

Hands-on Time Series Analysis with Python

From Basics to Bleeding Edge
Techniques

B V Vishwas
Ashish Patel

Apress®

Hands-on Time Series Analysis with Python

**From Basics to Bleeding
Edge Techniques**

**B V Vishwas
Ashish Patel**

Apress®

Hands-on Time Series Analysis with Python: From Basics to Bleeding Edge Techniques

B V Vishwas
Infosys
Bengaluru, India

Ashish Patel
Cygnet Infotech Pvt Ltd
Ahmedabad, India

ISBN-13 (pbk): 978-1-4842-5991-7
<https://doi.org/10.1007/978-1-4842-5992-4>

ISBN-13 (electronic): 978-1-4842-5992-4

Copyright © 2020 by B V Vishwas and Ashish Patel

This work is subject to copyright. All rights are reserved 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.

Trademarked names, logos, and images may appear in this Book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this Book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: James Markham
Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this Book is available to readers on GitHub via the Book's product page, located at www.apress.com/978-1-4842-5991-7. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Authors.....	xi
About the Technical Reviewer	xiii
Acknowledgments	xv
Introduction	xvii
Chapter 1: Time-Series Characteristics.....	1
Types of Data	2
Time-Series Data.....	2
Cross-Section Data.....	4
Panel Data/Longitudinal Data.....	4
Trend	6
Detecting Trend Using a Hodrick-Prescott Filter.....	6
Detrending a Time Series	7
Seasonality	11
Multiple Box Plots.....	12
Autocorrelation Plot.....	13
Deseasoning of Time-Series Data	14
Seasonal Decomposition	15
Cyclic Variations.....	16
Detecting Cyclical Variations	17
Errors, Unexpected Variations, and Residuals.....	18
Decomposing a Time Series into Its Components.....	18
Summary.....	21

TABLE OF CONTENTS

Chapter 2: Data Wrangling and Preparation for Time Series.....	23
Loading Data into Pandas	24
Loading Data Using CSV	24
Loading Data Using Excel	25
Loading Data Using JSON	25
Loading Data from a URL.....	26
Exploring Pandasql and Pandas Side by Side.....	27
Selecting the Top Five Records	27
Applying a Filter.....	28
Distinct (Unique)	29
IN	30
NOT IN.....	32
Ascending Data Order	33
Descending Data Order	34
Aggregation.....	35
GROUP BY	36
GROUP BY with Aggregation.....	38
Join (Merge).....	39
INNER JOIN	41
LEFT JOIN	43
RIGHT JOIN	46
OUTER JOIN	48
Summary of the DataFrame	51
Resampling	52
Resampling by Month.....	53
Resampling by Quarter.....	53
Resampling by Year	53

TABLE OF CONTENTS

Resampling by Week	54
Resampling on a Semimonthly Basis	55
Windowing Function	55
Rolling Window	56
Expanding Window	57
Exponentially Weighted Moving Window	57
Shifting.....	58
Handling Missing Data.....	60
BFILL.....	62
FFILL.....	62
FILLNA	63
INTERPOLATE.....	64
Summary.....	64
Chapter 3: Smoothing Methods	65
Introduction to Simple Exponential Smoothing.....	66
Simple Exponential Smoothing in Action	68
Introduction to Double Exponential Smoothing.....	76
Double Exponential Smoothing in Action.....	78
Introduction to Triple Exponential Smoothing	86
Triple Exponential Smoothing in Action	87
Summary.....	97
Chapter 4: Regression Extension Techniques for Time-Series Data ...	99
Types of Stationary Behavior in a Time Series.....	99
Making Data Stationery	101
Using Plots.....	101
Using Summary Statistics	101
Using Statistics Unit Root Tests.....	102

TABLE OF CONTENTS

Interpreting the P-value	104
Augmented Dickey-Fuller Test	104
Kwiatkowski-Phillips-Schmidt-Shin Test	105
Using Stationary Data Techniques	106
Differencing	106
Random Walk.....	107
First-Order Differencing (Trend Differencing)	108
Second-Order Differencing (Trend Differencing)	109
Seasonal Differencing	110
Autoregressive Models	111
Moving Average.....	113
Autocorrelation and Partial Autocorrelation Functions	116
Introduction to ARMA	117
Autoregressive Model.....	118
Moving Average	119
Introduction to Autoregressive Integrated Moving Average	119
The Integration (I)	121
ARIMA in Action	122
Introduction to Seasonal ARIMA.....	129
SARIMA in Action	131
Introduction to SARIMAX.....	143
SARIMAX in Action	144
Introduction to Vector Autoregression.....	154
VAR in Action	155
Introduction to VARMA	172
VARMA in Action	172
Summary.....	184

TABLE OF CONTENTS

Chapter 5: Bleeding-Edge Techniques	185
Introduction to Neural Networks	185
Perceptron	186
Activation Function	188
Binary Step Function	188
Linear Activation Function	189
Nonlinear Activation Function.....	190
Forward Propagation.....	195
Backward Propagation.....	196
Gradient Descent Optimization Algorithms.....	199
Learning Rate vs. Gradient Descent Optimizers	199
Recurrent Neural Networks.....	202
Feed-Forward Recurrent Neural Network	204
Backpropagation Through Time in RNN	206
Long Short-Term Memory	210
Step-by-Step Explanation of LSTM.....	212
Peephole LSTM.....	214
Peephole Convolutional LSTM	215
Gated Recurrent Units	216
Convolution Neural Networks	219
Generalized CNN Formula.....	222
One-Dimensional CNNs	223
Auto-encoders	224
Summary.....	226

TABLE OF CONTENTS

Chapter 6: Bleeding-Edge Techniques for Univariate Time Series....227

Single-Step Data Preparation for Time-Series Forecasting	227
Horizon-Style Data Preparation for Time-Series Forecasting	229
LSTM Univariate Single-Step Style in Action	230
LSTM Univariate Horizon Style in Action	242
Bidirectional LSTM Univariate Single-Step Style in Action	253
Bidirectional LSTM Univariate Horizon Style in Action	262
GRU Univariate Single-Step Style in Action.....	271
GRU Univariate Horizon Style in Action	279
Auto-encoder LSTM Univariate Single-Step Style in Action.....	288
Auto-encoder LSTM Univariate Horizon Style in Action	297
CNN Univariate Single-Step Style in Action	306
CNN Univariate Horizon Style in Action	315
Summary.....	324

Chapter 7: Bleeding-Edge Techniques for Multivariate Time Series .. 325

LSTM Multivariate Horizon Style in Action	325
Bidirectional LSTM Multivariate Horizon Style in Action	337
Auto-encoder LSTM Multivariate Horizon Style in Action.....	346
GRU Multivariate Horizon Style in Action	356
CNN Multivariate Horizon Style in Action	365
Summary.....	374

TABLE OF CONTENTS

Chapter 8: Prophet.....	375
The Prophet Model.....	375
Implementing Prophet.....	376
Adding Log Transformation.....	381
Adding Built-in Country Holidays.....	386
Adding Exogenous variables using add_regressors(function)	389
Summary.....	394
Index.....	395

About the Authors



B V Vishwas is a Data Scientist, AI researcher and AI Consultant, Currently living in Bengaluru(INDIA). His highest qualification is Master of Technology in Software Engineering from Birla Institute of Technology & Science, Pilani, India and his primary focus and inspiration is Data Warehousing, Big Data, Data Science (Machine Learning, Deep Learning, Timeseries, Natural Language Processing, Reinforcement Learning, and

Operation Research). He has over seven years of IT experience currently working at Infosys as Data Scientist & AI Consultant. He has also worked on Data Migration, Data Profiling, ETL & ELT, OWB, Python, PL/SQL, Unix Shell Scripting, Azure ML Studio, Azure Cognitive Services, and AWS.



Ashish Patel is a Senior Data Scientist, AI researcher, and AI Consultant with over seven years of experience in the field of AI, Currently living in Ahmedabad(INDIA). He has a Master of Engineering Degree from Gujarat Technological University and his keen interest and ambition to research in the following domains such as (Machine Learning, Deep Learning, Time series, Natural Language Processing, Reinforcement Learning, Audio Analytics, Signal Processing, Sensor

ABOUT THE AUTHORS

Technology, IoT, Computer Vision). He is currently working as Senior Data Scientist for Cynet infotech Pvt Ltd. He has published more than 15 + Research papers in the field of Data Science with Reputed Publications such as IEEE. He holds Rank 3 as a kernel master in Kaggle. Ashish has immense experience working on cross-domain projects involving a wide variety of data, platforms, and technologies.

About the Technical Reviewer



Over his nearly three-decade career, **Alexey Panchekha, PhD, CFA**, has spent 10 years in academia, where he focused on nonlinear and dynamic processes; 10 years in the technology industry, where he specialized in program design and development; and eight years in financial services. In the latter arena, he specialized in applying mathematical techniques and technology to risk management

and alpha generation. For example, Panchekha was involved in the equity derivative trading technology platform at Goldman Sachs, and he led the creation of the multi-asset multigeographies portfolio risk management system at Bloomberg. He also served as the head of research at Markov Process International, a leader in portfolio attribution and analytics. Most recently, Panchekha cofounded Turing Technology Associates, Inc., with Vadim Fishman. Turing is a technology and intellectual property company that sits at the intersection of mathematics, machine learning, and innovation. Its solutions typically service the financial technology (fintech) industry. Turing primarily focuses on enabling technology that supports the burgeoning ensemble active management (EAM). Prior to Turing, Panchekha was managing director at Incapital, and head of research at F-Squared Investments, where he designed innovative volatility-based risk-sensitive investment strategies. He is fluent in multiple computer programming languages and software and database programs and is certified in deep learning software. He earned a PhD from Kharkiv Polytechnic University with studies in physics and mathematics as well as an MS in physics. Panchekha is a CFA charterholder.

Acknowledgments

This being my first book, I found transforming idea and real-world experience into its current shape to be a strenuous task. I am grateful to almighty God for blessing and guiding me in all endeavors. I would like to thank my parents (Vijay Kumar and Rathnamma), brother (Shreyas), other family, and friends for helping me sail though the sea of life.

—B V Vishwas

First and foremost, praises and thanks to God, the almighty, for His showers of blessings throughout the book-writing process. I would like to express my deep and sincere gratitude to my parents (Dinesh Kumar N Patel and Javnika ben D Patel), my sister (Nisha Patel), and my family and friends (Shailesh Patel, Sanket Patel, Nikit Patel, Mansi Patel, Khushboo Shah) for their support and valuable prayers.

—Ashish Patel

Special thanks to Celestin Suresh John, Aditee Mirashi, James Markham, Alexey Panchekha, and the Apress team for bringing this book to life.

Introduction

This book explains the concepts of time series from traditional to bleeding-edge techniques with full-fledged examples.

The book begins by covering time-series fundamentals and their characteristics, Structure & Components of time series data, preprocessing, and ways of crafting features through data wrangling. Next, it covers traditional time-series techniques such as the smoothing methods ARMA, ARIMA, SARIMA, SARIMAX, VAR, and VARMA using trending frameworks such as [Statsmodels](#) and Pmdarima.

Further covers how to leverage advanced deep learning-based techniques such as ANN, CNN, RNN, LSTM, GRU, and Autoencoder to solve time-series problems using Tensorflow. It concludes by explaining how to use the popular framework fbprophet for modeling time-series analysis.

After completion of the book, the reader will have thorough knowledge of concepts and techniques to solve time-series problems. All the code presented in this book is available in Jupyter Notebooks; this allows readers to do hands-on experiments and enhance them in exciting ways.

CHAPTER 1

Time-Series Characteristics

A *time series* is a collection of data points that are stored with respect to their time. Mathematical and statistical analysis performed on this kind of data to find hidden patterns and meaningful insight is called *time-series analysis*. Time-series modeling techniques are used to understand past patterns from the data and try to forecast future horizons. These techniques and methodologies have been evolving for decades.

Observations with continuous timestamps and target variables
are sometimes framed as straightforward regression problems by
decomposing dates into minutes, hours, days, weeks, months, years, and
so on, which is not the right way to handle such data because the results
obtained are poor. In this chapter, you will learn the right approach for handling time-series data.

There are different kinds of data, such as structured, semistructured, and unstructured, and each type should be handled in its own way to gain maximum insight. In this book, we are going to be looking at time-series data that is structured in manner such as data from the stock market, weather, birth rates, traffic, bike-sharing apps, etc.

This chapter is a gentle introduction to the types of time-series data, its components, and ways to decompose it.

Types of Data

Time-series analysis is a statistical technique that measures a sequential set of data points. This is a standard measure in terms of time that comes in three types, as shown in Figure 1-1.

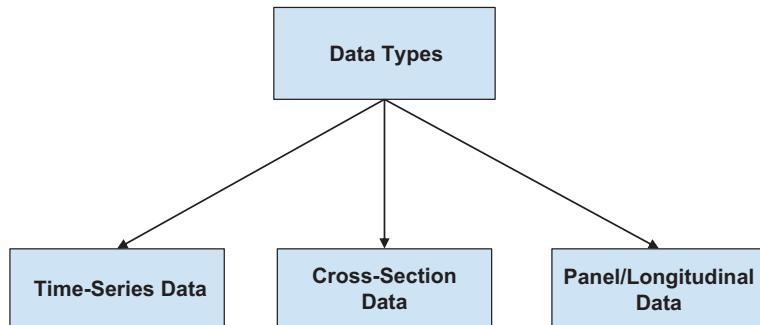


Figure 1-1. Types of data

Time-Series Data

A time series contains data points that increase, decrease, or otherwise change in chronological order over a period. A time series that incorporates the records of a single feature or variable is called a *univariate* time series. If the records incorporate more than one feature or variable, the series is called a *multivariate* time series. In addition, a time series can be designated in two ways: continuous or discrete.

In a *continuous* time series, data observation is carried out continuously throughout the period, as with earthquake seismograph magnitude data, speech data, etc. Figure 1-2 illustrates earthquake data measured continuously from 1975 to 2015.

