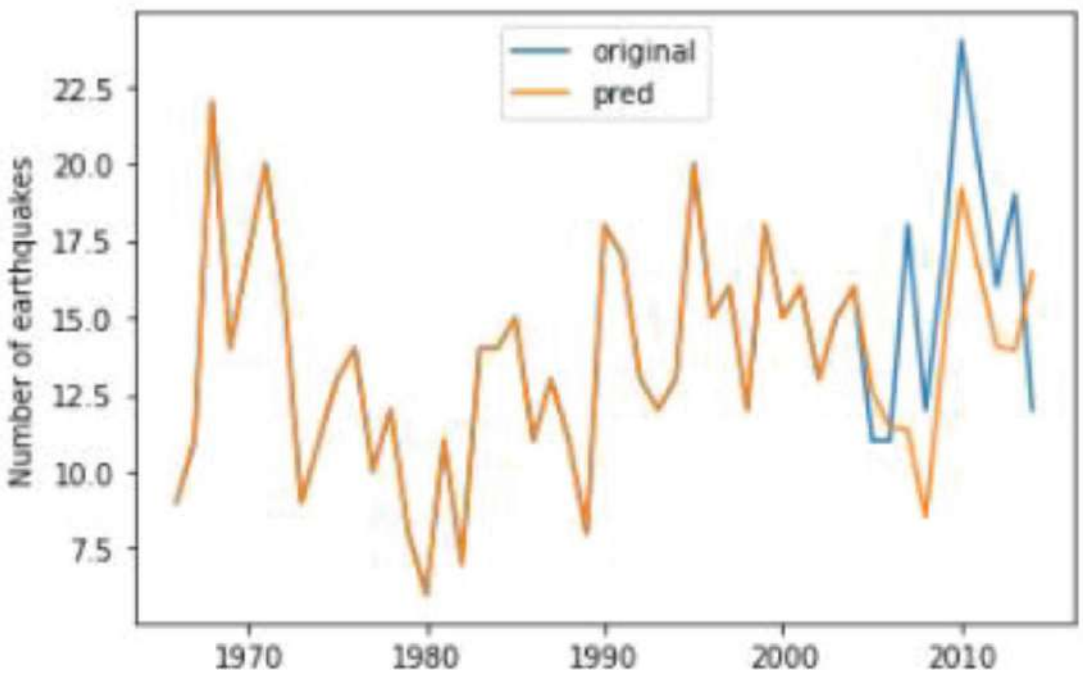


Figure 4-12 Forecast on the test set for the AR model with order 19

Saving the Final Model Using MLflow

As a last step for this chapter, let's see how you can save this model using MLflow. As you have seen in Chapter 3, MLflow allows you to save models and go back to them easily.

However, unfortunately, MLflow cannot perform autologging on the yule_walker estimation function that has been used in this chapter. So we'll need to use another approach here.

Listing 4-16 shows you how to use the statsmodels AutoReg object to fit the same model that we have created above. We have used the yule_walker functions for studying the earthquake autoregression model and finding out the right way to calibrate it. However, for saving the model to disk, it's easier to use this AutoReg object.

```
from statsmodels.tsa.ar_model import AutoReg
import mlflow

# Start mlflow autologging
mlflow.autolog()

# Create the autoregression with order 19 and fit
it
ar = AutoReg(list(earthquakes_per_year), 19,
old_names=False)
res = ar.fit()

# Show the results
print(res.summary())
```

Listing 4-16 Fit an AutoReg with MLflow logging

When running the code in Listing 4-16, you'll see that a new directory is created on your local file system. Inside that directory, you'll find the artifacts directory, in which you can find the model. If you want to load it later, you can load the model into the Python session with MLflow.

You will also see that a metrics directory appeared. As was explained in Chapter 3, inside this directory, you can retrieve all autologged model KPIs. It is worth noting that the model metrics that are autologged by MLflow on AutoReg are not based on a train-test split.

In Listing 4-15, you have seen how to build a train test split that allows you to obtain an estimate of r2_score outside of the sample. Despite it not being a part of the autologging metrics, it is an important metric to store. In Listing 4-17, you can see how to log this as a custom metric to MLflow.

```
# End the previous run
mlflow.end_run()

# Start a new run
with mlflow.start_run():

    # Start mlflow autologging because you still
want to log the model
    mlflow.autolog()
```

```
    # Create the autoregression with order 19 and
fit it
    ar = AutoReg(list(earthquakes_per_year), 19,
old_names=False)
    res = ar.fit()

    # Show the results
    print(res.summary())

    # Get the r2 score on the train set (as
calculated in listing 4-15)
    r2_test = float(results[results['scores'] ==
results.max()['scores']].iloc[0]['scores'])

    # Log this metric to mlflow
    mlflow.log_metric('r2_score_test', r2_test)
```

Listing 4-17 Log a custom metric to MLflow

Key Takeaways

- The AR model predicts the future of a variable by leveraging correlations between a variable's past and present values.
- Autocorrelation is the correlation between a time series and its previous values.
- Partial autocorrelation is autocorrelation and is conditional on earlier lags: it prevents double-counting correlations.
- The number of lags to include in the AR model can be based on theory (ACF and PACF plots) or can be determined by grid search.
- Grid search consists of doing a model evaluation for each value of the hyperparameters of a model. This is an optimization method for the choice of hyperparameters.
- Hyperparameters are different from model coefficients. Model coefficients are optimized by a model, while hyperparameters are chosen by the modeler. Yet, the model can use optimization techniques like grid search to find the best hyperparameters.
- Yule–Walker equations are used to fit the AR model. Fitting the model means finding the coefficients of the model.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_5

5. The MA Model

Joos Korstanje[1]
(1) Maisons Alfort, France

The MA model, short for the **Moving Average model**, is the second important building block in univariate time series (see Table 5-1). Like the AR model, it is a building block that is more often used as a part of more complex time series, but it can also be used as a standalone.

Table 5-1 The building blocks of univariate time series models

Name	Explanation	Chapter
AR	Autoregression	4
MA	Moving Average	5
ARMA	Combination of AR and MA models	6
ARIMA	Adding differencing (i) to the ARMA model	7
SARIMA	Adding seasonality (S) to the ARIMA model	8
SARIMAX	Adding external variables (X) to the SARIMA model *(note that external variables mean that it is not univariate anymore)*	9

Throughout this chapter, you will see the MA model applied to a forecasting example. Please note that, as in the previous chapter, the model is optimized as **a standalone model.** When applying univariate time series in practice, you will generally use the SARIMA or SARIMAX model directly: after all, the MA model is a component of this model. Yet, there is added value in seeing the MA model applied separately, as it is important to understand each building block of the SARIMAX model separately.

The Model Definition

The mathematical model underlying the MA model is quite like the AR model in form, but very different in intuitive understanding. There, where the AR model looks at lagged values of the target variable to predict the future, the MA model looks at the **model error of past predictions**. The model definition is as follows:

$$X_t = \mu + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \ldots + \theta_q \varepsilon_{t-q}$$

In this equation, **Xt** are the current values (unlagged), μ (mu) is the average of the series, ε (epsilon) is the error of the model, and the θ (thetas) are the coefficients to multiply with each of the past values.

As you can see, the model has only past errors in it and no past values. This means that to predict the future, you don't look at what happened in the past but only at how wrong you were in the past!

This may seem strange, but it has some logic in it. Imagine a case in which you try to make a prediction based on the past. At some points, the process differs strongly from what you expected, and you do not understand why. Let's say you had a huge, unexpected, and temporary drop in stock prices at some point.

Both an AR model and an MA model would be able to take this drop into account over time, but in a different way. The AR model would model it as a past low value. The MA model would model it as a large error, which can be interpreted as a large impulse. In this case, it makes sense to see the error as the impulse rather than the value itself, because the fact that the impulse was unexpected is what is going to be a driver for the stock prices in the next days, as stock traders will be reacting to this unexpected impulse.

In this way, *the model error in the past influences the future, and past errors can help you to make a forecast*. This is the idea behind the MA model.

Note The term Moving Average is also used for taking the average of the most recent values. In time series, this is often used for smoothing a time series. This has nothing to do with the MA model.

Fitting the MA Model

Fitting the MA model is more complicated than the AR model. From a high level, the MA model seems relatively comparable to an AR model: MA depends on past errors, whereas the AR model depends on past values.

Yet, there is a big difference here. When fitting the MA model, you do not yet have the past errors of the model. Since you have not yet defined the model, you cannot estimate the error of the model. And you need the error of the model to estimate the coefficients.

There is some sort of circular dependency here, which makes it impossible to find a straightforward method to fit the model at once. Yet, different techniques can be used for fitting an MA model, of which the most common is the Nonlinear Least Squares. It assumes that the error at time 0 is negligible. The thetas can then be estimated using the following formula:

$$\hat{\theta}_{nls} = \underset{\theta}{\operatorname{argmin}} \sum_{t=2}^{T} \left(x_t - \theta \sum_{k=0}^{t-2} (-\theta)^k \varphi_{t-1-k} \right)^2$$

Since this formula uses an **argmin**, it is a numerical search for the optimal values for theta. This means that fitting the model is simply done by iteratively searching through different values for theta and identifying which values for theta lead to the lowest value for the rest of the formula, i.e., the lowest of the following:

$$\sum_{t=2}^{T} \left(x_t - \theta \sum_{k=0}^{t-2} (-\theta)^k \varphi_{t-1-k} \right)^2.$$

An example of such a minimization can be found in Figure 5-1. The figure shows a curve of error, which depends on the parameters. The goal is to find the parameters that give you the lowest model error. This is not an easy task, as you generally have no clue what those parameters are.

I won’t go into depth on the optimization methods that can be used for this minimization task, as that would be out of scope for this book. But you should know that there are many algorithms that do such optimization tasks.

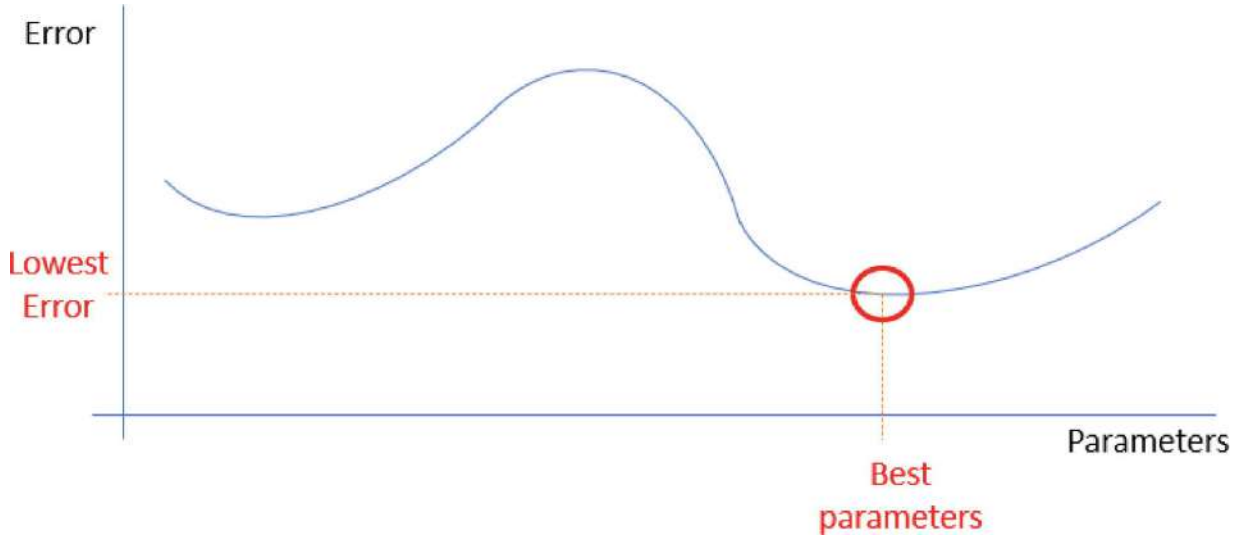


Figure 5-1 Parameters that minimize an error

Stationarity

Important to note is that the MA model has an absolute need for stationarity. Where the AR model can adapt to non-stationary situations, the MA model cannot. The Augmented Dicky–Fuller test and differencing are notions that have been covered in Chapter 3 and can be used here as well.

Choosing Between an AR and an MA Model

Just like the AR model, the lag of the MA model can be decided by looking at the PACF plot (the partial autocorrelation function). But there is more use to the PACF curve than just selecting the best theoretical lag.

The PACF plot can also help you choose the correct type of model. For instance, you have now seen the AR and the MA model, but you have no idea which one to use. The type of autocorrelation can be a factor in this decision.

The following are indicators for the type of model to use:

- Autoregression (typical PACF plot in Figure 5-2)
  - Autocorrelation decays to 0.
  - Autocorrelation decays exponentially.
  - Autocorrelation alternates between positive and negative and decays to 0.

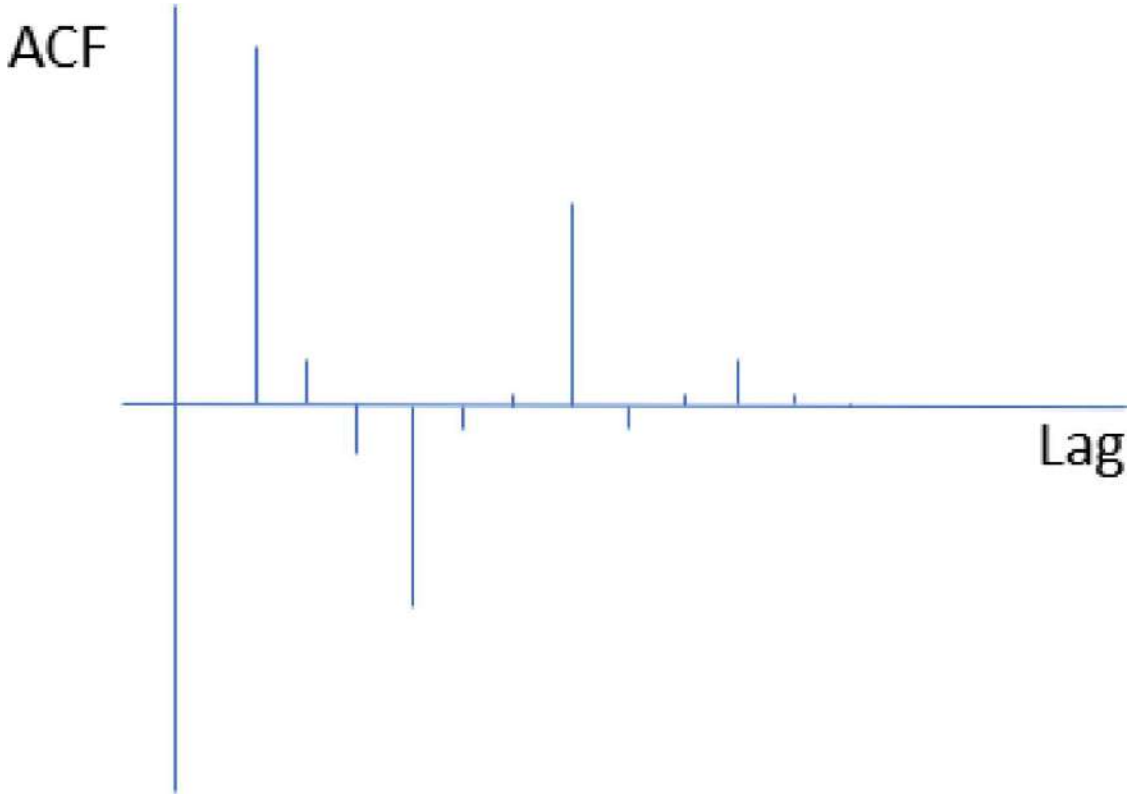


Figure 5-2 Typical AR model PACF

- Moving Average (typical ACF plot in Figure 5-3)
  - Many values of (almost) zero and a few spikes

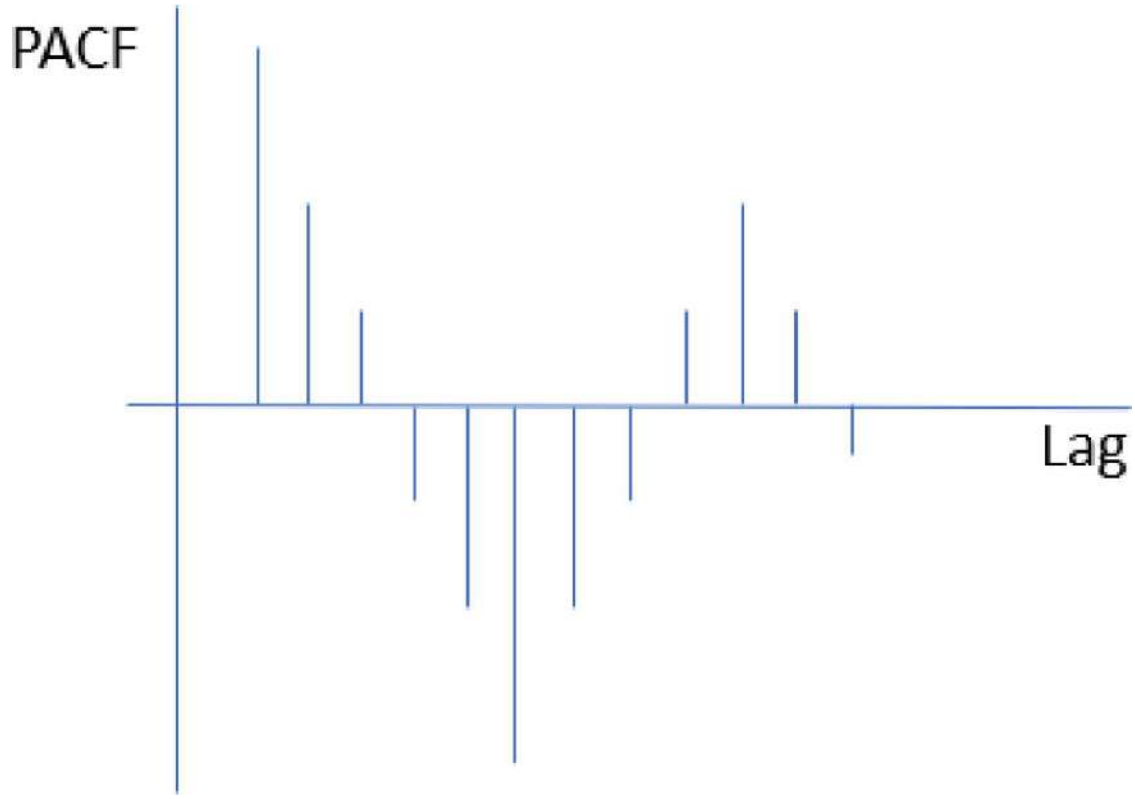


Figure 5-3 Typical MA model ACF

Application of the MA Model

Let's now apply this to an example. The data that you will use for this example are Microsoft's stock closing prices. You can get that data through the yfinance Python package, as shown in code Listing 5-1.

> **Note** The yfinance package is a Python package that allows downloading stock data.

```
import yfinance

data =
yfinance.Ticker('MSFT').history(start='2019-01-
01', end='2019-12-31')
data = data['Close']
```

Listing 5-1 Importing stock price data using the yfinance package

As you know, it is important to get a first idea of the data, so let's make a plot of the closing prices. This is done in Listing 5-2.

```
import matplotlib.pyplot as plt
ax = data.plot()
ax.set_ylabel("Stock Price")
plt.show()
```

Listing 5-2 Plotting the stock price data

This will output the graph in Figure 5-4.

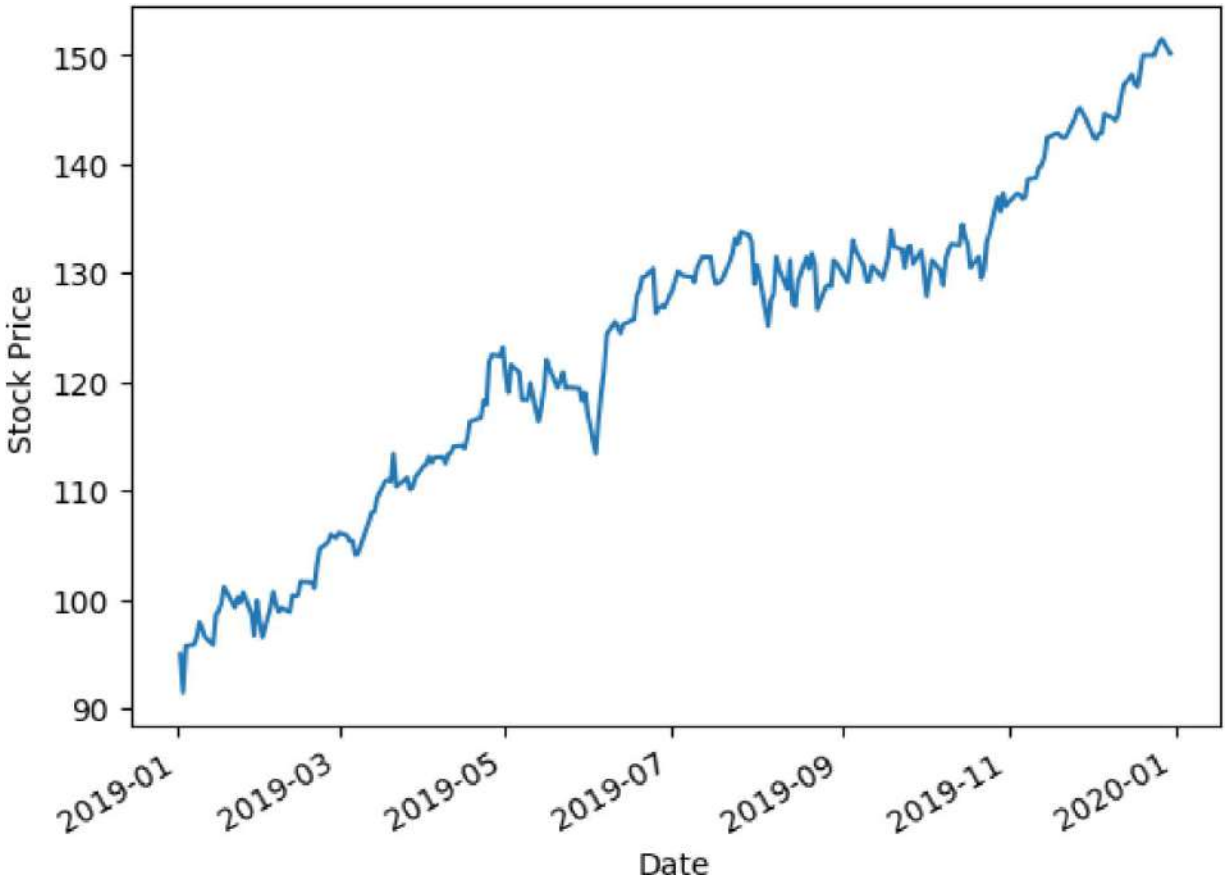


Figure 5-4 Plot of the original data

In this graph, you can clearly see an upward trend. It is not even necessary to apply an Augmented Dicky–Fuller (ADF) test here to see that this data is **not stationary**. Since the MA model cannot function without stationarity, let's apply the go-to solution: differencing. This is done in Listing 5-3 and will show the plot in Figure 5-5.

```
# Need to difference
data = data.diff().dropna()
ax = data.plot()
ax.set_ylabel("Daily Difference in Stock Price")
plt.show()
```

Listing 5-3 Computing the differenced data and plotting it

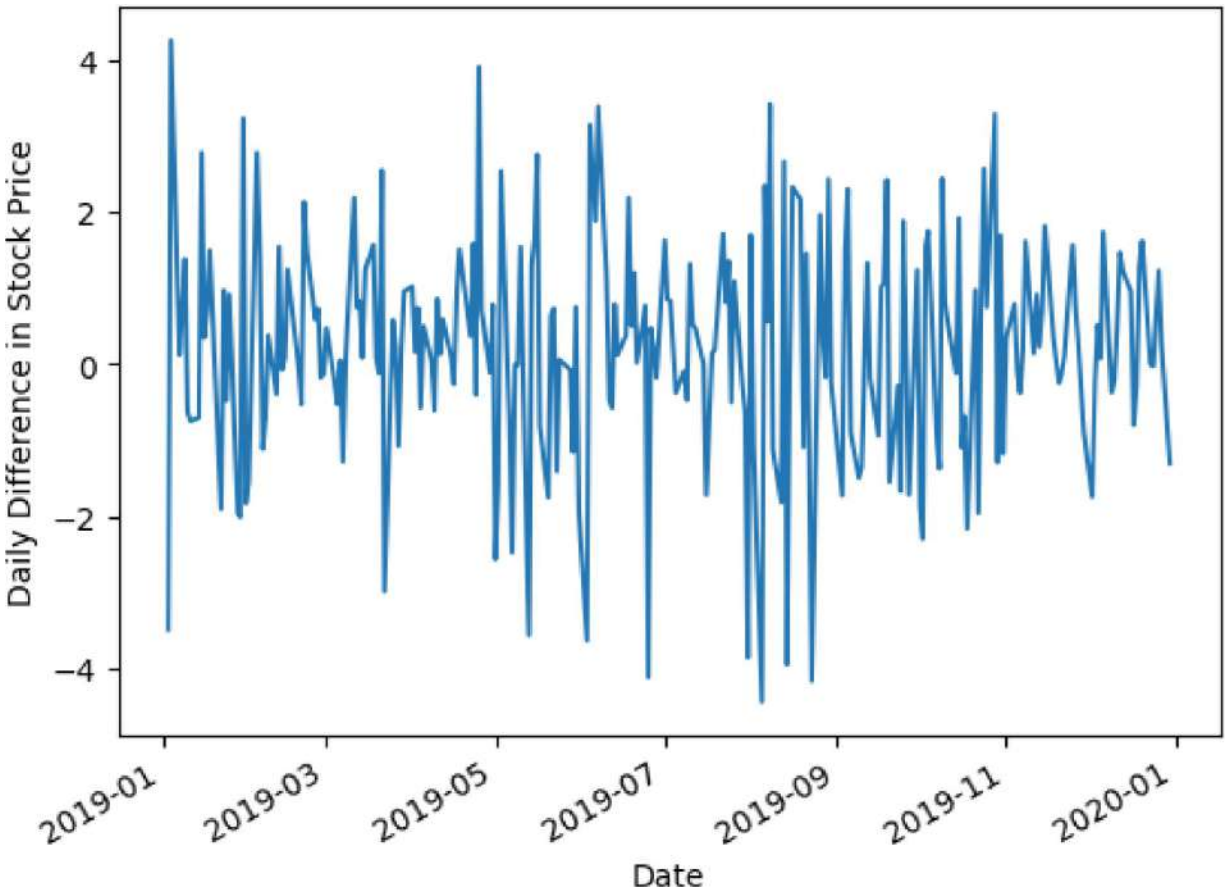


Figure 5-5 Plot of the differenced data

This differenced data seems stationary. For consistency's sake, Listing 5-4 shows how to use an Augmented Dickey–Fuller test on these differences, although by visual inspection, there is really no doubt about it: the data is clearly varying around a fixed mean of 0.

```
from statsmodels.tsa.stattools import adfuller
result = adfuller(data)
pvalue = result[1]
if pvalue < 0.05:
```

```
    print('stationary')
else:
    print('not stationary')
```

Listing 5-4 Applying an ADF test to the differenced data

This confirms stationarity of the differenced series. Let's have a look at the autocorrelation and partial autocorrelation functions to see whether there is an obvious choice for the lag using Listing 5-5 to obtain the ACF in Figure 5-6 and the PACF in Figure 5-7.

```
from statsmodels.graphics.tsaplots import
plot_acf, plot_pacf
import matplotlib.pyplot as plt
plot_acf(data, lags=20)
plot_pacf(data, lags=20)
plt.show()
```

Listing 5-5 Plotting the autocorrelation function and the partial autocorrelation function

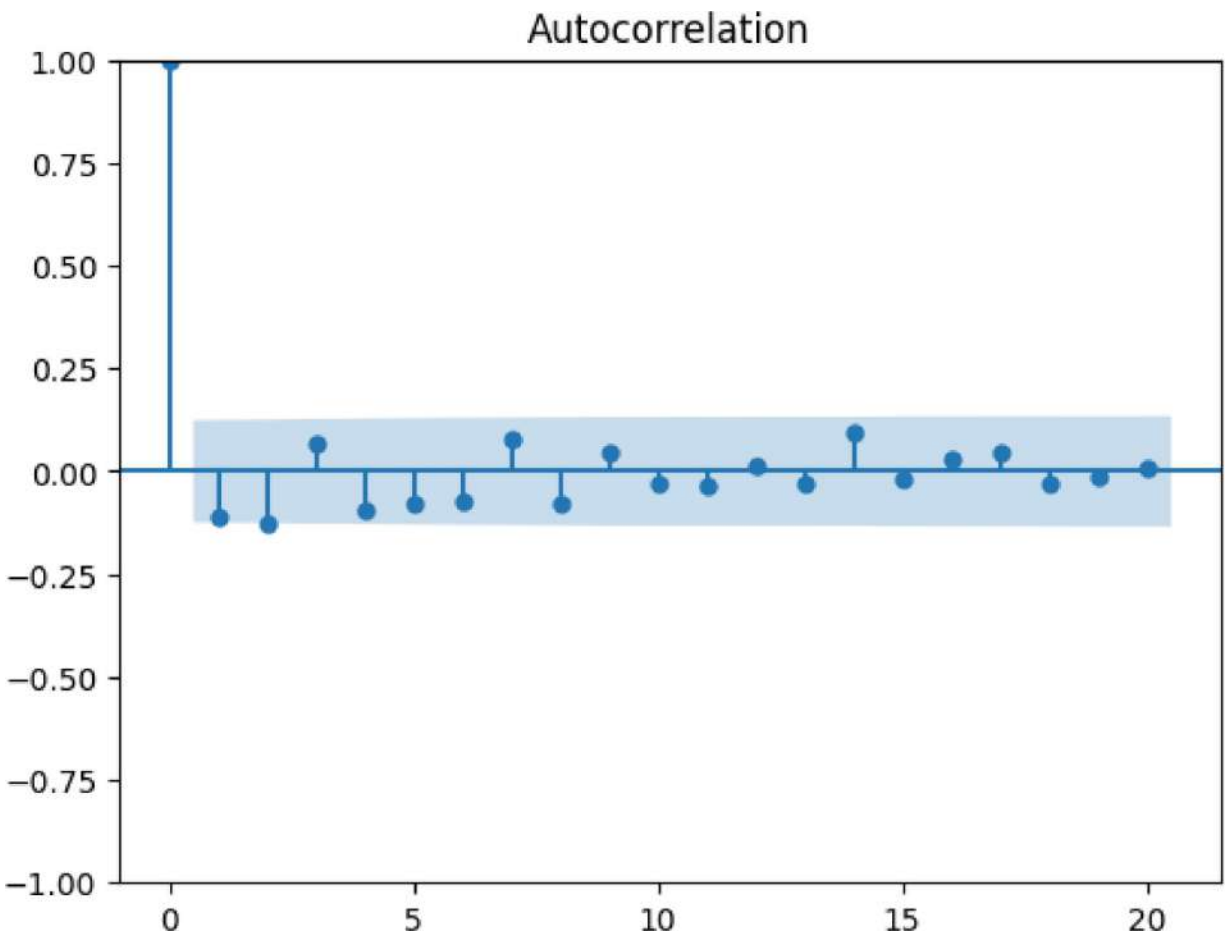


Figure 5-6 Autocorrelation function

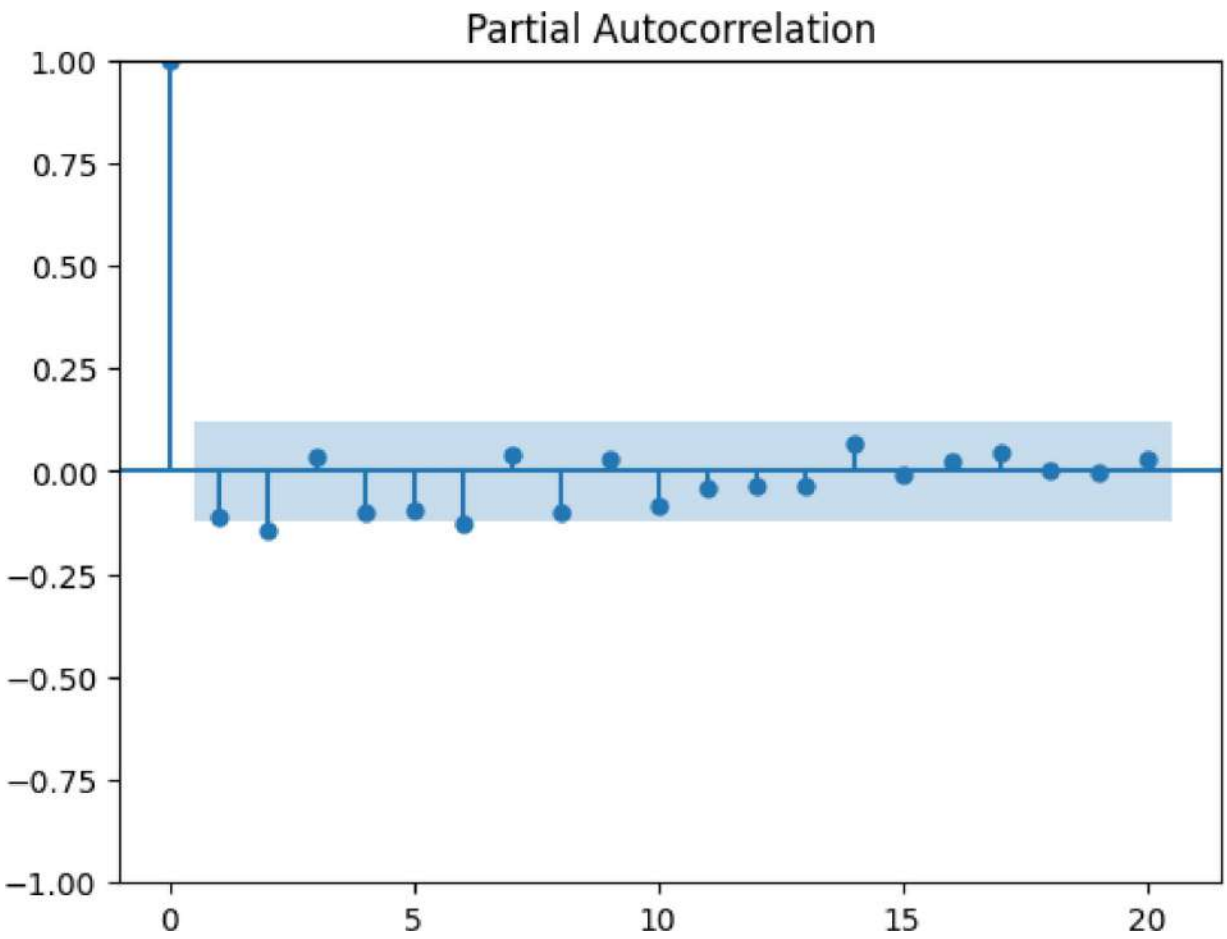


Figure 5-7 Partial autocorrelation function

Unfortunately, there is no very clear pattern in the autocorrelation or partial autocorrelation that would confirm a choice for the order. You can observe some decay in the partial autocorrelation function, which is a positive sign for using a time series approach. But nothing too obvious on the choice of lag. Let's fit the MA model with order 1 and see what it does.

To fit the MA model in Python, you need to use the ARIMA function from statsmodels. ARIMA is a model that you will see in the next chapter. It is a model that contains multiple building blocks of univariate time series, including AR for the AR model and MA for the MA model. You can specify the order for each of the "smaller" models separately, so by setting everything except MA to 0, you obtain the MA model. This can be done using Listing 5-6.

```
from sklearn.metrics import r2_score
from statsmodels.tsa.arima.model import ARIMA
```

```
# Forecast the first MA(1) model
mod = ARIMA(data.diff().dropna(), order=(0,0,1))
res = mod.fit()

orig_data = data.diff().dropna()
pred = res.predict()

plt.plot(orig_data)
plt.plot(pred)
plt.show()

print(r2_score(orig_data, pred))
```

Listing 5-6 Fitting the MA model and plotting the forecast

This gives you a plot of the fit (Figure 5-8) and an R2 score on the full period.

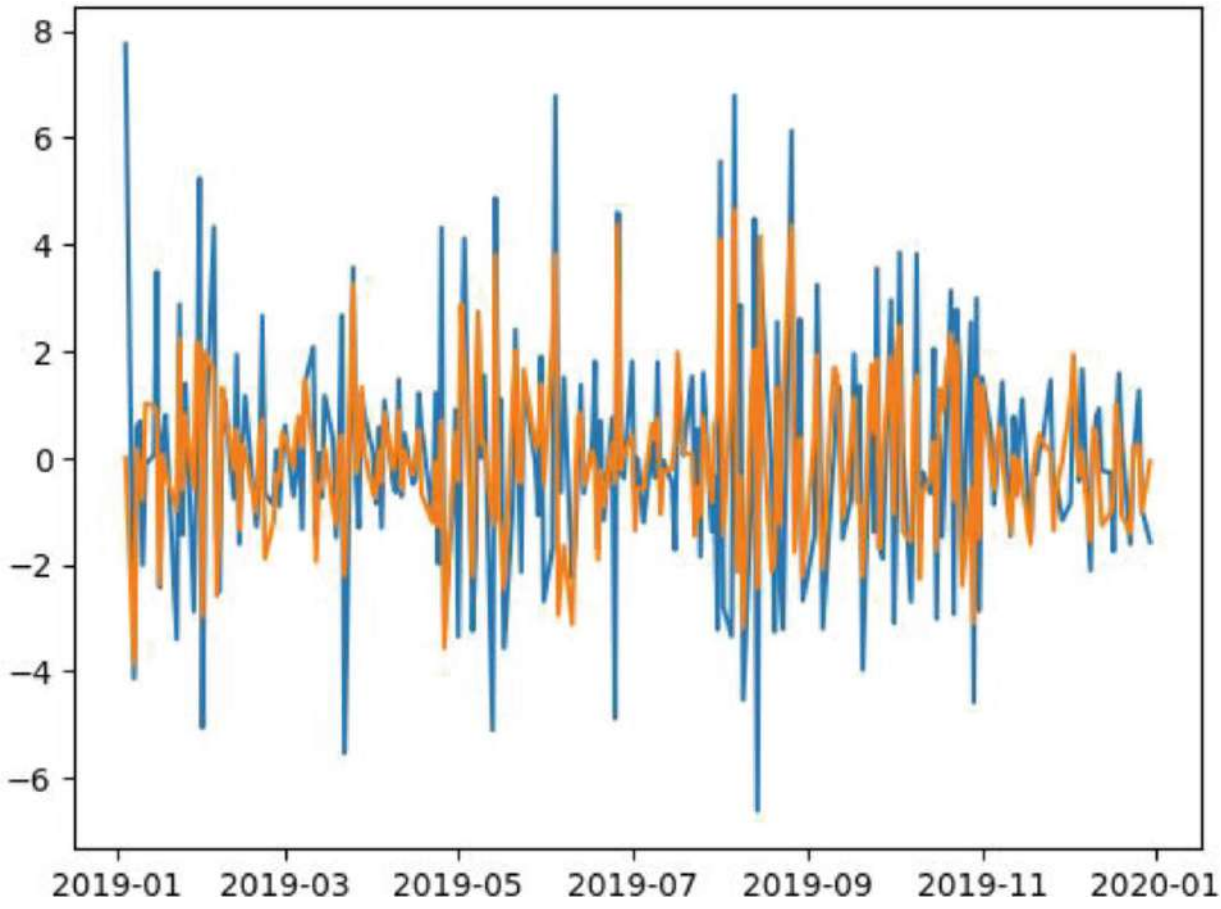


Figure 5-8 Actuals vs. forecast on train dataset

This plot does not look too bad: the orange curve (predicted) follows the general trend in the blue curve (actual values). You can observe that the predicted values are less extreme in their predictions (the highs are less high, and the lows are less low). The R2 score is **0.51**.

As an interpretation, you could observe that there is an underfit in this model: it does capture the basics, but it does not capture enough of the trend to be a very good model. At this stage, you can already have a check of its out-of-sample performance by creating a train and a test set. Listing 5-7 shows how this can be done.

```
train = data.diff().dropna()[0:240]
test = data.diff().dropna()[240:250]

# Import the ARMA module from statsmodels
from statsmodels.tsa.arima.model import ARIMA

# Forecast the first MA(1) model
mod = ARIMA(train, order=(0,0,1))
res = mod.fit()

orig_data = data.diff().dropna()
pred = res.predict()
fcst = res.forecast(steps = len(test))

print(r2_score(train, pred))
print(r2_score(test, fcst))
```

Listing 5-7 Fitting the MA model on train data and evaluating the R2 score on train and test data

This should give you a train R2 of **0.51** and a test R2 of **0.13**. This performance on the training data is not good, and the performance on the test data is even worse (only 13% of the variation in the test data can be explained by the MA model). A visual of the test data actuals vs. the forecast should confirm this bad fit (using Listing 5-8 to obtain Figure 5-9).

```
plt.plot(list(test))
plt.plot(list(fcst))
plt.legend(['Actual Prices', 'Predicted Prices'])
```

```
plt.show()
```

Listing 5-8 Plotting the out-of-sample forecast of the MA(1) model (MA with order 1)

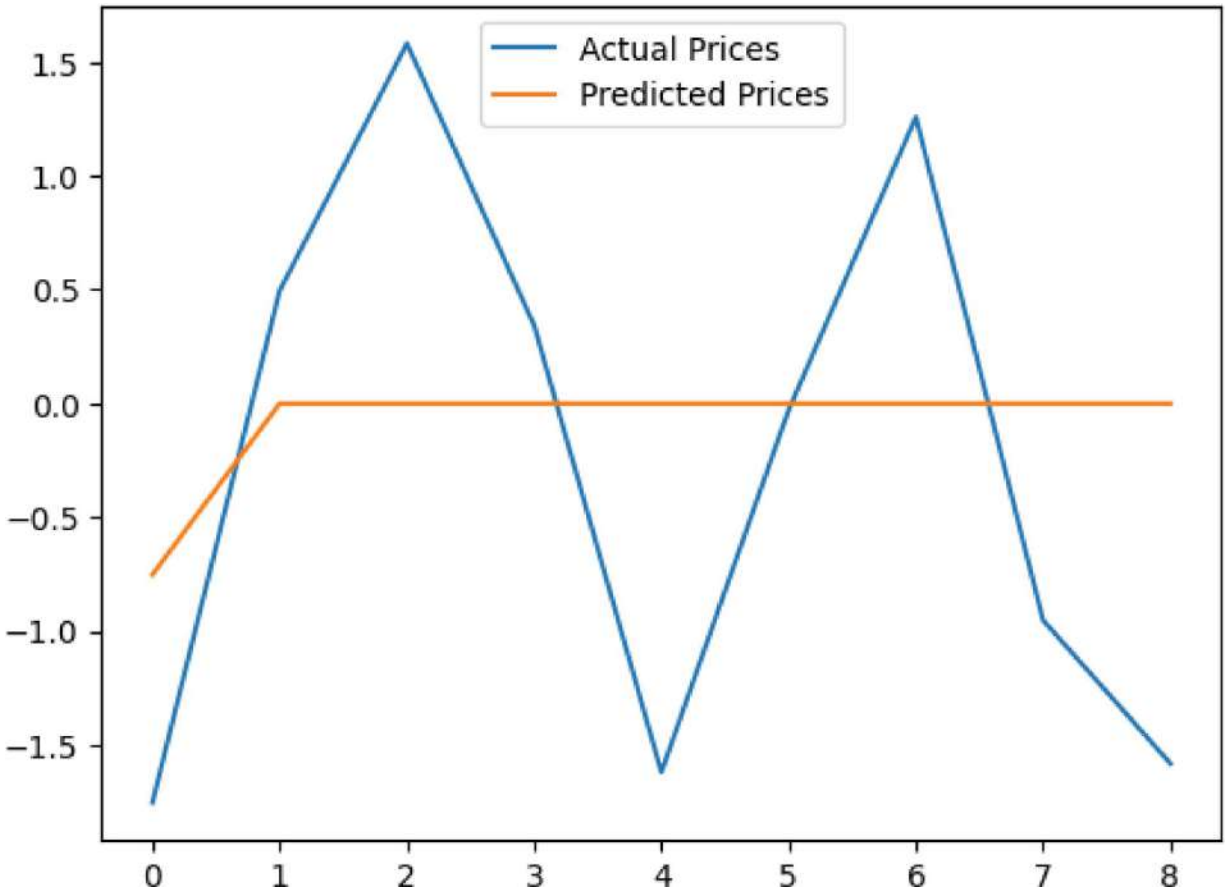


Figure 5-9 Out-of-sample performance of the MA(1) model

But when looking at this visual, you can see that there is something weird going on with the forecast on the test set. The MA(1) forecast has forecasted the average for every value except for the first step into the future.

And there is a mathematical reason for this: the MA(q) model uses the model error of the past q steps to predict the future. When predicting one step into the future, there is no problem: the model has the actual values and the fitted model of each time step before the prediction. But when doing a forecast for a second time step, the model does not know the actual values for time t+1, while this is an input that is needed for forecasting t+2.

Multistep Forecasting with Model Retraining

In the previous chapter, the forecast was made for ten steps forward. Yet, no attention has yet been paid to an essential difference in forecasting. This is the difference between one-step forecasting and multistep forecasting.

One-step forecasting is the basis. It is relatively easy to predict one step forward. Multistep forecasting is not an easy task using time series, as errors will accumulate with every step forward you take.

In many cases, the most accurate solution is to retrain the model at each time step. In fact, it is possible to do this if you update the model as soon as you obtain a new data point. This means that you would do repeated one-step forecasts. In some cases, this can work, while in other cases, it is necessary to forecast further in the future.

It is important to consider the duration of your forecast in model evaluation. If you do multistep forecasts, you should do the evaluation on a multistep train-test split. If you do one-step forecasts, you should do the evaluation on a one-step train-test split.

If you do one-step forecasts, you will find yourself in lack of a test dataset: you cannot compute a prediction error on a test dataset of one data point only. A solution that you can use is iteratively fitting the one-step model and constructing a multistep forecast based on multiple one-step forecasts. This can be done using the code in Listing 5-9.

```
import pandas as pd

train = list(data.diff().dropna()[0:240])
test = list(data.diff().dropna()[240:250])

orig_data = data.diff().dropna()

# Import the ARMA module from statsmodels
from statsmodels.tsa.arima.model import ARIMA
fcst = []
for step in range(len(test)):
    # Forecast the first MA(1) model
    mod = ARIMA(train, order=(0,0,1))
    res = mod.fit()
    pred = res.predict()
    fcst += list(res.forecast(steps = 1))
    train.append(test[step])
```

```
print(r2_score(test, fcst))
plt.plot(test)
plt.plot(fcst)
plt.legend(['Actual Prices', 'Predicted Prices'])
plt.show()
```

Listing 5-9 Estimating the error of the MA(1) model for ten refitted one-step forecasts

The R2 score that you obtain using this method is **0.38**: not perfect, but this starts to look like something useful. The graph of the forecasted values vs. the actual values is shown in Figure 5-10.

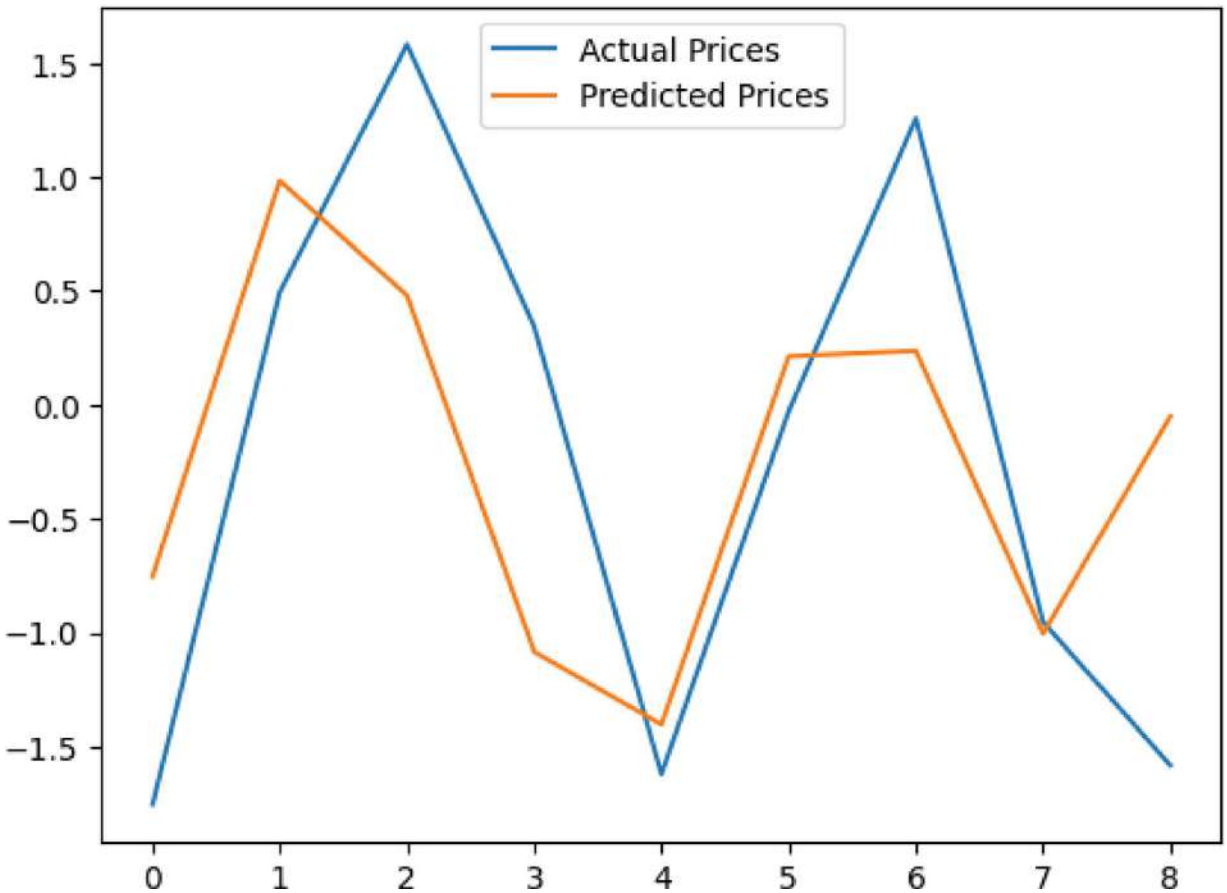


Figure 5-10 Forecasting one-step multiple times yields a relatively acceptable forecast

Of course, the error estimate that you obtain this way is only the error estimate for the forecast of the next time step, and this will not be valid for a longer period. How to use this in practice depends on your specific use case.

For this specific example, you can consider that if you can perfectly predict the stock market for the next day, this is already quite the accomplishment: let's retain the one-step error in this case.

Grid Search to Find the Best MA Order

As a next step, let's try to do a grid search to find the order for the MA model that obtains the lowest error on the test set. Remember the grid search from the previous chapter: the basic version of the grid search simply consists of looping through each possible value of the order and evaluating the model on the test data.

There are other and better automated ways of optimizing hyperparameters: you will see them throughout the next chapters, so make sure that you understand the idea behind this basic version now. Now, let's find out whether there is a specific order for the MA that delivers us better predictive performance using Listing 5-10.

```
def evaluate2(order):
    train = list(data.diff().dropna()[0:240])
    test = list(data.diff().dropna()[240:250])

    orig_data = data.diff().dropna()

    fcst = []
    for step in range(len(test)):
        # Forecast the first MA(1) model
        mod = ARIMA(train, order=(0,0,order))
        res = mod.fit()

        pred = res.predict()
        fcst += list(res.forecast(steps = 1))
        train.append(test[step])

    return r2_score(test, fcst)

scores = []
for i in range(1, 21):
    print(i)
```

```
    scores.append((i, evaluate2(i)))

# observe best order is 3 with R2 of 0.49
scores = pd.DataFrame(scores)
print(scores[scores[1] == scores.max()[1]])
```

Listing 5-10 Grid search to obtain the MA order that optimizes forecasting R2

This will give you the best order of 3 with an R2 out of sample of **0.49**. You can have a look at the final forecast that this yields using the code in Listing 5-11. The plot that you will obtain is shown in Figure 5-11.

```
train = list(data.diff().dropna()[0:240])
test = list(data.diff().dropna()[240:250])

orig_data = data.diff().dropna()

fcst = []
for step in range(len(test)):
    # Forecast the first MA(1) model
    mod = ARIMA(train, order=(0,0,3))
    res = mod.fit()

    pred = res.predict()

    fcst += list(res.forecast(steps = 1))

    train.append(test[step])

print(r2_score(test, fcst))

plt.plot(test)
plt.plot(fcst)
plt.legend(['Actual Prices', 'Forecasted Prices'])
plt.show()
```

Listing 5-11 Obtaining the final forecast

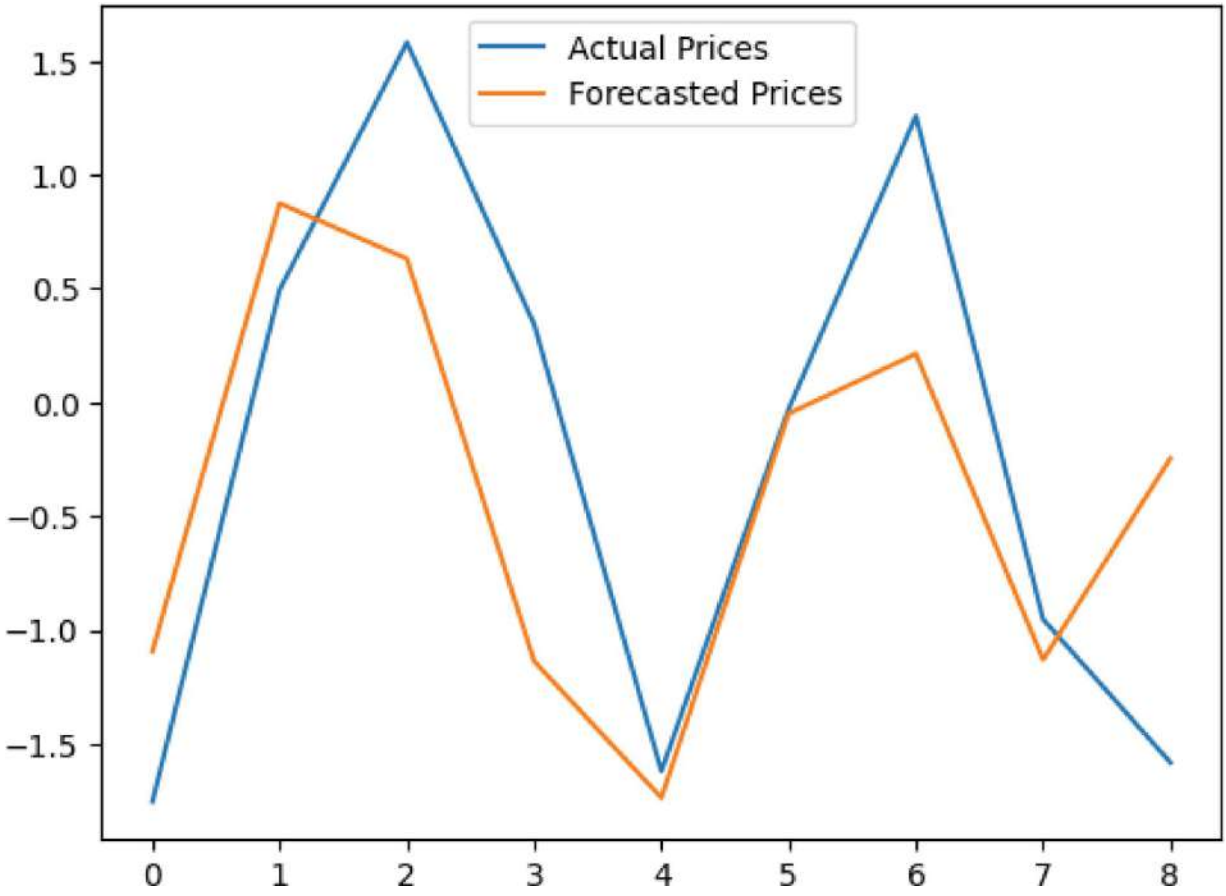


Figure 5-11 The final result: the MA(3) forecast

Saving This Model in MLflow

As a final step, let's save this model using MLflow. As in the previous chapters, we have computed an R2 score based on a test set. This is not one of the metrics that can be used in MLflow autologging. You can use the code in Listing 5-12 to autolog the model and log the test R2 score as an additional metric.

```
import mlflow

# Start a new run
with mlflow.start_run():

    # Start mlflow autologging to log the model
    mlflow.autolog()
```

```
train = list(data.diff().dropna()[0:240])
test = list(data.diff().dropna()[240:250])

orig_data = data.diff().dropna()

fcst = []
for step in range(len(test)):
    # Forecast the MA(3) model
    mod = ARIMA(train, order=(0,0,3))
    res = mod.fit()

    pred = res.predict()

    fcst += list(res.forecast(steps = 1))

    train.append(test[step])

r2 = r2_score(test, fcst)

# Log this metric to mlflow
mlflow.log_metric('r2_score_test', r2)
```

Listing 5-12 Obtaining the final forecast

Key Takeaways

- The Moving Average model predicts the future based on impulses in the past.
  - Those impulses are measured as model errors.
  - The idea behind this is that unexpected impacts can actually have a large impact on the future.
  - There is no one-shot computation for the MA model coefficients, so it takes a bit longer to estimate this model compared to the AR model.
  - The MA model is the second building block of the SARIMA model. The AR and MA models together form the ARMA model, the topic of the next chapter.
- Multistep predictions are predictions of multiple time steps into the future. This is difficult with MA models, especially when the order is

low. A solution can sometimes be to do multiple one-step predictions and retrain the model every time you receive new data.

- The autocorrelation function and the partial autocorrelation function can help you decide whether you are looking at an AR or an MA forecast. AR forecasts see their autocorrelation exponentially decay to 0 and may alternate between positive and negative. MA autocorrelation functions are characterized by many values of almost zero and a few spikes.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_6

6. The ARMA Model

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, you will discover the ARMA model. It is a combination of the AR model and the MA model, which you have seen in the two previous chapters. Since the theory of this chapter should already be relatively familiar to you, this chapter will also introduce a few additional model quality indicators that are valid for the ARMA model but can also be used for the AR and MA models separately.

The Idea Behind the ARMA Model

The ARMA model is a simple combination of the AR and MA models. The idea behind combining the AR and MA models into one model is that the models together are more performant than one model. The development over time of one variable can follow multiple processes at the same time.

Let's do a short recap of the AR and MA models separately. The AR model tries to predict the future by **multiplying past values by a coefficient**. This AR process is based on the presence of autocorrelation: the target variable's present value is correlated to a past value. The MA model does not use past values of the target value to predict its future but rather uses the **error of past predictions** as an impulse for forecasted values.

The ARMA model is nothing more than a model that allows both these processes to be in place at the same time:

- Part of the future of a variable is explained by past values (the AR effect).

- Part of it is explained by past errors (the MA effect).

The fact that both are combined into one model makes it simply easier to use: you can imagine that working with two separate models would be cumbersome.

The Mathematical Definition of the ARMA Model

The ARMA model is defined by the following equation:

$$X_t = c + \varepsilon_t + \sum_{i=1}^{p} \varphi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i}$$

In this equation, you clearly observe the AR part and the MA part. The AR part uses coefficients φ_i (phi) multiplied by past observations X_{t-i} for a **number of lags** defined as **p**, also called **the AR order**. The MA part uses coefficients θ_i (theta), multiplied by past errors ε_{t-i} (epsilon) for a **number of lags q: the MA order**.

Those p and q are the same hyperparameters that you have encountered before. For model notation, you use *AR(p) for an AR model of order p* and *MA(q) for an MA model of order q*. In the ARMA model, p and q remain the same. This gives you the following notation: **ARMA(p, q)**.

You should note here that there is no dependency between p and q, as there is no dependency between the AR and MA definitions in the equation. This means that an ARMA(p, q) model can take any order for p and q. Special cases are p = 0, which means you have an MA model, and when q = 0, you have an AR model.

As the p and q are hyperparameters, they are parameters that should be chosen by the modeler, while the c, φ, and θ are coefficients of the model and will therefore be estimated by the model.

An Example: Predicting Births Using ARMA

Now, let's get to work on an example and see whether the combined power of the AR and the MA models can deliver a good forecast. You will be working with the Births dataset, a dataset that contains very long historical observations of the number of births in a village. You can get the data into Python using Listing 6-1.

```
import pandas as pd

# Read the csv file
data = pd.read_csv('births_data.csv', sep=';')

# Keep useful columns
data = data[['Date', 'Births']]
```

Listing 6-1 Getting the Births data into Python

The Births dataset has a specific pattern of seasonality that you will discover in this chapter. The dataset is not very detailed: it just contains the data counts per month. Since there is a lot of data to work with, let's set the goal to do a forecast with yearly data. To do this, the data needs to be aggregated to yearly data. You can use Listing 6-2 to do this and you'll obtain the result show in Figure 6-1.

```
data['year'] = data.Date.apply(lambda x: x[-4:])
data = data[['Births',
'year']].groupby('year').sum()
data.head()
```

Listing 6-2 Aggregating the Births data to yearly data

	Births
year	
1749	80220
1750	78460
1751	27540
1752	27681
1753	11336

Figure 6-1 An extract of the Births data

Let's also make a plot over time to see what type of variation you are working with. You can do this using Listing 6-3. Using this code, you should obtain the graph in Figure 6-2.

```
import matplotlib.pyplot as plt
ax = data.plot()
ax.set_ylabel('Births')
plt.show()
```

Listing 6-3 Plotting the yearly Births data

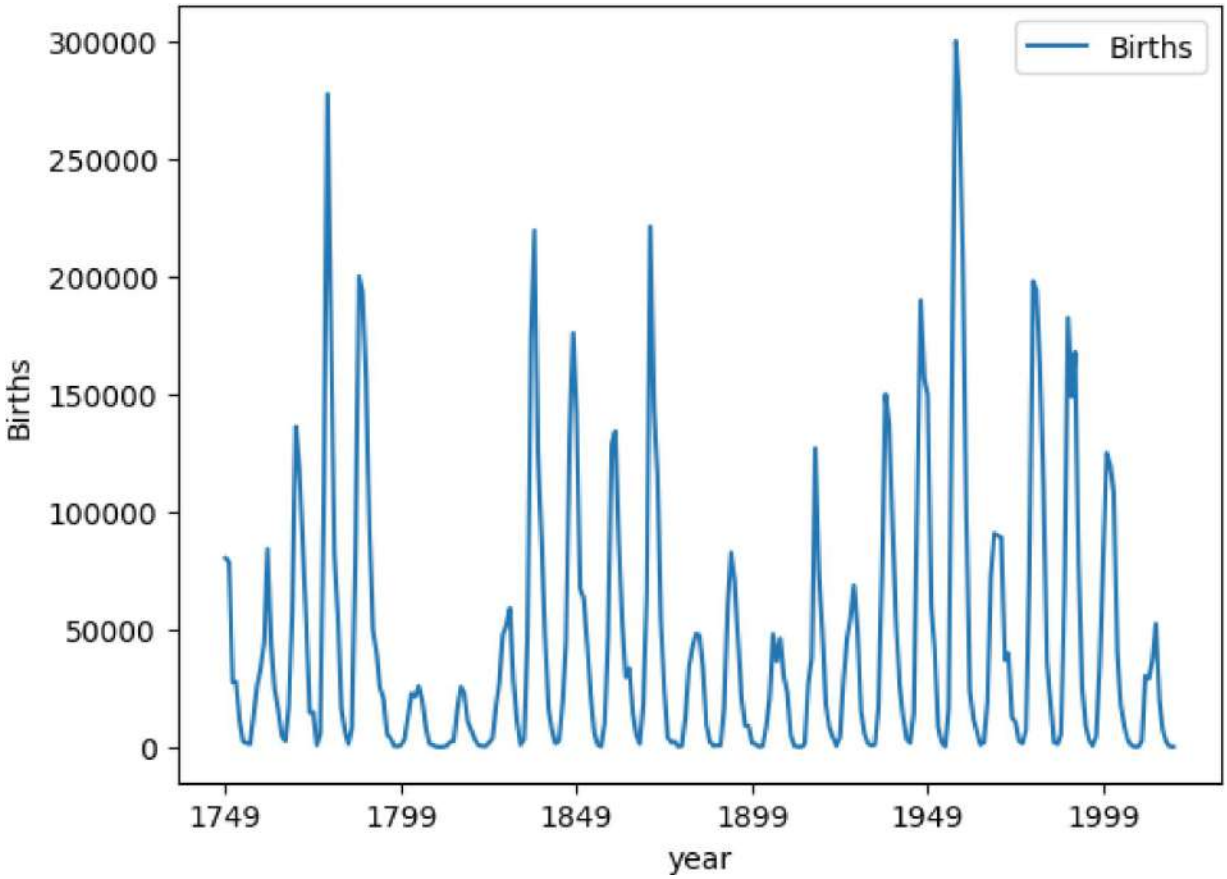


Figure 6-2 An extract of the Births data

Looking at this plot, the pattern is already strikingly clear: high peaks at regular intervals. Let's see how to use the ARMA model to capture this pattern and make predictions for the future. The first step, as always, is to verify whether the data is stationary or not.

Remember that stationarity means that there is no long-term trend in the data: the average is constant over time. When looking at the data, there is no obvious long-term trend, yet you can use the ADF test (Augmented Dickey–Fuller test) to confirm this for you. Listing 6-4 shows you how you can do this.

```
from statsmodels.tsa.stattools import adfuller

result = adfuller(data['Births'])
print(result)

pvalue = result[1]

if pvalue < 0.05:
    print('stationary')
else:
    print('not stationary')
```

Listing 6-4 Applying the ADF test to the Births yearly totals

The result that you obtain when applying this test is unexpected: the ADF test tells you that the data is **stationary.** We therefore do not have to apply differencing, and we can directly use the original data in the model.

Now, the second thing to look at is the autocorrelation function (ACF) and the partial autocorrelation function (PACF). Remember that the ACF and PACF plots can help you to define whether you are working with AR or MA processes. You can obtain the ACF and PACF using Listing 6-5.

```
from statsmodels.graphics.tsaplots import
plot_acf, plot_pacf
import matplotlib.pyplot as plt

plot_acf(data['Births'], lags=40)

plot_pacf(data['Births'], lags=40)

plt.show()
```

Listing 6-5 Creating the ACF and PACF plots

You will obtain the graphs in Figures 6-3 and 6-4.

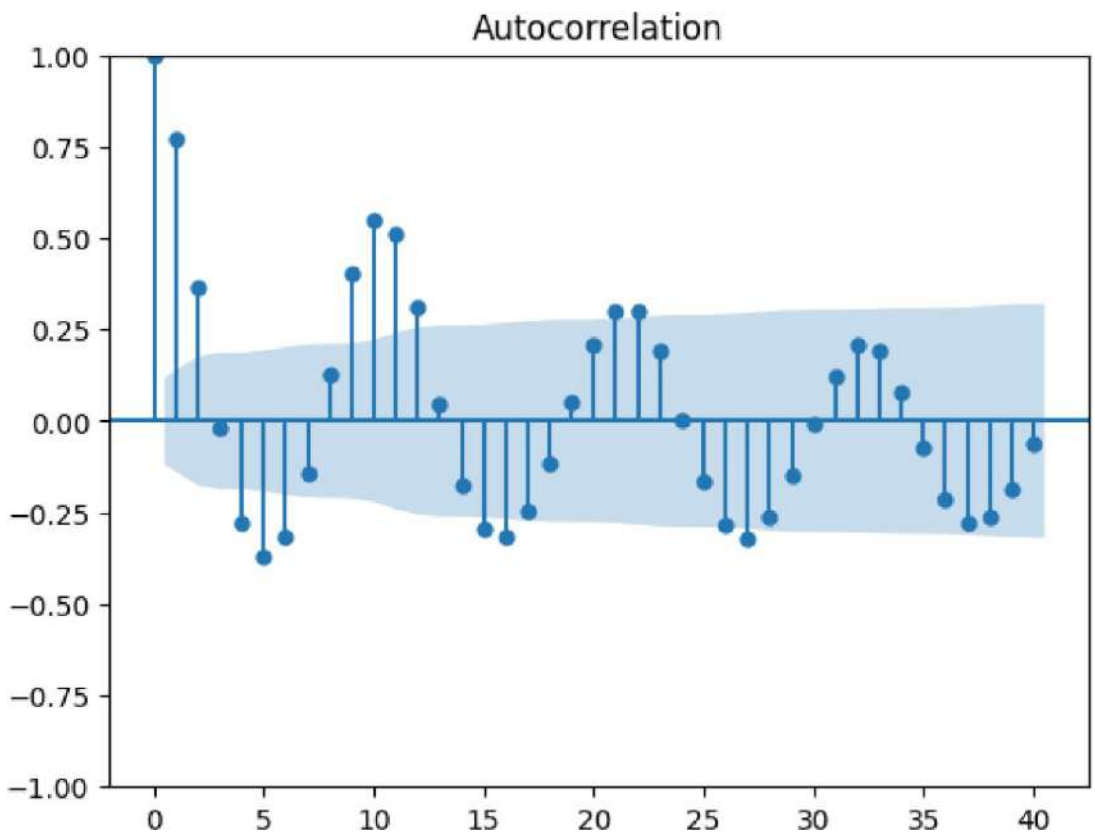


Figure 6-3 The autocorrelation function (ACF)

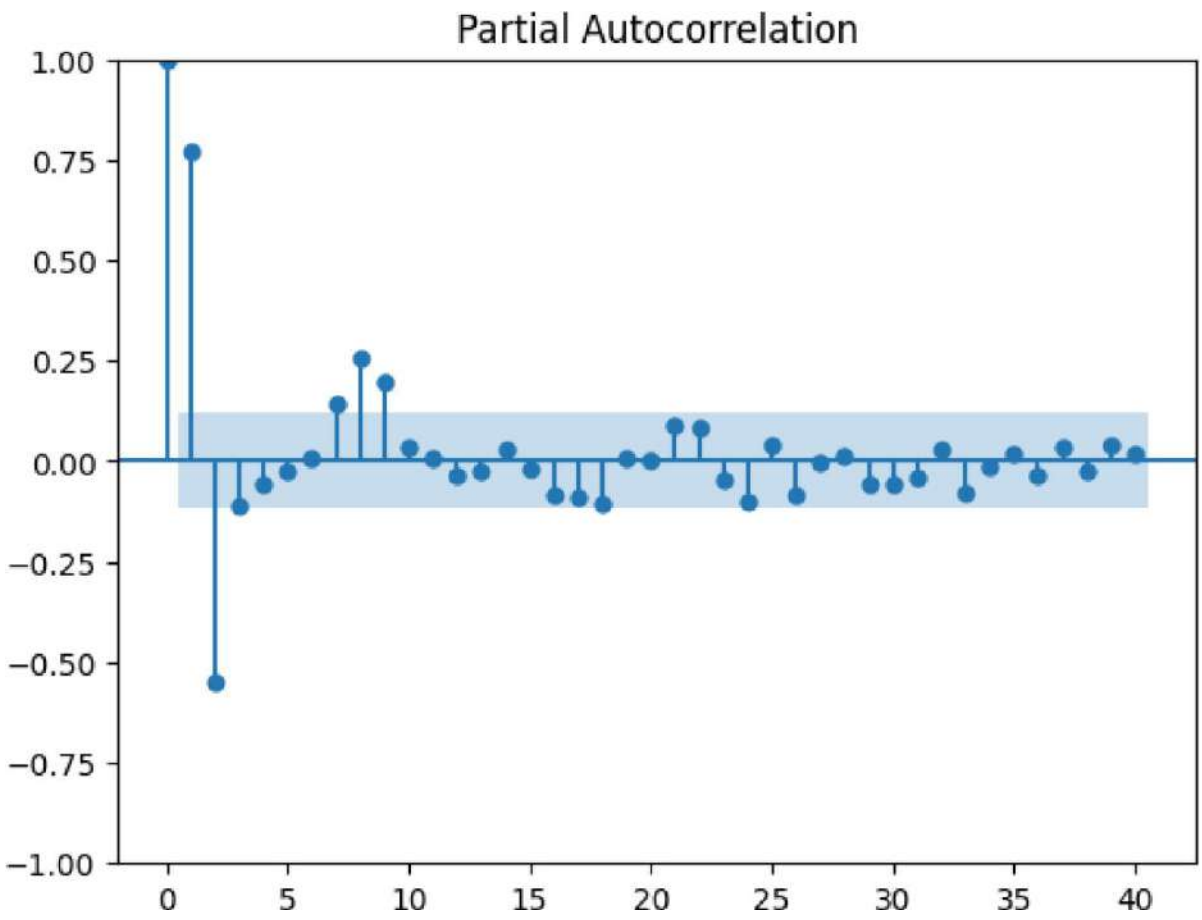


Figure 6-4 The partial autocorrelation function (PACF)

The ACF and PACF plots in Figures 6-3 and 6-4 are great cases to study time series patterns; it is rare to observe such strong patterns on real-life data. Remember from the previous chapters which type of patterns you should be looking at.

- AR processes are recognized by
  - Exponentially decaying partial autocorrelation
  - Partial autocorrelation decaying toward zero
  - Swings between negative and positive partial autocorrelation
- MA processes are identified by
  - Sudden spikes in the ACF and PACF

This means that in the Births data, you have a strong indicator for observing an AR pattern: there is exponential decay in the PACF, together

with switching from negative to positive. There is no real evidence for an MA process when looking at those plots.

Fitting an ARMA(1,1) Model

Given that the ARMA(p,q) model is simply the combination of the AR and MA models, it is important to realize that there is an optimization going on that finds the coefficients of the model that minimize the Mean Squared Error of the model. This is the same approach that has been discussed in the chapter on the MA model.

Remember that there is an important difference between coefficients and hyperparameters. Coefficients are estimated by the model, whereas hyperparameters must be chosen by the modeler. In the ARMA(p, q) model, p and q are the hyperparameters that you must decide on using plots, performance metrics, and more.

To get started, let's see how to fit an ARMA(1, 1) model in Python. An ARMA(1, 1) model means an ARMA model with an AR component of order 1 and an MA component of order 1. Remember that the order refers to the number of historical values that are used to explain the current value.

For this first trial with order (1,1), the choice is just to start with the simplest ARMA model possible. And throughout the example, you'll see how to fine-tune this decision.

Applying an ARMA(1, 1) model for the Births data means that you will explain tomorrow's value by looking at today's actual value (AR part) and today's error (MA part). You are not looking further back into the past, as the order is only 1.

This model can be created in Python using the code in Listing 6-6.

```
from sklearn.metrics import r2_score
from statsmodels.tsa.arima.model import ARIMA

# Forecast the first ARMA(1,1) model
mod = ARIMA(list(data['Births']), order=(1,0,1))
res = mod.fit()
pred = res.predict()

# Print the r2 score of the prediction
```

```
print(r2_score(data, pred))

# Create the plot
plt.plot(list(data['Births']))
plt.plot(pred)
plt.legend(['Actual Births', 'Predicted Births'])
plt.xlabel('Timesteps')
plt.show()
```

Listing 6-6 Fitting the ARMA(1,1) model

This code will also generate a plot of the actual values vs. the fitted values (Figure 6-5). It will also give you an R2 score of the model fit. Since you have not yet created a train-test split, this R2 score is not representative of any future performance. Yet, it can be used to see how wrong the current model is: this gives an intuition on how to improve the model. In this case, with an R2 score of **0.68,** you can conclude that the model is already fitting not too badly.

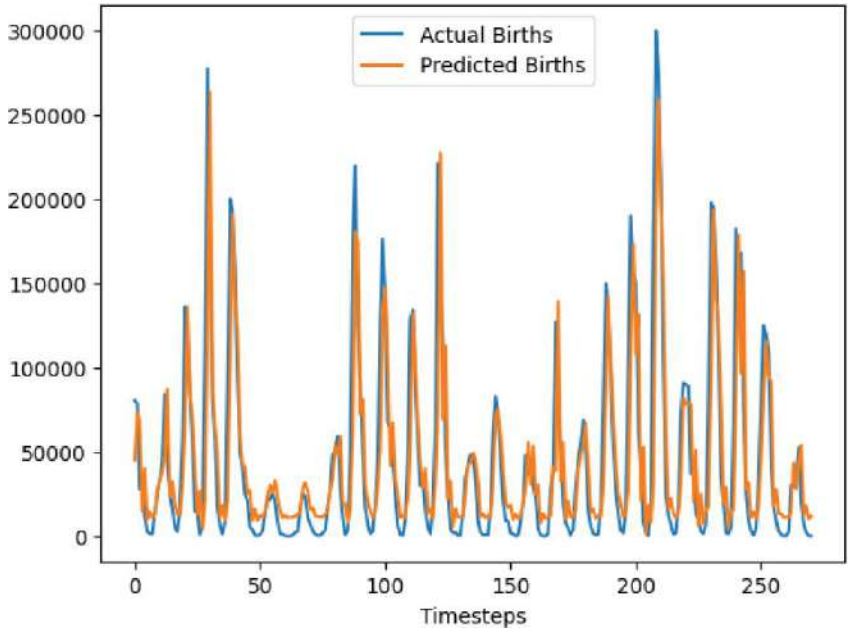


Figure 6-5 The actual values vs. the fitted values of the ARMA(1,1)

More Model Evaluation KPIs

As an addition to performance metrics, I want to take the opportunity to inspect a few more model fit KPIs. Until now, you have seen how to use the Augmented Dickey–Fuller test to help you decide on differencing, and you have seen the ACF and PACF plots to help you decide on the type of model needed (AR vs. MA) and to help you choose the best order. You have also seen the train-test-split combined with a hyperparameter search to identify the order that maximizes out-of-sample performance.

An additional KPI that can help you evaluate model fit is to look at the residuals of your model. If a model has a good fit, you will generally observe that the residuals follow a normal distribution. If you find the opposite, your model is generally missing out on important information.

You can make a histogram of the residuals of your model using the code in Listing 6-7.

```
ax = pd.Series(res.resid).hist()
ax.set_ylabel('Number of occurences')
ax.set_xlabel('Residual')
plt.show()
```

Listing 6-7 Plotting a histogram of the residuals

You should obtain the graph in Figure 6-6.

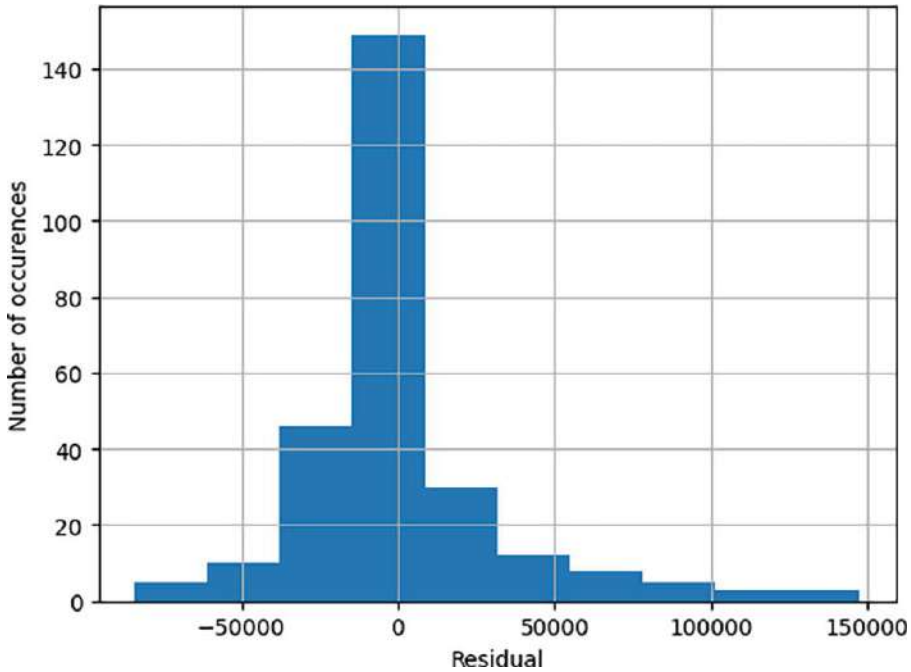


Figure 6-6 Histogram of the residuals of the ARMA(1,1) model

You can observe visually that the histogram of the residuals does not follow the bell-shaped curve that is the **normal distribution**. This means that there are probably some lags to be added, which is logical: you already know that the Births data has an 11-year seasonality, and for now, there is only one lagged period (one year back) included in the model.

> **Note** There are many different approaches for testing the distribution of a variable. This includes histograms, QQ-plots, and numerous hypothesis tests. Although hypothesis tests for normality are useful in many cases, the goal of a forecast is generally to maximize future performance. It is not necessary to have residuals that are perfectly normally distributed for this purpose, and the graphical way of identifying the normality of residuals is generally sufficient in practice.

When you've evaluated the normality of residuals of your model, you can also look at the summary table of your model. You can obtain this summary table using the code in Listing 6-8.

```
res.summary()
```

Listing 6-8 Obtaining the summary table of your model's fit

You should obtain the summary table shown in Figure 6-7.

SARIMAX Results

Dep. Variable:	y	**No. Observations:**	271
Model:	ARIMA(1, 0, 1)	**Log Likelihood**	-3204.666
Date:	Mon, 19 May 2025	**AIC**	6417.332
Time:	12:59:15	**BIC**	6431.741
Sample:	0	**HQIC**	6423.117
	- 271		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
const	4.508e+04	1.15e+04	3.922	0.000	2.26e+04	6.76e+04
ar.L1	0.6359	0.060	10.658	0.000	0.519	0.753
ma.L1	0.4774	0.066	7.185	0.000	0.347	0.608
sigma2	1.202e+09	1.215	9.89e+08	0.000	1.2e+09	1.2e+09

Ljung-Box (L1) (Q):	2.43	**Jarque-Bera (JB):**	426.71
Prob(Q):	0.12	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.35	**Skew:**	1.70
Prob(H) (two-sided):	0.16	**Kurtosis:**	8.12

Figure 6-7 Summary table of the ARMA(1,1) model

Important pieces of information that you can get from this table are **the estimates of the coefficients** and the corresponding **p-values**. Remember that you are working with an ARMA(1,1) model and that you therefore will have one coefficient for the AR model (for the data point one time step back) and one coefficient for the MA model (for the error of one time step back).

When you fit the model to the data, your model is estimating the best possible values for those coefficients. You can see the estimates of the coefficients in the table on the line that says **ar.L1** (AR coefficient for the first lag is **0.6359**) and on the line that says **ma.L1** (MA coefficient for the first lag is **0.474**).

You can interpret those by looking back at the formula. Basically, those estimates state that you can predict a future data point by filling in the ARMA formula using the **const** as c, the ar.L1 as the coefficient φ_i (phi), and the ma.L1 as the coefficient θ_i (theta). Filling in those coefficients

would **allow you to compute the next value**. This is great information for understanding what information is used by your model to predict the future.

Now that you know the estimated values of your coefficients, you can also look at the hypothesis tests for those coefficients. For each coefficient, the summary table shows a hypothesis test that tells you whether the coefficient is **significantly different from zero**. If a coefficient were to be zero, this would mean that it is not useful to include the parameter. Yet, if a hypothesis test proves that a coefficient is significantly different from zero, this means that it is useful to add the coefficient to your model. This helps you in deciding which order to use for your ARMA forecast.

The p-values of the hypothesis tests can be found on the line of the coefficient, under the column **P>|z|**. This column gives you the p-value of the hypothesis test. The p-value must be **smaller than 0.05** to prove that the coefficient is significantly different from zero.

In this case, the interpretation is as follows:

- The p-value for the ar.L1 coefficient is 0.000, so significant.
- The p-value for the ma.L1 coefficient is also 0.000, so significant.
- Both are smaller than 0.05, and therefore, you can conclude that both coefficients should be retained in the model.

As you can imagine, this is very useful information when creating a model. You can go back and forward between **increasing and decreasing order of AR and MA independently,** and try to find at which point the coefficients start to become insignificant (p-values higher than 0.05), in which case you take the order of the highest lag that was still significant as the best model. Of course, keep in mind that significance is one indicator among multiple indicators.

Automated Hyperparameter Tuning

Although this theoretical approach can be very interesting, you should combine it with tests of predictive performance, as you have done already in the AR and MA chapters using very basic approaches to hyperparameter optimization.

If you remember, what you did was create a train-test split and then loop through all possible values of the hyperparameter and evaluate the

performance on the test set. The order that gave the best performance on the test set was retained.

In this part, you will make an addition to this approach by formalizing the notions of grid search and cross-validation.

Grid Search: Tuning for Predictive Performance

I've purposely waited to explain **grid search** until here because it is better explained with an example with two hyperparameters. In the ARMA model, two hyperparameters need to be optimized at the same time: p and q.

The goal of a grid search for hyperparameter optimization is to make the task of choosing hyperparameters. You have seen multiple statistical tools that can help you make the choice:

- Autocorrelation function (ACF)
- Partial autocorrelation function (PACF)
- Model residuals
- Summary table

With multiple indicators, it is difficult to decide on the best parameters. Grid search is a method that tests every combination of **hyperparameters** and evaluates the predictive error of this combination of hyperparameters. This will give you an objective estimate of the hyperparameters that will obtain the best performance.

A basic approach for grid search could be to make a **train-test split** and to fit a model with each combination of hyperparameters on the train set and evaluate the model on the test set. In practice, grid search is often combined with **cross-validation**. As you've seen in Chapter 2, cross-validation is an augmentation of the train-test split approach in which the train-test split is repeated multiple times. This yields a more reliable error estimate.

To apply a grid search with cross-validation, you need to implement a loop through each combination of hyperparameters (in this case, p and q). For each combination, the code will split the data into multiple **cross-validation splits**: multiple combinations of train-test splits. Those splits are also called folds. For each of those folds, you train the model on the training set and test the model on the test set. This yields an error for each fold, of which you take the average to obtain an error for each

hyperparameter combination. The hyperparameter combination with the best error will be the model that you retain for your forecast.

Listing 6-9 shows how this is done in code.

```
import numpy as np
from sklearn.model_selection import
TimeSeriesSplit

def train_model(p, q):
    errors = []
    tscv = TimeSeriesSplit(test_size=10)
    for train_index, test_index in
tscv.split(data_array):
        X_train, X_test = data_array[train_index],
data_array[test_index]
        X_test_orig = X_test
        fcst = []
        for step in range(10):
            mod = ARIMA(X_train, order=(p,0,q))
            res = mod.fit()
            fcst.append(res.forecast(steps=1))
            X_train = np.concatenate((X_train,
X_test[0:1,:]))
            X_test = X_test[1:]
        errors.append(r2_score(X_test_orig, fcst))
    pq_result = [p, q, np.mean(errors)]
    return pq_result

data_array = data.values
avg_errors = []

for p in range(13):
    for q in range(13):

        try:
            pq_result = train_model(p, q)
            print(pq_result)
            avg_errors.append(pq_result)
```

```
        except:
            print(p,q,'failure')

avg_errors = pd.DataFrame(avg_errors)
avg_errors.columns = ['p', 'q', 'error']
result = avg_errors.pivot(index='p', columns='q')
['error']
```

Listing 6-9 Grid search with cross-validation for optimal p and q

As you may have noticed while running this code, a serious disadvantage of using grid search is the computation time. This approach requires building a model for each combination of p and q, so in this case, 144 models. In addition, it uses a ten-fold cross-validation, which means that it is doing 144×10 model fits, resulting in 1440 model fit calls per step. Then we need to do this times ten steps as we are predicting ten steps into the future, giving 14400 models. This needs to be done only during the model training phase and will not impact the performance while using the model for prediction, so it is not a problem.