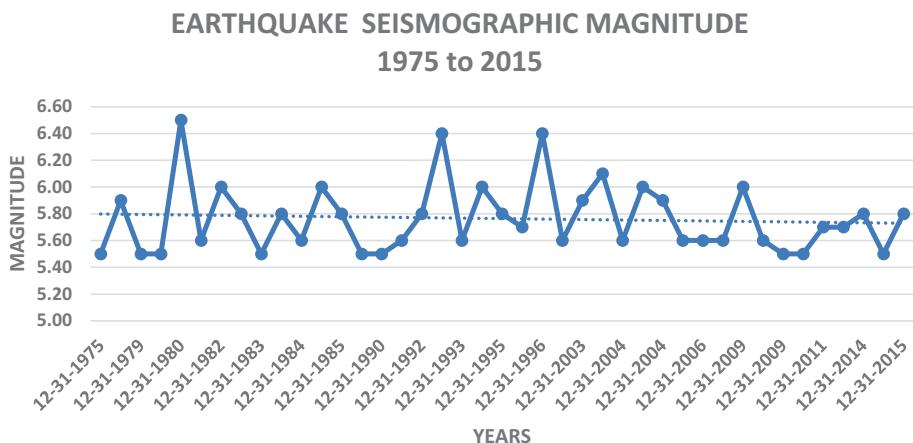


Figure 1-2. Forty years of earthquake seismograph magnitude data

Figure 1-3 Illustrates temperature behavior in India over a century and clearly shows that temperature is increasing monotonically.

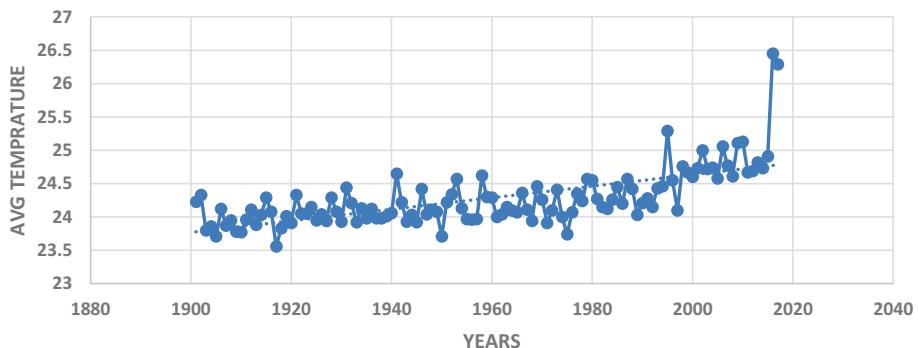


Figure 1-3. India's temperature data from 1901 to 2017

In a *discrete* time series, data observation is carried out at a specific time or equally spaced, as with temperature increases or decreases, exchange rates of currencies, air pressure data, etc. Figure 1-2 illustrates the analysis of the average temperature of India from 1901 to 2017, which either increases or decreases with time. This data behavior is discrete.

Cross-Section Data

Cross-section data is data gathered at a specific point of time for several subjects such as closing prices of a particular group of stocks on a specific date, opinion polls of elections, obesity level in population, etc. Cross-section studies are utilized in many research areas such as medical, economics, psychology, etc. For instance, high blood pressure is one of the significant risk factors for cause of death in India according to a 2017 WHO report. WHO has carried out the study of several risk factors (considered various subjects), which reflects cross-section survey data. Figure 1-4 illustrates the cross-section data.

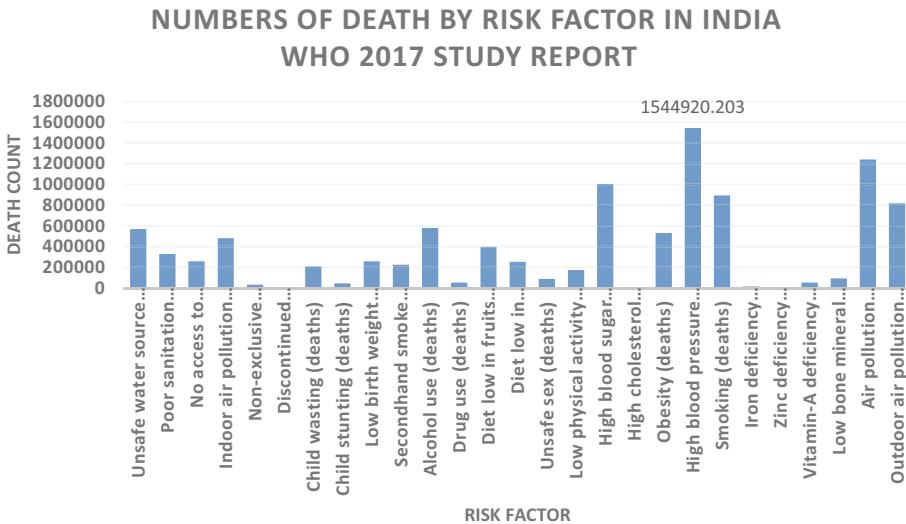


Figure 1-4. Number of deaths by risk factor in India

Panel Data/Longitudinal Data

Panel data/longitudinal data contains observations of multiple occurrences collected over various durations of time for the same individuals. It is data that is determined periodically by the number of observations in cross-sectional data units such as individuals, companies,

or government agencies. Table 1-1 provides examples of data available for multiple people over the course of a few years where the data gathered comprises income, age, and sex.

Table 1-1. Example of Panel Data

Panel Data A

Name	Year	Income	Age	Sex
Allen	2016	42145	24	Female
Allen	2017	47797	21	Female
Allen	2018	41391	23	Female
Malissa	2016	41100	22	Male
Malissa	2017	25800	23	Male
Malissa	2018	34508	22	Male

Panel Data B

Name	Year	Income	Age	Sex
Malissa	2016	42688	27	Female
Malissa	2017	21219	25	Female
Allen	2016	46340	26	Male
Allen	2017	22715	22	Male
Allen	2018	34653	21	Male
Alicia	2017	31553	29	Female

In Table 1-1, datasets A and B (with the attributes income, age, and sex) gathered throughout the years are for different people. Dataset A is a depiction of two people, Allen and Malissa, who were subject to observation over three years (2016, 2017, 2018); this is known as *balanced panel data*. Dataset B is called *unbalanced panel data* because data does not exist for every individual every year.

Trend

A **trend** is a pattern that is observed over a period of time and represents the mean rate of change with respect to time. A trend usually shows the tendency of the data to increase/up trend or decrease/downtrend during the long run. It is not always necessary that the increase or decrease is in the same direction throughout the given period of time. A trend line is also drawn using candlestick charts.

For example, you may have heard about an increase or decrease in different market commodities such as gold, silver, stock prices, gas, diesel, etc., or about the rate of interest for banks or home loans increasing or decreasing. These are all commodity market conditions, which may either increase or decrease over time, that show a trend in data.

Detecting Trend Using a Hodrick-Prescott Filter

The Hodrick-Prescott (HP) filter has become a benchmark for getting rid of trend movements in data. This method is broadly employed for econometric methods in applied macroeconomics research. The technique is nonparametric and is used to dissolve a time series into a trend; it is a cyclical component unaided by economic theory or prior trend specification. Like all nonparametric methods, the HP filter is contingent significantly on a tuning parameter that controls the degree of smoothing. This method is broadly employed in applied macroeconomics utilized in central banks, international economics agencies, industry, and government.

With the following example code, you can see how the EXINUS stock changes over a period of time:

```
import pandas as pd
from statsmodels.tsa.filters.hp_filter import hpfilter
df = pd.read_excel(r'\Data\India_Exchange_Rate_Dataset.xls',\
```

```
index_col=0,parse_dates=True)
EXINUS_cycle,EXINUS_trend = hpfilter(df['EXINUS'], lamb=1600)
EXINUS_trend.plot(figsize=(15,6)).autoscale(axis='x',tight=True)
```

Figure 1-5 shows an upward trend over the period.

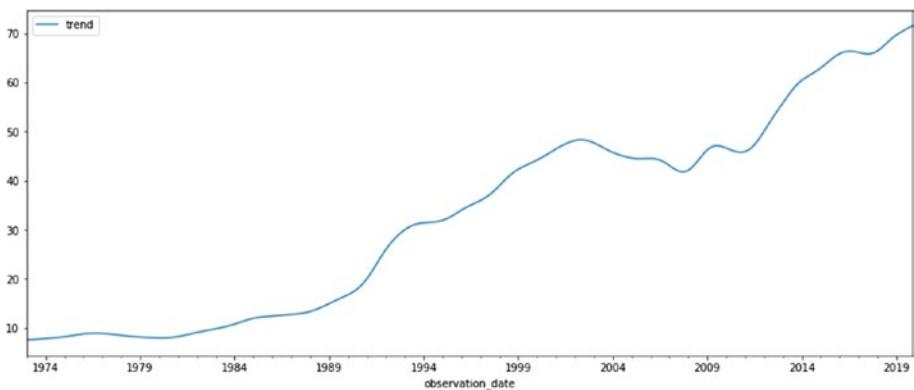


Figure 1-5. EXINUS stock showing an upward trend

Detrending a Time Series

Detrending is the process of removing a trend from time-series data, or it mentions a change in the mean over time. It is continuously increasing or decreasing over the duration of time. Identification, modeling, and even removing trend data from time-series datasets can be beneficial. The following are methods to detrend time-series data:

- Pandas differencing
- SciPy signal
- HP filter

Detrending Using Pandas Differencing

The Pandas library has a built-in function to calculate the difference in a dataset. This `diff()` function is used both for series and for DataFrames. It can provide a period value to shift in order to form the difference. The following code is an example of Pandas differencing.

- `Warning` is a built-in module of Python that handles the warning messages.
- `Pyplot` is a submodule of Matplotlib that is used to design the graphical representation of the data.

```
import pandas as pd
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
df = pd.read_excel(r'\Data\India_Exchange_Rate_Dataset.xls',\
index_col=0,parse_dates=True)
diff = df.EXINUS.diff()

plt.figure(figsize=(15,6))
plt.plot(diff)
plt.title('Detrending using Differencing', fontsize=16)
plt.xlabel('Year')
plt.ylabel('EXINUS exchange rate')
plt.show()
```

Figure 1-6 shows the data without a trend by using Pandas differencing.

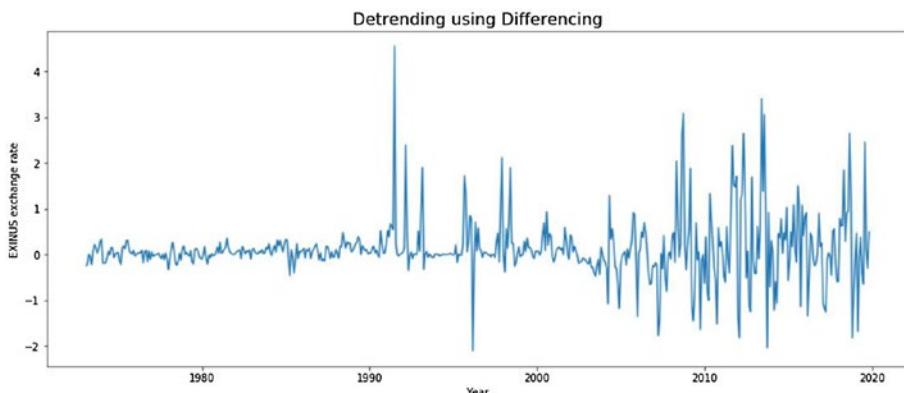


Figure 1-6. Trend removal using differencing

Detrending Using a SciPy Signal

A signal is another form of time-series data. Every signal either increases or decreases in a different order. Using the SciPy library, this can be removing the linear trend from the signal data. The following code shows an example of SciPy detrending.

- `Signal.detrend` is a submodule of SciPy that is used to remove a linear trend along an axis from data.

```
import pandas as pd
import matplotlib.pyplot as plt
from scipy import signal
import warnings
warnings.filterwarnings("ignore")
df = pd.read_excel(r'\Data\India_Exchange_Rate_Dataset.xls',\
index_col=0,parse_dates=True)
detrended = signal.detrend(df.EXINUS.values)

plt.figure(figsize=(15,6))
plt.plot(detrended)
plt.xlabel('EXINUS')
```

```
plt.ylabel('Frequency')
plt.title('Detrending using Scipy Signal', fontsize=16)
plt.show()
```

Figure 1-7 shows the detrended data using SciPy.

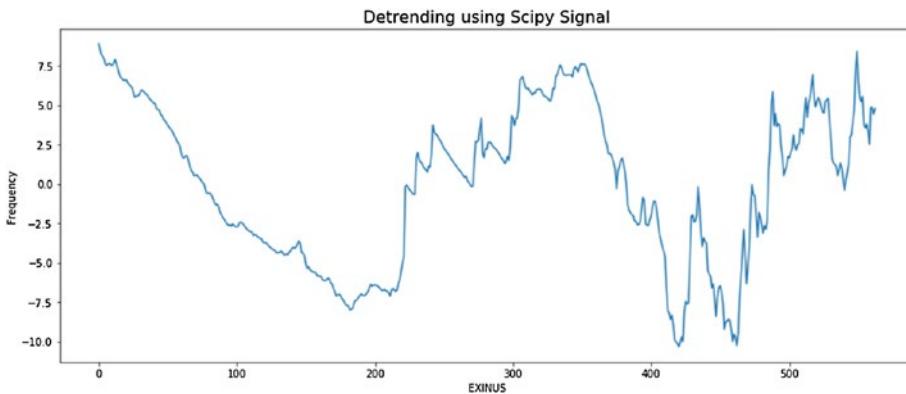


Figure 1-7. Removing a linear trend in a signal using SciPy

Detrend Using an HP Filter

An HP filter is also used to detrend a time series and smooth the data. It's used for removing short-term fluctuations. The following code shows an example of HP filter detrending.

- `HpfILTER` is a submodule of `Statmodels` that is used to remove a smooth trend.

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.filters.hp_filter import hpfILTER
import warnings
warnings.filterwarnings("ignore")
df = pd.read_excel(r'\Data\India_Exchange_Rate_Dataset.xls',\
index_col=0,parse_dates=True)
```

```

EXINUS_cycle,EXINUS_trend = hpfilter(df['EXINUS'],
lamb=1600)
df['trend'] = EXINUS_trend
.
detrended = df.EXINUS - df['trend']
plt.figure(figsize=(15,6))
plt.plot(detrended)
plt.title('Detrending using HP Filter', fontsize=16)
plt.xlabel('Year')
plt.ylabel('EXINUS exchange rate')
plt.show()

```

Figure 1-8 shows the data after removing a smooth trend.

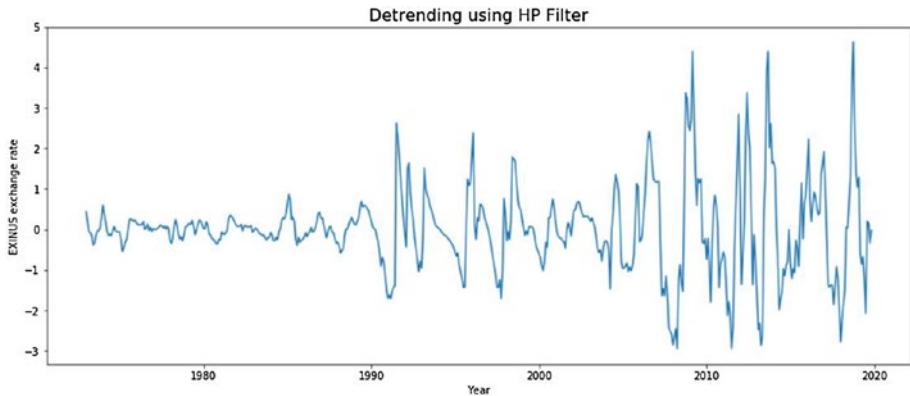


Figure 1-8. Trend removal using an HP filter

Seasonality

Seasonality is a periodical fluctuation where the same pattern occurs at a regular interval of time. It is a characteristic of economics, weather, and stock market time-series data; less often, it's observed in scientific data. In other industries, many phenomena are characterized by periodically recurring seasonal effects. For example, retail sales tend to increase during Christmas and decrease afterward.

The following methods can be used to detect seasonality:

- Multiple box plots
- Autocorrelation plots

Multiple Box Plots

A *box plot* is an essential graph to depict data spread out over a range. It is a standard approach to showing the minimum, first quartile, middle, third quartile, and maximum. The following code shows an example of detecting seasonality with the help of multiple box plots. See Figure 1-9.

- Seaborn is a graphical representation package similar to Matplotlib.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
df=pd.read_excel(r'\Data\India_Exchange_Rate_Dataset.xls',\
parse_dates=True)
df['month'] = df['observation_date'].dt.strftime('%b')
df['year'] = [d.year for d in df.observation_date]
df['month'] = [d.strftime('%b') for d in
df.observation_date]
years = df['year'].unique()
plt.figure(figsize=(15,6))
sns.boxplot(x='month', y='EXINUS', data=df).set_
title("Multi Month-wise Box Plot")
plt.show()
```

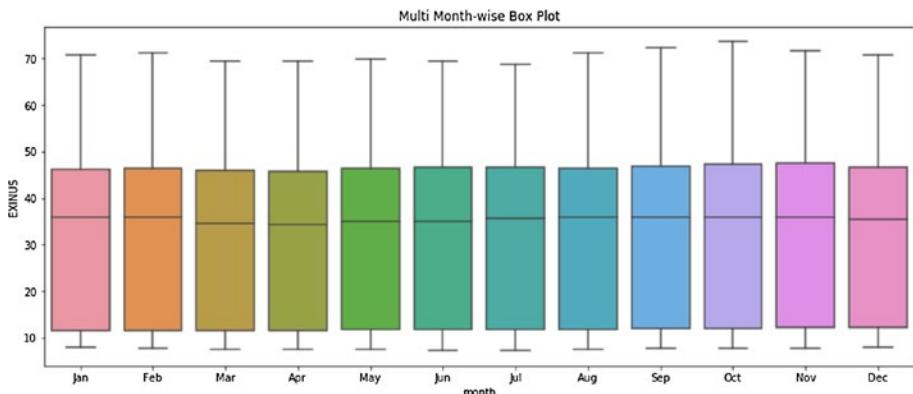


Figure 1-9. Multiple-box plot to identify seasonality

Autocorrelation Plot

Autocorrelation is used to check randomness in data. It helps to identify types of data where the period is not known. For instance, for the monthly data, if there is a regular seasonal effect, we would hope to see massive peak lags after every 12 months. Figure 1-10 demonstrates an example of detecting seasonality with the help of an autocorrelation plot.

```
from pandas.plotting import autocorrelation_plot
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_excel(r'\Data\India_Exchange_Rate_Dataset.xls',\
index_col=0,parse_dates=True)
plt.rcParams.update({'figure.figsize':(15,6), 'figure.dpi':220})
autocorrelation_plot(df.EXINUS.tolist())
```

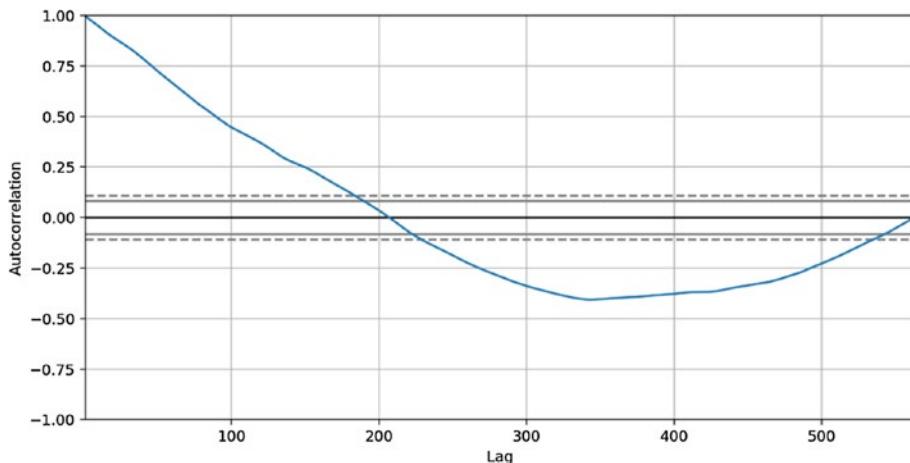


Figure 1-10. Autocorrelation plot to identify seasonality

Note Sometimes identifying seasonality is not easy; in that case, we need to evaluate other plots such as sequence or seasonal subseries plots.

Deseasoning of Time-Series Data

Deseasoning means to remove seasonality from time-series data. It is stripped of the pattern of seasonal effect to deseasonalize the impact. Time-series data contains four main components.

- **Level** means the average value of the time-series data.
- **Trend** means an increasing or decreasing value in time-series data.
- **Seasonality** means repeating the pattern of a cycle in the time-series data.
- **Noise** means random variance in time-series data.

Note An *additive model* is when time-series data combines these four components for linear trend and seasonality, and a *multiplicative model* is when components are multiplied to gather for nonlinear trends and seasonality.

Seasonal Decomposition

Decomposition is the process of understanding generalizations and problems related to time-series forecasting. We can leverage seasonal decomposition to remove seasonality from data and check the data only with the trend, cyclic, and irregular variations. Figure 1-11 illustrates data without seasonality.

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
import warnings
warnings.filterwarnings("ignore")
df = pd.read_excel(r'\Data\India_Exchange_Rate_Dataset.xls',
index_col=0,parse_dates=True)
result_mul = seasonal_decompose(df['EXINUS'],
model='multiplicative', extrapolate_trend='freq')
deseason = df['EXINUS'] - result_mul.seasonal

plt.figure(figsize=(15,6))
plt.plot(deseason)
plt.title('Deseasoning using seasonal_decompose', fontsize=16)
plt.xlabel('Year')
plt.ylabel('EXINUS exchange rate')
plt.show()
```

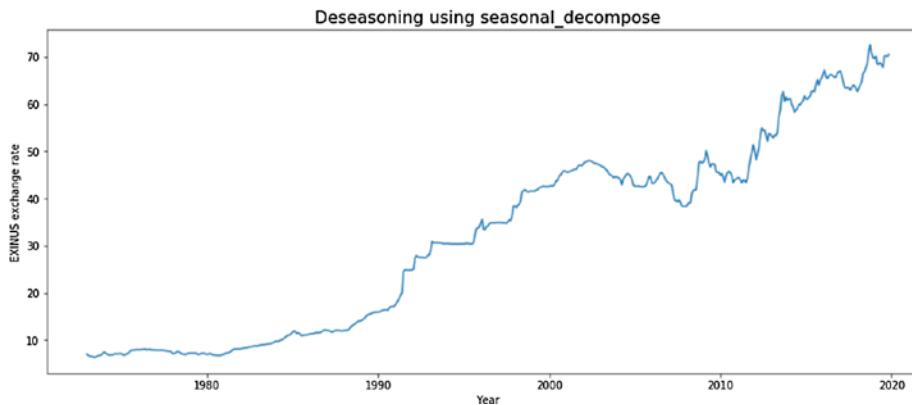


Figure 1-11. Deseasoning using seasonal_decompose from Statsmodels

Cyclic Variations

Cyclical components are fluctuations around a long trend observed every few units of time; this behavior is less frequent compared to seasonality. It is a recurrent process in a time series. In the field of business/economics, the following are three distinct types of cyclic variations examples:

- **Prosperity:** As we know, when organizations prosper, prices go up, but the benefits also increase. On the other hand, prosperity also causes over-development, challenges in transportation, increments in wage rate, insufficiency in labor, high rates of returns, deficiency of cash in the market and price concessions, etc., leading to depression
- **Depression:** As we know, when there is cynicism in exchange and enterprises, processing plants close down, organizations fall flat, joblessness spreads, and the wages and costs are low.

- **Accessibility:** This causes idealness of money, accessibility of cash at a low interest, an increase in demand for goods or money at a low interest rate, an increase in popular merchandise and ventures described by the circumstance of recuperation that at last prompts for prosperity or boom.

Detecting Cyclical Variations

The following code shows how to decompose time-series data and visualize only a cyclic pattern:

```
from statsmodels.tsa.filters.hp_filter import hpfilter
import pandas as pd
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
%matplotlib inline
df = pd.read_excel(r'\Data\India_Exchange_Rate_Dataset.xls',
index_col=0,parse_dates=True)
EXINUS_cycle,EXINUS_trend = hpfilter(df['EXINUS'], lamb=1600)
df[['cycle']] = EXINUS_cycle
df[['trend']] = EXINUS_trend
df[['cycle']].plot(figsize=(15,6)).autoscale(axis='x',tight=True)
plt.title('Extracting Cyclic Variations', fontsize=16)
plt.xlabel('Year')
plt.ylabel('EXINUS exchange rate')
plt.show()
```

Figure 1-12 illustrates isolated cyclic behavior from other data.

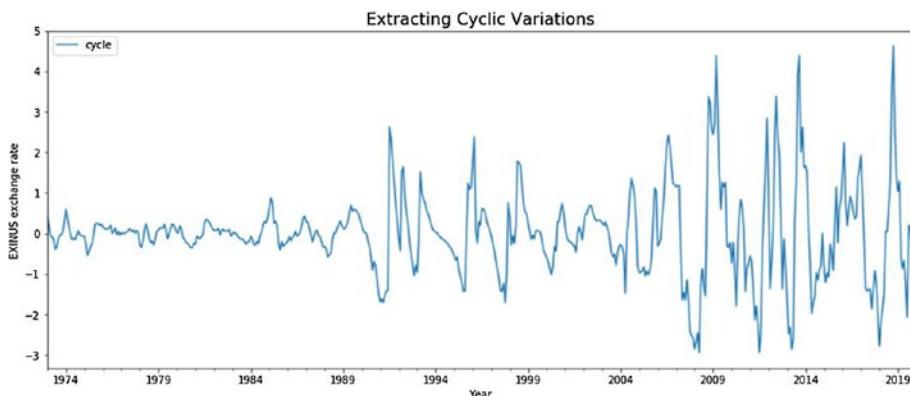


Figure 1-12. Extracting cyclic variations using an HP filter

Errors, Unexpected Variations, and Residuals

When trend and cyclical variations are removed from time-series data, the patterns left behind that cannot be explained are called errors, unexpected variations, or residuals. Various methods are available to check for irregular variations such as probability theory, moving averages, and autoregressive time-series methods. If we can find any cyclic variation in data, it is considered to be part of the residuals. These variations that occur due to unexpected circumstances are called unexpected variations or unpredictable errors.

Decomposing a Time Series into Its Components

Decomposition is a method used to isolate the time-series data into different elements such as trends, seasonality, cyclic variance, and residuals. We can leverage seasonal decomposition from a stats model

to decompose the data into its constituent parts, considering series as additive or multiplicative.

- **Trends($T[t]$)** means an increase or decrease in the value of ts data.
- **Seasonality($S[t]$)** means repeating a short-term cycle of ts data.
- **Cyclic variations($c[t]$)** means a fluctuation in long trends of ts data.
- **Residuals($e[t]$)** means an irregular variation of ts data.

The additive model works with linear trends of time-series data such as changes constantly over time. The additive model formula is as follows:

$$Y[t] = T[t] + S[t] + c[t] + e[t]$$

The multiplicative model works with a nonlinear type of data such as quadric or exponential. The multiplicative model formula is as follows:

$$Y[t] = T[t] * S[t] * c[t] * e[t]$$

The following example depicts the additive and multiplicative models:

```
from statsmodels.tsa.seasonal import seasonal_decompose
import pandas as pd
df = pd.read_excel(r'Data\India_Exchange_Rate_Dataset.xls',
index_col=0,parse_dates=True)

result = seasonal_decompose(df['EXINUS'], model='add')
result.plot()

result = seasonal_decompose(df['EXINUS'], model='mul')
result.plot()
```

Figure 1-13 and Figure 1-14 illustrate two methods of time-series decomposition.

CHAPTER 1 TIME-SERIES CHARACTERISTICS

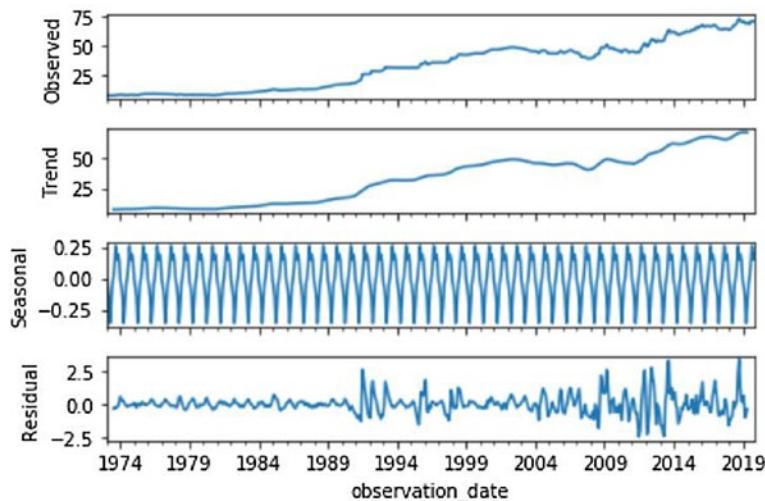


Figure 1-13. Decomposing time-series data using an additive model

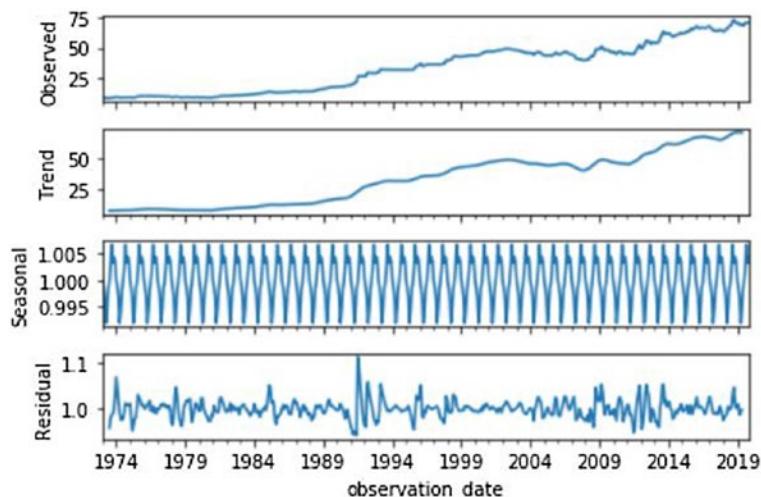


Figure 1-14. Decomposing time-series data using a multiplicative model

Summary

In this chapter, we discussed what time-series data is and glanced at various ways of isolating time-series components such as trend, seasonality, cyclic variations, and errors using multiple techniques. In the next chapter, you will learn how to perform data wrangling for time-series data.

CHAPTER 2

Data Wrangling and Preparation for Time Series

The expression “Garbage in, garbage out” certainly applies when it comes to data science.

Data wrangling is the art of transformation and mapping raw data into valuable information with the help of preprocessing strategies. It aims to provide rich data that can be utilized to gain maximum insights. It comprises operations such as loading, imputing, applying transformations, treating outliers, cleaning, integrating, dealing with inconsistency, reducing dimensionality, and engineering features. Data wrangling is an integral and main part of machine learning modeling. Real-world data is undoubtedly messy, and it is not reasonable to use data directly for modeling without performing some wrangling.

This chapter introduces how to read multiple data file formats with the help of Pandas. It covers how to do data selection, transformation, and cleansing as well as how to derive basic statistics with help of Pandas and Pandasql.

Notes The functions and methods introduced in this chapter have a lot more functionalities than covered here. You can explore them in more detail at the official Pandas and Pandasql sites.

Loading Data into Pandas

Pandas is the most notable framework for data wrangling. It includes data manipulation and analysis tools intended to make data analysis rapid and convenient. In the real world, data is generated via various tools and devices; hence, it comes in different formats such as CSV, Excel, and JSON. In addition, sometimes data needs to be read from a URL. All data comprises several records and variables. In this section, you will learn how various data formats are handled with Pandas.

Loading Data Using CSV

This dataset depicts the number of female births over time. Pandas has a built-in function to read CSV files. The following code imports the dataset:

```
import pandas as pd  
df = pd.read_csv(r'\Data\daily-total-female-births-CA.csv')  
df.head(5)
```

The following are the top five records from the CSV files:

	date	births
0	1959-01-01	35
1	1959-01-02	32
2	1959-01-03	30
3	1959-01-04	31
4	1959-01-05	44

If files have different separators, use `sep = ','` and fill in the separator, as in `(;, |,\t,',')`.

Loading Data Using Excel

The Istanbul Stock Exchange dataset comprises the returns of the Istanbul Stock Exchange along with seven other international indices (SP, DAX, FTSE, NIKKEI, BOVESPA, MSCE_EU, MSCI_EM) from June 5, 2009, to February 22, 2011. The data is organized by workday. This data is available in Excel format, and Pandas has a built-in function to read the Excel file. The following code imports the dataset:

```
import pandas as pd
dfExcel = pd.read_excel(r'\Data\istambul_stock_exchange.xlsx',
sheet_name = 'Data')
dfExcel.head(5)
```

The following are the top five records from the Excel file:

	date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
0	2009-01-05	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
1	2009-01-06	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2	2009-01-07	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
3	2009-01-08	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424
4	2009-01-09	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474	-0.009764	-0.010989	-0.007802

Loading Data Using JSON

This example dataset is in JSON format. The following code imports the dataset:

```
import pandas as pd
dfJson = pd.read_json(r'\Data\test.json')
dfJson.head(5)
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

The following are the records from the JSON:

	Names	Age
0	John	33
1	Sal	45
2	Tim	22
3	Rod	54

Loading Data from a URL

The Abalone dataset comprises 4,177 observations and 9 variables. It is not in any file structure; instead, we can display it as text at the specific URL. The following code imports the dataset:

```
import pandas as pd
dfURL = pd.read_csv(r'https://archive.ics.uci.edu/ml/
machine-learning-databases/abalone/abalone.data', names
=['Sex', 'Length','Diameter', 'Height','Whole weight',
'Shucked weight','Viscera weight', 'Shell weight', 'Rings'])
dfURL.head(5)
```

The following are the records from the URL:

Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

Exploring Pandasql and Pandas Side by Side

Pandasql is a Python framework for running SQL queries on Pandas DataFrames. It has plenty of essential attributes and a similar purpose as the sqldf framework in R. It helps us to query Pandas DataFrames using SQL syntax. It provides a SQL interface to perform data wrangling on a Pandas DataFrame. Pandasql helps analysts and data engineers who are masters of SQL to transition into Python (Pandas) smoothly.

Use `pip install pandasql` to install the library. In this chapter, we will explore ways to use both Pandas and Pandasql for solving different problems, such as selecting data, transforming data, and getting a basic summary of the data.

Selecting the Top Five Records

With the help of the `pandas.head()` function, we can fetch the first N records from the dataset. We can do the same operation with the help of Pandassql. The example illustrates how to do it with Pandas and Pandasql.

Here is how to do it with Pandas:

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp.head(5)
```

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time	Age
0	11	26	7	3	1	289		36	13 33
1	36	0	7	3	1	118		13	18 50
2	3	23	7	4	1	179		51	18 38
3	7	7	7	5	1	279		5	14 39
4	11	23	7	5	1	289		36	13 33

5 rows × 21 columns

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

Here is how to do it with Pandasql:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
Query_string = """ select * from dfpsql limit 5 """
sqldf(Query_string, globals())
```

The main function used in Pandasql is `sqldf`. It accepts two parameters: a SQL query string and a set of session/environment variables (`locals()` or `globals()`).

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time	Age
0	11	26	7	3	1	289	36	13 33
1	36	0	7	3	1	118	13	18 50
2	3	23	7	4	1	179	51	18 38
3	7	7	7	5	1	279	5	14 39
4	11	23	7	5	1	289	36	13 33

5 rows × 21 columns

Applying a Filter

Data filtering is a significant part of data preprocessing. With filtering, we can choose a smaller partition of the dataset and use that subset for viewing and munging; we need specific criteria or a rule to filter the data. This is also known as *subsetting data* or *drill-down data*. The following example illustrates how to apply a filter with Pandas and Pandasql.

Here is how to do it with Pandas:

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp[(dfp['Age'] >=30) & (dfp['Age'] <=45)]
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	11	26	7	3	1	289	36 13
1	3	23	7	4	1	179	51 18
2	7	7	7	5	1	279	5 14
3	11	23	7	5	1	289	36 13
4	3	23	7	6	1	179	51 18
5	20	23	7	6	1	260	50 11
6	14	19	7	2	1	155	12 14
7	1	22	7	2	1	235	11 14
8	20	1	7	2	1	260	50 11
9	20	1	7	3	1	260	50 11

Here is how to do it with Pandasql:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
Query_string = """ select * from dfpsql where age>=30 and age<=45 """
sqldf(Query_string, globals())
```

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	11	26	7	3	1	289	36 13
2	3	23	7	4	1	179	51 18
3	7	7	7	5	1	279	5 14
4	11	23	7	5	1	289	36 13
5	3	23	7	6	1	179	51 18
7	20	23	7	6	1	260	50 11
8	14	19	7	2	1	155	12 14
9	1	22	7	2	1	235	11 14
10	20	1	7	2	1	260	50 11
11	20	1	7	3	1	260	50 11

Distinct (Unique)

Several duplicate records exist in the dataset. If we want to select the number of unique values for the specific variable, then we can use the `unique()` function of Pandas. The following example illustrates how to do this with Pandas and Pandasql.

Here is how to do it with Pandas:

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp['ID'].unique()

array([11, 36, 3, 7, 10, 20, 14, 1, 24, 6, 33, 18, 30, 2, 19, 27, 34,
       5, 15, 29, 28, 13, 22, 17, 31, 23, 32, 9, 26, 21, 8, 25, 12, 16,
       4, 35], dtype=int64)
```

Here is how to do it with Pandasql:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
Query_string = """ select distinct ID from dfpsql;"""
sqldf(Query_string, globals())
```

ID
0 11
1 36
2 3
3 7
4 10
5 20
6 14
7 1
8 24
9 6
10 33

IN

Data filtering is a process of extracting essential data from a dataset by using some condition. There are several methods to do filtering on a dataset. Sometimes we want to investigate whether the data has been associated with a particular DataFrame or Series. In such an event, we can

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

use Pandas' `isin()` function, which checks whether values are present in the sequence. The procedure can also be carried out in Pandasql. The following example illustrates how to do it with Pandas and Pandasql.

Here is how to do it with Pandas:

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp[dfp.Age.isin([20,30,40])]
```

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	15	23	9	5	1	291	31 12
1	15	14	9	2	4	291	31 12
2	22	23	10	5	4	179	26 9
3	15	23	10	5	4	291	31 12
4	15	14	10	3	4	291	31 12
5	17	21	11	5	4	179	22 17
6	15	23	11	5	4	291	31 12
7	15	14	11	2	4	291	31 12
8	17	21	11	4	4	179	22 17
9	15	23	11	5	4	291	31 12

Here is how to do it with Pandasql:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
Query_string = """ select * from dfpsql where Age in (20,30,40);"""
sqldf(Query_string, globals())
```

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
47	15	23	9	5	1	291	31 12
49	15	14	9	2	4	291	31 12
65	22	23	10	5	4	179	26 9
71	15	23	10	5	4	291	31 12
75	15	14	10	3	4	291	31 12
83	17	21	11	5	4	179	22 17
84	15	23	11	5	4	291	31 12
87	15	14	11	2	4	291	31 12
91	17	21	11	4	4	179	22 17
97	15	23	11	5	4	291	31 12

NOT IN

The NOT IN operation is used for a similar purpose as explained earlier. If we want to check whether a value is not part of a sequence, we can use the tilde (~) symbol to perform a NOT IN operation.

Here is the example with Pandas:

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp[~dfp.Age.isin([20,30,40])]
```

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	11	26	7	3	1	289	36
1	36	0	7	3	1	118	13
2	3	23	7	4	1	179	51
3	7	7	7	5	1	279	5
4	11	23	7	5	1	289	36
5	3	23	7	6	1	179	51
6	10	22	7	6	1	361	52
7	20	23	7	6	1	260	50
8	14	19	7	2	1	155	12
9	1	22	7	2	1	235	11

Here is the example with Pandasql:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
Query_string = """ select * from dfpsql where Age not in(20,30,40);"""
sqldf(Query_string, globals())
```

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	11	26	7	3	1	289	36
1	36	0	7	3	1	118	13
2	3	23	7	4	1	179	51
3	7	7	7	5	1	279	5
4	11	23	7	5	1	289	36
5	3	23	7	6	1	179	51
6	10	22	7	6	1	361	52
7	20	23	7	6	1	260	50
8	14	19	7	2	1	155	12
9	1	22	7	2	1	235	11

Ascending Data Order

`ORDER BY` sorts the result-set in ascending or descending order based on the value selected. Pandas has a `sort_values()` function that can use different sorting algorithms, such as quicksort, mergesort, and heapsort. The default value is ascending order, whose Boolean value is `True`. Except for that axis-wise, we do perform sorting such as 0 for index and 1 for columns. SQL has an `ORDER BY` clause to perform a similar sorting operation.

Here is the example using Pandas:

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp.sort_values(['Age','Service_time'], ascending= True)
```

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
40	27	23	9	3	1	184	42
118	27	23	1	5	2	184	42
132	27	23	1	5	2	184	42
137	27	23	2	6	2	184	42
149	27	23	2	3	2	184	42
209	27	7	5	4	3	184	42
269	27	6	8	4	1	184	42
6	10	22	7	6	1	361	52
22	10	13	8	2	1	361	52
25	10	25	8	2	1	361	52

Here is the example using Pandasql:

`ORDER BY` in SQLite by default is ascending.

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
Query_string = """ select * from dfpsql order by Age,Service_time;"""
sqldf(Query_string, globals())
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
40	27	23	9	3	1	184	42
118	27	23	1	5	2	184	42
132	27	23	1	5	2	184	42
137	27	23	2	6	2	184	42
149	27	23	2	3	2	184	42
209	27	7	5	4	3	184	42
269	27	6	8	4	1	184	42
6	10	22	7	6	1	361	52
22	10	13	8	2	1	361	52
25	10	25	8	2	1	361	52

Descending Data Order

As mentioned, the `sort_value()` function can sort results in ascending or descending order. It needs the parameter `ascending = False` to sort values in descending order. You can also update some other parameters, as explained for `ORDER BY ascending`.