The result of the grid search is a table with an estimated error for each combination of p and q in the limits that you have specified (between 0 and 11). Thanks to the cross-validation approach, you can be confident in this error estimate: after all, it is the average of five separate error measurements, and this makes it unlikely that any bias has been caused by a favorable test set selection. You can see the results table in Figure 6-8.

q	0	1	2	3	4	5	6	7	8	9	10	11	12
p													
0	-0.552147	0.321259	0.461742	0.555505	0.568762	0.559429	0.563130	0.545076	0.563256	0.564998	0.557743	0.509542	0.555429
1	0.496821	0.539755	0.541083	0.570586	0.550879	0.548254	0.555083	0.562839	0.552935	0.581444	0.572143	0.558822	0.575166
2	0.556640	0.561521	0.556841	0.559763	0.536552	0.579318	0.614786	0.618531	0.592417	0.634507	0.632872	0.622011	0.617235
3	0.563754	0.558914	0.607426	0.553582	0.627372	0.616020	0.624376	0.625844	0.586794	0.631589	0.628973	0.622237	0.617452
4	0.558876	0.555752	0.607092	0.561269	0.625109	0.616702	0.620260	0.578066	0.586185	0.604715	0.596195	0.606357	0.597592
5	0.549850	0.548181	0.484629	NaN	0.598138	0.604675	0.603888	0.588146	0.605752	0.592703	0.600780	0.599046	0.597465
6	0.548639	0.542947	0.511568	NaN	0.573850	0.602310	0.602713	0.607587	0.600879	0.587890	0.561435	0.566828	0.608186
7	0.574454	-14948.588036	0.588800	-124115.100016	0.544212	0.604189	0.604758	0.606885	0.567326	0.605090	0.595855	0.581914	0.602880
8	0.570866	0.588649	0.594692	0.587513	0.564257	0.582936	0.596064	0.571309	0.590459	0.557474	0.621180	0.605360	0.600622
9	0.615605	0.605638	0.599715	0.590460	0.577021	0.566043	0.604985	0.591911	0.585137	0.588086	0.588852	0.569886	0.562923
10	0.603438	0.603265	0.605521	0.585704	0.601293	NaN	0.609429	0.604269	0.579033	0.612963	0.588162	0.620767	NaN
11	0.602976	0.599362	0.605517	0.570449	0.580241	0.575266	NaN	0.580806	0.559829	0.583965	0.562598	0.591993	0.540105
12	0.597799	0.604076	0.552598	0.569322	0.595942	0.606012	NaN	NaN	NaN	0.589261	0.601276	0.591573	NaN

Figure 6-8 Table with the R2 scores for each combination of p and q

The combination that has yielded the best R2 score is the combination of **p = 2 and q = 9**, which yields an estimated R2 score on the test set of **0.63**. As a final step, you can refit the model on a train-test split to plot this on a train-test set to see what this error looks like. This is done in Listing 6-10 and should give you the graph displayed in Figure 6-9.

```
data_array = data.values
X_train, X_test = data_array[:-10],
data_array[-10:]
X_test_orig = X_test

fcst = []
for step in range(10):
    mod = ARIMA(X_train, order=(2,0,9))
    res = mod.fit()
    fcst.append(res.forecast(steps=1))
    X_train = np.concatenate((X_train,
X_test[0:1,:]))
    X_test = X_test[1:]

plt.plot(X_test_orig)
plt.plot(fcst)
plt.legend(['Actual Births', 'Predicted Births'])
plt.xlabel('Time steps of test data')
plt.show()
```

Listing 6-10 Showing the test prediction of the final model

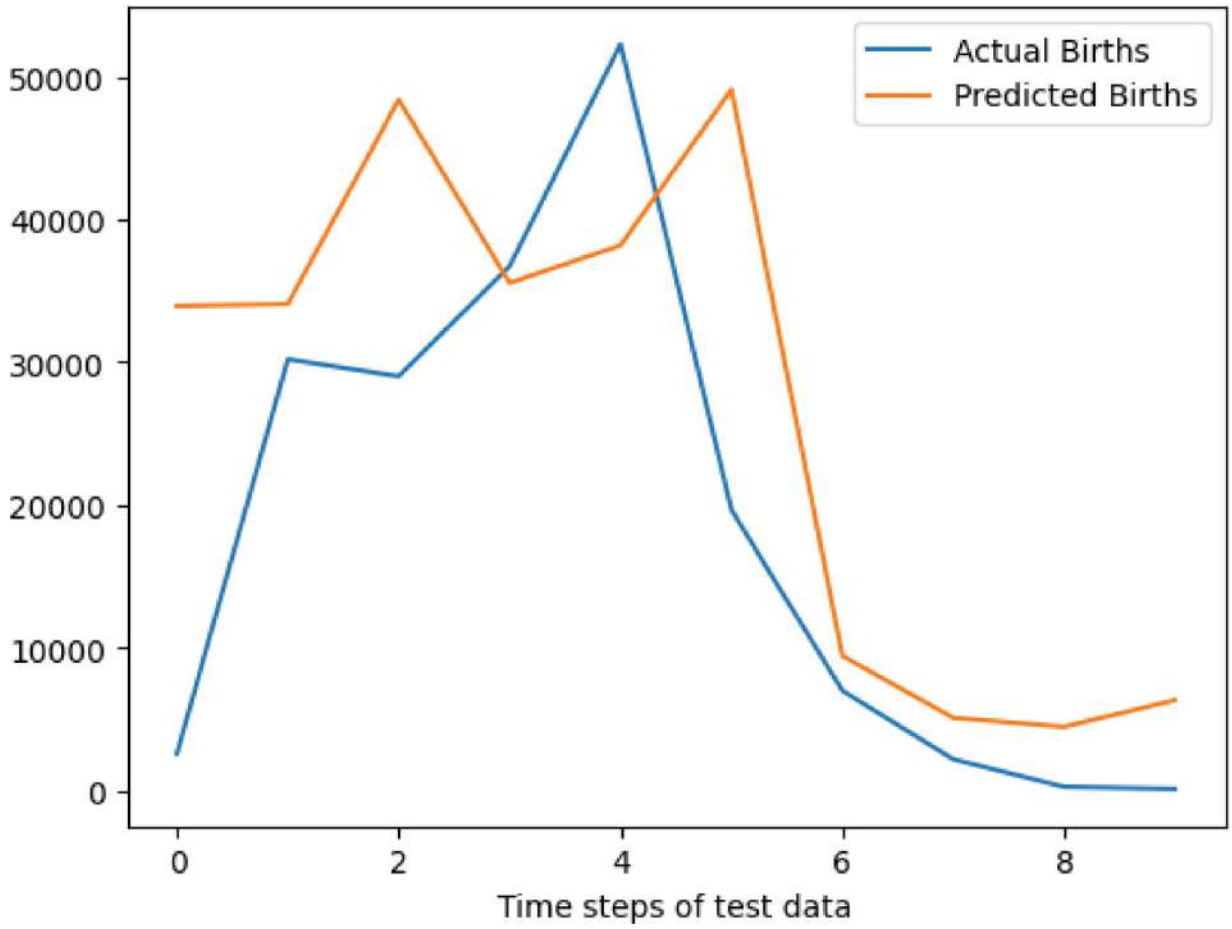


Figure 6-9 Prediction on the test set by the final model

Saving This Model in MLflow

As a final step, let's save the models using MLflow. As described earlier, as this is a multistep forecast, you are creating a new model for each step. Additionally, due to the grid search in combination with the cross-validation, we have created many models.

When getting new data, you are going to have to retrain a new model for each step. However, if you save your results, you won't have to rerun the whole grid search, nor the cross-validation. So, the important thing to have stored here is mainly the values of p and q resulting from the grid search and cross-validation, as well as the corresponding R2 score. You can do so with the code in Listing 6-11

```
import numpy as np
```

```
from sklearn.model_selection import
TimeSeriesSplit

mlflow.autolog()

def train_model(p, q):

    errors = []
    tscv = TimeSeriesSplit(test_size=10)
    for train_index, test_index in
tscv.split(data_array):
        X_train, X_test = data_array[train_index],
data_array[test_index]
        X_test_orig = X_test
        fcst = []
        for step in range(10):
            mod = ARIMA(X_train, order=(p,0,q))
            res = mod.fit()
            fcst.append(res.forecast(steps=1))
            X_train = np.concatenate((X_train,
X_test[0:1,:]))
            X_test = X_test[1:]
        errors.append(r2_score(X_test_orig, fcst))
    pq_result = [p, q, np.mean(errors)]
    return pq_result

with mlflow.start_run():

    data_array = data.values

    avg_errors = []

    for p in [2]:
        for q in [9]:

            try:
                pq_result = train_model(p, q)
```

```
                # Log this metric to mlflow
                mlflow.log_metric('cross-
validation-r2-score', pq_result[2])
                avg_errors.append(pq_result)

            except:
                print(p,q,'failure')

    avg_errors = pd.DataFrame(avg_errors)
    avg_errors.columns = ['p', 'q', 'error']
    result = avg_errors.pivot(index='p',
columns='q')['error']
```

Listing 6-11 Adding MLflow

Key Takeaways

- The ARMA model is the combination of the Autoregressive model and the Moving Average model.
- The hyperparameters of ARMA are p and q: p for the autoregressive order and q for the moving average order.
- Grid search is a common tool for optimizing the choice of hyperparameters. It can be combined with cross-validation to yield more reliable error estimates.
- You can use the distribution of residuals to evaluate the fit of a model. If the residuals do not follow a normal distribution, there is generally something wrong with the model specification.
- You can use the model summary to look at detailed indicators of model fit. You can also find the estimates of model coefficients and a hypothesis test that tests the model coefficients against zero.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_7

7. The ARIMA Model

Joos Korstanje[1] ✉
(1) Maisons Alfort, France

Having seen several building blocks of univariate time series, in this chapter, you're going to see a model that combines even more of the time series components: the ARIMA model.

To keep track of the different building blocks, let's get back to the table of time series components (Table 7-1) to see where the ARIMA model is at.

Table 7-1 The building blocks of univariate time series models

Name	Explanation	Chapter
AR	Autoregression	4
MA	Moving Average	5
ARMA	Combination of AR and MA models	6
ARIMA	Adding differencing (i) to the ARMA model	7
SARIMA	Adding seasonality (S) to the ARIMA model	8
SARIMAX	Adding external variables (X) to the SARIMA model *(note that external variables mean that it is not univariate anymore)*	9

As you can see, the ARIMA is getting close to the final model. The only parts that are not yet included are S (seasonality) and X (external variables). Some time series processes do not have seasonality or external variables, which means that the ARIMA is frequently used as a standalone model.

In practice, if you are doing univariate time series, you will generally go directly to ARIMA if you have no expectation of seasonality, or if you expect seasonality, you will go directly to SARIMA (which will be covered in the next chapter).

ARIMA Model Definition

The ARIMA model is a model that combines the AR and MA building blocks, just like you have seen with ARMA in the previous chapter. The addition in ARIMA is

the “I” building block, which stands for **automatic differencing of non-stationary time series**.

You should remember that during the past chapters, stationarity has been presented as an important concept in time series. A stationary time series is a time series that has no long-term trend. If a time series is not stationary, you can make it stationary by applying differencing: replacing the actual values with the difference between the actual and the previous value.

The “I” in ARIMA is for **integrating**. Integrating is a more mathematical synonym for differencing a non-stationary time series. In ARIMA, this differencing is not done in advance of the modeling phase, but it is done during the model fit.

Model Definition

The ARIMA model is short for Autoregressive, Integrated Moving Average. The difference with the ARMA model is small: there is just an additional effect that makes the time series non-stationary. A simple example of this would be a linearly increasing trend, as shown in the following equation:

$$X_t = c + \varepsilon_t + \sum_{i=1}^{p} \varphi_i X_{t-i} + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} + \delta t$$

When differencing a time series, you actually start to model the differences from one step to another rather than the original values. If the actual values of a variable are not stable over time, it is still possible that the differences are stable over time.

The linear trend is a great example of this. Imagine a linear trend that starts from 0 and increments by 2 every time step. The values will not be stationary at all: they will increase infinitely. Yet the differences between each value and the next are always 2, so the differenced time series is perfectly stationary.

ARIMA on the CO_2 Example

The fact that the differencing is part of the hyperparameters has a great added value for model building. This makes it possible to do automated hyperparameter tuning on the number of times that differencing should be applied. Let’s see this with an example.

The data in this example are weekly CO_2 data that are available through the statsmodels library. You can obtain this public domain data using the code in Listing 7-1.

Citation for the data set: Keeling, C.D. and T.P. Whorf. 2004. Atmospheric CO_2 concentrations derived from flask air samples at sites in the SIO network. In Trends: A Compendium of Data on Global Change. Carbon Dioxide Information Analysis

Center, Oak Ridge National Laboratory, U.S. Department of Energy, Oak Ridge, Tennessee, the USA.

```
import statsmodels.api as sm
data = sm.datasets.co2.load_pandas()
data = data.data
data.head()
```

Listing 7-1 Importing the data

To get an idea of the data that you're working with, as always, it is good to make a plot of the data over time. Use the code in Listing 7-2 to obtain the graph in Figure 7-1.

```
import matplotlib.pyplot as plt
ax = data.plot()
ax.set_ylabel('CO2 level')
plt.show()
```

Listing 7-2 Importing the data

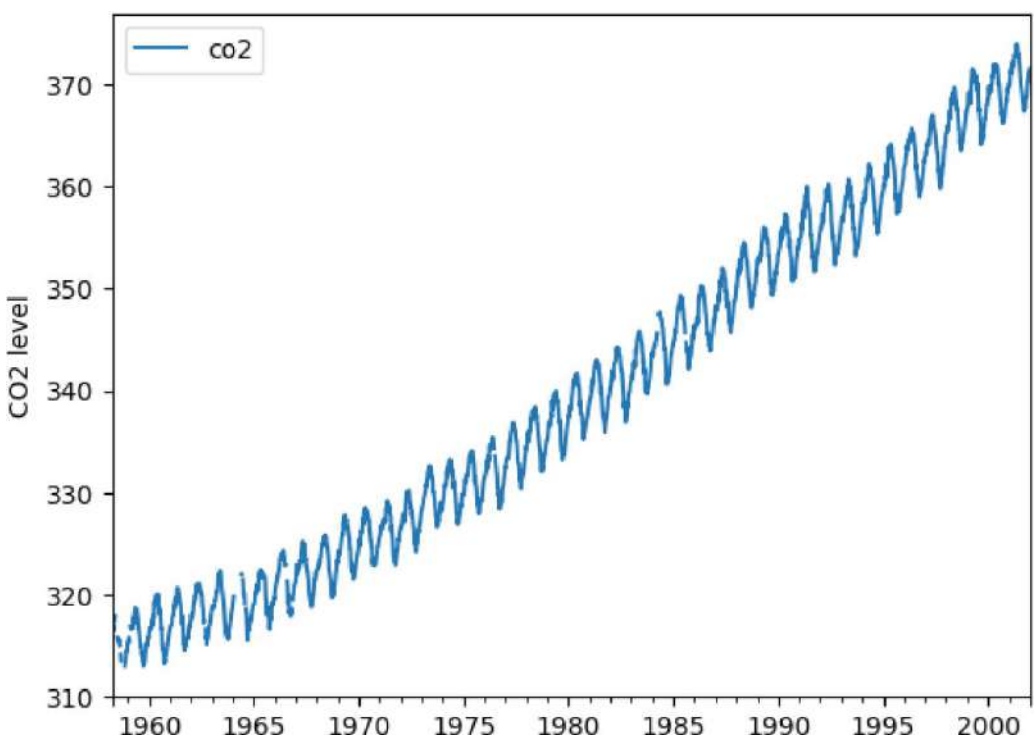


Figure 7-1 Plot of the CO_2 data over time

This data shows a very obvious sign of both an upward trend, which is fairly constant. And there is also a very clear seasonality pattern in this data (up and down).

Now, remember from the previous chapters, the first thing that you should generally look at is autocorrelation and partial autocorrelation functions. Remember that the autocorrelation (ACF) and partial autocorrelation (PACF) plots are relevant only when applied to stationary data. You can use Listing 7-3 to create ACF and PACF plots directly on the differenced data.

```
from statsmodels.graphics.tsaplots import plot_acf,
plot_pacf
plot_acf(data.diff().dropna(), lags=40)
plot_pacf(data.diff().dropna(), lags=40)
plt.show()
```

Listing 7-3 ACF and PACF plots

You should obtain the ACF plot shown in Figure 7-2 and the PACF plot in Figure 7-3.

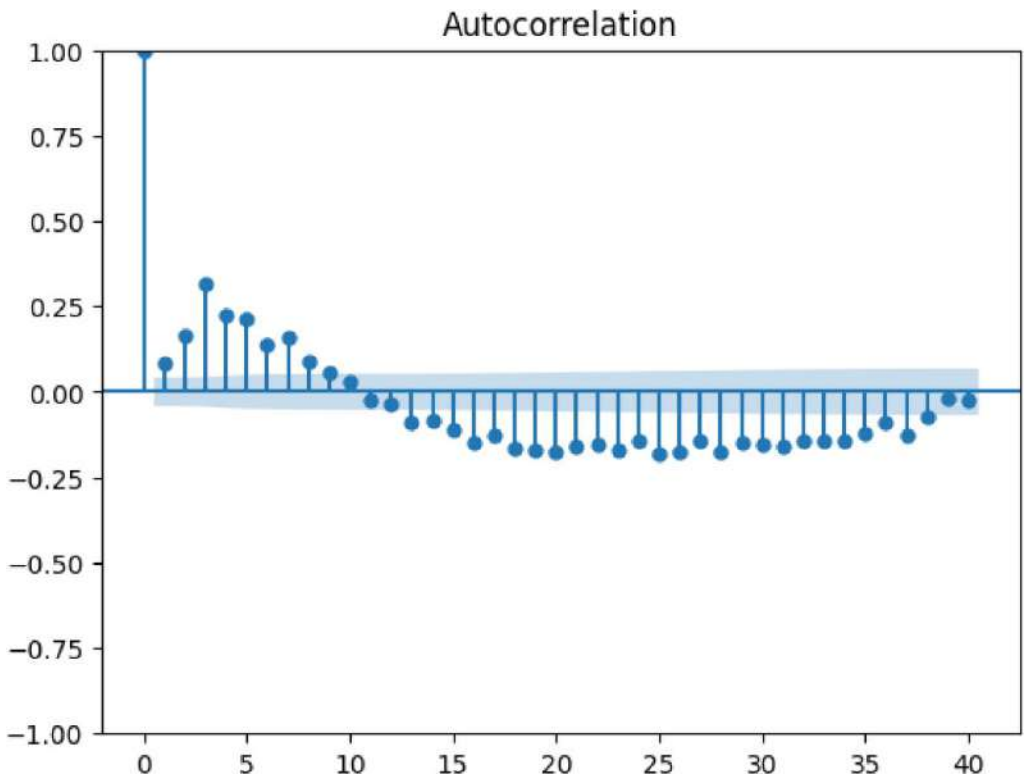


Figure 7-2 Autocorrelation function plot

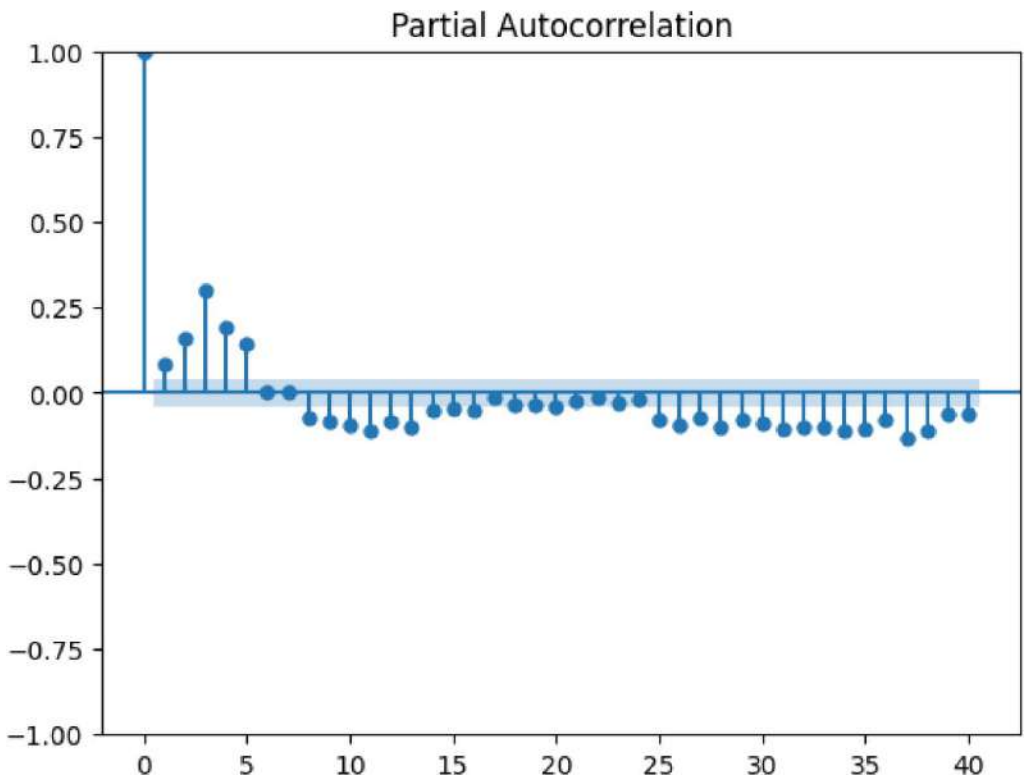


Figure 7-3 Partial autocorrelation function plot

It is interesting to note that there are many lagged values in the autocorrelation and partial autocorrelation. So, what does this mean? In general, if there is no decay of the correlations toward zero, this means the data is not stationary. Yet the data has been differenced and seems stationary.

The answer here is that the decay occurs relatively late in the autocorrelation. The ACF and PACF plots have 40 lags, but that is not enough for the current example. You can use Listing 7-4 to obtain ACF and PACF plots that go further back, and you will observe that there is a decay toward zero. The resulting graphs are shown in Figure 7-4 and Figure 7-5.

```
plot_acf(data.diff().dropna(), lags=600)
plot_pacf(data.diff().dropna(), lags=600)
plt.show()
```

Listing 7-4 ACF and PACF plots with more lags

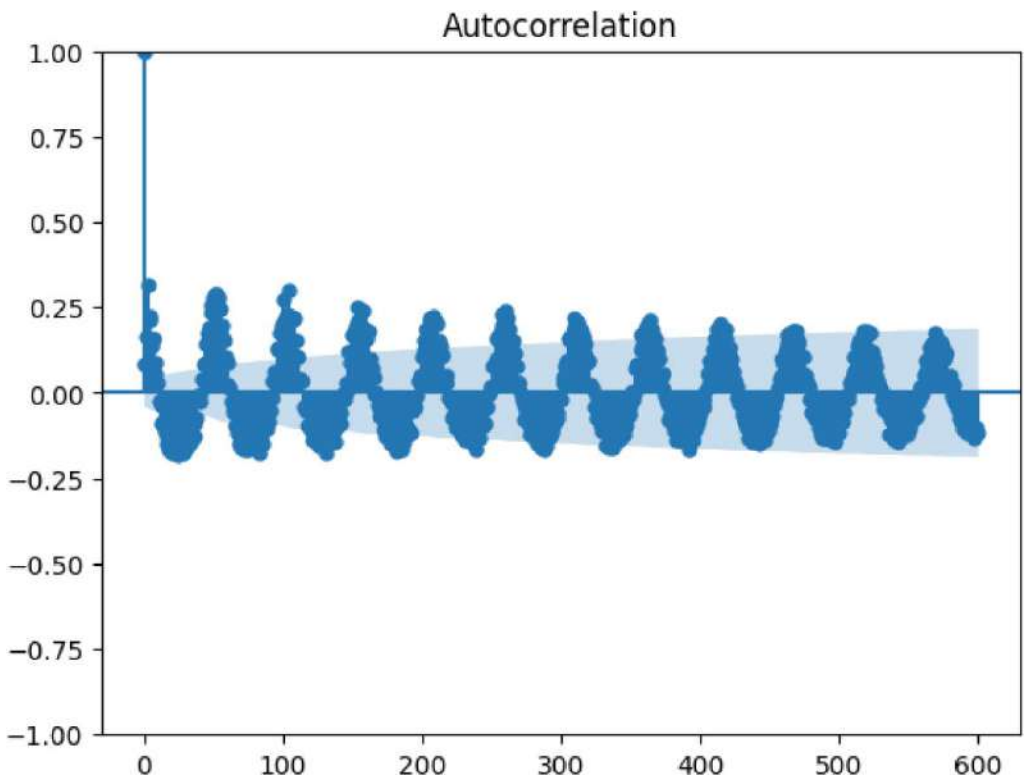


Figure 7-4 Autocorrelation function plot with 600 lags

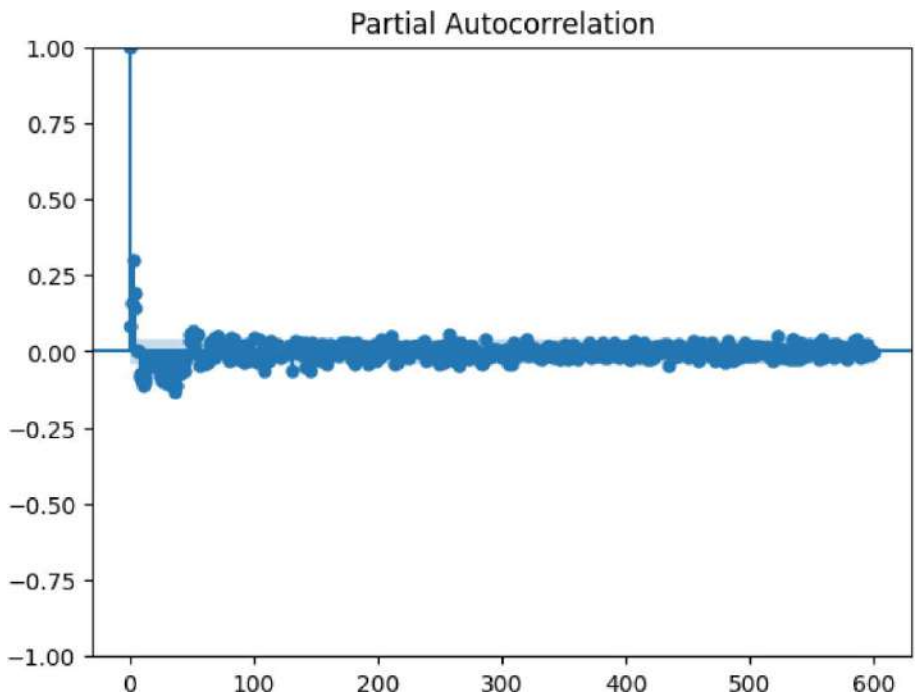


Figure 7-5 Partial autocorrelation function plot with 600 lags

This makes it an interesting case for the question of the order of the model. Let's see how to augment the previous chapters' grid search cross-validation by adding the I as

a third hyperparameter to use in the optimization. The code for this can be seen in Listing 7-5. You'll see that an MLflow experiment is created that logs the parameters and R2 scores to a local mlruns folder.

```
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import r2_score
import numpy as np
from sklearn.model_selection import TimeSeriesSplit
import mlflow

exp = mlflow.create_experiment(name='arima')
print('exp = ', exp)

data_array = data[['co2']].values

avg_errors = []

for p in range(6):
    for q in range(6):
        for i in range(3):

            run = mlflow.start_run(experiment_id=exp)
            print('run = ', run.info.run_id)

            mlflow.log_param('p', p)
            mlflow.log_param('q', q)
            mlflow.log_param('i', i)

            errors = []

            tscv = TimeSeriesSplit(test_size=10)

            for train_index, test_index in
tscv.split(data_array):

                X_train, X_test = data_array[train_index],
data_array[test_index]
                X_test_orig = X_test

                fcst = []
                for step in range(10):

                    try:
```

```
                        mod = ARIMA(X_train, order=
(p,i,q))
                        res = mod.fit()

                        fcst.append(res.forecast(steps=1))

                    except:
                        print('errorred')
                        fcst.append(np.array([9999999.0]))

                    X_train = np.concatenate((X_train,
X_test[0:1,:]))
                    X_test = X_test[1:]

                errors.append(r2_score(X_test_orig, fcst))

            mean_err = np.mean(errors)

            pq_result = [p, i, q, mean_err]

            print(pq_result)
            avg_errors.append(pq_result)

            mlflow.log_metric('10step-r2-score-
crossvalidated', mean_err)
            mlflow.end_run()

avg_errors = pd.DataFrame(avg_errors)
avg_errors.columns = ['p', 'i', 'q', 'error']
avg_errors.sort_values('error', ascending=False)
```

Listing 7-5 Hyperparameter tuning

The result of this code will give you a dataframe ordered by R2 scores. The best R2 score is **0.729**, and it is obtained for the combination of **p = 4, q = 3, and i = 1**. This means that there is an autoregressive effect of order 4 and a moving average effect of order 3. The data must be differenced one time. Listing 7-6 shows you how to show the test fit of the model, and the graph is shown in Figure 7-6.

```
X_train, X_test = data_array[:-10], data_array[-10:]
X_test_orig = X_test

fcst = []
for step in range(10):
```

```
    mod = ARIMA(X_train, order=(4,1,4))
    res = mod.fit()
    fcst.append(res.forecast(steps=1))
    X_train = np.concatenate((X_train, X_test[0:1,:]))
    X_test = X_test[1:]

plt.plot(fcst)
plt.plot(X_test_orig)
plt.legend(['Predicted', 'Actual'])
plt.ylabel('CO2 Level')
plt.xlabel('Time Step of Test Data')
plt.show()
```

Listing 7-6 Plot the final result

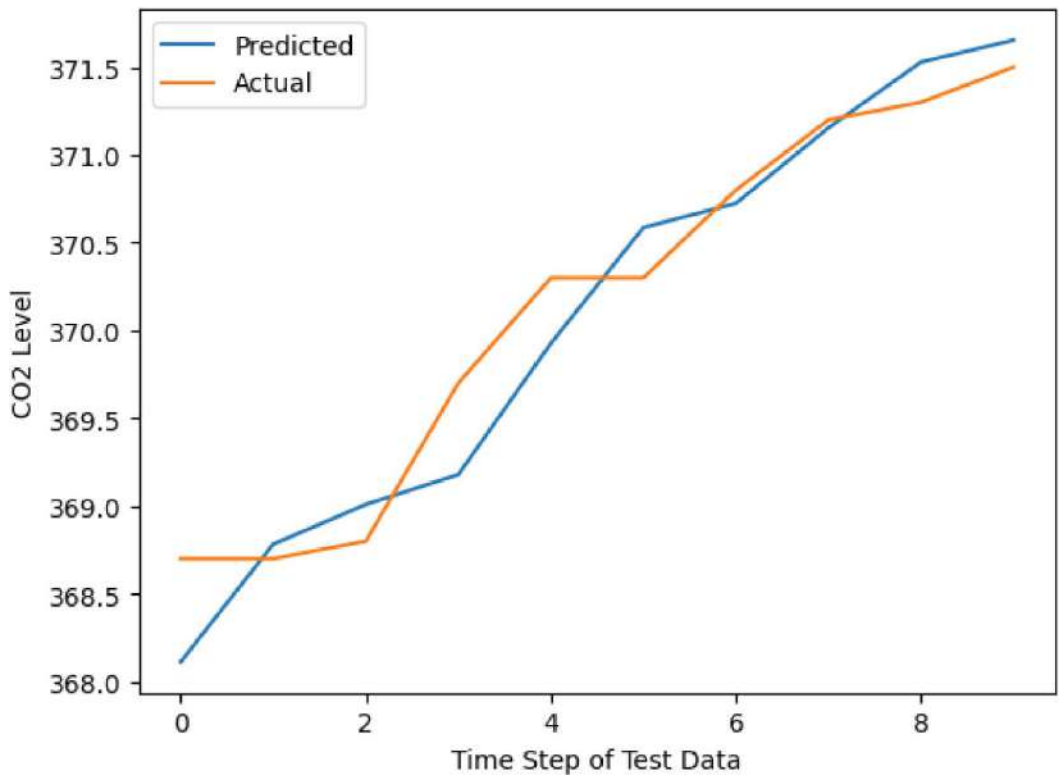


Figure 7-6 Result of the best forecast

Key Takeaways

- The ARIMA model combines three effects:
  - The AR process, based on autocorrelations between past and present values
  - The MA process, based on correlations between past errors and present values
  - Automatic integration of a time series if it is not stationary

- The ARIMA(p, I, q) model has three hyperparameters:
  - The order of the AR process is denoted by p.
  - The order of the MA process is denoted by q.
  - The order of integration is denoted by i (or d in some notations).

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_8

8. The SARIMA Model

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, you are going to discover the final model of univariate time series models: the SARIMA model. This will be the chapter in which everything on univariate time series comes together.

Table 8-1 The building blocks of univariate time series models

Name	Explanation	Chapter
AR	Autoregression	4
MA	Moving Average	5
ARMA	Combination of AR and MA models	6
ARIMA	Adding differencing (i) to the ARMA model	7
SARIMA	Adding seasonality (S) to the ARIMA model	8
SARIMAX	Adding external variables (X) to the SARIMA model *(note that external variables mean that it is not univariate anymore)*	9

Univariate Time Series Model Breakdown

With the ARIMA components covered in previous chapters, you can now model autoregressive processes, moving average components, and integration (differencing). A common process of time series is still missing from this: **seasonality**.

Let's have a look at the univariate time series breakdown table to see where this fits in (Table 8-1). You can see that there is only one model more complicated than the SARIMA model: the SARIMAX model.

The SARIMAX model is quite different from SARIMA: it uses external variables that correlate with the target variable. This makes SARIMAX a very

powerful tool as well, but it can be applied only if you have external variables. Using external variables makes the model not a univariate time series model.

The power of the SARIMA model is that it needs to use only the history of the target variable, which means that it can be applied to many cases of forecasting. Within univariate time series modeling, the SARIMA model is the most complete model, using AR, MA, integration for modeling trends, and seasonality.

The SARIMA Model Definition

The SARIMA model combines many different processes in one and the same model. Therefore, the model definition is going to be relatively complex mathematically. The following equation is the definition of the SARIMA model.

$$y_t = u_t + \eta_t$$

$$\varphi_p(L)\widetilde{\phi}_P(L^s)\Delta^d\Delta_s^D u_t = A(t) + \theta_q(L)\tilde{\theta}_Q(L^s)\zeta_t$$

In this model, you can observe the following parts:

- The regular AR part, with the coefficients φ_p (small phi)
- The seasonal AR part, with the coefficients $\widetilde{\phi}_P$ (capital phi)
- The regular MA part, with the coefficients θ_q (small theta)
- The seasonal MA part, with the coefficients $\tilde{\theta}_Q$ (capital theta)
- The regular integration part, indicated by order d
- The seasonal integration part, indicated by order D
- Coefficient of seasonality s

As you can see, adding seasonality to the ARIMA model makes the model much more complex. The seasonality is not just one building block added to the previous model. There is a seasonal part added for each of the building blocks. This means that the lag selection of the SARIMA model is also going to be much more difficult. The same hyperparameters as before still need to be decided on:

- p for the AR order
- q for the MA order
- I for the differencing order

To this are added three more orders:

- P for the seasonal AR order
- Q for the seasonal MA order
- D for the seasonal differencing

And lastly, the choice of parameter **s, the seasonal period,** must be made. This s is not a hyperparameter like those of order. S should not be tested using grid search; rather, it should be decided based on logic. The s is the periodicity of the seasonality. If you work with monthly data, you should choose 12; if you work with weekly data, you should choose 52, etc.

Example: SARIMA on Walmart Sales

In this chapter, you will apply the SARIMA model to a Walmart sales forecasting dataset that is available on Kaggle (https://www.kaggle.com/c/walmart-recruiting-store-sales-forecasting/).

Let's get this data in Python and start to see what it looks like using Listing 8-1. Note that the Walmart data contains many stores and products, but for this exercise, you will take the sum of the weekly data.

```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv(train.csv')
data = data.groupby('Date').sum()
ax = data['Weekly_Sales'].plot()
ax.set_ylabel('Weekly sales')
plt.gcf().autofmt_xdate()
plt.show()
```

Listing 8-1 Importing the data and creating a plot

This will give you the plot shown in Figure 8-1.

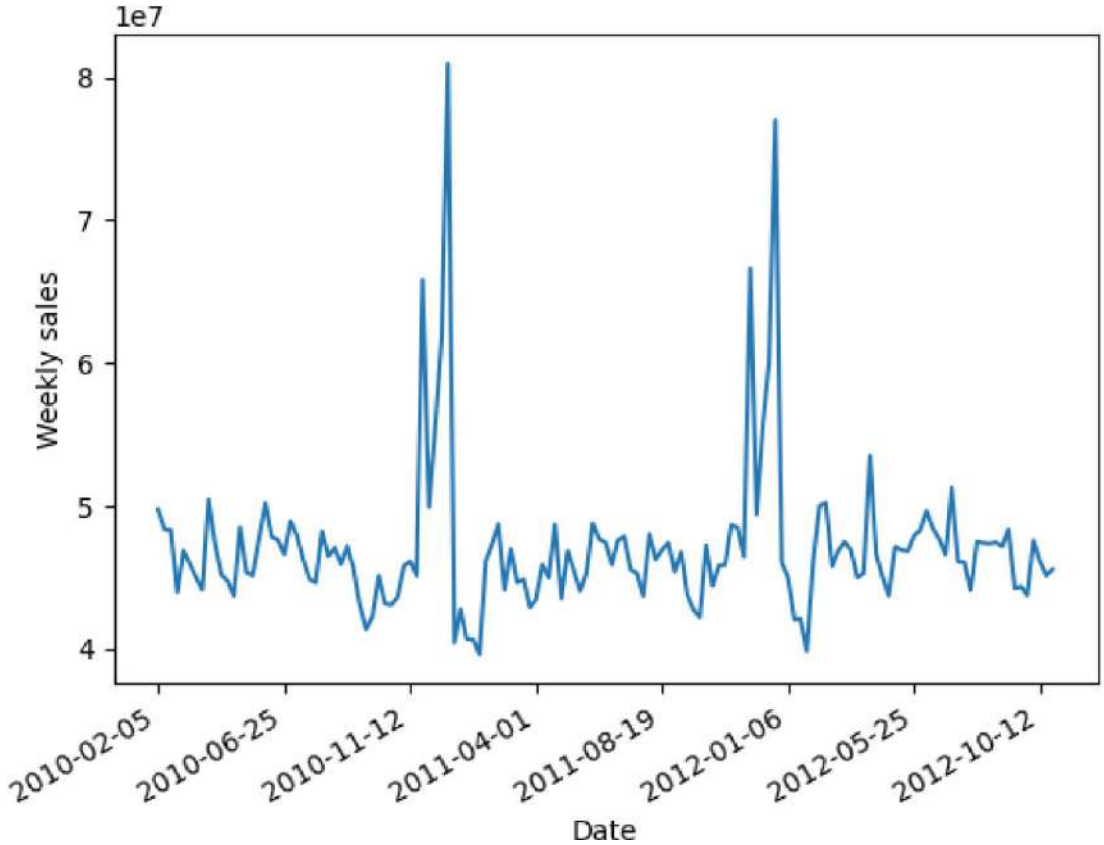


Figure 8-1 Walmart sales over time

Let's first try a first model: SARIMA(1, 1, 1)(1, 1, 1)52. This notation indicates an order of 1 for each of the hyperparameters. The seasonality is 52, because you're working with weekly data. You will be forecasting ten steps out rather than doing ten one-step forecasts. This is a case of multistep forecasting. Listing 8-2 shows how to fit the model and obtain a plot of the performance on the test set (Figure 8-2). You will also obtain the R2 score on the test data.

```
import random
random.seed(12345)
import statsmodels.api as sm
from sklearn.metrics import r2_score

train = data['Weekly_Sales'][:-10]
test = data['Weekly_Sales'][-10:]
mod = sm.tsa.statespace.SARIMAX(data['Weekly_Sales']
[:-10], order=(1,1,1), seasonal_order=(1,1,1,52))
res = mod.fit(disp=False)
fcst = res.forecast(steps=10)
```

```
plt.plot(list(test))
plt.plot(list(fcst))
plt.legend(['Actual data', 'Forecast'])
plt.ylabel('Sales')
plt.xlabel('Test Data Time Step')
r2_score(test, fcst)
```

Listing 8-2 Fitting a SARIMA(1,1,1)(1,1,1)52 model

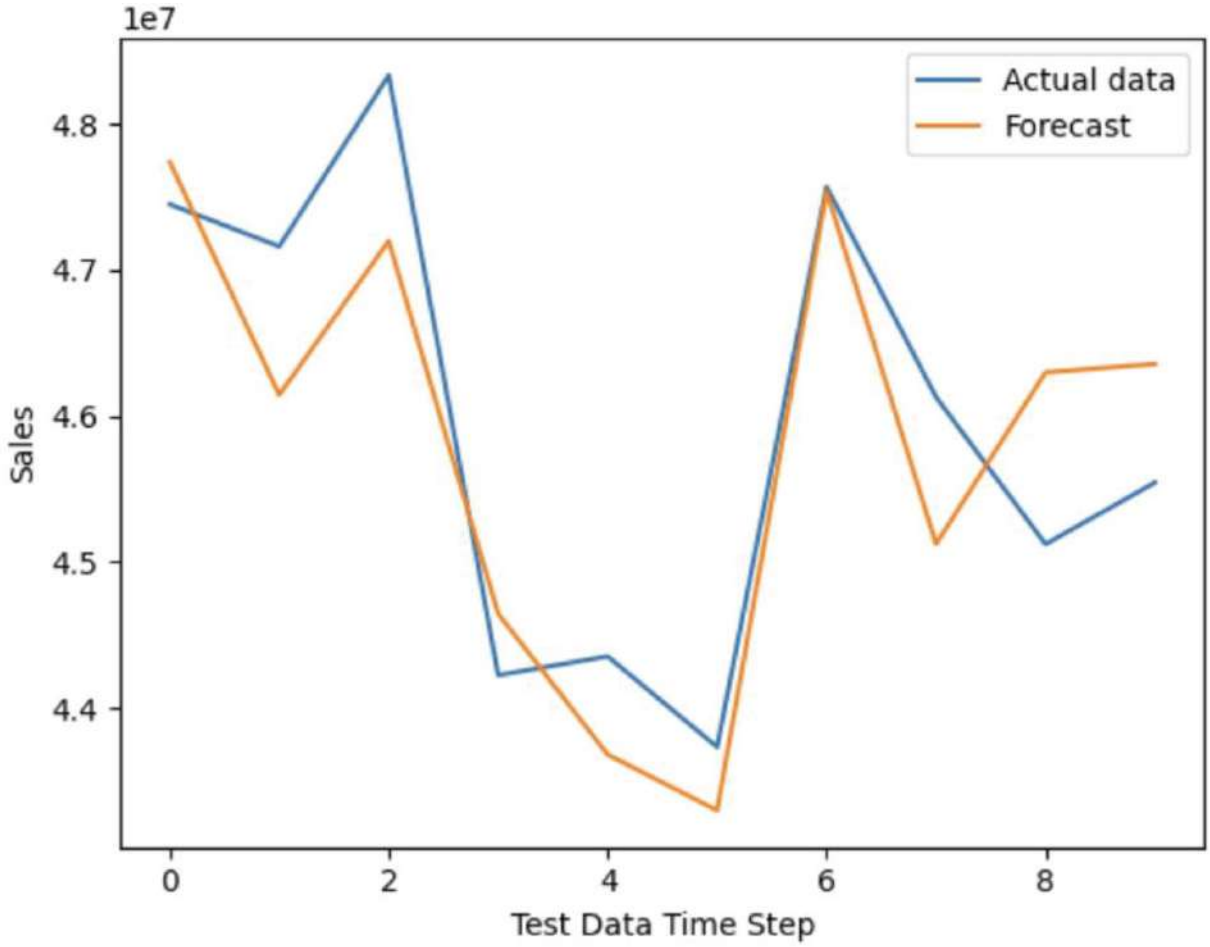


Figure 8-2 Predictive performance of the SARIMA model

In this version, without any hyperparameter optimization, the SARIMA model obtains an R2 score of **0.73**. This means that the model can explain 73% of the variation in the test data. This is a very promising score.

The reason that this score is already quite good is probably the flexibility of the SARIMA model. The fact that it is a combination of multiple effects means that it can fit many different underlying processes. You will now see how this score can improve using a grid search of the hyperparameters.

To speed up the code, you will not use cross-validation in the grid search here. Speed of execution is also an important aspect of machine learning. As a second optimization, you will be working with a ten-step forecast rather than with ten updated steps of a one-step forecast. In this way, it will be interesting to see how the seasonal effects may be more efficient in capturing long-term trends. This will be done in Listing 8-3. As usual, the code also sets up MLflow to track the experiment.

Note that a try-except block has been added to avoid the code from stopping in case an error occurs while fitting the model. Depending on your hardware, some of the fits may be too heavy and cause out-of-memory errors. This is not a problem, as lit will still allow you to find the best possible combination of hyperparameters.

```
import mlflow
exp = mlflow.create_experiment(name='sarima3')
print('exp = ', exp)

scores = []
for p in range(2):
    for i in range(2):
        for q in range(2):
            for P in range(2):
                for D in range(2):
                    for Q in range(2):

                        run =
mlflow.start_run(experiment_id=exp)

                        print('run = ',
run.info.run_id)

                        mlflow.log_param('small_p', p)
                        mlflow.log_param('small_q', q)
                        mlflow.log_param('i', i)
                        mlflow.log_param('cap_P', P)
                        mlflow.log_param('D', D)
                        mlflow.log_param('cap_Q', Q)

                        try:
```

```
                                mod =
sm.tsa.statespace.SARIMAX(train, order=(p,0,q),
seasonal_order=(P,D,Q,52))
                                res = mod.fit(disp=False)

                                r2 = r2_score(test,
res.forecast(steps=10))

                                score = [p,i,q,P,D,Q,r2]
                                print(score)
                                scores.append(score)

                                mlflow.log_metric('10step-
r2-score-test', r2)

                                del mod
                                del res

                            except:
                                print('errored')

                            mlflow.end_run()

res = pd.DataFrame(scores)
res.columns = ['p', 'i', 'q', 'P', 'D', 'Q', 'score']
res.sort_values('score')
```

Listing 8-3 Tuning the SARIMA model

You see that the best order here is (1,0,1)(1,0,0). In short, this notation means that it is a regular AR process, as well as a seasonal AR process. A regular MA process is present, but no seasonal MA process. Neither regular nor seasonal integration is needed.

This model gives you an R2 on the test data of **0.710**, just slightly better than the non-optimized model. As the last step, you can check out the plot of this forecast on the test data. This can be done by updating the order in Listing 8-2, and this will allow you to obtain the plot in Figure 8-3.

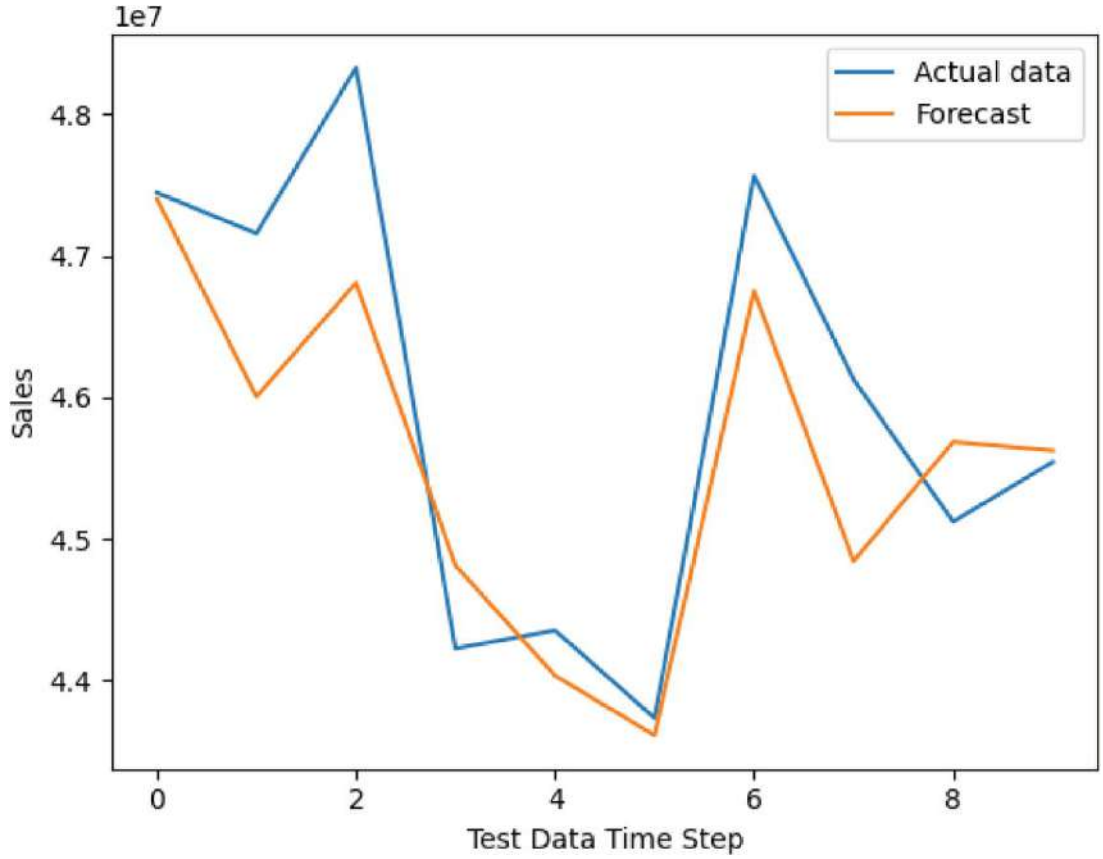


Figure 8-3 Predictive performance of the optimized SARIMA model

---

Key Takeaways

- The SARIMA model adds a seasonal effect to the ARIMA model.
- The SARIMA model is the complete version of univariate time series models.
- There are four more hyperparameters in the SARIMA model:
  - Seasonal AR order
  - Seasonal MA order
  - Seasonal integration order
  - Periodicity (based on the number of periods that a seasonality would logically return)

Part III
Multivariate Time Series Models

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_9

9. The SARIMAX Model

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, you will discover the SARIMAX model. This model is the most complete version of classical time series models, as it contains all of the components that you've discovered throughout the previous chapters of this book. It adds the X component: external variables.

Time Series Building Blocks

Let's have a quick look back at the different components of time series models that you have seen throughout the previous chapters using Table 9-1.

Table 9-1 The building blocks of univariate time series models

Name	Explanation	Chapter
AR	Autoregression	4
MA	Moving Average	5
ARMA	Combination of AR and MA models	6
ARIMA	Adding differencing (i) to the ARMA model	7
SARIMA	Adding seasonality (S) to the ARIMA model	8
SARIMAX	Adding external variables (X) to the SARIMA model *(note that external variables mean that it is not univariate anymore)*	9

The chapters build up time series models from easy to complex. The SARIMAX model is not considered a perfect example of a univariate time

series model. Univariate time series models only use variation in the target variable, while the SARIMAX model uses external variables as well.

Model Definition

The mathematical definition of the SARIMAX model is as follows:

$$y_t = \beta_t x_t + u_t$$

$$\varphi_p(L)\widetilde{\phi}_P(L^s)\Delta^d\Delta_s^D u_t = A(t) + \theta_q(L)\tilde{\theta}_Q(L^s)\zeta_t$$

The β (beta) part in the first formula represents the external variable. For the rest, the model is very similar to the SARIMA model, but for completeness, let's relist the hyperparameters of this model:

- p for the AR order
- q for the MA order
- I for the differencing order
- P for the seasonal AR order
- Q for the seasonal MA order
- D for the seasonal differencing
- s for the seasonal coefficients

Supervised Models vs. SARIMAX

In further chapters, you'll see many cases of supervised models. As you may recall from the first chapter, supervised models use external variables to predict a target variable. This idea is very similar to the X part in SARIMAX.

The question, of course, is which one of them you should use in practice. From a theoretical point of view, the SARIMAX may be preferred in cases where the time series part is more present than the external variables part. If the external variables alone can explain a lot, and this is complemented by a part of autocorrelation or seasonality, supervised models may be a better choice.

Yet, as always, the reasonable thing to do is to use multiple models in a model benchmark. The choice for a model should then simply be based on the predictive performance of the model.

Example of SARIMAX on the Walmart Dataset

In the previous chapter, you've seen the SARIMA model used to predict weekly sales at Walmart. Yet, the dataset does not just contain weekly sales data: there is also an indicator that tells you whether a week did or did not have a holiday in it.

You did not use the holiday information in the SARIMA model, as it is impossible to add external data to it. And this did not really matter as there were no extreme peaks due to holidays in the test data. Even though the holiday information was missing in the model, it was not represented in the error estimate.

Yet, it is important to add such information, as it may be able to explain a large part of the variation in the model. Let's see how to improve on the Walmart example using external variables.

As a first step, you should import and prepare the data. You can use Listing 9-1 for this. This code also shows you how to create a plot of sales over time.

```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('walmart/train.csv')
data = data.groupby('Date').sum()
data['IsHoliday'] = data['IsHoliday'] > 0
data['IsHoliday'] = data['IsHoliday'].apply(
    lambda x: float(x)
)

ax = data['Weekly_Sales'].plot()
ax.set_ylabel('Weekly Sales')
plt.gcf().autofmt_xdate()
plt.show()
```

Listing 9-1 Preparing the data and making a plot

You should obtain the graph displayed in Figure 9-1.

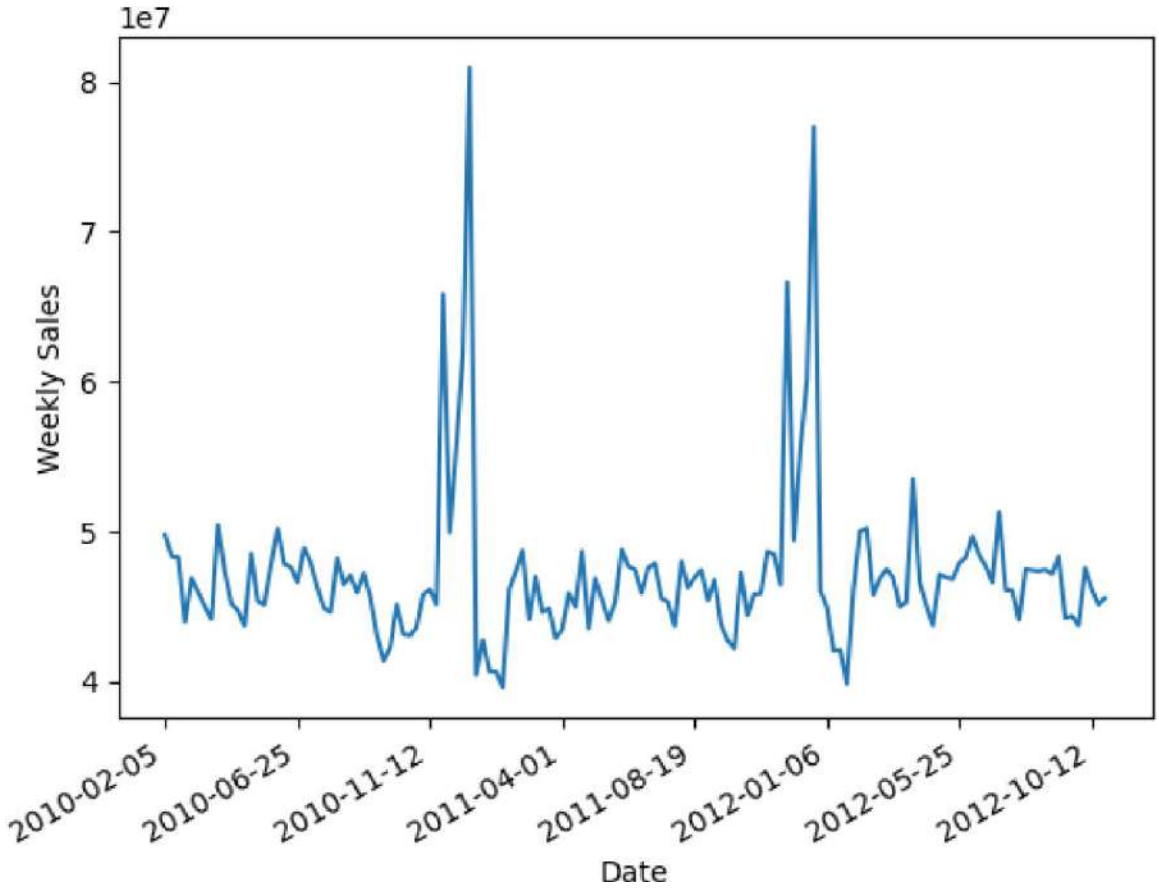


Figure 9-1 Plot of the weekly sales

The next step is to use correlation analysis to study whether we may expect an improvement from adding the holiday information into the model. If there is no correlation at all between sales and holidays, it would be unwise to add it to the model. Let's use Listing 9-2 to compute the correlation coefficient.

```
data[['Weekly_Sales', 'IsHoliday']].corr()
```

Listing 9-2 Is there a correlation between sales and holidays?

This will output a correlation matrix as shown in Figure 9-2.

	Weekly_Sales	IsHoliday
Weekly_Sales	1.000000	0.172683
IsHoliday	0.172683	1.000000

Figure 9-2 The correlation matrix

The correlation coefficient between sales and holidays is **0.17**. This is not very high, but enough to consider adding the variable to the model. The model itself will compute the most appropriate coefficient for the variable.

To create a SARIMAX model, you can use Listing 9-3. It is important to note the terminology of **endog** and **exog**:

- **Endogenous variables** (endog) are the target variable. This is where all the time series components will be estimated from. In the current case, it is the weekly sales.
- **Exogenous variables** (exog) are explanatory variables. This is where the model takes additional correlation from. In the current example, this is the holiday variable.

For this particular example, I will not do the hyperparameter search again. If you remember from the previous chapter, the optimal score was found using a SARIMA(0,1,1)(1,1,1),52. So, for this example, let's use a SARIMAX(0,1,1)(1,1,1)52. The code also computes an R2 score and shows the plot of performance on a test set of ten weeks. MLflow is set up to collect the model and some basic metrics using autologging.

```
import random
random.seed(12345)
import statsmodels.api as sm
from sklearn.metrics import r2_score
import mlflow

mlflow.autolog()

train = data['Weekly_Sales'][:-10]
```

```
test = data['Weekly_Sales'][-10:]

mod = sm.tsa.statespace.SARIMAX(
    endog=data['Weekly_Sales'][:-10],
    exog=data['IsHoliday'][:-10],
    order=(0,1,1),
    seasonal_order=(1,1,1,52),
)
res = mod.fit(disp=False)
fcst = res.forecast(steps=10, exog =
data['IsHoliday'][-10:])

plt.plot(list(test))
plt.plot(list(fcst))
plt.xlabel('Steps of the test data')
plt.ylabel('Weekly Sales')
plt.legend(['test', 'forecast'])
plt.show()

print(r2_score(test, fcst))
```

Listing 9-3 Fitting a SARIMAX model

The R2 score obtained by this forecast is **0.734**. The performance can be seen visually in Figure 9-3.

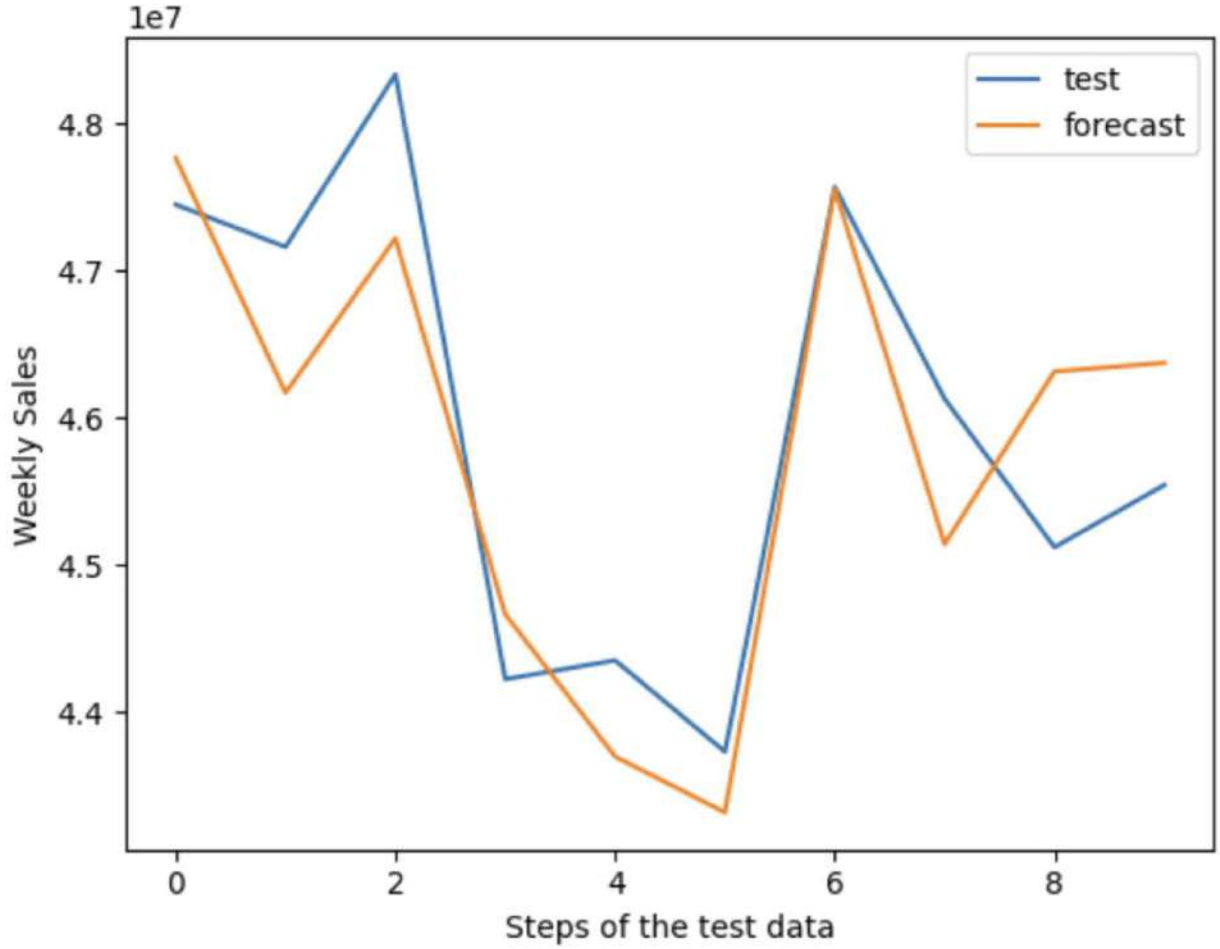


Figure 9-3 Predictive performance on the test data

This chapter has shown you how to use the SARIMAX model using one additional variable. Yet, remember that there is no limit to the number of exogenous variables included in a SARIMAX model. As long as you can provide future values for this variable, this may work.

Pay attention here: many variables that can improve model performance are not useful in practice. For example, you may be able to explain sales by using weather data. Yet, weather data for the future is not yet known. So while this variable may have an important impact on sales, it cannot be used for improving forecasting accuracy.

Key Takeaways

- The SARIMAX model allows for adding external variables to the SARIMA model.

- The SARIMAX has two types of variables:
  - *Endogenous*: The target variable
  - *Exogenous*: The external variables
- You need to specify the future values of the exogenous variable when forecasting. Therefore, when using exogenous variables, it is important to know that you have fixed information for the future about them.
  - A variable like holidays can work, as you know which holidays will occur in the future.
  - A variable like weather cannot work, as you will not know which weather will occur in the future.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_10

10. The VAR Model

Joos Korstanje[1] ✉
(1) Maisons Alfort, France

In this chapter, you will discover the VAR model, short for **Vector Autoregression**. A Vector Autoregression models the development over time of multiple variables at the same time.

The term autoregression should sound familiar from the previous chapters. The Vector Autoregression model is part of the family of time series models. Like other models from this category, it predicts the future based on developments in the past of the target variables.

There is an important specificity to the Vector Autoregression. Unlike most other models, a Vector Autoregression models multiple target variables at the same time. The multiple variables are used at the same time as target variables and as explanatory variables for each other. A model that uses multiple target variables in one model is called a **multivariate model** or, more specifically, a **multivariate time series**.

The Model Definition

Since the VAR model proposes one model for multiple target variables, it regroups those variables as a vector. This explains the name of the model. The model definition is as follows:

$$y_t = c + A_1 y_{t-1} + A_2 y_{t-2} + \ldots + A_p y_{t-p} + e_t$$

In this formula,

- $\boldsymbol{y_t}$ is a vector that contains the present values of each of the variables.

- The order of the VAR model, **p**, determines the number of time steps back that are used for predicting the future.
- *c* is a vector of constants: one for each target variable.
- There is a vector of coefficient *A* for each lag.
- The error is denoted as ***e***.

Order: Only One Hyperparameter

So, how does the number of lags work in the VAR model? In the previous models, you have seen relatively complex situations with hyperparameters to be estimated for the order of different types of processes.

The VAR model is much simpler in this regard. There is only one order to be defined: the order of the model as a whole. The notation for this order is **VAR(p)**, in which p is the order.

The order defines how many time steps back in time are taken into account for explaining the present. An order 1 means only the previous time step is taken into account. An order 2 means that the two previous time steps are used, etc. You can also call this the number of lags that are included.

If a lag is included, it is always included for all the variables. It is not possible with the VAR model to use different lags for different variables.

Stationarity

Another concept that you have seen before is stationarity, so let's find out how that works in the VAR model.

If you remember, stationarity means that a time series has no trend: it is stable over the long term. You can use the Augmented Dickey–Fuller test to test whether a time series is stationary or not.

You have also seen the technique called differencing, or integration, to make a non-stationary time series stationary.

This theory is also very important for the VAR model. A VAR model can work only if *each of the variables in the model is stationary*. When doing VAR models, you generally work on multiple variables at the same time, so it can be a bit cumbersome to handle if some of the time series are stationary and others are not. In that case, differencing should be applied only to the non-stationary time series.

Estimation of the VAR Coefficients

The coefficients of the VAR model can be estimated using a technique called Multivariate Least Squares. The fact that the VAR model uses this technique makes it computationally relatively efficient. To understand this estimation technique, let's pose the VAR model in matrix notation:

$$Y = BZ + U$$

In this case,

- Y is a matrix that contains the y values for each variable (each row) and each lag (each column) for future lags.
- B is the coefficient matrix.
- **Z is a matrix with the past lags of the y variables.**
- **U is a matrix with the model errors.**

The Multivariate Least Squares allows you to compute the coefficients using the following matrix computation:

$$\widehat{B} = YZ'\left(ZZ'\right)^{-1}$$

The coefficients that are estimated this way are what underlie the model that you'll see later on.

One Multivariate Model vs. Multiple Univariate Models

Vector Autoregression should be applied to multiple target variables that are correlated. If there is no or very little correlation between the variables, they cannot benefit from being combined in one and the same model.

Besides using a VAR model only in case it makes sense to combine variables in one and the same model, it is even more important to use objective model evaluation techniques using a train-test-set and cross-validation. This can help you to make the right choice between using one model for multiple variables or using a separate univariate model for each variable.

An Example: VAR for Forecasting Walmart Sales

In this example, you will continue to work on the Walmart sales data that was presented in the previous chapter. Yet, in the current example, rather than summing the sales per week, you will sum the weekly data per store. As there are 45 stores in the dataset, this yields 45 weekly time series. You can create this data with a plot using Listing 10-1.

```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('train.csv')
data = data.pivot_table(index = 'Date', columns =
'Store', values = 'Weekly_Sales')

ax = data.plot(figsize=(20,15))
ax.legend([])
ax.set_ylabel('Sales')
plt.show()
```

Listing 10-1 Preparing the Walmart data per store

This code will give you the plot that is shown in Figure 10-1. As you can see in this plot, the data per store follow the same pattern. For example, you can see that they all peak at almost the same moment. This shows the interest in using multivariate time series.

Figure 10-1 The plot of the 45 stores' time series

Now let's get into the fit of a VAR model. As before, let's use a ten-step forecast on a ten-step test dataset. The code to create a VAR model can be seen in Listing 10-2.

```
from sklearn.metrics import
mean_absolute_percentage_error
from statsmodels.tsa.api import VAR
import mlflow

mlflow.autolog()

train = data.iloc[:-10,:]
test = data.iloc[-10:,:]

model = VAR(train)
results = model.fit(maxlags=2)
```

```
lag_order = results.k_ar
fcst = results.forecast(train.values[-lag_order:],
10)

model_accuracy = 1 -
mean_absolute_percentage_error(test, fcst)
print(model_accuracy)
```

Listing 10-2 Fitting the VAR model

There are a few interesting things to notice in this code, which are different from what you have seen before.

Firstly, note that the data on which the model is fit is a multivariate dataset: it uses all the columns at once. This is as could be expected from a multivariate time series model.

Secondly, note that rather than doing a grid search, the example shows the use of an argument called **maxlags**. This means that the VAR model will optimize the choice for the order of the model itself, all while respecting a maximum lag. In this case, the maxlags has been chosen at two, as going above two would require more coefficients to be estimated than possible using the current data.

The maxlags chooses the ideal order of the model using the **AIC**, the **Akaike Information Criterion**. The Akaike Information Criterion is a famous KPI for goodness-of-fit of a model. The AIC is based only on the training data. It is therefore more prone to inducing overfitted models.

AIC is more used in classical statistical models and less in modern machine learning. Yet, it is a fast and practical way of choosing order and, therefore, important to know. The negative aspect of VAR is that it requires a very large number of parameters to be estimated. This means that it **requires enormous amounts of data** to fit models with higher orders. In the current example, it is impossible to use an order higher than two.

Also, you should notice in the code that when applying the forecast method, it is necessary to give the last part of the training data. This is needed so that the model can compute the future by applying coefficients to the lagged variables.

The model accuracy obtained with the model is **0.89**.

Despite the low possibility of model tuning with this model, it must be said that the performance is relatively good. But as always, remember that

in practice, the only way to know whether this is good enough is by doing benchmarking and model comparison with other models.

Key Takeaways

- VAR model uses multivariate correlation to make one model for multiple target variables.
- The order of the VAR model, p, determines the number of time steps back that are used for predicting the future.
- The VAR model implementation can define the ideal number of lags using the maxlags parameter and the Akaike Information Criterion.
- The VAR model needs to estimate a large number of parameters, which means that it requires a huge amount of historical data. This makes it difficult to estimate higher lags.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_11

11. The VARMAX Model

Joos Korstanje[1]
(1) Maisons Alfort, France

You have discovered the VAR model in the previous chapter. And just like in univariate time series, there are some building blocks that can be built upon the VAR model to account for different types of processes. This can build up from the smaller and more common **VAR** to the more complex **VARMAX**.

It must be said that techniques for multivariate time series modeling are a part of the more advanced techniques. They do not always have Python implementations and are less often used than the techniques for univariate time series. While the use of the VAR model is still relatively common, the more advanced models in this branch become less and less documented on the internet.

In this chapter, you will discover the VARMAX model. It is the go-to model for multivariate time series. It adds a **Moving Average component** to the VAR model, and it can allow for external or exogenous variables as well. The components in the VARMAX model are therefore

- **V** for vector indicates to us that it's a multivariate model.
- **AR** for autoregression.
- **MA** for Moving Average.
- **X** for the use of exogenous variables (in addition to the endogenous variables).

It must be noted that the VARMAX model does *not* have a seasonal component. This is not necessarily a problem, as seasonality can be included through the *exogenous variables*. For example, a monthly

seasonality can be modeled by adding the exogenous variable month. For a weekly seasonality, you could add the variable week number.

Another missing block is the integration block. The VARMAX model does not include the differencing of non-stationary time series. As for the VAR model, all the input time series must already be *stationary*.

Model Definition

The mathematical model of the VARMAX definition is as follows:

$$y_t = v + A_1 y_{t-1} + \ldots + A_p y_{t-p} + Bx_t + \epsilon_t + M_1 \epsilon_{t-1} + \ldots + M_q \epsilon_{t-q}$$

In this model, $\boldsymbol{y_t}$ is a vector of the values of the present, and the other $\boldsymbol{y}$'s are the lagged values. The **A**'s are the autocorrelation coefficients: for each lag, they are a vector of the same length as the number of time series. The **M**'s are the vectors of coefficients for the lagged model errors. They represent the Moving Average part of the model.

As the VARMAX model is merely a combination of building blocks that you have seen throughout the previous seven chapters, I will spare you the repetition over here. Don't hesitate to go back to the previous chapters for more details on the intuition and explanations behind different building blocks. Let's now dive into the Python implementation and the example.

Multiple Time Series with Exogenous Variables

As a first step, let's use Listing 11-1 to prepare the Walmart data on a by-store and by-week basis. As you have seen in the previous chapter, this dataset lends itself perfectly to multivariate time series modeling.

```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('train.csv')
exog = data.groupby('Date')['IsHoliday'].sum() > 0
exog = exog.apply(lambda x: float(x))

data = data.pivot_table(index = 'Date', columns =
'Store', values = 'Weekly_Sales')
```

```
ax = data.plot(figsize=(20,15))
ax.legend([])
ax.set_ylabel('Sales')
plt.show()
```

Listing 11-1 Prepare the Walmart data for the VARMAX model

Note that in this listing, the exogenous data have been created as well. They are the indicators of the presence of holidays per week. They will be the only variable used as exogenous data in this example, but there is no theoretical restriction to use more exogenous variables.

The plot obtained by this code is shown in Figure 11-1. You can observe that the time series follows patterns that seem relatively correlated.

Figure 11-1 Plot of the time series per store

In the previous chapter, you have estimated the VAR using only the order p. This p determines the order of the AR component. In the VARMAX model, there are both an AR component and an MA component.

Therefore, there is an order for the AR part and for the MA part to be decided. The hyperparameters of VARMAX(p, q) are

- The order of the AR component, denoted p
- The order of the MA component, denoted q

Now, to estimate a VARMAX model, you may use the code shown in Listing 11-2. To avoid the model taking too many computation resources on your hardware, the code example uses the first three stores of the dataset.

```
import statsmodels.api as sm
from sklearn.metrics import
mean_absolute_percentage_error
import mlflow

mlflow.autolog()

train = data.iloc[:-10,[0,1,2]]
test = data.iloc[-10:,[0,1,2]]

train_exog = exog[:-10]
test_exog = exog[-10:]

mod = sm.tsa.VARMAX(train, order=(1,1),
exog=train_exog)
res = mod.fit(maxiter=100, disp=False)

fcst = res.forecast(exog=test_exog.values,
steps=10)
mape = mean_absolute_percentage_error(test, fcst)
model_accuracy = 1 - mape
print(model_accuracy)
```

Listing 11-2 Running the VARMAX(1,1) model

The VARMAX model, especially when applied to examples with a large number of variables, can be very long to run. What's taking so long here is the estimation of the Moving Average part of the model. MA models are known to be exceptionally slow to fit, and in the current case, there is the difficulty of having 45 variables to estimate it.

The R2 that was obtained on this example was **0.96**. This is a great score. This was obtained on a **ten-step forecast**. As you may remember from the chapter on the MA model, the Moving Average component is *not suited for multistep forecasts*.

With a model training time that is such long already, it would hardly be a solution to add repetitive retraining procedures to the model. The VARMAX model should be used only in cases where training times are not a problem or where the VARMAX obtains performance that cannot be obtained using alternative training methods.

It is great to have the more advanced modeling technique of VARMAX in your forecasting toolbox. The model can fit more complex processes than many other time series models. Yet, it also has its disadvantages: training times are relatively long compared to simpler models, and it needs a relatively large amount of data to estimate correctly.

As a side note, you will find that those training time and data availability requirements are almost unavoidable for any of the more complex time series and machine learning techniques. This can partly be countered by computing power, but it is still important to evaluate model choice critically.

Key Takeaways

- The VARMAX model consists of
  - V for vector: it is a multivariate model as it models multiple time series at the same time
  - AR for autoregression
  - MA for Moving Average
  - X for the addition of an exogenous variable
- The VARMAX(p,q) model takes two hyperparameters:
  - p for the order of the AR part
  - q for the order of the MA part
- The time series in a VARMAX have to be stationary.

Part IV
Supervised Models

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_12

12. The Linear Regression

Joos Korstanje[1]
(1) Maisons Alfort, France

In the following chapters, you will see the most common supervised machine learning models. As you'll remember from Chapter 1, supervised machine learning algorithms work differently from time series models.

In supervised machine learning models, you try to identify relations between different variables:

- *A target variable*: The variable that you try to forecast
- *Explanatory variables*: Variables that help you to predict the target variable

For forecasting, it is important to understand which types of explanatory variables you can or cannot use. As an example, let's say that the sales of hot chocolate strongly depend on the temperature. When the weather is cold, sales are high. When the weather is warm, sales are low.

You could make a model that regroups this basic if/else logic. Yet, when you think about it, this logic could not be used for forecasting in the future. After all, if you want to forecast tomorrow's sales of hot chocolate, you must know tomorrow's temperature. And this is not something you know: it would require an additional forecast of temperature!

Another example is to predict hot chocolate not based on the weather, but on the week number. You could try to identify a relationship with the week number based on past data. Then, to predict future sales, you would input the next week's week number into the model and obtain a forecast. This is possible here because the week number is something that you know for certain in advance.

This is important to keep in mind when doing supervised models in general. Now, let's get more into depth in the Linear Regression.

Linear Regression

The idea behind the Linear Regression is to define a linear relationship between a target variable and numerous explanatory variables to predict the target variable. Linear Regression is widely used, not only for forecasting. Like any supervised model, as long as you put explanatory variables of the future as input to the model, this works perfectly.

Model Definition

The Linear Regression is defined as follows:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p + \varepsilon$$

In this formula,

- There are **p** explanatory variables, called **x**.
- There is one target variable called **y**.
- The value for y is computed as a constant (β_0) plus the values of the x variables multiplied by their coefficients β_1 to β_p.

Figure 12-1 shows how to interpret B0 and B1 visually. It shows that for an increase of 1 in the x variable, the increase in the y variable represents β_1. $\beta_0$is the value for y when x is 0.

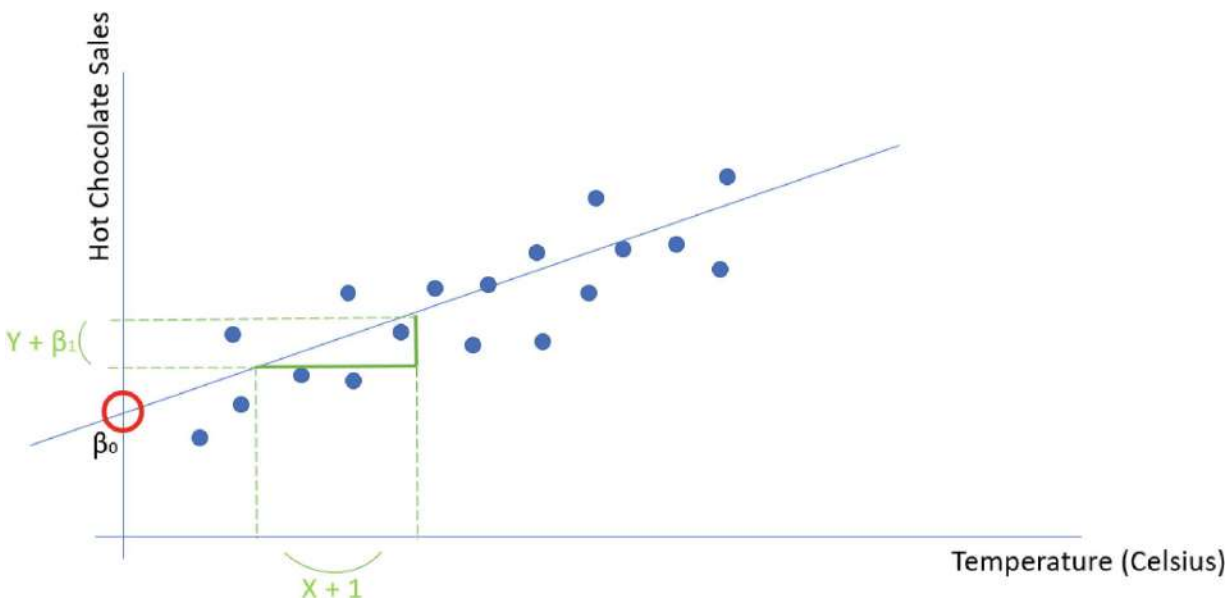


Figure 12-1 Visual interpretation of the Linear Regression

To be able to use the Linear Regression, you need to estimate the coefficients BETA on a training dataset. The coefficients can then be estimated using the following formula, in matrix notation:

$$\widehat{\beta} = \left(X^T X\right)^{-1} X^T y$$

This formula is known as **OLS**: the **Ordinary Least Squares method**. This model is very fast to fit, as it requires only matrix calculations to compute the betas. Although easy to fit, it is less suited for more complex processes. After all, it is a linear model, and it can therefore only fit linear processes.

A linear model can fit any type of relationship that goes in one direction. An example of this is: “if the temperature goes up, hot chocolate sales go down.” Yet, linear models cannot fit anything nonlinear. An example of a nonlinear process is this: “if the temperature is below zero, hot chocolate sales are low; if the temperature is between zero and ten, hot chocolate sales are high; if the temperature is high, hot chocolate sales are low.”

As you see, the second example is nonlinear because you could not draw a straight line from low to high hot chocolate sales. Rather, you could better make an if/else statement to capture this logic.

You should keep in mind that linear models are not very good at capturing nonlinear trends. Yet nonlinear models, when tuned correctly, can often approximate linear trends quite well. This is the reason that many of the more advanced machine learning techniques use a lot of nonlinear approaches. You will see this throughout the following chapters.

Example: Linear Model to Forecast CO_2 Levels

As an example for the linear model, you will work with the CO_2 dataset that you have already discovered in a previous chapter. This dataset has the advantage of being relatively easy to work with, and it will be a perfect case to show how to add **lagged variables, seasonality, and trend** into a supervised machine learning model.

Let's get started by importing the data into Python and plotting it. The code for this is shown in Listing 12-1.

```
import statsmodels.api as sm
import pandas as pd
import matplotlib.pyplot as plt

data = sm.datasets.co2.load_pandas()
co2 = data.data
co2 = co2.dropna()
ax = co2.plot()
ax.set_ylabel('CO2 level')
plt.show()
```

Listing 12-1 Importing the data and plotting it

Figure 12-2 shows the graph that you should obtain.

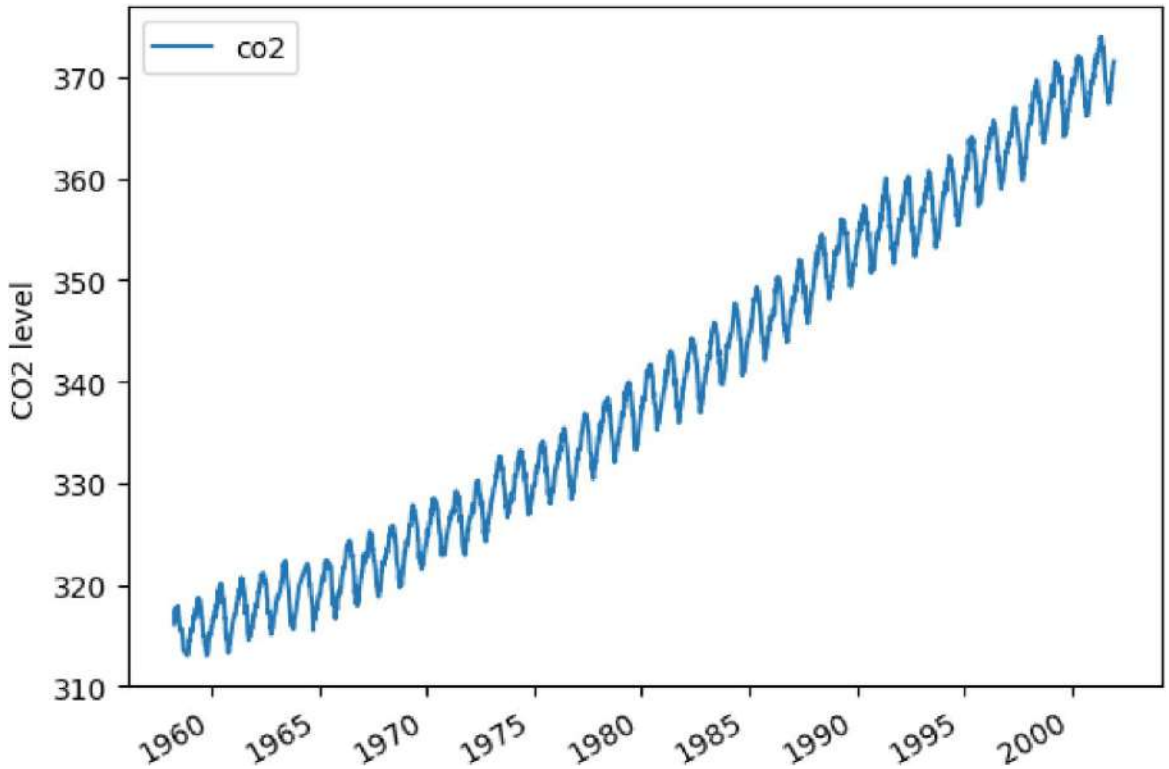


Figure 12-2 CO_2 levels over time

Now, you only have the dates and the CO_2 values. The interesting step here is to do **feature engineering**: creating additional variables based on the original variables. Even though there is very little information in this dataset, there are a lot of variables that you can create from it.

Let's start by extracting seasonal variables from the data variable. As we can see from the plot, there is a strong seasonal pattern going on. You could try to capture this by adding a monthly seasonality to the model. For this, it is necessary to create a variable month in your dataset. You can use Listing 12-2 to do this.

```
co2['month'] = pd.Series(co2['index']).apply(
                                lambda x: x.month
                                )
```

Listing 12-2 Creating the variable month

Now that you have this variable for monthly seasonality, let's see whether you can create a variable that captures the long-term upward trend. The solution to this problem is to add a variable year by extracting the year

from the data variable. As there is a yearly increase in the data, the trend effect could be captured by this variable. This is done in Listing 12-3.

```
co2['year'] = pd.Series(co2['index']).apply(lambda
x: x.year)
```

Listing 12-3 Creating the variable year

Now, for starters, let's just try to fit a Linear Regression with only those two explanatory variables, month and year. The package scikit-learn, which you have seen before, contains a large number of supervised models and will be used for this exercise. This basic model is created in Listing 12-4. Scikit-learn integrates well with MLflow. The code in Listing 12-4 also uses MLflow autologging to save the results to your local file system.

```
import mlflow
mlflow.autolog()

# Create X and y objects
X = co2[['year', 'month']]
y = co2['co2']

# Create Train test split
from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.20,
random_state=12345,shuffle=False)

# Fit model
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

my_lm = LinearRegression()
my_lm.fit(X = X_train, y = y_train)

train_fcst = my_lm.predict(X_train)
test_fcst = my_lm.predict(X_test)
```

```
train_r2 = r2_score(y_train, train_fcst)
test_r2 = r2_score(y_test, test_fcst)

print(train_r2, test_r2)

# Plot result
plt.plot(list(test_fcst))
plt.plot(list(y_test))
plt.xlabel('Steps into the test set')
plt.ylabel('CO2 levels')
plt.show()
```

Listing 12-4 Fitting a Linear Regression with two variables

The train R2 of this model is **0.96**, which is great. The test R2, however, is **0.34**, which is relatively bad. In Figure 12-3, you can see the fit on the test set and see that there is some improvement to be made.

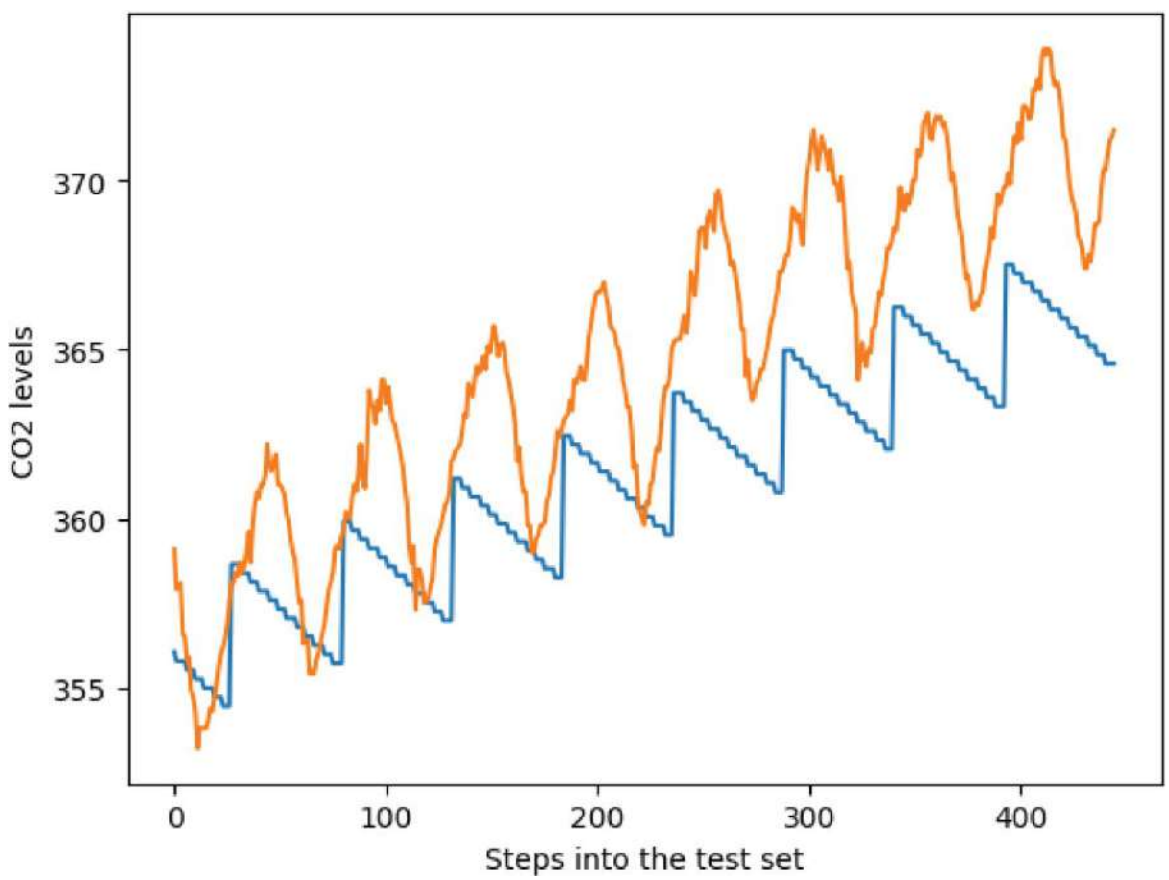


Figure 12-3 Predictive performance plot of the simple model

The next thing that you will add to the model is an autoregressive component. This will be done by feature engineering. As always in supervised models, you need to extract any information as explanatory variables. For this example, let's see how to use the shift method to create a lagged variable. You can, for example, add five lagged variables easily using Listing 12-5.

```
co2['co2_l1'] = co2['co2'].shift(1)
co2['co2_l2'] = co2['co2'].shift(2)
co2['co2_l3'] = co2['co2'].shift(3)
co2['co2_l4'] = co2['co2'].shift(4)
co2['co2_l5'] = co2['co2'].shift(5)
```

Listing 12-5 Adding lagged variables into the data

Note that adding lagged variables creates NA in the dataset. This is a border effect, and it is not a problem. Simply delete any missing data with Listing 12-6.

```
co2 = co2.dropna()
```

Listing 12-6 Drop missing values

As a final step, let's fit and evaluate the model with the monthly seasonality, the yearly trend, and the five autoregressive lagged variables. You must note here that since we added the lagged values into the train set, it would be impossible to do this for multiple steps forward. You calculate the first future value using the data of today. You calculate the second future value by using the data from tomorrow. Therefore, the error that you evaluate here should be interpreted as a one-step forecasting error, whereas the previous code block did a multistep forecast. This is done in Listing 12-7.

```
# Create X and y objects
X = co2[['year', 'month', 'co2_l1', 'co2_l2',
'co2_l3', 'co2_l4', 'co2_l5']]
y = co2['co2']

# Train Test Split
```

```
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.20,
random_state=12345,shuffle=False)

# Fit the model
my_lm = LinearRegression()
my_lm.fit(X = X_train, y = y_train)

train_fcst = my_lm.predict(X_train)
test_fcst = my_lm.predict(X_test)

train_r2 = r2_score(y_train, train_fcst)
test_r2 = r2_score(y_test, test_fcst)

print(train_r2, test_r2)

# Plot result
plt.plot(list(test_fcst))
plt.plot(list(y_test))
plt.xlabel('Steps into the test set')
plt.ylabel('CO2 levels')
plt.show()
```

Listing 12-7 Fitting the full Linear Regression model

The train R2 score of this model is **0.998**, and the test R2 score is **0.990**. This is a great performance! You can see in the plot (Figure 12-4) that the predictions follow the actual values almost perfectly. You'll also see that the run has been logged automatically by MLflow and that you have the r2 train and r2 test in the mlruns metrics directory. The entire model has also been stored, which allows you to use it later on without running the retraining.

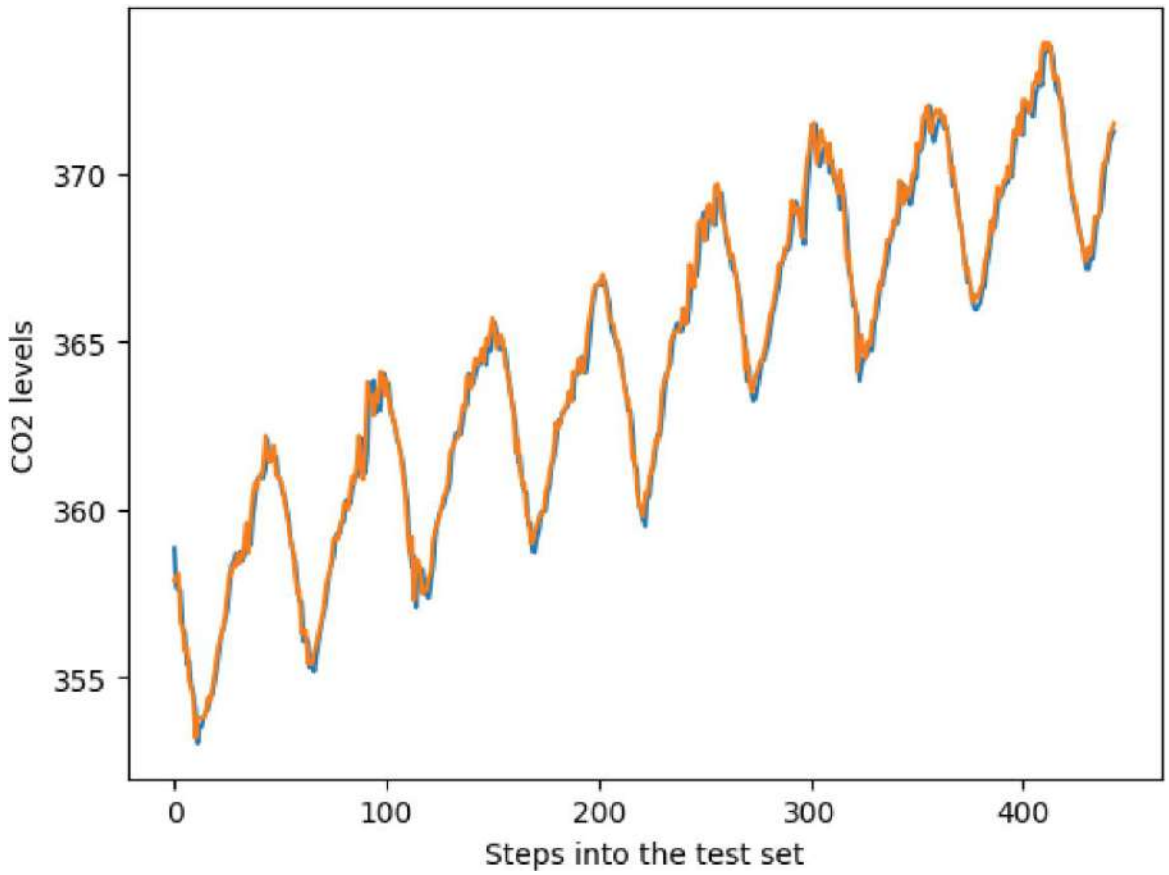


Figure 12-4 Predictive performance plot of the full model

In conclusion, you have seen how to add time series trends into a supervised model. You have used a linear model to obtain this result. In the following chapters, you'll discover how to apply more and more complex models in the same way.

Key Takeaways

- The linear model is the simplest supervised machine learning model.
- The linear model finds the best linear combination of external variables to forecast the future.
- Linear models cannot easily adapt to nonlinear situations.
- Feature engineering is the task of creating the best possible variables in your dataset, in order to obtain explanatory variables that can help your model obtain the best performance.
- Seasonality can be fitted by supervised models when you introduce a seasonal variable.

- A trend can be fitted by supervised models when you introduce a trend variable
- Autoregression can be fitted by supervised models when you add lagged versions of the target variable into the explanatory variables.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_13

13. The Decision Tree Model

Joos Korstanje[1]
(1) Maisons Alfort, France

As you've discovered in the previous chapter, there is a distinction in supervised machine learning models between linear and nonlinear models. In this chapter, you will discover the **Decision Tree** model. It is one of the simplest nonlinear machine learning models.

The idea behind the Decision Tree model can be intuitively understood as a long list of *if/else statements*. Those if/else decisions would be used at the prediction stage: the model predicts some **result x** if a certain condition is true, and it will **predict y** otherwise. As you see, there is no linear trend in this type of logic, and because of this, *a decision tree can fit nonlinear trends*.

Let's see an example of this decision tree logic in practice, without yet discussing where the decision tree comes from. Figure 13-1 shows an example decision tree that can help you to understand intuitively how a decision tree could work. It shows a decision tree that forecasts the average rainfall depending on climate zone and season.

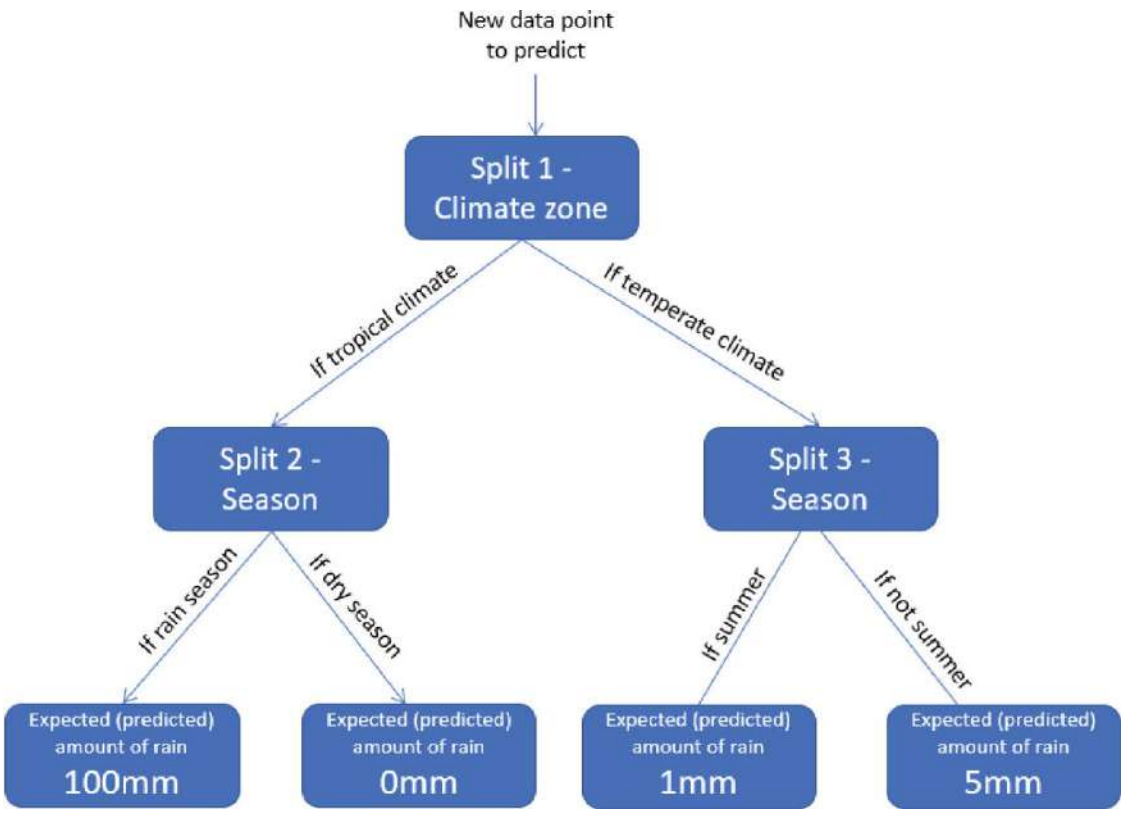


Figure 13-1 An intuitive example of a decision tree example

Mathematics

You'll now discover the mathematical and algorithmic components of the decision tree algorithm. There are different ways to fit decision trees, but the most common one is to have two steps: splitting (also called growing) the tree and pruning the tree.

Splitting

Now that you have seen that the decision tree is merely a hierarchical ordination of multiple decision splits, the logical question is where those decisions come from.

It all starts with a dataset in which you have a target variable and multiple explanatory variables. Now, a huge number of splits are possible, but you need to choose one. The first split should be the split that would obtain **the lowest Mean Squared Error**. In the end, even a model with only one split would be able to be used as a predictive model.

Now that you know the goal of your split, the only thing left to do is to test each possible split and evaluate which of them results in the lowest

Mean Squared Error. Once you've identified the best first split, you obtain two groups. You then repeat this procedure for each of the two groups, so that you then obtain four groups and so forth.

At some point, you cannot split any further. This point is whenever there is only one data point in each group. Logically, a single data point cannot be split.

Pruning and Reducing Complexity

Although it is possible to let the tree continue splitting until further splitting is impossible, it is not necessarily the best thing to do. There is also a possibility to allow having a bit more data points in a group to avoid overfitting.

This can be obtained in multiple ways. Some implementations allow a pruning process, which first forces a tree to split everything completely and then add the pruning phase to cut off those branches that are the least needed.

Other implementations let you simply add a complexity parameter that will directly avoid the trees from becoming too detailed. This is a parameter that is a great choice for optimizing using grid search.

Example

In this chapter, you'll be working with data that come from a bike-sharing company. You can download the data from the UCI machine learning archives over here: https://archive.ics.uci.edu/ml/datasets/bike+sharing+dataset.

This will be a difficult example. In this dataset, you'll be predicting the number of rental bike users per day. And as you can imagine, this will depend strongly on the weather. As I explained in the previous chapter, the weather is hard to predict, and if your forecast has a strong correlation with the weather, you know it will be a big challenge.

Let's first have a look at the data to know what you're working with here. This is done in Listing 13-1 and will obtain the plot in Figure 13-1.

```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('bikedata/day.csv')
```

```
ax = data['cnt'].plot()
ax.set_ylabel('Number of users')
ax.set_xlabel('Time')
plt.show()
```

Listing 13-1 Import the bike data

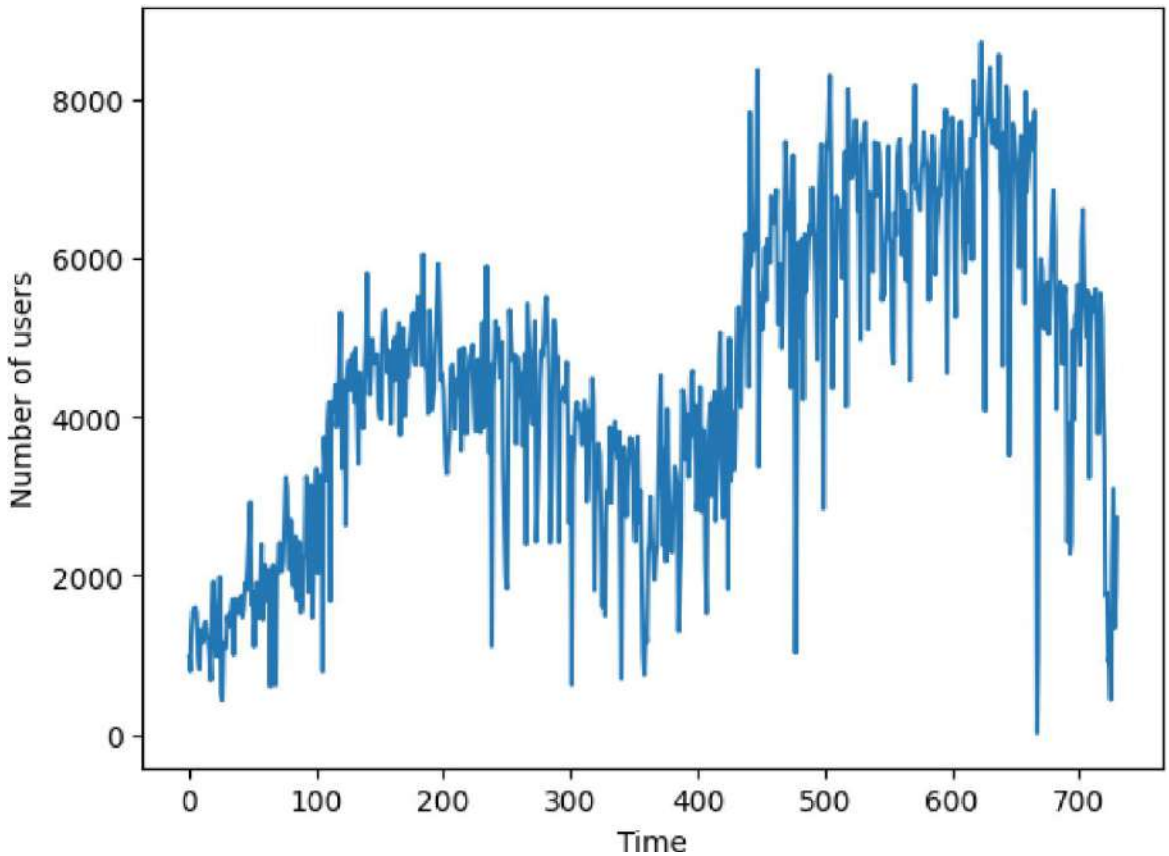


Figure 13-2 Plot of the bike-sharing users

As you can see in the plot, many things are going on here. There is a difficult trend to be estimated. There is also a lot of day-to-day variation. When you look at the dataset, you can see that there are a lot of data on weather. This is going to be the challenge: the number of bike-sharing users depends strongly on the day's weather, but the weather is not something that you can know in advance.

The only thing you can do in such cases is to try and do a lot of feature engineering to create many valuable variables for the model. In the code block in Listing 13-2, you will use some of the existing variables, and you'll see how to create a number of variables. The total list of explanatory values is

- Original variable ‘season’: (1:spring, 2:summer, 3:fall, 4:winter)
- Original variable ‘yr’: The year
- Original variable ‘mnth’: The month
- Original variable ‘holiday’: Whether the day is a holiday
- Original variable ‘weekday’: The day of the week
- Original variable ‘workingday’: Whether the day is a holiday/weekend or a working day
- The seven last days of ‘cnt’: An autoregressive component for the number of users
- The seven last days of ‘weathersit’: The weather, from 4 (very bad weather) to 1 (good weather)
- The seven last days of “temperature”
- The seven last days of “humidity”

You will also **delete an influential outlier value** in the dataset. This is an extreme observation, of which it is difficult to understand why it happened. You can see it in Figure 13-2, as a very low peak occurring at index 477. As it can influence the model negatively, it is best to not include it in the training data.

There is another low peak somewhere between 650 and 700, but as this part of the data will be our test set, it would be unfair to do a treatment to it. **Removing outliers from the test set would be cheating,** as this would make our test score higher than reality. The goal is always to have a model error estimate that is as reliable as possible, to anticipate its behavior when applying the model in practice.

```
# 7 last days of user count (autoregression)
data['usersL1'] = data['cnt'].shift(1)
data['usersL2'] = data['cnt'].shift(2)
data['usersL3'] = data['cnt'].shift(3)
data['usersL4'] = data['cnt'].shift(4)
data['usersL5'] = data['cnt'].shift(5)
data['usersL6'] = data['cnt'].shift(6)
data['usersL7'] = data['cnt'].shift(7)

# 7 last days of weathersit
data['weatherL1'] = data['weathersit'].shift(1)
data['weatherL2'] = data['weathersit'].shift(2)
```

```
data['weatherL3'] = data['weathersit'].shift(3)
data['weatherL4'] = data['weathersit'].shift(4)
data['weatherL5'] = data['weathersit'].shift(5)
data['weatherL6'] = data['weathersit'].shift(6)
data['weatherL7'] = data['weathersit'].shift(7)

# 7 last days of temperature
data['tempL1'] = data['temp'].shift(1)
data['tempL2'] = data['temp'].shift(2)
data['tempL3'] = data['temp'].shift(3)
data['tempL4'] = data['temp'].shift(4)
data['tempL5'] = data['temp'].shift(5)
data['tempL6'] = data['temp'].shift(6)
data['tempL7'] = data['temp'].shift(7)

# 7 last days of humidity
data['humL1'] = data['hum'].shift(1)
data['humL2'] = data['hum'].shift(2)
data['humL3'] = data['hum'].shift(3)
data['humL4'] = data['hum'].shift(4)
data['humL5'] = data['hum'].shift(5)
data['humL6'] = data['hum'].shift(6)
data['humL7'] = data['hum'].shift(7)

data = data.dropna()
data = data.drop(477)

X = data[['season', 'yr', 'mnth', 'holiday',
'weekday', 'workingday',
          'weatherL1', 'weatherL2', 'weatherL3',
'weatherL4', 'weatherL5', 'weatherL6', 'weatherL7',
         'usersL1','usersL2', 'usersL3', 'usersL4',
'usersL5', 'usersL6', 'usersL7',
         'tempL1', 'tempL2', 'tempL3', 'tempL4',
'tempL5', 'tempL6', 'tempL7',
         'humL1', 'humL2','humL3', 'humL4',
'humL5', 'humL6', 'humL7']]
```

```
y = data['cnt']
```

Listing 13-2 Creating the training dataset

Now, let's move on to the model building. To get started, let's do a model without any hyperparameter tuning. The example is shown in Listing 13-3.

```
import mlflow
mlflow.autolog()

# Create Train test split
from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.20,
random_state=12345, shuffle=False)

from sklearn.tree import DecisionTreeRegressor
my_dt = DecisionTreeRegressor(random_state=12345)
my_dt.fit(X_train, y_train)

from sklearn.metrics import
mean_absolute_percentage_r2_score
r2_score(list(y_test), list(my_dt.predict(X_test)))
```

Listing 13-3 Fitting the model

This first, unoptimized, model makes for an R2 of the test set of **0.165**. The metric has also been saved in your MLflow directory. This score is not great yet, so let's see what can be done using a grid search. In the GridSearch in Listing 13-4, we'll be tuning a few hyperparameters:

- min_samples_split: The minimum number of samples required to split a node. Higher values make for fewer splits. Having fewer splits makes a tree less specific, so this can help to prevent overfitting.
- max_features: The number of features (variables) to consider when searching for the best split. When using fewer features, you force the splits to be different from each other, but at the same time, you can be blocking the tree from finding the split that is actually the best split.
- Error criterion: As we're optimizing our R2, we would potentially want to use the R2 as an error criterion for the decision of the best splits. Yet it is

not available. It can therefore be interesting to try the MSE and the MAE and see which one allows us to optimize the R2 of the complete tree.

There are many more hyperparameters in the DecisionTreeRegressor. You can refer to the scikit-learn page for the DecisionTreeRegressor to find them. Don't hesitate to try out and play with those different parameters.

```
from sklearn.model_selection import GridSearchCV

my_dt =
GridSearchCV(DecisionTreeRegressor(random_state=44),
                 {'min_samples_split':
list(range(20,50, 2)),
                 'max_features': [0.6, 0.7, 0.8,
0.9, 1.],
                 'criterion': ['squared_error',
'absolute_error']},
                 scoring = 'r2', n_jobs = -1)

my_dt.fit(X_train, y_train)
print(r2_score(list(y_test),
list(my_dt.predict(X_test))))
```

Listing 13-4 Adding a grid search

When running this in the example notebook, this created an R2 score of **0.55**. Although this is still not amazing, it is a large improvement from the previous model. To find out which parameters have been chosen, you can get the best parameters using the code in Listing 13-5. As you have created a number of metrics using this GridSearch, you can also browse through the MLflow interface using `!mlflow ui` and going to `localhost:5000` with your preferred internet browser.

```
print(my_dt.best_estimator_)
```

Listing 13-5 Finding the best parameters

This will show you the best parameters as follows:

```
DecisionTreeRegressor(criterion='avsolute_error',
max_features=0.8,
```

```
min_samples_split=48,random_state=44)
```

Apparently, this combination of error criterion **MAE rather than MSE**, a **max_features of 0.8**, and a **min_samples_split of 48** is the model with the best predictive performance. Now, let's make a forecast with this model and plot how well it fits, using Listing 13-6.

```
fcst = my_dt.predict(X_test)

plt.plot(list(fcst))
plt.plot(list(y_test))
plt.ylabel('Sales')
plt.xlabel('Time')
plt.show()
```

Listing 13-6 Plotting the prediction

The plot that you'll obtain is the plot shown in Figure 13-3.

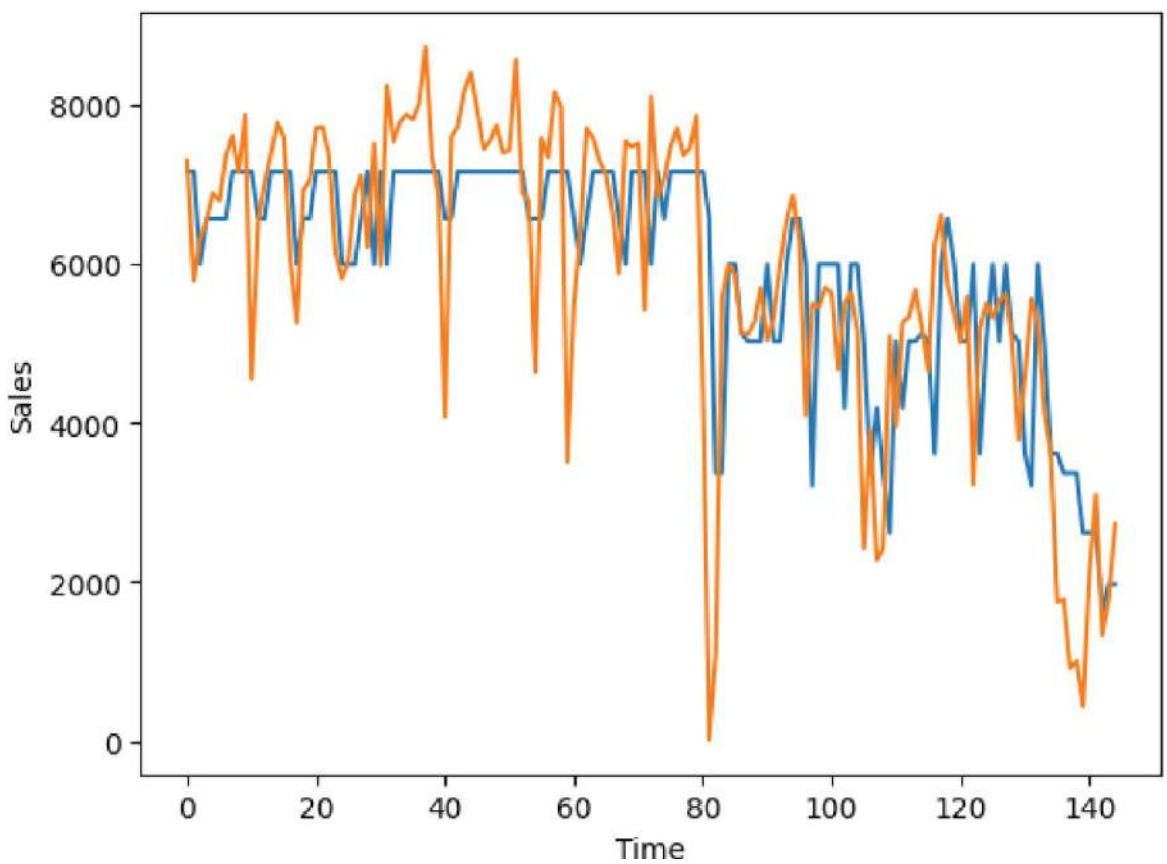


Figure 13-3 Plot of the bike-sharing users forecast

A last thing that is very practical with the Decision Tree model is that you can obtain a plot that shows you the different splits that have been identified. This type of plot is sometimes also referred to as a **dendrogram**. Despite the lower performances of a Decision Tree model, one of its strong points is that it allows for interpretation in detail of the decisions of the model.

```
from sklearn.tree import plot_tree
plot_tree(my_dt.best_estimator_, max_depth=1)
plt.show()
```

Listing 13-7 Plotting the prediction

The code in Listing 13-7 will generate the beginning of the decision tree, as shown in Figure 13-4. You can make larger extractions, by increasing the max_depth of the plot.

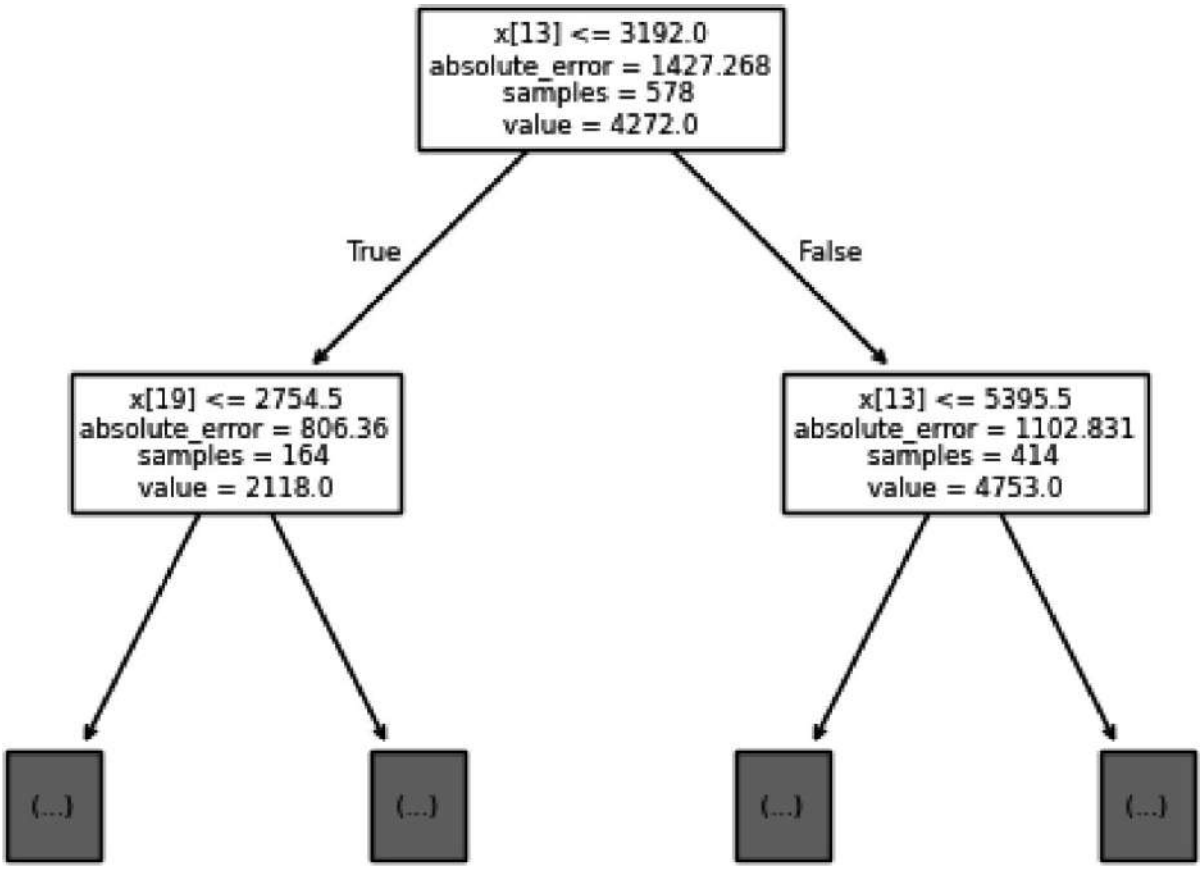


Figure 13-4 Plot the decision tree

This type of plot shows one of the real added values of using simpler models like the Decision Tree. They allow for an interpretation of the model, and this may be required in many circumstances. Models that you'll see in the next chapters will become less and less interpretable.

Key Takeaways

- The Decision Tree is one of the simplest nonlinear supervised machine learning models.
- You can tune the complexity of the decision tree, which defines how long and complex a decision tree becomes.
- Complex trees risk overfitting: they learn too detailed patterns in the training data. You can use grid search to optimize the complexity of the tree.
- You can create a dendrogram of the fitted Decision Tree to obtain all the splits and variables that have been used by your model.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_14

14. The kNN Model

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, you will discover the **kNN model**. The kNN model is the third supervised machine learning model that is covered in this book. Like the two previous chapters, the kNN model is also one of the simpler models. It is also intuitively easy to understand how the model works. As a downside, sometimes it is not performant enough to compete with the more advanced machine learning models that you will see in the following chapters.

Intuitive Explanation

So, what is the kNN model all about? The intuitive idea behind it is simple: you use the data points closest to a new data point to predict it. How could you best predict tomorrow's weather? Look at today's weather! And to predict next month's sales, it is probably reasonably close to this month's sales.

Yet, it gets more advanced when considering that a nearest data point can be in multiple dimensions. For example, if next month's sales are in December, the data point could be closest to another month of December in the dimension month.

As soon as you have multiple variables, this idea of distance becomes a very powerful concept for predicting new values. In short, the kNN model tries to find the **nearest neighbors** to a data point and uses their value as a prediction.

Mathematical Definition of Nearest Neighbors

The definition of nearest neighbors is based on the computation of the **Euclidean distance** from the new data point to each of the existing data points. The Euclidean distance is the most common distance measure. You probably use it on a daily basis when talking about your distances from home to work, etc. You can express it mathematically as follows:

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

In this formula, **p** and **q** are two data points with each n dimensions. This would generally mean that there are **n** explanatory variables in the dataset. You sum the squared distances for each dimension and finally take the square root.

The letter **k** is used to indicate the number of neighbors to use. To compute the *k nearest neighbors*, you simply compute the distance between your new data point and each of the data points in the training data. Depending on which number you have for k, you take the k data points that have the lowest distance.

The graph in Figure 14-1 shows how the algorithm works in a two-dimensional situation. All the data points in the training dataset are available to be chosen as a neighbor for any new point. As a dummy example, let's imagine it's again about hot chocolate sales based on the temperature.

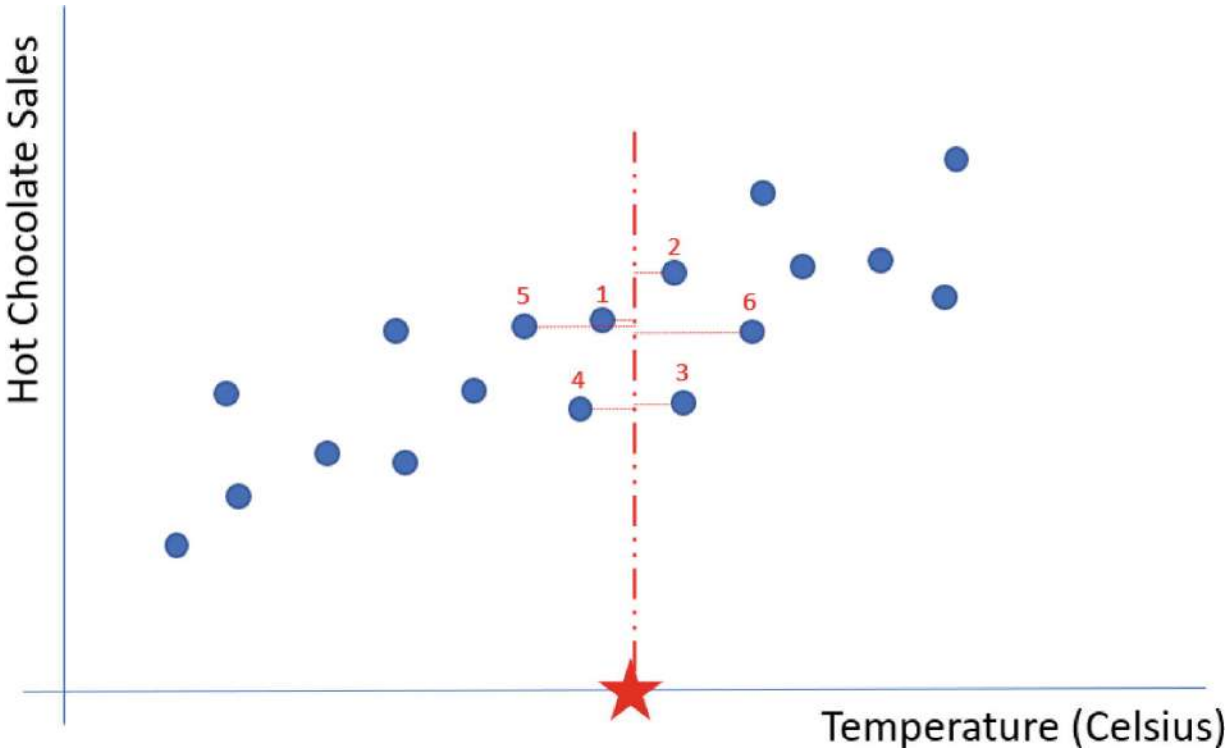


Figure 14-1 The six nearest neighbors of a new data point

The blue points in the graph are real observations from the past, in which we observed temperature and hot chocolate prices. Let's say we already know tomorrow's temperature and we want to predict the number of hot chocolate sales for tomorrow.

If we use the kNN algorithm for this, we need to identify the closest neighbors: in the graph, they are annotated based on distance. The notion of nearest, in this case, is based only on one variable: temperature. In cases with more variables, this would be the same computation using the formula stated earlier, yet it would be hard to visualize such a multivariate situation.

Combining k Neighbors into One Forecast

Once you have identified the **k neighbors** which are closest to your new data point, you do not yet have a prediction. There is one step remaining to convert the multiple neighbors into one prediction. There are two prevalent methods for it.

1. The first method is simply to take **the average** of the target value of the k nearest neighbors. This average is then used as the prediction.

2. The second method is to take **the weighted average** of the k nearest neighbors and use their distances as the inverse weight so that closer points are weighted heavier in the prediction.

Deciding on the Number of Neighbors k

A last thing remains to decide, and that is how many nearest neighbors you want to include in the prediction. This is decided by the value of k. To apply this to the previous example, let's see two different cases: 1-nearest neighbor and 3-nearest neighbors, and see what the difference in prediction is. The two are given in Figure 14-2.

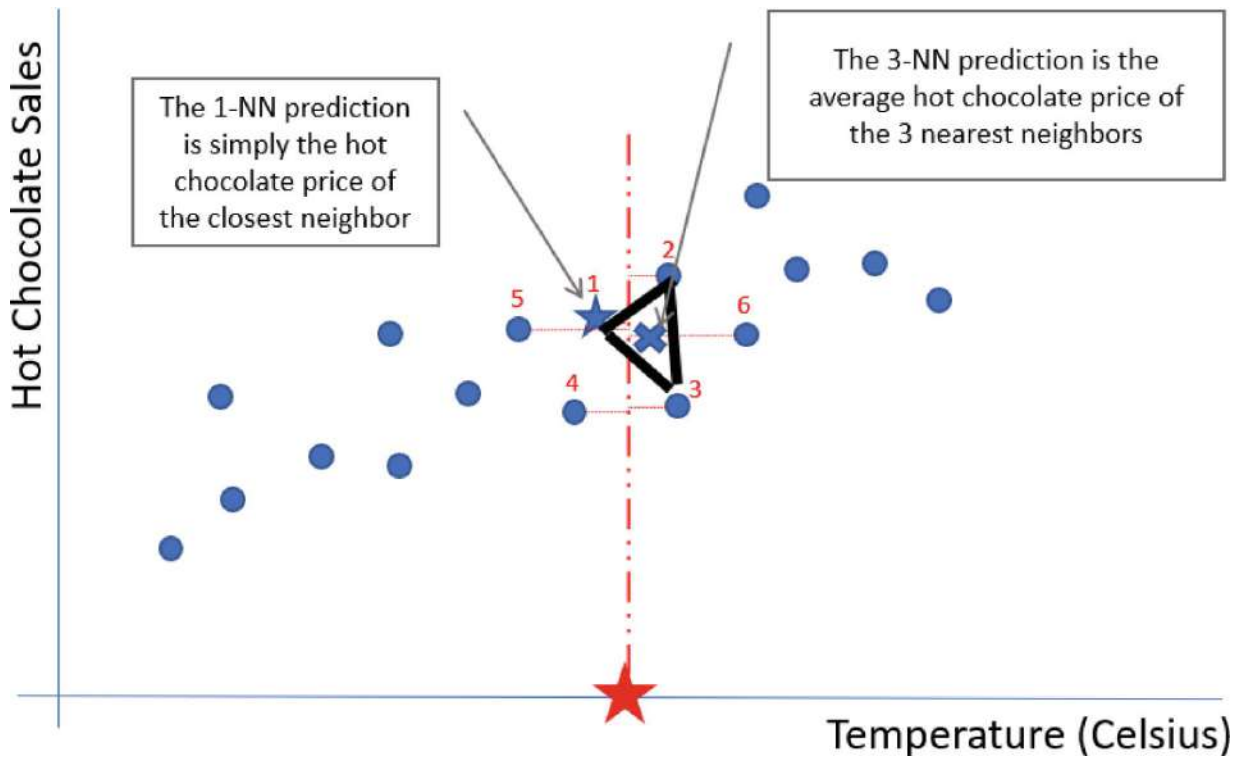


Figure 14-2 Two alternative predictions with a different number of neighbors used

The number of neighbors to use, or *k*, is a hyperparameter in the kNN model, and it is best optimized using hyperparameter tuning. The method that you've seen in this book is grid search CV, which is one of the go-to methods for hyperparameter tuning. In this chapter, you'll also discover an

alternative method: Random Search. Let's first introduce an example to make it more applied.

Predicting Traffic Using kNN

For this example, you'll be working with a dataset of hourly traffic volumes from Interstate 94. You can find this dataset on the UCI machine learning repository: https://archive.ics.uci.edu/ml/datasets/Metro+Interstate+Traffic+Volume.

If you download this data, you'll find that it has hourly traffic volume, together with some weather data and information about holidays. For the present example, let's avoid depending on weather data and do a forecast based on seasonality and holidays. After all, we could imagine that traffic depends heavily on the time of day, weekdays, and holidays.

To import the data into Python, you can use the code in Listing 14-1.

```
import pandas as pd
data =
pd.read_csv(Metro_Interstate_Traffic_Volume.csv')
```

Listing 14-1 Import the traffic data

After you've imported this data, let's create the seasonality variables that seem necessary for the modeling exercise for this example. Use the code in Listing 14-2 to create the following variables:

- Year
- Month
- Weekday
- Hour
- IsHoliday

```
data['year'] = data['date_time'].apply(lambda x:
x[:4])
data['month'] = data['date_time'].apply(lambda x:
x[5:7])
data['weekday'] =
pd.to_datetime(data['date_time']).apply(lambda x:
```

```
x.weekday())
data['hour'] =
pd.to_datetime(data['date_time']).apply(lambda x:
x.hour)
data['isholiday'] = (data['holiday'] ==
'None').apply(float)
```

Listing 14-2 Feature engineering to create the additional explanatory variables

The next step is to split the data into train and test and fit a default kNN model. This will give you a first feel of the R2 that could be obtained with this type of model. This code is shown in Listing 14-3.

```
import mlflow
mlflow.autolog()

# Create objects X and y
X = data[['year', 'month', 'weekday', 'hour',
'isholiday']]
y = data['traffic_volume']

# Create Train test split
from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=100,
random_state=12345, shuffle=False)

from sklearn.neighbors import KNeighborsRegressor
my_dt = KNeighborsRegressor()
my_dt.fit(X_train, y_train)

fcst = my_dt.predict(X_test)

from sklearn.metrics import r2_score
r2_score(list(y_test), list(fcst))
```

Listing 14-3 Creating the train test split and computing the R2 of the default model

As a positive surprise, the R2 score on this model is very good already: **0.9706**. This should mean that the forecast is not far from perfect, so let's try to verify this visually using the code in Listing 14-4.

```
import matplotlib.pyplot as plt
plt.figure(figsize=(20,20))
plt.plot(list(y_test))
plt.plot(list(fcst))
plt.legend(['actuals', 'forecast'])
plt.ylabel('Traffic Volume')
plt.xlabel('Steps in test data')
plt.show()
```

Listing 14-4 Creating a plot on the data of the test set

This code will generate the graph that is shown in Figure 14-3.

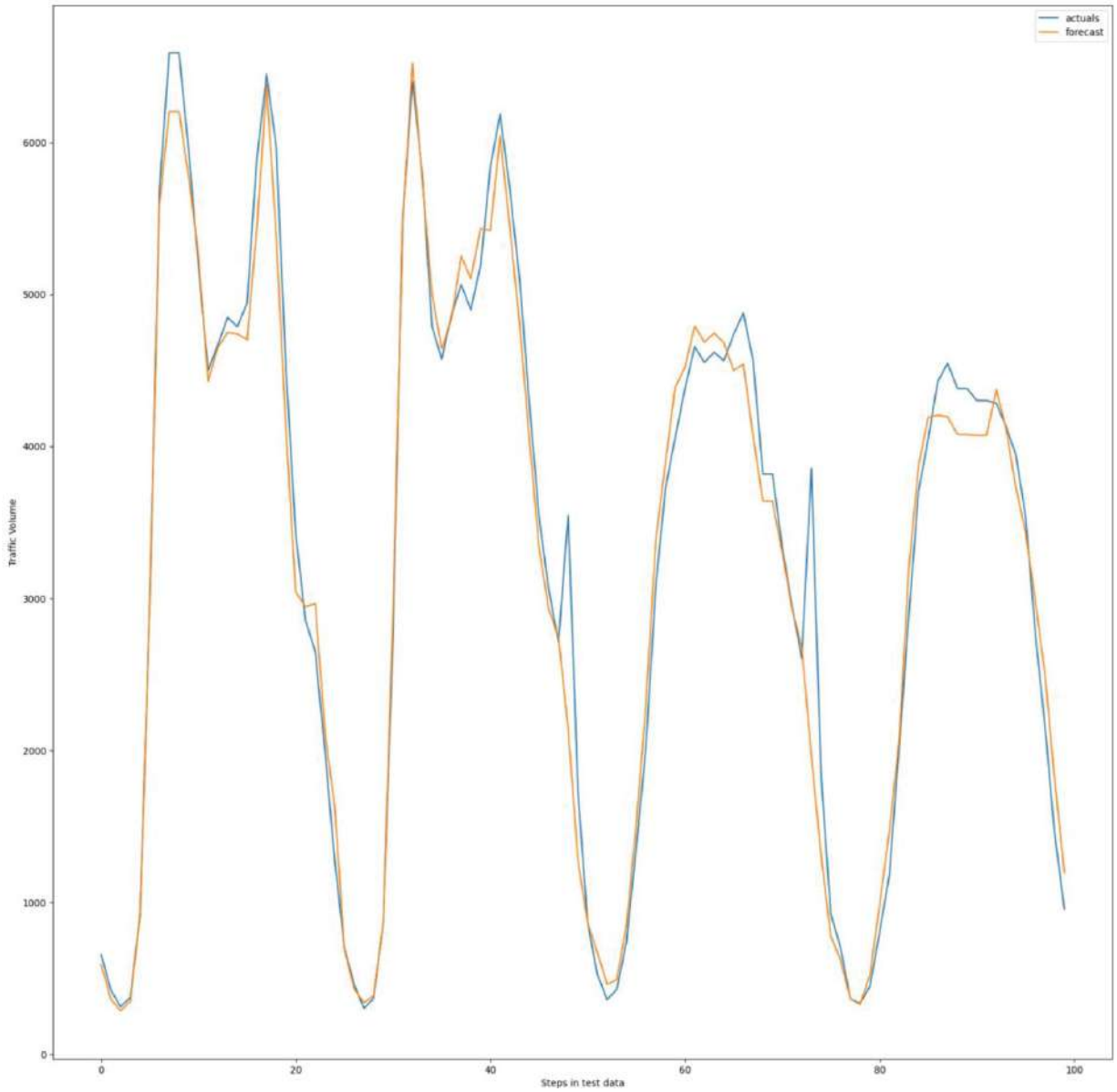


Figure 14-3 The plot shows the predictive performance on the test dataset

---

Grid Search on kNN

The number of neighbors that the scikit-learn implementation uses by default is five. Now, as a last step, let's try to add a grid search to see whether a different number of neighbors might get us a better performance. Let's test the number of neighbors by increments of 2, in order to speed it up a little. This is done in Listing 14-5.

```
from sklearn.model_selection import GridSearchCV
```

```
my_knn = GridSearchCV(KNeighborsRegressor(),
                 {'n_neighbors':[2, 4, 6, 8, 10,
12]},
                scoring = 'r2', n_jobs = -1)

my_knn.fit(X_train, y_train)
print(r2_score(list(y_test),
list(my_knn.predict(X_test))))
print(my_knn.best_estimator_)
```

Listing 14-5 Adding a grid search cross-validation to the kNN model

The R2 score obtained with this tuned model is **0.9724**. This model uses eight nearest neighbors rather than five, and it is better by 0.008 points on the R2 score. This is very little, and therefore, the improvement can hardly be considered significant. In this case, the default model was actually quite good from the start.

Random Search: An Alternative to Grid Search

As a last topic for this chapter, I want to introduce an alternative to the GridSearch. This alternative is called Random Search. Random Search is very easy to understand intuitively: rather than checking every combination of hyperparameters on the grid, it will check a random selection of hyperparameters.

The main reason for this is that it is much faster. Also, it has been found that Random Search is generally able to obtain results that are very close to grid search, and the speed improvement is therefore totally worth it. Listing 14-6 shows how to replace a grid search with a Random Search.

```
from sklearn.model_selection import
RandomizedSearchCV

my_knn = RandomizedSearchCV(KNeighborsRegressor(),
                 {'n_neighbors':list(range(1,
20))},
                scoring = 'r2', n_iter=10, n_jobs
= -1)
```

```
my_knn.fit(X_train, y_train)
print(r2_score(list(y_test),
list(my_knn.predict(X_test))))
print(my_knn.best_estimator_)
```

Listing 14-6 Adding a Random Search cross-validation to the kNN model

The Random Search has been allowed to choose any number for k between 1 and 20. The argument **n_iter** decides on the number of values that should be **randomly selected** within this range. Note that the selection is random, so this may give a different result than the grid search. There is no way to guarantee that a certain value will be included in the test.

To make a fair comparison with the grid search, we apply an n_iter of 6. As there are six values tested in the grid search, this makes it equivalent. When executing the code, the returned solution is exactly the same as the one returned by grid search.

Key Takeaways

- The kNN model bases its predictions on the values of its nearest neighbors.
- The neighbors' values are combined into a prediction by computing the average or a weighted average based on their distance.
- Euclidean distance is generally used for the distance measure.
- The number of neighbors to use in the combination is denoted by k and is a hyperparameter of the model.
- The value of k can be optimized using hyperparameter tuning, for example, through grid search cross-validation.
- Random Search is an alternative to grid search that is faster and generally gives results that are (almost) as good.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_15

15. The Random Forest

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, you will discover the **Random Forest model**. It is an easy-to-use model, and it is known to be very performant. The Random Forest and the XGBoost model that you will discover in the next chapter are two of the most used machine learning algorithms in modern applications.

A large number of variants on Random Forests and XGBoost exist on the market, yet if you understand the two basics, it will be relatively easy to adapt to any variant.

Intuitive Idea Behind Random Forests

The Random Forest is *strongly based on the Decision Tree model* but adds more complexity to it. As the name suggests, a Random Forest consists of a large number of Decision Trees, with each of them a slight variation.

A Random Forest is much more performant than a Decision Tree. Generally, a Random Forest can combine hundreds or even thousands of Decision Tree models. They will be fitted on slightly different data, so as to not be totally equal. So, in short, it's a large number of Decision Trees making a prediction that should be close to each other, yet not exactly the same.

Where one machine learning model can sometimes be wrong, the average prediction of a large number of machine learning models is less likely to be wrong. This idea is the foundation of **ensemble learning**.

In the Random Forest, ensemble learning is applied to a repetition of many Decision Trees. Ensemble learning can be applied to any combination

of a large number of Machine Learning models. The reason to use Decision Trees is that it has been proven a performant and easy-to-configure model.

Random Forest Concept 1: Ensemble Learning

So, how does ensemble learning work exactly? You can understand that having 1000 times the exact same Decision Tree does not have any added value to just using 1 time this Decision Tree.

In the ensemble model, each of the individual models has to be slightly different from each other. There are two famous methods for creating ensembles: **bagging** and **boosting**.

The Random Forest uses bagging to create an ensemble of Decision Trees. In the next chapter, you'll discover the XGBoost algorithm, which uses the alternative technique called boosting.

Let's discover the idea behind bagging. Bagging is short for Bootstrap Aggregation. The idea exists of two parts:

1. Bootstrap
2. Aggregation

Bagging Concept 1: Bootstrap

The **bootstrap** is one of the essential parts of the algorithm that makes sure that each of the individual learners fits a slightly different Decision Tree. Bootstrapping means that for each individual learner, the data set is created by a **resampling process**.

Resampling means creating a new data set based on the original data set. The new data set will be created by randomly selecting data points from the original data set. Yet, the important thing to realize here is that the resampling is done **with replacement**.

Resampling with replacement puts every sampled data point back into the mother population. This makes it possible for a single data point in the original data set to be selected multiple times in the bootstrapped data set. There will also be data points in the original data that are not selected for the bootstrapped data set. A schematic drawing is shown in Figure 15-1.

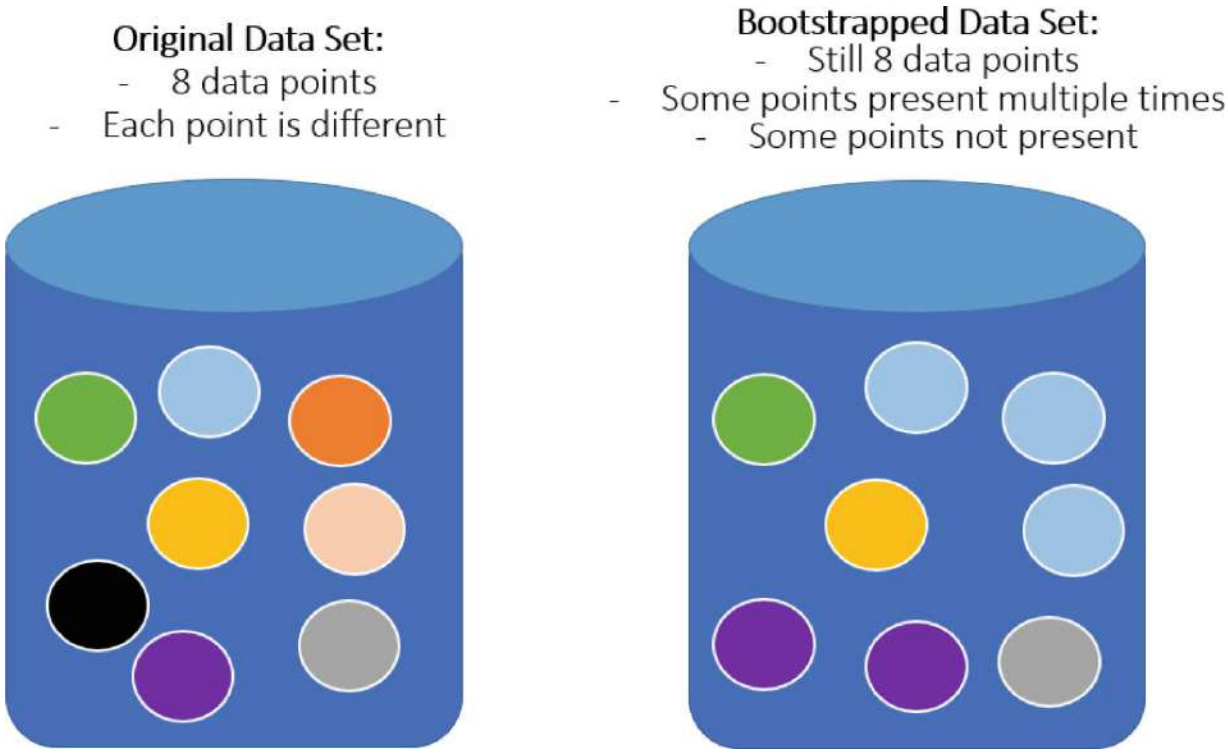


Figure 15-1 The bootstrap

Bagging Concept 2: Aggregation

The **aggregation** part describes the fact of using multiple learners. In the Random Forest, the bootstrap is executed many times. The exact number is defined by a hyperparameter called **n_estimators**.

For each n_estimators, a Decision Tree is built using a bootstrapped dataset. This will generate a large number of Decision Trees that are all slightly different due to the differences in the data sets. In the end, you can use each of the Decision Trees to make a prediction. However, you would end up with many slightly different predictions.

The solution to this problem is in the aggregation part. To combine all the individual Decision Tree predictions into one Random Forest prediction, you simply take the average of all the individual predictions. It is relatively straightforward, yet a crucial part of the algorithm.

The idea behind this is that the aggregation of many weak learners will result in errors that average each other out. This makes the Random Forest a very performant machine learning algorithm.

Random Forest Concept 2: Variable Subsets

Besides the bootstrapping approach, the Random Forest has a **second process** in place to make sure that the individual learners are not the same. This process does not apply to the data points that are or are not used, but rather **applies to the variables that are or are not used**.

As explained in the chapter on Decision Trees, the Decision Tree checks each of the variables to find the best split to add to the tree. Yet, when using a subset of each variable, you only let the tree check out the splits in a randomly selected number of variables. The exact quantity is defined by a hyperparameter called **mtry** in most mathematical descriptions and called **max_features** in scikit-learn. Although this can sometimes cause a selection of a suboptimal split, it does help to make the individual Decision Trees different from each other.

This value is generally around **80%**, but it can be higher or lower. Together, **n_estimators** and **max_features** are the main hyperparameters of the Random Forest model.

Predicting Births Using a Random Forest

Let's develop an example of the Random Forest model. For this example, you'll again take the Births data that you've used in Chapter 5. The ARMA model in Chapter 6 was able to obtain an R2 of **0.63**. This time, let's try to forecast the monthly number of Births rather than the yearly sum as was done in the previous example. You can import the data using Listing 15-1.

```
import pandas as pd

# Read the csv file
data = pd.read_csv('births_data.csv', sep=';')

# Keep useful columns
data = data[['Date', 'Births']]
```

Listing 15-1 Importing the data

Now, it is necessary to add some feature engineering. Let's add the variables Year and Month to account for seasonality and the lagged versions

of the target variable for the past 12 months. This task can be expected to be a bit harder, as there is a more detailed variation that the model needs to learn. The code in Listing 15-2 shows you how this can be done.

```
# Seasonality variables
data['Date'] = pd.to_datetime(data['Date'],
format='%d/%m/%Y')
data['Year'] = data['Date'].apply(lambda x:
x.year)
data['Month'] = data['Date'].apply(lambda x:
x.month)

# Adding a year of lagged data
data['L1'] = data['Births'].shift(1)
data['L2'] = data['Births'].shift(2)
data['L3'] = data['Births'].shift(3)
data['L4'] = data['Births'].shift(4)
data['L5'] = data['Births'].shift(5)
data['L6'] = data['Births'].shift(6)
data['L7'] = data['Births'].shift(7)
data['L8'] = data['Births'].shift(8)
data['L9'] = data['Births'].shift(9)
data['L10'] = data['Births'].shift(10)
data['L11'] = data['Births'].shift(11)
data['L12'] = data['Births'].shift(12)
```

Listing 15-2 Feature engineering

Now that you have got a data set, let's do a train-test split and fit the random forest with the default hyperparameters. Use the code in Listing 15-3 to fit this default model.

```
import mlflow
mlflow.autolog()

# Create X and y object
data = data.dropna()
y = data['Monthly Mean Total Births Number']
```

```
X = data[['Year', 'Month', 'L1', 'L2', 'L3', 'L4',
'L5', 'L6', 'L7', 'L8', 'L9', 'L10', 'L11',
'L12']]

# Create Train test split
from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.1,
random_state=12345, shuffle=False)

from sklearn.ensemble import RandomForestRegressor
my_rf = RandomForestRegressor()
my_rf.fit(X_train, y_train)
fcst = my_rf.predict(X_test)

from sklearn.metrics import r2_score
r2_score(list(y_test), list(fcst))
```

Listing 15-3 Fitting the default Random Forest Regressor

Using this model, you will obtain an R2 score for the prediction of **0.746**.

Grid Search on the Two Main Hyperparameters of the Random Forest

This is already a great result, but as always, let's see if that can be optimized using a grid search hyperparameter tuning. This will be shown in Listing 15-4.

```
from sklearn.model_selection import GridSearchCV

my_rf = GridSearchCV(RandomForestRegressor(),
                 {'max_features':[0.65, 0.7, 0.75,
0.8, 0.85, 0.9, 0.95],
                 'n_estimators': [10, 50, 100,
250, 500, 750, 1000]},
```

```
                    scoring = 'r2', n_jobs = -1)

my_rf.fit(X_train, y_train)
print(r2_score(list(y_test),
list(my_rf.predict(X_test))))
print(my_rf.best_params_)
```

Listing 15-4 Fitting the Random Forest Regressor with hyperparameter tuning

This will obtain an R2 score of **0.753**, slightly better than the default version. The optimal hyperparameters that have been identified are max_features (mtry) of **0.7** and n_estimators (ntrees) of **750**.

It would be useful now to have a look at the performance plot to see what it looks like on the test data. This can be done using Listing 15-5, and this will obtain Figure 15-2.

```
import matplotlib.pyplot as plt
plt.plot(list(fcst))
plt.plot(list(y_test))
plt.legend(['fcst', 'actu'])
plt.ylabel('Births')
plt.xlabel('Steps into the test data')
plt.show()
```

Listing 15-5 Obtaining the plot of the forecast on the test data

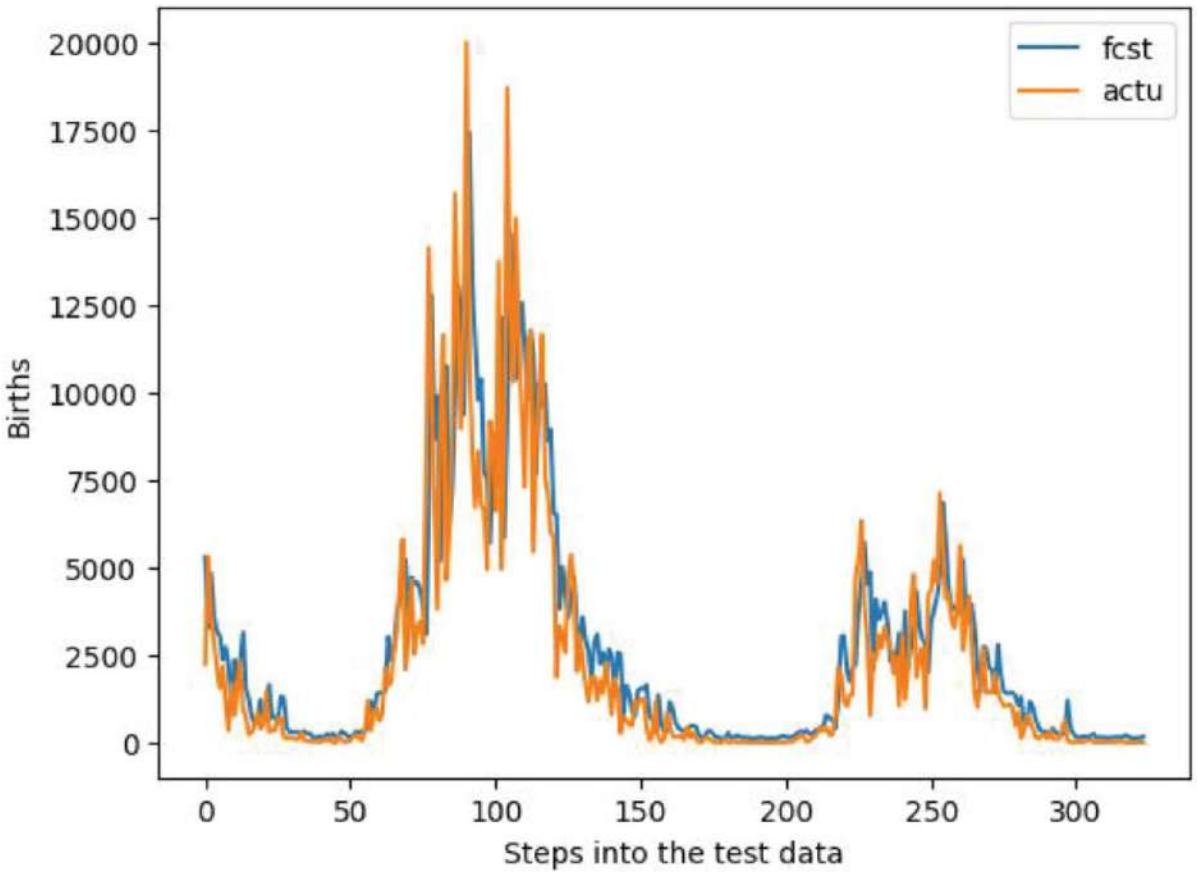


Figure 15-2 Predictive plot of Births per month

Random Search CV Using Distributions

As an additional step in this chapter, let's go a bit deeper into the **Random Search CV** that you discovered in the previous chapter. As you'll understand, the more complicated the models become, the heavier the grid search becomes. Alternatives become more and more interesting. You'll discover more methods for hyperparameter tuning in the next chapters.

In this application of Random Search, you'll use **distributions** rather than a list of possible values to specify which values you want to test for the different hyperparameters. This is a great functionality of RandomizedSearchCV, especially when you have an idea of what the most likely value for your hyperparameters is.

Let's start by identifying the distributions for max_features and n_estimators and then putting them into the RandomizedSearchCV.

Distribution for max_features

For the value of max_features, you previously tested values between 0.65 and 0.95 with steps of 0.05. The most used distribution is the **normal distribution**, so let's see how we could use the normal distribution for this.

You can use the code in Listing 15-6 to try out different normal distributions and check out whether they fit. The normal distribution is defined by a mean and a standard deviation, so those are the two values that you can play around with.

```
import numpy as np
import scipy.stats as stats
import math

mu = 0.8
variance = 0.005
sigma = math.sqrt(variance)
x = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)
plt.plot(x, stats.norm.pdf(x, mu, sigma))
plt.show()
```

Listing 15-6 Testing out a normal distribution for the max_features

As shown in Figure 15-3, the normal distribution with a mean of 0.8 and a standard deviation of 0.05 covers the range of 0.65–1.0 quite well. The x axis in this graph shows the value for max_features, and the y axis shows the corresponding probability for this value being sampled into the random search.

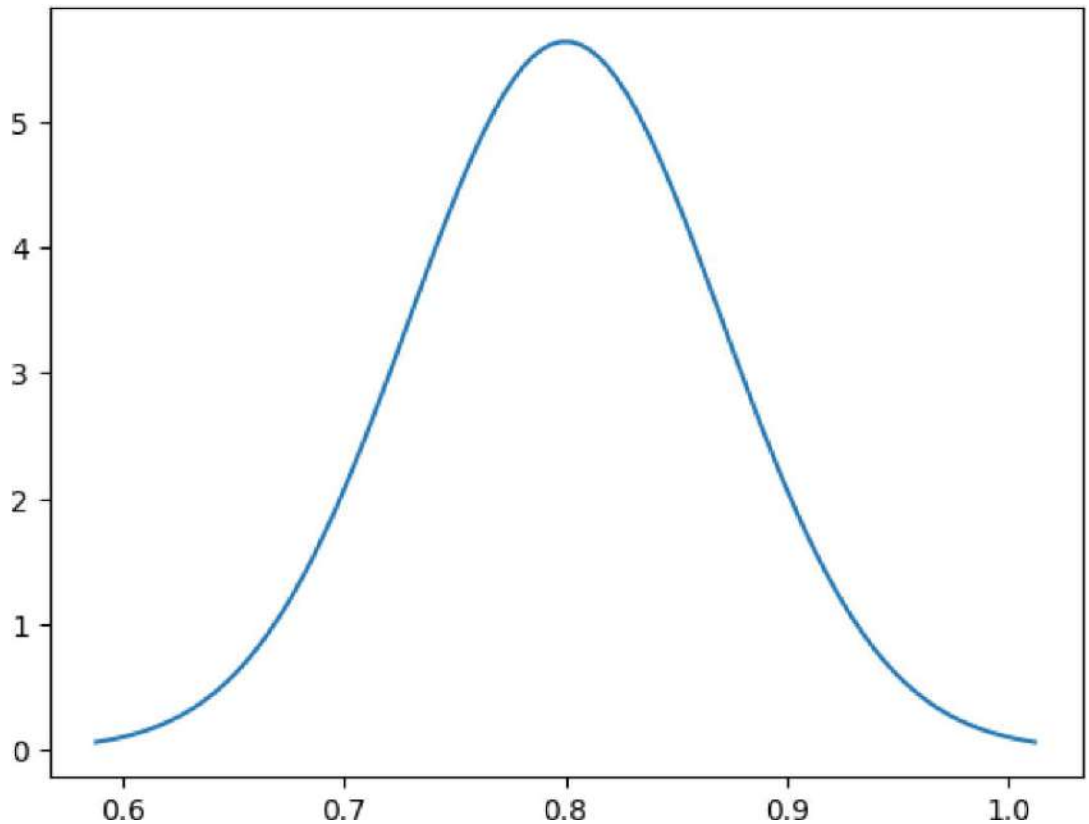


Figure 15-3 Normal distribution with mean 0.8 and st. dev. 0.05

As those values seem quite appropriate for max_features, we'll use this distribution in the RandomizedSearchCV.

Distribution for n_estimators

For the distribution of n_estimators, let's try something different. Let's say that we want to test values between 50 and 1000 but that we don't really have a good idea of what the actual value for n_estimators should be.

This means that a **uniform distribution** may be appropriate: in a uniform distribution, you specify a minimum and a maximum, and all values in between are **equally probable** to be selected. This is shown in Listing 15-7.

```
x = np.linspace(0, 2000, 100)
plt.plot(x, stats.uniform.pdf(x, 50, 950))
plt.show()
```

Listing 15-7 Testing out a uniform distribution for the n_estimators

The resulting plot is shown in Figure 15-4.

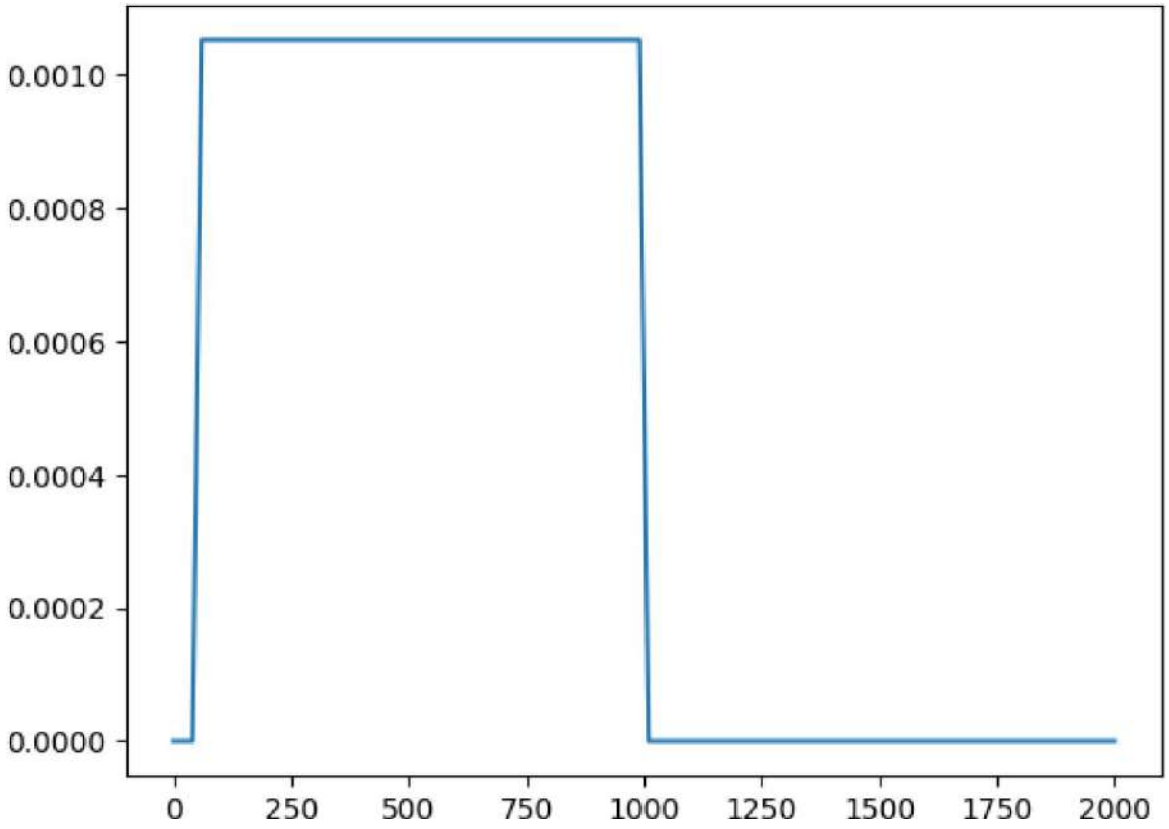


Figure 15-4 Uniform distribution with min 50 and max 1000

Yet, there is a small problem. The uniform distribution returns a floating number, while the n_estimators can only take an **integer**. The better alternative for a uniform integer is to use the function **scipy.stats.randint**, which returns a random integer between a minimum and a maximum.

Fitting the RandomizedSearchCV

Let's now fit the RandomizedSearchCV using the code in Listing 15-8.

```
from sklearn.model_selection import
RandomizedSearchCV

# Specifying the distributions to draw from
distributions = {
    'max_features': stats.norm(0.8,
math.sqrt(0.005)),
```

```
    'n_estimators': stats.randint(50, 1000)
}

# Creating the search
my_rf =
RandomizedSearchCV(RandomForestRegressor(),

                           distributions,
n_iter=10,
                           scoring = 'r2',
                           n_jobs = -1,
                           random_state = 12345)

# Fitting the search
my_rf.fit(X_train, y_train)

# Printing the results
print(r2_score(list(y_test),
list(my_rf.predict(X_test))))
print(my_rf.best_params_)
```

Listing 15-8 RandomizedSearchCv with two distributions

The results that you should obtain are an R2 of **0.7528**, and the selected hyperparameters are

- 'max_features': 0.70
- 'n_estimators': 819

The R2 is slightly lower than the one obtained by the Grid Search. Yet, you should observe a very strong increase in execution time. The RandomizedSearchCV is much faster while only losing a very slight amount of performance: a great argument for using RandomizedSearchCV rather than GridSearchCV.

Interpretation of Random Forests: Feature Importance

As you have understood by now, the Random Forest is a relatively complex model. When making a prediction, it combines the predictions of many Decision Trees. This gives the model great performance, but it also makes the model relatively difficult to interpret.

For the Decision Tree model, you have seen how you can plot the tree and follow its decisions based on a new observation. For the Random Forest, you would need to do this many times, making for a very difficult process.

Luckily, an alternative exists for interpreting the Random Forest. This method is called Feature Importance.

The Feature Importance of each variable is computed all the way throughout the fitting of the Random Forest. Every time a variable is used for a split, the error reduction that is brought about by this split is added to the variable's feature importance. At the end of the algorithm, those added-up errors are standardized so that the sum of the variable importance for all variables is 1.0.

The higher the future importance of a variable, the more important the role it has played in the model and therefore the more predictive value it has for your forecast.

You can obtain the feature importances from a Random Forest using Listing 15-9. Note that the Random Forest here is the one fit with RandomizedSearchCV, which is why you call best_estimator first. You also see how to combine the array of feature importances into a more accessible dataframe.

```
fi = pd.DataFrame({
        'feature': X_train.columns,
        'importance':
my_rf.best_estimator_.feature_importances_})

fi.sort_values('importance', ascending=False)
```

Listing 15-9 Feature Importances

The result looks as shown in Figure 15-5.

	feature	importance
2	L1	0.561466
3	L2	0.175941
4	L3	0.072618
5	L4	0.038316
6	L5	0.020826
7	L6	0.018465
12	L11	0.015895
10	L9	0.015860
11	L10	0.015599
13	L12	0.013989
9	L8	0.013970
0	Year	0.013061
8	L7	0.012917
1	Month	0.011078

Figure 15-5 Feature importances of the Random Forest

As you see, the most important feature here is the L1: the 1-lag autoregressive component. The following importances are the other lags. We can understand from this that the model is mainly learning an autoregressive effect with the more recent lags being more important. This is an interesting learning for understanding the model. In some cases, this information can also help you in improving the model even further by improving the selection of your variables, or it can give you hints on how to improve your feature engineering.

Key Takeaways

- The Random Forest adds bagging to the Decision Tree model.
- Bagging is an ensemble method: it fits the same model many times and takes their average as a prediction.
- The Random Forest has two main hyperparameters;
  - The number of trees to use in the forest
  - The number of randomly selected variables to use in each Decision Tree split
- Random Search is a faster way to tune hyperparameters and often gives results that are not much worse than Grid Search. You can give probability distributions for each hyperparameter to tune, and the Random Search will do a certain number of draws in those distributions.
- Variable importance is a way to interpret the results of the Random Forest model. It can also hint at how to improve the model further.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_16

16. Gradient Boosting with XGBoost, LightGBM, and CatBoost

Joos Korstanje[1] ✉
(1) Maisons Alfort, France

In this chapter, you will discover the **Gradient Boosting model**. In the previous chapter, you discovered the idea behind ensemble methods. As a recap, ensemble methods make powerful predictions by combining predictions of numerous small, less performant models.

Boosting: A Different Way of Ensemble Learning

Gradient Boosting combines numerous small decision tree models to make predictions. Of course, those small decision trees are different from each other: otherwise, there wouldn't be any advantage to using a larger number of them.

The important concept to understand here is how those decision trees come to be different from each other. This is achieved by a process called **boosting**. Boosting and bagging, which you've seen in the previous chapters, are the two principal methods of ensemble learning.

Boosting is an **iterative process**. It adds more and more weak learners to the ensemble model in an intelligent way. In each step, the individual data points are weighted. Data points that are already predicted well will not be important for the learner to be added. The new weak learners will therefore *focus on learning the things that are not yet understood and therefore improve the ensemble*.

You can see a schematic overview of the boosting process in Figure 16-1. With this approach, you iteratively fit weak models that focus on the

parts of the data that are not yet understood. While doing this, you keep all the intermediate weak models. The ensemble model is the combination of all those weak learners.

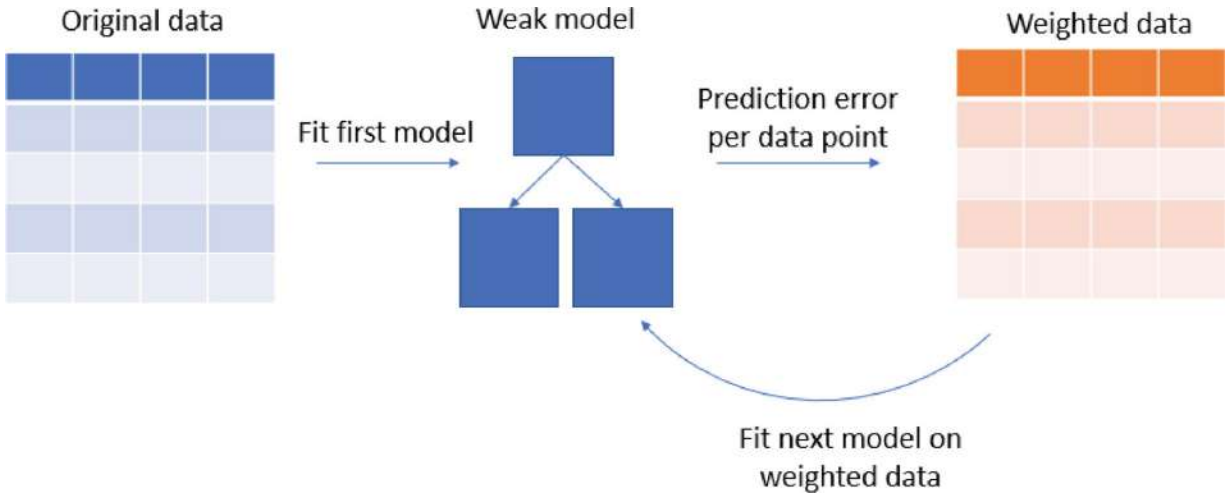


Figure 16-1 The boosting process

Gradient Boosting

There are many algorithms that each perform slightly different versions of gradient boosting. When the gradient boosting approach was first invented, the algorithms were not very performant, but that changed with the AdaBoost algorithm: the first algorithm that could adapt to weak learners.

Gradient Boosting algorithms are among the most performant machine learning tools on the market. After AdaBoost, a long list of slightly different boosting algorithms has been added to the literature, including XGBoost, LightGBM, LPBoost, BrownBoost, MadaBoost, LogitBoost, and TotalBoost. There are still many contributions being made to improve on gradient boosting theory. In the current chapter, three algorithms will be covered: XGBoost, LightGBM, and CatBoost.

XGBoost is one of the most used machine learning algorithms. XGBoost is a quick way to get good performances. As it is easy to use and very performant, it is the first go-to algorithm for many ML practitioners.

LightGBM is another Gradient Boosting algorithm that is important to know. It used to be a bit less widespread than XGBoost, but it has seriously gained in popularity. The expected advantage of LightGBM over XGBoost is a gain in speed and memory use.

CatBoost is another Gradient Boosting algorithm. It has been developed as an alternative to the existing boosting algorithms in order to improve the handling of categorical data. It has become a very popular algorithm over the past few years.

In this chapter, you will discover the implementations of those three boosting algorithms.

The Difference Between XGBoost and LightGBM

If you're going to use gradient boosting algorithms, it is important to understand in what way they differ. This can also give you an insight into the types of differences that make for such a large number of models on the market.

The difference here is in the way we identify the best splits inside the weak learners (the individual decision trees). Remember that the splitting in a decision tree is the moment where your tree needs to find the split that most improves the model.

The intuitively most simple idea for finding the best split is to loop through all the potential fits and find the best one. Yet, this takes a lot of time, and better alternatives have been proposed by recent algorithms.

The alternative proposed by XGBoost is to use a **histogram-based splitting**. In this case, rather than looping through all possible splits, the model builds histograms of each of the variables and uses those to find the best split per variable. The best overall split is then retained.

LightGBM was invented by Microsoft, and it has an even more efficient method to define the splits. This method is called **Gradient-based One-Side Sample (GOSS).** Goss computes gradients for each of the data points and uses this to filter out data points with a low gradient. After all, data points with a low gradient are already understood well, whereas individuals with a high gradient need to be learned better. LightGBM also uses an approach called **Exclusive Feature Bundling (EFB)**, which is a feature that allows for a speedup when having many correlated variables to choose from.

Another difference is that the LightGBM model fits leaf-wise (best-first) tree growth, whereas XGBoost grows the trees level-wise. You can see the difference in Figure 16-2. This difference is a feature that would

theoretically favor LightGBM in terms of accuracy, yet it comes at a higher risk of overfitting in the case of little data available.

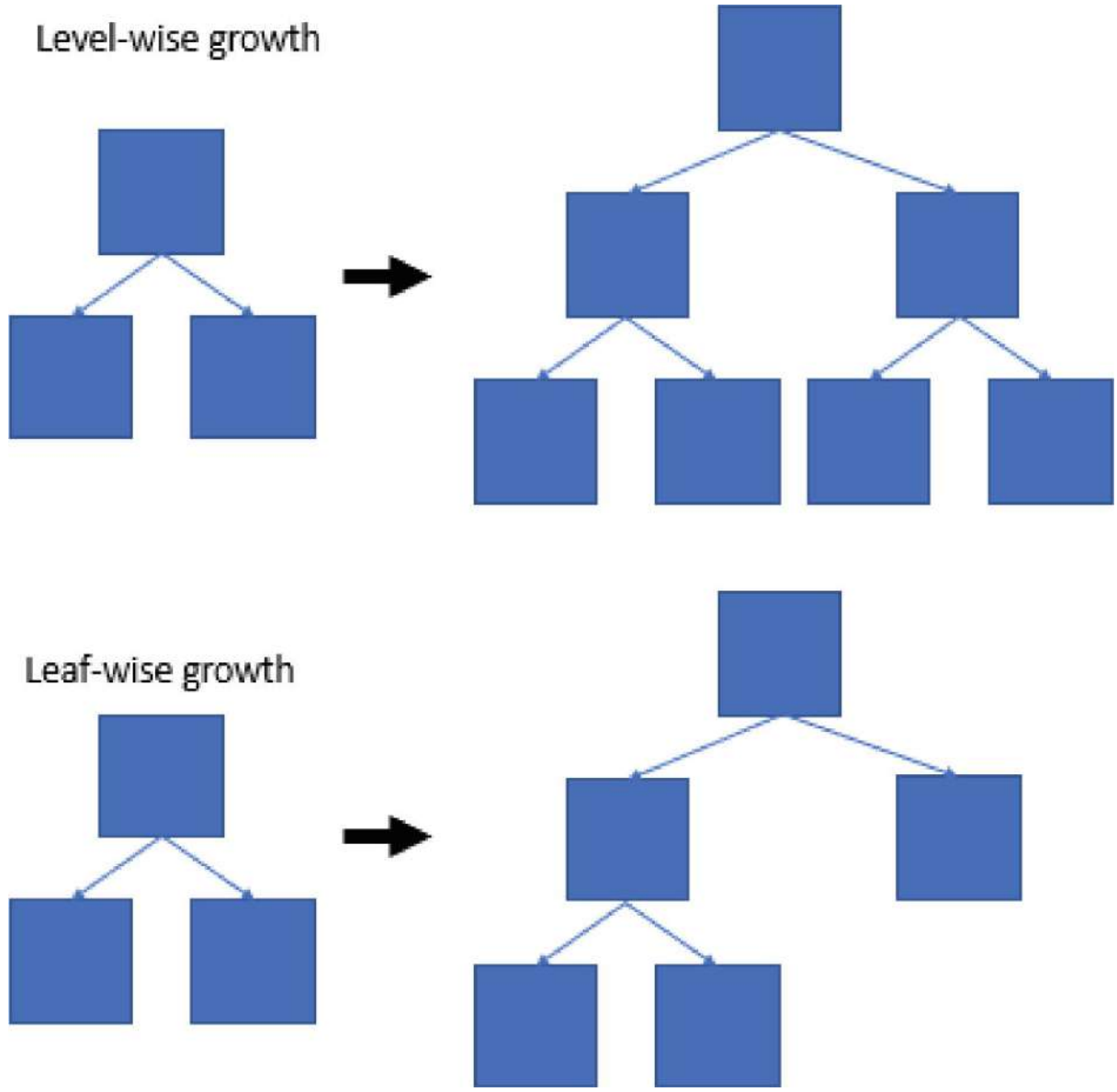


Figure 16-2 Leaf-wise growth vs. level-wise growth

For more details on the differences between the gradient boosting algorithms, you can check out the paper by Microsoft: (https://www.microsoft.com/en-us/research/publication/lightgbm-a-highly-efficient-gradient-boosting-decision-tree/).

Adding CatBoost to the Mix

CatBoost, the third boosting algorithm covered in this chapter, presents a number of essential differences with XGBoost and LightGBM. CatBoost was created by Russian technology company Yandex and has, as its main differencing point, the capability of handling categorical variables.

Categorical variables are extremely common in real-world datasets, yet they are difficult to handle in numerous machine learning algorithms, including XGBoost and LightGBM. If you find yourself working on a dataset with categorical variables, CatBoost would be a good algorithm to try.

Forecasting Traffic Volume with XGBoost

For this chapter, we'll be using the same dataset as the one used for the kNN model: interstate traffic. In the previous example, we were able to obtain an R2 of **0.9724**. Gradient Boosting is a more performant algorithm, so let's see whether we can use it to improve on this already great score.

You can find this dataset on the UCI machine learning repository: https://archive.ics.uci.edu/ml/datasets/Metro+Interstate+Traffic+Volume. You can import the data using listing 16-1.

```
import pandas as pd
data =
pd.read_csv('Metro_Interstate_Traffic_Volume.csv')
```

Listing 16-1 Importing the data

To make a fair benchmark between the Gradient Boosting models and the kNN model, let's do the exact same feature engineering as the one applied in the chapter on kNN. This feature engineering is shown in Listing 16-2.

```
data['year'] = data['date_time'].apply(lambda x:
int(x[:4]))
data['month'] = data['date_time'].apply(lambda x:
int(x[5:7]))
```

```
data['weekday'] =
pd.to_datetime(data['date_time']).apply(lambda x:
x.weekday())
data['hour'] =
pd.to_datetime(data['date_time']).apply(lambda x:
x.hour)
data['isholiday'] = (data['holiday'] ==
'None').apply(float)
```

Listing 16-2 Applying the same feature engineering as done previously for the kNN model

Let's now do a first test with XGBoost, using Listing 16-3. You use the XGBoost package for this.