Here is the example with Pandas:

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp.sort_values(['Age','Service_time'], ascending= False)
```

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
9	18	8	3	1	228	14	16
9	18	5	4	3	228	14	16
9	1	10	4	4	228	14	16
9	25	3	3	2	228	14	16
9	12	3	3	2	228	14	16
9	25	3	4	2	228	14	16
9	6	7	2	1	228	14	16
9	6	7	3	1	228	14	16
35	0	0	6	3	179	45	14
36	0	7	3	1	118	13	18

You can sort results in descending order by using a `desc` keyword next to the column names, as shown here:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
Query_string = """ select * from dfpsql order by Age
Desc,Service_time Desc;"""
sqldf(Query_string, globals())
```

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time	Age
9	18	8	3	1	228		14	16 58
9	18	5	4	3	228		14	16 58
9	1	10	4	4	228		14	16 58
9	25	3	3	2	228		14	16 58
9	12	3	3	2	228		14	16 58
9	25	3	4	2	228		14	16 58
9	6	7	2	1	228		14	16 58
9	6	7	3	1	228		14	16 58
35	0	0	6	3	179		45	14 53
36	0	7	3	1	118		13	18 50

Aggregation

Aggregation is the process of mining data so the data can be investigated, collected, and presented in a summarized manner. It allows you to perform a calculation on single or multiple vectors, usually accompanied by the `group_by()` functions in Pandas. It performs numerous operations such as splitting, applying, and combining. This example illustrates the aggregation operation in Pandas and Pandasql.

For the following example, we are leveraging `.agg` in Pandas to achieve the required result:

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp.agg({'Transportation_expense': ['count','min', 'max',
'mean']})
```

Transportation_expense	
count	740.00000
min	118.00000
max	388.00000
mean	221.32973

There is also an option to aggregate directly on a column, as shown here:

```
dfp.Transportation_expense.min()
```

Here is the example with Pandasql:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
Query_string = """ select count(Transportation_expense) as count,
min(Transportation_expense) as min, max(Transportation_expense)
as max, avg(Transportation_expense) as mean from dfp;"""
sqldf(Query_string, globals())
```

	count	min	max	mean
0	740	118	388	221.32973

GROUP BY

We can use GROUP BY to arrange identical data into groups based on the aggregation function used. In other words, it groups rows that have similar values into summary rows. We perform this operation with the groupby() function in Pandas. SQL has its GROUP BY clause to perform a similar operation. The example illustrates the GROUP BY operation in Pandas and Pandasql.

Here is the example with Pandas:

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp.groupby('ID')['Service_time'].sum()
```

ID	
1	322
2	72
3	2034
4	13
5	247
6	104
7	84
8	28
9	128
10	72
11	520
12	7
13	180
14	406
15	444
16	48
17	340

Here is the example with Pandasql:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
Query_string = """ select ID , sum(Service_time) as Sum_Service_time from dfp group by ID;"""
sqldf(Query_string, globals())
```

ID	Sum_Service_time
1	322
2	72
3	2034
4	13
5	247
6	104
7	84
8	28
9	128
10	72
11	520

GROUP BY with Aggregation

The groupby() function provides a group of similar data based on a selected feature with that data; we can perform different aggregation operations with the .agg() function on other features in Pandas. In SQL, we use GROUP BY to apply aggregate functions on groups of data returned from a query. FILTER is a modifier used with an aggregate function to bound the values used in an aggregation.

Here is the example with Pandas:

```
import pandas as pd
dfp = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
dfp.groupby('Reason_for_absence').agg({'Age':
['mean','min','max']})
```

		Age		
		mean	min	max
Reason_for_absence				
	0	39.604651	28	53
	1	37.687500	28	58
	2	28.000000	28	28
	3	40.000000	40	40
	4	45.000000	41	49
	5	41.666667	37	50
	6	38.500000	27	58
	7	32.866667	27	46
	8	36.500000	28	40

Here is the example with Pandasql:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'\Data\Absenteeism_at_work.xls')
```

```
Query_string = """ select Reason_for_absence , avg(Age) as mean, min(Age) as min, max(Age) as max from dfp
group by Reason_for_absence;"""
sqldf(Query_string, globals())
```

Reason_for_absence	mean	min	max
0	39.604651	28	53
1	37.687500	28	58
2	28.000000	28	28
3	40.000000	40	40
4	45.000000	41	49
5	41.666667	37	50
6	38.500000	27	58
7	32.866667	27	46
8	36.500000	28	40

Join (Merge)

The terms *join* and *merge* mean the same thing in Pandas, Python, and other languages like SQL and R. In reality, their fundamental operation is equivalent, but their way of executing an operation is different. A *join* in Pandas utilizes an index to consolidate two data sources, while a *merge* looks for overlapping columns to merge. In Pandas, the `merge()` and `join()` functions are used. In SQL, the MERGE and JOIN operators perform the same operation as in Pandas. This example illustrates the aggregation operation in Pandas and Pandasql.

We've created Sample Employee and Department tables to illustrate the join examples.

```
import pandas as pd
data1 = {
    'Empid': [1011, 1012, 1013, 1014, 1015],
    'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

```
'Designation': ['Manager', 'Research  
Engineer', 'Research Engineer', 'VP',  
'Delivery Manager'],  
'Date_of_joining': ['01-Jan-2000', '23-sep-2006',  
'11-Jan-2012', '21-Jan-1991',  
'12-Jan-1990']}  
  
Emp_df = pd.DataFrame(data1, columns = ['Empid',  
                                         'Name', 'Designation','Date_of_  
                                         joining'])  
print(Emp_df.head(5))
```

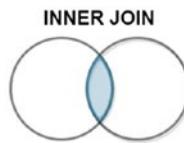
Empid	Name	Designation	Date_of_joining
1011	John	Manager	01-Jan-2000
1012	Rahul	Research Engineer	23-sep-2006
1013	Rick	Research Engineer	11-Jan-2012
1014	Morty	VP	21-Jan-1991
1015	Tim	Delivery Manager	12-Jan-1990

```
import pandas as pd  
data2 = {  
    'Empid': [1011, 1017, 1013, 1019, 1015],  
    'Deptartment': ['Management', 'Research',  
                    'Research', 'Management', 'Delivery'],  
    'Total_Experience': [18, 10, 10, 28, 22]}  
  
Dept_df = pd.DataFrame(data2, columns = ['Empid',  
                                         'Deptartment', 'Total_Experience'])  
print(Dept_df.head(5))
```

Empid	Deptartment	Total_Experience
1011	Management	18
1017	Research	10
1013	Research	10
1019	Management	28
1015	Delivery	22

INNER JOIN

An inner merge/inner join keeps rows where the merge value contains an “on” value in both the DataFrames. It selects the records that have matching values (joining columns) in one or more tables.



Here is the example with Pandas:

```
import pandas as pd
data1 = {
    'Empid': [1011, 1012, 1013, 1014, 1015],
    'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
    'Designation': ['Manager', 'Research Engineer', 'Research Engineer', 'VP', 'Delivery Manager'],
    'Date_of_joining': ['01-Jan-2000', '23-sep-2006', '11-Jan-2012', '21-Jan-1991', '12-Jan-1990']}
df1 = pd.DataFrame(data1)
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

```
Emp_df = pd.DataFrame(data1, columns = ['Empid',
                                         'Name', 'Designation','Date_of_joining'])
data2 = {
    'Empid': [1011, 1017, 1013, 1019, 1015],
    'Deptartment': ['Management', 'Research',
                    'Research', 'Management', 'Delivery'],
    'Total_Experience': [18, 10, 10, 28, 22]}
Dept_df = pd.DataFrame(data2, columns = ['Empid',
                                         'Deptartment', 'Total_Experience'])

pd.merge(Emp_df, Dept_df, left_on='Empid',right_on='Empid',
how='inner')
```

Empid	Name	Designation	Date_of_joining	Empid	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	1011	Management	18
1013	Rick	Research Engineer	11-Jan-2012	1013	Research	10
1015	Tim	Delivery Manager	12-Jan-1990	1015	Delivery	22

Here is the example with Pandasql:

```
import pandas as pd
data1 = {
    'Empid': [1011, 1012, 1013, 1014, 1015],
    'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
    'Designation': ['Manager', 'Research
Engineer', 'Research Engineer', 'VP', 'Delivery Manager'],
    'Date_of_joining': ['01-Jan-2000', '23-sep-2006',
                        '11-Jan-2012','21-Jan-1991',
                        '12-Jan-1990']}}

Emp_df = pd.DataFrame(data1, columns = ['Empid',
                                         'Name', 'Designation','Date_of_joining'])
```

```

data2 = {
    'Empid': [1011, 1017, 1013, 1019, 1015],
    'Deptartment': ['Management', 'Research',
                    'Research', 'Management', 'Delivery'],
    'Total_Experience': [18, 10, 10, 28, 22]}

Dept_df = pd.DataFrame(data2, columns = ['Empid',
                                         'Deptartment', 'Total_Experience'])

Query_string = """ select * from Emp_df a INNER JOIN Dept_df
b ON a.Empid = b.Empid;"""

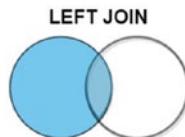
sqldf(Query_string, globals())

```

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18
1013	Rick	Research Engineer	11-Jan-2012	Research	10
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22

LEFT JOIN

A left merge or left join keeps a row on the left DataFrame when the missing value finds the “on” variable in the right DataFrame and adds the empty or NaN value in that result. All the records from the left table and selected matching records based on the joining condition from one or more tables.



CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

Here is the example with Pandas:

```
import pandas as pd
data1 = {
    'Empid': [1011, 1012, 1013, 1014, 1015],
    'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
    'Designation': ['Manager', 'Research Engineer',
                    'Research Engineer', 'VP',
                    'Delivery Manager'],
    'Date_of_joining': ['01-Jan-2000', '23-sep-2006',
                        '11-Jan-2012', '21-Jan-1991',
                        '12-Jan-1990']}
Emp_df = pd.DataFrame(data1, columns = ['Empid',
                                         'Name', 'Designation','Date_of_
                                         joining'])
data2 = {
    'Empid': [1011, 1017, 1013, 1019, 1015],
    'Deptartment': ['Management', 'Research',
                    'Research', 'Management', 'Delivery'],
    'Total_Experience': [18, 10, 10, 28, 22]}
Dept_df = pd.DataFrame(data2, columns = ['Empid',
                                         'Deptartment', 'Total_Experience'])
pd.merge(Emp_df, Dept_df, left_on='Empid', right_on='Empid',
how='left')
```

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18.0
1012	Rahul	Research Engineer	23-sep-2006	NaN	NaN
1013	Rick	Research Engineer	11-Jan-2012	Research	10.0
1014	Morty		21-Jan-1991	NaN	NaN
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22.0

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

Here is the example with Pandasql:

```

import pandas as pd
data1 = {
    'Empid': [1011, 1012, 1013, 1014, 1015],
    'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
    'Designation': ['Manager', 'Research
Engineer', 'Research Engineer', 'VP', 'Delivery Manager'],
    'Date_of_joining': ['01-Jan-2000', '23-sep-2006',
                        '11-Jan-2012', '21-Jan-1991',
                        '12-Jan-1990']}
Emp_df = pd.DataFrame(data1, columns = ['Empid',
                                         'Name', 'Designation','Date_of_
joining'])
data2 = {
    'Empid': [1011, 1017, 1013, 1019, 1015],
    'Deptartment': ['Management', 'Research',
                    'Research', 'Management', 'Delivery'],
    'Total_Experience': [18, 10, 10, 28, 22]}
Dept_df = pd.DataFrame(data2, columns = ['Empid',
                                         'Deptartment', 'Total_Experience'])

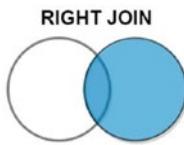
Query_string = """ select * from Emp_df a LEFT JOIN Dept_df
b ON a.Empid = b.Empid;"""
sqldf(Query_string, globals())

```

Empid	Name	Designation	Date_of_joining	Empid	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	1011.0	Management	18.0
1012	Rahul	Research Engineer	23-sep-2006	NaN	None	NaN
1013	Rick	Research Engineer	11-Jan-2012	1013.0	Research	10.0
1014	Morty	VP	21-Jan-1991	NaN	None	NaN
1015	Tim	Delivery Manager	12-Jan-1990	1015.0	Delivery	22.0

RIGHT JOIN

A right merge or right join keeps every row on the right DataFrame. Where the missing values find the “on” variable in the left columns, you add the empty/NaN values to the result. All the records from the left table and selected matching records based on the joining condition from one or more tables.



Here is the example with Pandas:

```
import pandas as pd
data1 = {
    'Empid': [1011, 1012, 1013, 1014, 1015],
    'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
    'Designation': ['Manager', 'Research
Engineer', 'Research Engineer', 'VP',
'Delivery Manager'],
    'Date_of_joining': ['01-Jan-2000', '23-sep-2006',
                      '11-Jan-2012', '21-Jan-1991',
                      '12-Jan-1990']]}

Emp_df = pd.DataFrame(data1, columns = ['Empid',
                                         'Name', 'Designation','Date_of_
joining'])

data2 = {
    'Empid': [1011, 1017, 1013, 1019, 1015],
    'Deptartment': ['Management', 'Research',
                    'Research', 'Management', 'Delivery'],
    'Total_Experience': [18, 10, 10, 28, 22]}


```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

```
Dept_df = pd.DataFrame(data2, columns = ['Empid',
                                         'Deptartment', 'Total_Experience'])
pd.merge(Emp_df, Dept_df, left_on='Empid', right_on='Empid',
how='right')
```

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18
1013	Rick	Research Engineer	11-Jan-2012	Research	10
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22
1017	NaN	NaN	NaN	Research	10
1019	NaN	NaN	NaN	Management	28

Here is the example with Pandasql:

```
import pandas as pd
data1 = {
    'Empid': [1011, 1012, 1013, 1014, 1015],
    'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
    'Designation': ['Manager', 'Research
Engineer', 'Research Engineer', 'VP',
'Delivery Manager'],
    'Date_of_joining': ['01-Jan-2000', '23-sep-2006',
    '11-Jan-2012','21-Jan-1991',
    '12-Jan-1990']]}

Emp_df = pd.DataFrame(data1, columns = ['Empid',
                                         'Name', 'Designation','Date_of_
joining'])

data2 = {
    'Empid': [1011, 1017, 1013, 1019, 1015],
    'Deptartment': ['Management', 'Research',
'Research', 'Management', 'Delivery'],
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

```
'Total_Experience': [18, 10, 10, 28, 22]}

Dept_df = pd.DataFrame(data2, columns = ['Empid',
                                         'Deptartment', 'Total_Experience'])

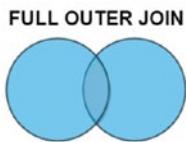
Query_string = """ select a.Empid,Name,Designation,Date_of_
joining,Deptartment,TOTAL_Experience from Dept_df a LEFT JOIN
Emp_df b ON a.Empid = b.Empid;"""

sqldf(Query_string, globals())
```

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18
1017	None	None	None	Research	10
1013	Rick	Research Engineer	11-Jan-2012	Research	10
1019	None	None	None	Management	28
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22

OUTER JOIN

An outer merge/full outer join returns all the rows from left and right DataFrames only where it matches or returns NaNs or empty values. It returns all the records from both the tables based on the joining condition regardless of whether they match or not. Not matching ones will be nulls.



Here is the example with Pandas:

```
import pandas as pd
data1 = {
    'Empid': [1011, 1012, 1013, 1014, 1015],
    'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

```

'Designation': ['Manager', 'Research
Engineer', 'Research Engineer', 'VP', 'Delivery Manager'],
'Date_of_joining': ['01-Jan-2000', '23-sep-2006',
                     '11-Jan-2012', '21-Jan-1991',
                     '12-Jan-1990']]}

Emp_df = pd.DataFrame(data1, columns = ['Empid',
                                         'Name', 'Designation','Date_of_
joining'])

data2 = {
    'Empid': [1011, 1017, 1013, 1019, 1015],
    'Deptartment': ['Management', 'Research',
                    'Research', 'Management', 'Delivery'],
    'Total_Experience': [18, 10, 10, 28, 22]}

Dept_df = pd.DataFrame(data2, columns = ['Empid',
                                         'Deptartment', 'Total_Experience'])

pd.merge(Emp_df, Dept_df, left_on='Empid', right_on='Empid',
how='outer')

```

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18.0
1012	Rahul	Research Engineer	23-sep-2006	NaN	NaN
1013	Rick	Research Engineer	11-Jan-2012	Research	10.0
1014	Morty	VP	21-Jan-1991	NaN	NaN
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22.0
1017	NaN	NaN	NaN	Research	10.0
1019	NaN	NaN	NaN	Management	28.0

Here is the example with Pandasql:

```

import pandas as pd
data1 = {

```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

```
'Empid': [1011, 1012, 1013, 1014, 1015],
'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
      'Designation': ['Manager', 'Research
Engineer', 'Research Engineer', 'VP',
'Delivery Manager'],
'Date_of_joining': ['01-Jan-2000', '23-sep-2006',
      '11-Jan-2012', '21-Jan-1991',
      '12-Jan-1990']}}

Emp_df = pd.DataFrame(data1, columns = ['Empid',
      'Name', 'Designation','Date_of_
joining'])

data2 = {
      'Empid': [1011, 1017, 1013, 1019, 1015],
      'Deptartment': ['Management', 'Research',
      'Research', 'Management', 'Delivery'],
      'Total_Experience': [18, 10, 10, 28, 22]}

Dept_df = pd.DataFrame(data2, columns = ['Empid',
      'Deptartment', 'Total_Experience'])
```

OUTER join in not currently supported in SQL lite

```
Query_string = """ select * from Emp_df a left OUTER JOIN Dept_
df b ON a.Empid = b.Empid;"""
sqldf(Query_string, globals())
```

Empid	Name	Designation	Date_of_joining	Empid	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	1011.0	Management	18.0
1012	Rahul	Research Engineer	23-sep-2006	NaN	None	NaN
1013	Rick	Research Engineer	11-Jan-2012	1013.0	Research	10.0
1014	Morty	VP	21-Jan-1991	NaN	None	NaN
1015	Tim	Delivery Manager	12-Jan-1990	1015.0	Delivery	22.0

Summary of the DataFrame

Descriptive statistics is a method used to depict data in a meaningful way to represent either the entire population or a sample. It has two sections: the *measure of central tendency*, which is used to measure the summary of a sample and population (e.g., mean, median, and mode), and the *measure of variability*, which is used to measure the spread or dispersion in the dataset (e.g., range, interquartile, variance, and standard deviation). In Pandas, it provides a built-in function called `describe()`. The following example illustrates the detailed result in Pandas.

In Pandas, the `describe()` function returns a number of descriptive statistics values (e.g., mean, standard deviation, min, median, max, first quartile, third quartile).

```
import pandas as pd
dfsumm = pd.read_csv(r'https://archive.ics.uci.edu/ml/machine-
learning-databases/abalone/abalone.data', names =['Sex',
                                                 'Length', 'Diameter', 'Height', 'Whole
                                                 weight', 'Shucked weight', 'Viscera
                                                 weight', 'Shell weight', 'Rings'])
dfsumm.head(5)
```

Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

`dfsumm.describe()`

	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
count	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000
mean	0.523992	0.407881	0.139516	0.828742	0.359367	0.180594	0.238831	9.933684
std	0.120093	0.099240	0.041827	0.490389	0.221963	0.109614	0.139203	3.224169
min	0.075000	0.055000	0.000000	0.002000	0.001000	0.000500	0.001500	1.000000
25%	0.450000	0.350000	0.115000	0.441500	0.186000	0.093500	0.130000	8.000000
50%	0.545000	0.425000	0.140000	0.799500	0.336000	0.171000	0.234000	9.000000
75%	0.615000	0.480000	0.165000	1.153000	0.502000	0.253000	0.329000	11.000000
max	0.815000	0.650000	1.130000	2.825500	1.488000	0.760000	1.005000	29.000000

Resampling

Resampling is a method to convert the time-based observed data into a different interval. In other words, it is used to change the time frequency into another time-frequency format. For instance, say we want to change monthly data into a year-wise format or upsample week data into hours. We would perform either upsample or downsample operations. For that, every data object must have a DateTime-like index(Datetimeindex, PeriodIndex, TimedeltaIndex). The following example illustrates the result of resampling operations in Pandas:

```
import pandas as pd
df = pd.read_csv(r'\Data\daily-total-female-births-CA.csv',index_col =0, parse_dates=['date'])
df.head(5)
```

births	
	date
1959-01-01	35
1959-01-02	32
1959-01-03	30
1959-01-04	31
1959-01-05	44

Resampling by Month

Here is how to resample by month:

```
df.births.resample('M').mean()
```

```
date
1959-01-31    39.129032
1959-02-28    41.000000
1959-03-31    39.290323
1959-04-30    39.833333
1959-05-31    38.967742
1959-06-30    40.400000
1959-07-31    41.935484
1959-08-31    43.580645
1959-09-30    48.200000
1959-10-31    44.129032
1959-11-30    45.000000
1959-12-31    42.387097
Freq: M, Name: births, dtype: float64
```

Resampling by Quarter

Here is how to resample by quarter:

```
df.births.resample('Q').mean()
```

```
date
1959-03-31    39.766667
1959-06-30    39.725275
1959-09-30    44.532609
1959-12-31    43.826087
Freq: Q-DEC, Name: births, dtype: float64
```

Resampling by Year

Here is how to resample by year:

```
df.births.resample('Y').mean()
```

```
date
1959-12-31    41.980822
Freq: A-DEC, Name: births, dtype: float64
```

Resampling by Week

Here is how to resample by week:

```
df.births.resample('W').mean()
```

date	
1959-01-04	32.000000
1959-01-11	37.714286
1959-01-18	44.285714
1959-01-25	41.142857
1959-02-01	35.142857
1959-02-08	40.428571
1959-02-15	42.857143
1959-02-22	42.428571
1959-03-01	40.000000
1959-03-08	39.428571
1959-03-15	36.571429
1959-03-22	40.857143
1959-03-29	39.142857
1959-04-05	41.142857
1959-04-12	37.857143
1959-04-19	37.285714
1959-04-26	40.142857

Resampling on a Semimonthly Basis

Here is how to resample on a semimonthly basis:

```
df.births.resample('SM').mean()
```

date	
1958-12-31	37.642857
1959-01-15	41.375000
1959-01-31	38.533333
1959-02-15	43.384615
1959-02-28	38.000000
1959-03-15	39.812500
1959-03-31	38.333333
1959-04-15	40.666667
1959-04-30	39.133333
1959-05-15	39.625000
1959-05-31	40.800000
1959-06-15	38.600000
1959-06-30	43.733333
1959-07-15	41.375000
1959-07-31	43.733333
1959-08-15	43.250000

Windowing Function

A *windowing function* is used to calculate the number of operations, such as rolling count, rolling sum, rolling mean, rolling median, rolling variance, rolling standard deviation, rolling min, rolling max, rolling correlation, rolling covariance, rolling skewness, rolling kurtosis, rolling quantile, etc. In addition, we can perform some other operations such as expanding window and exponential weighted moving window. The following example illustrates the result of windowing operations in Pandas:

```
import pandas as pd
dfExcelwin = pd.read_excel(r'\Data\istambul_stock_exchange.xlsx', sheet_name = 'Data',index_col =0,parse_dates=['date'])
dfExcelwin.head(5)
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
date									
2009-01-05	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-06	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-07	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-08	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424
2009-01-09	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474	-0.009764	-0.010989	-0.007802

Rolling Window

The rolling window feature supports the following methods: count, sum, mean, median, var, std, min, max, corr, cov, skew, kurt, quantile, sum, and aggregate.

```
import pandas as pd
dfExcelwin = pd.read_excel(r'\Data\istambul_stock_exchange.xlsx', sheet_name = 'Data',index_col =0,parse_dates=['date'])
dfExcelwin.rolling(window=4).mean().head(10)
```

	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
date									
2009-01-05	NaN								
2009-01-06	NaN								
2009-01-07	NaN								
2009-01-08	-0.007473	-0.010220	-0.005993	-0.004728	-0.003110	-0.004651	0.010624	0.000351	-0.000536
2009-01-09	-0.013946	-0.017400	-0.010206	-0.010244	-0.007261	-0.005770	0.000385	-0.005570	-0.009617
2009-01-12	-0.027600	-0.035943	-0.017858	-0.015739	-0.011734	-0.019070	-0.017807	-0.011518	-0.017468
2009-01-13	-0.016523	-0.029423	-0.009802	-0.015700	-0.006086	-0.023393	-0.007940	-0.010305	-0.013671
2009-01-14	-0.011263	-0.017132	-0.019158	-0.024614	-0.018705	-0.012650	-0.025086	-0.020220	-0.010984
2009-01-15	-0.013563	-0.023863	-0.013443	-0.024533	-0.019112	-0.024143	-0.015066	-0.020491	-0.014891

Expanding Window

Expanding window supports the following methods: count, sum, mean, median, var, std, min, max, corr, cov, skew, kurt, quantile, sum, aggregate, and quantile.

```
import pandas as pd
dfExcelwin = pd.read_excel(r'\Data\istambul_stock_exchange.
xlsx', sheet_name = 'Data',index_col =0,parse_dates=['date'])
dfExcelwin.expanding(min_periods=4).mean().head(10)
```

	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
date									
2009-01-05	NaN								
2009-01-06	NaN								
2009-01-07	NaN								
2009-01-08	-0.007473	-0.010220	-0.005993	-0.004728	-0.003110	-0.004651	0.010624	0.000351	-0.000536
2009-01-09	-0.004006	-0.006244	-0.009101	-0.007757	-0.005030	-0.004616	0.006546	-0.001917	-0.001989
2009-01-12	-0.008204	-0.012264	-0.011388	-0.008718	-0.005029	-0.012020	-0.003520	-0.003672	-0.005429
2009-01-13	-0.004825	-0.010551	-0.009510	-0.009998	-0.005188	-0.010303	-0.002507	-0.004894	-0.005343
2009-01-14	-0.009368	-0.013676	-0.012575	-0.014671	-0.010908	-0.008651	-0.007231	-0.009934	-0.005760
2009-01-15	-0.008254	-0.014075	-0.011030	-0.015213	-0.011289	-0.013295	-0.003059	-0.010172	-0.007723

Exponentially Weighted Moving Window

Expanding window supports the following methods: mean, std, var, corr, and cov.

```
import pandas as pd
dfExcelwin = pd.read_excel(r'\Data\istambul_stock_exchange.
xlsx', sheet_name = 'Data',index_col =0,parse_dates=['date'])
dfExcelwin.ewm(com=0.5).mean().head(10)
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
date									
2009-01-05	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-06	0.028008	0.033454	0.004670	0.006890	0.010623	0.003122	0.021987	0.011690	0.013711
2009-01-07	-0.011363	-0.007951	-0.019657	-0.010226	-0.016625	0.012933	-0.018088	-0.008226	-0.009638
2009-01-08	-0.045684	-0.059767	-0.004099	-0.011239	-0.005718	-0.022838	0.013213	-0.006427	-0.016243
2009-01-09	-0.008502	-0.013292	-0.015770	-0.017019	-0.010398	-0.010545	-0.002168	-0.009481	-0.010593
2009-01-12	-0.022313	-0.032698	-0.020478	-0.014687	-0.006812	-0.036242	-0.036670	-0.011464	-0.018628
2009-01-13	0.002871	-0.011071	-0.005648	-0.016679	-0.006365	-0.012070	-0.009830	-0.011968	-0.009423
2009-01-14	-0.026493	-0.027394	-0.024574	-0.037152	-0.036090	-0.002080	-0.030147	-0.034140	-0.008925

Shifting

Shifting is also known as “lag” and moves a value back or forward in time. Pandas has a `df.shift()` function to shift an index by the desired number of periods with an optional time frequency. The following example illustrates the shifting operations result in Pandas:

```
import pandas as pd
dfshift = pd.read_excel(r'\Data\istambul stock exchange.xlsx',
sheet_name = 'Data',index_col =0, parse_dates=['date'])

dfshift.head(5)
```

	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
date									
2009-01-05	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-06	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-07	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-08	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424
2009-01-09	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474	-0.009764	-0.010989	-0.007802