> **Note** Installing libraries using Python can sometimes be a pain due to the need for different dependencies. If you have trouble installing those boosting libraries, you could check out Google Collaboratory Notebooks (https://colab.research.google.com) or Kaggle notebooks (kaggle.com/notebooks) which are free notebook environments that have all the modern libraries for machine learning at the ready for you.

```
import mlflow
mlflow.autolog()

# Create objects X and y
X = data[['year', 'month', 'weekday', 'hour',
'isholiday']]
y = data['traffic_volume']

# Create Train test split
from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=100,
random_state=12345, shuffle=False)

from xgboost import XGBRegressor
my_xgb = XGBRegressor()
```

```
my_xgb.fit(X_train, y_train)

fcst = my_xgb.predict(X_test)

from sklearn.metrics import r2_score
r2_score(list(y_test), list(fcst))
```

Listing 16-3 Applying the default XGBoost model

This results in an R2 score of **0.9726.** This is more or less the same as the **0.9724** that was obtained using kNN.

Forecasting Traffic Volume with LightGBM

Now let's do the same thing with LightGBM and see how that compares using Listing 16-4.

```
from lightgbm import LGBMRegressor
my_lgbm = LGBMRegressor()
my_lgbm.fit(X_train, y_train)

lgbm_fcst = my_lgbm.predict(X_test)

r2_score(list(y_test), list(lgbm_fcst))
```

Listing 16-4 Applying the default LightGBM model

The R2 score of the LightGBM is **0.9728.** This is very slightly better than the default XGBoost model, and it is also very slightly better than the tuned kNN model.

Forecasting Traffic Volume with CatBoost

As a third and final example, let's apply CatBoost to the benchmark. As described earlier, CatBoost can be expected to have an advantage when there is categorical data. It happens to be the case that there are multiple categorical variables in the dataset, for example: 'month', 'weekday', and 'isholiday'. Let's apply CatBoost using the code in Listing 16-5.

```
from catboost import CatBoostRegressor
```

```
my_catboost = CatBoostRegressor()
my_catboost.fit(X_train, y_train)

catboost_fcst = my_catboost.predict(X_test)
print(r2_score(list(y_test), list(catboost_fcst)))
```

Listing 16-5 Applying the default CatBoost model

The obtained R2 score is **0.9731**, which beats the other forecasts by a very small amount.

Inspecting the Differences in Predictions

Let's create a graph to see where the three models predicted something different. You can use Listing 16-6 to create the graph. The graph should be the one in Figure 16-3.

```
import matplotlib.pyplot as plt
plt.figure(figsize=(20,10))
plt.plot(list(y_test))
plt.plot(list(xgb_fcst))
plt.plot(list(lgbm_fcst))
plt.plot(list(catboost_fcst))
plt.legend(['actual', 'xgb forecast', 'lgbm
forecast', 'catboost forecast'])
plt.ylabel('Traffic Volume')
plt.xlabel('Steps in test data')
plt.show()
```

Listing 16-6 Create a graph to compare the XGBoost and LightGBM forecast to the actuals

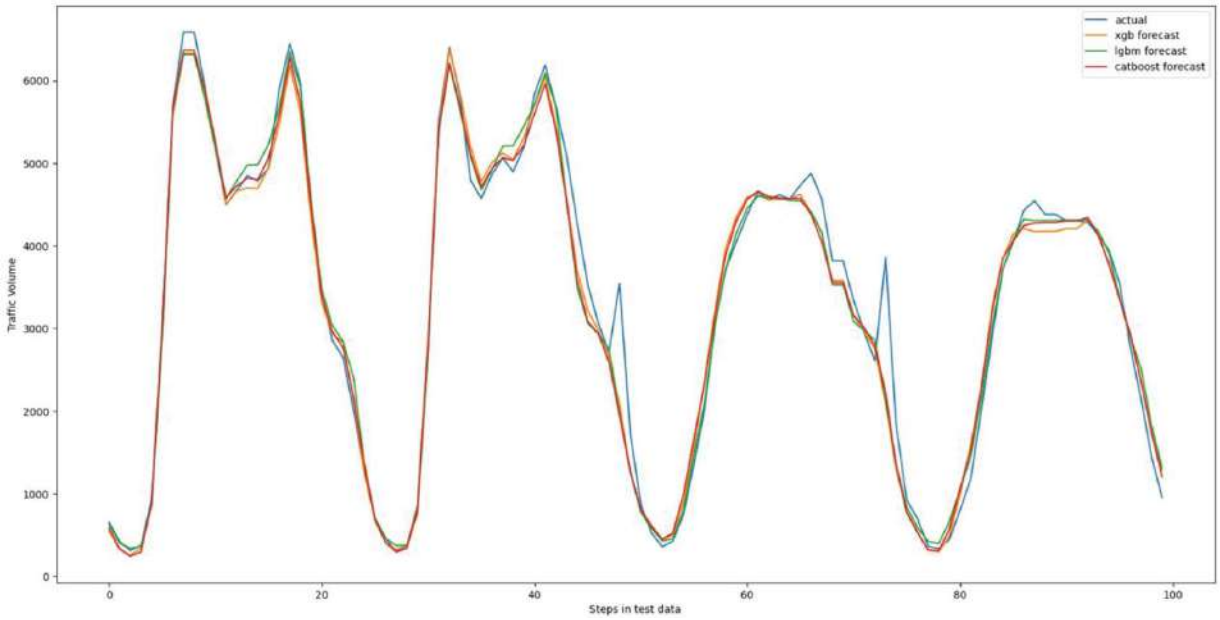


Figure 16-3 Comparing the two gradient boosting models

As you can see in this graph, the difference in the three model performances is not very visible. The three models are mainly off in the same places. This could indicate that there is some variation that should be explained by other external variables than those presented to the model. Overall, the predictions fit quite closely to the actual data.

Hyperparameter Tuning Using Bayesian Optimization

As a next step, we now want to tune the three models to be able to see how the tuned versions perform. This could make a big difference.

Rather than using GridSearchCV, we'll be using **Bayesian Optimization** here. At this stage, you should be comfortable with the two approaches for hyperparameter tuning that were presented earlier: GridSearchCV and RandomizedSearchCV.

The Theory of Bayesian Optimization

In Bayesian Optimization, rather than trying out every combination of hyperparameters, or just trying out random guesses, Bayesian Optimization allows you to make intelligent guesses by modeling the likelihood of a

certain value for the hyperparameters to be the optimum. This is done by modeling a probability distribution around the hyperparameters.

The concept is shown in Figure 16-4. The picture shows only one hyperparameter, but it can be projected onto a situation with multiple hyperparameters. The Bayesian Optimization will start sampling a number of values of the hyperparameter and observe the model performance at those points.

The goal is not to randomly fall on one value that performs well, but rather to estimate the *gray zone around the curve*. Some values of the hyperparameter the values will give better performance than others. While testing values, the Bayesian Optimization makes a model of the probability distribution of the optimum being at this location.

In this way, Bayesian Optimization is much more **intelligent** than the two tuning methods that you have seen until here. It can make intelligent, probability-based guesses about the points that should be tested next. This makes it much more powerful for optimizing hyperparameters.

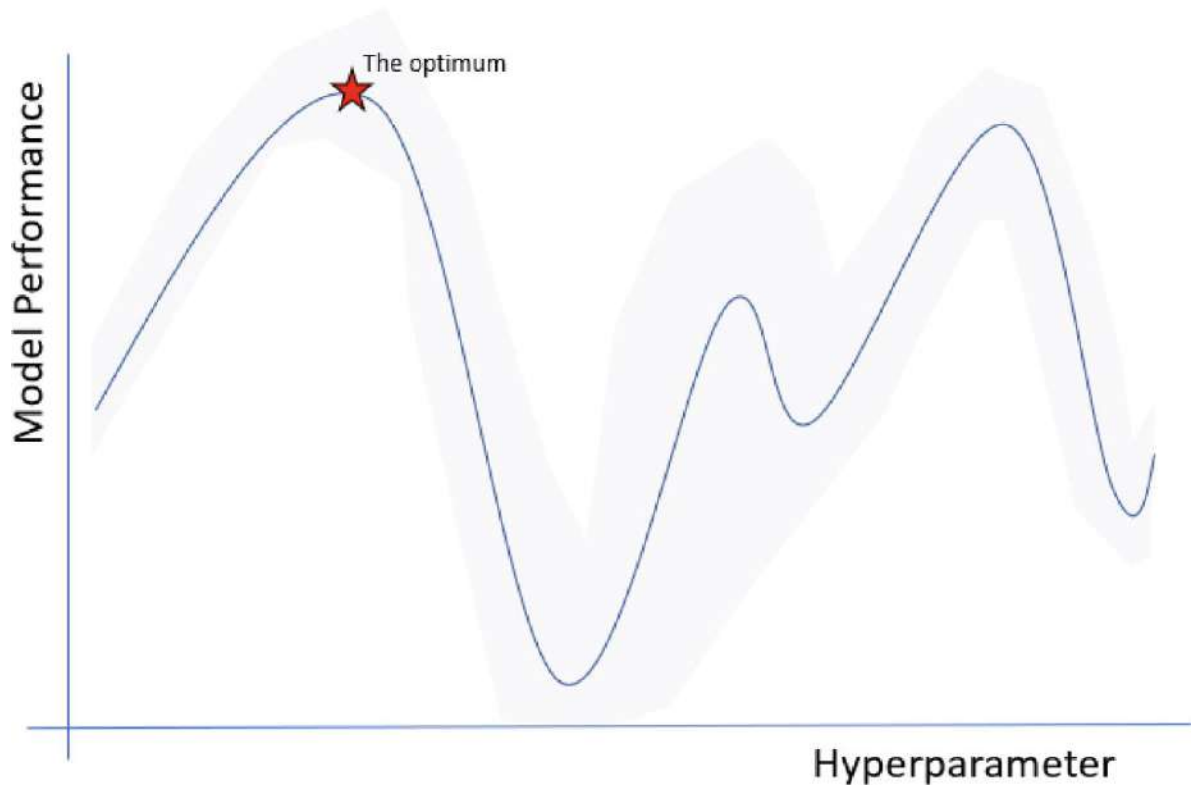


Figure 16-4 Bayesian Optimization

Bayesian Optimization Using scikit-optimize

In this part, you will see how to use the scikit-optimize package to apply a Bayesian hyperparameter tuning to the XGBoost, LightGBM, and CatBoost. After that, you'll do a comparison of the three models. In Listing 16-7, you can see this applied to XGBoost.

```
from bayes_opt import BayesianOptimization

# Create cost function to optimize
def xgb_opt_func(learning_rate, max_depth,
n_estimators):

    my_xgb = XGBRegressor(
        learning_rate=learning_rate,
        max_depth=int(max_depth),
        n_estimators=int(n_estimators)
    )

    my_xgb.fit(X_train, y_train)

    xgb_fcst = my_xgb.predict(X_test)

    return r2_score(list(y_test), list(xgb_fcst))

# Set up the Bayesian Optimization
xgb_opt = BayesianOptimization(
    xgb_opt_func,
    {
        'learning_rate': (10e-6, 1.0),
        'max_depth': (0, 50),
        'n_estimators' : (10, 1000),
    },
    random_state=123
)

# Run the optimization
xgb_opt.maximize(init_points=20, n_iter=100)
```

```
# Create the model with best params from the
optimization
tuned_xgb = XGBRegressor(
    learning_rate=xgb_opt.max['params']
['learning_rate'],
    max_depth=int(xgb_opt.max['params']
['max_depth']),
    n_estimators=int(xgb_opt.max['params']
['n_estimators'])
)

# Fit the optimal model and obtain R2 score
tuned_xgb.fit(X_train, y_train)
xgb_tuned_fcst = tuned_xgb.predict(X_test)

r2_score(list(y_test), list(xgb_tuned_fcst))
```

Listing 16-7 Applying a Bayesian Optimization to the XGBoost

The resulting R2 score from the tuned model is **0.9699**. This is worse than the default settings, which shows two things. First of all, it shows that more iterations may be needed on the optimizer to allow it to find the maximum. Second, it shows that the default settings are actually already good enough, and tuning has little added value in this case. Let's continue directly with the LightGBM model, as shown in Listing 16-8.

```
from bayes_opt import BayesianOptimization

# Create cost function to optimize
def lgbm_opt_func(learning_rate, max_depth,
n_estimators):

    my_lgbm = LGBMRegressor(
        learning_rate=learning_rate,
        max_depth=int(max_depth),
        n_estimators=int(n_estimators)
    )

    my_lgbm.fit(X_train, y_train)
```

```
    lgbm_fcst = my_lgbm.predict(X_test)

    return r2_score(list(y_test), list(lgbm_fcst))

# Set up the Bayesian Optimization
lgbm_opt = BayesianOptimization(
    lgbm_opt_func,
    {
        'learning_rate': (10e-6, 1.0),
        'max_depth': (0, 50),
        'n_estimators' : (10, 1000),
    },
    random_state=123
)

# Run the optimization
lgbm_opt.maximize(init_points=20, n_iter=100)

# Create the model with best params from the
optimization
tuned_lgbm = LGBMRegressor(
    learning_rate=lgbm_opt.max['params']
['learning_rate'],
    max_depth=int(lgbm_opt.max['params']
['max_depth']),
    n_estimators=int(lgbm_opt.max['params']
['n_estimators'])
)

# Fit the optimal model and obtain R2 score
tuned_lgbm.fit(X_train, y_train)
lgbm_tuned_fcst = tuned_lgbm.predict(X_test)

r2_score(list(y_test), list(lgbm_tuned_fcst))
```

Listing 16-8 Applying a Bayesian Optimization to the LightGBM

The R2 score from the LightGBM is **0.9744**. So, after using Bayesian Optimization for the tuning of the two models, the best current score is the

tuned LightGBM. Although not by a lot, you can see the score slightly increasing throughout the different exercises. Let's see if a tuned version of CatBoost can deliver any performance increases using Listing 16-9.

```
# Create cost function to optimize
def catboost_opt_func(learning_rate, max_depth,
n_estimators):

    my_catboost = CatBoostRegressor(
        learning_rate=learning_rate,
        max_depth=int(max_depth),
        n_estimators=int(n_estimators)
    )

    my_catboost.fit(X_train, y_train)

    catboost_fcst = my_catboost.predict(X_test)

    return r2_score(list(y_test),
list(catboost_fcst))

# Set up the Bayesian Optimization
catboost_opt = BayesianOptimization(
    catboost_opt_func,
    {
        'learning_rate': (10e-6, 1.0),
        'max_depth': (0, 16),
        'n_estimators' : (10, 1000),
    },
    random_state=123
)

# Run the optimization
catboost_opt.maximize(init_points=20, n_iter=100)

# Create the model with best params from the
optimization
tuned_catboost = CatBoostRegressor(
```

```
    learning_rate=catboost_opt.max['params']
['learning_rate'],
    max_depth=int(catboost_opt.max['params']
['max_depth']),
    n_estimators=int(catboost_opt.max['params']
['n_estimators'])
)

# Fit the optimal model and obtain R2 score
tuned_catboost.fit(X_train, y_train)
catboost_tuned_fcst =
tuned_catboost.predict(X_test)

r2_score(list(y_test), list(catboost_tuned_fcst))
```

Listing 16-9 Applying a Bayesian Optimization to the CatBoost

This code obtains an R2 score of **0.9739**. Despite the long run time, the tuned CatBoost does not beat the tuned LightGBM's **0.9744**. This means that LightGBM is the winner of the benchmark.

As a final confirmation, let's plot the predictions that have been made by the two models, using the code in Listing 16-10, and you'll obtain the plot in Figure 16-5.

```
import matplotlib.pyplot as plt
plt.figure(figsize=(20,10))
plt.plot(list(y_test))
plt.plot(list(xgb_tuned_fcst))
plt.plot(list(lgbm_tuned_fcst))
plt.plot(list(catboost_tuned_fcst))
plt.legend(['actual', 'xgb forecast', 'lgbm
forecast', 'catboost forecast'])
plt.ylabel('Traffic Volume')
plt.xlabel('Steps in test data')
plt.show()
```

Listing 16-10 Plotting the tuned models

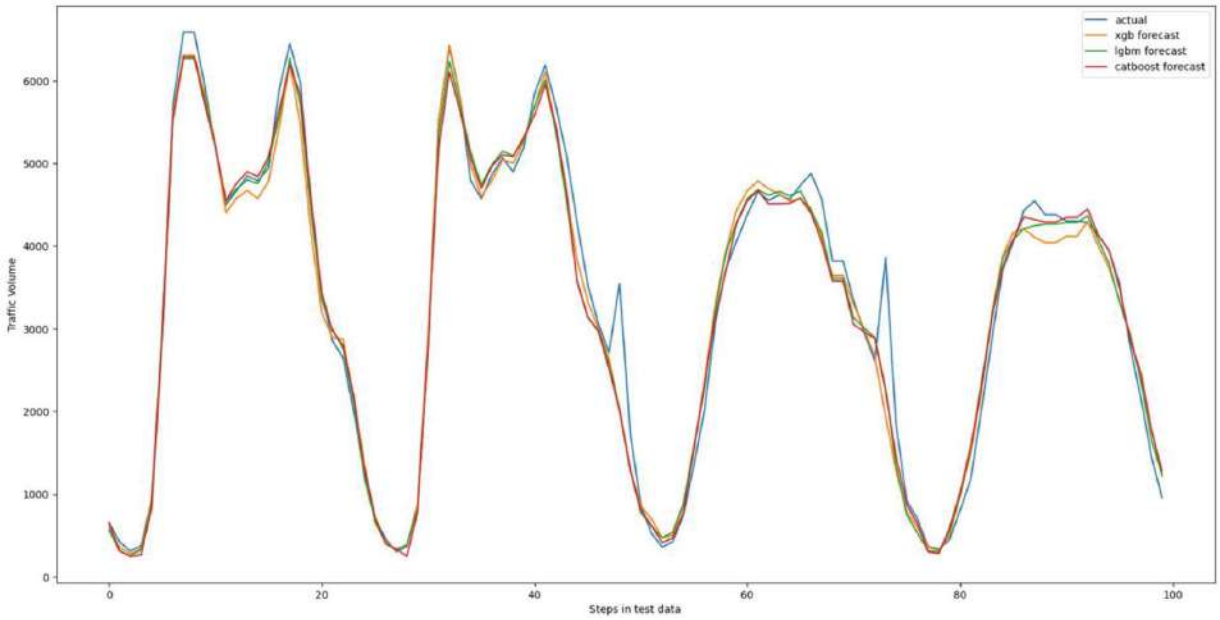


Figure 16-5 Compare the performances of the tuned models

In the current execution, you have seen how to tune a few of the hyperparameters of the boosting models. The execution time with this number of iterations is already quite long. If you want to play around with it, you could test if other hyperparameters and other value ranges would allow the Bayesian Optimization to find even higher values.

Conclusion

In conclusion, you have discovered three very performant machine learning algorithms in this chapter. You have also seen throughout this chapter how you could do a benchmark between models. In practice, you'll often do benchmarks between even more models. You'll see in a later chapter of this book that benchmarking and hyperparameter tuning are strongly focused on performance improvement only. Yet, other parameters like dataset selection and feature engineering have a large impact on model performance as well. Furthermore, your business case may require you to build models with a fast run time or a model that is easily explicable to management, in which case running complex optimization tools may not be the right choice.

Key Takeaways

- Gradient Boosting is an ensemble learning technique that lets subsequent individual models from the ensemble learn on the parts that are not yet understood well by the model.
- You have seen three boosting models:
  - XGBoost: A more traditional method for gradient boosting
  - LightGBM: A newer but very performant competitor
  - CatBoost: An alternative that is great with categorical data
- Bayesian Optimization is a more intelligent method for tuning hyperparameters. It estimates the probability of the optimum being on a certain location and therefore makes intelligent guesses for the optimum.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_17

17. Bayesian Models with pyBATS

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, you will discover Bayesian models and the pyBATS package. We will first dive into the idea behind Bayesian statistics in general and then see how Bayesian modeling can be used in machine learning.

There is not just one model called the Bayesian model. Rather, Bayesian modeling is a whole family of techniques based on Bayesian statistics. We will use the pyBATS package and use its core functionalities as an entry point into the world of Bayesian forecasting.

Intuitive Idea Behind Bayesian Models

In the previous chapter, you have already been in touch with Bayesian statistics. If you remember well, we used Bayesian optimization for hyperparameter tuning. As explained, in GridSearch or RandomSearch hyperparameter tuning, we would either try out all combinations of hyperparameters (GridSearch) or a randomly chosen set of combinations of parameters.

The whole idea of Bayesian optimization is to go beyond trying out different values for hyperparameters randomly. Rather, the Bayesian approach is to make an educated guess of the values of hyperparameters that are most likely to be higher. The Bayesian tuning does start with a number of random initializations (comparable to RandomSearch), but then it quickly starts identifying probability distributions. It will then do more trials in the ranges that are more probable to contain the maximum.

Imagine a situation where we are searching for a maximum value of y within a range of x between 0 and 10. You do four evaluations at values 2, 4, 6, 8, with the results as shown in Table 17-1.

Table 17-1 The building blocks of univariate time

series models

x Value	y Value
2	0
4	0
6	10
8	10

Given those first results, it is not possible to conclude with absolute certainty that the maximum in y is somewhere around the x values 6 and 8. However, if we accept that we are never going to be able to have absolute certainty, we may want to focus on deciding between more and less likely x values for the y maximum to be found.

Given that nobody has unlimited time and computing resources, we might as well focus on the most probable solutions. The Bayesian approach would be to do the next few guesses mostly in x values around 6 and 8.

Bayesian vs. Frequentist Statistics

This approach sounds very intuitive, and it would be hard to question it from a common-sense point of view. Yet, the Bayesian approach is not the standard in practicing statistics.

The frequentist approach to statistics is the paradigm opposing Bayesian statistics. Frequentists do not incorporate prior beliefs or information. When doing inference, a frequentist will follow a methodical approach in which they first define a hypothesis, then decide on an experiment, and collect exactly the data that was decided in their experiment plan. Once the experiment is finished, they will use p-values to decide whether the data allows them to reject their hypothesis.

This means that **a frequentist approach requires you to finish an experiment before making a conclusion**. Just as the Bayesian idea is to include all possible information as soon as possible, you can now see that there is also a good argument for the opposite idea: being more patient and waiting for an experiment to finish before drawing conclusions.

You are likely wondering whether one of those options is the better decision. Actually, the Bayesian and frequentist approaches are two paradigms that co-exist in the world of statistics. Some professionals are strongly in favor of one or the other, but neither has been able to dominate the field: we must learn to work with both.

In the current book, we are mainly looking into statistical models and machine learning techniques to obtain the most accurate forecasts possible. As we have done throughout this book, benchmarking models is the best way to evaluate the out-of-sample performance of your forecasting model.

Bayesian models can be very performant on forecasting tasks, so it is valuable for machine learning practitioners to master the required tools to add them to their benchmarks. The remainder of this chapter will focus on Python implementations to implement Bayesian forecasting models.

---

pyBATS As an Implementation of Bayesian Forecasting

Now that you have seen the idea behind Bayesian modeling, let's move on to an implementation of it. As you have seen throughout this book, when practicing applied data science, theoretical algorithms also need an easy-to-use implementation. Otherwise, it is difficult to implement those algorithms in our applied data science benchmarks.

The pyBATS package, although not extremely extensive, is a well-maintained package that implements a set of Bayesian models applied specifically to forecasting time series data. The foundation of the pyBATS package is the DGLM: the Dynamic Generalized Linear Model (DGLM). The LM part of this title is similar to the linear model that you have seen in an earlier chapter. Generalized Linear Models are variations of this linear model.

The linear model has multiple assumptions, which are a sort of statistical requirements that have to be met. For example, the dependent variable (y variable) needs to be a continuous numerical variable. A normal distribution is also assumed.

The G in the DGLM stands for Generalized. GLMs are a family of models of which the linear model is one member. GLMs adapt the linear model to target variables that have other distributions. An example is the Poisson distribution, in which the y variable is only positive and only integers, think of counting data. Another example is the Bernoulli distribution, which models binary events like coin flips.

The last element of the DGLM is the Dynamic element. The Dynamic element allows the linear relationship to evolve over time. This makes the DGLM ideal for time series modeling.

Forecasting Sales Using pyBATS

Let's now move on to an example implementation of Bayesian forecasting using the pyBATS package. We'll be using a dataset (1) that collects sales of an Italian grocery store. The data contains daily sales amounts per brand per product and also contains information on promotions. After all, promotions may strongly impact sales.

(1) Source: A machine learning approach for forecasting hierarchical time series (Paolo Mancuso, Veronica Piccialli, and Antonio M. Sudoso)

Sales Forecast: Exploratory Data Analysis

You can use the code in Listing 17-1 to import the data. The data is a bit too detailed. Therefore, this code also regroups the data. We will be building a monthly forecast of total sales, so the code sums the data for all products per month.

```
import  pandas as pd
import matplotlib.pyplot as plt

# Read the data
df = pd.read_csv('hierarchical_sales_data.csv')

df_sum = df[['DATE']]

# Sum the sales and promos of the different products
df_sum['SALES'] = df[[x for x in df.columns if
x.startswith('QTY')]].sum(axis=1)
df_sum['PROMO'] = df[[x for x in df.columns if
x.startswith('PROMO')]].sum(axis=1)

# Create sum of sales and promos per month
df_sum['YEAR_MONTH'] = df_sum['DATE'].apply(lambda x:
x[:7])
df_sum = df_sum[['SALES', 'PROMO',
'YEAR_MONTH']].groupby('YEAR_MONTH').sum()
df_sum.head()

# Preview the data
df_sum.head()
```

Listing 17-1 Importing the data

You will obtain the following view of the dataset (Figure 17-1).

YEAR_MONTH	SALES	PROMO
2014-01	12819	990
2014-02	17906	1329
2014-03	12047	896
2014-04	15998	1235
2014-05	17453	1354

Figure 17-1 The grouped data

To get a feeling of the patterns in the data, you can use Listing 17-2 to create a plot that shows both variables (SALES and PROMO) over time. You can see that there is an overall downtrend (sales are decreasing over time), as well as a strong fluctuation (low points are around 10k, whereas high sales are around 17.5k).

```
df_sum.plot()
plt.xticks(rotation=45)
```

Listing 17-2 Plot the data

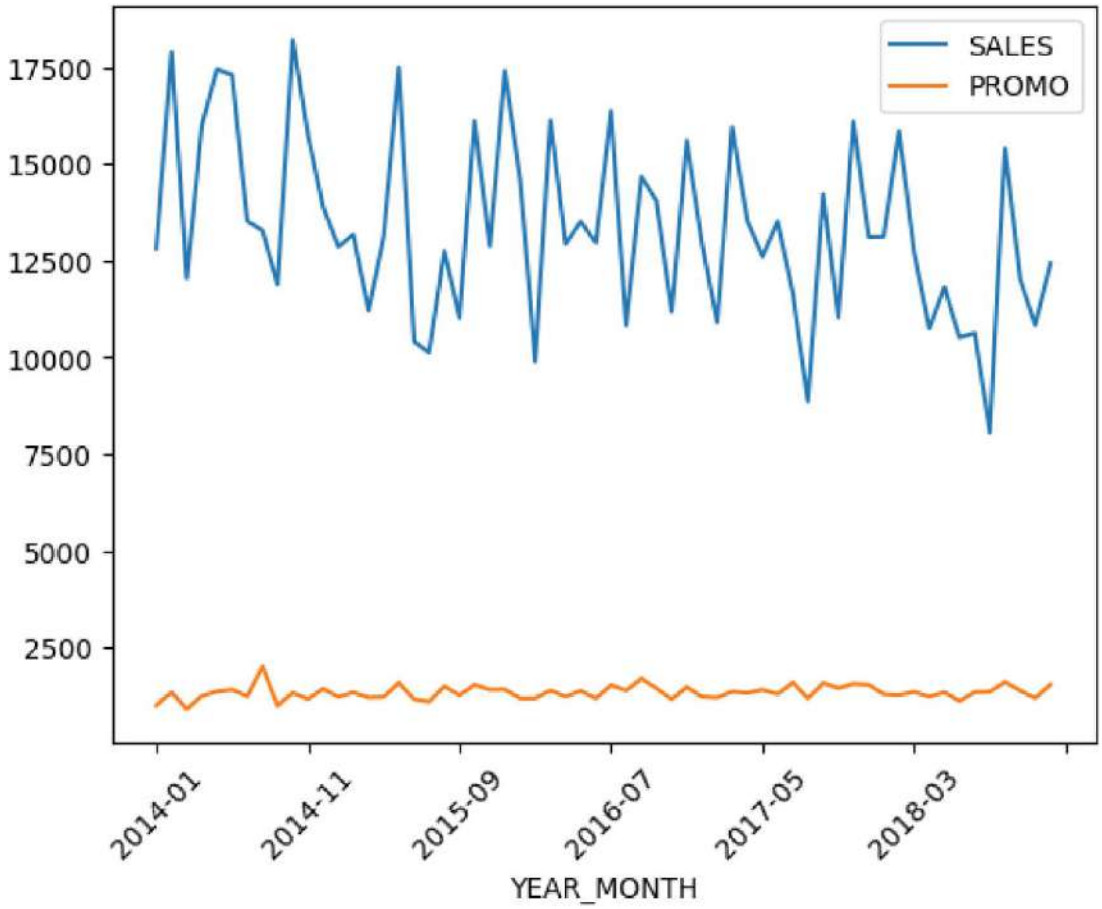


Figure 17-2 The plot

In this plot, it may seem that the promotion variable is almost constant over time. However, we must not forget that the percentual fluctuation of promotions is very hard to observe, as the graph forces the two variables on the same y axis. An easy way to scale the PROMO data is shown in Listing 17-3. Scaling the data for the plot will show us whether there is a potential correlation between promotions and sales.

```
df_scaled = df_sum[['SALES']]
df_scaled['PROMO'] = df_sum['PROMO'] * 8

df_scaled.plot()
plt.xticks(rotation=45)
```

Listing 17-3 Investigate if there is a correlation

The code in Listing 17-3 will obtain the graph in Figure 17-3.

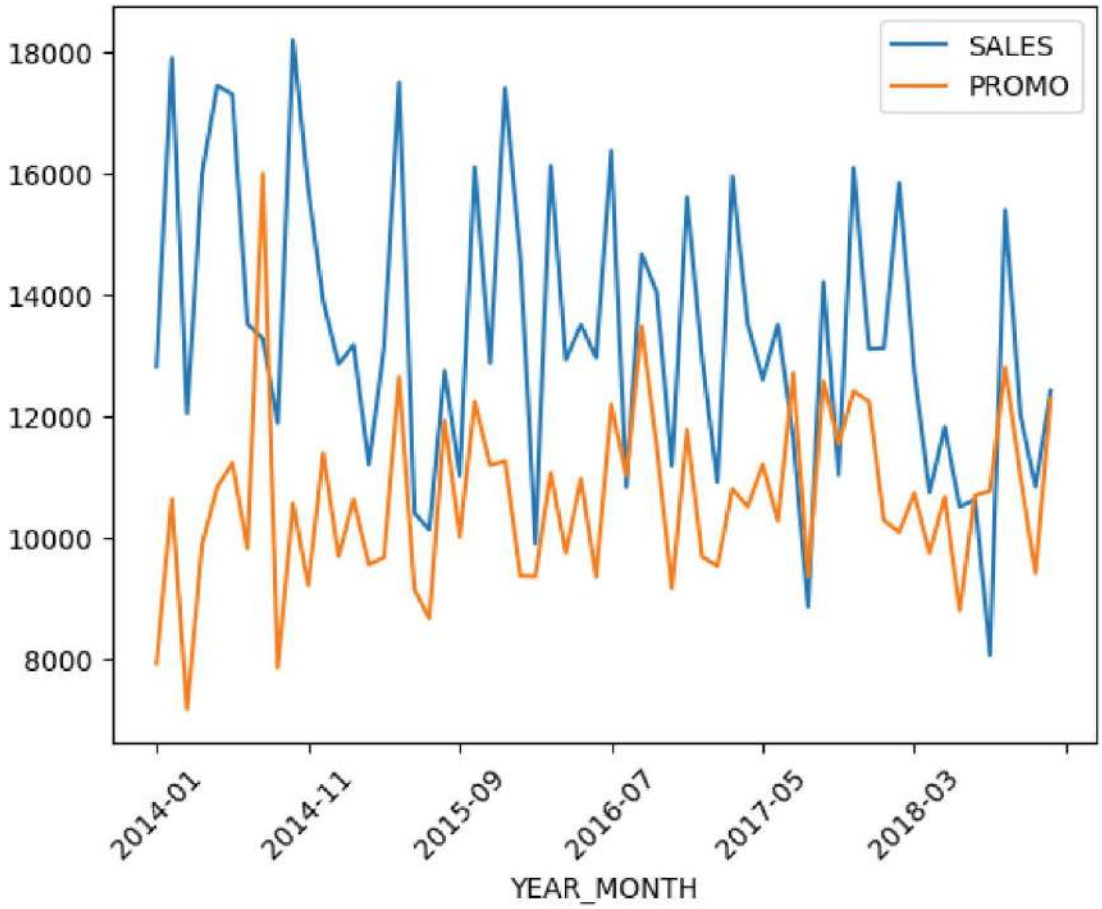


Figure 17-3 The plot

The graph in Figure 17-3 is much more informative than the graph in Figure 17-2. We can now see that some of the low points in sales coincide with low points in promotions. For example, the low point in sales of summer 2018 coincided with a low point in promotions. The peak in sales that follows just after also coincides with a peak in promotions. Although correlation does not imply causation, we may start to think that the PROMO variable has at least some predictive capacity for the SALES. To get a better view of this, let's use Listing 17-4 to create a scatterplot of the two variables.

```
df_sum.plot.scatter(x='PROMO',y='SALES')
```

Listing 17-4 Investigate if there is a correlation, part 2

You will obtain the scatterplot as shown in Figure 17-4.

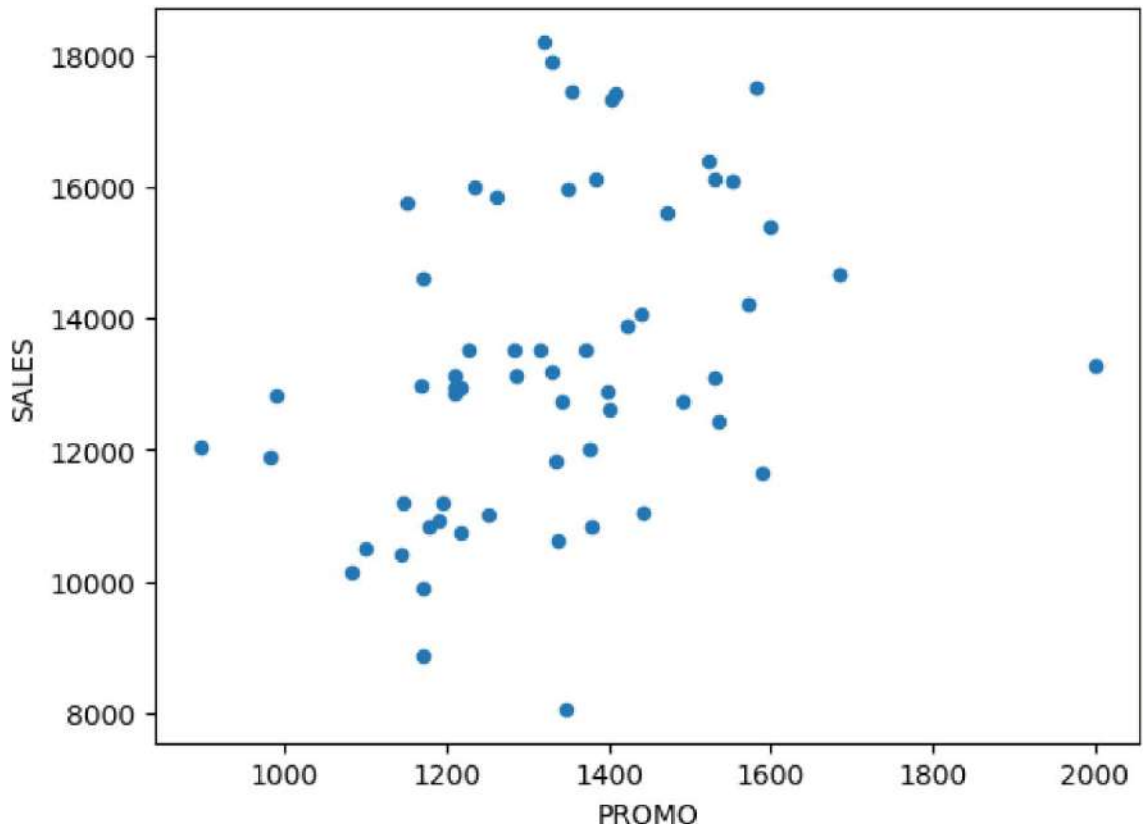


Figure 17-4 The plot

The scatterplot (Figure 17-4) shows a somewhat of a relation between PROMO and sales. If we ignore the outlier on the far right of the scatterplot, we could imagine there being a line going from the bottom left of the graph to the top middle of the graph, indicating a positive linear correlation between promotions and sales: the more promotions there are, the more sales there will be. Let's use the code in Listing 17-5 to confirm this correlation.

```
df_sum.corr()
```

Listing 17-5 Investigate if there is a correlation, part 3

The code in Listing 17-5 will create the correlation matrix as shown in Figure 17-5.

	SALES	PROMO
SALES	1.000000	0.358155
PROMO	0.358155	1.000000

Figure 17-5 The correlation

You'll see that there is a positive correlation of **0.358**, confirming our intuition of a positive linear correlation. Based on this quick exploratory data analysis, we understand that our forecasting model will probably be better if we include an independent variable (X variable) in our forecasting model.

Sales Forecast: Univariate Poisson DGLM

As explained before, the main topic of this chapter is the Dynamic Generalized Linear Model. Let's now start using the pyBATS package to build a forecast for our grocery sales use case.

As a first, simple model, we are going to build a univariate model. This means that for now we will only use the SALES and the variability in SALES to predict itself. This is a technique that has been shown extensively in the earlier chapters in univariate time series modeling; only now we use the DGLM and pyBATS package to estimate this.

Before starting, we need to choose a family. As explained, the G in DGLM is the Generalization of the Linear Model to a number of probability distributions. We are working with sales data, which, in theory, is counting data. After all, we are counting the number of sales: it must always be positive and always a whole number. As the Poisson distribution is the distribution for counting data, it seems to be a good choice to apply a Poisson DGLM first.

Listing 17-6 will use the analysis function of the pybats package to fit the model. We use MLflow to save the model.

```
from pybats.analysis import *
from pybats.point_forecast import *
from pybats.plot import *

import mlflow
mlflow.autolog()
```

```
Y = df_sum['SALES']

k = 1 # forecasting one step ahead
forecast_start = 0 # starting forecast at time step 0
forecast_end = len(df_sum)-1 # ending forecast at the
same time our data ends

mod, samples = analysis(Y,
                        family="poisson",
                        forecast_start=forecast_start,
                        forecast_end=forecast_end,
                        k=k,
                        nsamps=100,
                        prior_length=6,
                        rho=.5,
                        deltrend=0.95,
                        delregn=0.95
                        )
```

Listing 17-6 Univariate forecast, using the Poisson family

Now that the model is estimated, we can inspect the model coefficients using the code in Listing 17-7.

```
print(mod.get_coef())
```

Listing 17-7 Get coefficients

The resulting model coefficients are shown in Figure 17-6. As we have used only the Y variable, we see that only an Intercept has been fitted.

	Mean	Standard Deviation
Intercept	9.43	0.0

Figure 17-6 The coefficients

Let's now use Listing 17-8 to estimate a forecast. As we have set the parameter k (number of time steps) to 1, let's generate a one-step forecast for each data point in the dataset. We are therefore not obliged to create a train and test step as we have done in earlier chapters. This code will also compute a performance score defined as 1 - MAPE (if needed, you can go back to the

chapter on model evaluation for more reference on this). Finally, the code also plots the forecasted values for the last 12 months of the dataset's period.

```
# Compute the forecast for the full period
fcst = median(samples)

# Compute R2 score
from sklearn.metrics import
mean_absolute_percentage_error
percentage_score = 1 -
mean_absolute_percentage_error(Y[-12:],
pd.Series(fcst.reshape(60)[-12:]))
print(percentage_score)

# Plot the forecast of the last year
Y[-12:].plot()
pd.Series(fcst.reshape(len(Y))[-12:]).plot()
```

Listing 17-8 Build the forecast and analyze results

After executing this code, you will see that we have obtained a performance score of **0.833**. This sounds not too bad, but when looking at the obtained graph in Figure 17-7, we see that the model did not catch much of the fluctuation: it basically only captured the overall trend in the data.

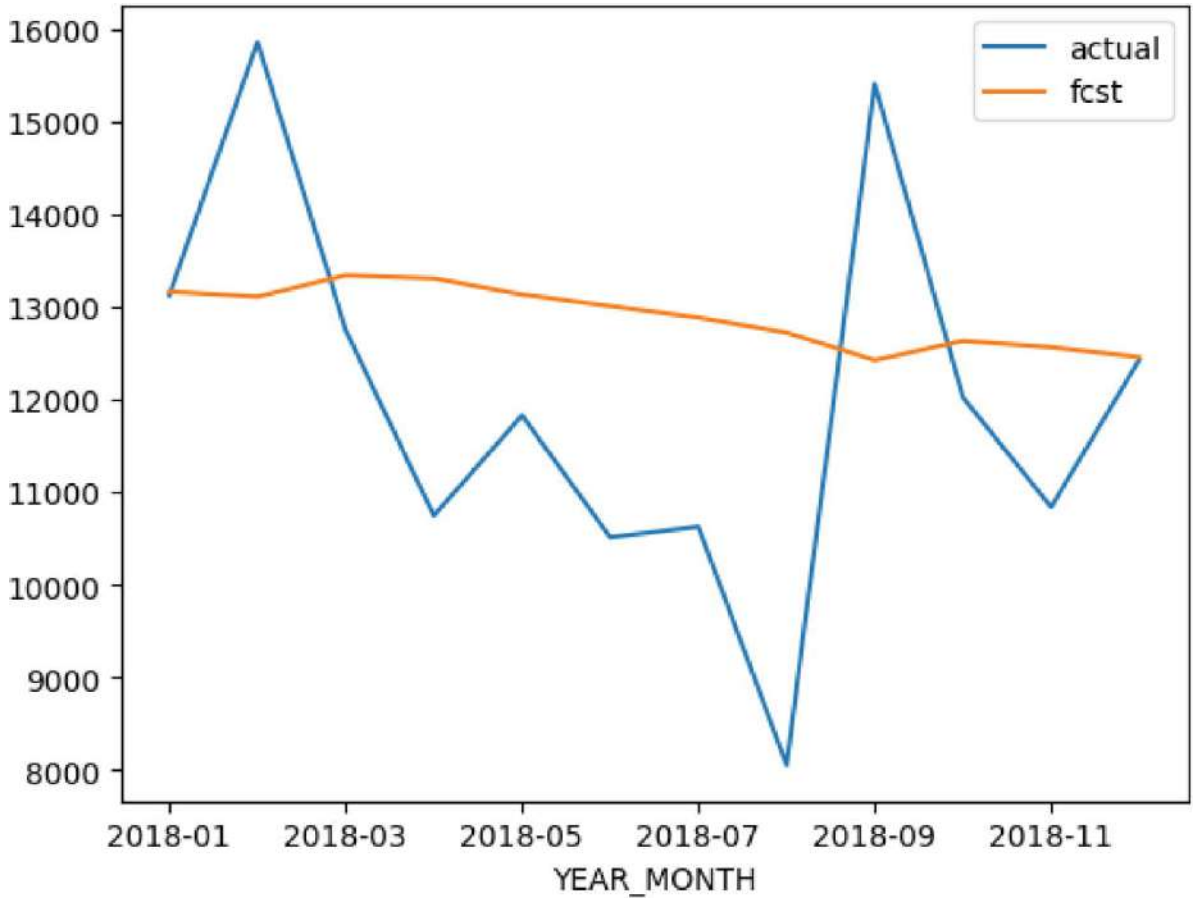


Figure 17-7 The prediction plot

Sales Forecast: Normal DGLM with X Variable

As we have seen in the EDA, the additional variable PROMO seems to have an additional explanatory value for this forecasting use case. In this second model, let's see how to add an external variable as a predictor to our DGLM.

In this second example, we are also switching to normal; as we are forecasting a monthly sum of multiple products, we are actually modeling a sum of Poisson distributions. In our case, this may actually very well be approached by a normal distribution. In statistics, if possible, we tend to assume a normal distribution when possible, as this opens up a larger number of statistical methods.

The code in Listing 17-9 fits this new model, computes the score, creates the graph, and, as MLflow autologging has been turned on earlier in the notebook, even stores the result in MLflow.

```
X = df_sum['PROMO'].values
```

```
mod, samples = analysis(Y.values,X,
                        family="normal",
                        forecast_start=forecast_start,
                        forecast_end=forecast_end,
                        k=k,
                        nsamps=100,
                        prior_length=20,
                        rho=.5,
                        deltrend=0.95,
                        delregn=0.95
                        )

# Estimate the forecast
fcst = median(samples)

# Compute the score
percentage_score = 1 -
mean_absolute_percentage_error(Y[-12:],
pd.Series(fcst.reshape(60)[-12:]))
print(percentage_score)

# Plot the forecast of the last year
Y[-12:].plot()
pd.Series(fcst.reshape(len(Y))[-12:]).plot()
scaled_x_future = X[-12:] * 8
pd.Series(scaled_x_future).plot()
```

Listing 17-9 Forecast with an additional X variable, using normal family

This improved model has obtained a performance score (1-MAPE) of **0.867**, which is an interesting improvement coming from 0.833 with the previous model. Even more interesting is to see what the graph in Figure 17-8 shows. Although the performance improvement isn't huge, we can see that this second model actually fits much better to the data, as it benefits from the additional input of the PROMO data.

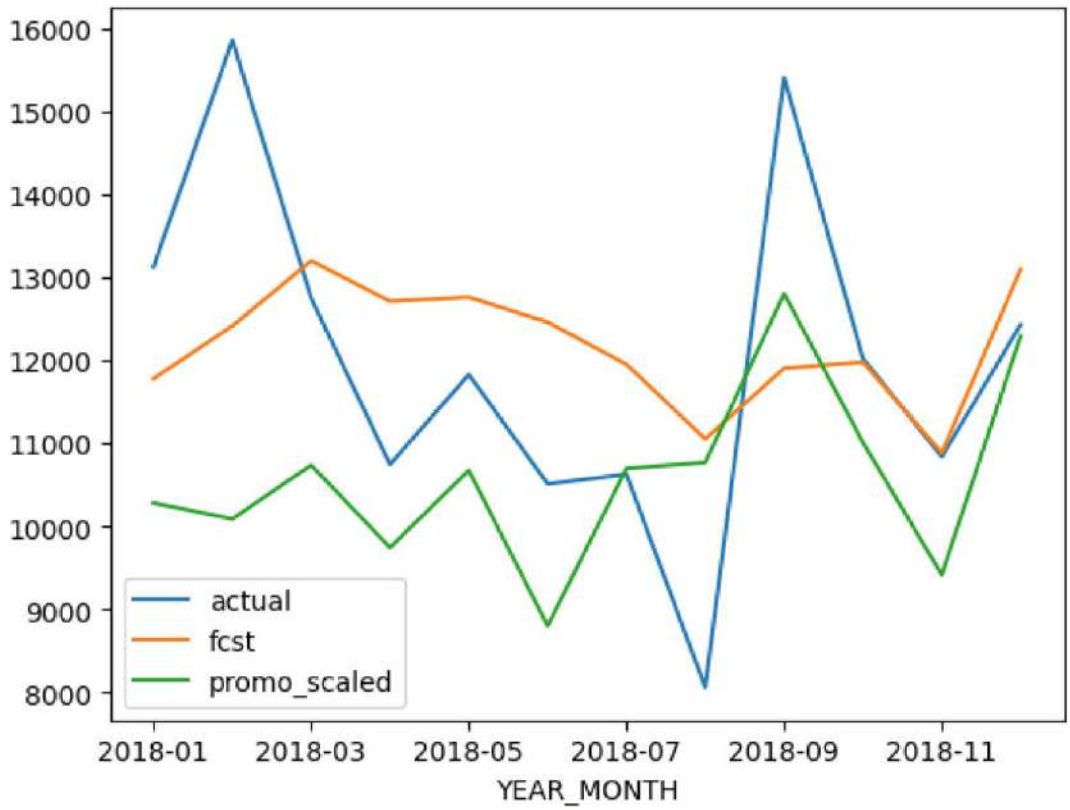


Figure 17-8 The better prediction plot

The forecast in Figure 17-7 was basically a flat trend line with no fluctuation, but Figure 17-8 shows that the model has become much more than just a trend line. For example, it estimates two low points in 2018-08 and 2018-11. Certainly, it does not estimate them low enough, but it is already a good sign of model improvement that general tendencies are taken into account.

As we have covered hyperparameter tuning extensively in previous chapters, I leave it as an exercise to try and play around with the numerous hyperparameters of the pyBATS package, like prior_length, rho, deltrend, and delregn, but also other hyperparameters that you may get from the pyBATS documentation (*https://lavinei.github.io/pybats/analysis.html*). With all this tuning, you may very well be able to come up with an even better fit!

Key Takeaways

- The two main paradigms in statistics are frequentist and Bayesian.
- Whereas frequentists focus on respecting experimental design and focus on giving one definitive answer, Bayesian statistics focus on quickly estimating

distributions with a best effort, which allows them to change their estimates as more data come in.

- The DGLM, the Dynamic Generalized Linear Model, is a Bayesian model for forecasting.
  - The Poisson family of DGLMs can be used when forecasting a positive integer variable.
  - The normal family of DGLMs can be used when forecasting a continuous numeric target variable, but the normal distribution can also sometimes be used as a replacement for other distributions.
- The pyBATS is an easy-to-use package that implements DGLMs.
- The analysis function in pyBATS is a good entry point for doing your model specification and training the forecast.

Part V
Neural Networks

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_18

18. Neural Networks

Joos Korstanje[1]
(1) Maisons Alfort, France

In previous chapters, you have discovered a number of supervised machine learning models, starting from Linear Regression to Gradient Boosting. In this chapter, you'll discover **Neural Networks**.

The scope of Neural Networks is huge. The version that you'll see in this chapter is a subgroup called **fully connected neural networks**. Those neural networks are intuitively quite close to the previous supervised models. In the following chapters, you'll also discover **Recurrent Neural Networks**, and you'll see a specific network called the **LSTM** Neural Network.

The goal of this chapter is to get familiar with the general idea of Neural Networks.

So let's start with a schematic overview of the model in Figure 18-1.

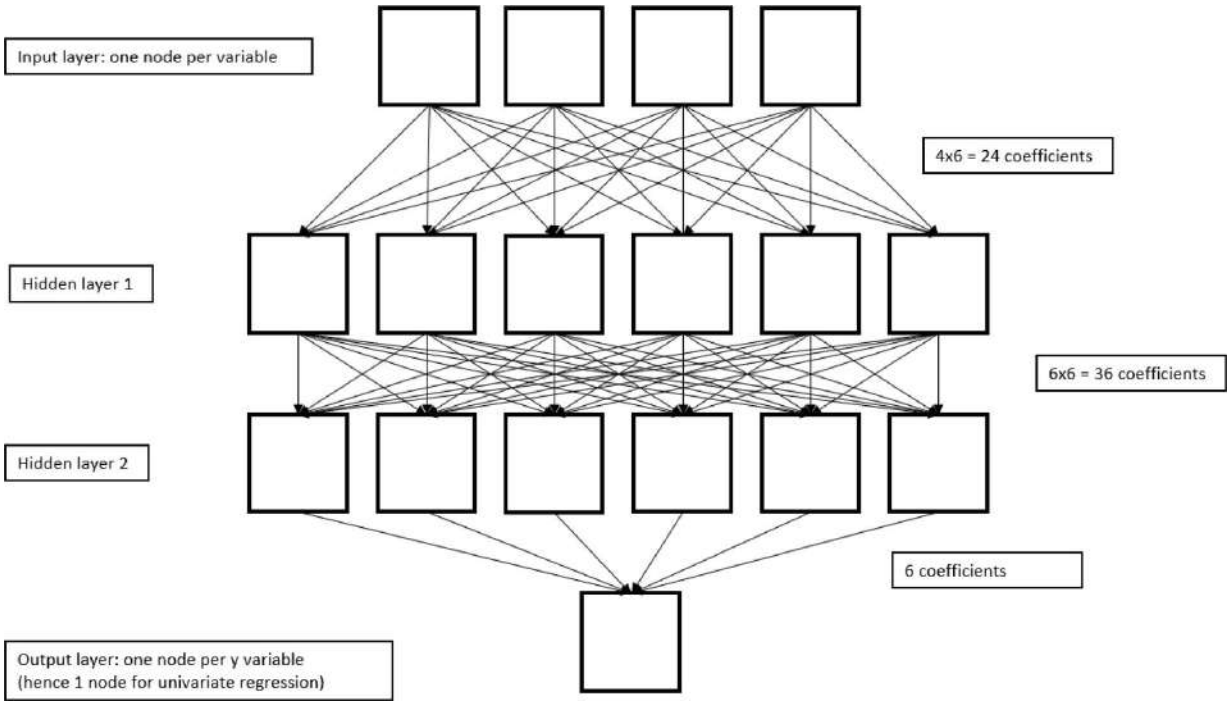


Figure 18-1 Fully connected Neural Networks schema

Fully Connected Neural Networks

You can read this schema from top to bottom. The model always contains an input layer and an output layer. The input layer of the network contains one node for each variable. The output layer contains the same number of nodes as there are output variables. In the case of univariate regression, there is one output variable (the target variable y), but sometimes, there can be multiple outcome variables at the same time, as you've seen, for example, in the VAR model. In this case, there would be multiple nodes in the output layer.

After the input, you go to the first hidden layer. Roughly speaking, the values of the hidden layer are computed by **multiplying the input by a weight** and then **passing the value through an activation function**. This combination of multiplication and activation functions repeats itself in each node until arriving at the output node. This model is called a fully connected model because each node is connected to each node in the following layer. There are other shapes of architecture in which this is not the case: we'll see some examples in the next two chapters.

Activation Functions

When you choose the architecture of your neural network, you also need to choose the type of activation functions that you want at each location. The activation functions are applied in each node of the network before going to the next layer.

Three relatively common choices of activation function are **tanh**, **ReLU**, and **sigmoid**. You can see their specific behaviors in Figure 18-2. The reason that we use activation functions is that they allow the model to fit more complex problems with fewer nodes, as they add nonlinearity into the computation.

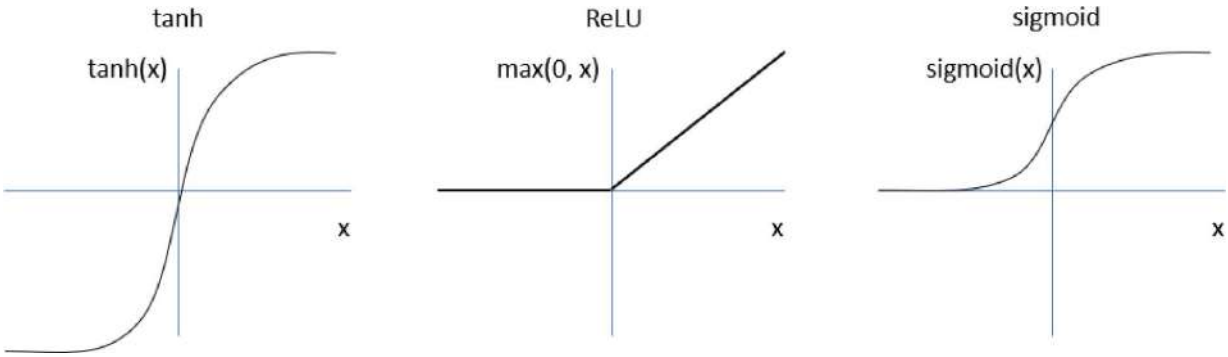


Figure 18-2 Three common activation layers

Without getting into too much detail on the activation layers, it is important to know how to use them. ReLU is often a good choice for starting, whereas tanh is often used for Recurrent Neural Networks that you'll see in the next chapter.

The choice for activation layers, just like many things in fitting neural networks, is a choice that is not obvious: trial and error to obtain a working neural network is the only way to go. Sometimes, if you're lucky, you can also find examples in the literature of neural networks that are applied to a similar case as yours. The important thing to remember is that it takes time and effort to come up with something accurate, but if you're successful, you can obtain great performances.

The Weights: Backpropagation

So while moving an input value through your neural network, you encounter something else besides the activation layer: the weights. These weights are estimated at the time of model fitting, through an algorithm called the backpropagation algorithm.

The backpropagation algorithm computes the gradient of the loss function with respect to the weights of the network, one layer at a time. It then iterates backward from the last layer, which is why it is called backpropagation. It is an

efficient way for fitting the huge number of weights that is needed for a neural network. Yet, it is still a complex algorithm, and it takes time to compute.

Optimizers

The backpropagation algorithm's performances depend on the choice of the optimizer. This is a decision that has to be made again by the modeler. There are many optimizers available. The first available optimizer was gradient descent, as described in the previous paragraph.

SGD, short for **Stochastic Gradient Descent**, is an improved version of the gradient descent algorithm. It is more efficient as it computes not on the whole dataset at every iteration, but only a subset of the dataset.

Two improvements have been made on SGD, and those have delivered two new optimizers: **RMSProp** and **AdaGrad**. An even more improved optimizer called Adam uses a combination of those two.

Although I won't go into the deep mathematical difference between optimizers, it is important to know that there are different optimizers available. Also, a lot of work is still ongoing, and newer and better versions may well arise in the near future. The important thing to retain here is that the choice for an optimizer is a hyperparameter to choose by the model developer. Playing around with different optimizers may just help you to add those last points of accuracy to your model.

Learning Rate of the Optimizer

The next thing that you need to choose as a hyperparameter is the learning rate of your optimizer. You can see it as follows. Your optimizer is going to find the right way to move in and it is going to take a step in that direction. Yet, those steps can be either large or small steps. This depends on the learning rate that you choose.

Choosing large learning rates means that you take large steps in the right direction. Yet, a risk is that you take too large steps, and therefore, you step over the optimum and thus miss it.

A small learning rate, on the other hand, may let you get stuck in a local optimum and not be able to get out, as your step size is too small. You may also take a long time to converge.

Hyperparameters at Play in Developing a NN

Now until here, the overview is not much different than what we've seen before, for example, in Random Forests and XGBoost. Yet, there is a big difference in developing Neural Networks: the number of hyperparameters and the time of training the model are so huge that it becomes impossible to simply launch a hyperparameter optimization tool.

This is all due to the backpropagation algorithm used for fitting the neural network. In short, this algorithm goes back and forth through the network, and it updates the weights. It does not pass all the data at once, but it passes batch by batch until all the data has been passed. When all data has been passed, this is called an **epoch**.

As you have seen, for neural networks, a lot of hyperparameter tuning has to be done. A lot of this is done by hand, using specific tools to judge the quality of the model fit. Building Neural Networks is much more complex than fitting classical machine learning models.

As an overview, the main neural network's hyperparameters are

- The number of layers.
- The number of nodes in each of the layers.
- The **optimizer** is the method to update the weights throughout backpropagation. One of the standard optimizers is Adam, but there are many more.
- The **learning rate** of the optimizer influences the step size of the optimizer. Too small learning rates make the steps toward the optimum too small, and that can make it too slow, or you can get blocked in a local optimum.

Two more hyperparameters that you need to choose are less related to mathematics, but they are still very important:

- The **batch size** specifies the number of individuals that will be used for each pass through the algorithm. If it's too large, you may run out of RAM, but if it's too small, it may be too slow.
- The number of **epochs is** the number of times that the whole dataset is passed through the network. The more epochs, the longer the model continues training. But this should really depend on what moment you reach the optimum.

Introducing the Example Data

Before moving onto the fitting and optimization of the neural network, let's introduce the example data and two data preparation methods. In this chapter, we'll be using the example data from the Max Planck Institute for

Biogeochemistry. The dataset is called the Jena Climate dataset. It contains weather measures like temperature, humidity, and more, recorded every ten minutes.

You can download the data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/jena_climate_2009_2016.csv.zip. You can also download and unzip it and use pandas to import it with Listing 18-1.

```
import keras
import pandas as pd
from zipfile import ZipFile
import os

uri = "https://storage.googleapis.com/tensorflow/tf-
keras-datasets/jena_climate_2009_2016.csv.zip"
zip_path = keras.utils.get_file(origin=uri,
fname="jena_climate_2009_2016.csv.zip")
zip_file = ZipFile(zip_path)
zip_file.extractall()
csv_path = "jena_climate_2009_2016.csv"

df = pd.read_csv(csv_path)
del zip_file

df = df.drop('Date Time', axis=1)
cols = ['p',  'T', 'Tpot', 'Tdew', 'rh', 'VPmax',
'VPact', 'VPdef', 'sh', 'H2OC', 'rho', 'wv', 'mwv',
'wd']
df.columns = cols
```

Listing 18-1 Importing the data

In this example, we'll do a **forecast of the temperature 12 hours later**. To do this, we create lagged variables for the independent variables and make a correct dataframe. We have quite a lot of data, so we can add multiple lagged values to add the most information possible to the model. We'll add the lags for 72 lags (72 times 10 minutes, and then for 84, 96, 108, 120, and 132 steps back in time).

```
y = df.loc[2*72:,'T']
lagged_x = []
```

```
for lag in range(72,2*72,12):
  lagged = df.shift(lag)
  lagged.columns = [x + '.lag' + str(lag) for x in
lagged.columns]
  lagged_x.append(lagged)

df = pd.concat(lagged_x, axis=1)
df = df.iloc[2*72:,:] #drop missing values due to lags
```

Listing 18-2 Creating the lagged dataset

Specific Data Prep Needs for NN

Neural Networks are very sensitive to problems with the input data. Let's have a look at two tools that are often useful in data preparation.

Scaling and Standardization

A Neural network will not be able to learn if you do not standardize the input data. Standardizing means *getting the data onto the same scale*. Two examples for this are a standard scaler and the MinMax scaler:

1. A standard scaler maps a variable to follow a standard normal distribution. That is, the new mean of the variable is 0 and the new standard deviation is 1. It is obtained by taking each value minus the average of the variable and then dividing it by the standard deviation.
2. The MinMax scaler brings a variable into the range of 0–1 by subtracting the variables minimum from each value and then dividing it by the range of the variable.

You can apply a scaler using the syntax in Listing 18-3.

```
# apply a min max scaler
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df = pd.DataFrame(scaler.fit_transform(df), columns =
cols)
```

Listing 18-3 Fitting the MinMaxScaler

Principal Component Analysis (PCA)

The **PCA** is a machine learning model in itself. It comes from the family of dimension reduction models. It allows taking a dataset with a large number of variables and *reducing the number of variables to a projection onto a number of dimensions*. Those dimensions will contain the larger part of the information and will contain much less noise.

Making sure that the input data contains less noise will strongly help during the fitting of your Neural Network. So, how does the PCA work? The idea is to make new variables, called **components**, based on combinations of strongly correlated variables. In Figure 18-3, you can see a hypothetical example with two variables, Rain and Humidity, that you can expect to be correlated. The first principal component captures the most possible variation.

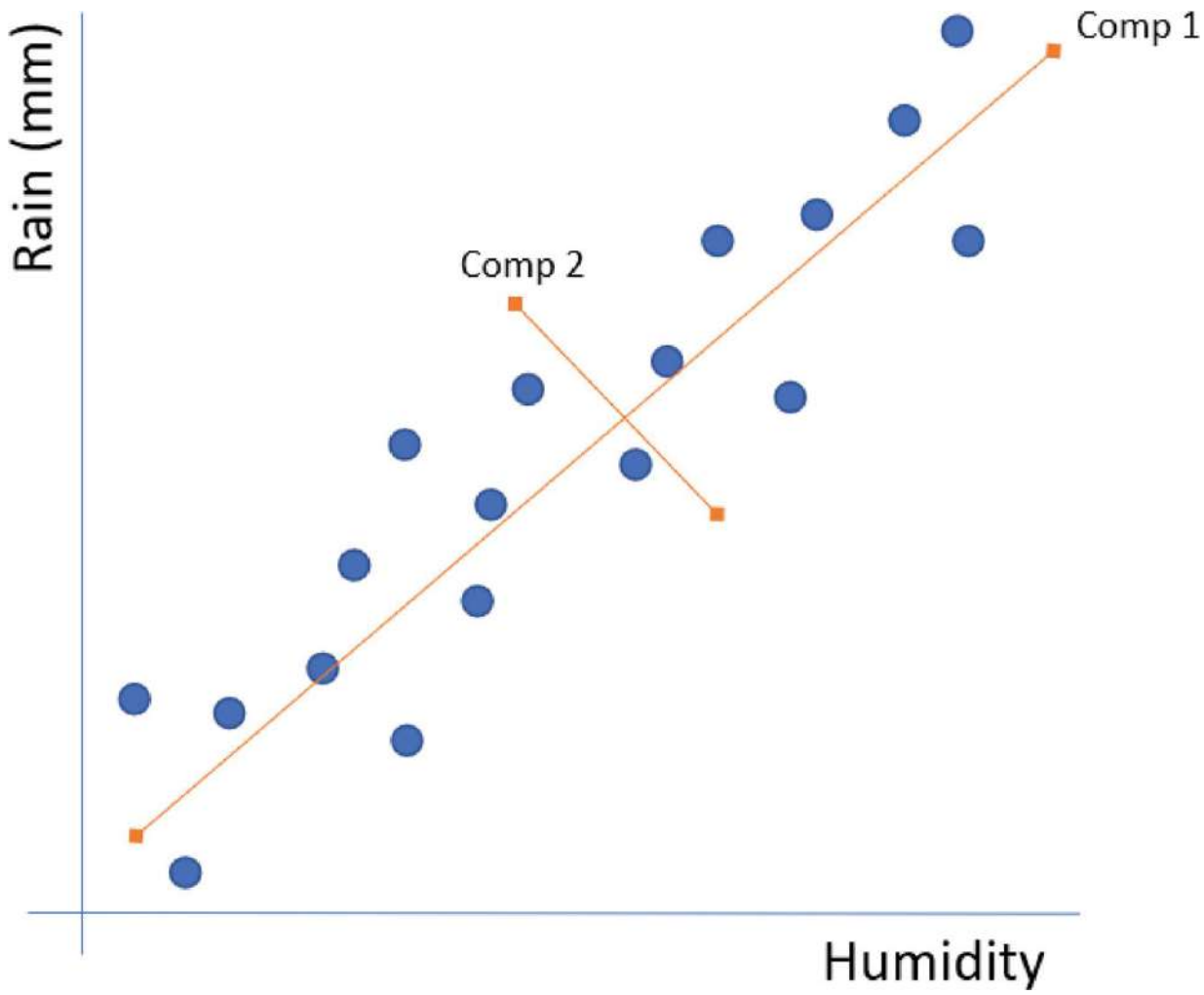


Figure 18-3 PCA

The component is a mathematical formula that is a linear combination of the original variables. You can use this score as a new variable. If the principal

component is capturing a lot of the original variables, it can be interesting to use the component in your machine learning model rather than the original variables.

To fit a PCA, you generally start with a PCA with all the components. This is done in Listing 18-4. Let's enable MLflow autologging at this step so that the PCA can be saved into the mlruns directory together with all upcoming Neural Network models.

```
import mlflow
mlflow.autolog()

# Fit a PCA with maximum number of components
from sklearn.decomposition import PCA
mypca = PCA()
mypca.fit(df)
```

Listing 18-4 Fitting the full PCA

You use this PCA to make a **scree plot**. A scree plot is one of multiple tools used to decide on the number of components to retain. At the point of the elbow, you choose the number of components. You can use Listing 18-5 to make a scree plot. It is shown in Figure 18-4.

```
# Make a scree plot
import matplotlib.pyplot as plt
plt.plot(mypca.explained_variance_ratio_)
```

Listing 18-5 Plotting the full PCA

You can see the scree plot in Figure 18-3.

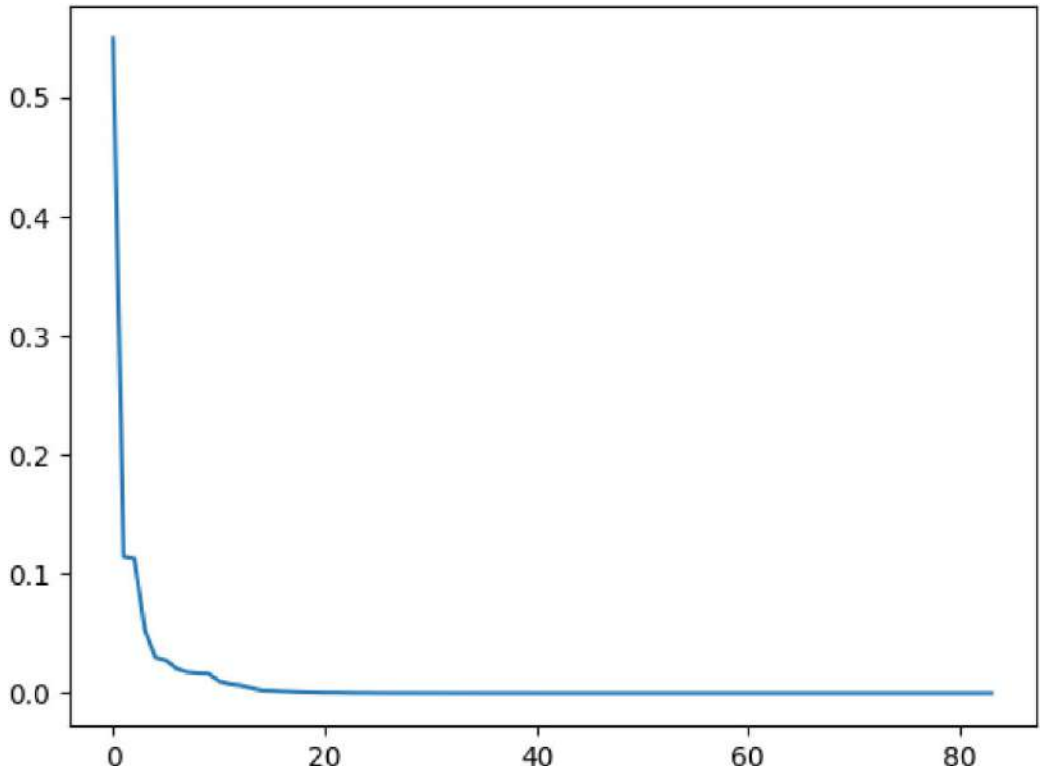


Figure 18-4 Scree plot

In this plot, the x axis shows the components from the first component to the last. The y variable shows the amount of variation that is captured in those components. So, you clearly see that the first *five to ten components* have much more information in them than the higher components (those more to the right). You could see the elbow being somewhere around 5, but it seems a good idea to retain 10 components so that we retain almost all information but while having 10 variables rather than over 80. Finally, you refit the PCA with ten components and transform the data using Listing 18-6.

```
mypca = PCA(10)
df = mypca.fit_transform(df)
```

Listing 18-6 Fitting the PCA with ten components

The Neural Network Using Keras

Now that we have made sure that our data are correctly prepared, we can finally move on to the actual neural network.

Building Neural Networks is a lot of work, and I want to find a good balance in showing you the way to get started and to work on improving your network rather than just showing a final performant network.

A great first start is to start with a relatively simple network and work your way up from there. In this case, let's start with a network using two dense layers with 64 nodes. That would make the architecture look as follows.

For the other hyperparameters, let's take things that are a little bit standard:

- Optimizer Adam
- Learning rate 0.01
- Batch size 32 (reduce this if you don't have enough RAM)
- Epochs 10

Before starting, let's do a train-test split as shown in Listing 18-7.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test =
train_test_split(df, y, test_size=0.33,
random_state=42)
```

Listing 18-7 Train-test split

Now, you build the model using the **Keras library** using the following code. First, you specify the architecture using Listing 18-8. Keras is the go-to library for neural networks in Python.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import random
random.seed(42)

simple_model = Sequential([
  Dense(64, activation='relu', input_shape=
(X_train.shape[1],)),
  Dense(64, activation='relu'),
  Dense(1),
])
```

Listing 18-8 Specify the model and its architecture

You can obtain a summary to check if everything is alright using Listing 18-9.

```
simple_model.summary()
```

Listing 18-9 Obtain a summary of the model architecture

Then, you compile the model using Listing 18-10. In the compilation part, you specify the optimizer and the learning rate. You also specify the loss, in our case the mean absolute error.

```
simple_model.compile(
  optimizer=keras.optimizers.Adam(learning_rate=0.01),
  loss='mean_absolute_error',
  metrics=['mean_absolute_error'],
)
```

Listing 18-10 Compile the model

And then you fit the model using Listing 18-11. At the fitting call, you specify the epochs and the batch size. You can also specify a validation split so that you obtain a train-validation-test scenario in which you still have the test set for a final check of the R2 score that is not biased by the model development process.

```
smod_history = simple_model.fit(X_train, y_train,
          validation_split=0.2,
          epochs=10,
          batch_size=32,
          shuffle = True
)
```

Listing 18-11 Fit the model

Be aware that fitting neural networks can take a lot of time. Running on a GPU is generally fast but not always possible depending on your computer hardware.

Now, the important part here is to figure out whether or not this model has learned something using those hyperparameters. There is a key graph that is going to help you infinitely while building neural networks. You can obtain this graph using Listing 18-12 and see it in Figure 18-5.

Be aware that you may get slightly different results. Setting the random seed is not enough to force randomness to be the same in Keras. Although it is possible to force exact reproducibility in Keras, it is quite complex, so I prefer to leave it out and accept that results are not 100% reproducible. You can check out these instructions for more information on how to fix the randomness in Keras: https://keras.io/getting_started/faq/#how-can-i-obtain-reproducible-results-using-keras-during-development.

```
plt.plot(smod_history.history['loss'])
plt.plot(smod_history.history['val_loss'])
plt.title('model loss')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

Listing 18-12 Plot the training history

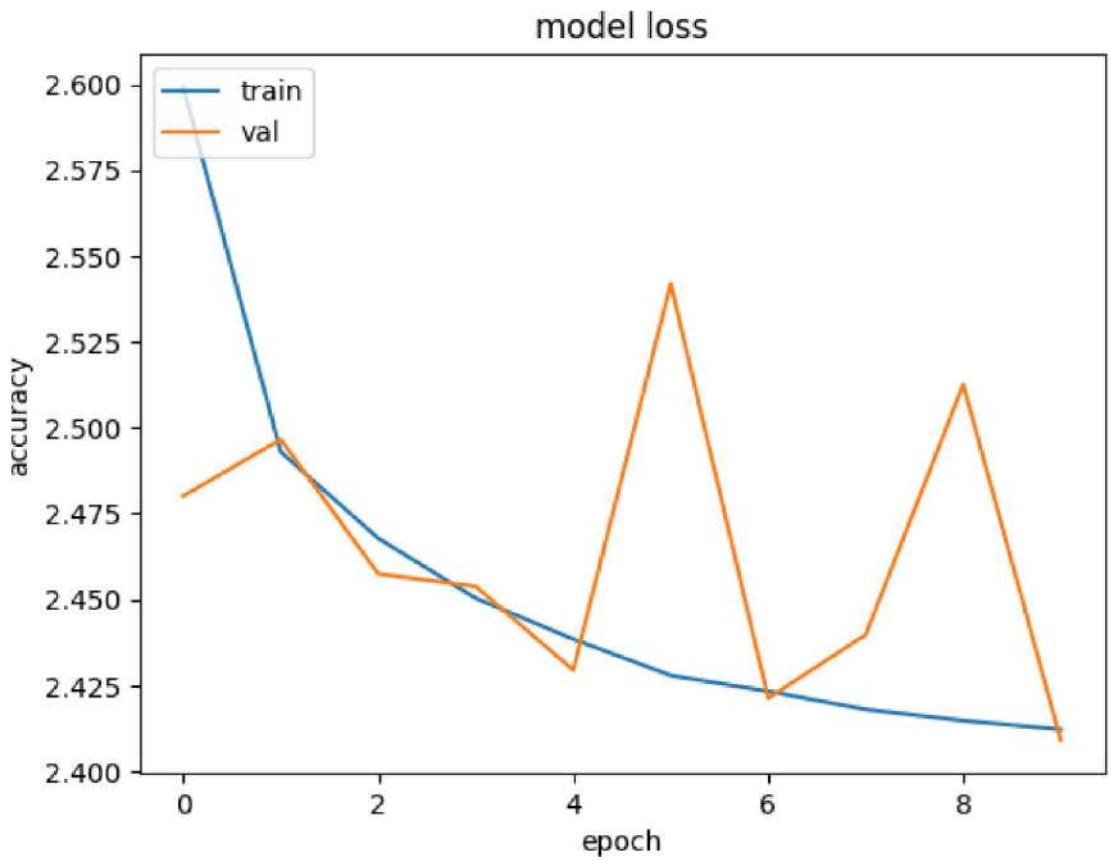


Figure 18-5 History plot that shows a model that doesn't learn too well

In Figure 18-6, you see a few examples of graphs that you are or are not looking for. In graph 1, you see the ideal curve that you are trying to obtain.

In graph 2, you see a neural net that is learning well, and it also shows overfit that happens in practice: the training loss goes down a lot, and the validation loss does too. Yet, at some point, you need to reduce the number of epochs, or else your model will start to overfit. If you remember from a previous chapter, overfitting means that the model is learning on specifics from the training data that are actually noise, and therefore the learned trends do not generalize.

In the third graph, you see a neural net that is not learning anything. You can see this because the training loss is not going down. In this case, you may either have a problem with your data, or you might want to test out a very different network setup.

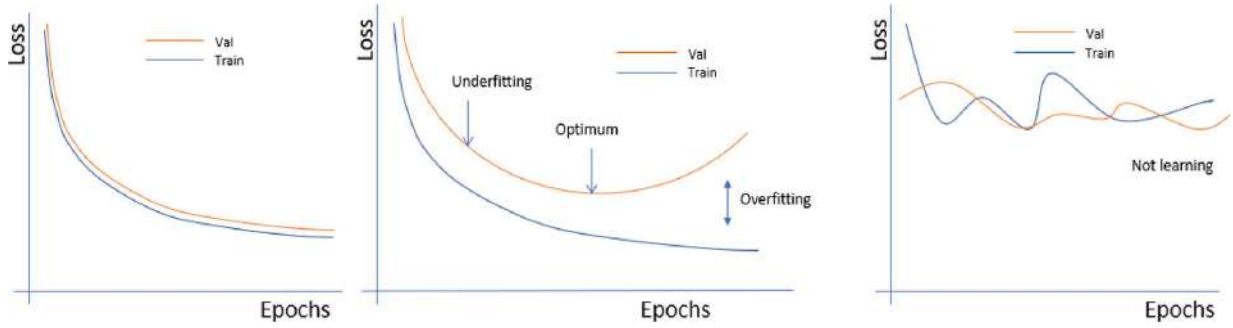


Figure 18-6 Ideal history plots

Now in our case, what happened? Most likely a case 3: the model is not learning anything. And why is the model not learning? We should try out whether there are just too few neurons to learn it. From here on, you know that you need to increase the number of layers and neurons and see whether this is getting any better.

From here on, it is honestly a real process of trial and error. The first model that you should try is generally a very simple one, and it will not be learning enough. The next model that you should search for is a model that does learn, even if, in the worst case, it is overfitting. Then, as the last step, you fine-tune until you obtain a graph close enough to the graph on the left in Figure 18-6, and you obtain an error score that is good enough for you.

After trial and error, a better model architecture that has been found is the one in Listing 18-13. Far from saying that this is the best model, at least this model is able to obtain a relatively good R2 score on the test data of **0.915**. Feel free to try and tweak this model more and see what happens and whether you can improve it.

Listing 18-13. A better architecture

```
random.seed(42)
model = Sequential([
  Dense(256, activation='relu', input_shape=
(X_train.shape[1],)),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
```

```
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(256, activation='relu'),
  Dense(1), ])
model.compile(
  optimizer=keras.optimizers.Adam(learning_rate=0.001),
  loss='mean_absolute_error',
  metrics=['mean_absolute_error'],
)

history = model.fit(X_train, y_train,
          #validation_data=(X_test, y_test),
          validation_split=0.2,
          epochs=100,
          batch_size=32,
          shuffle = True
)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()

preds = model.predict(X_test)
print(r2_score(preds, y_test))
```

The resulting graph is shown in Figure 18-7.

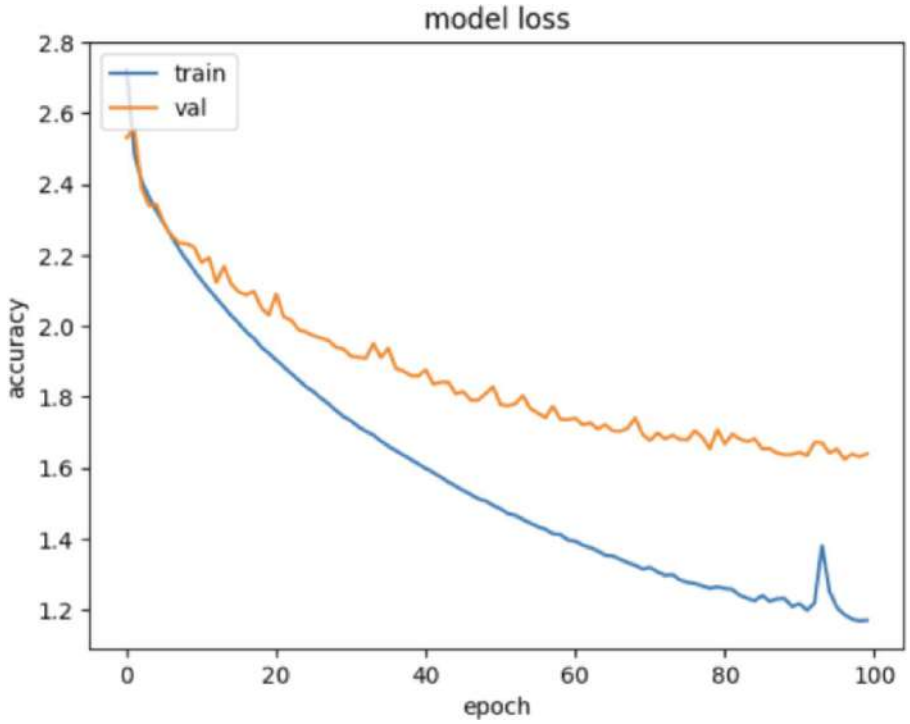


Figure 18-7 History plot of the more complex model

Conclusion

The really difficult thing with fitting Neural Networks is that it is very hard to know if you've achieved something that could be improved or whether it's quite good already. For now, this chapter has shown how to work on improving from a very simple network and trying to get into an optimized version.

There are some libraries that you could use for hyperparameter optimization. Of course, you could even grid search your architecture. Just know that there is no wonder cure for fitting neural networks. It requires much more effort and dedication than classic machine learning models. Training times are very long, and you will need significant computing power to start thinking about such optimization. Yet, on the upside, neural networks can sometimes obtain results that are much more powerful than classical machine learning, and it is an important tool to have in your machine learning toolbox.

There are a lot of developments in the scientific field of neural networks and AI. In the coming two chapters, you'll discover neural network structures that are more advanced than the dense structure, but that may apply very well in the case of forecasting.

Key Takeaways

- Neural Networks are a powerful supervised machine learning model, but it is more difficult to build them than classical machine learning models.
- The hyperparameters are
  - The number of layers
  - The number of nodes in each of the layers
  - The **optimizer**
  - The **learning rate**
  - The **batch size**
  - The number of **epochs**
- The graph showing the descent of the train and validation loss is a very important aspect in developing neural networks.
- PCA is a model for dimension reduction. It can also be used for data preparation, especially when there are many correlated variables.
- Scaling the input data is necessary for neural networks. Some commonly used methods are the standard scaler and the MinMax scaler.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_19

19. RNNs Using SimpleRNN and GRU

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, you'll discover a more advanced version of Neural Networks called **Recurrent Neural Networks**. There are three very common versions of RNNs: SimpleRNN, GRU (Gated Recurrent Unit), and LSTM (Long Short-Term Memory). In practice, SimpleRNNs are hardly used anymore for a number of problems that I'll talk about later. Therefore, I have regrouped SimpleRNN and GRU in this chapter, and LSTMs have their own chapter.

What Are RNNs: Architecture

So, what are Recurrent Neural Networks, and what makes them different from the dense Neural Networks that you've seen just before? The following graph shows a node of a fully connected net against an example with a SimpleRNN node.

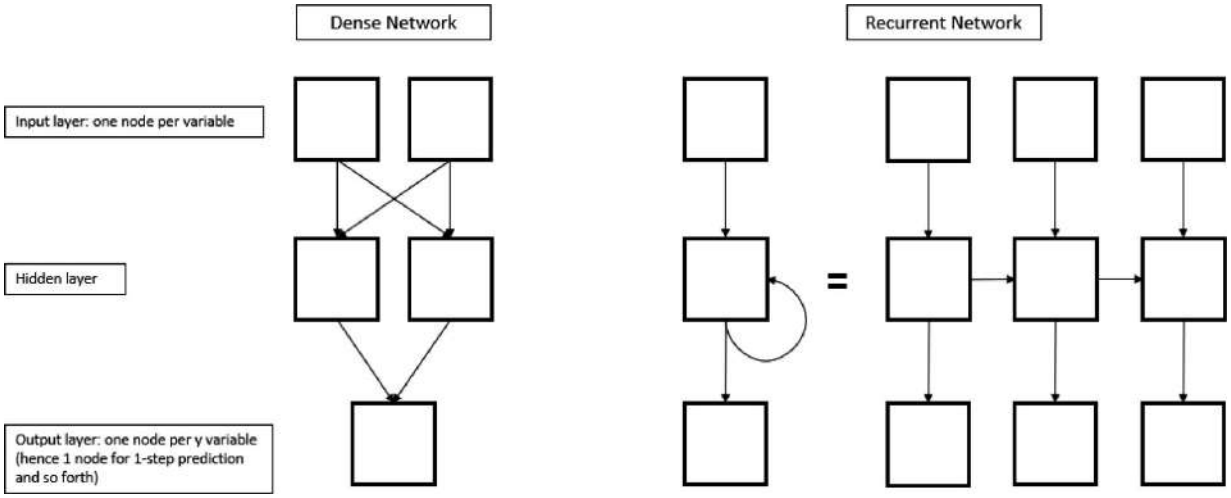


Figure 19-1 Recurrent network vs. dense network

As you can see in Figure 19-1, the big difference in the RNN block is that there is a feedback loop. Where each input of a fully connected network is completely independent, the inputs of an RNN have a feedback relation with each other. This makes **RNNs great for data that has sequences**, for example:

- Time series
- Written text (sequences of words)
- DNA sequences
- Geolocation sequences

Inside the SimpleRNN Unit

Due to this difference in architecture, there is also a difference inside the units. As you can see, there is not just one input in each unit, but there are two inputs. Those two inputs have to be taken into account. As a schematic overview this looks as shown in Figure 19-2, in which X is the input at time t, Y is the target variable at time t, and the a's are the weights that go from left to right in Figure 19-1, so the weights that make the model fit as a sequence through time.

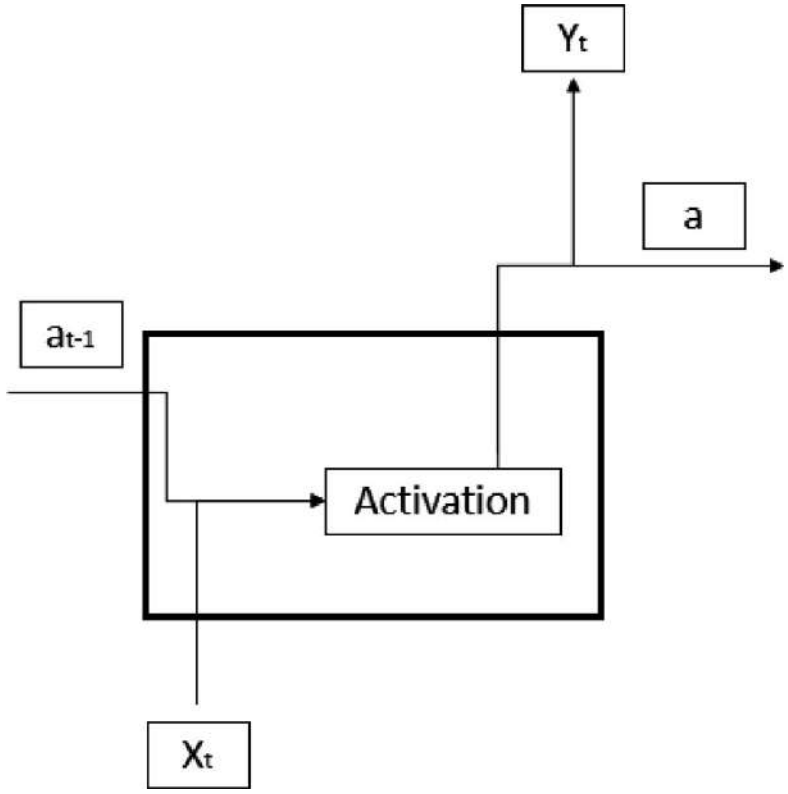


Figure 19-2 RNN cell

In the previous chapter, you have mainly seen the **ReLU** activation layer being used. In Recurrent Neural Networks, however, the **tanh** layer is the standard. The reason for this is that for long sequences, the ReLU layer suffers from the exploding gradient problem: the repeated multiplication with weights is acting like an exponentiation that makes it explode. The tanh activation layer does not have this problem, as the values are forces to stay between -1 and 1.

The Example

The RNN learns on sequences. Therefore, we will have to adapt the problem statement. In this chapter, let's work with the same data as in the previous chapter to get a good feel of how the data preparation and use of the data are different from fully connected architectures.

Just to remember quickly: you have a dataset with measures on weather data and we try to predict the temperature 12 hours (72 timesteps of ten minutes) into the future.

Now, what we did in the fully connected model is to create lagged variables. There was one y variable (which was not lagged), and the independent variables were lagged values of the y variable and a lot of other variables. The first lag was at 72 time steps, so that the model would use the data from 12 hours ago to predict now (having a history from now to 12 hours back to predict the now is equivalent to having data from now to predict 12 hours later).

Predicting a Sequence Rather Than a Value

In the RNN, this is not what we should do, as the RNN learns sequences. A big jump of 72 time steps is not really respecting the sequential variation. Yet, we do not just want to predict one time step later, as that would mean that we predict the temperature in ten minutes from now, not really interesting.

What we will do is *create a matrix of y variables*, with lags as well. Before, we wanted to predict one value 72 time steps into the future, but let's model to predict each of those 72. Even though it might be possible to do it with one y variable, this is also an interesting case of *multistep forecasting*, which is often useful.

Univariate Model Rather Than Multivariable

A second thing that we change from the previous model is that in this case, we will use only the temperature data and not the other variables. This may make the

task slightly harder to accomplish, but it'll be easier to get your head around the use of sequences and RNNs. However, you must know that it is possible to add other explanatory variables into an RNN. For forecasting tomorrow's temperature, you may want to use not only today's temperature but also today's wind direction, wind speed, and humidity, for example. In this case, you could add a third dimension to the input data.

Preparing the Data

You can prepare the data using the following steps. The first step is to import the data using Listing 19-1.