```
dfshift.shift(periods=3).head(7)
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
date									
2009-01-05	NaN								
2009-01-06	NaN								
2009-01-07	NaN								
2009-01-08	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-09	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-12	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-13	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424

`dfshift.shift(periods=-1).head(7)`

	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
date									
2009-01-05	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-06	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-07	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424
2009-01-08	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474	-0.009764	-0.010989	-0.007802
2009-01-09	-0.029191	-0.042361	-0.022823	-0.013526	-0.005026	-0.049039	-0.053849	-0.012451	-0.022630
2009-01-12	0.015445	-0.000272	0.001757	-0.017674	-0.006141	0.000000	0.003572	-0.012220	-0.004827
2009-01-13	-0.041168	-0.035552	-0.034032	-0.047383	-0.050945	0.002912	-0.040302	-0.045220	-0.008677

`dfshift.shift(periods=3, axis =1).head(7)`

	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
date									
2009-01-05	NaN	NaN	NaN	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000
2009-01-06	NaN	NaN	NaN	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162
2009-01-07	NaN	NaN	NaN	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293
2009-01-08	NaN	NaN	NaN	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061
2009-01-09	NaN	NaN	NaN	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474
2009-01-12	NaN	NaN	NaN	-0.029191	-0.042361	-0.022823	-0.013526	-0.005026	-0.049039
2009-01-13	NaN	NaN	NaN	0.015445	-0.000272	0.001757	-0.017674	-0.006141	0.000000

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

```
dfshift.shift(periods=3,fill_value=0).head(7)
```

	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
date									
2009-01-05	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2009-01-06	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2009-01-07	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2009-01-08	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-09	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-12	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-13	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424

Handling Missing Data

A missing value or missing data occurs when no data value is found at a respective number of places in either a DataFrame or dataset. A missing value presents many problems in the dataset. It reduces the statistical power of data, and the lost data can increase the bias in the dataset. So, it is essential to handle this missing value to maintain the characteristics of the data. Pandas has a number of methods to handle the missing values such as bfill(), ffill(), interpolate(), andfillna(). The following example illustrates the missing value-handling operations result in Pandas:

```
import pandas as pd
dfmiss = pd.read_csv(r'\Data\daily-total-female-births-CA-with-
nulls.csv',index_col =0, parse_dates=['date'])

dfmiss
```

CHAPTER 2 DATA WRANGLING AND PREPARATION FOR TIME SERIES

births	
date	
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	NaN
1959-01-09	38.0
1959-01-10	27.0

- `Isnull()` is a function to detect missing values for an array-like object.

```
dfmiss.isnull().sum()
```

```
births    16
dtype: int64
```

BFILL

The backward method fills the missing values in the dataset and uses the next valid observation to fill the gap.

```
dfmiss.bfill()
```

births	
	date
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	38.0
1959-01-09	38.0
1959-01-10	27.0
1959-01-11	38.0

FFILL

The forward fill method propagates the last valid observation forward to the next valid one.

```
dfmiss.ffill()
```

births	
date	
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	45.0
1959-01-09	38.0
1959-01-10	27.0
1959-01-11	38.0

FILLNA

This replaces NA/NaN values using a constant value.

```
dfmiss.fillna(10)
```

births	
date	
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	10.0
1959-01-09	38.0
1959-01-10	27.0
1959-01-11	38.0

INTERPOLATE

This method interpolates values linearly in a forward direction.

```
dfmiss.interpolate(method='linear',limit_direction='forward')
```

births	
date	
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	41.5
1959-01-09	38.0
1959-01-10	27.0
1959-01-11	38.0

Summary

In this chapter, we discussed different techniques for performing data wrangling with the help of Pandas and Pandasql. In the next chapter, you will learn some simple, double, and triple exponential smoothing techniques.

CHAPTER 3

Smoothing Methods

Smoothing is a statistical method we can use to create an approximation function to remove irregularities in data and attempt to capture significant patterns. Robert Goodell Brown was the father of exponential smoothing, and in 1956 he published “Exponential Smoothing for Predicting Demand” (<https://industrydocuments.library.ucsf.edu/tobacco/docs/#id=jzlc0130>). In 1957, professor Charles C. Holt was working at CMU on forecasting and published a paper called “Forecasting Seasonals and Trends by Exponentially Weighted Moving Averages” that discussed double exponential smoothing. Three years, in 1960, a student of Holt’s, Peter R. Winters, developed an algorithm by uniting seasonality and published “Forecasting Sales by Exponentially Weighted Moving Averages” (<https://pubsonline.informs.org/doi/abs/10.1287/mnsc.6.3.324>).

The smoothing technique is a family of time-series forecasting algorithms, which utilizes the weighted averages of a previous observation to predict or forecast a new value. The main idea of this technique is to overweight recent values in a time series. It is utilized for short-term forecasting models. This technique is more efficient when time-series data is moving slowly over time. It harmonizes errors, trends, and seasonal components into computing smoothening parameters. These components are pooled either additively or multiplicatively.

This chapter introduces the math behind simple, double, and triple exponential smoothing and how to leverage them efficiently to solve time-series problems.

Introduction to Simple Exponential Smoothing

Simple exponential smoothing (SES) is one of the minimal models of the exponential smoothing algorithm. Such techniques are employed for a univariate observation (data), which has no clear trend or seasonal patterns. It involves a parameter called *alpha*, which is the smoothing parameter. The core idea is to employ a weighted moving average, including exponentially decreasing weights, which designates higher weight to the most recent observation. The concept behind SES is forecasting future values by using a weighted average of all the previous values in the series. This method can be used to predict series that do not have trends or seasonality.

In this section, let's look at the high-level math behind simple exponential smoothing.

Assume that a series has the following:

- Level (L_t)
- No trends
- No seasonality
- Noise (somewhat constant)

SES is the forecast estimate level at a most recent point in time.

$$F_{t+k} = L_t$$

Let's estimate and update the level (L_t).

Here is the level updating equation:

$$L_t = \alpha Y_t + (1-\alpha) L_{t-1}$$

We are taking the level at time L_t and updating the previous level, L_{t-1} , by integrating information from most recent data point, Y_t . We can see that it is a weighted average, where α and $1-\alpha$ are the weights.

The smoothing constant equals α .

The $0 \leq \alpha \leq 1$ alpha value lies between 0 and 1.

The algorithm learns a new level from the new data that it sees. One of the possible ways of initializing the whole system is to assign L_1 equal to the first record in the series.

$$F_1 = L_1 = Y_1$$

Why is it called exponential smoothing?

Here is the level update equation:

$$L_t = \alpha Y_t + (1-\alpha) L_{t-1}$$

Substitute L_t with its formula, as shown here:

$$\begin{aligned} L_t &= \alpha Y_t + (1-\alpha) [(\alpha Y_{t-1} + (1-\alpha) L_{t-2})] = \\ &= \alpha Y_t + \alpha(1-\alpha) Y_{t-1} + (1-\alpha)^2 L_{t-2} = \dots \\ &= \alpha Y_t + \alpha(1-\alpha) Y_{t-1} + \alpha (1-\alpha)^2 Y_{t-2} + \dots \end{aligned}$$

We can see that we are ending up with an average of all the values in a series, but they have weights that are decaying exponentially into the past. Hence, this is called *exponential smoothing* when $\alpha = 1$ and past values do not influence the forecast. It's called *under-smoothing* when $\alpha = 0$ and past values have equal weight in the average. In this case, we do not give more weight to recent information, which is called *over-smoothing*.

Here is the effect of alpha:

α	$\alpha (1 - \alpha)$	$\alpha (1 - \alpha)^2$	$\alpha (1 - \alpha)^3$
0.9	0.089	0.0089	0.00089
0.5	0.25	0.125	0.0625
0.1	0.09	0.081	0.0729

In all cases, decay as we go into the past with alpha is fast, whereas with a smaller alpha decay it is slower.

Simple Exponential Smoothing in Action

In the previous section, you learned about the math behind simple exponential smoothing. Now let's implement it on a time-series dataset.

Import the required libraries and load the CSV data, which is Facebook stock data from 2014 to 2019.

- **NumPy** is the core library for scientific computing in Python. It provides a high-performance multidimensional array object and tools for working with these arrays.
- **sklearn** is a machine learning library for the Python programming language.

```
from statsmodels.tsa.api import SimpleExpSmoothing  
import pandas as pd  
import numpy as np  
from sklearn import metrics  
df = pd.read_csv(r'\Data\FB.csv')
```

Figure 3-1 provides a sneak peek at the data.

```
Print(df)
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2014-12-08	76.180000	77.250000	75.400002	76.519997	76.519997	25733900
1	2014-12-09	75.199997	76.930000	74.779999	76.839996	76.839996	25358600
2	2014-12-10	76.650002	77.550003	76.070000	76.180000	76.180000	32210500
3	2014-12-11	76.519997	78.519997	76.480003	77.730003	77.730003	33462100
4	2014-12-12	77.160004	78.879997	77.019997	77.830002	77.830002	28091600
5	2014-12-15	78.459999	78.580002	76.559998	76.989998	76.989998	29396500
6	2014-12-16	76.190002	77.389999	74.589996	74.690002	74.690002	31554600
7	2014-12-17	75.010002	76.410004	74.900002	76.110001	76.110001	29203900
8	2014-12-18	76.889999	78.400002	76.510002	78.400002	78.400002	34222100
9	2014-12-19	78.750000	80.000000	78.330002	79.879997	79.879997	43335000
10	2014-12-22	80.080002	81.889999	80.000000	81.449997	81.449997	31395800

Figure 3-1. First ten rows of dataset

Modeling will be done only for the Close column in the dataset. Make a copy of the data to perform the train/test split, where the train is used for training, and after getting satisfactory results, we will evaluate the results on the test data.

The train will have all the data expected for the last 30 days, and the test contains only the last 30 days to evaluate against predictions.

```
X = df['Close']
test = X.iloc[-30:]
train = X.iloc[:-30]
```

Let's look at the simple exponential smoothing algorithm's hyperparameters for reference.

The SimpleExpSmoothing parameter specifies the following:

smoothing_level (float, optional): This is the smoothing_level value of the simple exponential smoothing. If the value is set, then this value will be used as the value.

optimized (bool): This specifies whether the values that have not been set earlier should be optimized automatically.

Create a function that has all the required evaluation metrics, which will give us the results in one go. This function helps us understand how far off our forecasts are against the actuals.

- **Mean squared error (MSE)** tells you how close a regression line is to a set of points. It does this by taking the distances from the points to the regression line (these distances are the errors) and squaring them. The closer to zero the error is, the better the model.
- **Mean absolute error (MAE)** measures the average magnitude of the errors in a set of predictions, without considering their direction. The closer to zero the error is, the better the model.
- **Root mean square error (RMSE)** is a quadratic scoring rule that also measures the average magnitude of the error. It's the square root of the average of squared differences between prediction and actual observation. The closer to zero the error is, the better the model.
- **Mean absolute percentage error (MAPE)** is a statistical measure of how accurate a forecast system is. It is a measure in terms of percentage. It is mostly used for time-series forecasting. The closer to zero the error is, the better the model.
- **R-squared** determines the proportion of variance in the dependent variable that can be explained by the **independent variable**. The closer to zero the error is, the better the model.

MAPE and RMSE are notable statistical measures used to check the accuracy of the forecasting model.

```
def timeseries_evaluation_metrics_func(y_true, y_pred):
    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end='
\n\n')
```

The following snippet is used to find the best smoothing parameter, which ranges from 0 to 1. The smoothing level model will be fit, and its results will be captured in temp_df. The following results show that 1.0 gives the least RMSE.

The SimpleExpSmoothing parameter was imported from the Statsmodels package. Here we are using it to train the data. The training data is converted into a NumPy array. See Figure 3-2 for the results.

```
resu = []
temp_df = pd.DataFrame()
for i in [0, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80,
0.90,1]:
    print(f'RMSE for alpha={i} is {temp_df["RMSE"][-1]}' , end='|')
```

CHAPTER 3 SMOOTHING METHODS

```
print(f'Fitting for smoothing level= {i}')
fit_v = SimpleExpSmoothing(np.asarray(train)).fit(i)
fcst_pred_v= fit_v.forecast(30)
timeseries_evaluation_metrics_func(test,fcst_pred_v)
rmse = np.sqrt(metrics.mean_squared_error(test, fcst_
pred_v))
df3 = {'smoothing parameter':i, 'RMSE': rmse}
temp_df = temp_df.append(df3, ignore_index=True)
temp_df.sort_values(by=['RMSE']).head(3)
```

```
Fitting for smoothing level= 0
Evaluation metric results:-
MSE is : 3060.521317348409
MAE is : 55.15630993446825
RMSE is : 55.32197861020888
MAPE is : 28.209605312764108
R2 is : -166.21609234618037
```

```
Fitting for smoothing level= 0.1
Evaluation metric results:-
MSE is : 130.19890782430633
MAE is : 10.578096050945014
RMSE is : 11.41047360210374
MAPE is : 5.371184257014935
R2 is : -6.113609198116423
```

	RMSE	smoothing parameter
10	9.878157	1.0
9	9.926882	0.9
8	10.008074	0.8

Figure 3-2. Optimal parameter search results

From the previous search, the least RMSE was achieved with smoothing_level equal to 0.1. Let's use the same value and train the model.

```

fitSES = SimpleExpSmoothing(np.asarray(train)).fit(smoothing_
level = 0.1,optimized= False)
fcst_gs_pred = fitSES.forecast(30)
timeseries_evaluation_metrics_func(test,fcst_gs_pred)

Evaluation metric results:-
MSE is : 130.19890782430633
MAE is : 10.578096050945014
RMSE is : 11.41047360210374
MAPE is : 5.371184257014935
R2 is : -6.113609198116423

```

Let's check if `SimpleExpSmoothing` is able to find the best parameters by itself by adjusting a few settings. See Figure 3-3.

- `optimized` = `True` estimates the model parameters by maximizing the log likelihood.
- `use_brute`= `True` searches for good starting values using a brute-force (grid) optimizer.