```
import keras
import pandas as pd

from zipfile import ZipFile
import os

uri = "https://storage.googleapis.com/tensorflow/tf-
keras-datasets/jena_climate_2009_2016.csv.zip"
zip_path = keras.utils.get_file(origin=uri,
fname="jena_climate_2009_2016.csv.zip")
zip_file = ZipFile(zip_path)
zip_file.extractall()
csv_path = "jena_climate_2009_2016.csv"

df = pd.read_csv(csv_path)
del zip_file
```

Listing 19-1 Importing the data

The next step is to delete all columns other than the temperature, as we are building a univariate model. This is done in Listing 19-2.

```
df = df[['T (degC)']]
```

Listing 19-2 Keep only temperature data

Now, as you remember from Chapter 16, Neural Networks need data that has been standardized. So let's apply a MinMaxScaler as in Listing 19-3.

```
# apply a min max scaler
```

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df = pd.DataFrame(scaler.fit_transform(df), columns =
['T'])
```

Listing 19-3 Apply a MinMaxScaler

Now, a part that may be harder to get your head around intuitively. We need to split the data into a shape in which we have sequences of past data and sequences of future data. We want to predict 72 steps into the future, and we'll use 3*72 steps into the past. This is an arbitrary choice, and please feel free to try out using more or less past data. The code in Listing 19-4 loops through the data and creates sequences for the model training.

```
ylist = list(df['T'])

n_future = 72
n_past = 3*72
total_period = 4*72

idx_end = len(ylist)
idx_start = idx_end - total_period

X_new = []
y_new = []
while idx_start > 0:
  x_line = ylist[idx_start:idx_start+n_past]
  y_line =
ylist[idx_start+n_past:idx_start+total_period]

  X_new.append(x_line)
  y_new.append(y_line)

  idx_start = idx_start – 1

# converting list of lists to numpy array
import numpy as np
X_new = np.array(X_new)
y_new = np.array(y_new)
```

Listing 19-4 Preparing the sequence data

Now that we have obtained an X and a Y matrix for the model training, as always, we need to split it into a train and test set in order to be able to do a fair model evaluation. This is done in Listing 19-5.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test =
train_test_split(X_new, y_new, test_size=0.33,
random_state=42)
```

Listing 19-5 Splitting train and test

A final step that is necessary for fitting the SimpleRNN. This is a step that may be hard to understand intuitively, as there is not really a reason for it. The SimpleRNN layer needs an input format that is 3D, and the shape has to correspond to (n_samples, n_timesteps, n_features). This can be obtained using reshape. This reshape is, unfortunately, necessary sometimes when using Keras, as it is very specific as to the exact shape of the input data.

When working with layer types that you don't know yet, this can sometimes give real headaches. But it is crucial to learn how to work with it. Listing 19-6 shows you how to reshape the data into the right format for Keras to recognize it.

```
batch_size = 32

n_samples = X_train.shape[0]
n_timesteps = X_train.shape[1]
n_steps = y_train.shape[1]
n_features = 1

X_train_rs = X_train.reshape(n_samples, n_timesteps,
n_features )

X_test_rs = X_test.reshape(X_test.shape[0],
n_timesteps, n_features )
```

Listing 19-6 Reshape the data to be recognized by Keras

A Simple SimpleRNN

Now, let's start with a very small neural network using the SimpleRNN layer. You can parametrize it with Listing 19-7.

```
import random
random.seed(42)

import mlflow
mlflow.autolog()

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN

simple_model = Sequential([
  SimpleRNN(8, activation='tanh',input_shape=
(n_timesteps, n_features)),
  Dense(y_train.shape[1]),
])

simple_model.summary()

simple_model.compile(
  optimizer=keras.optimizers.Adam(learning_rate=0.001),
  loss='mean_absolute_error',
  metrics=['mean_absolute_error'],
)

smod_history = simple_model.fit(X_train_rs, y_train,
          validation_split=0.2,
          epochs=5,
          batch_size=batch_size,
          shuffle = True
)

preds = simple_model.predict(X_test_rs)

from sklearn.metrics import r2_score
print(r2_score(preds, y_test))

import matplotlib.pyplot as plt
plt.plot(smod_history.history['loss'])
plt.plot(smod_history.history['val_loss'])
plt.title('model loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
```

```
plt.show()
```

Listing 19-7 Parametrize a small network with SimpleRNN

The obtained plot is shown in Figure 19-3 and the obtained R2 score is **0.716**. This is not a great score, and the training history confirms that the model is not correctly specified: it is going down, but it seems to be not learning enough. The loss is what we call the accuracy throughout the model, as the error functions of the neural network is often referred to as the loss function. A more complex model should allow to let the loss go down more.

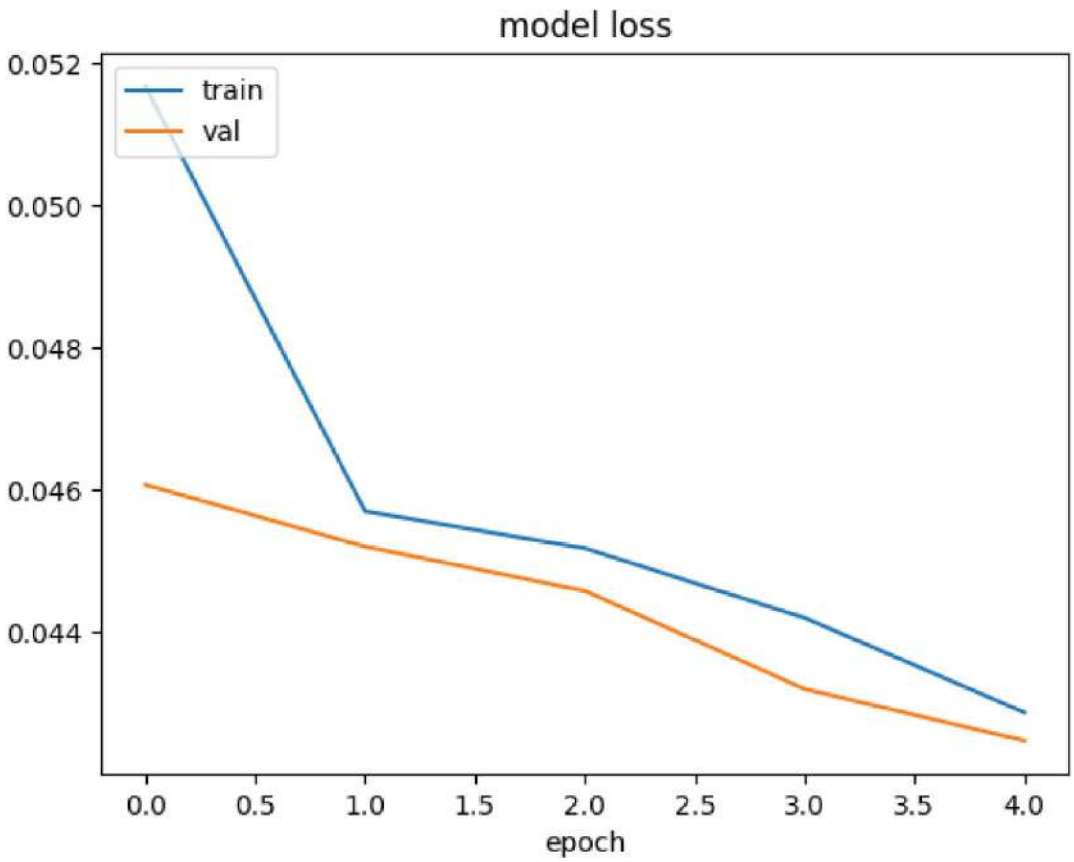


Figure 19-3 Training history of the SimpleRNN

SimpleRNN with Hidden Layers

As an improvement to this, let's add a second layer to this network. To do this, you need to specify in every layer but the last "**return_sequences = True**". Fitting RNNs is even slower than fitting **Feedforward Neural Networks**, so be aware that the computation times in this chapter may be long.

You could build the network in Listing 19-8 that is slightly more complex and see how it performs.

```
random.seed(42)

simple_model = Sequential([
  SimpleRNN(32, activation='tanh',input_shape=
(n_timesteps, n_features), return_sequences=True),
  SimpleRNN(32, activation='tanh', return_sequences =
True),
  SimpleRNN(32, activation='tanh'),
  Dense(y_train.shape[1]),
])

simple_model.summary()

simple_model.compile(
  optimizer=keras.optimizers.Adam(learning_rate=0.001),
  loss='mean_absolute_error',
  metrics=['mean_absolute_error'],
)

smod_history = simple_model.fit(X_train_rs, y_train,
          validation_split=0.2,
          epochs=5,
          batch_size=batch_size,
          shuffle = True
)

preds = simple_model.predict(X_test_rs)

print(r2_score(preds, y_test))

plt.plot(smod_history.history['loss'])
plt.plot(smod_history.history['val_loss'])
plt.title('model loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

Listing 19-8 A more complex network with three layers of SimpleRNN

This gives a multivariate R2 score of **0.895**. You will observe the history plot shown in Figure 19-4.

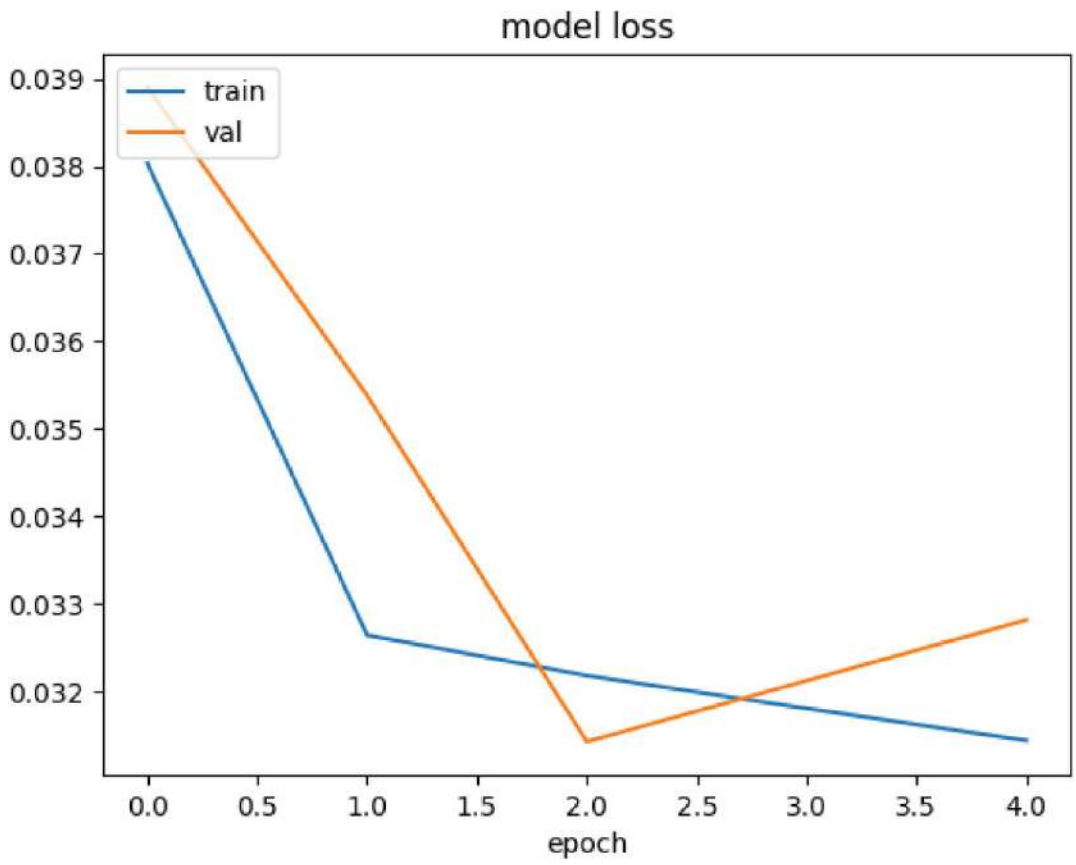


Figure 19-4 Training history of the three-layer SimpleRNN

This network allows the loss to go down earlier and lower, even though the curve is still not looking very impressive. At the same time, the obtained R2 is quite good, so it feels like things are moving in the right direction.

Simple GRU

Now we could go further into the SimpleRNN and try to optimize it. Yet, we won't do that here. The SimpleRNN is not up to today's standards anymore. It has some serious problems. One big effect of this is that it is not able to remember very long time back. There are also some more technical issues with it.

A more advanced RNN layer has been invented called GRU, for Gated Recurrent Unit. The GRU cell has more parameters, as shown in Figure 19-5. This shows that there is an extra passage inside the cell, and this allows for an extra parameter to be estimated. This helps with learning long-term trends.

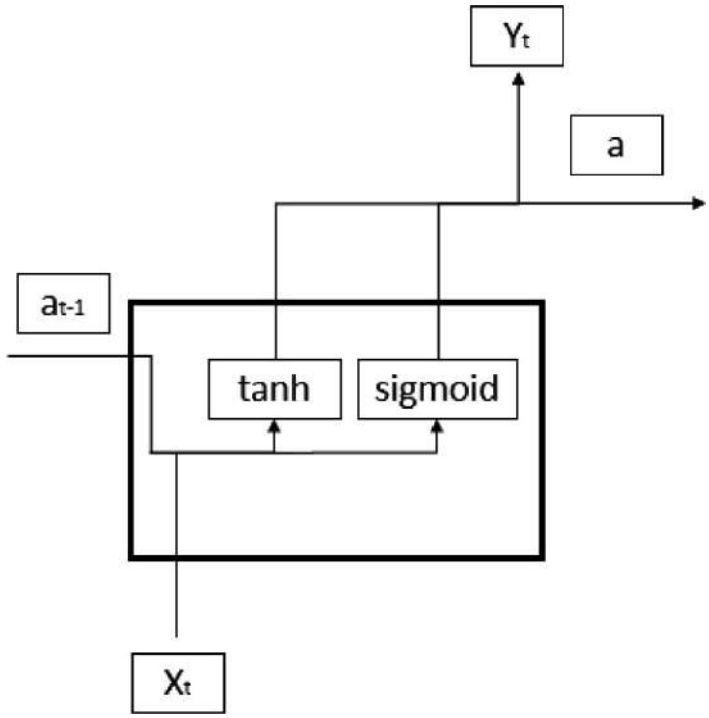


Figure 19-5 The GRU cell

Due to those differences, the GRU obtains better general performances than the SimpleRNN, and the SimpleRNN has become very little used. Let's build a simple network with GRU and see how it performs on the data. This is done in Listing 19-9.

```
random.seed(42)

simple_model = Sequential([
   GRU(8, activation='tanh',input_shape=(n_timesteps,
n_features)),
  Dense(y_train.shape[1]),
])

simple_model.summary()

simple_model.compile(
  optimizer=keras.optimizers.Adam(learning_rate=0.01),
  loss='mean_absolute_error',
  metrics=['mean_absolute_error'],
)

smod_history = simple_model.fit(X_train_rs, y_train,
```

```
          validation_split=0.2,
          epochs=10,
          batch_size=batch_size,
          shuffle = True
)

preds = simple_model.predict(X_test_rs)

print(r2_score(preds, y_test))

plt.plot(smod_history.history['loss'])
plt.plot(smod_history.history['val_loss'])
plt.title('model loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

Listing 19-9 A simple architecture with one GRU layer

You will obtain an R2 of **0.928**, which is not too bad for a one-layer model, and you will obtain the history plot that is shown in Figure 19-6. The history plot on the other hand shows that the fitting really did not go very well. The loss is not going down as it is supposed to be, and some changes should be made to the model architecture.

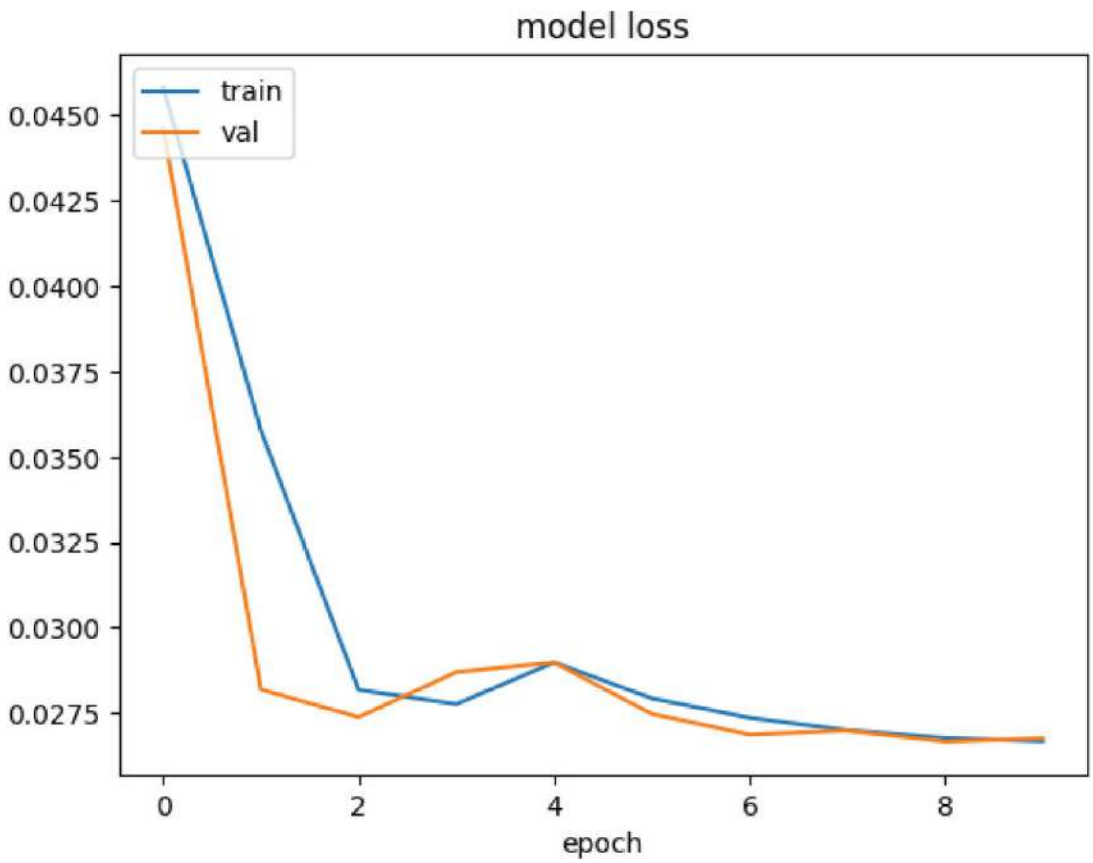


Figure 19-6 The history of the one-layer GRU model

GRU with Hidden Layers

Now, as a next step, let's try to improve on this network by adding some more layers and see if we can get the loss to go down more. To be totally transparent here, it takes a lot of time to work on the optimization of the architecture and hyperparameters. It is a work of trial and error while sticking to a number of model metrics, including loss and R2 score, but also the loss graph.

It also takes experience to test out different hyperparameters and to develop a feel for the type of changes that you'd need to make. But that's totally normal; there simply is a learning curve in those models. It is also important to realize that it takes a lot of time for one model to fit. This makes it hard to run grid search or other hyperparameter optimizations, yet at the same time, it can make it frustrating to wait a long time only to see a bad score arrive.

The model that you see in Listing 19-10 obtains an R2 score of **0.938**, so better than the simple GRU model.

```
random.seed(42)
```

```
simple_model = Sequential([
   GRU(64, activation='tanh',input_shape=(n_timesteps,
n_features), return_sequences=True),
    GRU(64, activation='tanh', return_sequences=True),
    GRU(64, activation='tanh'),
  Dense(y_train.shape[1]),
])

simple_model.summary()

simple_model.compile(
  optimizer=keras.optimizers.Adam(learning_rate=0.001),
  loss='mean_absolute_error',
  metrics=['mean_absolute_error'],
)

smod_history = simple_model.fit(X_train_rs, y_train,
          validation_split=0.2,
          epochs=10,
          batch_size=batch_size,
          shuffle = True
)

preds = simple_model.predict(X_test_rs)

print(r2_score(preds, y_test))

plt.plot(smod_history.history['loss'])
plt.plot(smod_history.history['val_loss'])
plt.title('model loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

Listing 19-10 A more complex network with three layers of GRU

The history graph of this model is shown in Figure 19-7. It is probably not the best model, but at least we see that the training and validation losses are both going down. The obtained R2 is not too bad. I leave it as an exercise for you to try and improve on this architecture using the loss graph and model metrics.

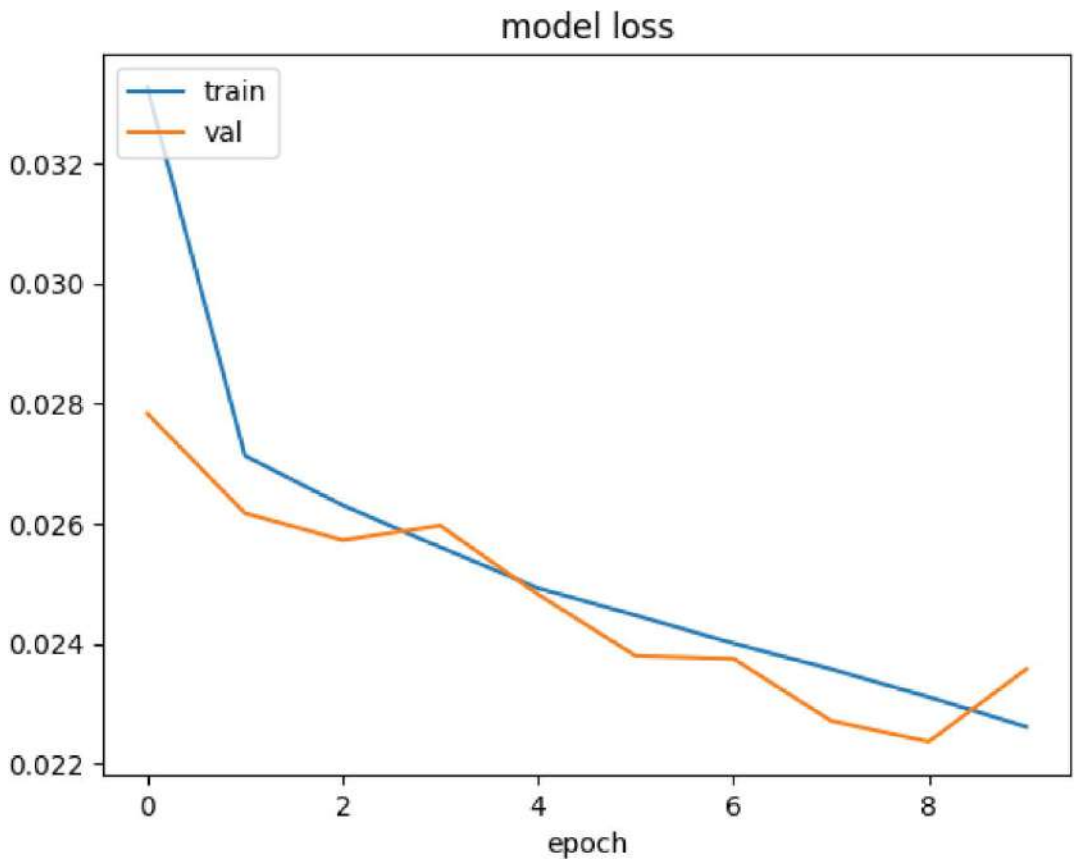


Figure 19-7 The history of the three-layer GRU model

Key Takeaways

- Recurrent Neural Networks have an integrated feedback loop that makes them great for modeling sequences.
- There are multiple types of RNN cells:
  - The SimpleRNN is a basic RNN cell that is not much used anymore, as it has numerous theoretical problems and is not able to learn long-term trends.
  - The GRU cell is an improvement of the RNN cell, and it has additional parameters that make it easier to remember long-term trends.
  - The LSTM will be covered in Chapter 20.
- RNNs, especially when training on long sequences, benefit from using a tanh activation function rather than the ReLU that is common in dense networks.
- You can use RNNs to predict a sequence of multiple steps to make a multistep forecast.
- You have seen a number of practical modeling examples, and this will help you to understand which history plots are good or not.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_20

20. LSTM RNNs

Joos Korstanje[1]
(1) Maisons Alfort, France

In the previous chapter, you have discovered two types of Recurrent Neural Network cells called SimpleRNN and GRU (Gated Recurrent Unit). In this chapter, you'll discover a third type of cell called **LSTM, for Long Short-Term Memory**.

What Is LSTM

LSTMs, as a third and last type of RNN cell, are even more advanced than the GRU cell. If you remember from the last chapter, the SimpleRNN cell allows having recurrent architectures by adding a feedback loop between consecutive values. It, therefore, is an improvement on "simple" feedforward cells that do not allow for this.

A problem with the SimpleRNN is the longer-term trends, which you could call a longer-term memory. The GRU cell is an improvement on the SimpleRNN that adds a weight to the cell that serves to learn longer-term processes.

The LSTM cell adds long-term memory in an even more performant way because it allows even more parameters to be learned. This makes it the most powerful RNN to do forecasting, especially when you have a longer-term trend in your data. LSTMs are one of the state-of-the-art models for forecasting at this moment.

The LSTM Cell

Let's see how the LSTM cell differs from the GRU cell. It has many more components. Notably, there are *three sigmoids and two tanh operations* all combined into the same cell. You can see a schematic overview in Figure 20-1, where there are two weights coming in from the past cell (c and a of the time t-1)

and, once transformed, another two weights are going (c and a of the time t). The X is the input to this cell, and the Y is the output.

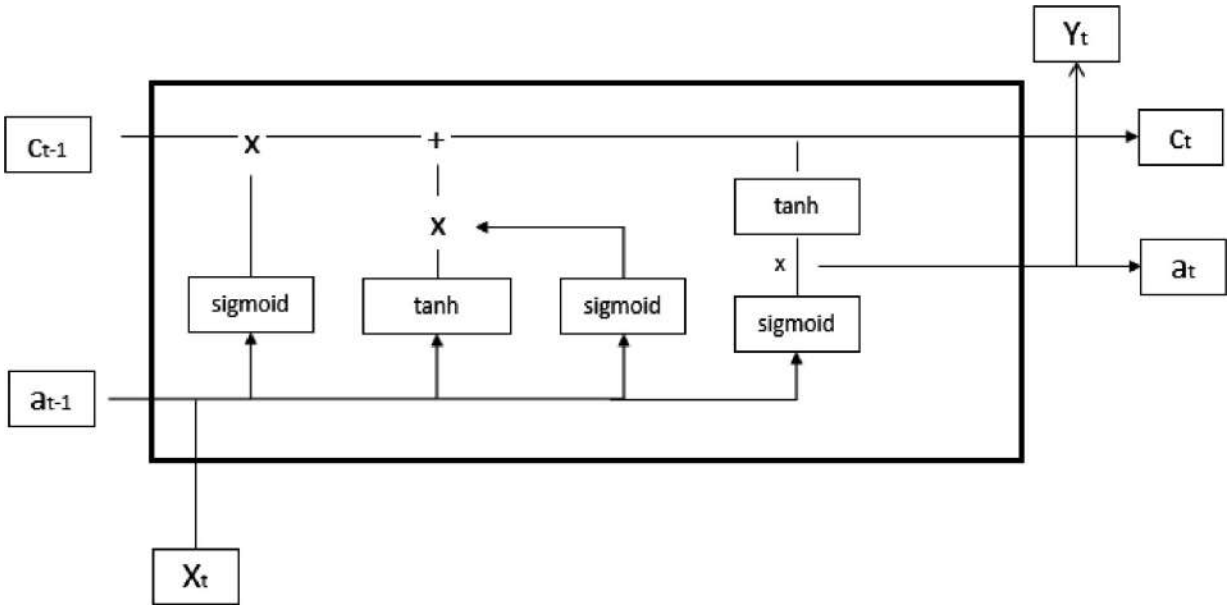


Figure 20-1 The LSTM cell

In practice, this all comes down to having more parameters to estimate, and the LSTM can therefore better fit on data that have short- and long-term trends.

Example

In applied machine learning, it is important to benchmark models against each other. Let's therefore use the same data for the last time and make it into a benchmark of the three RNN models that we have seen. As a short recap, the performance that was obtained by the models in Chapter 19 is repeated in Table 20-1.

Table 20-1 Performance of the SimpleRNN and GRU models

Model	SimpleRNN	GRU
1 layer of 8	0.716	0.928
3 layers of 64	0.895	0.95

Let's see how the LSTM compares to this. You can use the same data preparation as used in Chapter 19, which is repeated in Listing 20-1.

```
import keras
import pandas as pd

from zipfile import ZipFile
import os

uri = "https://storage.googleapis.com/tensorflow/tf-
keras-datasets/jena_climate_2009_2016.csv.zip"
zip_path = keras.utils.get_file(origin=uri,
fname="jena_climate_2009_2016.csv.zip")
zip_file = ZipFile(zip_path)
zip_file.extractall()
csv_path = "jena_climate_2009_2016.csv"

df = pd.read_csv(csv_path)
del zip_file

# retain only temperature
df = df[['T (degC)']]

# apply a min max scaler
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df = pd.DataFrame(scaler.fit_transform(df), columns =
['T'])

# convert to windowed data sets
ylist = list(df['T'])

n_future = 72
n_past = 3*72
total_period = 4*72

idx_end = len(ylist)
idx_start = idx_end - total_period

X_new = []
y_new = []
while idx_start > 0:
  x_line = ylist[idx_start:idx_start+n_past]
```

```
  y_line =
ylist[idx_start+n_past:idx_start+total_period]

  X_new.append(x_line)
  y_new.append(y_line)

  idx_start = idx_start - 1

import numpy as np
X_new = np.array(X_new)
y_new = np.array(y_new)

# train test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X_new, y_new, test_size=0.33,
random_state=42)

# reshape data into the right format for RNNs
n_samples = X_train.shape[0]
n_timesteps = X_train.shape[1]
n_steps = y_train.shape[1]
n_features = 1

X_train_rs = X_train.reshape(n_samples, n_timesteps,
n_features )
X_test_rs = X_test.reshape(X_test.shape[0],
n_timesteps, n_features )
```

Listing 20-1 Importing the weather data

LSTM with 1 Layer of 8

Now, as we did in the previous chapter, let's start with fitting a simple one-layer network. The code is almost the same as Chapter 19's code; you just need to change the name of the layer to LSTM. The total code is shown in Listing 20-2.

```
import random
random.seed(42)

from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense, LSTM
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt

import mlflow
mlflow.autolog()

batch_size = 32
simple_model = Sequential([
   LSTM(8, activation='tanh',input_shape=(n_timesteps,
n_features)),
  Dense(y_train.shape[1]),
])

simple_model.summary()

simple_model.compile(
  optimizer=keras.optimizers.Adam(learning_rate=0.01),
  loss='mean_absolute_error',
  metrics=['mean_absolute_error'],
)

smod_history = simple_model.fit(X_train_rs, y_train,
          validation_split=0.2,
          epochs=5,
          batch_size=batch_size,
          shuffle = True
)

preds = simple_model.predict(X_test_rs)

print(r2_score(preds, y_test))

plt.plot(smod_history.history['loss'])
plt.plot(smod_history.history['val_loss'])
plt.title('model loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

Listing 20-2 One-layer LSTM

The results of this model are the following. The R2 score obtained by this model is **0.927**. The plot of the training loss is shown in Figure 20-2.

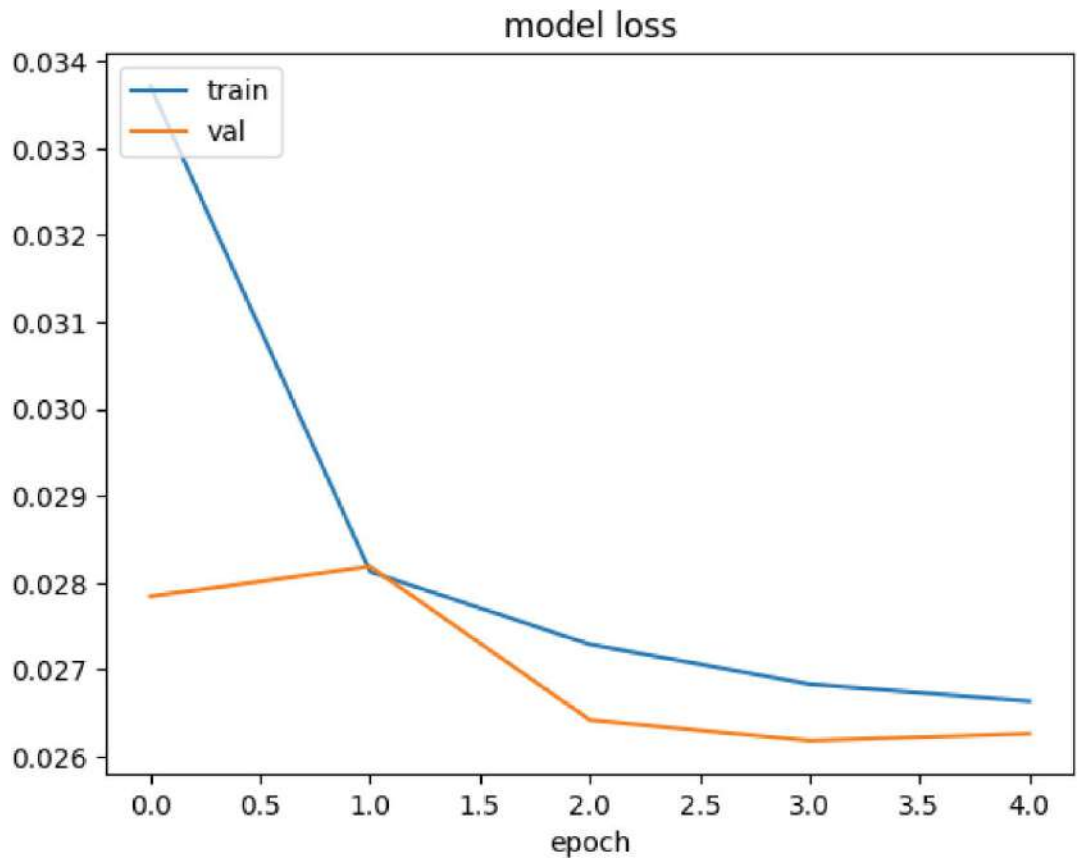


Figure 20-2 Training history of the one-layer LSTM

In this graph, you can see that the train loss and val loss both go down a little bit. Yet, the drop in validation loss is not very steep. This leads to the conclusion that the model is learning a bit, but that we are probably able to get more out of this model. We'd want to see a bigger drop in validation loss before it plateaus. In the current graph, you're almost directly on a plateau. In Table 20-2, you see the table updated with the new value.

Table 20-2 Performance of the SimpleRNN and GRU models

Model	SimpleRNN	GRU	LSTM
1 layer of 8	0.716	0.928	0.927
3 layers of 64	0.895	0.938	

LSTM with 3 Layers of 64

As you remember from the previous chapter, and as you can see in the table, the GRU model with 3 layers of 64 cells worked quite well. Before going deeper into the tuning of the LSTM model, let's also try out the performance of this architecture. You can use Listing 20-3 to do this.

```
random.seed(42)

simple_model = Sequential([
   LSTM(64, activation='tanh',input_shape=(n_timesteps,
n_features), return_sequences=True),
    LSTM(64, activation='tanh', return_sequences=True),
    LSTM(64, activation='tanh'),
  Dense(y_train.shape[1]),
])

simple_model.summary()

simple_model.compile(
  optimizer=keras.optimizers.Adam(learning_rate=0.001),
  loss='mean_absolute_error',
  metrics=['mean_absolute_error'],
)

smod_history = simple_model.fit(X_train_rs, y_train,
          validation_split=0.2,
          epochs=10,
          batch_size=batch_size,
          shuffle = True
)

preds = simple_model.predict(X_test_rs)

print(r2_score(preds, y_test))

plt.plot(smod_history.history['loss'])
plt.plot(smod_history.history['val_loss'])
plt.title('model loss')
plt.ylabel('accuracy')
plt.xlabel('epoch')
```

```
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```

Listing 20-3 Three-layer LSTM

This model obtains an R2 score of **0.941**. When comparing this to the other values in Table 20-3, you can observe that it functions a bit less than the GRU three-layer model.

Table 20-3 Performance of the SimpleRNN and GRU models

Model	SimpleRNN	GRU	LSTM
1 Layer of 8	0.716	0.928	0.927
3 Layers of 64	0.895	0.938	**0.941**

Let's also have a look at the history graph shown in Figure 20-3. What we can see in this training graph is that the loss drops more than in the architecture with one layer. The shape of the train loss is looking not too bad: it has the right shape, although it seems that the drop could go on a bit longer, and therefore, the plateau would be reached later and at a lower loss. The validation, on the other hand, does not really have the right shape, which tells us that there is probably some R2 to win by tweaking the model.

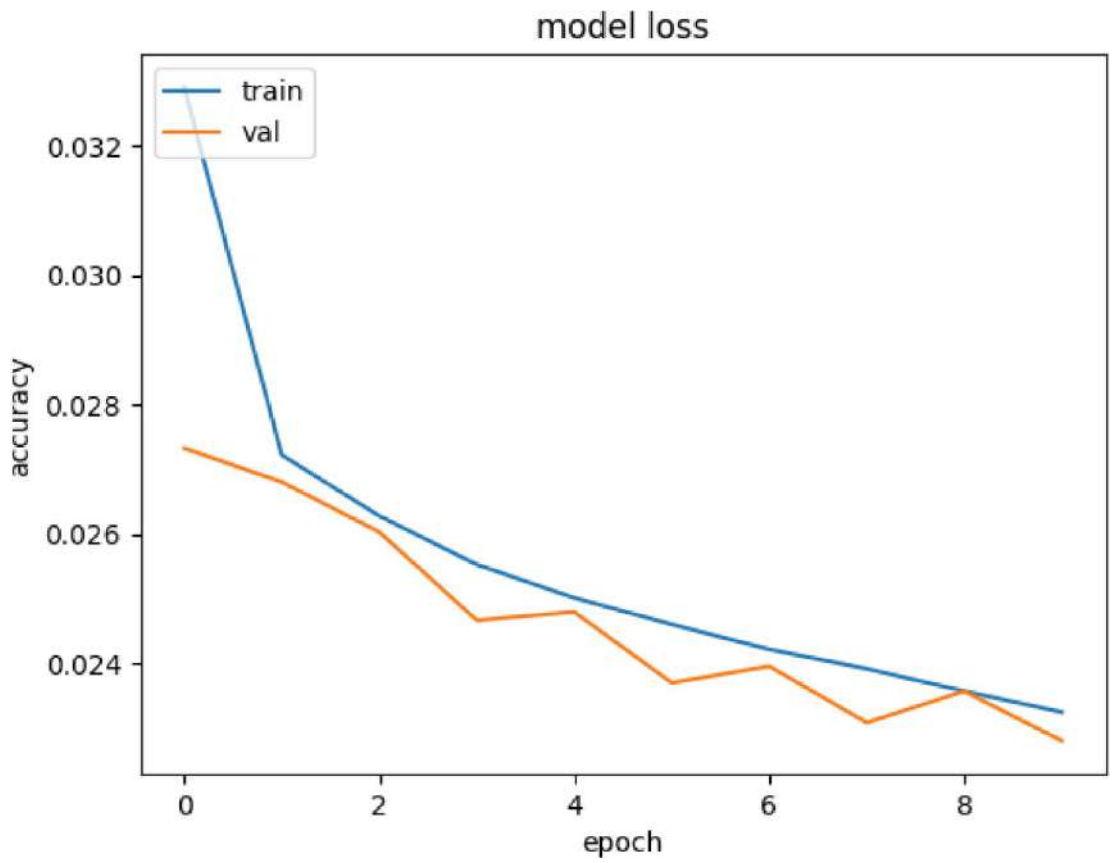


Figure 20-3 Training history of the three-layer LSTM

Conclusion

So, where does that leave us? The best RNN is the three-layer LSTM, very closely followed by the three-layer GRU. The differences in scores clearly show why the SimpleRNN layer is not much used anymore. The GRU and the LSTM are two RNN models that are performant for forecasting tasks.

The LSTM model has the advantage over GRU in being able to fit more long-term trends. A hypothetical reason for the better performance of the GRU is that the data did not contain very long-term trends, as the data were reworked to contain only 36 hours of past data for each line of data.

When doing modeling in practice, it is very common to run benchmarks like the one done in this example. The important thing is to master the different models to be able to tune and benchmark them effectively. After that, the decision should be objective based on performance during the benchmark. In this case, it was the LSTM that won the benchmark.

Key Takeaways

- The LSTM is the most advanced type of RNN as it has a large number of components that allow it to remember longer-term trends.
- In the example, you have seen an example of how to do a model benchmark between multiple models on the same dataset. This is crucial in applied machine learning.
- You have also seen an example of how to interpret and work with neural network history graphs that show the descent in loss. You can use them to draw conclusions on the way your model is learning and how to improve that.

Part VI
Black Box and Cloud-Based Models

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_21

21. The N-BEATS Model with Darts

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, we will dive into the N-BEATS model, short for Neural Basis Expansion Analysis for Time Series. This model is designed specifically for time series forecasting.

The N-BEATS model is rooted in (deep) Neural Network architectures. You have already seen some examples of this in the previous chapters. However, the approach that you have seen in previous chapters was all based on finding the right architecture for your Neural Network.

N-BEATS works differently: it takes a number of hyperparameters, but a large part of it is already predefined. Predefined settings tend to make models more black box: they can be easier to use, but also less transparent. N-BEATS tries to be an exception to this by making the results as interpretable as possible.

Throughout this chapter, you'll discover the intuitive and theoretical explanations of N-BEATS, and you will dive into an applied example of this powerful and easy-to-use forecasting model.

Intuition and Mathematics of N-BEATS

Let's get an understanding of how the N-BEATS model works deep down. The main building block of N-BEATS is Feed Forward Neural Networks. You have seen these in an earlier chapter, and surprisingly, this is the "simplest" approach to Neural Networks that you have seen.

Interpretability

So why would an advanced modeling package like N-BEATS be basing itself on these Feed Forward Neural Networks rather than on LSTMs or other RNNs? The answer is Interpretability.

As you have seen in the chapter on Neural Networks, Feed Forward Neural Networks are actually part of the family of supervised machine learning: they take a set of X variables (which would include time data in the case of forecasting) and model

them into a target variable. You have also seen that RNNs have a more complex, recursive node layer and that they generally do not have the notion of X variables.

The N-BEATS package, by using Feed Forward Neural Networks, is able to show you a variable importance like interpretation of the model, which will tell you why certain predictions have been made. This would have been much harder to obtain using RNNs.

Stacking

A second element that is very important in N-BEATS is stacking. Rather than using just one Feed Forward Neural Network, N-BEATS will use an ensemble technique called stacking to combine a larger number of individual Neural Networks into one big stacked ensemble model.

This approach is very widespread, for example, in forecasting competitions, where participants need to use as much “gunpowder” as possible on the performance metric of a predefined test dataset and will build complex stacked models that certainly do improve performance, but also go further away from the traditional methods of statistical modeling. The methods tend to be hard to explain to businesses, which may or may not be a problem depending on your needs.

The stacked blocks do not each fit a model on the target variable. Rather, each of the blocks takes the residuals from the previous block (the prediction errors) and builds a new Neural Network that has a goal to predict those residuals. This continues until stopped.

Automatic Feature Engineering and Decomposition

A final important element of N-BEATS is that it automatically performs time series decomposition. You have seen time series decomposition extensively in earlier chapters of this book: the goal is to extract a general linear trend, as well as different forms of seasonality from a time series. Although there are good methods available to do this manually, it is practical that N-BEATS proposes it directly.

The Darts Package and Its Implementation of N-BEATS

The N-BEATS model is an algorithm that has been implemented by different Python packages. One good example of an N-BEATS implementation is the Nixtla package. We’ll cover the Nixtla package in much detail in a later chapter.

The Darts package also has a good implementation of N-BEATS. The Darts package is a very interesting package to discover for professionals working in time series and forecasting. It is a Python package focused on time series. As such, it proposes a variety of models for forecasting.

The Darts package has alternative implementations for the classical univariate time series that you have studied in earlier chapters of this book. The examples in those chapters were strongly rooted in the statsmodels library.

Dart also proposes a large list of very modern models, including Catboost and LightGBM, that we have seen in earlier chapters as well, and much more. As you'll learn to work with the N-BEATS model in Darts, you'll not just be mastering the N-BEATS model but also the Darts interface. If you're motivated to go beyond the scope, you could try to replace the N-BEATS model in the code with other models that Darts proposes. For references, you can have a look at their forecasting model documentation: https://unit8co.github.io/darts/generated_api/darts.models.forecasting.html

---

Forecasting Sales Using N-BEATS in Darts

Let's now move on to an example implementation of the N-BEATS model using the Darts package. We'll be using a dataset (1) that collects sales of an Italian grocery store. You have seen this dataset before: in the chapter on pyBATS. The data contains daily sales amounts per brand per product and also contains information on promotions. After all, promotions may strongly impact sales.

(1) Source: A machine learning approach for forecasting hierarchical time series (Paolo Mancuso, Veronica Piccialli, and Antonio M. Sudoso)

Sales Forecast: Preparing the Data

Let's start by applying the same data preparation as that applied in the chapter on pyBATS. You may remember that we regrouped the data. In the input data, we have detailed sales per brand, per product, per day. In the prepared dataset, we have the total sales per month. The data preparation is shown in Listing 21-1.

```
import pandas as pd
import matplotlib.pyplot as plt

# Read the data
df = pd.read_csv('hierarchical_sales_data.csv')

df_sum = df[['DATE']]

# Sum the sales and promos of the different products
df_sum['SALES'] = df[[x for x in df.columns if
x.startswith('QTY')]].sum(axis=1)
df_sum['PROMO'] = df[[x for x in df.columns if
x.startswith('PROMO')]].sum(axis=1)

# Create sum of sales and promos per month
df_sum['YEAR_MONTH'] = df_sum['DATE'].apply(lambda x:
x[:7])
```

```
df_sum = df_sum[['SALES', 'PROMO',
'YEAR_MONTH']].groupby('YEAR_MONTH').sum()
df_sum.head()

# Preview the data
df_sum.head()
```

Listing 21-1 Importing the data

Once you've applied the data preparation from Listing 21-1, you'll obtain a dataset that looks like the one in Figure 21-1.

YEAR_MONTH	SALES	PROMO
2014-01	12819	990
2014-02	17906	1329
2014-03	12047	896
2014-04	15998	1235
2014-05	17453	1354

Figure 21-1 The prepared data

It is important to remember the logic of this dataset. It is not a univariate dataset, as we have an explanatory variable, PROMO. This PROMO variable indicates whether there were any promotions, which could obviously impact total sales in addition to trend and seasonality effects. Listing 21-2 shows you how to recreate one of the plots that we inspected earlier in the book, which shows in what way PROMO impacts SALES.

```
df_scaled = df_sum[['SALES']]
df_scaled['PROMO'] = df_sum['PROMO'] * 8

df_scaled.plot()
plt.xticks(rotation=45)
```

Listing 21-2 Remember the impact of promos on sales

The code in Listing 21-2 will generate the plot shown in Figure 21-2. You clearly see that some of the peaks in PROMO coincide with peaks in SALES, indicating a potential "PROMO effect." This is important for model building, as it means that a

multivariate model with PROMO as an independent variable will probably be an absolute necessity, especially if you want your model to reflect a part of business logic.

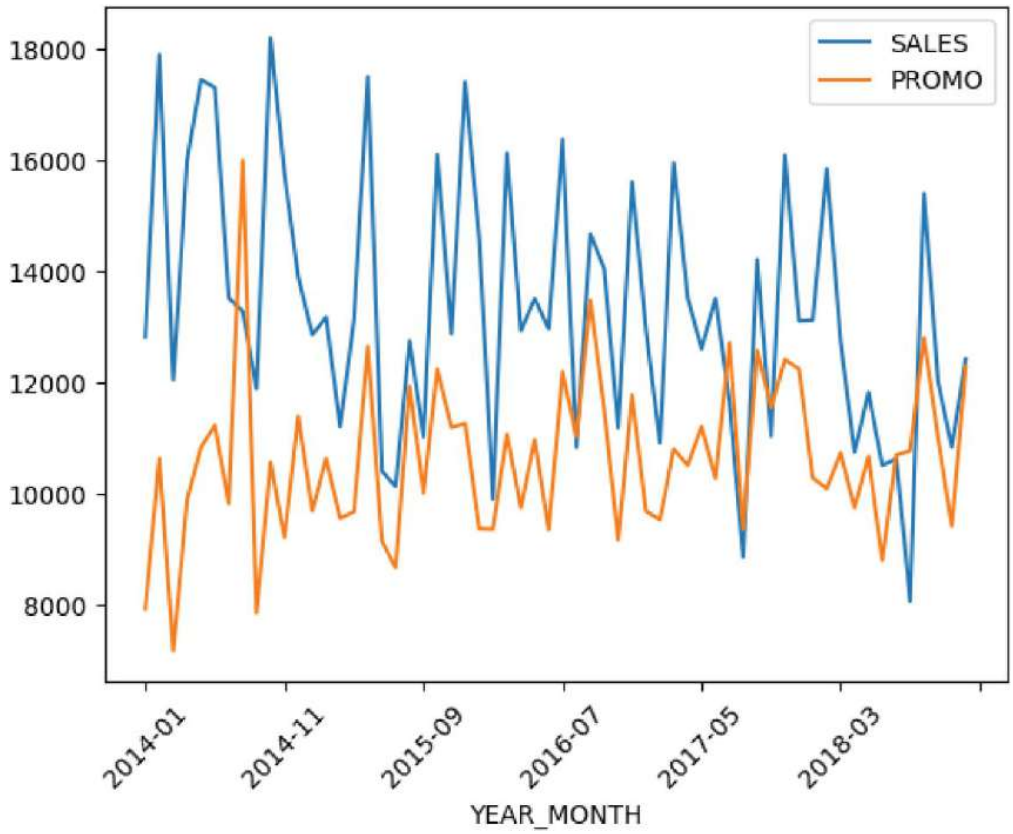


Figure 21-2 A plot of the data

As discussed in the theory part of this chapter, we'll be using the Darts forecasting package. The Darts package requires some specific data preparation. The most important element of this is to convert your data into a Darts time series. You can use Listing 21-3 to prepare this data preparation, as well as create a train-test split for the model-building phase.

```
import numpy as np
from darts.timeseries import TimeSeries

# Convert sales data to float32 for Darts
df_sum['SALES'] = df_sum['SALES'].map(np.float32)

# Convert the dataframe index to a datetime index as
required by Darts
df_sum.index = pd.DatetimeIndex(df_sum.index)

# Full year 2014 to 2017 is the training data
```

```
train = df_sum.iloc[:-12]

# Full year 2018 is the test data
test = df_sum.iloc[-12:]

# Convert the training data to a Darts time series
train_series = TimeSeries.from_dataframe(train[['SALES']])

# Inspect the time series format
train_series
```

Listing 21-3 Prepare for Darts and create univariate train-test split

You can see what this Darts time series format looks like in Figure 21-3. If you're executing the code through Jupyter Notebook, you'll see that the Darts time series will give you an interactive tool that allows you to inspect the different elements of the time series, like the data array, and other elements that you can set depending on the use case that you have for the Darts package. If you want to dive deeper into the time series format of the Darts package, I recommend checking out their extensive documentation:https://unit8co.github.io/darts/generated_api/darts.timeseries.html.

```
TimeSeries (DataArray)    (YEAR_MONTH: 48, component: 1, sample: 1)

array([[[12819.]],
       [[17906.]],
       [[12047.]],
       [[15998.]],
       [[17453.]],
       [[17310.]],
       [[13523.]],
       [[13279.]],
       [[11887.]],
       [[18202.]],
  ...
       [[15957.]],
       [[13524.]],
       [[12601.]],
       [[13516.]],
       [[11642.]],
       [[ 8858.]],
       [[14215.]],
       [[11029.]],
       [[16095.]],
       [[13111.]]], dtype=float32)
▼ Coordinates:
  YEAR_MONTH   (YEAR_MONTH)  datetime64[ns]  2014-01-01 ... 2017-12-01
  component    (component)          <U5  'SALES'
► Indexes:  (2)
▼ Attributes:
  static_covariate...  None
  hierarchy :          None
  metadata :           None
```

Figure 21-3 The Darts time series format

Sales Forecast: Create Default N-BEATS Model

Now that we have prepared the data in accordance with the Darts package, let's start building a first N-BEATS model. As a first step, let's do the easiest version possible:

we'll do a univariate model (so excluding the PROMO variable for now), and we'll use as many default hyperparameters as possible. There are a few settings that we do need to set:

- input_chunk_length: The number of data points to use for each N-BEATS block. Let's (arbitrarily) give each model chunk two years of data, so 24 months.
- output_chunk_length: We want to create a 12-month forecast, so let's ask the model to produce 12 output steps.
- n_epochs: Let's request five epochs for now. This is low but will allow us to get a first benchmark model.

The code in Listing 21-4 shows you how to instantiate and train the N-BEATS model using these settings.

```
# Use MLFlow autologging for Darts
import mlflow
mlflow.autolog()

# import the NBEATSModel
from darts.models.forecasting.nbeats import NBEATSModel

# Specifying a default model
model = NBEATSModel(
    # with input chunk length 24 (number of time steps
that each NBEATS block uses as input)
    input_chunk_length=24,
    # with output chunk 12 (number of time steps that each
NBEATS block uses as output)
    output_chunk_length=12,
    # 5 epochs to have a quick first result
    n_epochs=5,
    random_state=123
)

# Train the model
model.fit(train_series)
```

Listing 21-4 Create a default model

Having now built and trained the model, let's use Listing 21-5 to make a one-year monthly prediction (12 steps forward) and compute the performance on the test data using the 1-MAPE KPI that was also used in Chapter 17 (with the same data).

```
from sklearn.metrics import mean_absolute_percentage_error
```

```
# Forecast 12 steps using this model
fcst = model.predict(12).values()

# Compute metric
metric = 1 -
mean_absolute_percentage_error(list(test['SALES']), fcst)
print(metric)

plt.plot(fcst)
plt.plot(list(test['SALES']))
plt.legend(['fcst', 'test'])
```

Listing 21-5 Build the forecast and analyze results

This will generate a performance KPI 1-MAPE of **0.81.** The code in Listing 21-5 will also generate the forecasting plot shown in Figure 21-4, showing a moderately good fit: overall variation is more or less captured, but the forecast is still lacking in detail.

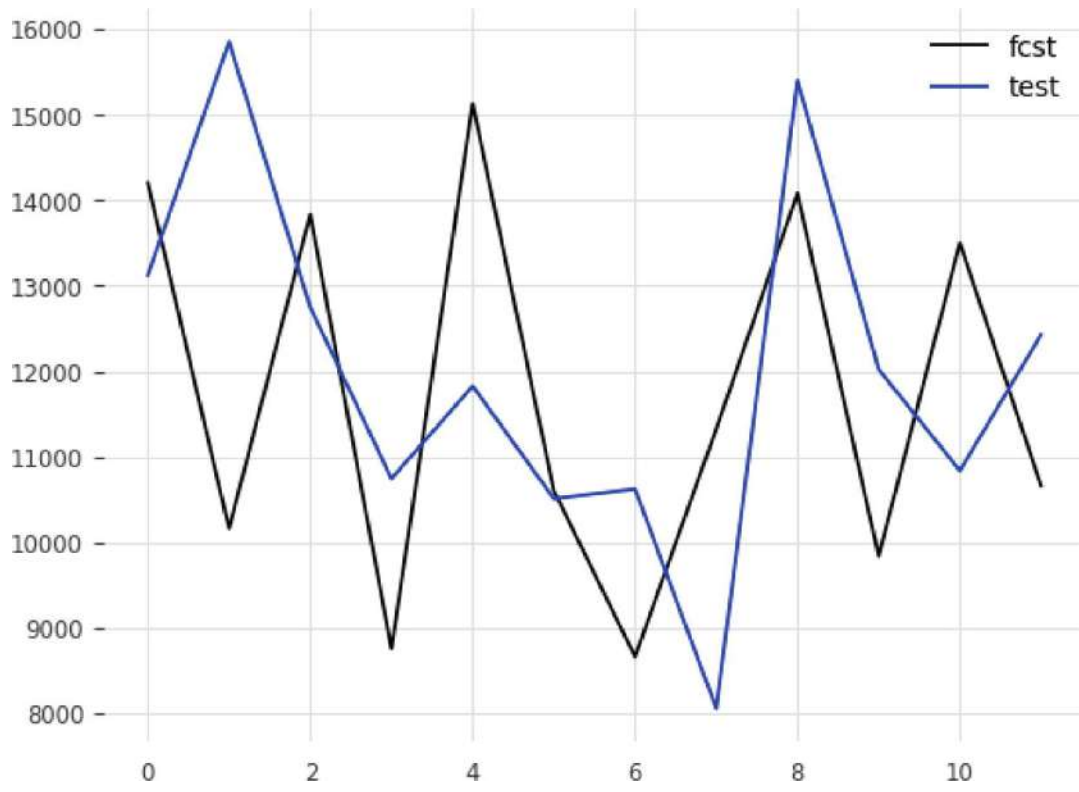


Figure 21-4 The forecasting plot of the simple model

Sales Forecast: Create Multivariate N-BEATS Model

As a next step, let's see if we can improve the model by adding the PROMO variable. There is an important constraint when using the N-BEATS model in Darts. Although

we can use an independent variable for past data points (training period), the model does not allow us to add the independent variable in the future. Although this may not be a problem for some datasets, it is likely to negatively impact performance in the current applied example.

You can use Listing 21-6 to convert the training data for promos into a Darts time series.

```
df_sum['PROMO'] = df_sum['PROMO'].map(np.float32)

# Convert the training data to a Darts time series
promos_train = TimeSeries.from_dataframe(train[['PROMO']])

# Convert to float32 for Darts
promos_train = promos_train.astype(np.float32)
```

Listing 21-6 Create multivariate data

Having prepared the data, you can use Listing 21-7 to build the model by specifying the same settings as in the simple model. However, as shown in the code, you need to add the training PROMO time series as **past_covariates** in the **fit** step.

```
# Specifying a default model
model = NBEATSModel(
    # with input chunk length 24 (number of time steps
that each NBEATS block uses as input)
    input_chunk_length=24,
    # with output chunk 12 (number of time steps that each
NBEATS block uses as output)
    output_chunk_length=12,
    # 5 epochs to have a quick first result
    n_epochs=5,
    random_state=123

)

# Train the model
model.fit(
    train_series,

    # add the promotions as past covariates
    past_covariates=promos_train
)

# Forecast 12 steps using this model
fcst = model.predict(
```

```
    n=12,

    # add past covariates (promos train)
    past_covariates=promos_train,

).values()

# Compute metric
metric = 1 -
mean_absolute_percentage_error(list(test['SALES']), fcst)
print(metric)

plt.plot(fcst)
plt.plot(list(test['SALES']))
plt.legend(['fcst', 'test'])
```

Listing 21-7 Create a multivariate model

The code in Listing 21-7 will generate a multivariate model that obtains a 1-MAPE performance score of **0.82**. It is only very slightly better than the simple model. This can be explained intuitively by the fact that future PROMOS are not taken into account, and otherwise, the settings are the same as the previous model. The code will also generate the forecasting plot as shown in Figure 21-5.

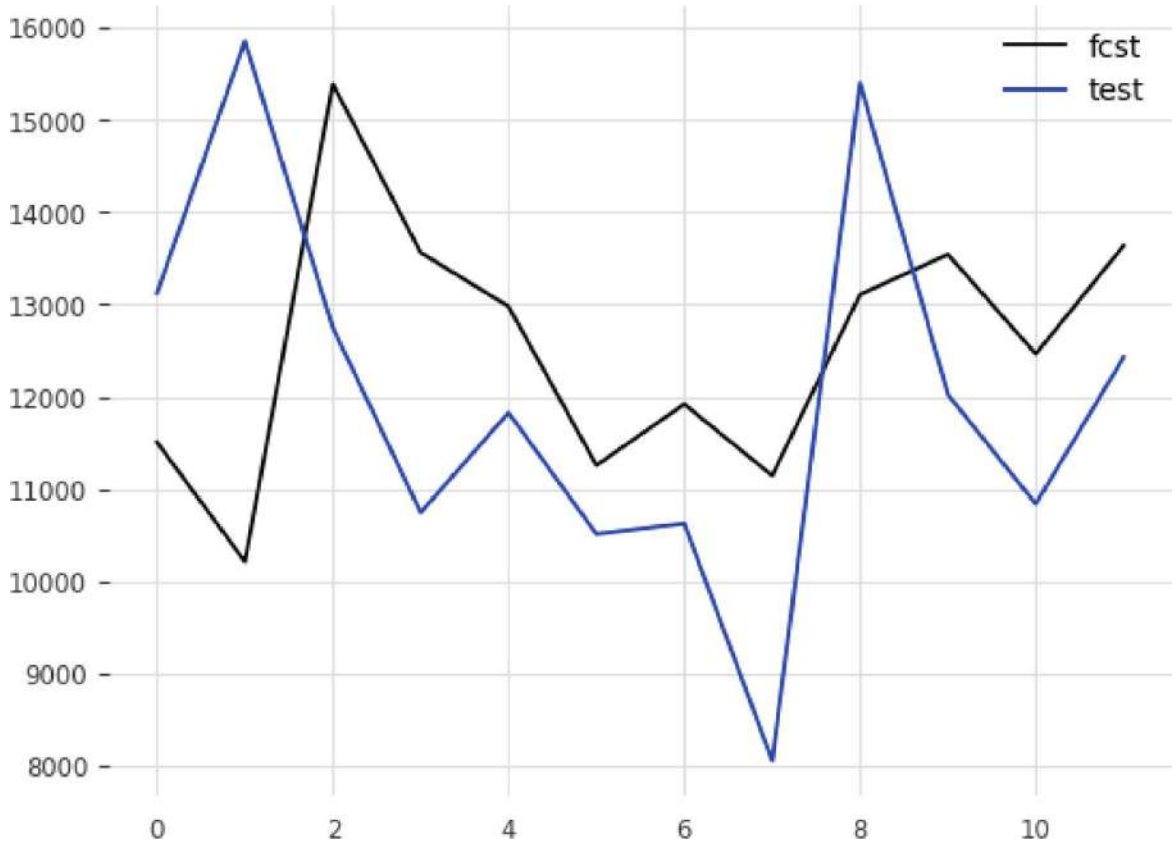


Figure 21-5 The forecasting plot of the multivariate model

Sales Forecast: Tuning the Multivariate N-BEATS Model

Although the multivariate model has performed slightly better than the simple model, there is still a lot of optimization possible. Indeed, N-BEATS has a lot of hyperparameters that can be tuned. You can get a list of the hyperparameters on the Darts documentation:

https://unit8co.github.io/darts/generated_api/darts.models.forecasting.nbeats.html#darts.models.forecasting.nbeats.NBEATSModel

In Listing 21-8, we'll look at an example that tunes the following hyperparameters:

- input_chunk_length
- n_epochs
- num_stacks
- expansion_coefficient_dim

This is not meant to be an exhaustive tuning, so feel free to pick some hyperparameters from the documentation and add them into the code and see whether that can improve the performance: after all, an important part of model tuning remains trial-and-error.

```
def score_nbeats(input_chunck_length, n_epochs, num_stacks,
expansion_coefficient_dim):

    # Specifying a model using the gridsearch parameters
    model = NBEATSModel(
        input_chunk_length=input_chunck_length,
        output_chunk_length=12,
        n_epochs=n_epochs,
        num_stacks=num_stacks,
        random_state=123,
        expansion_coefficient_dim=expansion_coefficient_dim
    )

    # Train the model
    model.fit(
        train_series,

        # add the promotions as past covariates
        past_covariates=promos_train
    )

    # Forecast 12 steps using this model
    fcst = model.predict(
        n=12,

        # add past covariates (promos train)
        past_covariates=promos_train,

    ).values()

    # Compute metric
    metric = 1 -
mean_absolute_percentage_error(list(test['SALES']), fcst)

    return metric

# Run a Grid Search
grid_result = []
for input_chunck_length in [6, 12, 18, 24]:
    for n_epochs in [5, 10, 20]:
        for num_stacks in [15, 30, 45]:
            for expansion_coefficient_dim in [3, 5, 7]:
                r2_test = score_nbeats(input_chunck_length,
n_epochs, num_stacks, expansion_coefficient_dim)
```

```
                result = [input_chunck_length, n_epochs,
num_stacks, expansion_coefficient_dim, r2_test]
                grid_result.append(result)

# Inspect the results in a DataFrame
grid_output = pd.DataFrame(grid_result, columns=[
    'input_chunck_length',
    'n_epochs',
    'num_stacks',
    'expansion_coefficient_dim',
    'r2_test'
])

grid_output.sort_values('r2_test', ascending=False).head()
```

Listing 21-8 Tuning the multivariate model

The result of this grid search is the table shown in Figure 21-6.

	input_chunck_length	n_epochs	num_stacks	expansion_coefficient_dim	r2_test
73	18	20	15	5	0.881027
74	18	20	15	7	0.871356
22	6	20	30	5	0.867765
9	6	10	15	3	0.864698
14	6	10	30	7	0.863693

Figure 21-6 The GridSearch results

From Figure 21-6, we can conclude that the best tuned multivariate model from our experiment is the model with the following hyperparameters:

- input_chunk_length=18
- n_epochs=20
- num_stacks=15
- expansion_coefficient_dim=5

This model obtains a test 1-MAPE score of **0.88**, which is a serious improvement over the previous score. Let's rebuild the best model using Listing 21-9 and use it to create a forecasting plot.

```
input_chunck_length = 18
n_epochs = 20
num_stacks=15
expansion_coefficient_dim=5
```

```
# Specifying a model using the gridsearch parameters
model = NBEATSModel(
    input_chunk_length=input_chunck_length,
    output_chunk_length=12,
    n_epochs=n_epochs,
    num_stacks=num_stacks,
    expansion_coefficient_dim=expansion_coefficient_dim,
    random_state=123

)

# Train the model
model.fit(
    train_series,

    # add the promotions as past covariates
    past_covariates=promos_train
)

# Forecast 12 steps using this model
fcst = model.predict(
    n=12,

    # add past covariates (promos train)
    past_covariates=promos_train,

).values()

# Compute metric
metric = 1 -
mean_absolute_percentage_error(list(test['SALES']), fcst)
print(metric)

# Plot the best forecast
plt.plot(fcst)
plt.plot(list(test['SALES']))
plt.legend(['fcst', 'test'])
```

Listing 21-9 Building the tuned multivariate model

This code will confirm the 1-MAPE score of **0.88** and also generate the forecasting plot shown in Figure 21-7. You can see that the forecast has become less fluctuating, but still has two peaks close to the peaks in sales. The low point in the test data has not been predicted well, as this was due to a low point of PROMO, of which future values cannot be incorporated: a serious drawback of the N-BEATS model.

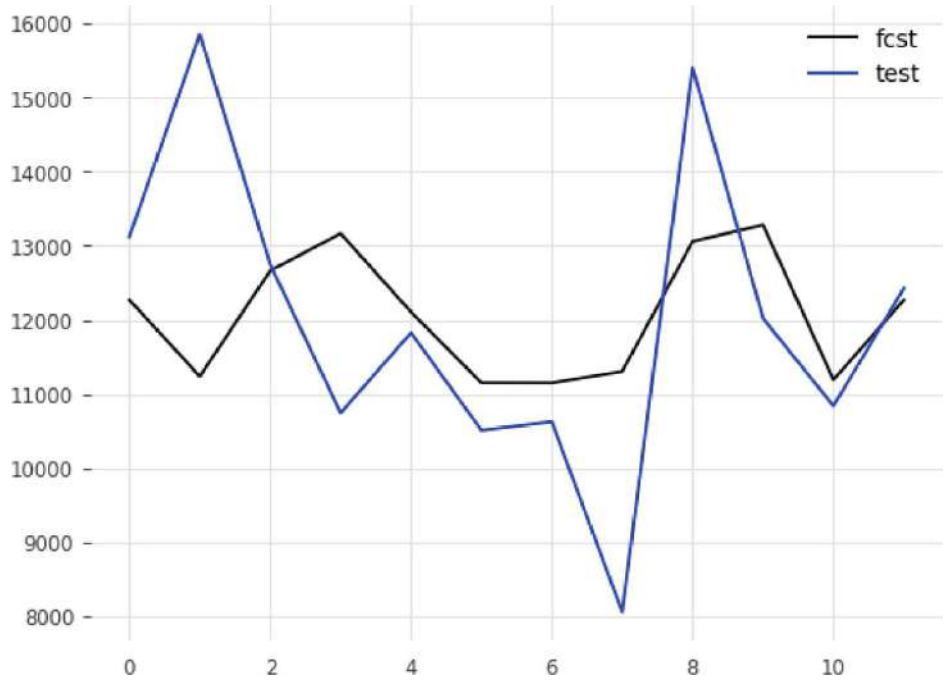


Figure 21-7 The forecast of the tuned model

Benchmark Results

As we have worked with this exact same dataset in Chapter 17 (the pyBATS model), let's do a final benchmark summary of all models of Chapter 17 and the current chapter. This benchmark is shown in Table 21-1. At this stage, the best performing model is the tuned multivariate N-BEATS, even though the test period PROMO data could not be included. The second place goes to the pyBATS Multivariate Normal DGLM, which has a respectable score as well.

Table 21-1 The building blocks of univariate time series models

Model	Specification	1-MAPE
pyBATS	Univariate Poisson DGLM	0.833
pyBATS	Multivariate normal DGLM	0.867
N-BEATS	Univariate default N-BEATS	0.81
N-BEATS	Multivariate default N-BEATS	0.82
N-BEATS	Tuned multivariate N-BEATS	0.88

Key Takeaways

- The N-BEATS package is a black box forecasting model that uses stacked Feed Forward Neural Networks.

- The Darts package is a Python package that regroups a large number of forecasting models as well as other time series analysis methods.
- The N-BEATS model is that it is easy to set up compared to using Neural Networks directly, and the algorithm based on stacking can improve performance over a regular Neural Network.
- Hyperparameter tuning is easy to do using N-BEATS in Darts, and a large number of hyperparameters are available for tuning.
- Disadvantages of N-BEATS are the complexity of the algorithm (difficult to explain to business stakeholders, for example) and the impossibility to add future covariate data (which is a must-have in a use case like promotions and sales where covariate data strongly impacts the target variable).

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_22

22. The Transformer Model with Darts

Joos Korstanje[1]
(1) Maisons Alfort, France

In the previous chapters, we have covered numerous approaches to Neural Network architectures. In earlier chapters, you have seen how to build forecasting models using Feed Forward Neural Networks, Recurrent Neural Networks, and LSTMs. In the previous chapter, you also saw the N-BEATS, which uses stacks of Feed Forward Neural Networks to build performant forecasting models.

In the current chapter, you will discover another member of the Neural Network family: Transformers. The transformer is a specific type of Neural Network. It was developed in 2017 and is therefore a quite recent addition to the machine learning toolkit. Over the past years, it has proven to be a very useful and valuable model, which is why we will cover it in depth in this chapter.

There are many implementations available to build transformer models. In this chapter, you'll be using the Darts package. You have used the Darts package before for building the N-BEATS model. Darts is a package specifically built for time series forecasting, and one of its main advantages is that it is very easy to use.

Intuition and Mathematics of Transformers

Let's get started by doing a deep dive into the transformer model. The transformer model has been developed especially to deal with sequential data. Current AI use cases are making tremendous progress in use cases involving text or speech. Thanks to these developments, we see a huge spike in models capable of treating sequence data. It just so happens that

forecasting is also treating sequence data, and a lot of the sequential models developed initially for treating text can also be adapted to the field of forecasting.

Attention Is All You Need

In 2017, a paper called “Attention Is All You Need” by Vaswani and others caused an important paradigm shift in the AI and Machine Learning community. The innovation of this paper lies in the fact that it proposes a focus on **attention**.

To understand the concept of attention, let’s look at a simple example from text treatment. We take the following sentence: “The cat sat on the mat because it was tired.” What does the word “it” refer to? To the cat, of course, because the cat is the only thing that can be tired! The invention of the transformer model is that it looks at the whole **sentence at once**, which is a radical change from RNNs and LSTMs.

Transformers Move Away from Recurrent Units

In the previous chapters, you have discovered RNNs and LSTMs. They are all Recurrent Neural Networks. A Recurrent Neural Network would take a sequence and go through it **step-by-step**. This was a major invention that allowed for major advances in Natural Language Processing and Forecasting.

Now, the interesting thing is that Transformers basically cancels this invention of the recurrent approach. The Transformer architecture resembles much closer to the Feed Forward Neural Network architecture, although with a big twist. After all, it wasn’t that easy to model sequence data into Feed Forward Neural Networks. There are two main criteria that make transformers understand sequences.

Positional Encoding

The first important element of Transformers is **Positional Encoding**. Positional Encoding is basically an adaptation that allows Feed Forward Neural Networks to learn positions of values in time, by learning not only their values but also their position in time. We have accomplished this in the Feed Forward Neural Network by doing feature engineering (including the day of week, day of year, etc.), but the Transformer model is designed in

such a way that it will learn these positions without having to do feature engineering.

Self-Attention

The second important element added by Transformers is **Self-Attention**. Self-attention means that the model will be able to focus more on data that is closer to the point being forecasted while still learning from all the data. Throughout this book, you have many times learned how to decide on a number of lagged variables to include in the model and thereby allow a model to learn from its own past. Yet, you always had to decide for yourself on a number of lags to input into the model. Transformers take this difficulty away by learning from all the data and applying higher weights to more recent data.

The Darts Package and Its Implementation of Transformers

Many implementations of the Transformers algorithm exist. Examples are packages like PyTorch and TensorFlow, which are great for Neural Networks and deep learning in general use cases.

In this chapter, we'll use the Transformers algorithm in the Darts package. You have used the Darts package in the previous chapter when using the N-BEATS algorithm. As you'll recall, Darts is a great package specifically focusing on time series. Darts is easy to use, and it'll save you a lot of the hassle of setting up PyTorch or TensorFlow.

In addition, as a forecasting practitioner, it is interesting for you to learn to work with Darts. The package contains a large number of forecasting models, and once you learn how to work with the package, it'll be quite easy to substitute one model with another in your code. For references, you can have a look at their forecasting model documentation to see why it is interesting to master their modeling interface: https://unit8co.github.io/darts/generated_api/darts.models.forecasting.html.

Forecasting Sales Using Transformer in Darts

Let’s now move on to an example implementation of the Transformer model using the Darts package. We’ll be using a dataset (1) that collects sales of an Italian grocery store. You have seen this dataset before: in the chapter on pyBATS and again with N-BEATS. The data contains daily sales amounts per brand per product and also contains information on promotions. After all, promotions may strongly impact sales. At the end of the chapter, you’ll also be able to benchmark Transformers against the previous results from pyBATS and N-BEATS.

(1) Source: A machine learning approach for forecasting hierarchical time series (Paolo Mancuso, Veronica Piccialli, and Antonio M. Sudoso)

Sales Forecast: Preparing the Data

We’ll simply prepare the data in the same manner as before. You may remember that we regrouped the data. In the input data, we have detailed sales per brand, per product, per day. In the prepared dataset, we have the total sales per month. The data preparation is shown in Listing 22-1.

```
import pandas as pd
import matplotlib.pyplot as plt

# Read the data
df = pd.read_csv('hierarchical_sales_data.csv')

df_sum = df[['DATE']]

# Sum the sales and promos of the different
products
df_sum['SALES'] = df[[x for x in df.columns if
x.startswith('QTY')]].sum(axis=1)
df_sum['PROMO'] = df[[x for x in df.columns if
x.startswith('PROMO')]].sum(axis=1)

# Create sum of sales and promos per month
df_sum['YEAR_MONTH'] = df_sum['DATE'].apply(lambda
x: x[:7])
df_sum = df_sum[['SALES', 'PROMO',
'YEAR_MONTH']].groupby('YEAR_MONTH').sum()
df_sum.head()
```

```
# Preview the data
df_sum.head()
```

Listing 22-1 Importing the data

Once you've applied the data preparation from Listing 22-1, you'll obtain a dataset that looks like the one in Figure 22-1.

YEAR_MONTH	SALES	PROMO
2014-01	12819	990
2014-02	17906	1329
2014-03	12047	896
2014-04	15998	1235
2014-05	17453	1354

Figure 22-1 The prepared data

As you have seen in the previous chapter, working with Darts requires some specific formatting. Listing 22-2 converts the sales data into a Darts time series format, which will allow you to access other Darts functionalities. This code also creates a train-test split for model evaluation.

```
import numpy as np
from darts.timeseries import TimeSeries

# Convert sales data to float32 for Darts
df_sum['SALES'] = df_sum['SALES'].map(np.float32)

# Convert the dataframe index to a datetime index
as required by Darts
df_sum.index = pd.DatetimeIndex(df_sum.index)

# Full year 2014 to 2017 is the training data
```

```
train = df_sum.iloc[:-12]

# Full year 2018 is the test data
test = df_sum.iloc[-12:]

# Convert the training data to a Darts time series
train_series =
TimeSeries.from_dataframe(train[['SALES']])

# Inspect the time series format
train_series
```

Listing 22-2 Prepare for Darts and create a univariate train-test split

Sales Forecast: Create a Default N-BEATS Model

As you have already worked with the Darts package, as well as with the sales data, we can quickly dive into the modeling part. Before starting, there is one specific pre-processing step that we need to do. When using the Transformer model in Darts, you need to make sure that your data has been scaled. You can use Darts' built-in scaler for this, as shown in Listing 22-3.

```
# Scale the data based on the training data
from darts.dataprocessing.transformers import
Scaler
my_scaler = Scaler()
train_scaled =
my_scaler.fit_transform(train_series)
```

Listing 22-3 Scaling the data

We can now move on to building a Transformer model. As a first step, let's build a univariate model (using only the sales data) and use as many of the default settings as possible. This will give you a good benchmark performance score and give you a feel of whether or not this model could (at least more or less) fit this data. This is done in Listing 22-4. This code also activates MLflow, which will allow you to retrieve all results from the current chapter stored on your file system (great if training times are long and when results vary based on random initializations).

```
# Use MLFlow autologging for Darts
import mlflow
mlflow.autolog()

# import the TransformerModel
from darts.models import TransformerModel

# Specifying a default model
model = TransformerModel(
    # with input chunk length 24 (number of time
steps used as input)
    input_chunk_length=24,
    # with output chunk 12 (number of time steps
used as output)
    output_chunk_length=12,
    n_epochs=20,
    random_state=123
)

# Train the model
model.fit(train_scaled)
```

Listing 22-4 Create the default model

The model is now trained. You can use the code in Listing 22-5 to compute the 1-MAPE metric and create the forecasting plot. Remember that we had to scale the input data. Because of this, the predictions will also be scaled, and we need to rescale them to the original scale to be able to compare them with the test dataset.

```
from sklearn.metrics import
mean_absolute_percentage_error

# Forecast 12 steps using this model
fcst = model.predict(12)

# Rescale the predictions to the original scale
fcst = my_scaler.inverse_transform(fcst).values()
```

```
# Compute metric
metric = 1 -
mean_absolute_percentage_error(list(test['SALES']),
fcst)
print(metric)

plt.plot(fcst)
plt.plot(list(test['SALES']))
plt.legend(['fcst', 'test'])
```

Listing 22-5 Predict, rescale, and evaluate

You will obtain a 1-MAPE score of **0.78**, which is not great, but not horrible either. The plot of the forecast is shown in Figure 22-2.

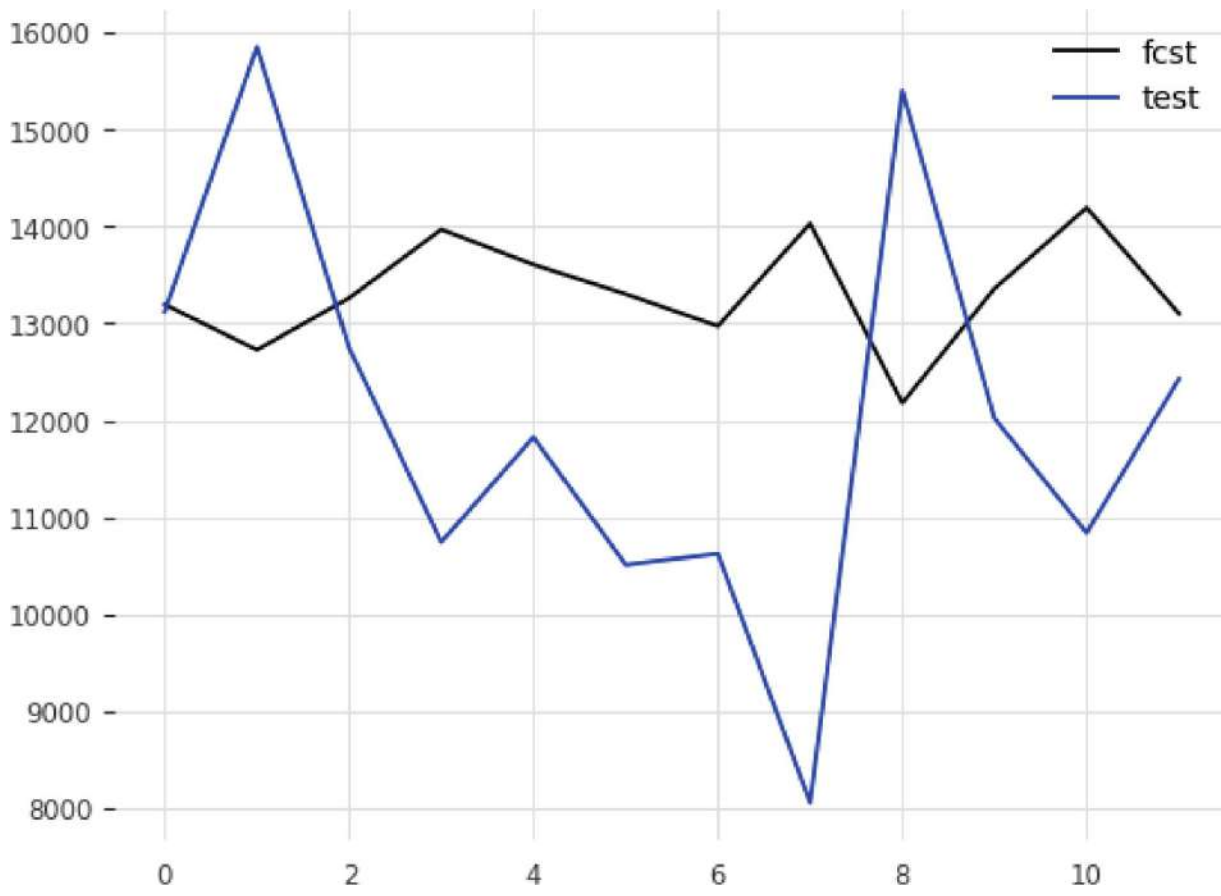


Figure 22-2 The plot of the simple transformer forecast

You can see that the forecast has not been able to capture much variation. It is almost a flat line, very close to the overall average. We have

a clear case of an underfitting model. Adding the PROMO variable may be able to help us improve performance.

Sales Forecast: Create the Multivariate Transformer Model

Let's now add the PROMO variable into the model. As we have seen previously, a drawback of using Darts is that it does not currently allow using future values of the independent variable. Despite that, let's see whether we can improve the performance by adding PROMO into the mix. As a first step, we need to convert PROMO into a Darts time series and build a scaler to adapt it for the Transformer model. This is done in Listing 22-6.

```
df_sum['PROMO'] = df_sum['PROMO'].map(np.float32)

# Convert the training data to a Darts time series
promos_train =
TimeSeries.from_dataframe(train[['PROMO']])

# Convert to float32 for Darts
promos_train = promos_train.astype(np.float32)

promo_scaler = Scaler()
promos_train =
promo_scaler.fit_transform(promos_train)
```

Listing 22-6 Preparing the PROMO data

When this is done, you can use the code in Listing 22-7 to create, train, and evaluate the multivariate transformer model.

```
# Specifying a default model
model = TransformerModel(
    # with input chunk length 24 (number of time
steps used as input)
    input_chunk_length=24,
    # with output chunk 12 (number of time steps
used as output)
    output_chunk_length=12,
    n_epochs=20,
```

```
    random_state=123
)

# Train the model
model.fit(
    train_scaled,

    # add the promotions as past covariates
    past_covariates=promos_train
)

# Forecast 12 steps using this model
fcst = model.predict(
    n=12,

    # add past covariates (promos train)
    past_covariates=promos_train,
)

# Rescale the predictions to the original scale
fcst = my_scaler.inverse_transform(fcst).values()

# Compute metric
metric = 1 -
mean_absolute_percentage_error(list(test['SALES']),
fcst)
print(metric)

plt.plot(fcst)
plt.plot(list(test['SALES']))
plt.legend(['fcst', 'test'])
```

Listing 22-7 The multivariate transformer model

The multivariate Transformer model slightly improves the performance metric: we now obtain **0.81**, which is an improvement of around 0.03.

Sales Forecast: Tuning the Multivariate Transformer Model

As a last step, let's see how tuning the hyperparameters of the multivariate transformer model may impact the performance on the test data. Transformers are complex models, and they have numerous hyperparameters that can be tuned. A subset has been chosen for this chapter.

If you have the motivation to discover more about Transformers, it will be an interesting exercise to find other tunable hyperparameters in the documentation (https://unit8co.github.io/darts/generated_api/darts.models.forecasting.transformer_model.html#darts.models.forecasting.transformer_model.TransformerModel) and add them into the loop. Be careful: when doing GridSearches like this, execution time will increase exponentially when adding more for loops with hyperparameter combinations. A good strategy can be to first find a good variable for one or multiple hyperparameters, then fix it and tune for other hyperparameters.

The grid search is shown in Listing 22-8, and it optimizes for the following hyperparameters:

- input_chunck_length
- n_epochs
- d_model: The number of expected features in the encoder/decoder
- nhead: The number of heads in the multihead attention mechanism
- num_encoder_layers: The number of encoder layers
- num_decoder_layers: The number of decoder layers

```
def score_nbeats(input_chunck_length, n_epochs,
d_model, nhead, num_encoder_layers,
num_decoder_layers):

    # Specifying a model using the gridsearch
parameters
    model = TransformerModel(
        input_chunk_length=input_chunck_length,
        output_chunk_length=12,
        n_epochs=n_epochs,
        d_model=d_model,
        nhead=nhead,
```

```
        num_encoder_layers=num_encoder_layers,
        num_decoder_layers=num_decoder_layers,
        random_state=123
    )

    # Train the model
    model.fit(
        train_scaled,

        # add the promotions as past covariates
        past_covariates=promos_train
    )

    # Forecast 12 steps using this model
    fcst = model.predict(
        n=12,

        # add past covariates (promos train)
        past_covariates=promos_train,

    )

    # Rescale the predictions to the original scale
    fcst =
my_scaler.inverse_transform(fcst).values()

    # Compute metric
    metric = 1 -
mean_absolute_percentage_error(list(test['SALES']),
fcst)

    return metric

#d_model=64, nhead=4, num_encoder_layers=3,
num_decoder_layers=3

# Run a Grid Search
grid_result = []
```

```
for input_chunck_length in [12, 24]:
    for n_epochs in [5, 10]:
        for d_model in [32, 64]:
            for nhead in [4, 8]:
                for num_encoder_layers in [2, 3,
4]:
                    for num_decoder_layers in [3,
4]:
                        score =
score_nbeats(input_chunck_length, n_epochs,
d_model, nhead, num_encoder_layers,
num_decoder_layers)
                        result =
[input_chunck_length, n_epochs, d_model, nhead,
num_encoder_layers, num_decoder_layers, score]
                        grid_result.append(result)

# Inspect the results in a DataFrame
grid_output = pd.DataFrame(grid_result, columns=[
    'input_chunck_length',
    'n_epochs',
    'd_model',
    'nhead',
    'num_encoder_layers',
    'num_decoder_layers',
    'score'
])

grid_output.sort_values('score',
ascending=False).head()
```

Listing 22-8 Tuning the multivariate transformer model

The code in Listing 22-8 will generate the table shown in Figure 22-3.

	input_chunck_length	n_epochs	d_model	nhead	num_encoder_layers	num_decoder_layers	score
88	24	10	64	4	4	3	0.839653
89	24	10	64	4	4	4	0.839218
95	24	10	64	8	4	4	0.836948
60	24	5	64	4	2	3	0.836611
94	24	10	64	8	4	3	0.836331

Figure 22-3 The results of the grid search

In this table, you can see that the highest obtained performance is (rounded up to two decimals) **0.84**. This model has the following hyperparameters:

- input_chunck_length=24
- n_epochs=10
- d_model=64
- nhead=4
- num_encoder_layers=4
- num_decoder_layers=3

You can use Listing 22-9 to fit the best model and obtain a plot of the forecast against the test data.

```
input_chunck_length=24
n_epochs=10
d_model=64
nhead=4
num_encoder_layers=4
num_decoder_layers=3

# Specifying a model using the gridsearch
parameters
model = TransformerModel(
    input_chunk_length=input_chunck_length,
    output_chunk_length=12,
    n_epochs=n_epochs,
    d_model=d_model,
    nhead=nhead,
    num_encoder_layers=num_encoder_layers,
    num_decoder_layers=num_decoder_layers,
```

```
    random_state=123
)

# Train the model
model.fit(
    train_scaled,

    # add the promotions as past covariates
    past_covariates=promos_train
)

# Forecast 12 steps using this model
fcst = model.predict(
    n=12,

    # add past covariates (promos train)
    past_covariates=promos_train,

)

# Rescale the predictions to the original scale
fcst = my_scaler.inverse_transform(fcst).values()

# Compute metric
metric = 1 -
mean_absolute_percentage_error(list(test['SALES']),
fcst)
print(metric)

# Plot the best forecast
plt.plot(fcst)
plt.plot(list(test['SALES']))
plt.legend(['fcst', 'test'])
```

Listing 22-9 Recreate the best model and evaluate it

This code will confirm the performance score (1-MAPE) of **0.84**. It will also generate the plot shown in Figure 22-4.

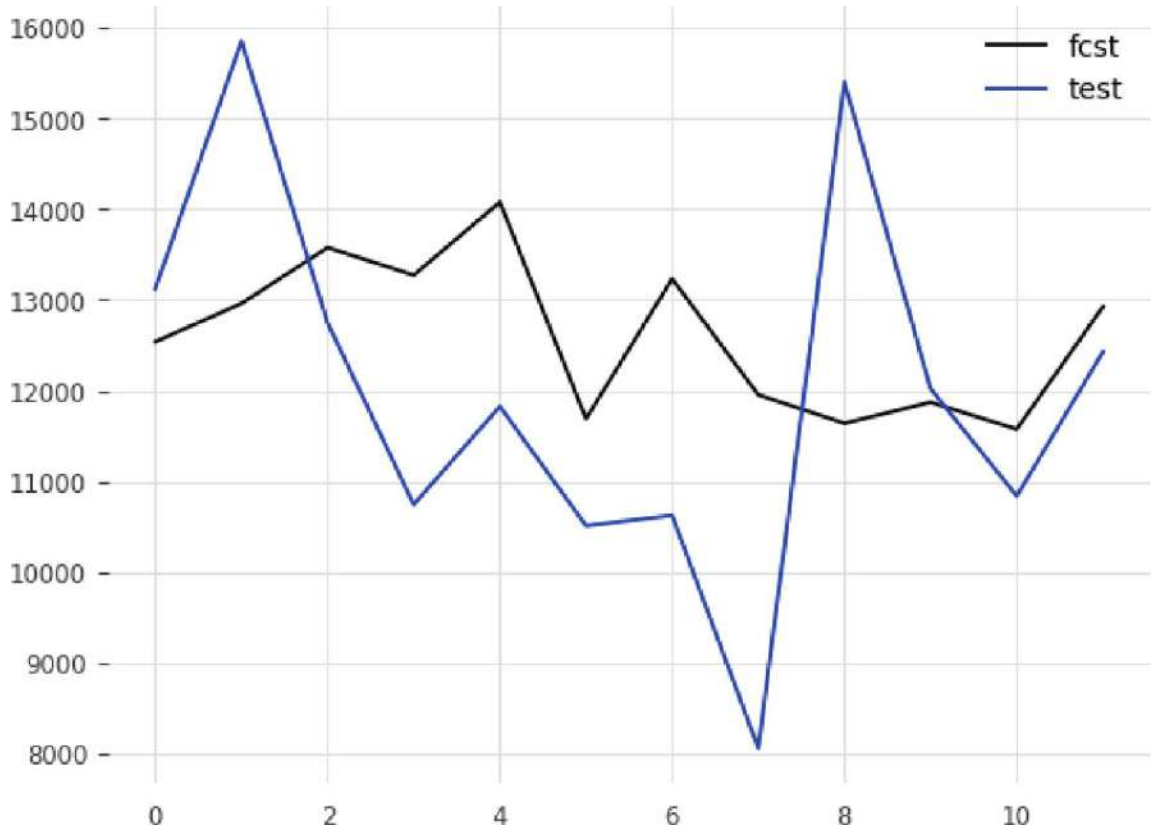


Figure 22-4 The forecast made by the tuned transformer

As you can see in Figure 22-4, the performance metric of this model is higher than the other, but the forecast is still lacking in fit. This is a reality of building forecasting models on applied use cases: not all models are able to reach peak performance on all datasets, which is why benchmarking your results between different models is such an important exercise.

Benchmark Results

As you'll certainly remember, we have used the Grocery Sales Forecast for three state-of-the-art models now: pyBATS, N-BEATS, and Transformers. Each of these models is state of the art, and it would be impossible to know up front which of the three should be applied to a given use case.

This makes for quite an interesting benchmark. Table 22-1 shows all the results of the three models that were applied to this use case. The detail of each version of the models is also specified.

Table 22-1 Benchmark of pyBATS, N-BEATS, and Transformers

Model	Specification	1-MAPE
pyBATS	Univariate Poisson DGLM	0.833
pyBATS	Multivariate normal DGLM	0.867
N-BEATS	Univariate default N-BEATS	0.81
N-BEATS	Multivariate default N-BEATS	0.82
N-BEATS	Tuned multivariate N-BEATS	0.88
Transformers	Univariate default Transformer	0.78
Transformers	Multivariate default Transformer	0.81
Transformers	Multivariate tuned Transformer	0.84

The conclusion from reading this table is that the best performance has been obtained by the tuned multivariate N-BEATS model. The Transformers model has not been able to beat the tuned N-BEATs, and its performance is also worse than the Multivariate Normal DGLM that we built using pyBATS. It is well possible that more tuning would allow the Transformer's performance to improve. Another hypothesis is that the N-BEATS and pyBATS models simply are a better fit to learn from this data.

This once again shows the importance of mastering model evaluation: it is important to steer away from choosing models based on "popularity." It will be much more valuable to your business case to have a performant model, and using recently published models is not always a guarantee for performance on your particular dataset.

Key Takeaways

- Transformers are an important development in the AI field of treating sequence data.
- The idea of transformers is to drop the idea of Recurrent Neural Networks as used in RNNs and LSTMs.
- Transformers are based on encoders/decoders and add two important approaches:
  - Positional Encoding
  - Self-Attention
- Darts contains an implementation of Transformers for time series data. Other, more standard implementations can be found in PyTorch and

TensorFlow.

- Although this state-of-the-art model can often generate great performance on sequence data, it also has disadvantages. Disadvantages of Transformers can be the complexity of the model, causing it to need a lot of data, as well as the difficulty in obtaining clear business logic explanations of what the model is learning.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_23

23. The NeuralProphet Model

Joos Korstanje[1]
(1) Maisons Alfort, France

The **Prophet model** is *an automated procedure* for building forecasting models, which was open sourced by Facebook in 2017. It gained strong popularity in the period of 2018–2021. The main selling point was that the Prophet model does a lot of the work for you. It has a high level of user-friendliness: it does not require any theory to get started, as opposed to the previous chapters.

A disadvantage of this is that you have fewer possibilities to understand exactly how and what the model is learning. There are also fewer parameters to tweak, which may be an advantage if the model works, but it may be a disadvantage if you do not obtain the required model precision.

A quote from the developers explains the goal of Facebook's Prophet:

> *We use a simple, modular regression model that often works well with*
>
> *default parameters, that allows analysts to select the components that are relevant to their forecasting problem and easily make adjustments as needed. The second component is a system for measuring and tracking forecast accuracy, and flagging forecasts that should be checked manually to help analysts make incremental improvements.*
> —https://peerj.com/preprints/3190/

Despite Prophet gaining popularity quickly, it soon came under widespread criticism. A scandal happened at the company Zillow, which used Prophet for price forecasting. Zillow lost a large amount of money because of wrong forecasts by Prophet.

In the end, Facebook has shut down the support for Prophet, and although it is still usable as of today, Prophet has not been maintained properly since 2022. It would therefore be better to steer clear of it.

The NeuralProphet Model

A replacement for Prophet is the NeuralProphet model, which was initially backed by Facebook researchers as well. NeuralProphet is a hybrid algorithm between Prophet and Neural Networks, all built on PyTorch. It solves some of the problems that were encountered by Prophet.

At the same time, NeuralProphet has not been able to create a wave of popularity as Prophet has. The reparations that the NeuralProphet proposes also come at a cost: they do solve for the fundamental lacks in Prophet, but the algorithm is slower, and it suffers from less interpretability.

The initial added value of Prophet was its simplicity. The simplicity came at a cost of problems with performance and a lack of flexibility. NeuralProphet solves some of the performance and flexibility issues, but now the simplicity of Prophet cannot be reached anymore.

Despite Prophet having become a controversial topic in the field of forecasting, it will be interesting to learn what all the fuss is about. Given that Prophet is not maintained anymore, in this chapter, we'll work through a use case with NeuralProphet.

Predicting Heat Waves with NeuralProphet

In this chapter, we will be using Wikipedia data about the number of Page Views of the Wikipedia page on Heat Waves (https://en.wikipedia.org/wiki/Heat_wave). The hypothesis here is that when heat waves happen, people may look up information about heat waves on Wikipedia. Therefore, we may be able to use spikes in heat wave page views as a proxy indicator for actual heat waves occurring. If we can correctly predict the next spike in heat wave page views, we may actually have predicted a coming heat wave!

Wikipedia Pageview Tool

Wikipedia has a great tool that allows you to obtain page views on any page that you may be interested in, which can be of great use in use cases like demand forecasting. The Wikipedia views data can help you understand trends and seasonality in the popularity of certain topics. The tool is accessible through https://pageviews.wmcloud.org/.

To follow along with the current example, you'll need to download the data on Heat Waves from July 2015 to May 2025, in CSV format, which you can do

through this link:https://pageviews.wmcloud.org/?project=en.wikipedia.org&platform=all-access&agent=user&redirects=0&start=2015-07&end=2025-05&pages=Heat_wave.

Preparing the Data in Python

You can use the code in Listing 23-1 to prepare the data.

```
import pandas as pd
import matplotlib.pyplot as plt

data = pd.read_csv('heatwave-pageviews-20150701-
20250531.csv')
data
```

Listing 23-1 Importing the data

This will yield the table as shown in Figure 23-1.

	Date	Heat wave
0	2015-07	26379
1	2015-08	18584
2	2015-09	12529
3	2015-10	15343
4	2015-11	13545
...	...	...
114	2025-01	11067
115	2025-02	11158
116	2025-03	17925
117	2025-04	24095
118	2025-05	32662

119 rows × 2 columns

Figure 23-1 The data

Let's use the code in Listing 23-2 to plot the page views over time.

```
data.set_index('Date').plot()
```

Listing 23-2 Plotting the data

This code will generate the plot shown in Figure 23-2.

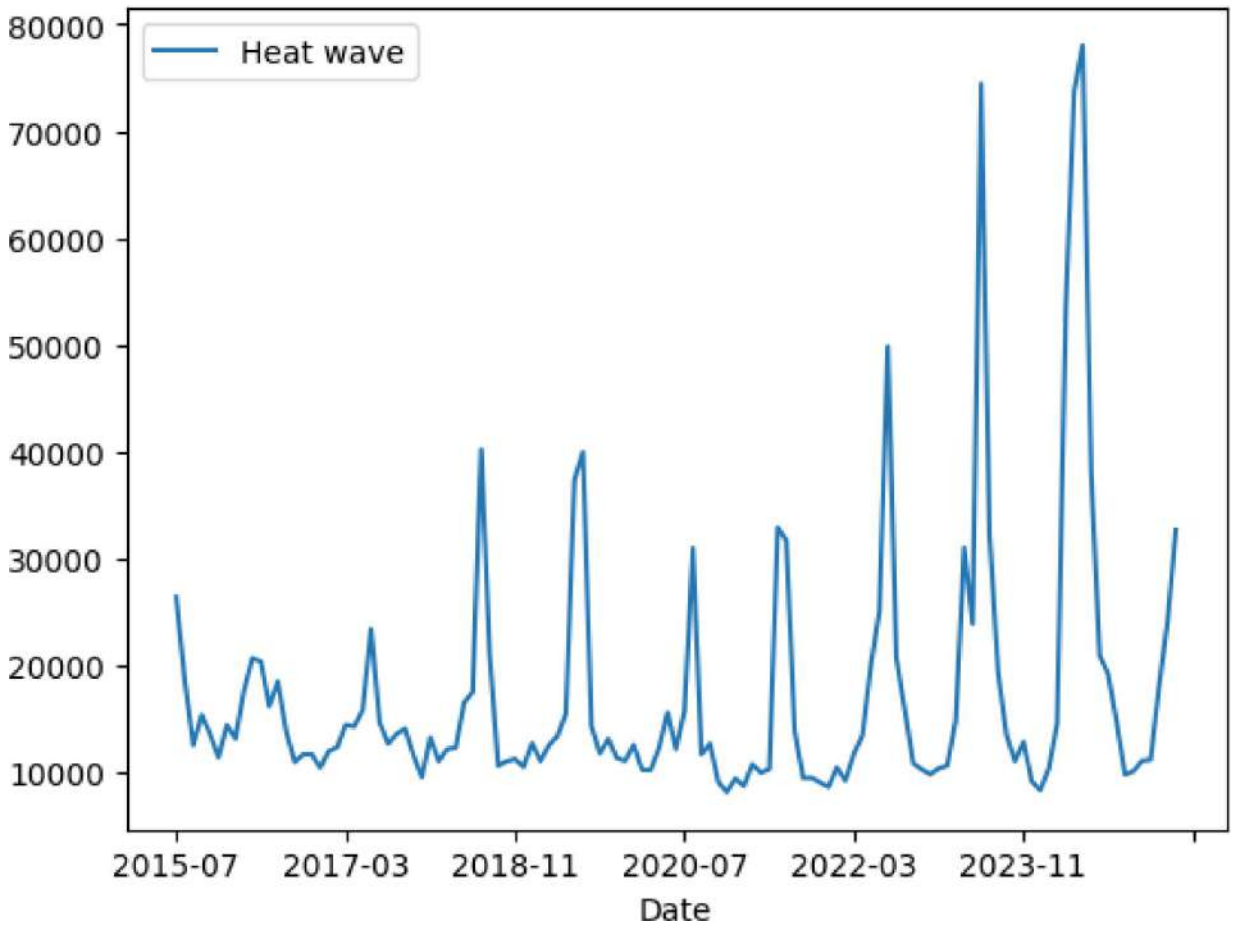


Figure 23-2 Plot of the time series

Even from this simple plot, we can already see clearly that there are multiple time series processes at play. First of all, we see a clear **seasonality**: we have a basis point of around 10,000 monthly page views and regular spikes going from 20,000 to 80 000 monthly page views. Secondly, we also see a clear **trend** in the peaks: the peaks get higher over time. As a first data exploration, this gives us a good feel for the data that we are working with.

Time Series Decomposition Using NeuralProphet

As you should remember from earlier chapters, there are great methods available to identify trends and seasonality in a time series. NeuralProphet also allows you to inspect a decomposed time series. Unfortunately, the decomposition method of NeuralProphet is quite complex: it requires you to train a NeuralProphet model on your data, then build a forecast on your own data period, and only then can you look at the decomposed results. If your goal is simply to decompose a time series into trend and a yearly seasonality, you'll probably be better off using the methods presented in earlier chapters. Yet, as we are building a model with NeuralProphet in this chapter, it will be interesting to see how to obtain the decomposed time series as well.

Before being able to apply this methodology, you'll need to adapt your data to the NeuralProphet package. For this, you need to rename the “Date” column to “ds” and the “Heat wave” column to “y”. Listing 23-3 shows you how to do this.

```
data = data.rename({
    'Date': 'ds',
    'Heat wave': 'y',
}, axis=1)

data.head()
```

Listing 23-3 Plotting the data

You'll obtain the prepared dataset as shown in Figure 23-3.

	ds	y
0	2015-07	26379
1	2015-08	18584
2	2015-09	12529
3	2015-10	15343
4	2015-11	13545

Figure 23-3 Data prepared for NeuralProphet

When your data is prepared, you can use the code in Listing 23-4 to apply the decomposition methodology. Note the following settings:

- Trend
  - growth='linear'
  - n_changepoints=0
- Seasonality
  - yearly_seasonality=True
  - weekly_seasonality=False
  - daily_seasonality=False

This means that we apply a linear trend (n_changepoints means we have only one constant slope) and that we fit only a yearly seasonality pattern.

```
from neuralprophet import NeuralProphet

# Fit a model to be able to inspect decomposition
m = NeuralProphet(
    # Trend settings: fit one linear trend without
changepoints
    growth='linear',
    n_changepoints=0,

    # Seasonality settings: fit yearly seasonality
    yearly_seasonality=True,
    weekly_seasonality=False,
    daily_seasonality=False,
)

metrics = m.fit(data)

# We need to build a forecast to be able to inspect
decomposition
fcst = m.predict(data)

# Use plot components to inspect decomposition
m.set_plotting_backend("plotly-static")
m.plot_components(fcst)
```

Listing 23-4 Applying the decomposition

You'll obtain the decomposition plot shown in Figure 23-4.

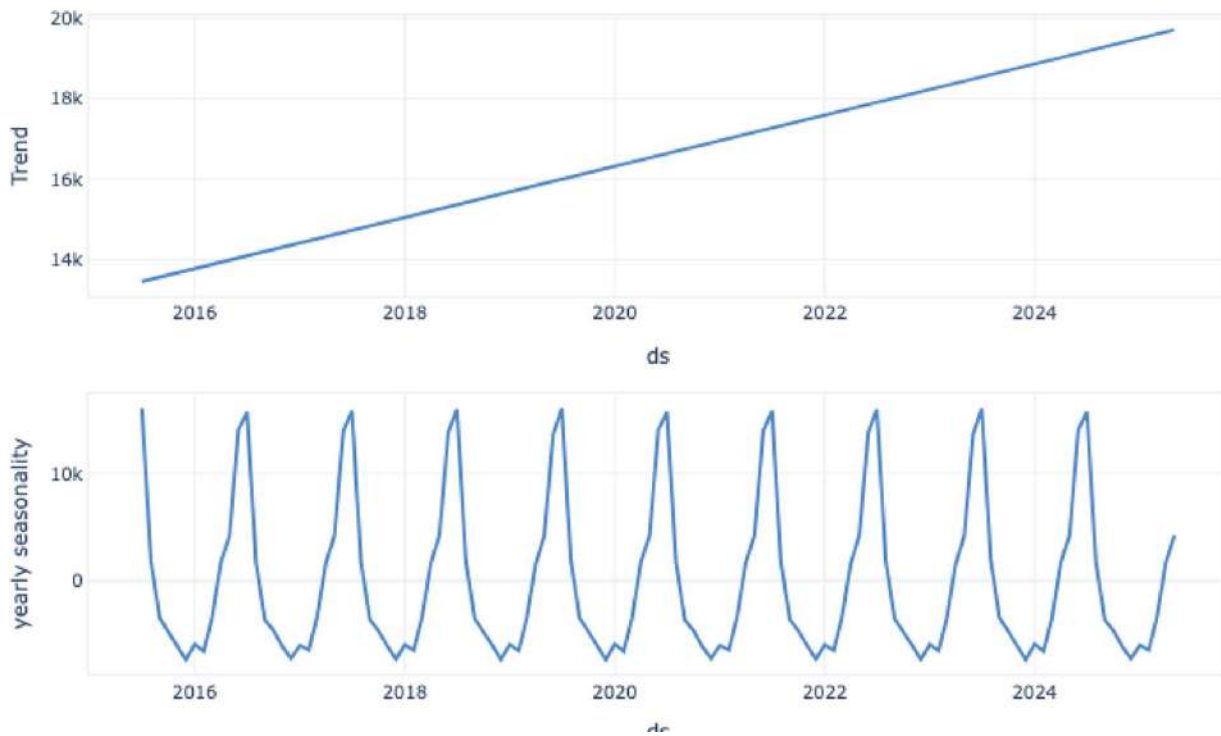


Figure 23-4 The time series is decomposed into a linear trend and a yearly seasonality

You can clearly see the linear trend and the yearly seasonality. As we requested a linear trend without changepoints, you see the constant upward trend that should more or less reflect the increase in yearly peaks that we saw in the initial plot of the time series. The seasonality seems to reflect the peaks and troughs of the heat wave data quite well.

Advanced Decomposition Setting

Now, there would be no interest in using a complex neural network-based technique to create a decomposition plot if the only goal was to fit a trend and yearly seasonality. NeuralProphet allows you to model a bunch of much more complex patterns in the decomposition. Let's have a look at some of the options that NeuralProphet provides:

- Trend settings
  - Growth
    - Linear
    - Discontinuous
  - changepoints: Manually set dates at which you want your trend to be split.

    - n_changepoints: Define a specific number of changepoints that the model will identify.
    - Numerous settings related to how the trend line and changepoints are identified.
- Seasonality settings
    - Seasonality
        - Yearly
        - Weekly
        - Daily
    - Seasonality mode
        - Additive
        - Multiplicative
    - Fit each seasonality automatically or specify the number of Fourier/linear terms to generate

Of course, it will depend on your data whether or not all these settings are needed. In the current example on the heat wave page views, we could imagine that we are unhappy with the linear constant trend in page views. If you go back to Figure 23-2, you can see that the peaks in later dates are much higher than the peaks in earlier dates. We could imagine having three phases in peaks:

1. Peaks from 2015 to 2017 are quite low.
2. Peaks from 2018 to 2021 are medium-high.
3. Peaks after 2021 are very high.

A linear trend does not seem to capture this quite well, so let's try to allow the trend to fit three separate slopes. To do this, we'll set the n_changepoints to 2 (two changepoints will cut the total time series into three periods). This is shown in Listing 23-5.

```
# Fit a model to be able to inspect decomposition
m = NeuralProphet(
    # Trend settings: fit a linear trend with 2
changepoints
    growth='linear',
```

```
    n_changepoints=2,

    # Seasonality settings: fit yearly seasonality
    yearly_seasonality=True,
    weekly_seasonality=False,
    daily_seasonality=False,
)

metrics = m.fit(data)

# We need to build a forecast to be able to inspect
decomposition
fcst = m.predict(data)

# Use plot components to inspect decomposition
m.set_plotting_backend("plotly-static")
m.plot_components(fcst)
```

Listing 23-5 Applying the decomposition with two changepoints

You will obtain the plot shown in Figure 23-5.

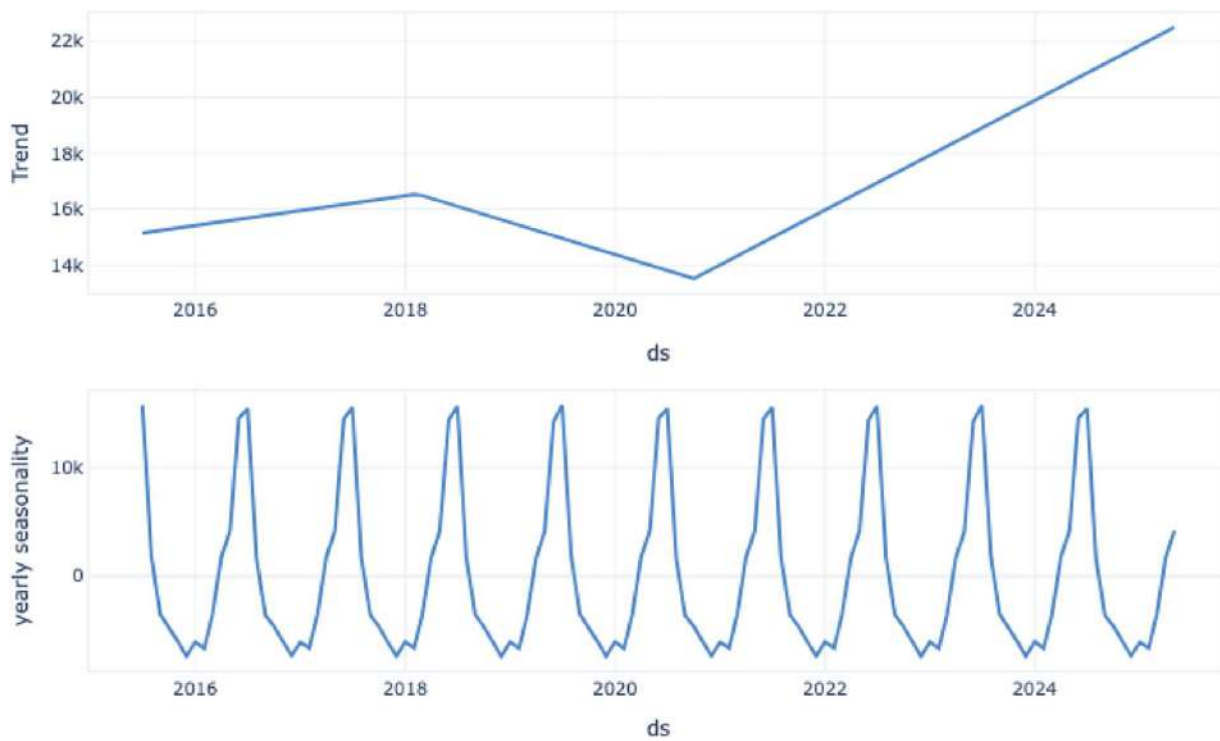


Figure 23-5 The trend has been cut into three phases

The trend line in Figure 23-5 shows how the two changepoints have been estimated by the model. It has chosen a first cut of the trend line at the beginning of 2018 and a second one around the end of 2020. This fits well with the three phases that were expected based on Figure 23-2.

An advantage of having these three phases of trend line is that we may be able to create a better forecasting model. If we'd force the model to use the trend line from Figure 23-4, we might have a lot of difficulty forecasting very high peaks. The trend line in Figure 23-5 is much steeper and therefore may be able to forecast high peaks into the future.

Train-Test Split

There is an important detail that we have not yet spoken about here: train-test splits. Of course, it is very easy to build a trend line that fits well when you have access to all the data. However, the goal is to predict the future, and who knows whether we will remain in the third phase of high peaks or whether we will remain stable or go back to low peaks.

The goal is not just to find a model that fits the training data: this would cause strong overfitting to the training data, and predictive performance would likely be bad. As we are moving into the modeling phase, let's create a train-test split using Listing 23-6.

```
train = data.iloc[:-17]
test = data.iloc[-17:-5]
test.set_index('ds').plot()
```

Listing 23-6 Apply the train-test split

We will use the full year of 2024 as a test period. It contains a nice peak, which will allow us to make a good model evaluation. Figure 23-6 shows the peak that will be used as test data.

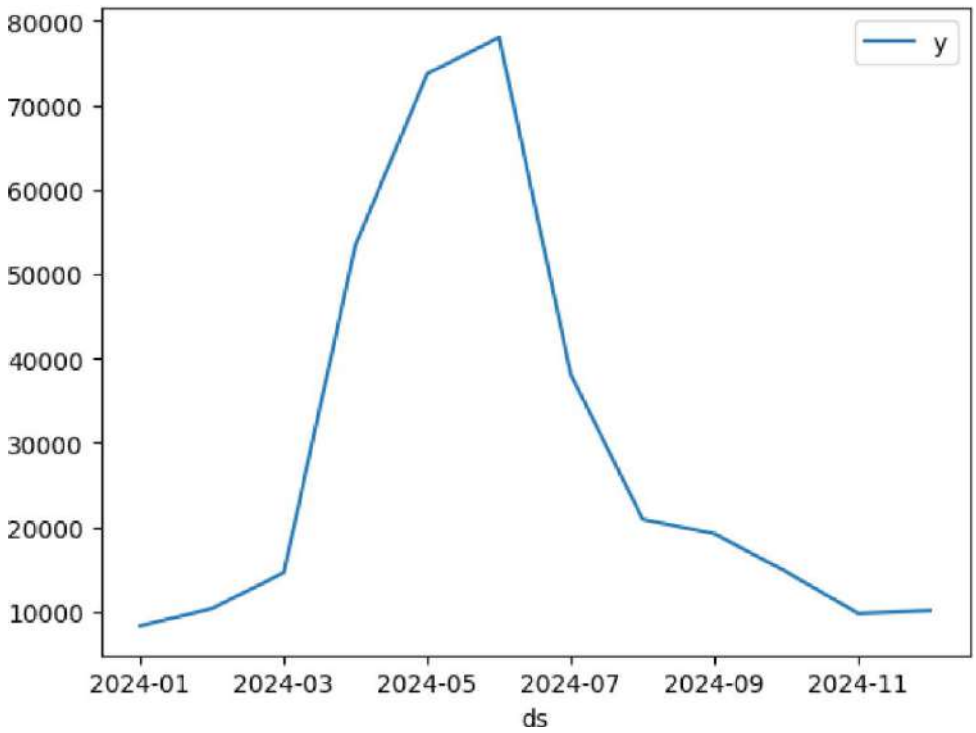


Figure 23-6 The test data

Default Model

As a first predictive model, let's take the first default model that was also used for time series decomposition: it allows for one linear trend and yearly seasonality. We initially trained it on the full dataset, so we need to adapt the code to fit only the training data. Then we need to create the forecast only on the test data. This is shown in Listing 23-7.

```
# Set seed for reproducibility
from neuralprophet import set_random_seed
set_random_seed(0)

m = NeuralProphet(
    # Trend settings: fit one linear trend without
changepoints
    growth='linear',
    n_changepoints=0,

    # Seasonality settings: fit yearly seasonality
    yearly_seasonality=True,
    weekly_seasonality=False,
```

```
    daily_seasonality=False,
)

# Fit only on train data
metrics = m.fit(train)

#Forecast on test data
fcst = m.predict(test)

fcst
```

Listing 23-7 Train a default model

When running this code, you'll see that the forecast is a dataframe as shown in Figure 23-7.

	ds	y	yhat1	trend	season_yearly
0	2024-01-01	8292	12674.435547	16429.958984	-3755.523193
1	2024-02-01	10330	11060.708008	16446.187500	-5385.480469
2	2024-03-01	14590	13283.302734	16461.371094	-3178.068115
3	2024-04-01	53466	16168.674805	16477.599609	-308.924316
4	2024-05-01	73760	17860.632812	16493.304688	1367.327026
5	2024-06-01	78038	28227.638672	16509.535156	11718.105469
6	2024-07-01	38028	32455.425781	16525.242188	15930.184570
7	2024-08-01	20907	19307.664062	16541.468750	2766.193359
8	2024-09-01	19192	12812.809570	16557.699219	-3744.889648
9	2024-10-01	14689	13074.042969	16573.404297	-3499.360840
10	2024-11-01	9757	11376.944336	16589.632812	-5212.689453
11	2024-12-01	10098	10858.982422	16605.339844	-5746.356934

Figure 23-7 The forecast dataframe

This dataframe contains the test data ('ds' and 'y'), as well as the predictions ('yhat1'), and their decomposed values ('trend' and 'season_yearly'). You can use Listing 23-8 to compute a performance score (1-MAPE) of this forecast, as well as generate a plot of the actual test data vs. the predicted forecast.

```
# Compute 1-MAPE performance indicator
from sklearn.metrics import
mean_absolute_percentage_error
perf=1-mean_absolute_percentage_error(fcst['y'],
fcst['yhat1'])
print(perf)

# Plot the forecast against the test data
fcst.set_index('ds')['y'].plot()
fcst.set_index('ds')['yhat1'].plot()
plt.legend(['y', 'yhat1'])
```

Listing 23-8 Evaluate the default model

This default model obtains a 1-MAPE score of **0.69**, and the code generates the forecasting plot shown in Figure 23-8.

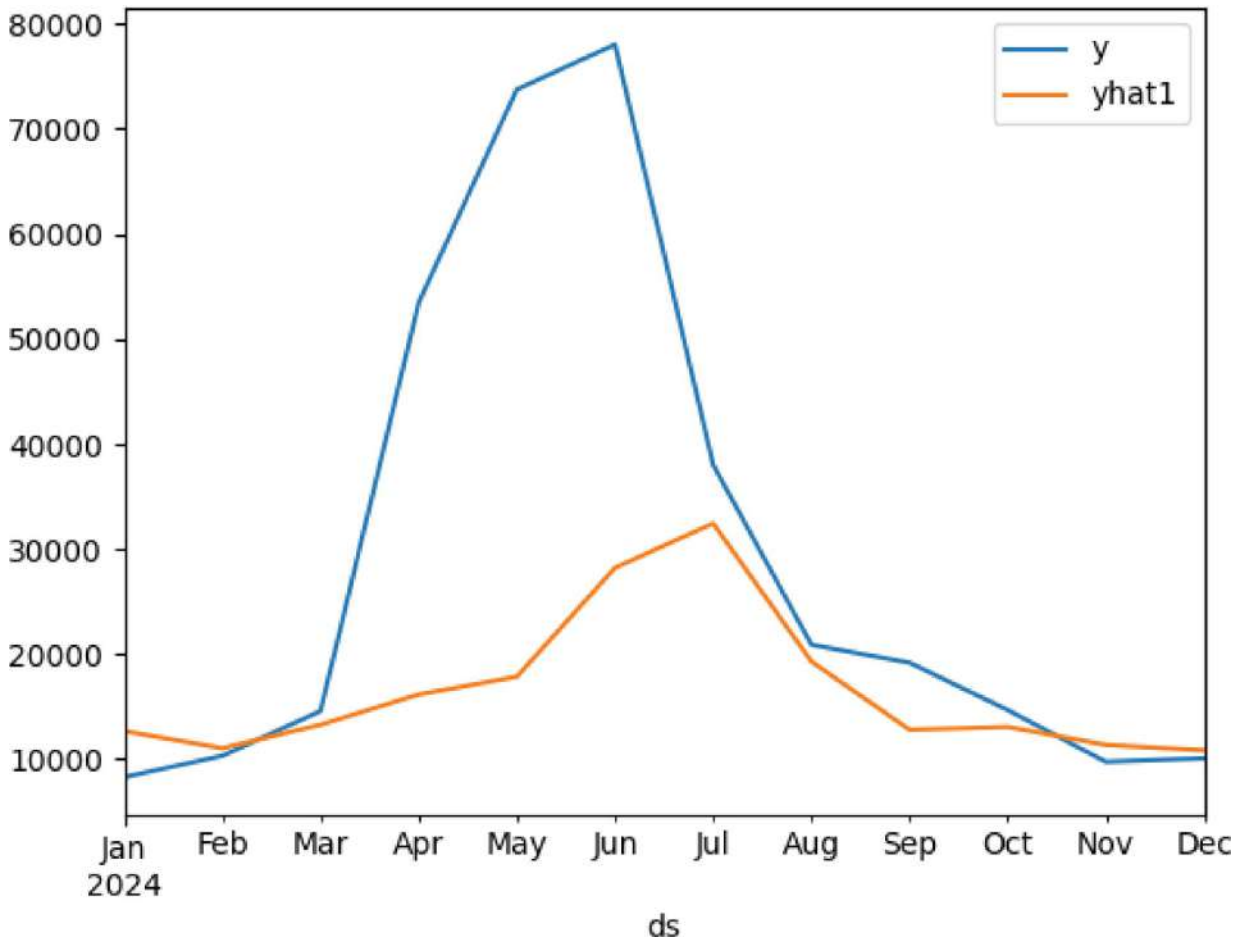


Figure 23-8 The forecast plot

As you can see, both from the score of 0.69 and from the plot, this fit is not great. Especially the plot shows what has gone wrong: the seasonality was fitted more or less well (peak where a peak should be, trough where a trough should be), but the height of the peak is totally off.

Tuned Model

We do not have to stop at this performance, though. NeuralProphet has a lot of hyperparameters and settings that we can try to tune. In the next example, we'll be applying a grid search to tune the following hyperparameters:

- Trend settings
  - Growth
    - Linear
    - Discontinuous
  - n_changepoints: [5, 10, 20]

- Seasonality settings
  - Seasonality mode
    - Additive
    - Multiplicative
- Autoregressive effect: We'll try adding an autoregressive effect with the following trials:
  - n_lags: The number of autoregressive lags to include, we'll try 1, 2
- Loss function (loss_func): We'll replace it with Mean Absolute Error, which is closer to our MAPE indicator than the default SmoothL1Loss.
- Prioritize more recent samples:
  - newer_samples_weight: An additional weight for more recent samples; we'll try 2, 5, or 10.
  - newer_samples_start: Defines when newer samples start; set this to 88%, so that the whole year of 2023 is considered newer samples.

This is done in Listing 23-9.

```
def fit_evaluate_neuralprophet(growth,
n_changepoints, seasonality_mode, n_lags,
newer_samples_weight):
    m = NeuralProphet(
        growth=growth,
        n_changepoints=n_changepoints,
        yearly_seasonality=True,
        weekly_seasonality=False,
        daily_seasonality=False,
        seasonality_mode=seasonality_mode,
        n_lags=n_lags,
        loss_func='MAE',
        newer_samples_weight=newer_samples_weight,
        newer_samples_start=0.88,
        trainer_config={'logger': None}
    )

    metrics = m.fit(train)

    fcst = m.predict(test)
```

```
    mape = 1-
mean_absolute_percentage_error(fcst['y'].iloc[2:],
fcst['yhat1'].iloc[2:])

    return mape

results = []
for growth in ['linear', 'discontinuous']:
    for n_changepoints in [5, 10, 20]:
        for n_lags in [1, 2]:
            for seasonality_mode in ['additive',
'multiplicative']:
                for newer_samples_weight in [10, 20]:
                    mape =
fit_evaluate_neuralprophet(growth, n_changepoints,
seasonality_mode, n_lags, newer_samples_weight)
                    result = [growth, n_changepoints,
seasonality_mode, n_lags, newer_samples_weight, mape]
                    print(result)
                    results.append(result)

results = pd.DataFrame(results, columns=['growth',
'n_changepoints', 'seasonality_mode' 'n_lags',
'newer_samples_weight', 'mape'])
results.sort_values('mape', ascending=False)
```

Listing 23-9 Tune the NeuralProphet

You will obtain the results as shown in Figure 23-9.

	growth	n_changepoints	seasonality_mode	n_lags	newer_samples_weight	mape
35	discontinuous	10	multiplicative	1	20	0.776845
47	discontinuous	20	multiplicative	2	20	0.772628
13	linear	10	additive	2	20	0.768706
38	discontinuous	10	multiplicative	2	10	0.763307
9	linear	10	additive	1	20	0.763227

Figure 23-9 The grid search results

This shows that the best model is the one with discontinuous growth, ten changepoints, seasonality_mode is multiplicative, one lag, and 20 newer_samples_weights. This obtains a forecasting score of 1-MAPE of **0.7768**. The code in Listing 23-10 recreates this model and plots the forecasting plot.

```
growth='discontinuous'
n_changepoints=10
seasonality_mode = 'multiplicative'
n_lags=1
newer_samples_weight=20

m = NeuralProphet(
    growth=growth,
    n_changepoints=n_changepoints,
    yearly_seasonality=True,
    weekly_seasonality=False,
    daily_seasonality=False,
    seasonality_mode=seasonality_mode,
    n_lags=n_lags,
    loss_func='MAE',
    newer_samples_weight=newer_samples_weight,
    newer_samples_start=0.88,
    trainer_config={'logger': None}
)

metrics = m.fit(train)

fcst = m.predict(test)

mape = 1-
mean_absolute_percentage_error(fcst['y'].iloc[2:],
fcst['yhat1'].iloc[2:])
print(mape)

# Plot the forecast against the test data
fcst.set_index('ds')['y'].plot()
fcst.set_index('ds')['yhat1'].plot()
plt.legend(['y', 'yhat1'])
```

Listing 23-10 The best model

The resulting plot is shown in Figure 23-10.

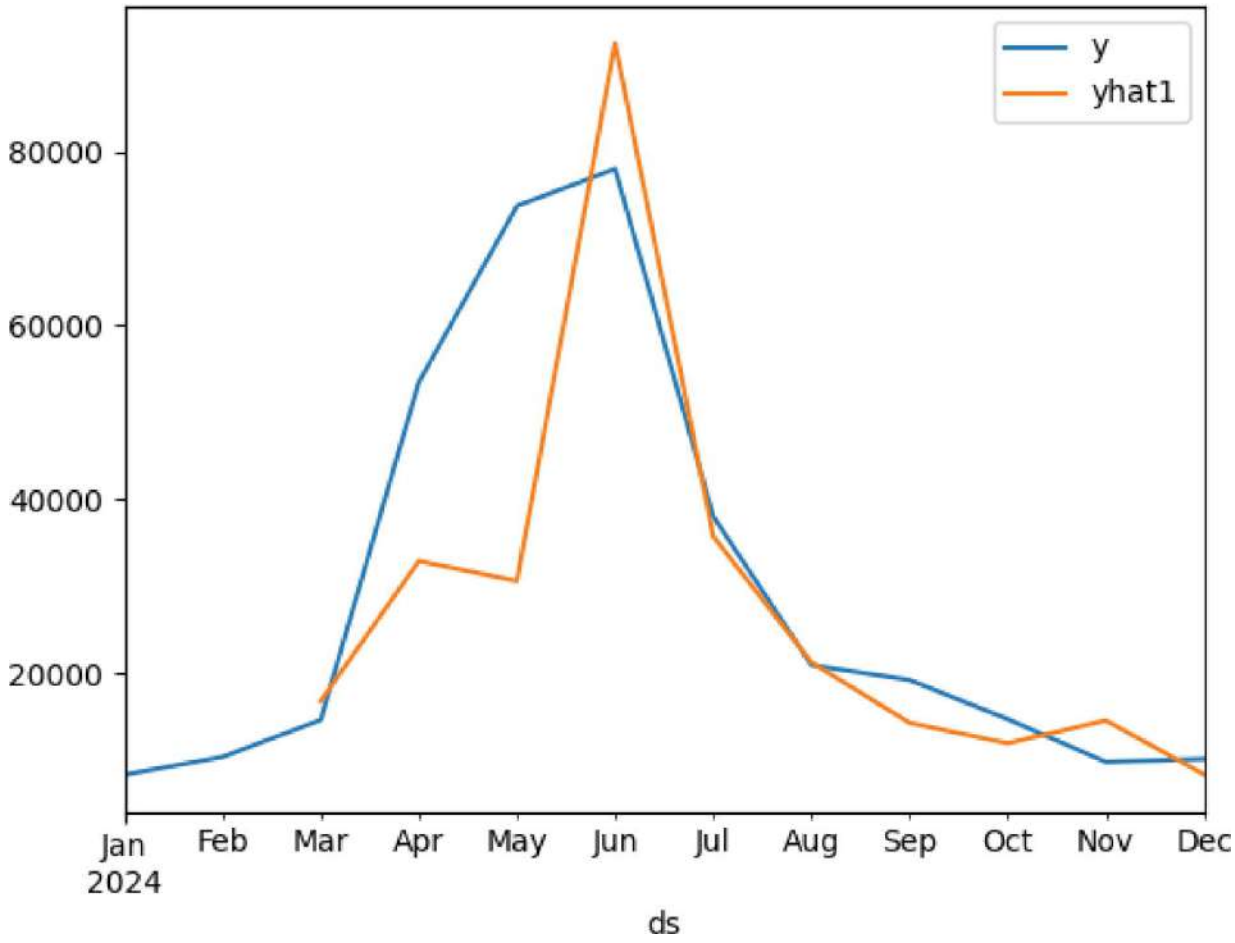


Figure 23-10 The final forecast

You can see in Figure 23-10 that the final forecast fits the peak much better than the one that was generated by the default model. We can conclude that the tuning was successful.

MLflow

We have used MLflow throughout this book to log model results. NeuralProphet also interacts easily with MLflow. You can use the code in Listing 23-11 to save the final model in MLflow.

```
from neuralprophet import save
import mlflow

with mlflow.start_run() as run:
```

```
        # Save the tuned parameters to MLFlow
        params = {
            'growth':'discontinuous',
            'n_changepoints':10,
            'seasonality_mode':'mutiplicative',
            'n_lags':1,
            'newer_samples_weight':20
        }
        mlflow.log_params(params)

        # Fit the model
        growth='discontinuous'
        n_changepoints=10
        seasonality_mode='multiplicative'
        n_lags=1
        newer_samples_weight=20

        m = NeuralProphet(
            growth=growth,
            n_changepoints=n_changepoints,
            yearly_seasonality=True,
            weekly_seasonality=False,
            daily_seasonality=False,
            seasonality_mode=seasonality_mode,
            n_lags=n_lags,
            loss_func='MAE',
            newer_samples_weight=newer_samples_weight,
            newer_samples_start=0.88,
            trainer_config={'logger': None}
        )

        metrics = m.fit(train)
        fcst = m.predict(test)
        mape = 1-
mean_absolute_percentage_error(fcst['y'].iloc[2:],
fcst['yhat1'].iloc[2:])

        # Save the test MAPE
        mlflow.log_metric("MAPE_test", value=mape)
```

```
        # Save the model
        model_path = "tuned-neuralprophet.np"
        save(m, model_path)

        # Log the model in MLflow
        mlflow.log_artifact(model_path, "tuned-
neuralprophet")
```

Listing 23-11 Integrate with MLflow

Once you have a saved model, you can check it in MLflow UI simply by running a notebook cell with '!mlflow ui', and go to the localhost URL that the commands give you. You'll see that the latest run is your NeuralProphet model, which will look like the screenshot in Figure 23-11.

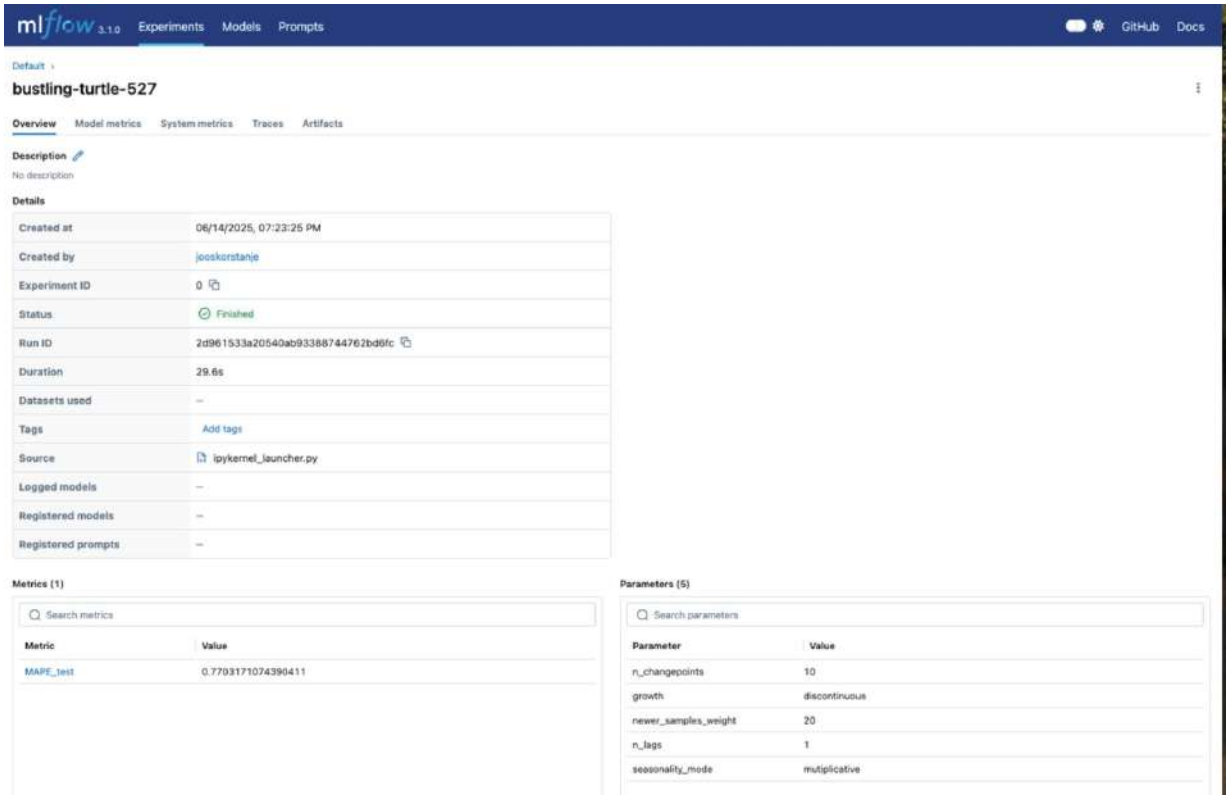


Figure 23-11 Inspecting the model through the MLflow user interface

---

Key Takeaways

- Facebook Prophet is an automated procedure for building forecasting models developed by Facebook, but it has faced strong criticism and is not really

maintained anymore.

- NeuralProphet is the replacement model. It was also initially built by Facebook. It is a more complex version of Prophet, which takes some of its predecessors' functionalities, mixed with PyTorch-based Neural Networks.
- Despite the added complexity, the package is still quite easy to use and has interesting functionalities, as demonstrated in the applied example using Wikipedia Page View Statistics.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_24

24. The DeepAR Model and AWS SageMaker AI

Joos Korstanje[1]
(1) Maisons Alfort, France

The **DeepAR** is a model developed by researchers at Amazon. DeepAR provides an interface for building time series models using a deep learning architecture based on RNNs. The advantage of using DeepAR is that it comes with an interface that is easier to use for model building when compared to Keras.

DeepAR can be considered a competitor of Facebook's Prophet, which you have seen in the previous chapter. Both those models try to deliver a simple-to-use model interface for models that are very complex under the hood. If you want to obtain good models with relatively little work, DeepAR and Prophet are definitely worth adding to your model benchmark.

About DeepAR

DeepAR forecasts univariate or multivariate time series using RNNs. The theoretical added value of DeepAR is that it fits a single model to all the time series at the same time. It is designed to benefit from correlations between multiple time series, and therefore, it is a great model for multivariate forecasting. Yet, it can also be applied to a single time series.

DeepAR models multiple types of seasonality variables. It also creates variables for the day of the month, day of the year, and other derived variables. As a user of the DeepAR model, we do not have much control over those variables and the inner workings of the algorithm. The model is relatively black box, aside from a few hyperparameters that we will come back to later.

Forecasting Heat Wave Page Views Using gluonts DeepAR Locally

Let's get started by building a forecast for Heat Wave Page view: the same dataset as we used for the Prophet model. To follow along with the current example, you'll need to download the data on Heat Waves from July 2015 to May 2025, in CSV format, which you can do through this link: https://pageviews.wmcloud.org/?project=en.wikipedia.org&platform=all-access&agent=user&redirects=0&start=2015-07&end=2025-05&pages=Heat_wave.

Once you have downloaded the data, you can use the code in Listing 24-1 to prepare the data. Note that this code drops the last five months of data in order to have a full year as the final year in the time series.

```
import pandas as pd
import matplotlib.pyplot as plt

y = pd.read_csv('heatwave-pageviews-20150701-20250531.csv')
y.columns = ['date', 'y']
y['date'] = pd.to_datetime(y['date'])
y = y[:-5]
```

Listing 24-1 Importing the data

Now, in this first example, we are going to run DeepAR locally. The gluonts package has a DeepAR implementation. However, this requires making some specific adaptations to the data format: we need to create something called a ListDataset in order for the model to work. The code in Listing 24-2 shows you how to convert your pandas dataframe into ListDatasets.

```
from gluonts.dataset.common import ListDataset
```

```
import pandas as pd

start = pd.Timestamp("01-07-2015", unit="h")
train_ds = ListDataset([{'target': y[:-12]['y'].values, 'start': start}],
freq='ME')
test_ds = ListDataset([{'target': y[-12:]['y'].values, 'start':
start}],freq='ME')
```

Listing 24-2 Convert the data to ListDatasets

With the data in the correct format, we can now move on to building a first DeepAR model with default settings. The code in Listing 24-3 shows you how to instantiate the DeepAREstimator and then train the model and save it as a predictor.

An important choice has to be made here: the gluonts package allows you to build on a PyTorch back end or on an MXNet back end. In this example, we are choosing the PyTorch approach. If you want to follow along, you can decide to pip install the gluonts as follows: `pip install gluonts[torch]`. This will automatically give you all the gluonts tools with Torch, which may save you some environment setup troubles.

```
from gluonts.torch import DeepAREstimator
import numpy as np
np.random.seed(7)

estimator = DeepAREstimator(
    prediction_length=12,
    context_length=12,
    freq='ME',
)

predictor = estimator.train(train_ds)
```

Listing 24-3 Fitting the default DeepAR model

This will take some time to train. Once the predictor is finished training, you can use the model to estimate a forecast on the test dataset. The code in Listing 24-4 does exactly this.

You can see that the predictions are computed as the 0.5 quantiles of the forecast. It is important to understand that the DeepAR generates numerous different future forecasts. In the current example, we are mainly interested in finding the most likely and most precise possible forecast. In this case, we need to take the median forecast, which is the same as the 0.5 quantile.

You could use the other quantiles for other use cases. For example, imagine that you have a business case in which you want to predict a "worst-case scenario": you could take the 0.9 (90%) quantile to obtain the highest 10% of forecasts and use this as a scenario that is a reasonable high estimate.

```
predictions_generator = predictor.predict(test_ds)

forecast = next(predictions_generator)

# Get the 50th percentile of the forecast (median)
predictions = forecast.quantile(0.5)
```

Listing 24-4 Estimating the forecast

Let’s now use the code in Listing 24-5 to do an evaluation of the model.

```
test_values = test_ds[0]['target']
fcst_values = forecast.samples.mean(axis=0)

import matplotlib.pyplot as plt
plt.plot(test_values)
plt.plot(fcst_values)
plt.legend(['test', 'forecast'])
```

```
plt.show()
```

Listing 24-5 Evaluating the prediction on the test set

Using this code, you will obtain the graph shown in Figure 24-1.

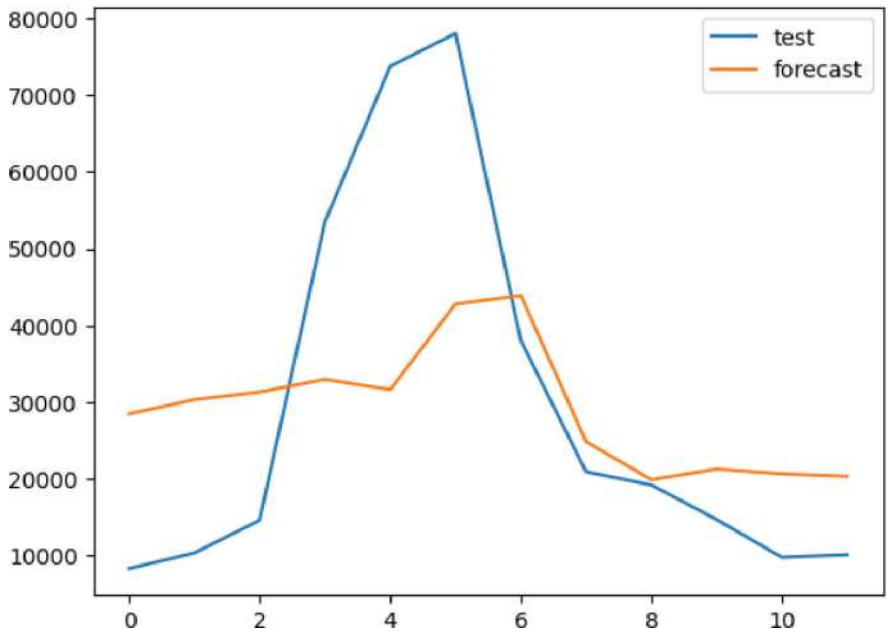


Figure 24-1 The forecast plot

You'll quickly see that the model has not been very performant at all. You can use the code in Listing 24-6 to confirm this with a performance metric.

```
from sklearn.metrics import mean_absolute_percentage_error
mape_metric = 1-mean_absolute_percentage_error(test_values, fcst_values)
print(mape_metric)
```

Listing 24-6 Evaluating the prediction on the test set

This code results in a performance score (1-MAPE) of **0.177:** quite disappointing. As a reminder, the default NeuralProphet was able to obtain a 1-MAPE of 0.69 on this dataset. This doesn't mean that DeepAR is a worse model than NeuralProphet, but it does show that DeepAR might experience difficulty fitting this particular dataset. As always, keep in mind the importance of benchmarking for model selection.

Tuning the DeepAR Model Using Ray HyperOptSearch

Let's now try and tune the gluonts DeepAR model to see whether we can improve the performance a little bit. In this chapter, we are going to explore Ray HyperOptSearch for tuning.

What Is Ray HyperOptSearch?

HyperOptSearch is a search algorithm wrapper in Ray Tune that leverages Hyperopt, a popular Python library for Bayesian optimization of hyperparameters. It's used to efficiently explore a hyperparameter space using Tree-structured Parzen Estimator (TPE)—a form of Bayesian optimization—rather than random or grid search, which can be inefficient when having a lot of hyperparameters to tune. Using HyperOptSearch through the Ray package allows you to benefit from its scalability and, therefore, faster execution (if your hardware allows it).

The code in Listing 24-7 shows you how to build a HyperOptSearch tuning to obtain the best model.

```
from gluonts.torch import DeepAREstimator
import matplotlib.pyplot as plt

from ray import tune
```

```
from ray.tune.search.hyperopt import HyperOptSearch

# Define the tune function
def tune_deepar(config):
    estimator = DeepAREstimator(
        prediction_length=12,
        context_length=config["context_length"],
        freq='ME',
        hidden_size=config["hidden_size"],
        num_layers=config["num_layers"],
        dropout_rate=config["dropout_rate"],
        batch_size=config["batch_size"]
    )

    predictor = estimator.train(train_ds)
    predictions_generator = predictor.predict(test_ds)
    forecast = next(predictions_generator)
    fcst_values = forecast.samples.mean(axis=0)
    mape_metric = 1 - mean_absolute_percentage_error(test_values,
fcst_values)

    tune.report({'mape':mape_metric})

# Define the search space
search_space = {
    "context_length": tune.choice([3, 6, 12]),
    "hidden_size": tune.choice([20, 30, 40]),
    "num_layers": tune.choice([2, 3]),
    "dropout_rate": tune.uniform(0.0, 0.1),
    "batch_size": tune.choice([16, 64, 128])
}

# Set up the HyperOptSearch
search_alg = HyperOptSearch(metric="mape", mode="max")

# Run the tuning process
analysis = tune.run(
    tune_deepar,
    config=search_space,
    num_samples=10,  # Number of trials to run
    search_alg=search_alg,
    verbose=1,
    resources_per_trial={"cpu": 2, "gpu": 0}
)
```

Listing 24-7 Hyperparameter tuning using Ray

The result of this tuning is a 1-MAPE of **0.73**. Still not great, but a least it got close to the 1-MAPE of **0.77** that was obtained by NeuralProphet. The best hyperparameters that were identified are

- context_length=3
- hidden_size=20
- num_layers=2
- dropout_rate=0.0745556
- batch_size=64

To build a forecasting plot of this model, we need to retrain it with these best hyperparameters. This is done using Listing 24-8.

```
# Use the best configuration to train the final model

estimator_tuned = DeepAREstimator(
    prediction_length=12,
    context_length=3,
    freq='ME',
    hidden_size=20,
    num_layers=2,
    dropout_rate=0.0745556,
    batch_size=64
)

predictor_tuned = estimator_tuned.train(train_ds)
predictions_generator_tuned = predictor_tuned.predict(test_ds)
forecast_tuned = next(predictions_generator_tuned)
fcst_values_tuned = forecast_tuned.samples.mean(axis=0)

plt.plot(test_values)
plt.plot(fcst_values_tuned)
plt.legend(['test', 'forecast (tuned)'])
plt.show()
```

Listing 24-8 Plot the best model

The resulting plot is shown in Figure 24-2. Unfortunately, the tuned forecast is still not great. As you have seen how to parametrize the HyperOptSearch, I'll leave it as an exercise to try and improve the score by running more tuning.

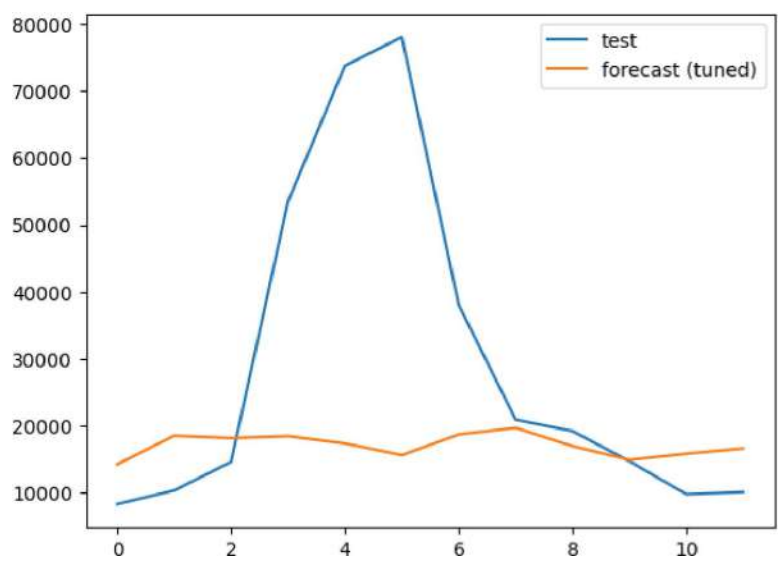


Figure 24-2 The forecast plot

DeepAR vs. AWS SageMaker AI

You have now seen how to work with the DeepAR model. As explained, the DeepAR model has been developed by Amazon. While gluonts is free and open source, it is an AWS project, and many of its models (like DeepAR) are also available in Amazon SageMaker as managed, scalable services.

While the main focus of this book is to showcase Advanced Forecasting with Python, it is also interesting to have a peek into Cloud environments and what they can add to the forecasting toolkit. After all, many companies are moving to the cloud, and it is important to adapt our machine learning workflows to specific cloud

environments. As the DeepAR is an Amazon model, we'll see how to build a DeepAR model on Amazon's cloud, called AWS (Amazon Web Services).

Forecasting Heat Wave Page Views Using DeepAR Inside an AWS SageMaker AI Training Job

Let's now start with the procedure of creating a DeepAR training job on AWS SageMaker AI. As you'll understand, it takes more than Python code to get a Cloud environment running. In this part of the chapter, we'll have to go back and forth between the platform, UI, and Python code to obtain a working model and obtain its forecast. If you are going to follow along, be careful: AWS is a paid service, so it charges money for any services used.

Step 1: Prepare the Data File

There are two issues with data preparation that we need to solve to go further on the AWS platform. A first problem is that the SageMaker AI training job requires only .parquet files or .json files. Also, they are extremely precise with the data formatting. Create a JSON file that looks like Listing 24-9 and call the file train.json.

```
{"start": "2015-07-01", "target": ["26379", "18584", "12529", "15343",
"13545", "11383", "14403", "13127", "17599", "20669", "20343", "16154",
"18474", "13745", "10949", "11659", "11657",.....]}
```

Listing 24-9 Plot the best model

Secondly, the DeepAR model on SageMaker AI only allows us to fit a dataset with at least 300 data points, which is not the case. There is not really a good solution for this, but you can try to "hack" the system by deduplicating your data. You can repeat your time series until you reach at least 300 data points, and the model may still be able to identify the seasonality (of course, it will be difficult to estimate the trend, as the trend is not correct due to the duplication).

Step 2: Upload the Data to S3

Once you have this file ready, we need to upload the data to an S3 bucket on AWS. For this step, you'll need an AWS account. It is easy to create if you don't have one yet. I'll not detail the procedure; you can find it on aws.amazon.com.

Now with your AWS account at the ready, you need to go to the S3 console on `https://s3.console.aws.amazon.com/s3/`. Once you're on there, you need to create a new bucket (choose any name of your choice). Inside the bucket, you need to create a folder called train. Inside this folder called train, you need to upload the file that you generated earlier.

Step 3: SageMaker AI

Your data is prepared and waiting for you in the S3 bucket. We can now start building the model. To do so, you need to go to Amazon SageMaker AI and go to the training jobs menu. Create a new training job as shown in Figure 24-3.

It will ask you to choose some settings. An important one is the IAM role. If you're not familiar with AWS at all, you just need to know that IAM is the AWS tool that manages roles and permissions. Your Amazon SageMaker AI training job is going to read the data from your S3 bucket and write outputs back to your bucket. To do so, the training job needs to have access to the S3 bucket. You need to give the training job a role that has these permissions.

Of course, you also need to select the algorithm that you want your training job to use. In the current example, select the DeepAR model, as that is the goal of this chapter. Feel free to play around with other models (pay attention, though: they may sometimes require you to format your data differently!)

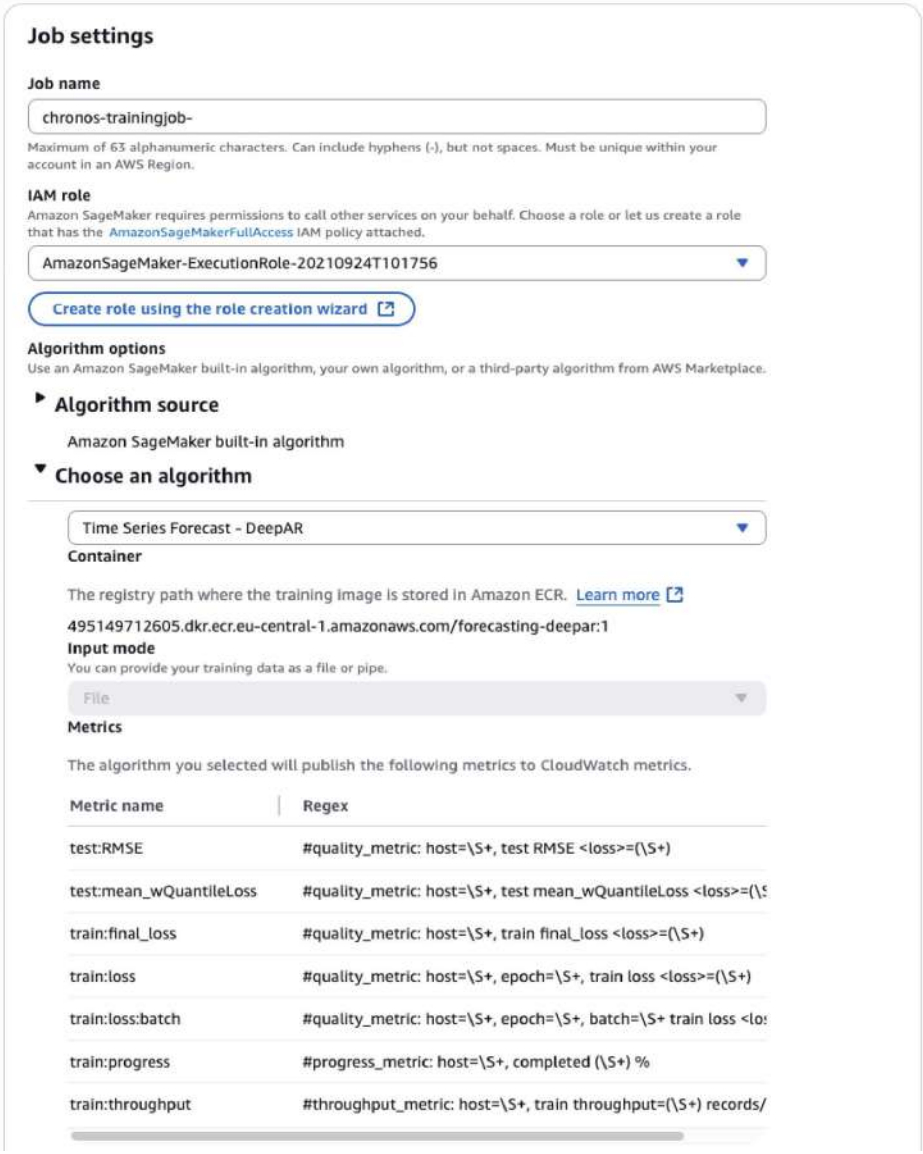


Figure 24-3 Build the training job

You also need to specify the path to your data. This is shown in Figure 24-4. For the input path, you need to refer to the train.json that you uploaded earlier. For the output, you can choose where you want the output to go, as long as the training job has IAM rights to write there.

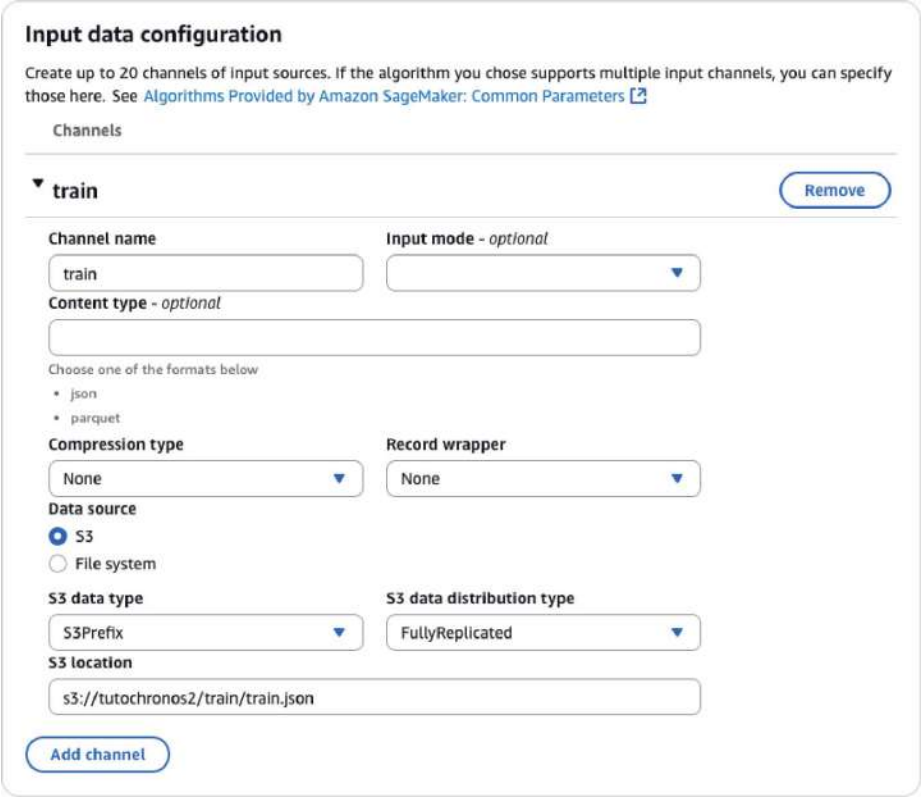


Figure 24-4 Input data configuration

A final setting that needs to be set is the hyperparameters for the model training. You can use the settings in Figure 24-5 or any other choice you see fit.

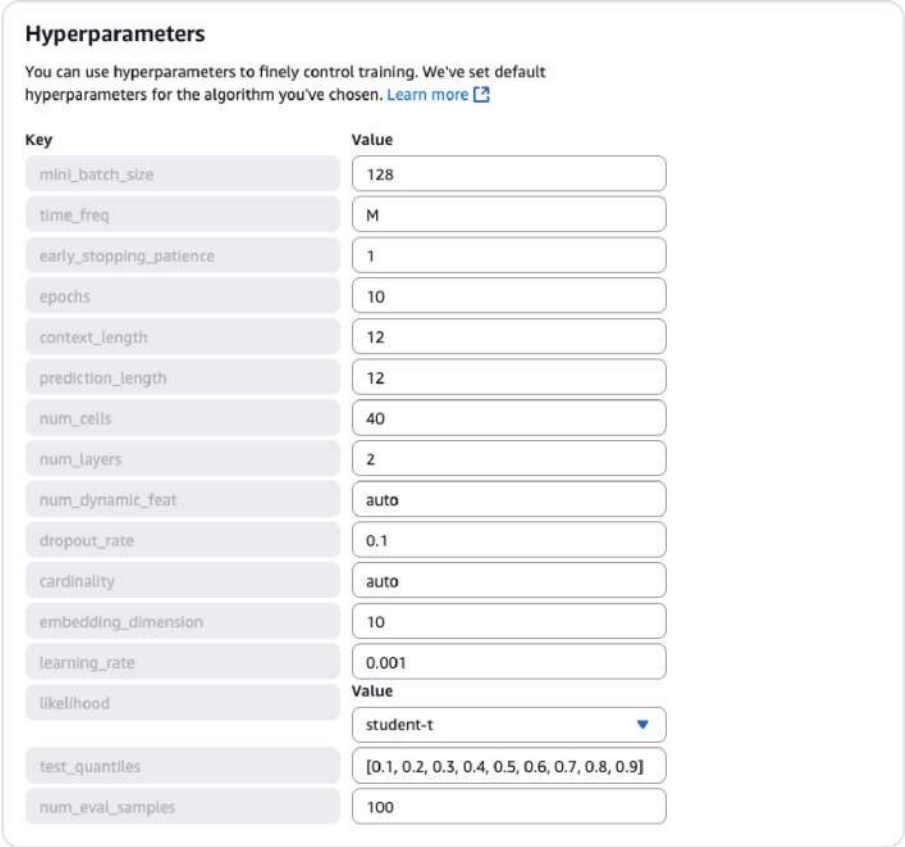


Figure 24-5 Hyperparameter configuration

With your data ready in S3 and all this model configuration done, you can now launch the training job. You now come back to the training job menu, where you can wait for the result. If everything is successful, you'll get something like Figure 24-6.

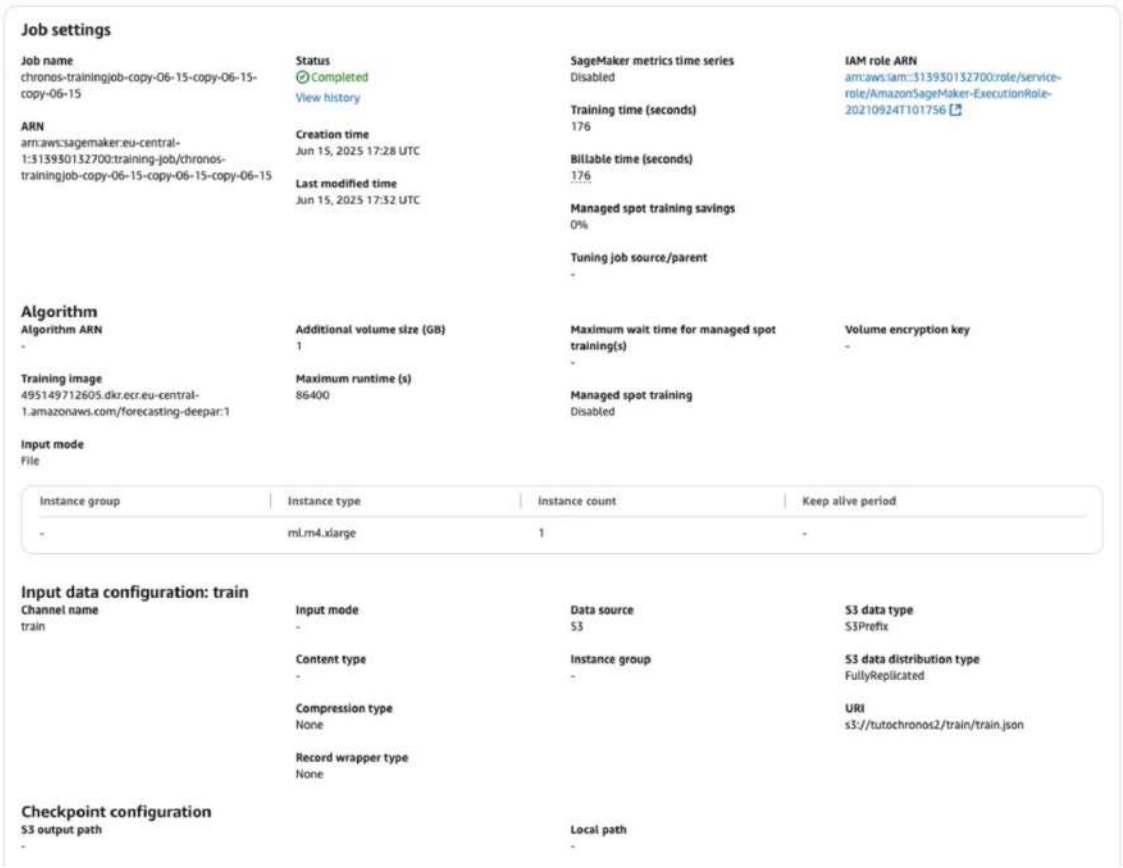


Figure 24-6 Model training successful

You have now successfully trained the model. However, for this to be useful, you'll need to be able to use the model for predictions. This requires you to deploy the model. You can click on Deploy the model, and you'll see the menu as shown in Figure 24-7.

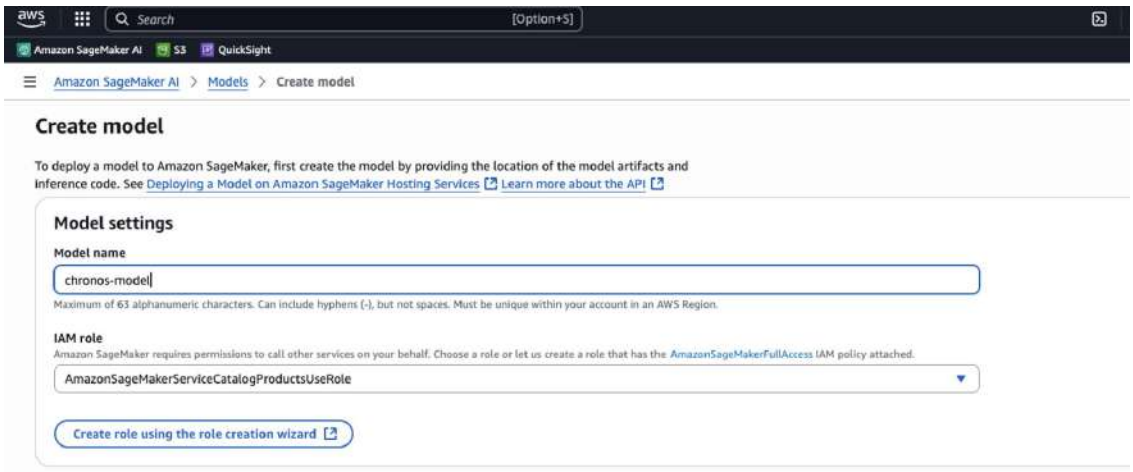


Figure 24-7 Deploy model

Once you've deployed, you should be able to see the model listed on the Models tab, as shown in Figure 24-8.

Figure 24-8 Model has been created

If you click on the model, you should be able to see some basic settings, as shown in Figure 24-9.

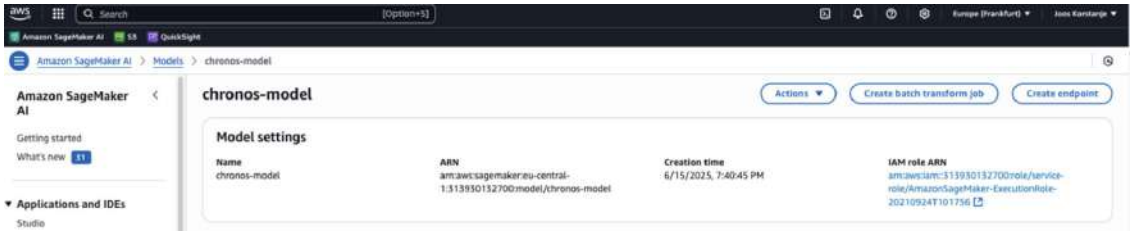


Figure 24-9 Model details

Now, you have stored the model for longer-term reuse. Great, but we still haven't generated any predictions yet. To generate predictions of a model stored in SageMaker, there are multiple possibilities. A good entry point is to create a batch transform job. When you are inside the model page, click Create batch transform job to get started. You can select the settings as shown in Figure 24-10.

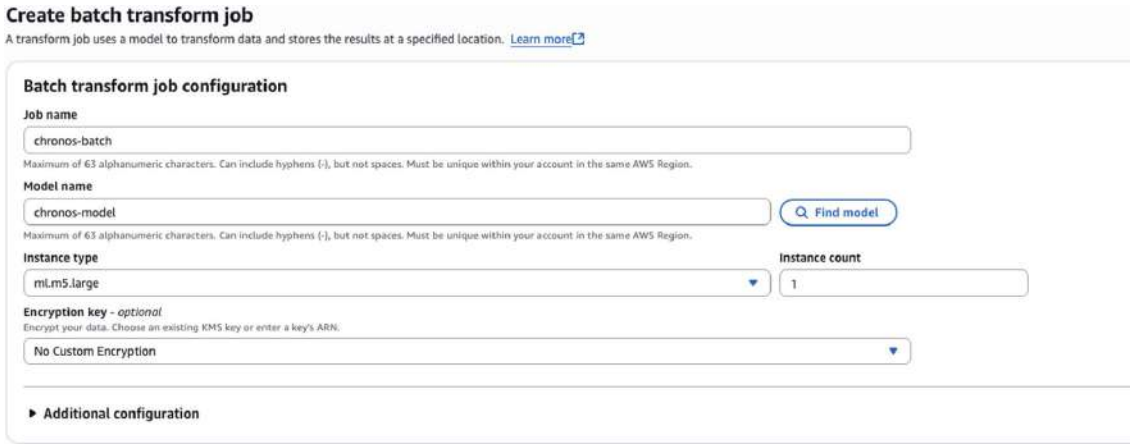


Figure 24-10 Create a batch transform job

You need to specify the path to your data and to your output. Then you can create the job. It will run upon creation. When your batch transform job finishes running, you can go to your S3 output folder to obtain your forecast in the train.json.out file.

When opening the train.json.out file (Listing 24-10), you'll be interested to look at the mean of the predictions and use them as predicted values. You'll see that the model has predicted 12 steps forward, as requested. You can use the mean or the quantiles, as was the case with the local DeepAR model.

```
{
"mean":
      [31665.51171875,25284.865234375,22499.8984375,18628. 939453125,16683.85:
"quantiles":
      {"0.1":
[25211.859375,16905.78125,16345.796875,12894.548828125,10356.4375,8333.309570:
[27768.708984375,20660.244140625,19394.15625,15146.17578125,12124.484375,1138:
[29165.923828125,21993.443359375,20468.2421875,16221.005859375,14077.48046875,
[30481.875,23223.5703125,21829.533203125,16890.599609375,15437.232421875,1387:
}
```

Listing 24-10 Truncated part of the output file

You have now successfully created an S3 bucket with training data, trained and deployed a DeepAR model through a training job on AWS SageMaker AI, as well as invoked this model through a batch transform job.

Very Important If you have followed along, delete ALL your resources at the end, as they WILL incur important costs.

Key Takeaways

- DeepAR is a model built by Amazon that builds deep forecasts based on RNNs.
- The gluonts package implements DeepAR in the DeepAREstimator.
- Amazon SageMaker AI is the machine learning tool of the AWS platform.
- The DeepAR model also exists as an implementation on Amazon SageMaker AI. Training jobs create models that you can then deploy. Batch transform jobs call existing models to score data.
- You can use S3 to store data and results, and it will interact seamlessly with SageMaker AI.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_25

25. Uber's Orbit Model

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, we are going to look at the Orbit model, developed by Uber. Just like Amazon and Facebook, Uber also has important use cases for forecasting, and they built Orbit as an internal model. They made the decision to share it as an open source model.

The model is relatively similar to Prophet and DeepAR: it is a black box model, but with a user-friendly interface layer. Something that differentiates Orbit from Prophet and DeepAR is that it has as a goal to provide high interpretability and that it accepts that this comes at a cost of moderate scalability. Orbit is the better choice for smaller datasets, whereas DeepAR may be the right choice for larger datasets.

About Orbit

Let's discuss what's underneath the Orbit model. For this, it is important to remember what has been discussed in the earlier chapter on Bayesian modeling. Under the hood, Orbit is also a Bayesian modeling package, just wrapped in a user-friendly layer. The Bayesian forecasting models that exist inside Orbit are Exponential Smoothing, Damped Local Trend (DLT), and Local-Global Trend (LGT).

The models in Orbit use Bayesian inference to estimate model parameters. The package also provides time series decomposition into three categories: trend, seasonality, and events. The Bayesian inference, as you have seen before, will generate a distribution of forecasts, which is great for all sorts of analysis purposes. However, it will be slower than other models.

Forecasting Heat Wave Page Views Using Orbit's Local-Global Trend Model

The Orbit package provides an interface to do Bayesian forecasting models. As this theory has already been covered in an earlier chapter of the book, let's dive right into the Python example and see how easy-to-use this package really is. We'll continue the benchmark with NeuralProphet and DeepAR, so we'll have to work with the same data.

As a reminder, we use the number of page views of the Wikipedia page on Heat Waves to try and predict the number of monthly heat wave page views for the entire year of the time series. There is a trend and a seasonality present in the data.

You can download the data on Heat Waves from July 2015 to May 2025, in CSV format, which you can do through this link: https://pageviews.wmcloud.org/?project=en.wikipedia.org&platform=all-access&agent=user&redirects=0&start=2015-07&end=2025-05&pages=Heat_wave.

Once you have downloaded the data, you can use the code in Listing 25-1 to prepare the data. Note that this code drops the last five months of data in order to have a full year as the final year in the time series.

```
import pandas as pd
import matplotlib.pyplot as plt

y = pd.read_csv('heatwave-pageviews-20150701-20250531.csv')
y.columns = ['date', 'y']
y['date'] = pd.to_datetime(y['date'])
y = y[:-5]
```

Listing 25-1 Importing the data

An example of what the data looks like is shown in Figure 25-1.

	date	y
0	2015-07-01	26379
1	2015-08-01	18584
2	2015-09-01	12529
3	2015-10-01	15343
4	2015-11-01	13545

Figure 25-1 The data

Before moving on to model building, we need to create the train-test set using Listing 25-2. This is the same train-test split as the one applied in previous chapters, which will allow for a benchmark comparison with NeuralProphet and DeepAR.

```
train = y.iloc[:-12]
test = y.iloc[-12:]
```

Listing 25-2 Train-test split

Let's now fit a Local-Global Trend model. This Bayesian model is characterized by two trends: a local one and a global one combined into one model. Together with the Damped

Local Trend model, these are the main models of Orbit. The model is instantiated using Listing 25-3.

```
from orbit.models.lgt import LGT

lgt = LGT(
    response_col='y',
    date_col='date',
    seasonality=12,
    seed=123,
)
```

Listing 25-3 Instantiate the default LGT model

You can then fit the model using Listing 25-4.

```
lgt.fit(df=train)
```

Listing 25-4 Fit the model

The code in Listing 25-5 shows you how to use this trained model to forecast on the test set.

```
fcst = lgt.predict(df=test)
fcst
```

Listing 25-5 Predict the LGT model

The predict function generates not only the forecasted values but a dataframe that also contains the 5 percentile forecast and the 95th percentile forecast. As explained earlier, these are the forecasts related to Bayesian outputs. The full forecast dataset is shown in Figure 25-2.

	date	prediction_5	prediction	prediction_95
0	2024-01-01	4326.002904	12499.586343	19796.141890
1	2024-02-01	4573.943522	11792.212060	19249.531257
2	2024-03-01	7738.111719	14913.255658	22350.213258
3	2024-04-01	10507.211610	19484.048125	25882.062643
4	2024-05-01	10644.578766	18176.888196	25502.924607
5	2024-06-01	26462.380776	35486.991264	46433.027468
6	2024-07-01	18900.322105	28163.131027	37967.431046
7	2024-08-01	4644.674841	14313.839618	20820.483025
8	2024-09-01	-1940.368283	8556.604367	16532.990941
9	2024-10-01	-2512.178028	6736.167238	16753.110562
10	2024-11-01	-3663.750736	6463.633891	17752.677935
11	2024-12-01	-3905.917673	5908.170281	14731.899423

Figure 25-2 The forecast dataset

We can now use this forecast to evaluate the model. The code in Listing 25-6 computes the 1-MAPE forecasting metric.

```
actuals = list(test['y'].reset_index(drop=True))
fcst = fcst['prediction']

from sklearn.metrics import mean_absolute_percentage_error
metric = 1 - mean_absolute_percentage_error(actuals, fcst)
print(metric)
```

Listing 25-6 Evaluate the model

You'll see that the model obtains a 1-MAPE score of **0.58** (Table 25-1).

Table 25-1 The benchmark

Model	Version	1-MAPE
DeepAR	Default	0.177

Model	Version	1-MAPE
DeepAR	Tuned	0.73
NeuralProphet	Default	0.69
NeuralProphet	Tuned	0.77
Orbit LGT	Default	0.58

You can see what this forecast looks like in practice by creating a plot of the actual test data against the forecast on the test data. You can use the code in Listing 25-7 to generate the plot.

```
plt.plot(actuals)
plt.plot(fcst)
plt.legend(['test', 'forecast'])
plt.show()
```

Listing 25-7 Plot the graph

This code will generate the plot shown in Figure 25-3.

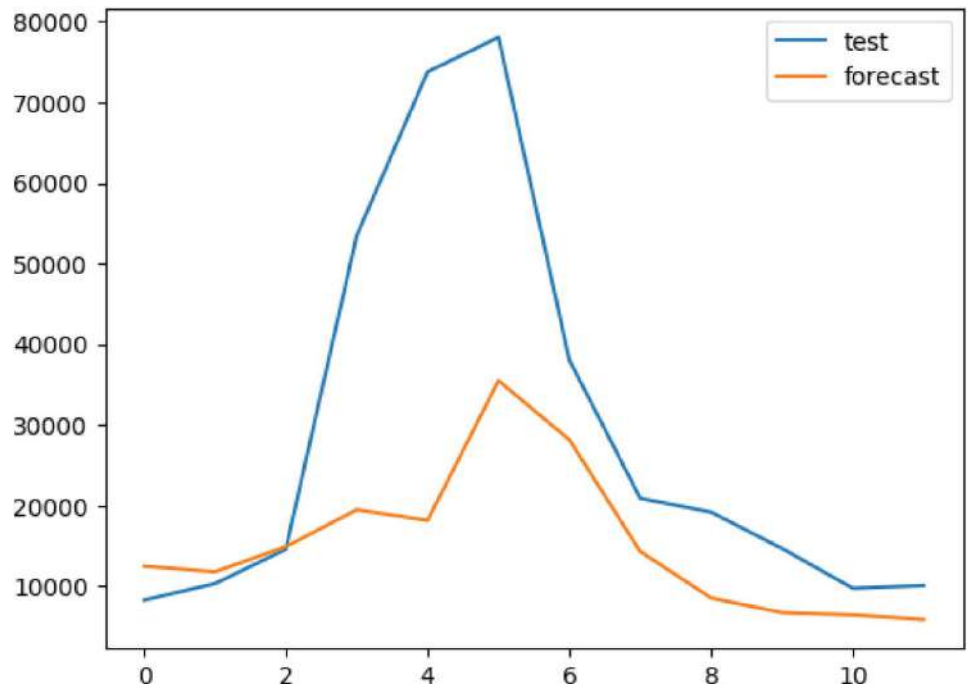


Figure 25-3 The forecast plot

As you can see, the plot shows that the model has been able to understand the seasonality in the model (low points are where they should be; the peak is also where it should be).

Tuning Orbit's GLT

As a next step, we are going to tune this model. Two hyperparameters that can be tuned are

- seasonality_sm_input: A larger value puts more weight on the current seasonality.
- level_sm_input: A larger value puts more weight on the current level.

You can use the code in Listing 25-8 to run the Grid Search.

```
results = []
for seasonality_sm_input in [0.1, 0.25, 0.5, 0.75, 0.9]:
    for level_sm_input in [0.1, 0.25, 0.5, 0.75, 0.9]:

        lgt = LGT(
            response_col='y',
            date_col='date',
            seasonality=12,
            seed=123,
            seasonality_sm_input=seasonality_sm_input,
            level_sm_input=level_sm_input
        )

        lgt.fit(df=train)

        fcst = lgt.predict(df=test)

        actuals = list(test['y'].reset_index(drop=True))
        fcst = fcst['prediction']

        metric = 1 - mean_absolute_percentage_error(actuals,
fcst)
        results.append([seasonality_sm_input, level_sm_input,
metric])

results_df=pd.DataFrame(results, columns =
['seasonality_sm_input', 'level_sm_input', 'metric'])
results_df.sort_values('metric', ascending=False).head()
```

Listing 25-8 Tuned LGT

The result of the grid search is shown in Figure 25-4.

	seasonality_sm_input	level_sm_input	metric
20	0.90	0.1	0.715469
15	0.75	0.1	0.702552
10	0.50	0.1	0.693794
5	0.25	0.1	0.653431
0	0.10	0.1	0.606013

Figure 25-4 The grid search results

As the table in Figure 25-4 shows, the best model is the one with seasonality_sm_input 0.90 and level_sm_input 0.1, which obtains a 1-MAPE metric of **0.715.** Table 25-2 is updated with this result.

Table 25-2 The benchmark

Model	Version	1-MAPE
DeepAR	Default	0.177
DeepAR	Tuned	0.73
NeuralProphet	Default	0.69
NeuralProphet	Tuned	0.77
Orbit LGT	Default	0.58
Orbit LGT	Tuned	0.715

You can also use the code in Listing 25-9 to generate the forecasting plot with the new model.

```
lgt = LGT(
    response_col='y',
    date_col='date',
    seasonality=12,
    seed=123,
    seasonality_sm_input=0.9,
    level_sm_input=0.1
)

lgt.fit(df=train)

fcst = lgt.predict(df=test)
```

```
actuals = list(test['y'].reset_index(drop=True))
fcst = fcst['prediction']

plt.plot(actuals)
plt.plot(fcst)
plt.legend(['test', 'forecast'])
plt.show()
```

Listing 25-9 The best Local-Global Trend model

The graph that results from Listing 25-9 is shown in Figure 25-5.

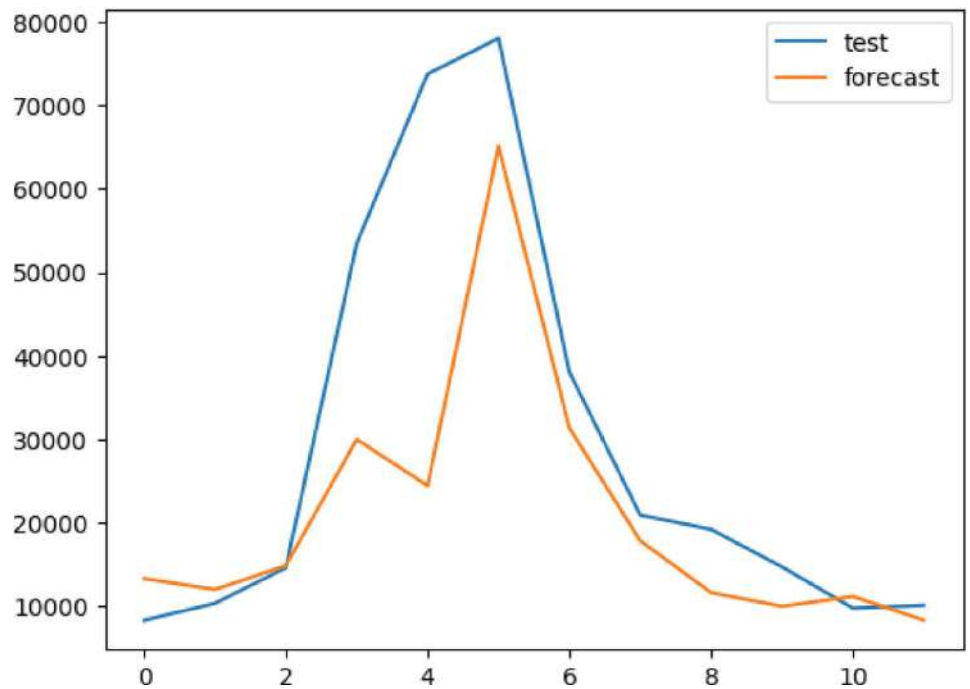


Figure 25-5 The tuned forecast plot

As you can see from this graph, the tuned model has been able to find the peak much better than the default model.

Forecasting Heat Wave Page Views Using Orbit's Damped Local Trend Model

As mentioned in the introduction, the second important model in Orbit is the Damped Local Trend. You can use the code in Listing 25-10 to instantiate a default DLT model.

```
from orbit.models.dlt import DLT

dlt = DLT(
    response_col='y',
    date_col='date',
```

```
    seasonality=12,
    seed=123,
)
```

Listing 25-10 Default DLT

Now, the model can be trained using Listing 25-11.

```
dlt.fit(df=train)
```

Listing 25-11 Train the DLT

The default DLT model has now been trained, and we can make a forecast on the test set using the code in Listing 25-12.

```
fcst = dlt.predict(df=test)
fcst
```

Listing 25-12 Forecast using the DLT

You'll obtain a forecast dataset as shown in Figure 25-6.

	date	prediction_5	prediction	prediction_95
0	2024-01-01	7815.762657	15052.022381	21772.561500
1	2024-02-01	5519.051571	12791.435551	21599.806277
2	2024-03-01	6088.620644	15411.510465	22363.441987
3	2024-04-01	11662.315827	21331.884434	31208.695058
4	2024-05-01	11508.224573	20474.202659	31570.946658
5	2024-06-01	28054.495459	39997.158501	53384.027283
6	2024-07-01	20201.521979	31302.333500	40708.644650
7	2024-08-01	5878.731176	15842.307256	26171.539220
8	2024-09-01	-2249.241738	9985.168267	20690.051924
9	2024-10-01	-1853.646987	8668.114178	18729.164324
10	2024-11-01	-1036.057591	8982.286551	17610.234536
11	2024-12-01	-6159.065925	7673.336508	16293.265685

Figure 25-6 The forecast dataset

To evaluate this default model, we can compute the 1-MAPE forecasting metric using Listing 25-13.

```
actuals = list(test['y'].reset_index(drop=True))
fcst = fcst['prediction']

metric = 1 - mean_absolute_percentage_error(actuals, fcst)
print(metric)
```

Listing 25-13 Evaluate the DLT

The obtained score is **0.62**. Table 25-3 has been updated with this result.

Table 25-3 The benchmark

Model	Version	1-MAPE

Model	Version	1-MAPE
DeepAR	Default	0.177
DeepAR	Tuned	0.73
NeuralProphet	Default	0.69
NeuralProphet	Tuned	0.77
Orbit LGT	Default	0.58
Orbit LGT	Tuned	0.715
Orbit DLT	Default	0.62

Listing 25-14 shows how to generate the forecasting plot with the default DLT model.

```
plt.plot(actuals)
plt.plot(fcst)
plt.legend(['test', 'forecast'])
plt.show()
```

Listing 25-14 Plot the forecast

The generated plot is shown in Figure 25-7.

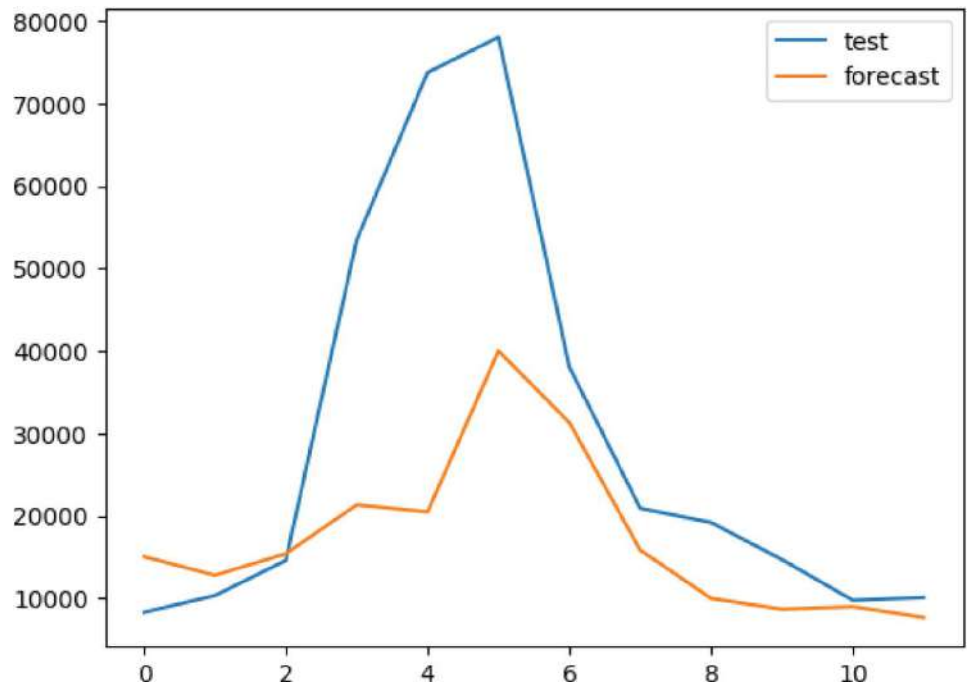


Figure 25-7 The forecast plot

Like the default LGT model, the seasonality is well fit, but the height of the peak is missing.

Tuning Orbit's DLT

We'll now do a hyperparameter tuning on the DLT to see if the model can improve. The code for this tuning is shown in Listing 25-15. In DLT, we can tune two additional hyperparameters compared to GLT:

- damped_factor: A smaller value further dampens the previous global trend value.
- global_trend_option: Transformation function for the shape of the forecasted global trend.

```
results = []
for seasonality_sm_input in [0.1, 0.5, 0.9]:
    for level_sm_input in [0.1,  0.5, 0.9]:
        for damped_factor in [0.1,  0.5, 0.9]:
            for global_trend_option in ['linear', 'loglinear',
'logistic', 'flat']:

                dlt = DLT(
                    response_col='y',
                    date_col='date',
                    seasonality=12,
                    seed=123,
                    seasonality_sm_input=seasonality_sm_input,
                    level_sm_input=level_sm_input,
                    damped_factor=damped_factor,
                    global_trend_option=global_trend_option
                )

                dlt.fit(df=train)

                fcst = dlt.predict(df=test)

                actuals =
list(test['y'].reset_index(drop=True))
                fcst = fcst['prediction']

                metric = 1 -
mean_absolute_percentage_error(actuals, fcst)
                results.append([seasonality_sm_input,
level_sm_input, damped_factor, global_trend_option, metric])

results_df=pd.DataFrame(results, columns =
['seasonality_sm_input', 'level_sm_input', 'damped_factor',
'global_trend_option', 'metric'])
results_df.sort_values('metric', ascending=False).head()
```

Listing 25-15 Tune the DLT model

You'll obtain the result as shown in Figure 25-8.

	seasonality_sm_input	level_sm_input	damped_factor	global_trend_option	metric
75	0.9	0.1	0.1	flat	0.714176
73	0.9	0.1	0.1	loglinear	0.708693
79	0.9	0.1	0.5	flat	0.705170
72	0.9	0.1	0.1	linear	0.704368
77	0.9	0.1	0.5	loglinear	0.704148

Figure 25-8 The grid search results

As you can see in Figure 25-8, the best DLT model is the one with

- seasonality_sm_input=0.9
- level_sm_input=0.1
- damped_factor=0.1
- global_trend_option='flat'

The obtained 1-MAPE score is **0.71**. This is updated in Table 25-4.

Table 25-4 The benchmark

Model	Version	1-MAPE
DeepAR	Default	0.177
DeepAR	Tuned	0.73
NeuralProphet	Default	0.69
NeuralProphet	Tuned	0.77
Orbit LGT	Default	0.58
Orbit LGT	Tuned	0.715
Orbit DLT	Default	0.62
Orbit DLT	Tuned	0.71

We can now conclude that the best-tuned versions of Orbit LGT and Orbit DLT are not able to beat the tuned versions of DeepAR or Prophet. However, it is also interesting to note that the default scores are not that bad, compared to, for example, the default DeepAR. Even though the Orbit models are not winning this benchmark, you can conclude that Orbit models could be a safe bet for building a quick and easy baseline model for smaller datasets. The best-tuned model is built using the code in Listing 25-16.

```
dlt = DLT(
    response_col='y',
    date_col='date',
    seasonality=12,
    seed=123,
    seasonality_sm_input=0.9,
    level_sm_input=0.1,
```

```
    damped_factor=0.1,
    global_trend_option='flat'
)

dlt.fit(df=train)

fcst = lgt.predict(df=test)

actuals = list(test['y'].reset_index(drop=True))
fcst = fcst['prediction']

plt.plot(actuals)
plt.plot(fcst)
plt.legend(['test', 'forecast'])
plt.show()
```

Listing 25-16 Best-tuned DLT

The plot of this final forecast is shown in Figure 25-9. As you can see, it has been able to fit the seasonality quite well. It is also not too far from the peak in May. However, the forecasted peak is too spikey. The spike in actual test data already starts in March, whereas the forecast is still low in March and April.

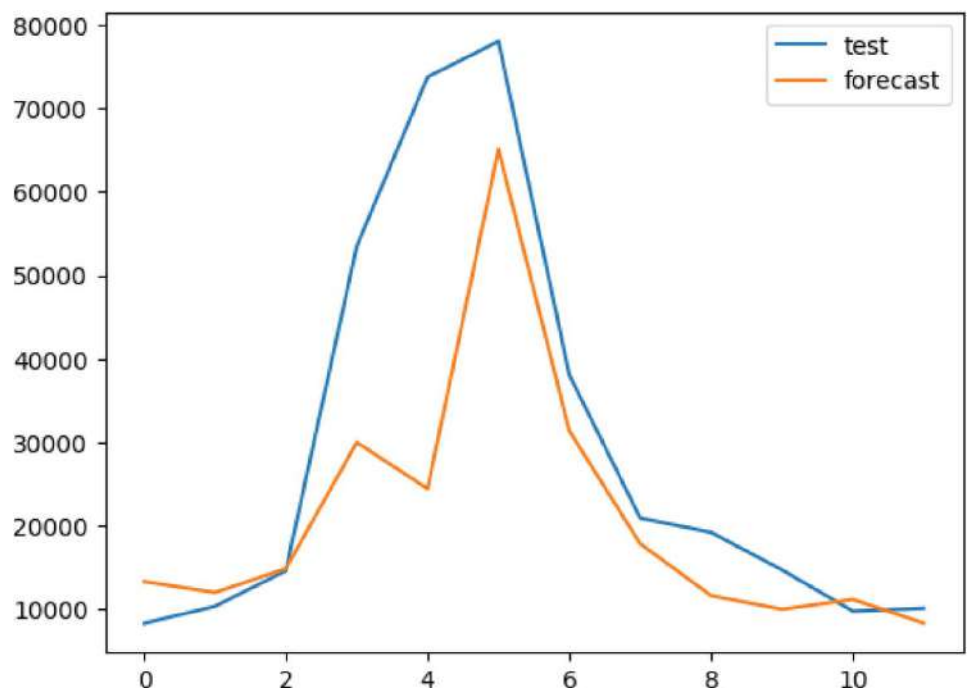


Figure 25-9 The tuned forecast plot

Key Takeaways

- Orbit is a package built by Uber that builds forecasts based on Bayesian models.

- Examples of implemented models are Damped Local Trend and Global Local Trend.
- Multiple hyperparameters can be tuned, including
  - seasonality_sm_input
  - level_sm_input
  - damped_factor
  - global_trend_option

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_26

26. AutoML with Microsoft Azure

Joos Korstanje[1]
(1) Maisons Alfort, France

Throughout this book, you have already discovered numerous advanced forecasting models, all with Python implementations. The first parts that we covered were univariate time series models, supervised modeling, Bayesian models, neural networks, and deep learning. These parts all represent models by theoretical and mathematical groups.

After the Neural Network chapters, we went into models that were presented not by a theoretical approach but rather by the provider of the models. For example, we have already seen models provided by Facebook/Meta, Amazon, and Uber.

There is a distinction to be made between models and packages that have been open sourced and models that are available only on a paid cloud. As an example, some algorithms provided by tech giants do have a clear mathematical foundation, with a paper explaining how the model works. These models are often based on either Bayesian models or deep neural nets.

At the same time, they are partly black box, as they make a lot of automated decisions under the hood. Before the existence of these sorts of ML packages, these decisions would have to be made by machine learning practitioners, either through knowledge or through testing.

Cloud Computing

The recent years, cloud computing has become very popular. Many companies used to manage their own servers and IT systems, but nowadays,

companies more and more often decide to let a cloud provider manage this physical part of their IT systems for them.

As an example of this, you have discovered how to run the DeepAR algorithm on Amazon's AWS Cloud in the previous chapter. The fact that you can easily set up an API on a remote server and allow people to call your model to make predictions is a strong argument for using a cloud provider.

From Cloud-Based Maths to AutoML

Many Cloud computing providers compete to provide the best services to attract machine learning users. One topic on which there is much competition is data science and AI services. With all this competition ongoing, cloud providers are moving in the direction of providing performant and easy-to-use models that even inexperienced users could set up.

Now, there is an entire family of modeling techniques that we have not covered yet: AutoML and full black box models. As you have seen throughout this book, it takes a good background in model evaluation and benchmarking to be able to provide a model that will be able to accurately forecast on a test set. Therefore, we could wonder if it is really a necessity to have one-click machine learning models for forecasting use cases.

Today's reality is that many cloud providers do provide black box AutoML tools. Whether you plan to use black box models or more traditional models depends strongly on your use case and other elements of the decision (we will come back to this in Chapter 29).

For now, as a forecasting practitioner, it will be good to have at least a basic understanding of cloud models and also do some experimentation with AutoML. We will take this chapter to build an AutoML model on Microsoft's Azure Cloud. Going into model deployment strategies and corresponding architectures would be outside the scope of this book, but feel free to use the model that we'll build in this chapter and study model deployments and hosting using the Azure documentation.

Microsoft Azure Cloud

Azure Cloud is the cloud service provided by Microsoft. As Microsoft is very strong in enterprise software, Microsoft already has a foot in the door in many large companies. It is therefore only logical that Microsoft has tried to launch a cloud program to compete with Amazon's AWS. As Microsoft Azure has become an important player in the cloud domain, we'll build an introductory forecasting use case in this chapter.

Forecasting Heat Wave Page Views with Microsoft Azure Cloud AutoML

To get started with Azure (portal.azure.com), you first need to create an account and set up a subscription. As this is well documented on the Azure website, I'll skip these details here. If you are going to follow along in this chapter, be careful: you will be charged money by Microsoft for all resources used.

Step 1: Create an Azure Machine Learning Studio

If you have succeeded in setting up your account and your subscription, you can go directly to ml.azure.com to set up an Azure Machine Learning Studio. When you go to Azure Machine Learning Studio, the first step in the menu is to create a workspace. This is shown in Figure 26-1. A workspace regroups all work and resources, which is very practical.

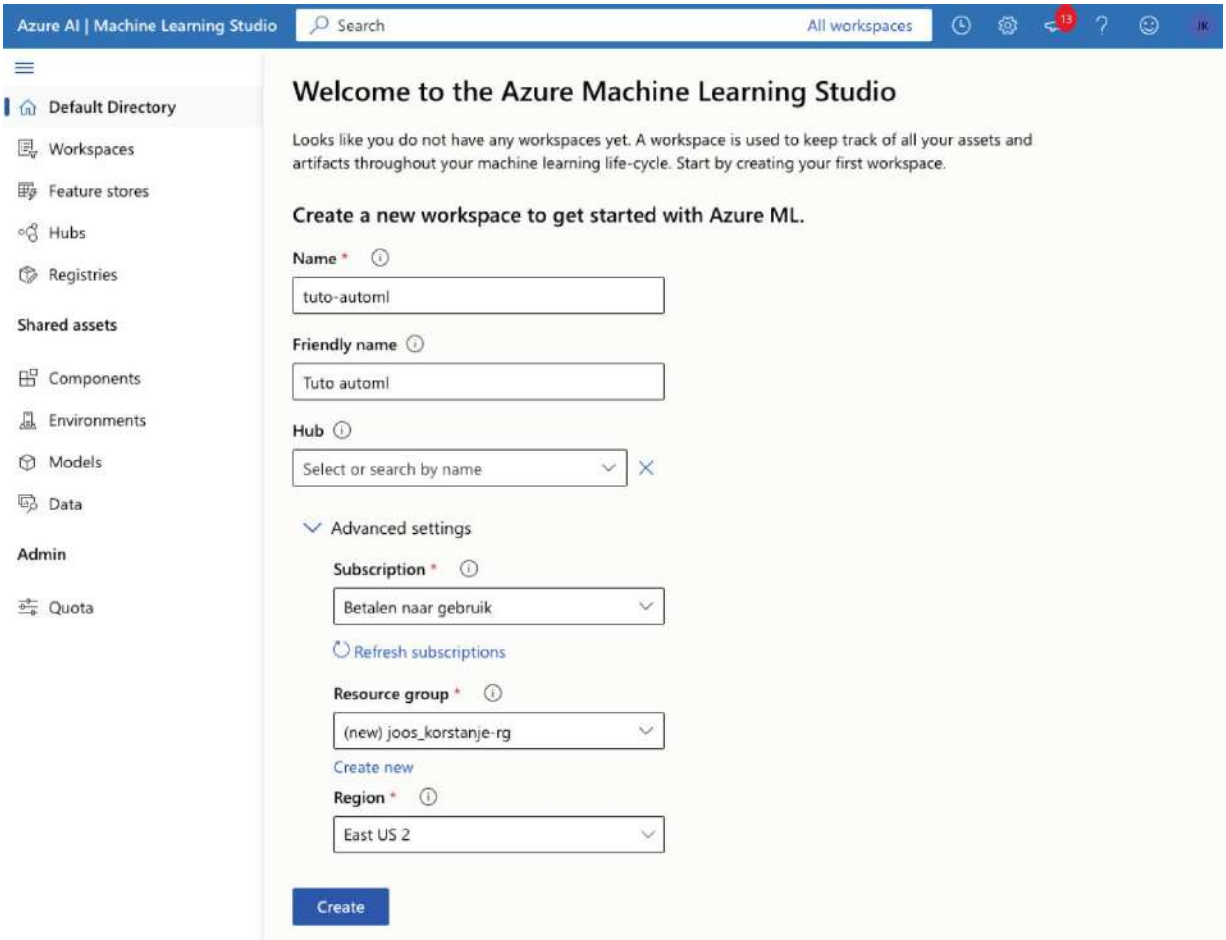


Figure 26-1 Create a workspace

Fill in the information in the form and click Create. You need to wait for a few minutes, and when your workspace is created, you click on it to enter it.

Step 2: Create an Automated ML Run

Inside your workspace, on the left, go to Automated ML. Click New Automated ML job, as shown in Figure 26-2.

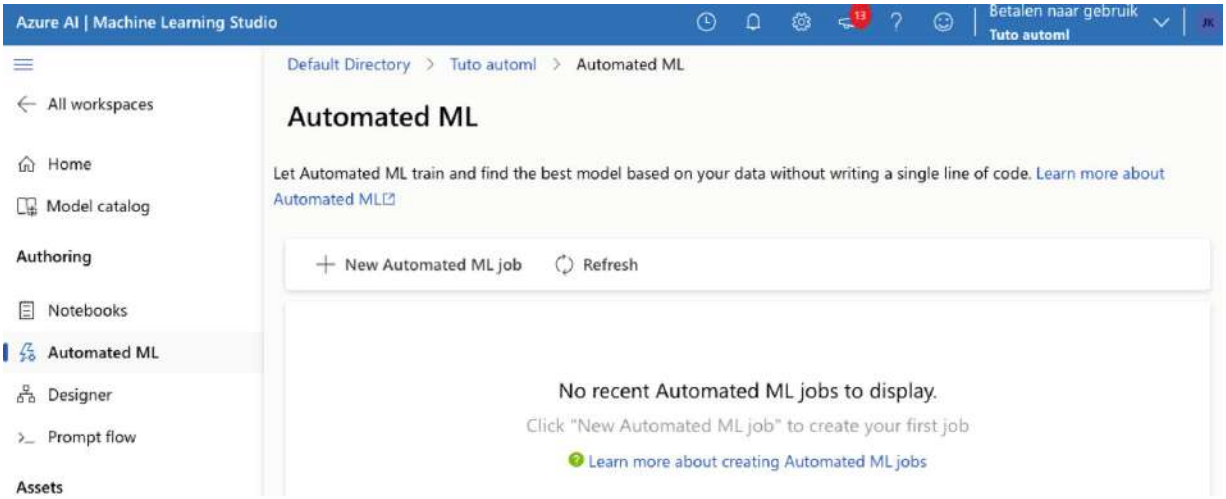


Figure 26-2 Create an Automated ML Run

You'll then need to configure your job's basic settings. You can copy the settings from Figure 26-3.

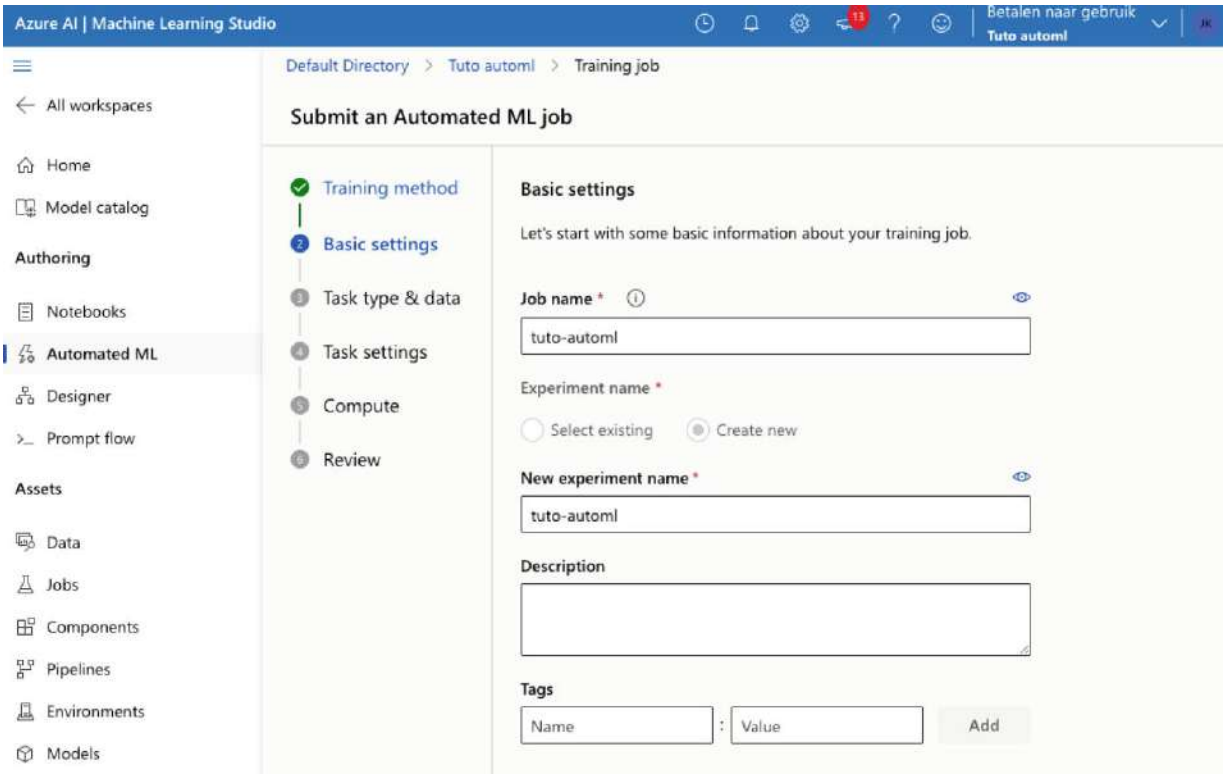


Figure 26-3 Create an Automated ML Run

When you click Next, you'll see the next step of submitting an Automated ML job, which is called Task type & data. For task type, select

time series forecasting (Figure 26-4).

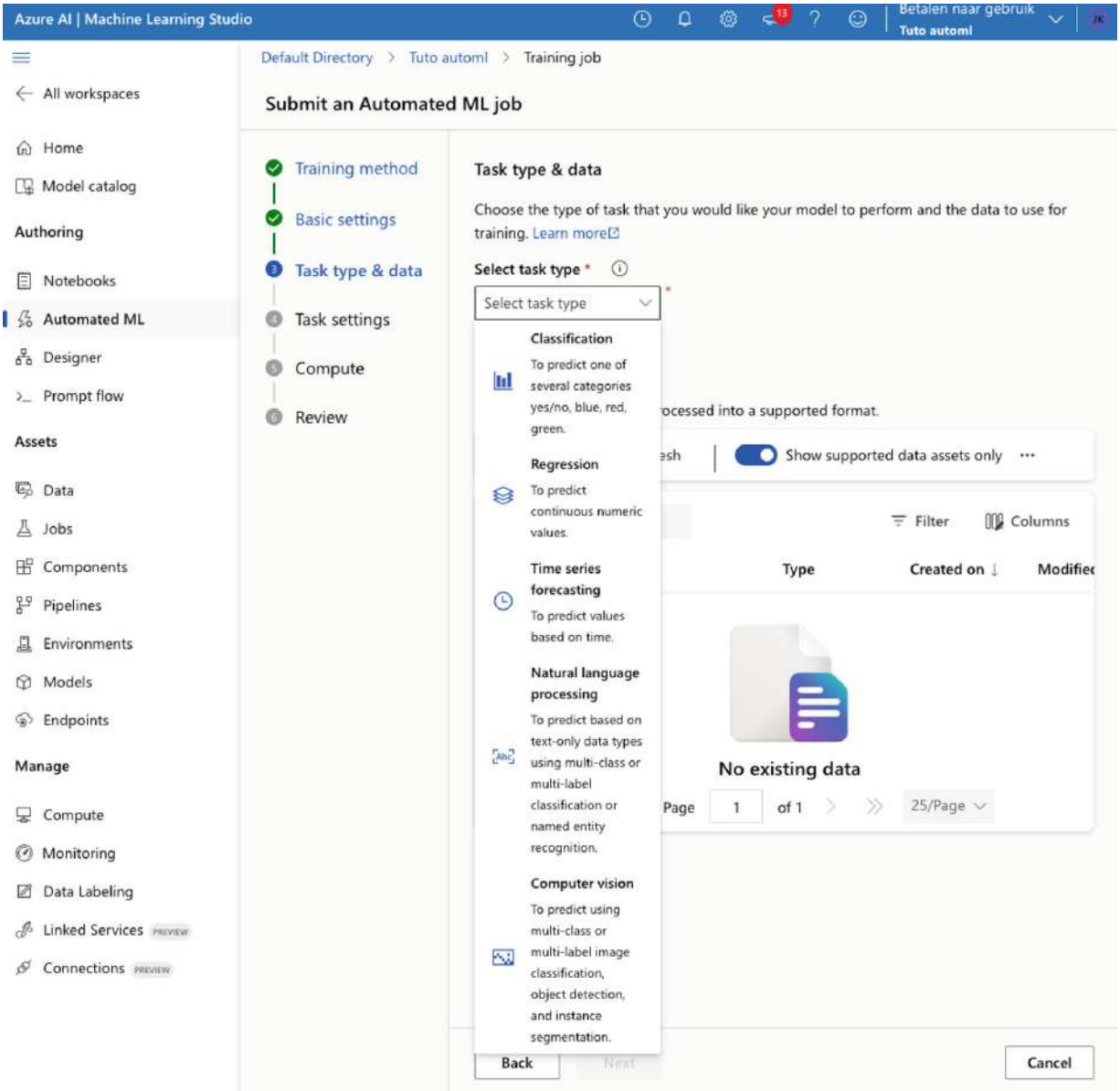


Figure 26-4 Select task type: time series forecasting

To create the data, you need to click the Create data option. You'll then see the Create data asset menu pop-up (Figure 26-5).

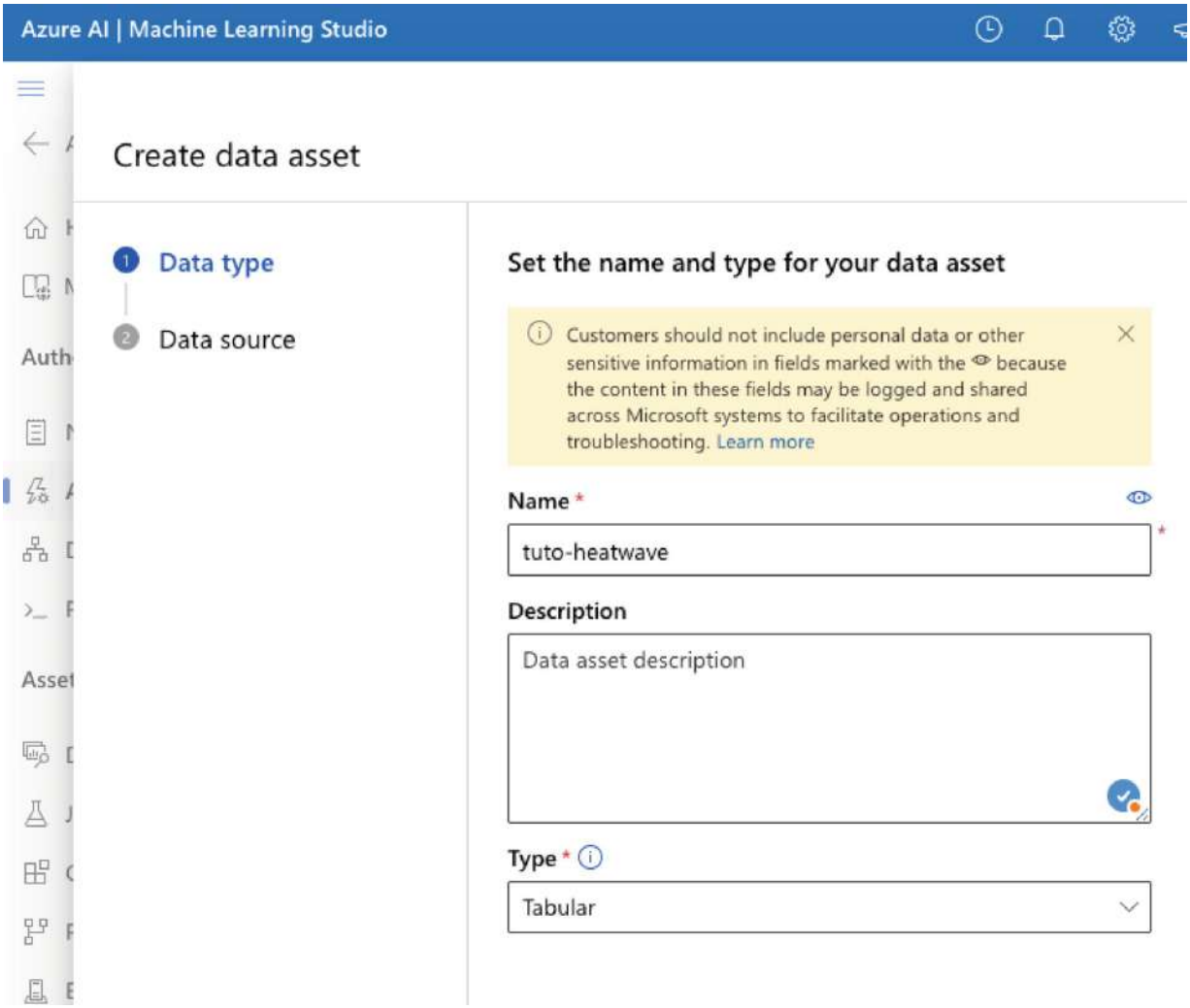


Figure 26-5 Select the data type tabular

You can select the data type tabular. After that, you can see in Figure 26-6 how to set up a data import from a local file. We'll be using the Heat Wave data that we used before. Before uploading the file, make sure to delete the test set from your data, as we did in previous chapters. You can do this locally in Excel. Delete 2024 and 2025 to only provide the training data to the Azure Automated ML job.

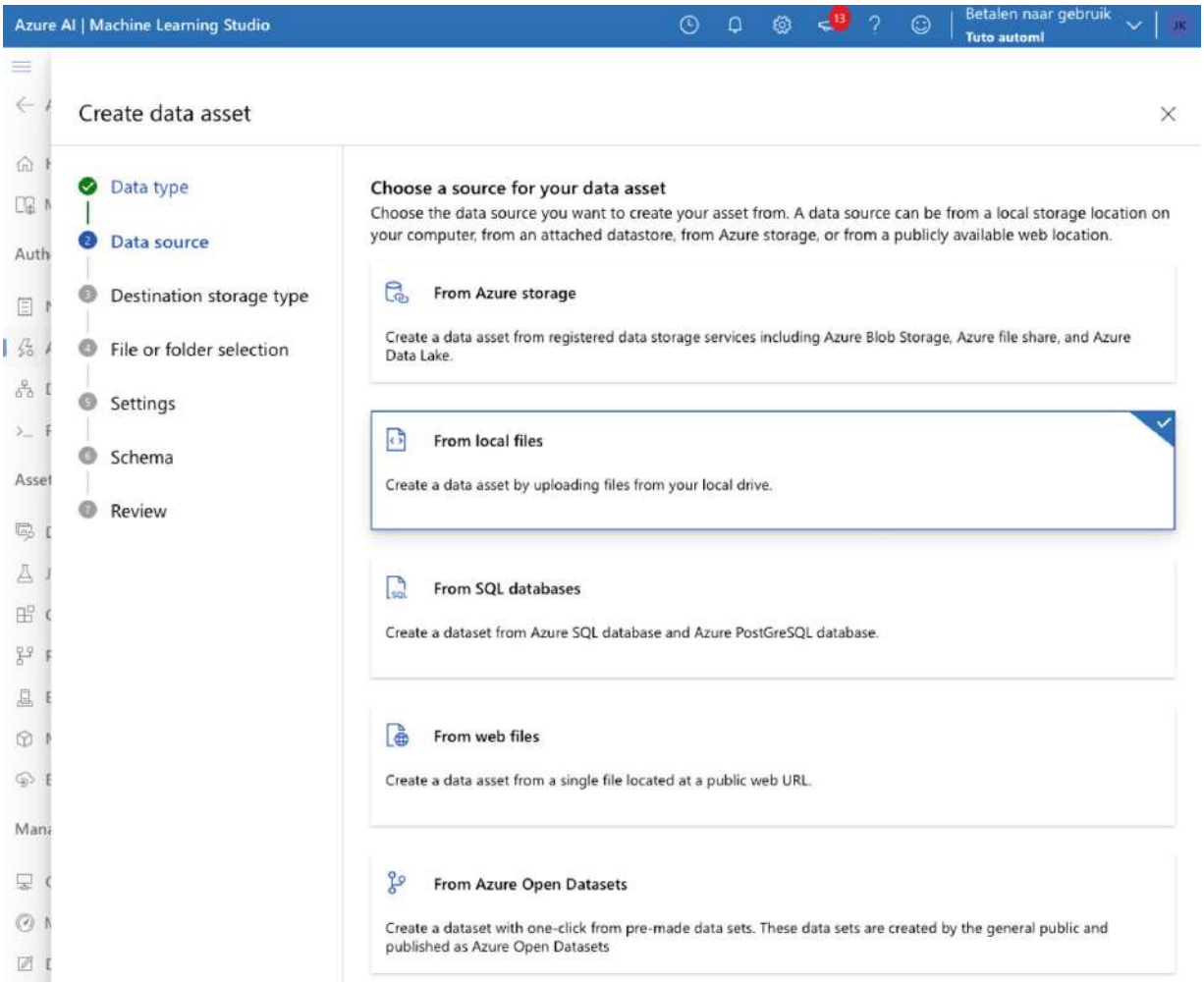


Figure 26-6 Set up data import from local files

Once you have clicked “From local files”, you need to select a datastore. As shown in Figure 26-7, you can select the Azure Blob Storage for this.

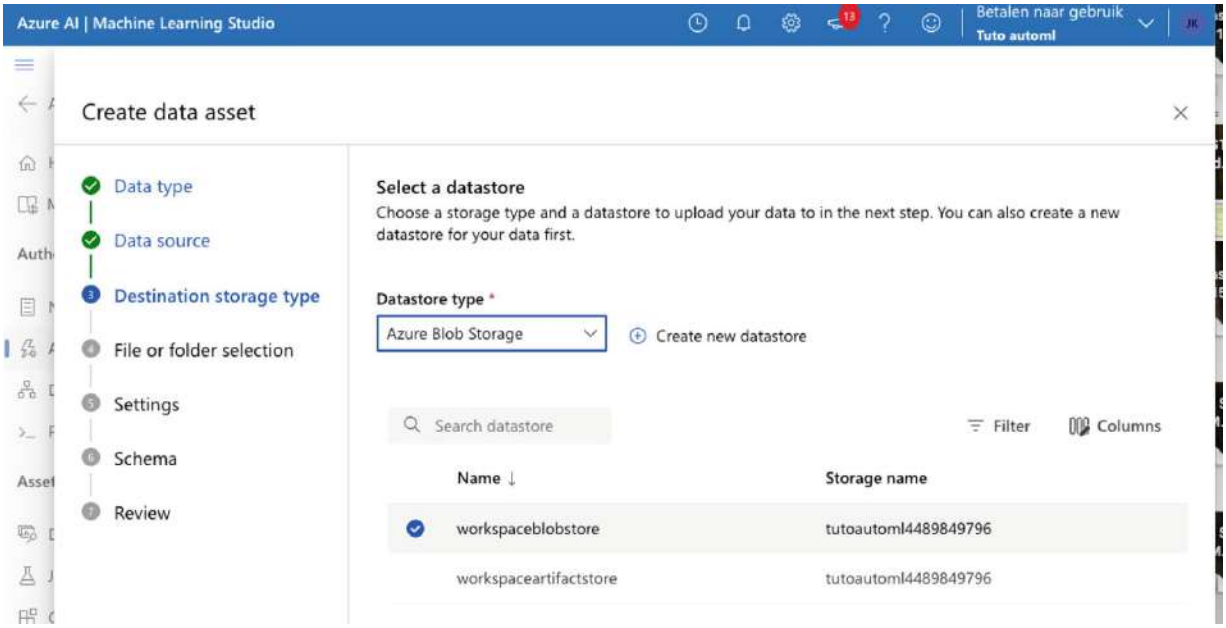


Figure 26-7 Select Azure Blob Storage

Now, you can upload your file, as shown in Figure 26-8.

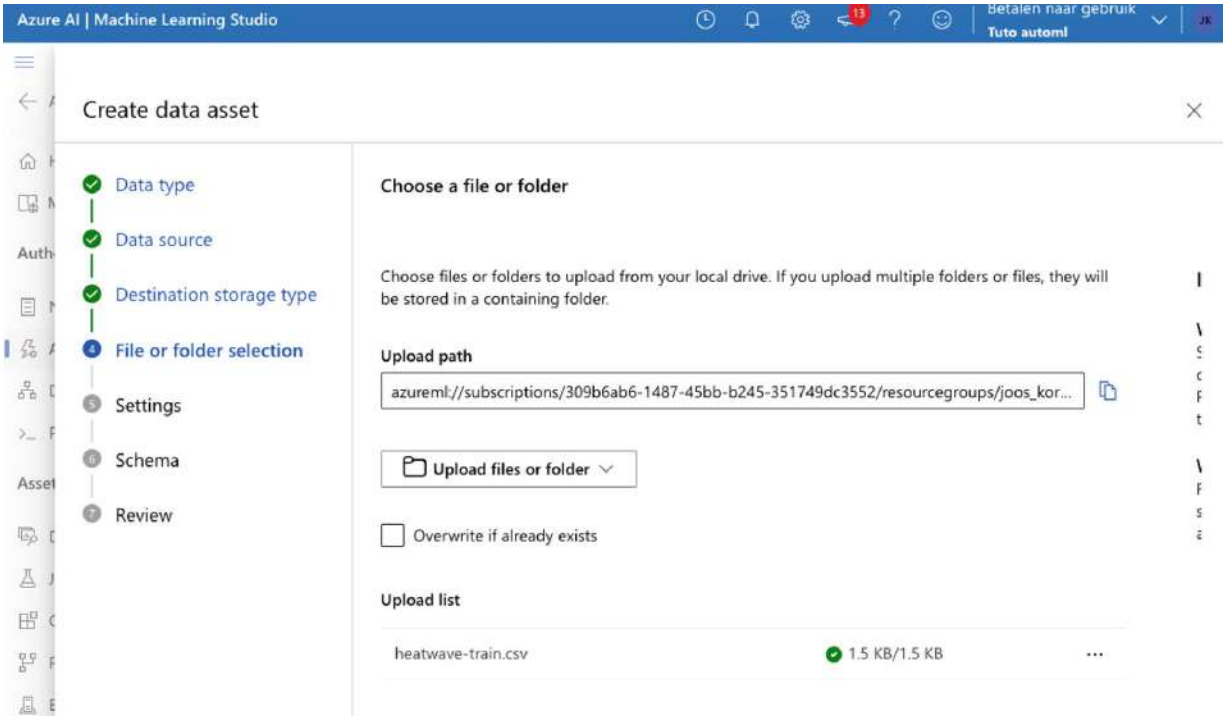


Figure 26-8 Upload the data

Once you're data is uploaded, you need to specify some settings to make sure the data is uploaded correctly. The dataset is a comma-delimited,

UTF-8 encoded CSV file with headers. You should see that the data is correctly imported in the preview, as shown in Figure 26-9.

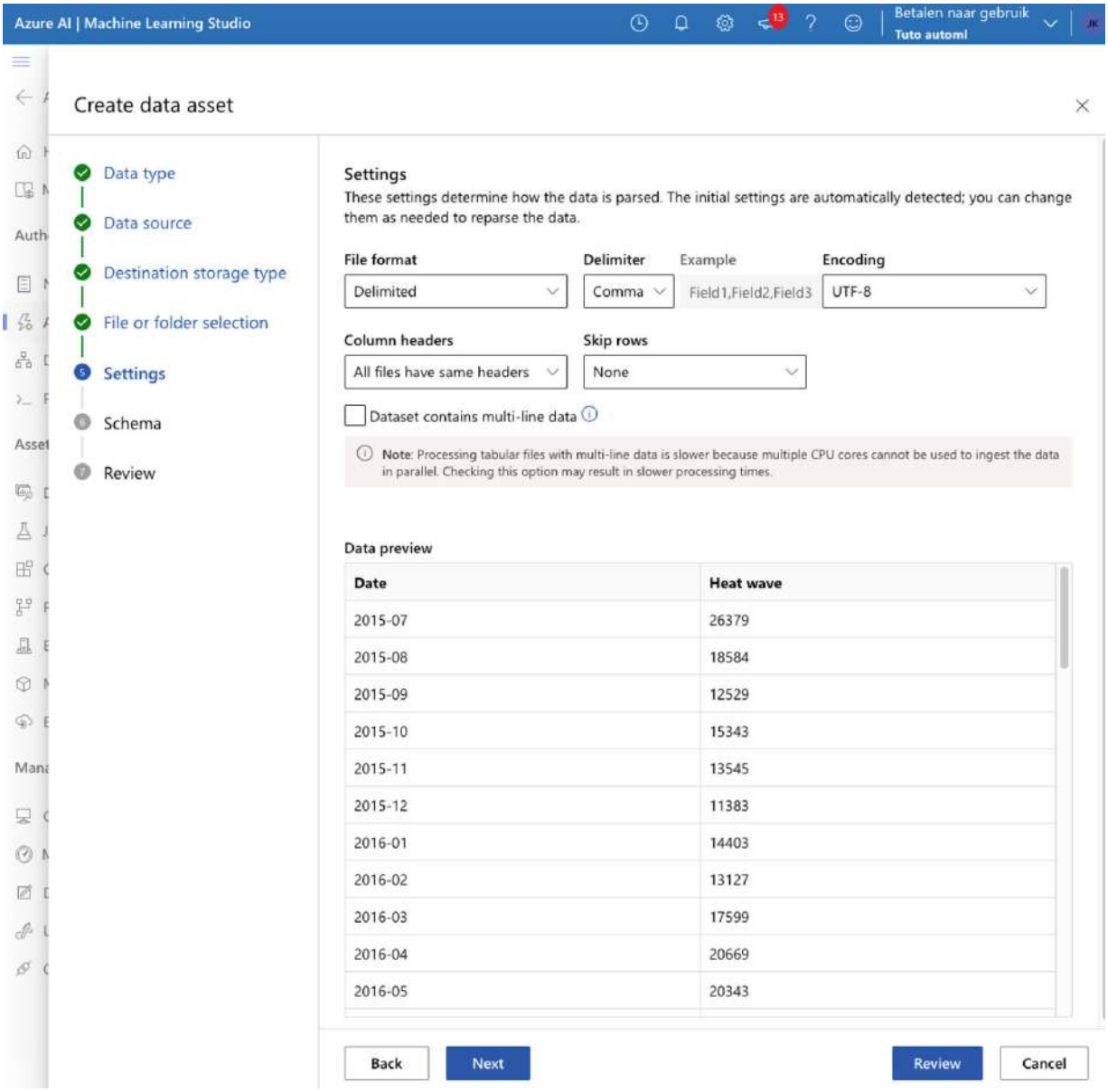


Figure 26-9 Preview of the data

When you click Next, you will get the Schema menu as shown in Figure 26-10. You need to select the Date format for the column called Date and specify the right format. “%Y-%m” can be used for data that has the year in four numbers, a dash, and the month in two numbers.

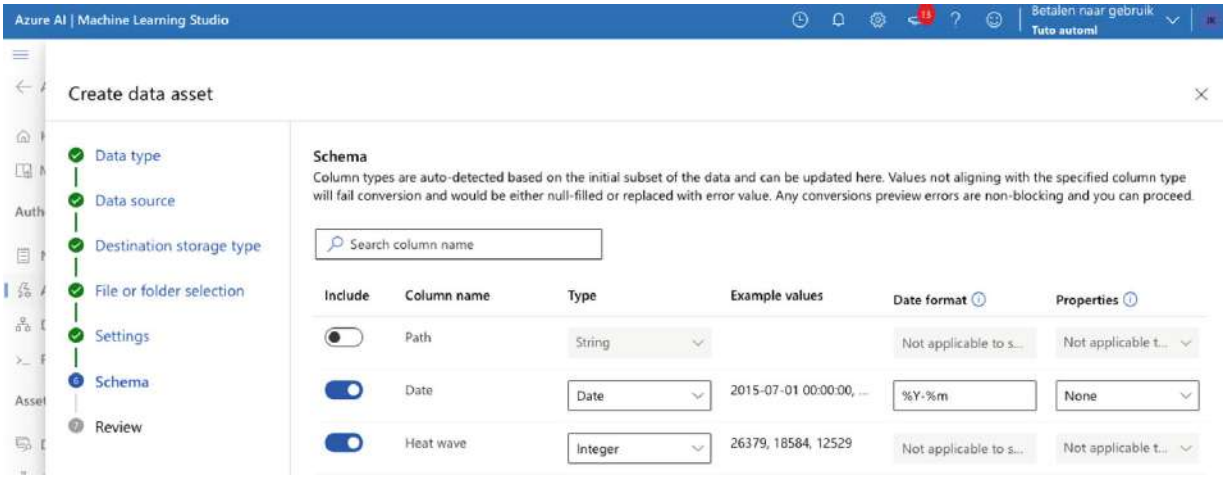


Figure 26-10 Specify the schema of the data

In the final Review tab, click Create. You'll now come back to your Automated ML job, and your data has been selected. This is shown in Figure 26-11.

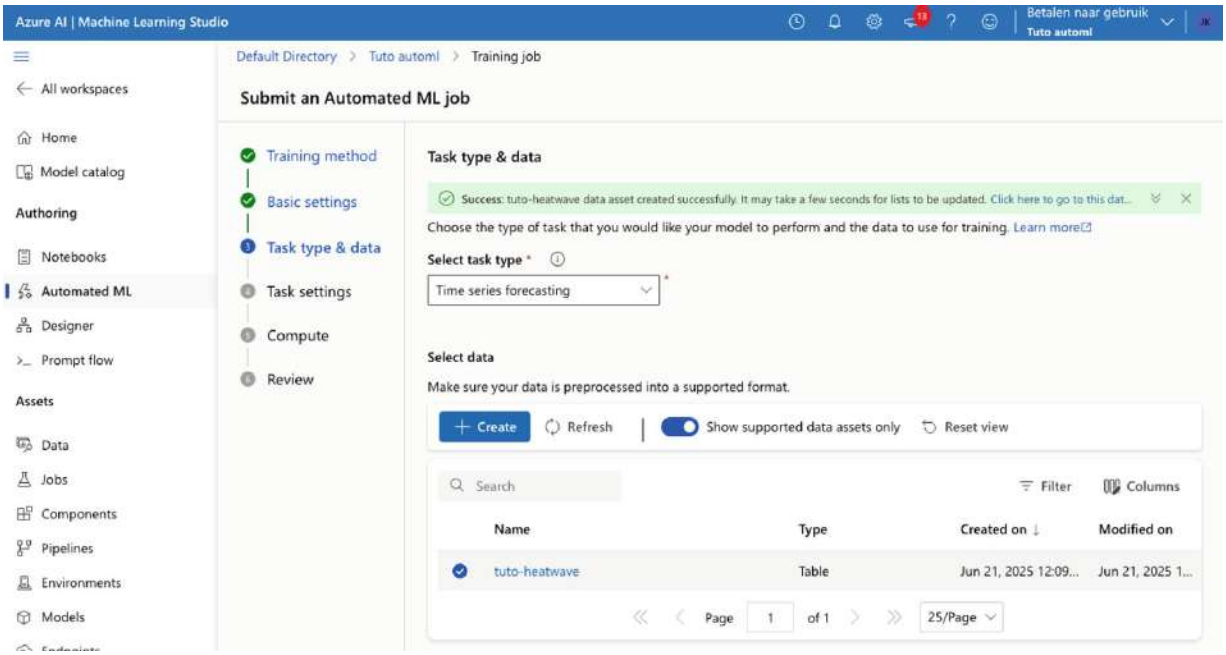


Figure 26-11 The menu for the Automated ML job is now updated with the data

When continuing, we now come into the Task settings menu. Select the following options (an example can be found in Figure 26-12):

- The Target column is Heat wave.
- The Time column is Date.

- The Autodetect forecast horizon must be set to 12.

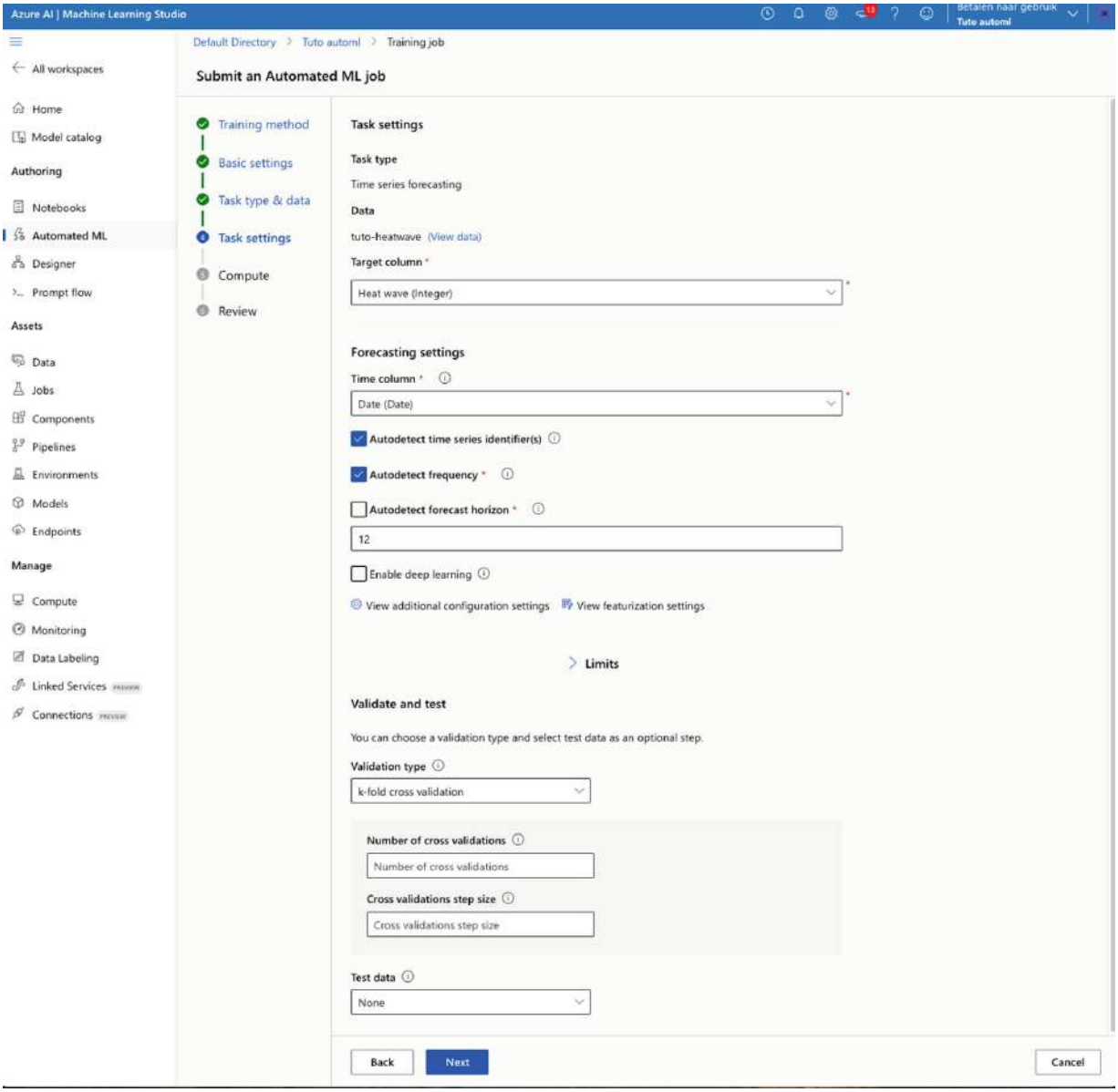


Figure 26-12 Task settings for the Automated ML job

You should also set a **limit**, for example, set a 15-minute max compute time, to limit costs. Also, set Enable early termination. An example is shown in Figure 26-13.

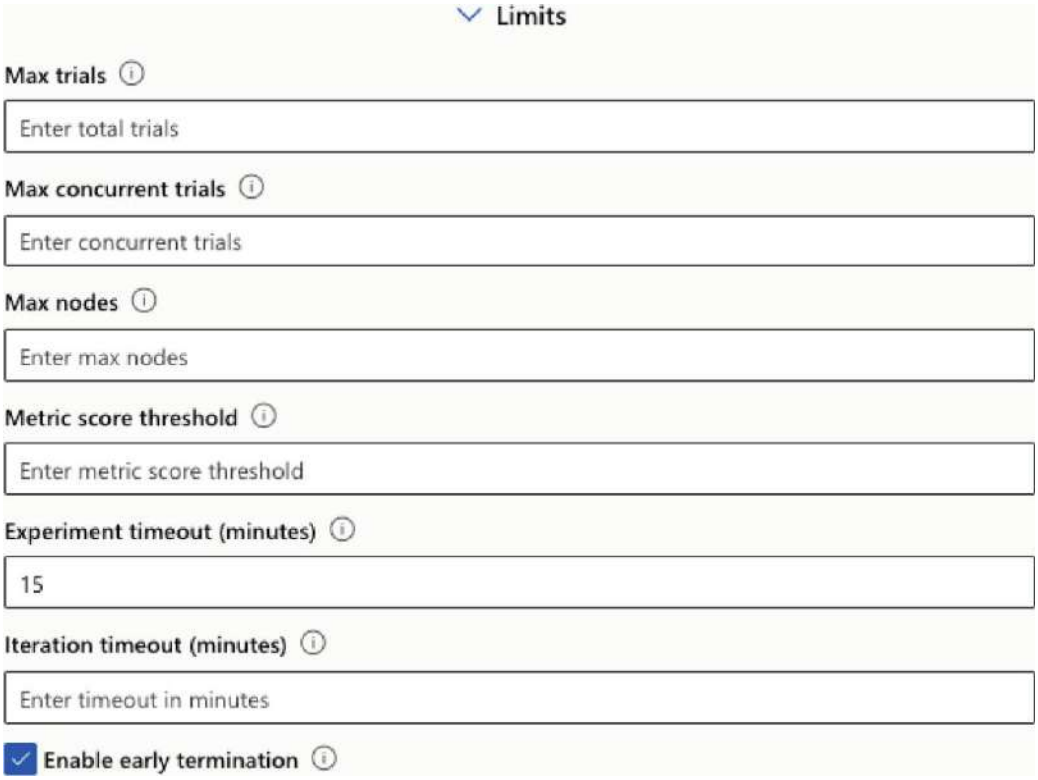


Figure 26-13 Set up the limit

When continuing, you will get to the Compute menu. You can take a simple and cheap virtual machine, as shown in Figure 26-14.

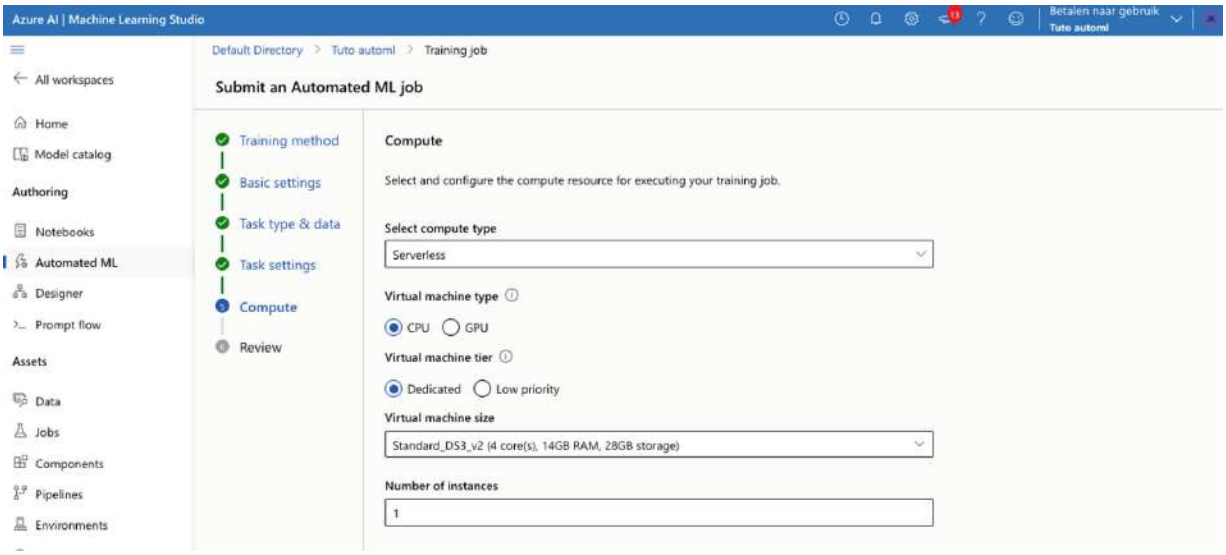


Figure 26-14 Set up the compute

In the review chapter, click on submit training job (as mentioned, running this can cost money). If you run the job, you'll see the screen as in Figure 26-15.

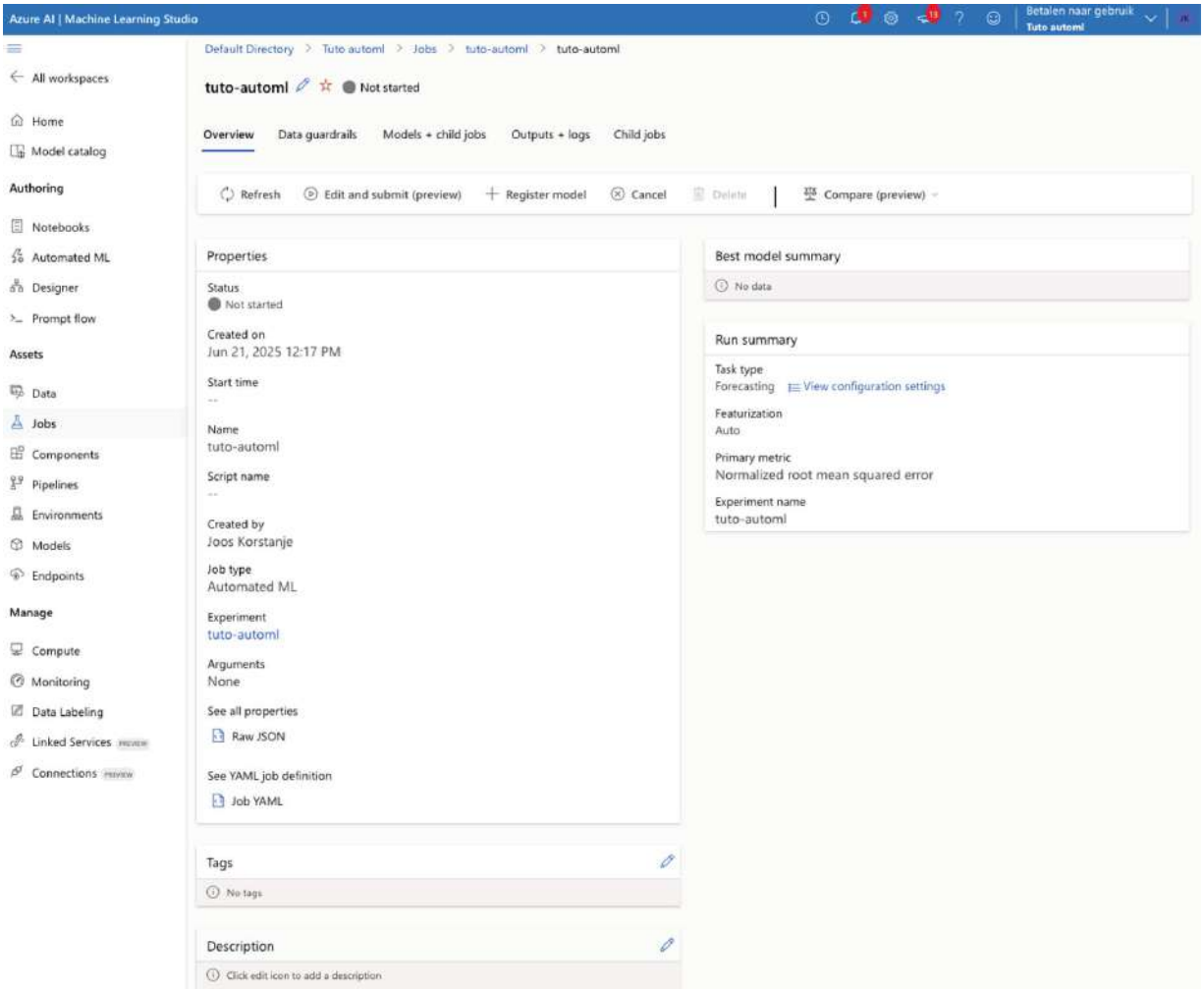
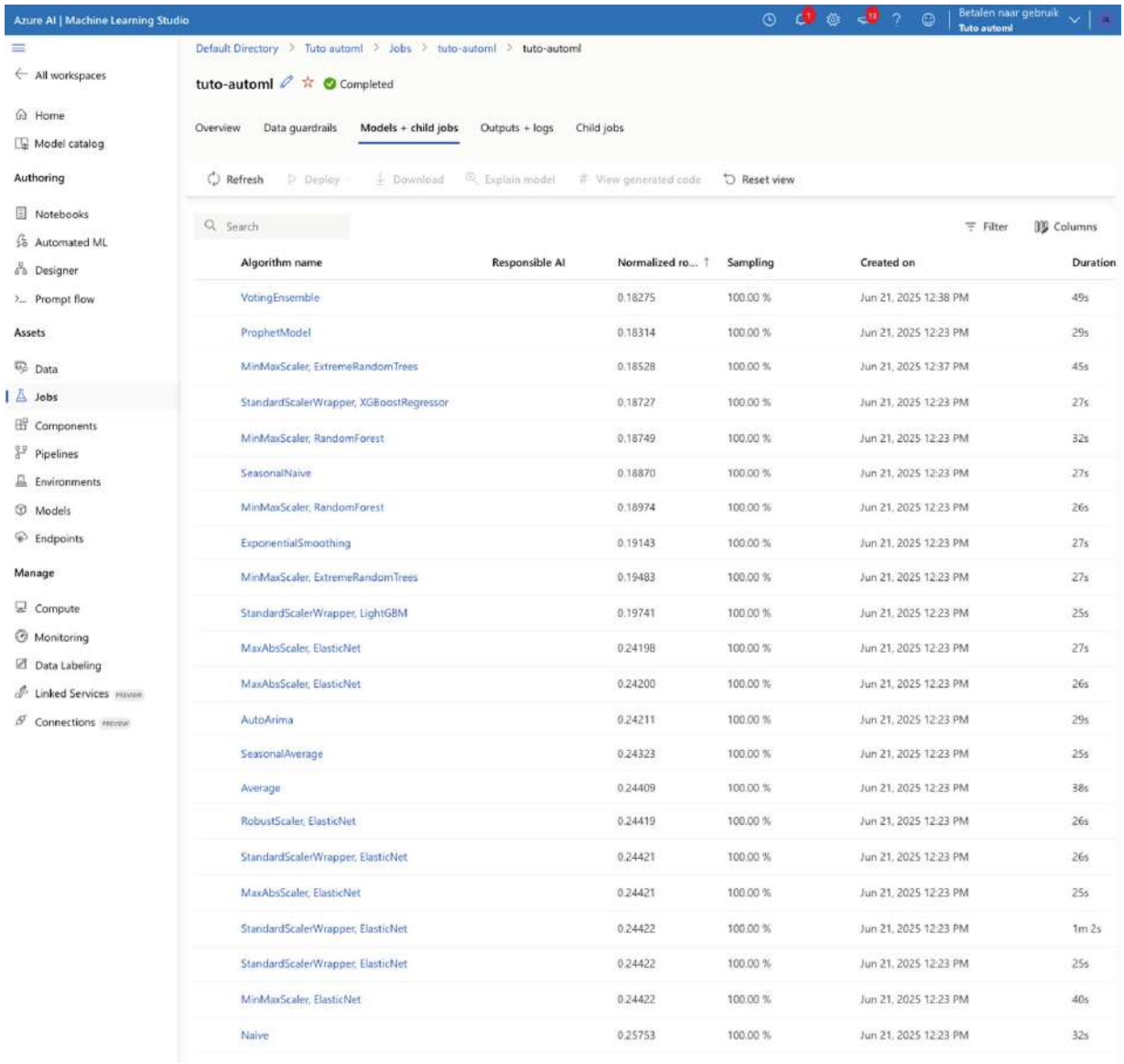


Figure 26-15 The Automated ML job

Now, you have to wait until the job finishes. Once the model is completed, you can see all the models that Azure AutoML has benchmarked. This is shown in Figure 26-16. You'll actually recognize a lot of them that you have seen throughout the book!



Algorithm name	Responsible AI	Normalized ro...	Sampling	Created on	Duration
VotingEnsemble		0.18275	100.00 %	Jun 21, 2025 12:38 PM	49s
ProphetModel		0.18314	100.00 %	Jun 21, 2025 12:23 PM	29s
MinMaxScaler, ExtremeRandomTrees		0.18528	100.00 %	Jun 21, 2025 12:37 PM	45s
StandardScalerWrapper, XGBoostRegressor		0.18727	100.00 %	Jun 21, 2025 12:23 PM	27s
MinMaxScaler, RandomForest		0.18749	100.00 %	Jun 21, 2025 12:23 PM	32s
SeasonalNaive		0.18870	100.00 %	Jun 21, 2025 12:23 PM	27s
MinMaxScaler, RandomForest		0.18974	100.00 %	Jun 21, 2025 12:23 PM	26s
ExponentialSmoothing		0.19143	100.00 %	Jun 21, 2025 12:23 PM	27s
MinMaxScaler, ExtremeRandomTrees		0.19483	100.00 %	Jun 21, 2025 12:23 PM	27s
StandardScalerWrapper, LightGBM		0.19741	100.00 %	Jun 21, 2025 12:23 PM	25s
MaxAbsScaler, ElasticNet		0.24198	100.00 %	Jun 21, 2025 12:23 PM	27s
MaxAbsScaler, ElasticNet		0.24200	100.00 %	Jun 21, 2025 12:23 PM	26s
AutoArima		0.24211	100.00 %	Jun 21, 2025 12:23 PM	29s
SeasonalAverage		0.24323	100.00 %	Jun 21, 2025 12:23 PM	25s
Average		0.24409	100.00 %	Jun 21, 2025 12:23 PM	38s
RobustScaler, ElasticNet		0.24419	100.00 %	Jun 21, 2025 12:23 PM	26s
StandardScalerWrapper, ElasticNet		0.24421	100.00 %	Jun 21, 2025 12:23 PM	26s
MaxAbsScaler, ElasticNet		0.24421	100.00 %	Jun 21, 2025 12:23 PM	25s
StandardScalerWrapper, ElasticNet		0.24422	100.00 %	Jun 21, 2025 12:23 PM	1m 2s
StandardScalerWrapper, ElasticNet		0.24422	100.00 %	Jun 21, 2025 12:23 PM	25s
MinMaxScaler, ElasticNet		0.24422	100.00 %	Jun 21, 2025 12:23 PM	40s
Naive		0.25753	100.00 %	Jun 21, 2025 12:23 PM	32s

Figure 26-16 The results of your Azure AutoML

You have now successfully run an ML training job on Microsoft Azure! You can see that the Azure AutoML job has done an internal benchmark, and it found that the VotingEnsemble model is the best choice for our Heat Wave use case.

Optional: Download MLflow Artifact of the Model

When you go to the best model, you click on the artifact, and you'll be able to download an MLflow format of the model. We are now coming back into known territory.

As we have seen before, this is a folder with

- conda.yaml
- MLmodel
- model.pkl
- python_env.yaml
- requirements.txt

If you want, you can spin up a local Python Notebook and use MLflow to import the model as we have seen before. However, downloading the model to the local would not necessarily be the logical choice. After all, when using Cloud computing, we would often keep the model in the cloud and try to deploy it in the cloud.

If you want to go further in Azure, you can either choose to set up a batch transform job, which will allow you to score the model on a test dataset. You can also choose to set up a real-time data set behind an API. For further reading on this topic, I recommend the Azure documentation (https://learn.microsoft.com/en-us/azure/machine-learning/concept-endpoints?view=azureml-api-2).

Attention If you have followed along with this chapter, make sure to shut down all resources and end your subscription if you want to avoid costs.

Key Takeaways

- Azure is the Cloud computing service from Microsoft.
- Automated ML jobs are one of the many ML services provided by Azure.
- Inside the Automated ML jobs, there is a specific category for time series forecasting.
- The Automated ML job does not provide its own mathematical model. Rather, it benchmarks the existing model and provides a ranking.
- Once you finish training the Automated ML job, you can download the model in an MLflow format or decide to deploy it as a batch or real-time scoring endpoint.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_27

27. AutoML with Vertex AI on Google Cloud Platform

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, we will dive into the Google Cloud Platform. Just like Amazon and Microsoft, Google also has its own cloud. At the moment, Amazon's AWS, Microsoft's Azure, and Google's GCP are the top clouds for general usage. Besides those three main players, there are other, smaller, general cloud providers.

The goal of this book is to provide advanced forecasting techniques, with a focus on Python and cloud use. As GCP is a big player in the cloud field, this chapter is dedicated to building an AutoML forecasting model using GCP's Vertex AI. By doing this, you will see how to get started with machine learning in the cloud, and if this is something of interest to you, it will allow you to start your cloud journey.

Be careful: if you are following along with the instructions in this chapter, Google will bill you for costs incurred. Running cloud resources is not free.

GCP and BigQuery

Google Cloud Platform launched officially in 2008, proposing only an App Engine for developers. Between 2010 and 2013, GCP became interested in the field of data, as it started to propose BigQuery. When considering the data stack, BigQuery is still an important tool on GCP. BigQuery is a fully managed, serverless data warehouse that allows you to run fast SQL queries over massive datasets. It is one of the flagship solutions of GCP.

Throughout recent years, GCP has increased a lot and has now become a full cloud generalist. A service in which they made great progress as well is machine learning and AI, through the service called Vertex AI.

Vertex AI

Vertex AI is Google Cloud's platform for Machine Learning and AI. Just like Amazon SageMaker and Microsoft Azure's ML Studio, Vertex AI also proposes a full package of services. These range from running Jupyter Notebooks in the cloud to model building using AutoML or with custom training code and to industrializing model training pipelines and managing inference jobs and endpoints.

Forecasting Heat Wave Page Views with Google Cloud Platform's Vertex AI AutoML

In previous chapters, you have already seen how to build models on Amazon's AWS and on Microsoft Azure. In this example, we'll be doing the exact same use case as we did on the other two cloud platforms. As getting started is often the hardest part when going into the cloud, the goal of this exercise is to make sure that you are able to set up the cloud environment for a modeling use case.

If you're set up and understand how to get started with the basic cloud tools, you'll be much more efficient with your journey into the cloud of your choice. After all, the real added value of cloud computing is that it takes away the physical management of IT services, which allows you to set up whole systems from your desk chair.

Step 1: Data Preparation for GCP

When working with data in GCP, specifically when working with Vertex AI's AutoML, our data needs to match a specific set of requirements. We have two options here: using either BigQuery or CSV files.

BigQuery is a great tool for data analytics, but it is not worth it to go through the setup of BigQuery just to run a Machine Learning job. Therefore, in the current example, we are going to use a CSV file. However,

if you want to discover GCP more, feel free to try and set up the example using BigQuery.

When working with CSV files on Vertex AI Forecast, we need to respect the following requirements:

- Three columns requested by GCP:
  - Target
  - Timestamp
  - Time series ID
- The first line must contain headers.
- Column names can only contain alphanumeric characters or underscore (no spaces).
- File size maximum 10 GB.
- Comma delimiter.
- Minimum 1000 rows.
- Maximum 3000 rows.

These requirements are quite restrictive. The minimum and maximum number of rows is even questionable for building a machine learning model. But as we're working on the GCP cloud, we need to adapt.

We'll take the heat wave train data that we used in the previous chapter. However, we need to transform it to look as shown in Figure 27-1. You can use the tool of your choice, for example, Excel or Python, to do this.

```
Date,Heatwave,ID
01/07/2015,26379,A
01/08/2015,18584,A
01/09/2015,12529,A
01/10/2015,15343,A
01/11/2015,13545,A
01/12/2015,11383,A
```

Figure 27-1 The data

In order to obtain the 1000 rows, we unfortunately need to apply a hack. Let's do the following: convert the monthly data to daily data by adding duplicates. We can add the same value for each day of the same month. You

should obtain something like that shown in Figure 27-2. Of course, this is very unfortunate and will bias any evaluation that we could do. However, it also shows some of the important drawbacks one may encounter when relying entirely on click-button systems.

Date	Heatwave	ID
2015-07-01	26379	A
2015-07-02	26379	A
2015-07-03	26379	A
2015-07-04	26379	A
2015-07-05	26379	A

Figure 27-2 The manipulated data

Step 2: Create a Bucket

Before running the upload, we need to set up a bucket. We need to make sure that we have a bucket on GCP that respects Vertex AI bucket requirements. This bucket should ideally be inside the same project as Vertex AI. Otherwise, you'd need to set up roles to make it accessible.

- Location type: Region (important for Vertex AI integration)
- Storage class: Standard
- Make sure it is in the same project as the Vertex AI job.

To set up a bucket, first go to Cloud Storage and then to Buckets (Figure 27-3).

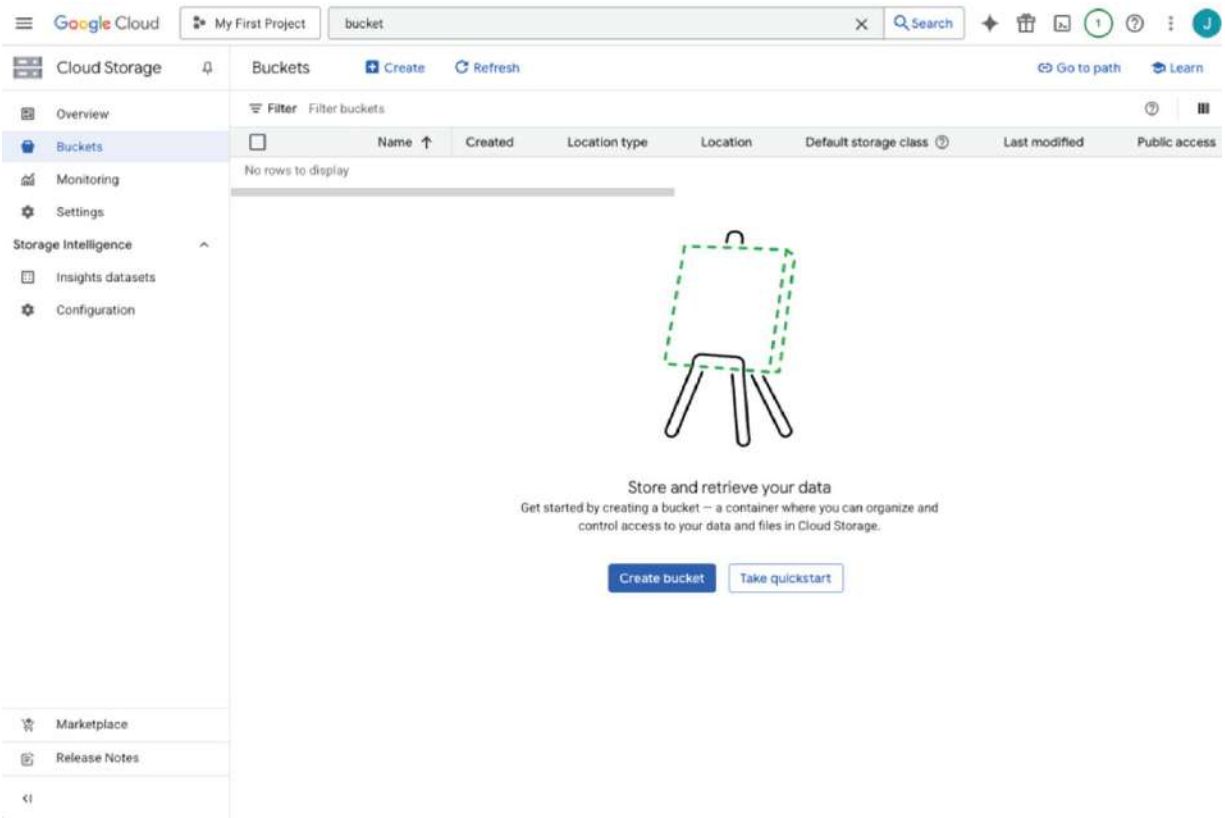


Figure 27-3 Buckets page

Then you click Create, and you will get to the Bucket Creation Settings (Figure 27-4).

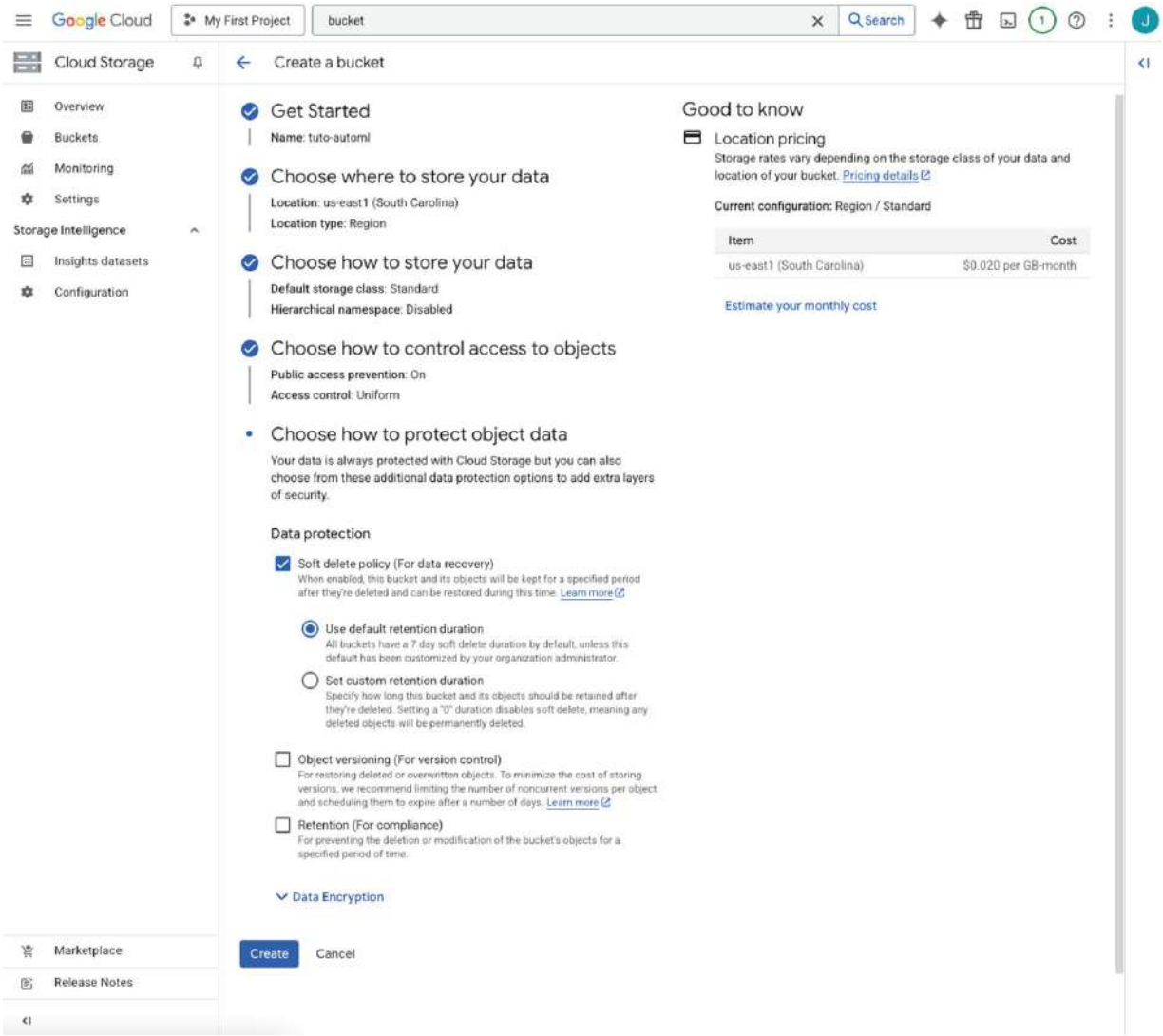


Figure 27-4 Bucket details

Make sure that you fill in the right settings. Mainly, the region is important here. If you have applied your settings, click Create.

Step 3: Uploading the Data

The bucket is still empty for the moment. To fill it, you need to go to the GCP console and go to Vertex AI. Inside Vertex AI, you need to go to the Datasets page and click Create (Figure 27-5).

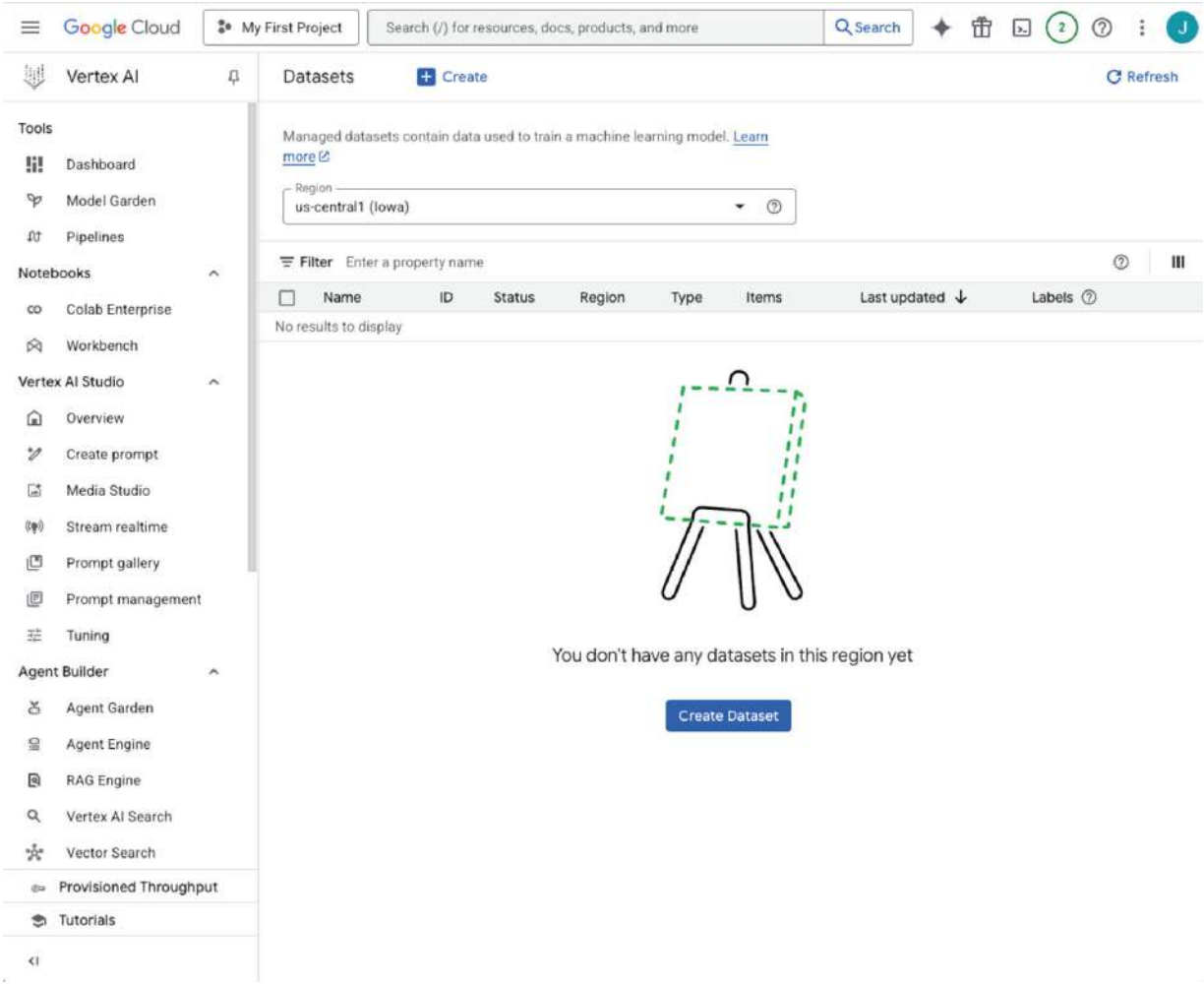


Figure 27-5 Datasets page

You are then requested a number of settings related to your dataset, including

- Dataset name
- Data type (tabular in our case)
- The objective (forecasting in our case)
- Region

This is shown in Figure 27-6.

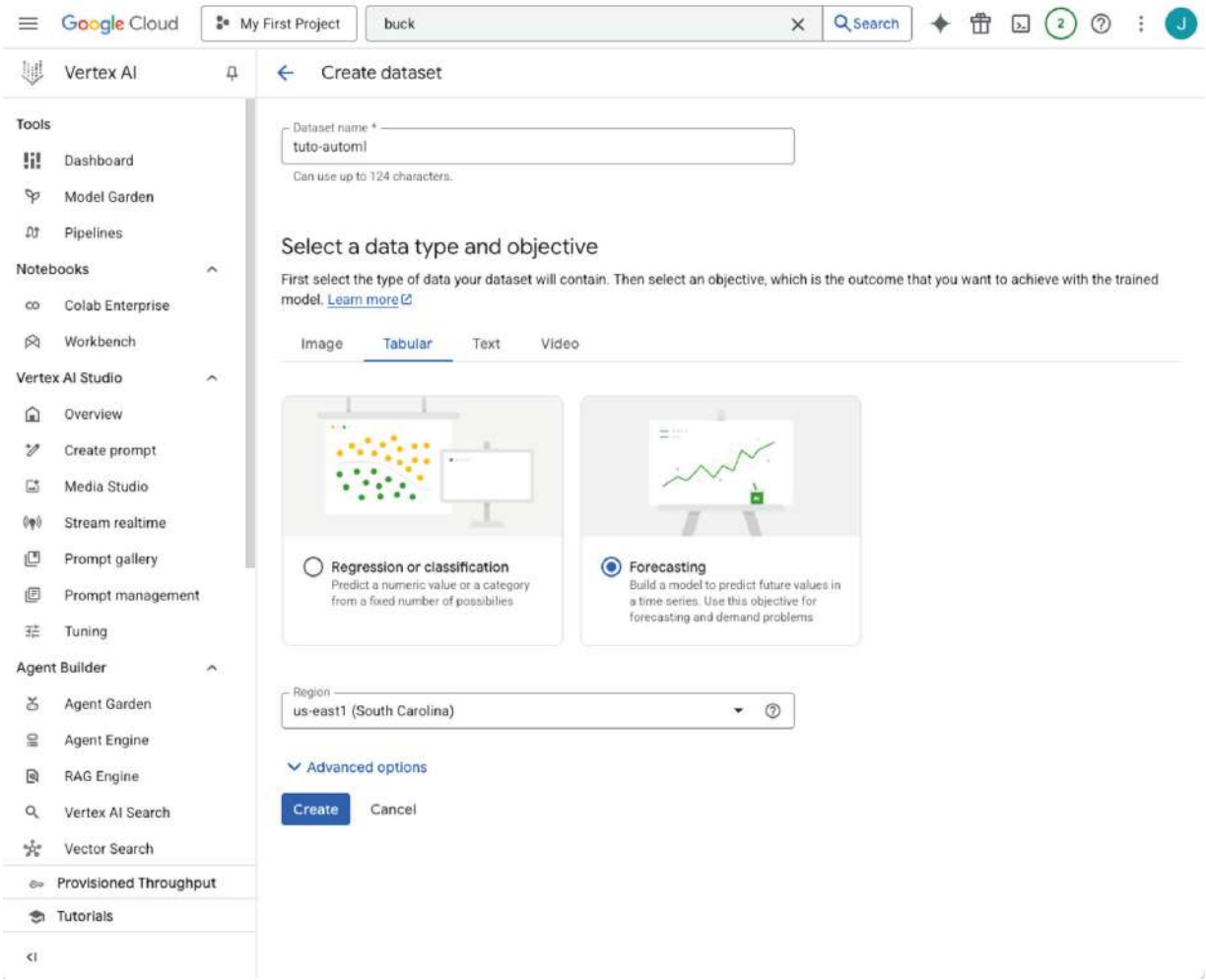


Figure 27-6 Dataset setup

When you continue, GCP will request you to set up the source. In the current example, we'll choose Upload CSV files from your computer. When you select the Cloud Storage path, you need to select the Cloud Storage bucket that you have created before. This is shown in Figure 27-7.

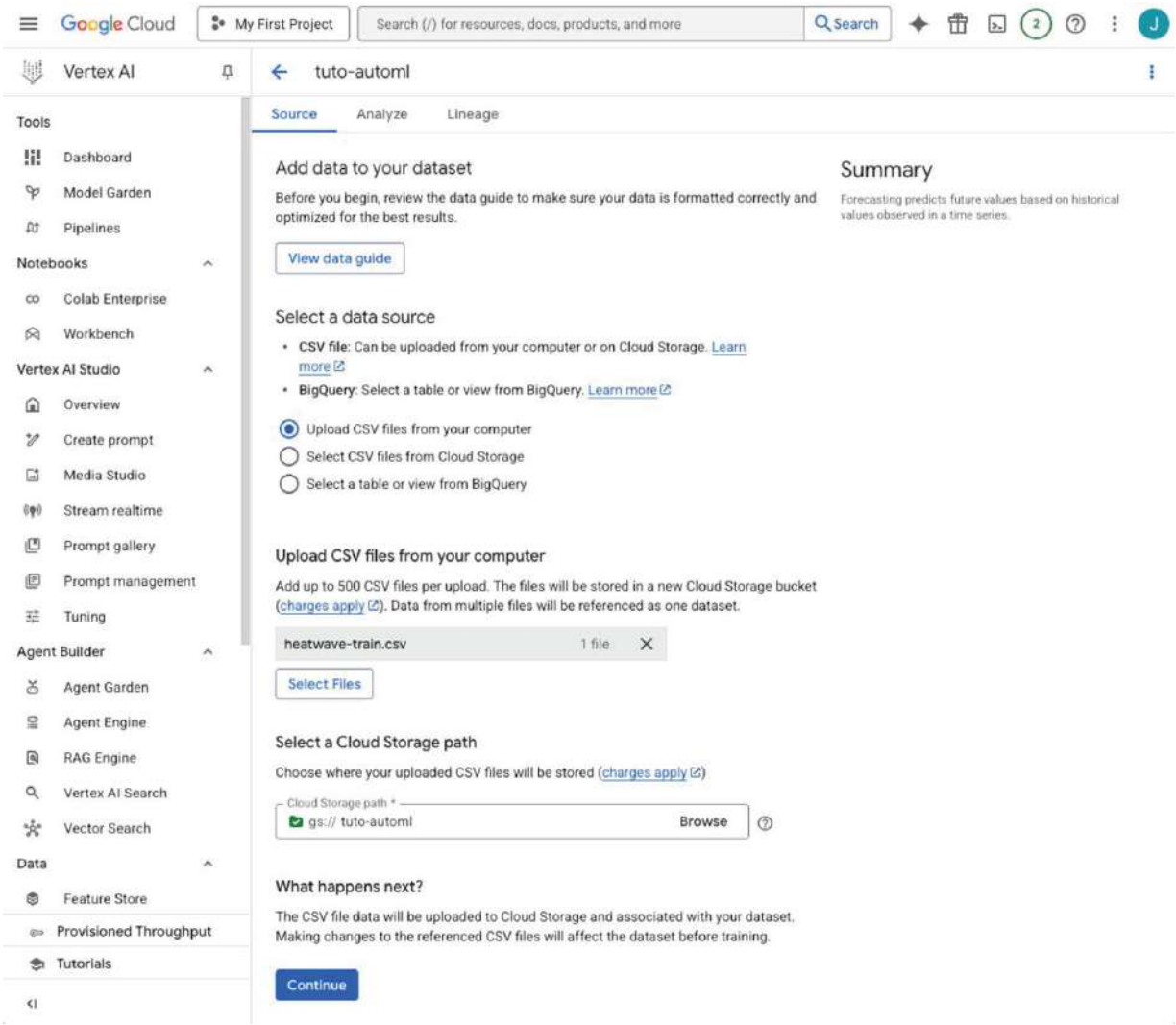


Figure 27-7 Dataset source

Finally, you'll get to the Analyze tab, in which it is important to specify the columns as the Time column and the Identifier column. This is shown in Figure 27-8.

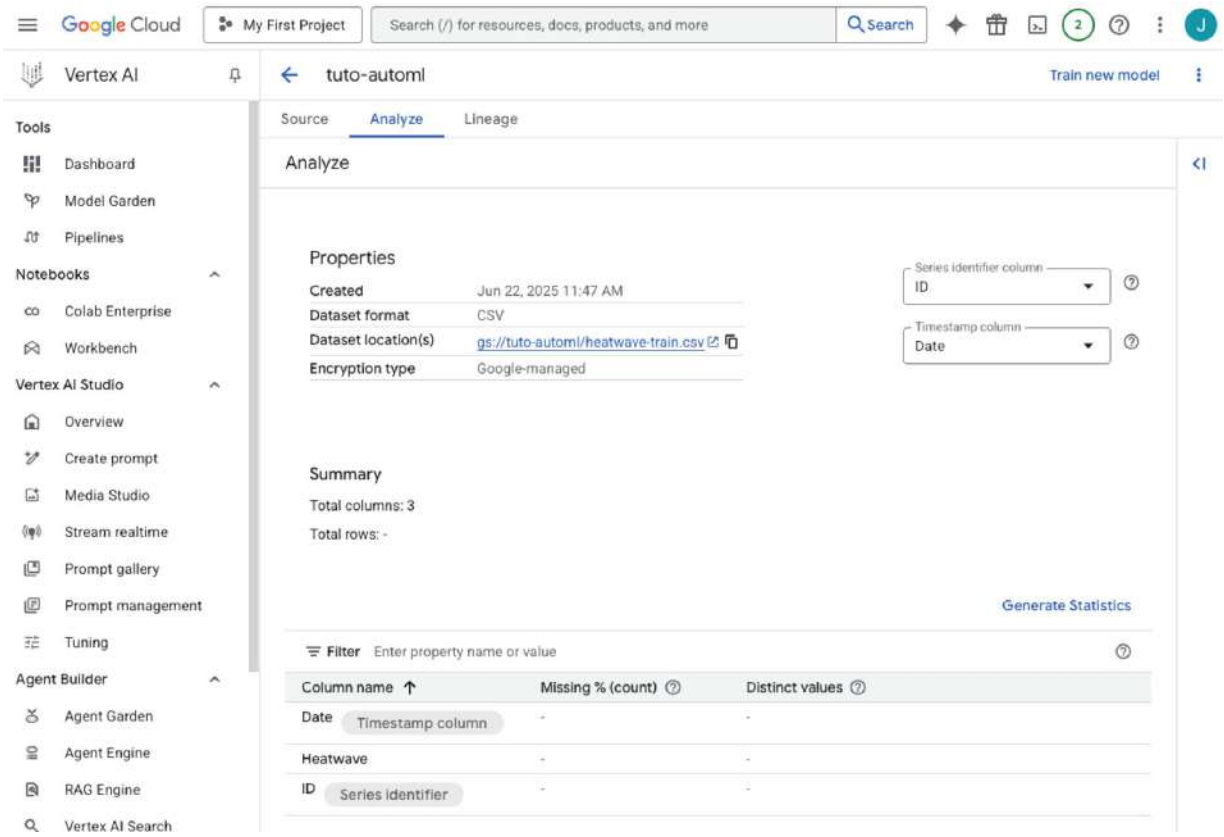


Figure 27-8 Dataset Analyze tab

You'll need to wait for a moment while Vertex AI creates the dataset. When it is done creating, you can simply launch the model-building process by clicking "train new model". This will open the menu shown in Figure 27-9.

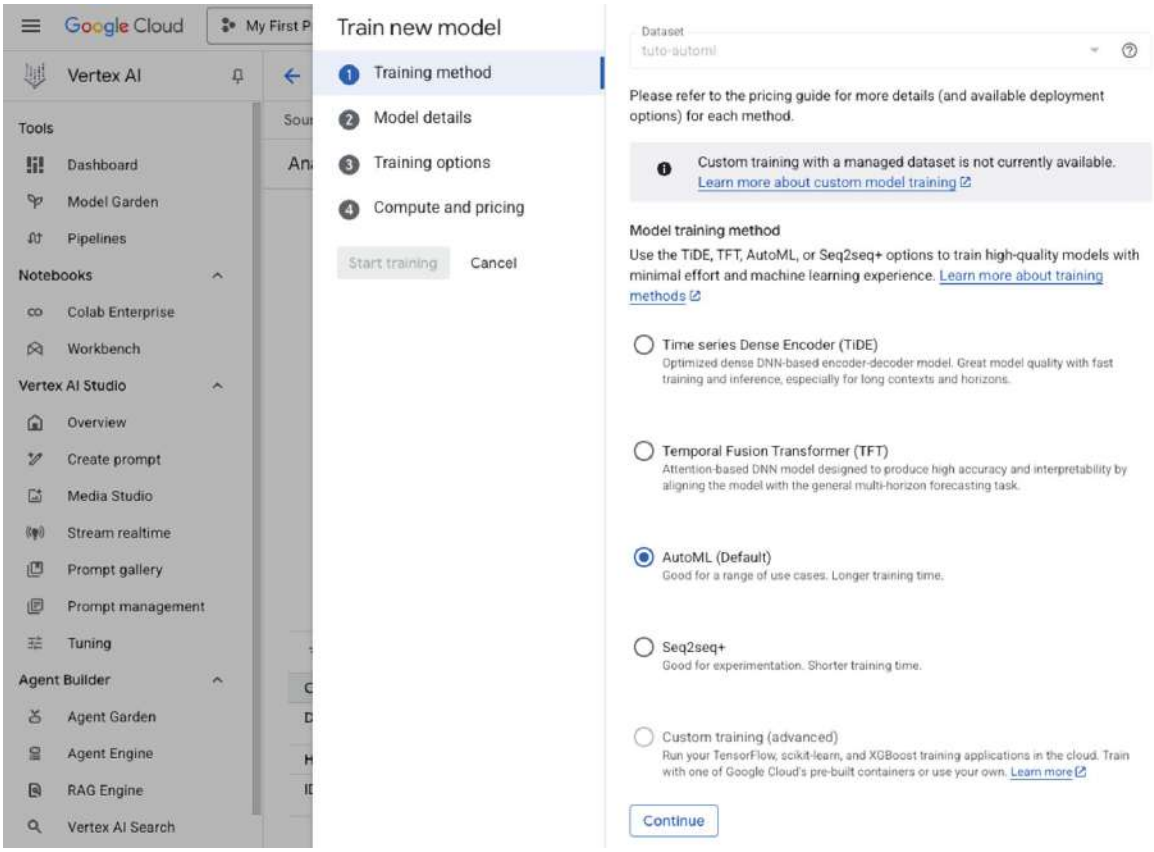


Figure 27-9 Model training settings

When clicking Continue, you'll get to the next step of the menu: Model details. You can set the settings Target Column, Data granularity, Forecast horizon, and Context window. An example is shown in Figure 27-10.

Train new model

1. Training method
2. Model details
3. Training options
4. Compute and pricing

Start training Cancel

Train new model
Creates a new model group and assigns the trained model as version 1

Train new version
Trains model as a version of an existing model

Name *
tuto-ml2

Description

Target column *
Heatwave

Series identifier column *
ID

Timestamp column *
Date

Forecasting configuration

Data granularity *
Daily

The granularity level of the timestamp column. Granularity must be the same for all rows. For example, if "days" is selected, timestamps must be within one day of each other. Data granularity also sets the time period granularity for the forecast horizon and context window.

Holiday regions

Adds a holiday column to your dataset for each selected region. A holiday column adds holiday name values to rows with holiday timestamps. Learn more about holiday regions

Forecast horizon *
365

The number of time periods into the future for which forecasts will be created. Future periods start from the most recent timestamp in the dataset.

Context window *
365

Defines the input lags to the model for each time series. For most use cases, the context window is between 0-5 times the forecast horizon value. For a starting point, try setting the context window equal to the forecast horizon value. Learn more

Export test dataset to BigQuery

Advanced options

Continue

Figure 27-10 Model details

You can also set some advanced options for cross-validation settings, etc. We have seen those throughout the book, so feel free to play around with the settings. For now, let's go with the default settings. This is shown in Figure 27-11.

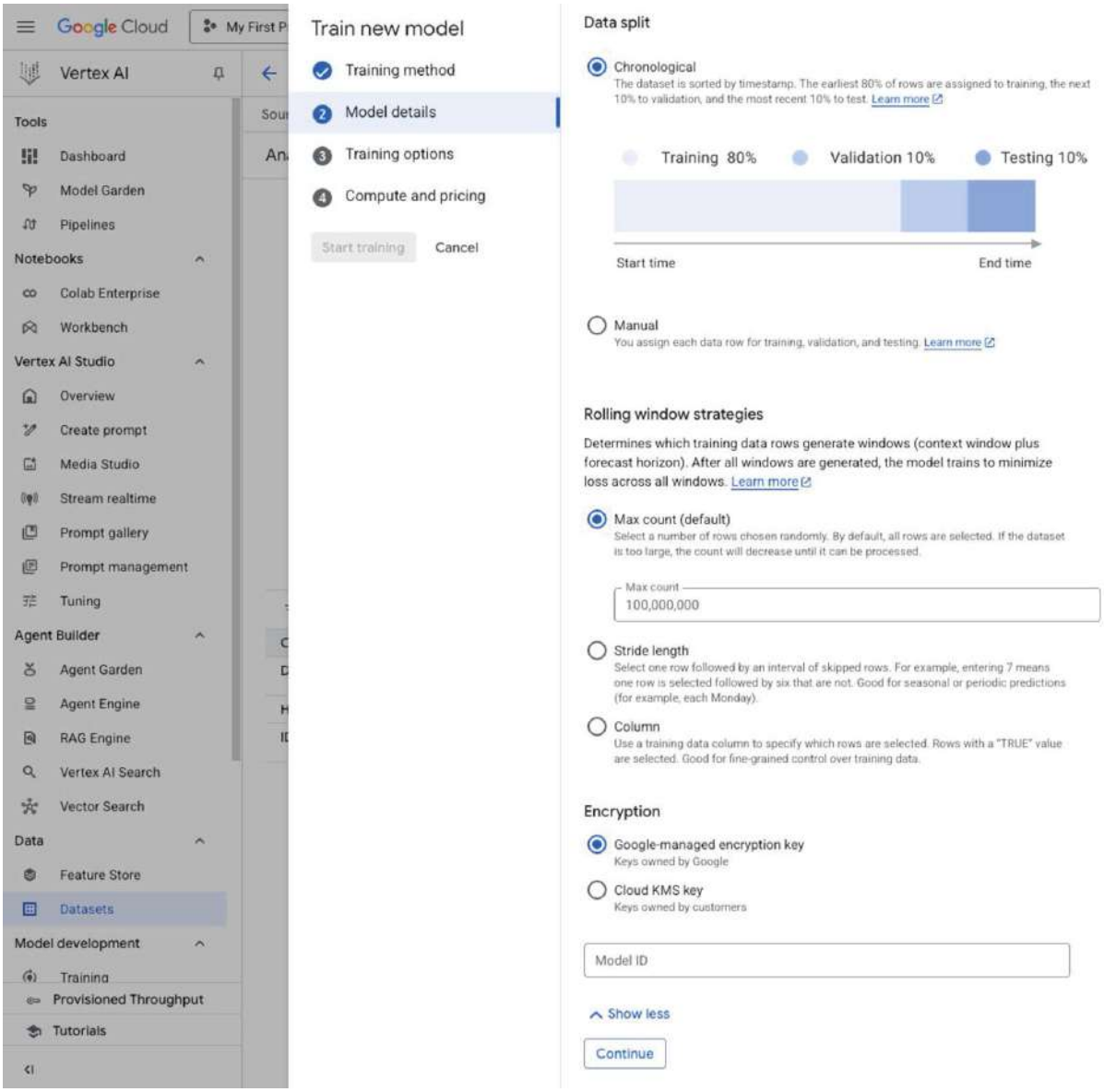


Figure 27-11 Advanced options

Training options are automatically selected, as you can see in Figure 27-12. You could choose some advanced options if you want, but for the sake of simplicity, we'll keep the defaults for now.

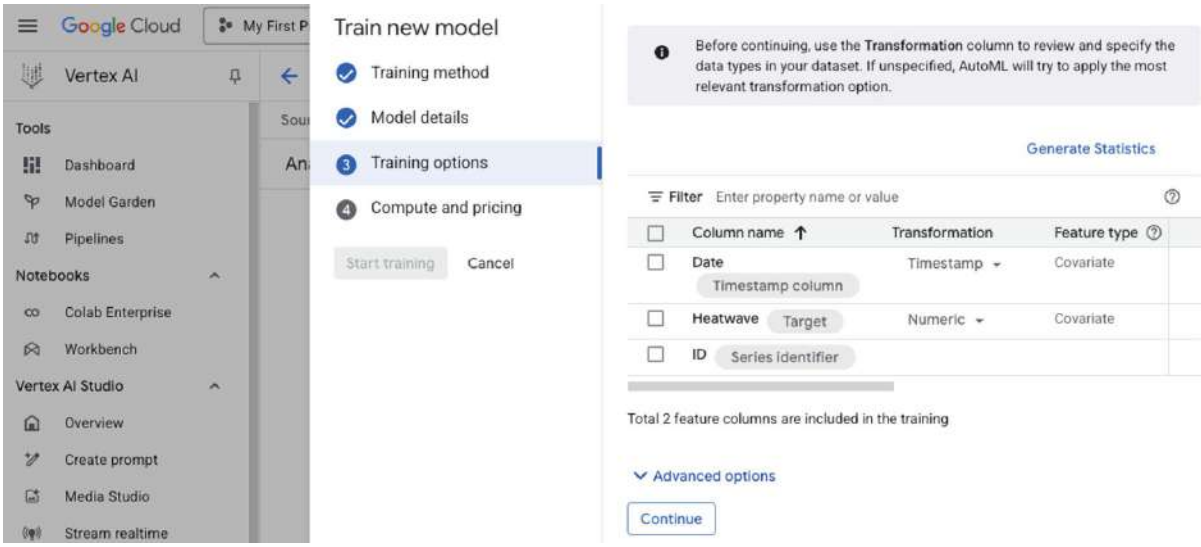


Figure 27-12 Default training options

The next and final step is to choose Compute and pricing, as shown in Figure 27-13. Pricing minimum is 1 hour. Be careful, because running GCP AutoML is quite expensive.

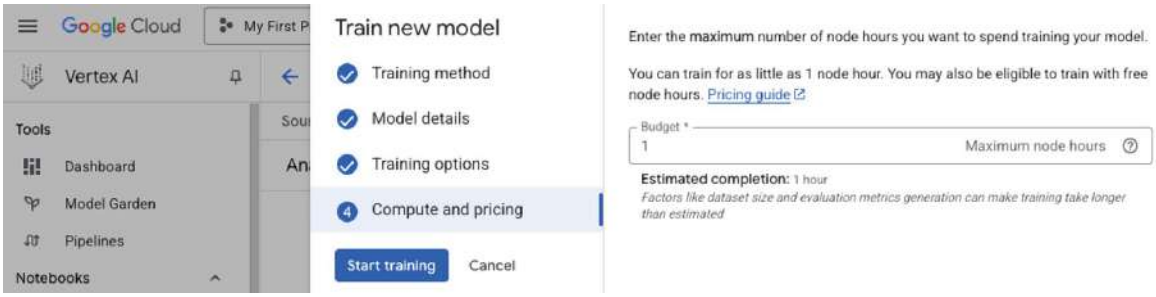


Figure 27-13 Compute and pricing

If you want to follow along, you can start the training. If you start it, you can see the resulting screen in Figure 27-14.

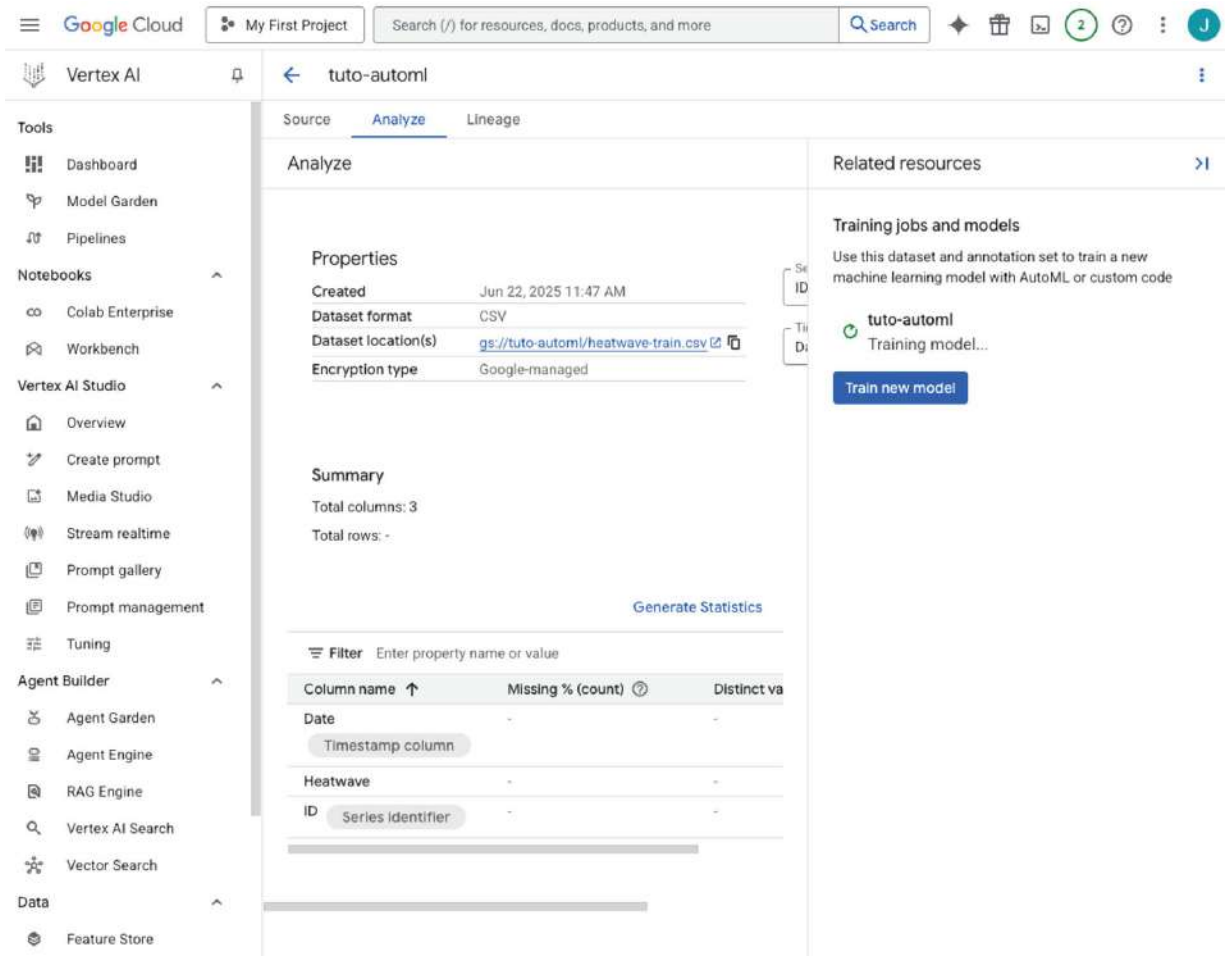


Figure 27-14 The job starts running

In the training page (Figure 27-15), you can follow up with the model. As we have specified 1 hour, now we need to wait.

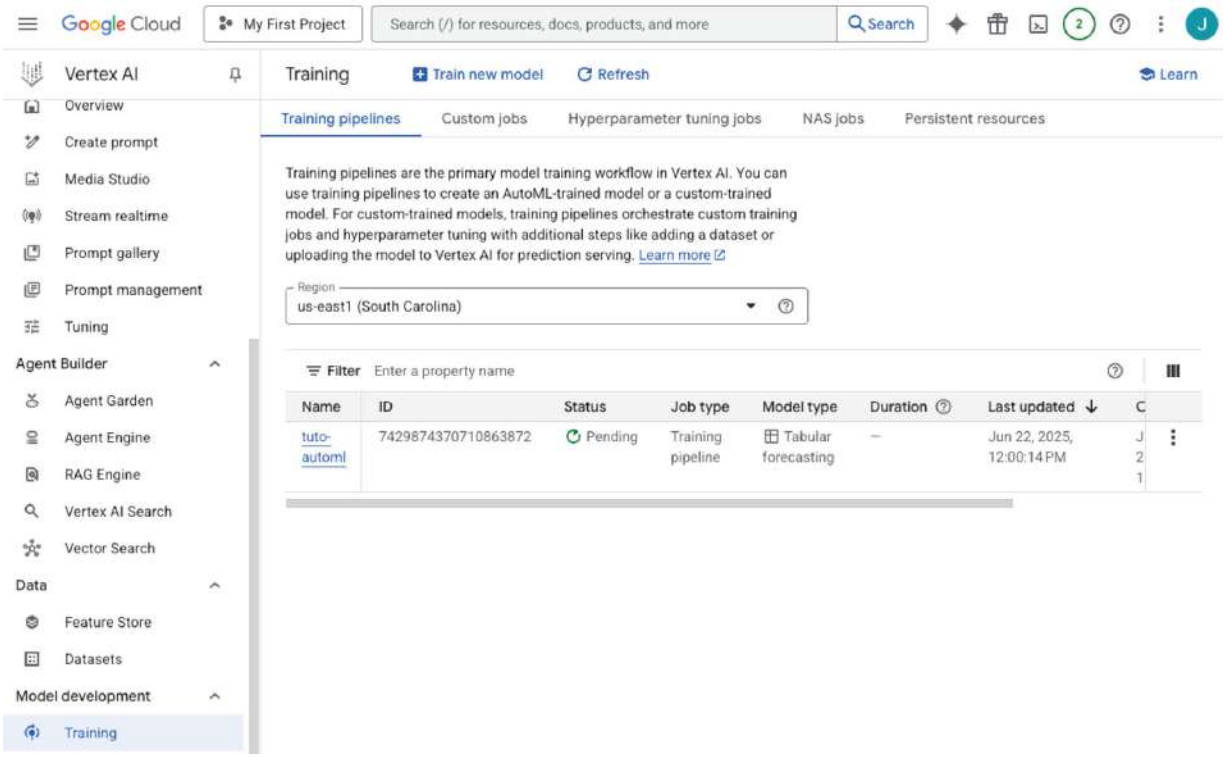


Figure 27-15 The training page

Even if you set the maximum time to 1 hour, GCP will likely take between 1h30 and 2 hours to finish. Take this into account when choosing your setup. When your job is finished, you'll be able to see your model in the model registry, as shown in Figure 27-16.

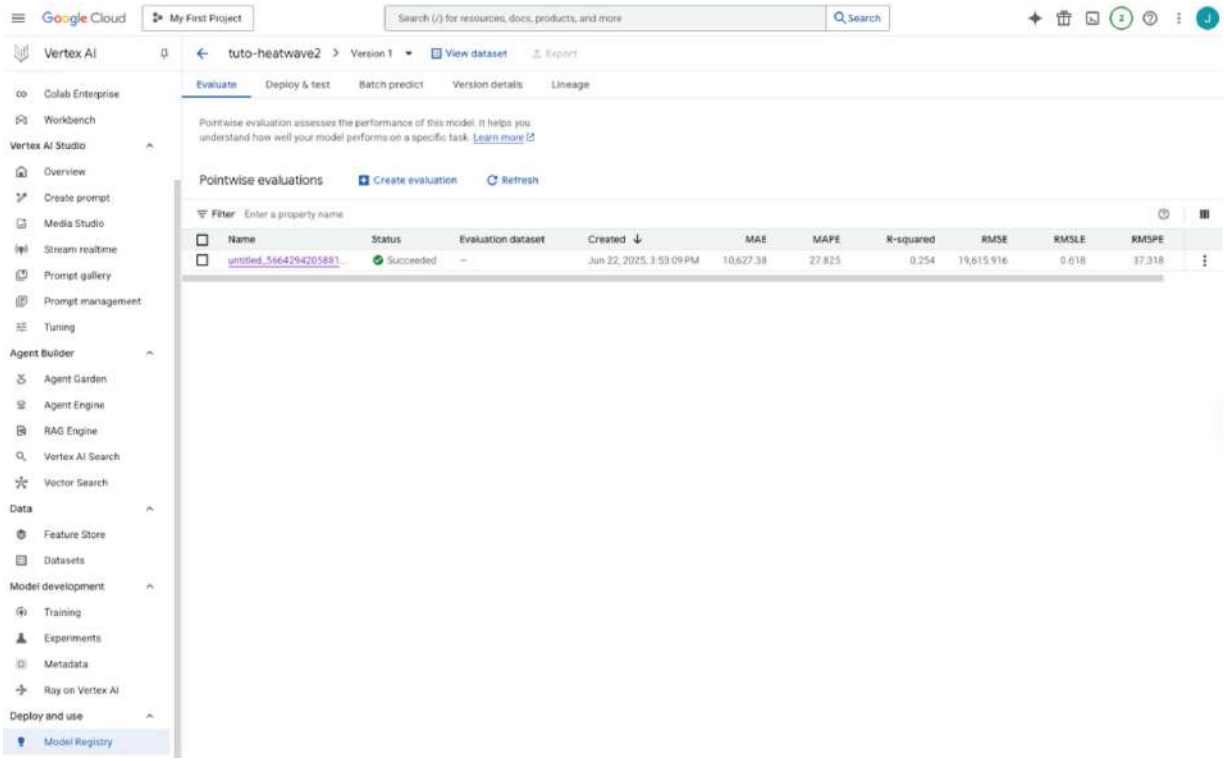


Figure 27-16 The model registry

When you click on it, you can see Metrics and Feature importance, as shown in Figure 27-17.

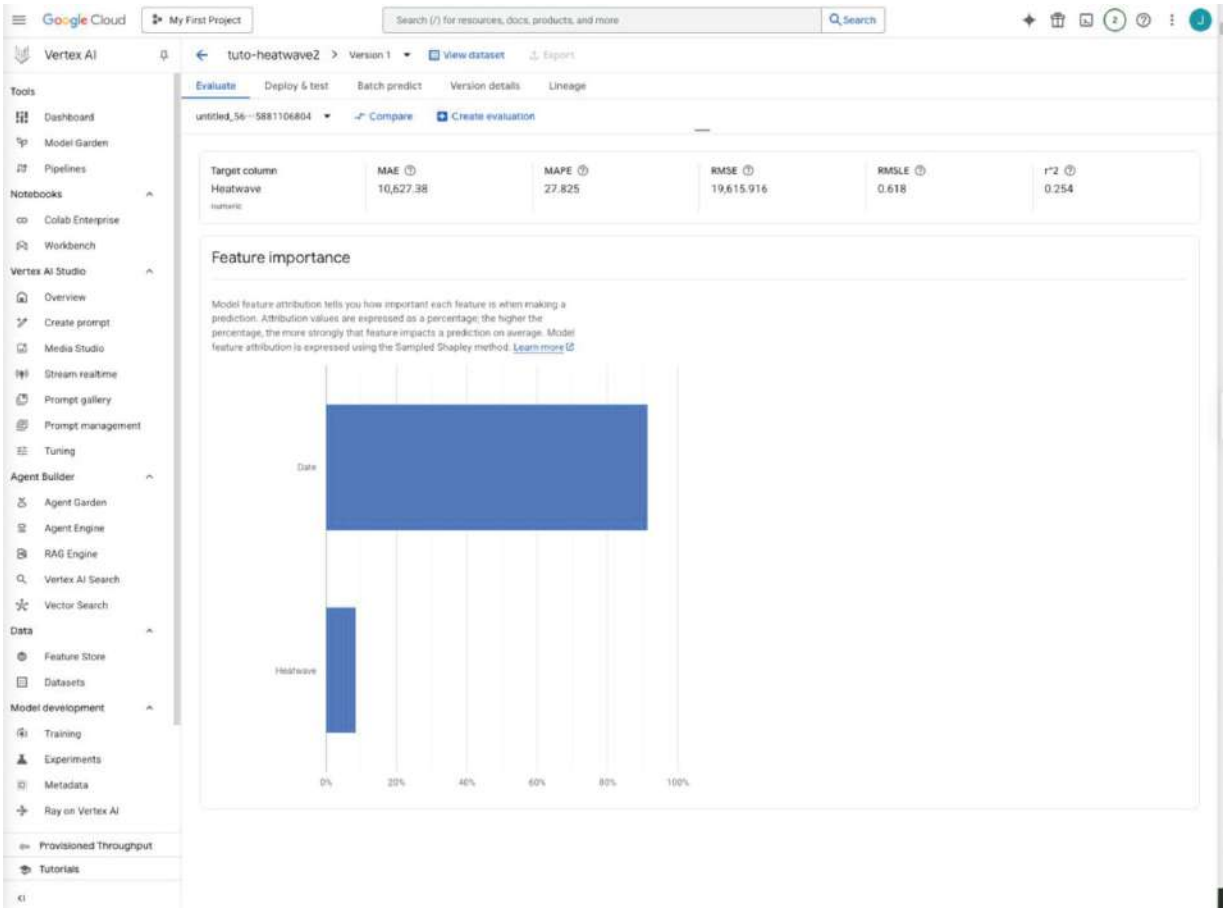


Figure 27-17 The details of our trained model

In this result, you will see that there is a big difference between Azure and GCP. In Azure, we saw a list of models that were tested and benchmarked. In GCP, it is impossible to know which model is hiding behind AutoML. Whatever model it is, the tutorial is successful: you have successfully created and trained a model through GCP AutoML. As you can see, in working with cloud, we often need to focus more on setup and interaction between services than that we spend time on actually building statistical models.

Optional Next Steps

You have successfully created and trained a model through GCP AutoML, but what should you do with it? It would go above the scope of this book to dive into cloud architecture. If you are interested in building out this use case further, I recommend deciding on a deployment architecture. You can choose to use GCP to build an endpoint for real-time scoring (Figure 27-

18). Of course, if you go this route, you need to make sure that your endpoint is secured and that your traffic is managed correctly (API and firewall settings, etc.) to be configured correctly.

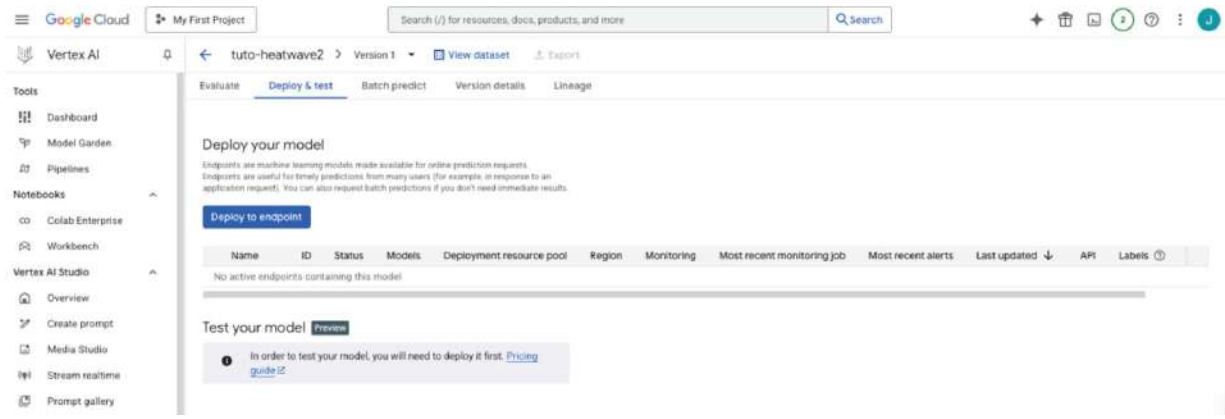


Figure 27-18 Real-time Scoring menu

Another option is to try and build a Batch Predict job. This will take the model and convert it into a prediction job automatically. You can then create a test dataset using the technique that we have seen and send this data to your Batch Predict job to generate forecasted predictions. This can be done using the menu shown in Figure 27-19.

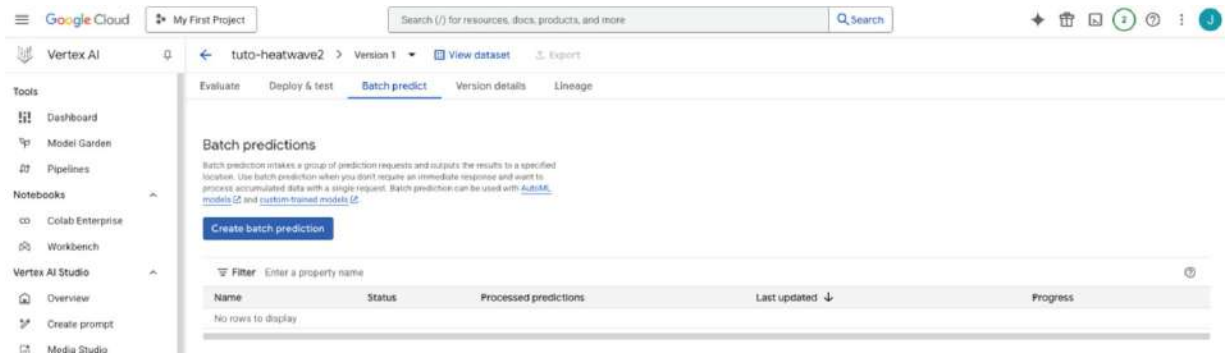


Figure 27-19 Batch prediction menu

Attention We are now finished with GCP. If you have followed along with this chapter, make sure to shut down all resources and end your subscription if you want to avoid costs.

Key Takeaways

- GCP is the cloud computing service from Google.
- AutoML in Vertex AI is one of the many ML services provided by GCP.
- Inside Vertex AI AutoML, there is a specific category for time series forecasting.
- The Vertex AI AutoML is a black box: we do not know what mathematical model is built for us.
- Once you finish training the AutoML model, GCP will allow you to use it for numerous other services that exist on the platform, including the creation of a Real-Time Endpoint or a Batch Predict job.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_28

28. Nixtla Suite and TimeGPT

Joos Korstanje[1]
(1) Maisons Alfort, France

In this chapter, we're diving into the Nixtla suite, as well as its TimeGPT model. The Nixtla package used to be only an open source package containing a certain number of time series models, but in July 2024, Nixtla changed course and presented TimeGPT.

There is currently strong traction on the AI market for LLMs and Generative AI models, like ChatGPT and the like. As we have seen in previous chapters, the underlying mathematical models used for treating human-readable text sequences are often usable or adaptable to time series as well. After all, the order of words in a sentence is important, just like the order of values in a time series.

At the same time, there is a large impulse in software as a service and cloud computing. Nixtla has jumped on this train with their TimeGPT model, as they are now not focusing on the open source package anymore but rather are focusing on providing the TimeGPT service behind a paid cloud service.

In this chapter, we'll first showcase the original functionalities of the open source Nixtla package and then move on to do a use case with the new TimeGPT.

The NixtlaVerse

Nixtla's open source proposition is not just one package, but rather it proposes a list of packages. The NixtlaVerse is the name for the combined set of packages proposed by Nixtla. Among others, it contains the following packages:

- **statsforecast** for statistical forecasting
- **mlforecast** for machine learning-based forecasting
- **neuralforecast** for a number of machine learning models based on neural networks

You can consider the NixtaVerse as an ecosystem of forecasting models.

Simple Use Case with the NixtlaVerse

To begin exploring the NixtlaVerse, let's showcase how to do data prep and a general model with the Nixtla Suite. To follow along, you can download the data on Heat Waves from July 2015 to May 2025, in CSV format, which you can do through this link: https://pageviews.wmcloud.org/?project=en.wikipedia.org&platform=all-access&agent=user&redirects=0&start=2015-07&end=2025-05&pages=Heat_wave. Listing 28-1 shows how to import this data into Python.

```
import pandas as pd
y = pd.read_csv('heatwave-pageviews-20150701-
20250531.csv')
y.columns = ['date', 'y']
y['date'] = pd.to_datetime(y['date'])
y = y[:-5]
y.head()
```

Listing 28-1 Importing the data

You will obtain the data as shown in Figure 28-1.

	date	y
0	2015-07-01	26379
1	2015-08-01	18584
2	2015-09-01	12529
3	2015-10-01	15343
4	2015-11-01	13545

Figure 28-1 The data

As an example, we're going to build a simple AutoArima model on the Heat Wave Page Views. To do so, it will be necessary to adapt the data to the NixtlaVerse. This means that our data needs three columns:

- unique_id: Always 1 in our example, as we only have one time series
- ds: The datetime column
- y: The target (the time series values)

This data preparation is done in Listing 28-2.

```
y["unique_id"]="1"
y.columns=["ds", "y", "unique_id"]
y.head()
```

Listing 28-2 Adapting the data to the NixtlaVerse

You'll obtain the data as shown in Figure 28-2.

	ds	y	unique_id
0	2015-07-01	26379	1
1	2015-08-01	18584	1
2	2015-09-01	12529	1
3	2015-10-01	15343	1
4	2015-11-01	13545	1

Figure 28-2 The prepared data

You can use Listing 28-3 to apply a train-test split.

```
train = y.iloc[:-12]
test = y.iloc[-12:]
```

Listing 28-3 Train-test split

Let's now start building the model. This is shown in Listing 28-4.

```
import scipy.stats as stats
from statsforecast import StatsForecast
from statsforecast.models import AutoARIMA
from statsforecast.arima import arima_string

# Specify the list of models and season length
models = [AutoARIMA(season_length=12)]

# Instantiate the Statsforecast, freq MS for month
start
arima = StatsForecast(models=models, freq='MS')

# Fit the auto arima on the training data
```

```
arima.fit(df=train)

# Print selected arima
arima_string(arima.fitted_[0,0].model_)
```

Listing 28-4 Fit AutoArima using Statsforecast

Once you have built this model on the train set, you can use Listing 28-5 to predict the forecast using this model.

```
fcst = arima.forecast(df=test, h=12, fitted=True)
fcst
```

Listing 28-5 Predict on test set

The forecast is not just the values, but rather a dataframe containing three columns:

- unique_id: Always 1, because we have just one time series
- ds: The datetime of the forecast step
- AutoARIMA: The forecasted value by the AutoARIMA

This result is shown in Figure 28-3.

	unique_id	ds	AutoARIMA
0	1	2025-01-01	8484.261719
1	1	2025-02-01	7128.412109
2	1	2025-03-01	5989.237305
3	1	2025-04-01	5032.111328
4	1	2025-05-01	4227.941406
5	1	2025-06-01	3552.283691
6	1	2025-07-01	2984.601562
7	1	2025-08-01	2507.639404
8	1	2025-09-01	2106.899414
9	1	2025-10-01	1770.200684
10	1	2025-11-01	1487.309082
11	1	2025-12-01	1249.625732

Figure 28-3 The prediction

We can also compute the 1-MAPE metric. This is shown in Listing 28-6.

```
from sklearn.metrics import
mean_absolute_percentage_error

actual_values=test['y'].values
fcst_values=fcst['AutoARIMA'].values

mape=mean_absolute_percentage_error(actual_values,
fcst_values)
mape
```

Listing 28-6 Compute metric

The model has obtained a 1-MAPE of 0.75. You can use Listing 28-7 to show how this looks in a forecast plot.

```
import matplotlib.pyplot as plt

plt.plot(actual_values)
plt.plot(fcst_values)
plt.legend(['test', 'forecast'])
plt.show()
```

Listing 28-7 Forecast plot

This code will result in the plot in Figure 28-4.

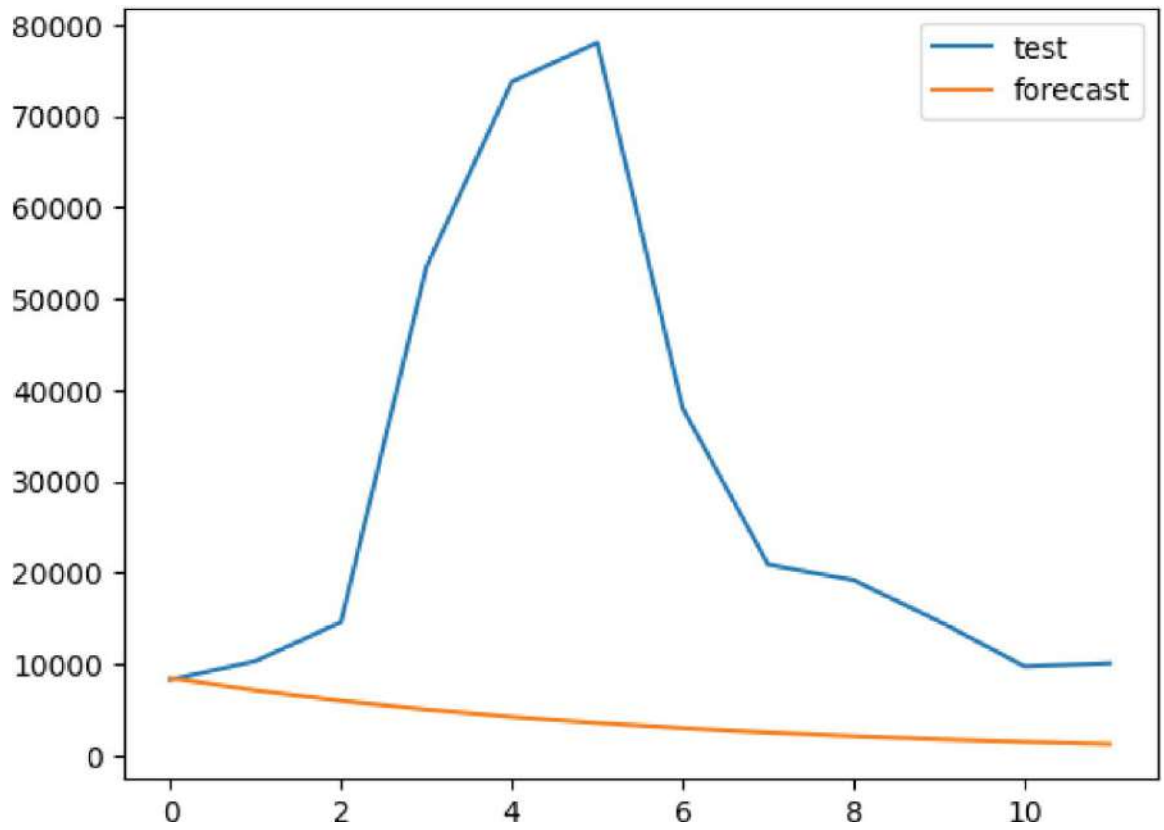


Figure 28-4 The forecasting plot

The GPT in TimeGPT

TimeGPT is based on the same GPT technology as ChatGPT. GPT is short for Generative Pre-trained Transformer, which is a neural network architecture. TimeGPT, like ChatGPT and GPT in general, is based on the transformer model, which we have seen in a previous chapter.

TimeGPT is pre-trained, meaning that it has been fed a large amount of time series data, in order to have learned general tendencies in forecasting datasets.

Nixtla's TimeGPT API

Nixtla's TimeGPT is a service behind an API, which can be called easily through a Python SDK. In order to use the paid TimeGPT model, you need to create an account and generate your API key on the Nixtla website. You can then use a Python notebook using the code in Listing 28-8 to set up the API connection.

```
from nixtla import NixtlaClient
nixtla_client = NixtlaClient(api_key = '...')
```

Listing 28-8 NixtlaClient

After that, using the model is very simple: the code in Listing 28-9 shows you how to.

```
model = nixtla_client.forecast(
    df=train,
    model="timegpt-1",
    h=12,
    freq="MS",
    time_col="ds",
    target_col="y",
)
```

Listing 28-9 Call the API

You will then obtain the same prediction dataset as you have seen in Figure 28-3, generated by the GPT behind Nixtla's TimeGPT API.

Attention We are now finished with Nixtla. If you have followed along with this chapter, make sure to shut down all resources and end your subscription if you want to avoid costs.

Key Takeaways

- Nixtla is an ecosystem of forecasting tools, including
- statsforecast
  - mlforecast
  - neuralforecast
- TimeGPT is an innovative paid service proposed by Nixtla, which is available through an API.
- TimeGPT is based on the Generative Pre-trained Transformer architecture, which in turn is based on the Transformer model.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2025
J. Korstanje, *Advanced Forecasting with Python*
https://doi.org/10.1007/979-8-8688-2028-1_29

29. Model Selection

Joos Korstanje[1]
(1) Maisons Alfort, France

In the last 18 chapters, you’ve seen a long list of Machine Learning models that can be used for forecasting. In this chapter, you’ll find some final reflections on how to decide which of the models to use for your practical use cases.

Model Selection Based on Metrics

The book started with an overview of model metrics. Throughout the different chapters, a number of model benchmarks have been done based on R2 scores. This was done firstly in the chapter on Gradient Boosting models, where XGBoost performances were benchmarked against LightGBM performances. Another benchmark was presented in the chapter on Recurrent Neural Networks, between the SimpleRNN, GRU, and LSTM.

The benchmarks presented in this book were based on metrics. When using this approach, there are two general approaches. A first approach is to build a train and a test dataset. You then tune and optimize a number of models on the train data, after which you compute the R2 score on the test data. The model that performs best on the test data is selected.

The second common way to do metrics-based model selection is to use cross-validation. When you have little data, it can be costly to keep out data for a test set, and cross-validation can be a solution to have metrics-based model selection. Another advantage of using cross-validation is that each model is trained and tested multiple times, so randomness is less influential. After all, the test could be favorable to a certain model, purely due to the random selection process and bad luck.

Yet, in practice, there are other things than metrics to consider when choosing the right model for your use case. Let's see a few of them.

Model Structure and Inputs

A second thing to think about when selecting models is that it needs to fit with the underlying variation of your dataset. For example, it does not make sense to use a model with autoregressive components if you know empirically that there is no autoregressive effect in your use case. Let's do a quick recap of the type of components that can be modeled in the different chapters of this book.

We started with **univariate time series models**, which allow forecasting one variable based on past variation in itself. Those models are useful when you either don't have any other data to use in the modeling process or, more importantly, when there really is **a time series component** (like autoregression, moving average, etc.) present in the data.

After that, we covered **Multivariate Time Series models**, which do the same thing but applied to multiple time series at the same time. They have an advantage when you need to forecast multiple time series at the same time, as you have to build only one model for multiple time series rather than building a model for each. Also, it can benefit from shared variation between the time series and have an advantage in performance.

Then, you've seen how to use a number of classical **supervised machine learning models** for forecasting. From a high level, those models all work in the same way. They convert a number of input variables into a target variable. To add seasonality to those models, you can convert the seasonality information into input variables. This type of model is great if you have not just a target variable, but you also have other information about the future that you can use for forecasting, for example, using the number of restaurant reservations when trying to forecast restaurant visits.

We then moved on to using multiple types of **Neural Networks**. The first type of Neural Network that we saw was a **Dense Feed Forward Neural Network**. This neural network is really similar to the way that classical supervised machine learning models work: there are input variables and one (or multiple) target variable(s).

The other Neural Networks that we have seen were **Recurrent Neural Networks.** They are quite different as they are made to learn and predict

sequences of data. The input data for RNNs have therefore to be prepared as sequences. You can also add external variables to RNN models.

Neural Networks can be very performant when parametrized well. Yet, tuning and parametrizing a neural network can be hard: it takes a lot of time from the developer but also computation time to get to a good result.

Lastly, you've seen two modeling techniques from **large technology companies**. You have seen Facebook's Prophet and Amazon's DeepAR. Those models are generally using neural networks under the hood. Those approaches try to automate complex technology with easy-to-use interfaces.

In some ways, this works: it is easier to tune Prophet or DeepAR than to tune a neural network. In other ways, there is still work to do: we have seen that tuning is still necessary to obtain a great result. It is therefore not totally automated.

Prophet and DeepAR are also able to learn both seasonal components and additional data. Prophet is not able to model multiple time series at once, whereas that is possible with DeepAR.

One-Step Forecasts vs. Multistep Forecasts

Related to the question of models being able to use additional regressors is the question of models being able to do multistep forecasts. As we've seen, there are basically three categories of models.

The first category of models simply can not do forecasts of multiple steps. This is the case for models that need the latest data point each time you want to predict the next data point.

The second category is when the model can do a multistep forecast by iterating over the forecasted values and redoing one-step forecasts every time. In this case, it is taking the forecasted value as a past data point at the risk of adding up more and more errors.

The third category can easily learn multistep forecasts. In some cases, this is by doing one multivariate forecast in which you treat each time step as a new dependent variable. An alternative is to learn relations between input sequences and output sequences.

Model Complexity vs. Gain

Besides metrics and the type of variables that can be used by a model, I want to come back to model complexity. For some models, it takes a long time to obtain a slight improvement in R2. You should wonder if, for your use case, there is a way to say that a certain improvement of accuracy is still worth it. For example, spending a year for 0.001 points of R2 improvement is probably not worth it when doing sales forecasting, yet it may be totally worth it when doing medical work or aircraft safety or things for the army.

You must consider that model tuning costs time for a modeler, but it also costs computing time, which can become expensive when running, for example, cloud computing setups for very long periods.

This type of consideration can be seen when we moved from classical supervised models to Neural Networks. Neural Networks are very performant and can obtain great results. It is much more complex to parametrize and optimize a neural network architecture than to do a hyperparameter tuning of, for example, a gradient boosting. In short, you would need to have an idea of the impact of your R2 score to be sure how perfect your forecast still has a practical impact on your use case.

Model Complexity vs. Interpretability

A second problem with complex models is that it often becomes very difficult to understand what the models have learned. For univariate time series, or for classical supervised models, it is relatively easy to extract the model coefficients or to use other tools for model interpretation, like printing a decision tree or extracting variable importance.

For more complex models, including multivariate time series, but mainly neural networks, it is much harder to go through huge coefficient matrices and understand in humanly understandable terms what it is that the model has learned.

For Prophet and DeepAR, model interpretability is even worse, as the black box approach of those models has put the model builder even further away from the actual technicalities of what has been learned.

Model interpretability is important. As a forecaster, you need to be confident in your model. To be confident, you should at least be totally sure of what it does. This is often not the case with complex models and can lead to very difficult situations.

Model Stability and Variation

As another reflection for choosing a model for your forecasts, we can come back to the question of model stability. In the chapter for DeepAR, we have observed that DeepAR is great at giving a confidence region in which we may expect our model to be, but that when repeating the model training, it is generally not capable of reproducing the same result. This problem of model stability may be a serious problem when you are trying to make a reliable forecast.

Model variation is also important when talking about overfitting models. We have seen that when models overfit, they can get to learn the training data by heart, and therefore, they are not learning a general truth anymore, but rather, they are learning the noise of the training data.

This is also a type of variation: when you expect a great result based on your modeling process, but you obtain bad results when forecasting in practice, this may be a variation in metrics that is due to overfitting. In a way, it is also a model metric. It may not be the one we generally talk about first, yet it is still a very important concept to keep in mind when moving from the development phase to the actual forecasting in the field.

Open Source vs. Cloud Services Black Box

Finally, in this second edition of the book, you have also seen numerous cloud services and black box models. Using these services can sometimes result in improved metrics. However, for model selection, it is important to take into account that there are a few potential drawbacks of these services.

Lack of explainability is one important consideration for these cloud-based models, but this has already been covered above. A second important thing to consider when using the cloud is the cost. Running cloud-based models is expensive. If you are able to write the code for the algorithms, depending on your datasets, you may be able to run them locally as well. At the same time, in a professional setting, maintaining a local environment is not free either.

Finally, it is important to consider vendor lock-in. As you have seen, when building a model on one specific cloud, the cloud provider will require you to have the data stored in their cloud and adapt your data to their format. If you do this kind of developments, you are building a

solution which is specifically adapted to this cloud. You might encounter vendor lock-in, as it would take a lot of work to get your stuff out of this cloud and onto another cloud. There is little to no interoperability between cloud providers.

Conclusion

Throughout the book, you have seen how to manage model benchmarks, and you have seen different techniques for model evaluation, including the train-test set, cross-validation, and more. Until here, model performance has been generally defined by a metric, like the R2 score. Those metrics show how good a model performs on average.

The average performance is generally the first thing to consider, but in this chapter, you have seen a number of additional reflections on model selection that are also important to have during the model development phase.

Having gone through the first 20 chapters of this book, you should have all the necessary input for making performant forecasts using Python. The general reflections posed in this chapter should help you to avoid some easily made yet important mistakes.

After all, when you make a forecasting model, the most important thing is not whether it works only on your training data and not even whether it works on your test data. When applying in practice, all that matters is whether your prediction for the future comes true.

Key Takeaways

- Model metrics generally tell us whether a model is good on average.
- Model variation tells us whether there is a lot of variation in a model's quality metric, for example, if we retrain it on a slightly different dataset.
- More complex models can sometimes give better model metrics, but they often come at a cost of time spent on modeling, computing costs, and a cost of lower model interpretability.
- When selecting a model type, you should take into account whether the model you use is capable of fitting the types of effects that you would theoretically expect to be present in your dataset, including the possibility

of adding extra regressors and the possibility of doing multistep forecasts.
- When using cloud-based black box models, it is important to take costs and vendor lock-in into account.

Index

A

ACF
See Autocorrelation function (ACF)
AdaGrad 239
ADF
See Augmented Dicky–Fuller (ADF) test
Aggregation 193
AIC
See Akaike Information Criterion (AIC)
Akaike Information Criterion (AIC) 151, 152
Anaconda Navigator 43
Argmin 83
ARIMA model 101, 119
ACF and PACF plots 105
AR and MA models 101
automatic differencing, non-stationary time series 120
on CO_2 example
ACF and PACF plot 122, 124
autocorrelation function plot 123, 124
hyperparameter tuning 125
import data 121
partial autocorrelation function plot 123, 125
R2 scores 127
statsmodels library 121
fit ARMA(1, 1) model 107, 108
grid search 112–115
hyperparameter optimization 111
integrating 120
linear trend 120
mathematical definition 102
model definition 120
model evaluation KPIs 108–111
normal distribution 109
predict births 103–107

save models using MLflow 116, 117
stationarity 104, 120
AR
See Autoregressive (AR) model
Artificial Intelligence 3
Attention 304
"Attention Is All You Need" 304
Augmented Dicky–Fuller (ADF) test 60, 61, 84, 87, 88
Autocorrelation function (ACF) 64, 65, 105, 106, 108, 112
Automated ML
create 382
create data asset 384
job 391
preview, data 387
results 392
Schema menu 388
Select Azure Blob Storage 386
select data type tabular 384
select task type, time series forecasting 383
set up data import from local files 385
set up the compute 390
set up the limit 390
task settings 389
task type & data 383
upload data 386
AutoML 380, 396
Autoregression 169
Autoregressive (AR) model 53, 54
autocorrelation 54
lags 63–65, 67, 68
partial 65–67
positive and negative 59
compute autocorrelation in Earthquake counts 54–58
differenced data 61–63
differencing, time series 61–63
estimate coefficients, using Yule–Walker method 69–73
estimating the coefficients 68

fit AutoReg with MLflow logging 78, 79
grid search 75
mathematical definition 68
non-stationarity, time series 59
order of the time series 67
predictive performance 54, 68
to SARIMA model 53
stationarity, time series 59
train, test, evaluation and tuning 73–77
univariate time series 53
AWS SageMaker AI
create batch transform job 357, 358
DeepAR 351
deploy model 357
hyperparameter configuration 355
input data configuration 354
model details 357
S3 bucket 352
training job 353
train.json.out file 358
Azure Cloud 380
Azure Machine Learning Studio 381

B

Backpropagation algorithm 239, 340
Backtesting 37, 38
Bagging 192, 193, 204, 205
Bayesian forecasting 221, 223, 224
Bayesian inference 361
Bayesian model 221, 223, 361
Bayesian modeling package 361
Bayesian Optimization 213, 214, 221
applied to CatBoost 217
applied to LightGBM 215, 216
applied to XGBoost 214
concept 213
GridSearchCV 213

intelligence 213
testing values 213
use scikit-optimize package 214
Bayesian *vs*. frequentist statistics 222, 223
BigQuery 395, 396
Boosting 205
and bagging 205
ensemble model 206
gradient boosting
See Gradient boosting
iterative process 205
process 206
Bootstrap 192, 193
Buckets 398, 399

C

CatBoost 207, 208, 218, 285
Bayesian optimization 217
categorical variables 208
default CatBoost model 211
forecasting traffic volume 211
vs. XGBoost and LightGBM 208
ChatGPT 415, 420
Cloud-based black box models 427, 428
Cloud-based Maths to AutoML 380
Cloud computing 379, 380, 393
Common bias 28
Correlation coefficient 15, 16, 19, 64
Correlation matrix 16, 58, 65, 142
Cross-validation 32, 423

D

Damped Local Trend (DLT) 361, 369–371
Darts 285, 303, 307, 308, 310, 319
Darts forecasting package 288
Darts package 285, 288, 289, 301, 305
N-BEATS model 285
Decision Tree model

example 171
gradient boosting
See Gradient boosting
if/else statements 171
mathematics
example 173–179
pruning and reducing complexity 172
splitting 172
plot 179
Random Forest
See Random Forest
DecisionTreeRegressor 177
DeepAR 361, 375, 425, 426
advantage 343
Amazon 343, 359
vs. AWS SageMaker AI 351
Facebook's Prophet 343
RNNs 343
seasonality variables 343
DeepAR algorithm 380
Default model 331–334
Dendrogram 179, 180
Dense Feed Forward Neural Network 424
DGLM
See Dynamic Generalized Linear Model (DGLM)
DLT
See Damped Local Trend (DLT)
Dynamic Generalized Linear Model (DGLM) 223, 224, 229, 232, 233

E

EFB
See Exclusive Feature Bundling (EFB)
Epochs 240, 241, 247, 249
Error criterion 177, 178
Euclidean distance 181, 189
Exclusive Feature Bundling (EFB) 207
Exogenous variables 143, 153–157

Exponential Smoothing 361

F

Facebook 321, 322, 341, 379
Facebook Prophet
  building forecasting models 341
  DeepAR 343
  goal 321
Feature engineering 164, 169, 174, 185
Feature importance 201–203, 410
Feed Forward Neural Networks 263, 284, 304
Forecasting 3
  AI 3
  Machine Learning
    See also Machine Learning models
  model evaluation 3
    See Model evaluation
Forecasting Heat Wave Page Views
  DeepAR training job, AWS SageMaker AI
    data preparation 351
    SageMaker AI 352–359
    upload the Data to S3 352
  GCP's Vertex AI AutoML
    batch prediction menu 412
    bucket creation 398, 399
    data preparation, GCP 396, 397
    data upload 399–411
    real-time scoring menu 412
  Microsoft Azure Cloud AutoML
    Automated ML 382–391
    Azure Machine Learning Studio 381, 382
    download MLflow artifact 392, 393
  Orbit
    DLT model 369–373
    LGT model 362–366
Forecasting model 4, 19, 29, 32, 37, 38, 54, 229, 305, 428
Frequentist statistics 222, 223

G

Gated Recurrent Unit (GRU) 255
architecture 265
cell 265, 269, 271
hidden layers 267, 268
history
one-layer GRU model 266
three-layer GRU model 268
performance 272, 276, 278
GCP
See Google Cloud Platform (GCP)
Generative AI models 415
Generative Pre-trained Transformer (GPT) 420, 421
Google Cloud Platform (GCP) 395, 396, 411, 413
GOSS
See Gradient-based One-Side Sample (GOSS)
GPT
See Generative Pre-trained Transformer (GPT)
Gradient-based One-Side Sample (GOSS) 207
Gradient boosting 205
algorithms 206
CatBoost
See also CatBoost
differences in predictions 207, 212
hyperparameter tuning using Bayesian Optimization
See Bayesian Optimization
LightGBM
See also LightGBM
XGBoost 206, 206
See also XGBoost
Gradient Boosting models 205, 209, 212, 423
Grid search 75, 96, 112–114, 116, 187, 188, 221, 312
GRU
See Gated Recurrent Unit (GRU)

H

Heat waves prediction, NeuralProphet

data preparation, Python 323, 324
default model 332
forecast dataframe 332, 333
forecast evaluation 333
forecast plot 334
time series decomposition 331
MLflow 339–341
time series decomposition
advanced decomposition setting 327, 328, 330
decomposition 326
identify trends 325
linear trend 327
prepared dataset 325
seasonality 325
settings 326
yearly seasonality 327
train-test splits 330, 331
tuned model 335–337, 339
Wikipedia 322
Wikipedia Pageview Tool 322, 323
Hyperparameters 75
optimization tool 240
tuning 184, 189

I

Interpretability 284, 426, 428

J

Jena Climate dataset 241
Jupyter Notebooks 50, 396

K

Keras 247, 248, 260
K-fold cross-validation 32–34
k nearest neighbors 182
kNN model
the average 183
definition 181

grid search 187, 188
hyperparameter 184
hyperparameter tuning 184
nearest neighbors 181
Random Search 188, 189
traffic prediction 184–186
Kronecker delta function 69

L

Lags, autocorrelation 63–66
Large technology companies 425
LGT
See Local-Global Trend model (LGT)
LightGBM 207, 218, 285
Bayesian optimization 215–217
default LightGBM model 211
EFB 207
forecasting traffic volume 211
GOSS 207
gradient boosting algorithm 206
leaf-wise *vs*. level-wise growth 207, 208
performances 423
R2 score 211
vs. XGBoost 207, 208, 212
Linear model 163, 169, 223
Linear Regression
coefficients BETA, training dataset 163
definition 162
forecast CO_2 levels
add lagged variables 167
autoregressive component 167
code 165
drop missing values 167
extracting seasonal variables 164
feature engineering 164
import data 163, 164
over time 164

package scikit-learn 165
predictive performance plot 166, 168
train R2 score 168
nonlinear process 163
OLS 163
visual interpretation 162
Local-Global Trend model (LGT) 361, 363, 368
Long short-term memory (LSTM) 255
advantage 279
cell 271, 272
definition 271
example 272, 273
1 layer of 8 274–276
3 layers of 64 276–278
training history
one-layer LSTM 276
three-layer LSTM 278
Lowest Mean Squared Error 172
LSTM
See Long short-term memory (LSTM)

M

MA
See Moving Average (MA) model
Machine Learning models 4
classification *vs*. regression models 17, 18
coffee prices example 6
linearly increasing trend 5
supervised machine learning
See Supervised machine learning models
supervised *vs*. unsupervised models 17
time series models 7
hot chocolate 7–9
univariate time series 4
univariate *vs*. multivariate models 18
MAE
See Mean Absolute Error (MAE)

MAPE
See Mean Absolute Percent Error (MAPE)
max_features 177, 178, 194, 197–199
maxlags 151, 152
Mean Absolute Error (MAE) 26, 27, 247, 335
Mean Absolute Percent Error (MAPE) 27, 230
Mean Squared Error (MSE) 24, 25, 39, 107
Metrics-based model selection 423
Microsoft 380, 395
Microsoft Azure 380, 391, 396
Microsoft Azure Cloud AutoML
forecasting heat wave page views
See Forecasting heat wave page views
MinMax scaler 243, 253
min_samples_split 177, 178
MLflow 41, 78, 79, 116, 117, 176, 339–341, 393
artifacts directory 46
autologging 44, 232
data 45
experiments interface 48
INFO message 44
interface 177
local *vs*. hosted 42
metrics directory 46
MLflow UI 48, 49
model code 43
open source tool 42
Params 47
run Python session 47, 48
setup and installation 42
tags 47
tracking 44, 50
UI 340
!mlflow ui 48, 177, 340
mlforecast 415, 421
Model complexity
vs. gain 425, 426

vs. interpretability 426
Model evaluation
add errors, forecasted value 23, 24
error metrics
MAE 26, 27
MAPE 27
MSE 24, 25
R2 26
RMSE 25
fitting 39
get stock data example into Python 22
stock prices data 21, 22
stock prices *vs*. forecasted stock prices over time 23
Model evaluation strategies
backtesting 37, 38
combined strategy 38
cross-validation 32
K-fold 32, 33
rolling time series 36, 37
time series 34–36
overfit model 28
train-test split 28, 29
train-validation-test split 30–32
Model interpretability 426, 428
Model metrics 428
Model selection
metrics 423
model complexity *vs*. gain 425, 426
model complexity *vs*. interpretability 426
model stability and variation 426
model structure and inputs 424, 425
one-step forecasts *vs*. multistep forecasts 425
open source *vs*. cloud services black box 427
Model stability 426–427
Model variation 427, 428
Moving Average (MA) model
application 90–98

and AR model 82
fit MA model 83
grid search 96, 97
indicators 84
MLflow autologging 98
model definition 82
model error, past predictions 82
SARIMA/SARIMAX model 81
as standalone model 81
stationarity 84
typical AR model PACF 85
typical MA model ACF 85
MSE
See Mean Squared Error (MSE)
Multistep forecasting 94, 132, 257
Multivariate Least Squares 148, 149
Multivariate models 18, 147
Multivariate N-BEATS model 293, 295
Multivariate Normal DGLM 300, 318
Multivariate time series 147, 153
Multivariate time series models 151, 153, 154, 424
Multivariate transformer model 310–313

N

Nan value 58
N-BEATS model 317
Darts package 285
disadvantages 301
hyperparameters 283
intuition and mathematics
automatic feature engineering 284
decomposition 284
interpretability 284
stacking 284
Neural Network architectures 283
sales forecast
See Sales forecast

Negative autocorrelation 56, 59, 65
n_estimators 193, 194, 197–200
neuralforecast 415, 421
Neural networks 237
  activation functions 238, 239
  backpropagation algorithm 239
  example data 241
  fully connected neural networks 237, 238
  hyperparameters 240, 241
  learning rate of optimizer 240
  MinMax scaler 243
  optimizers 239, 240
  PCA 243–246
  Recurrent Neural Networks 237, 239
  standard scaler 243
  use Keras 247, 248
Neural Networks 424, 426
NeuralProphet model
  Facebook 341
  heat waves prediction
    See Heat waves prediction, NeuralProphet
  performance and flexibility issues 322
  Prophet and Neural Networks 322
  simplicity 322
Nixtla 415, 421
Nixtla's TimeGPT API 420, 421
NixtlaVerse
  mlforecast 415
  neuralforecast 415
  simple use case
    adapt the data to the NixtlaVerse 416, 417
    AutoArima using Statsforecast 418
    compute metric 419
    data 416
    forecast plot 419, 420
    import data 416
    prediction 419

predict, test set 418
prepared data 417
train-test split 417
statsforecast 415
Nonlinear Least Squares 83

O

OLS
See Ordinary Least Squares (OLS) method
One-step forecasting 94, 167
One-step forecasts *vs*. multistep forecasts 425
Open source *vs*. cloud services black box 427
Optimizers 239–241
Orbit model
Bayesian forecasting models 361
Bayesian inference 361
Bayesian modeling package 361
benchmark 365, 368, 372, 375
best-tuned DLT 375
forecasting heat wave page views
DLT 369–373
LGT 362–366
LGT 368
tuned forecast plot 369, 376
tuned LGT 366, 368
tune DLT model 373, 374
Uber 376
Order 148
Ordinary Least Squares (OLS) method 70, 163

P, Q

Params 47
Partial autocorrelation 65–67
Partial autocorrelation function (PACF) 66, 67, 84, 85, 105, 106, 108, 112
PACF
See Partial autocorrelation function (PACF)
PCA
See Principal Component Analysis (PCA)

Pearson correlation coefficient 58
Picklefile, model.pkl 46
Positional Encoding 304, 319
Positive autocorrelation 59, 61
Principal Component Analysis (PCA) 243–246
Prophet 322, 361, 375, 425
Prophet model 321, 344
pyBATS 317, 318
  Multivariate Normal DGLM 300
  package 223, 224, 229, 233
Python 185, 323, 324
  Notebook 393
  packages 43
Python 3.9 43
PyTorch 305, 319, 322, 345

R

Random Forest
  Decision Tree model 191
  ensemble learning 191, 192
    aggregation 193
    bagging and boosting 192
    bootstrap 192, 193
  feature importance 202, 203
  grid search hyperparameter tuning 196
  predict births 194, 195
  variables 194
RandomizedSearchCV
  use distributions 198
    for max_features 198, 199
    for n_estimators 199, 200
  with two distributions 200, 201
Random Search 188, 189, 221
Recurrent Neural Networks (RNNs) 304, 424
  architecture 255
  cells 256, 269
  data preparation

import data 258
Keras 260
MinMaxScaler 259
sequence data 259
SimpleRNN 260
split train and test 260
temperature data 259
vs. dense network 255
feedback loop 256
GRU 255, 265, 266
hidden layers
GRU 267, 268
SimpleRNN 263, 264
long sequences 269
LSTM 255
ReLU activation layer 257
sequences 257
sequences prediction 257
SimpleRNN 255, 261, 262
SimpleRNN Unit 256
tanh layer 257
univariate model 258
ReLU 238, 239, 257, 269
return_sequences = True 263
RMSE
See Root Mean Squared Error (RMSE)
RMSProp 239
RNNs
See Recurrent Neural Networks (RNNs)
Rolling time series cross-validation 36, 37
Rolling time series split 36, 37
Root Mean Squared Error (RMSE) 25–27, 40
R2 score 423
R squared (R2) metric 26

S

Sales forecast

N-BEATS in Darts
create default model 291, 292
create multivariate model 293, 295
Darts time series format 290
data preparation 286–289
dataset 285
forecasting plot 293
hyperparameter tuning 301
tune multivariate model 295–299
univariate time series models 300
transformer in Darts
create default model 308–310
create multivariate transformer model 310–312
data preparation 306, 307
dataset 305
tuning the multivariate transformer model 312–317
univariate time series models 318
using pyBATS 224
exploratory data analysis 224–229
normal DGLM with X variable 232, 233
univariate Poisson DGLM 229–231
SARIMA model 129, 139
grid search 134
hyperparameters 140
mathematical definition 140
model definition 130
power 129
predictive performance 133, 136
regular MA process 135
seasonality 129, 130
seasonal period 131
vs. supervised models 140
tuning 134
on Walmart dataset 141
correlation coefficient 142, 143
correlation matrix 142, 143
fit SARIMAX model 143

hyperparameter search 143
import and prepare data 141
predictive performance 144
on Walmart sales forecasting dataset 131–135
Seasonality 7, 18, 169, 324, 328
Self-attention 304, 319
SGD
See Stochastic Gradient Descent (SGD)
Sigmoid 238
SimpleRNN 255, 256, 260, 262, 264, 269
cell 271
hidden layers 263, 264
longer-term trends 271
parametrize small network 261
performance 272, 276, 278
training history 262
training history, hidden layers 264
Stacking 284
Standard scaler 243, 353
Stationarity 59–60, 84, 148
Statsforecast 415, 418, 421
Stochastic Gradient Descent (SGD) 239
Supervised machine learning models 424
correlation coefficient 15, 16
correlation matrix 16
direction of the correlation 16
explanatory variables 11–15, 161
quarterly sales example 9–11, 14
strength of the correlation 16
target variable 161
target variable *vs*. explanatory variables 162
Supervised models 17

T

Tags 47
tanh 238, 239, 257, 269
Task type & data 383

TensorFlow 305, 319
TimeGPT 421
GPT technology 420
model 415
Nixtla 420
Time series component 119, 143, 424
Time series models 5, 7, 34–36
Train-test splits 28, 29, 34, 108, 112, 115, 330, 331
Train-validation-test split 30–32
Transformer model 318
intuition and mathematics
"Attention Is All You Need" 304
Darts package 305
Positional Encoding 304
recurrent approach 304
self-attention 304
sequential data 303
Neural Network 303
sales forecast
See Sales forecast
Trend 169, 324
Tuned model 335–337, 339

U

Uber 361
Orbit model
See Orbit model
Univariate models 18, 19
Univariate time series models 4, 53, 54, 59, 77, 81, 139, 222, 424
ARIMA
See ARIMA model
AR model
See Autoregressive (AR) model
MA model
See Moving Average (MA) model
SARIMA model
See SARIMA model

SARIMAX model
See SARIMAX model
Unsupervised models 17

V

Validation split 30, 40
VAR
See Vector Autoregression (VAR) model
VARMAX model
components 153
definition 154
exogenous variables 153
multiple time series, exogenous variables
code 156
order p 155
plot, time series per store 155
prepare Walmart data 154
R2 156
training time 156, 157
variables 156
VARMAX(p, q), hyperparameters 155
multivariate time series modeling 153
Vector Autoregression (VAR) model
definition 147
model definition
order 148
stationarity 148
VAR coefficients, estimation 148
multiple variables 147
one multivariate model *vs*. multiple univariate models 149
time series models 147
Walmart sales 149–152
Vertex AI 396
Vertex AI AutoML
forecasting heat wave page views
See Forecasting heat wave page views

W

Walmart data 131, 132, 141–145
Wikipedia 322
Wikipedia Pageview Tool 322, 323

X

XGBoost 206
  Bayesian optimization 214
  default model 210, 211
  forecaste traffic volume 209, 210
  vs. LightGBM 207, 208, 212
  machine learning algorithms 206
  model 191
  performances 423

Y, Z

ydata-profiling package 55
yfinance package 86
Yule–Walker AR coefficients 70, 77
yule_walker estimation function 78
Yule–Walker method 69