```

fitSESAuto = SimpleExpSmoothing(np.asarray(train)).fit(
optimized= True, use_brute = True)
fcst_auto_pred = fitSESAuto.forecast(30)
timeseries_evaluation_metrics_func(test,fcst_auto_pred)

```

```

Evaluation metric results:-
MSE is : 97.68044338619876
MAE is : 8.909413652138449
RMSE is : 9.883341711496104
MAPE is : 4.516280753437192
R2 is : -4.336914972326928

```

```
fitSESAuto.summary()
```

CHAPTER 3 SMOOTHING METHODS

SimpleExpSmoothing Model Results

Dep. Variable:	endog	No. Observations:	1229
Model:	SimpleExpSmoothing	SSE	8965.774
Optimized:	True	AIC	2446.285
Trend:	None	BIC	2456.513
Seasonal:	None	AICC	2446.318
Seasonal Periods:	None	Date:	Sun, 29 Dec 2019
Box-Cox:	False	Time:	08:37:50
Box-Cox Coeff.:	None		
	coeff	code	optimized
smoothing_level	0.9808449	alpha	True
initial_level	76.519998	1.0	True

Figure 3-3. Summary of best parameter selected

Now that we have the results, let's set the index similar to the test index, such that plots align properly. See Figure 3-4.

```
df_fcst_gs_pred = pd.DataFrame(fcst_gs_pred, columns=['Close_grid_Search'])
df_fcst_gs_pred["new_index"] = range(1229, 1259)
df_fcst_gs_pred = df_fcst_gs_pred.set_index("new_index")

df_fcst_auto_pred = pd.DataFrame(fcst_auto_pred,
columns=['Close_auto_search'])
df_fcst_auto_pred["new_index"] = range(1229, 1259)
df_fcst_auto_pred = df_fcst_auto_pred.set_index("new_index")

import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.rcParams["figure.figsize"] = [16,9]
```

```
plt.plot( train, label='Train')
plt.plot(test, label='Test')
plt.plot(fcst_auto_pred, label='Simple Exponential Smoothing
using optimized =True')
plt.plot(fcst_gs_pred, label='Simple Exponential Smoothing
using custom grid search')
plt.legend(loc='best')
plt.show()
```

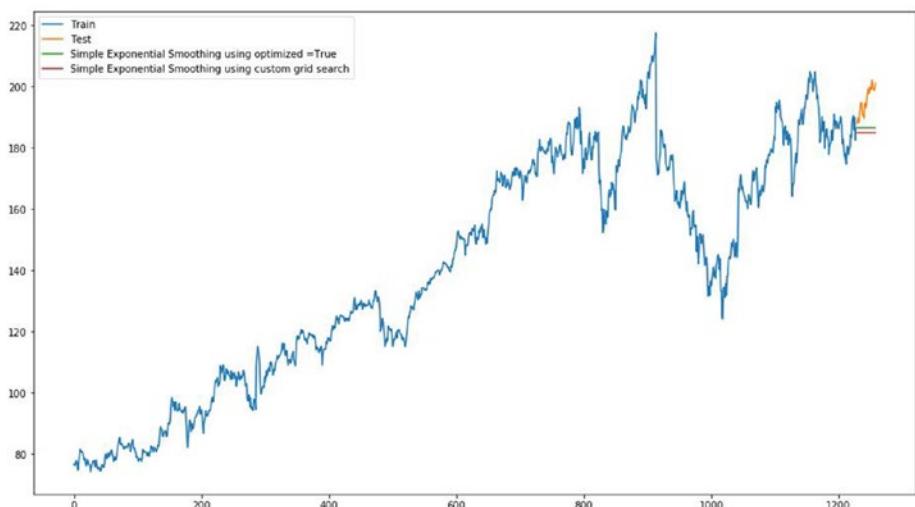


Figure 3-4. Line plot depicting results comparision between different configurations of simple exponential smoothing

We can clearly see that the simple exponential smoothing is not performing well as the stock market data, which will have trends and seasonality. Our basic model will not be able to capture these details.

In the next section, we will learn the math behind double exponential smoothing and also how to use it efficiently.

Introduction to Double Exponential Smoothing

As you saw in the previous section, simple exponential smoothing does not function well when the data has trends. In those cases, we can use double exponential smoothing. This is a more reliable method for handling data that consumes trends without seasonality than compared to other methods. This method adds a time trend equation in the formulation. Two different weights, or smoothing parameters, are used to update these two components at a time.

Holt's exponential smoothing is also sometimes called *double exponential smoothing*. The main idea here is to use SES and advance it to capture the trend component.

Assume that a series has the following:

- Level
- Trend s
- No seasonality
- Noise

Forecast = estimate level + trend at the most recent time point.

$$F_{t+k} = l_t + kT_t$$

l is the most recent level.

T_t is the most recent time point.

K is how many steps into the future we are trying to forecast.

We will have two updated equations, level and trends, and we learn both from the data.

Here is the updated equation for level:

$$L_t = \alpha Y_t + (1-\alpha) (L_{t-1} + T_{t-1})$$

L_{t-1} adjusts the previous level.

T_{t-1} adds the trend.

$(L_{t-1} + T_{t-1})$ adjusts the previous level by adding a trend.

Here is the updated equation for trends:

$$T_t = \beta (L_t - L_{t-1}) + (1-\beta) T_{t-1}$$

Take the trend in the previous time period, T_{t-1} , and update it by showing the difference in most recent level estimates: $L_t - L_{t-1}$. This setup allows the trend to vary over time adaptively.

β is controlling the speed of adjusting the trend. If the trend changes quickly in the series, we want it to learn faster, so we have this:

$$\alpha > 0 < 1$$

$$\beta > 0 < 1$$

We can adjust Holt's exponential smoothing forecast formula to capture either an additive or multiplicative trend.

Here is the additive trend:

Forecast = most recent estimate level + estimate trend.

$$F_{t+k} = L_t + K T_t$$

Here is the multiplicative trend:

Forecast = most recent estimate level + estimate trend.

$$F_{t+k} = L_t \times (T_t)^k$$

Dampening means decreasing future time trends on a straight line (no trend). The predictions generated by Holt's linear method show a constant trend (increasing or decreasing) in the future.

Double Exponential Smoothing in Action

In the previous section, you learned about the math behind double exponential smoothing. Now let's implement it on a time-series dataset.

Import the required libraries and load the CSV data, which is Facebook stock data from 2014 to 2019.

```
import pandas as pd
import numpy as np
from sklearn import metrics
from timeit import default_timer as timer
from statsmodels.tsa.api import ExponentialSmoothing,
SimpleExpSmoothing, Holt
df = pd.read_csv(r'\Data\FB.csv')
```

Let's take a sneak peek at the data (see Figure 3-5).

```
Print(df)
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2014-12-08	76.180000	77.250000	75.400002	76.519997	76.519997	25733900
1	2014-12-09	75.199997	76.930000	74.779999	76.839996	76.839996	25358600
2	2014-12-10	76.650002	77.550003	76.070000	76.180000	76.180000	32210500
3	2014-12-11	76.519997	78.519997	76.480003	77.730003	77.730003	33462100
4	2014-12-12	77.160004	78.879997	77.019997	77.830002	77.830002	28091600
5	2014-12-15	78.459999	78.580002	76.559998	76.989998	76.989998	29396500
6	2014-12-16	76.190002	77.389999	74.589996	74.690002	74.690002	31554600
7	2014-12-17	75.010002	76.410004	74.900002	76.110001	76.110001	29203900
8	2014-12-18	76.889999	78.400002	76.510002	78.400002	78.400002	34222100
9	2014-12-19	78.750000	80.000000	78.330002	79.879997	79.879997	43335000
10	2014-12-22	80.080002	81.889999	80.000000	81.449997	81.449997	31395800

Figure 3-5. First ten rows of loaded dataset

Modeling will be done only for the Close column in the dataset. Let's perform a test train split.

The train will have all the data except the last 30 days. The test contains only the last 30 days to evaluate against predictions.

```
train = df.Close[0:-30]
test = df.Close[-30:]
```

Create a function that has all the required evaluation metrics, which will give us results in one go.

```
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end='
\n\n')
```

The following snippet is a simple grid search. First we are defining a range of parameters for `smoothing_level`, `smoothing_slope`, `damping_slope`, and `damped`, and then we are trying to find the best possible combination that yields at least RMSE and R2.

CHAPTER 3 SMOOTHING METHODS

```
from sklearn.model_selection import ParameterGrid

param_grid = {'smoothing_level': [0.10, 0.20,.30,.40,.50,.60,
        .70,.80,.90], 'smoothing_slope':[0.10, 0.20,.30,
        .40,.50,.60,.70,.80,.90],
        'damping_slope': [0.10, 0.20,.30,.40,.50,.60,.70,
        .80,.90],'damped' : [True, False]}

pg = list(ParameterGrid(param_grid))
```

Let's discuss the double explanation smoothing algorithm's hyperparameters for reference.

Here are the Holt parameters:

endog (array-like): Time series

exponential (bool, optional): Type of trend component

damped (bool, optional): Whether the trend component should be damped

Here are the fit parameters:

smoothing_level (float, optional): This is the alpha value of the simple exponential smoothing. If the value is set, then this setting will be used as the value.

smoothing_slope (float, optional): This is the beta value of the Holt trend method. If the value is set, then this setting will be used as the value.

damping_slope (float, optional): This is the phi value of the damped method. If the value is set, then this setting will be used as the value.

Optimized (bool, optional): This specifies whether the values that have not been set be optimized automatically.

```

df_results_moni = pd.DataFrame(columns=['smoothing_level',
'smoothing_slope', 'damping_slope','damped','RMSE','r2'])
start = timer()

for a,b in enumerate(pg):
    smoothing_level = b.get('smoothing_level')
    smoothing_slope = b.get('smoothing_slope')
    damping_slope = b.get('damping_slope')
    damped = b.get('damped')
    print(smoothing_level, smoothing_slope, damping_
slope,damped)
    fit1 = Holt(train,damped =damped ).fit(smoothing_
level=smoothing_level, smoothing_slope=smoothing_slope,
damping_slope = damping_slope, optimized=False)
    #fit1.summary
    z = fit1.forecast(30)
    print(z)
    df_pred = pd.DataFrame(z, columns=['Forecasted_result'])
    RMSE = np.sqrt(metrics.mean_squared_error(test, df_
pred.Forecasted_result))
    r2 = metrics.r2_score(test, df_pred.Forecasted_result)
    print(f' RMSE is {np.sqrt(metrics.mean_squared_error(test,
df_pred.Forecasted_result))}')
    df_results_moni = df_results_moni.append({'smoothing_level':
smoothing_level, 'smoothing_slope':smoothing_slope,
'damping_slope' :damping_slope,'damped':damped,'RMSE':
RMSE,'r2':r2}, ignore_index=True)
end = timer()
print(f' Total time taken to complete grid search in seconds:
{((end - start)})')

print(f' Below mentioned parameter gives least RMSE and r2')
df_results_moni.sort_values(by=['RMSE','r2']).head(1)

```

CHAPTER 3 SMOOTHING METHODS

See Figure 3-6 and Figure 3-7.

Below mentioned parameter gives least RMSE and r2

smoothing_level	smoothing_slope	damping_slope	damped	RMSE	r2
806	0.9	0.6	0.1	False	2.176486 0.741182

Figure 3-6. Best parameter selected from custom grid search

```
fit1 = Holt(train,damped =False).fit(smoothing_level=0.9,  
smoothing_slope=0.6, damping_slope = 0.1, optimized=False)
```

```
Forecast_custom_pred = fit1.forecast(30)  
fit1.summary()
```

Holt Model Results

Dep. Variable:	endog	No. Observations:	1229
Model:	Holt	SSE	12220.966
Optimized:	False	AIC	2830.954
Trend:	Additive	BIC	2851.410
Seasonal:	None	AICC	2831.023
Seasonal Periods:	None	Date:	Sun, 31 May 2020
Box-Cox:	False	Time:	10:53:56
Box-Cox Coeff.:	None		
	coeff	code	optimized
smoothing_level	0.9000000	alpha	False
smoothing_slope	0.6000000	beta	False
initial_level	76.519997	l.0	False
initial_slope	0.3199990	b.0	False

Figure 3-7. Custom grid search model summary

Now let's fit the double exponential smoothing model custom grid search results and evaluate the results. See Figure 3-8.

```
timeseries_evaluation_metrics_func(test,Forecast_custom_pred)

Evaluation metric results:-
MSE is : 4.737090445686646
MAE is : 1.8121525217302568
RMSE is : 2.1764858018573534
MAPE is : 0.9266024548104845
R2 is : 0.7411820826314564
```

Figure 3-8. Results after using custom grid search parameters

Let's check whether the double exponential smoothing is able to find the best parameters by itself by adjusting a few settings, as follows:

- `optimized = True` estimates model parameters by maximizing the log likelihood.
- `use_brute= True` searches for good starting values using a brute-force (grid) optimizer.

```
fitESAUTO = Holt(train).fit(optimized= True, use_brute = True)
fitESAUTO.summary()
```

See Figure 3-9, Figure 3-10, and Figure 3-11.

CHAPTER 3 SMOOTHING METHODS

Holt Model Results

Dep. Variable:	endog	No. Observations:	1229
Model:	Holt	SSE	8955.552
Optimized:	True	AIC	2448.884
Trend:	Additive	BIC	2469.339
Seasonal:	None	AICC	2448.952
Seasonal Periods:	None	Date:	Sun, 31 May 2020
Box-Cox:	False	Time:	10:54:14
Box-Cox Coeff.:	None		
	coeff	code	optimized
smoothing_level	0.9796487	alpha	True
smoothing_slope	0.000000	beta	True
initial_level	76.503540	i.0	True
initial_slope	0.0893640	b.0	True

Figure 3-9. Automated grid search model summary

```
fitESAUTOpred = fitESAUTO.forecast(30)
timeseries_evaluation_metrics_func(test,fitESAUTOpred)

Evaluation metric results:-
MSE is : 69.53371207295187
MAE is : 7.522860460944151
RMSE is : 8.338687670907927
MAPE is : 3.8137603870283012
R2 is : -2.799076828269582
```

Figure 3-10. Results after using automated grid search parameters

```
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.rcParams["figure.figsize"] = [16,9]
```

```
plt.plot( train, label='Train')
plt.plot(test, label='Test')
plt.plot(fitESAUTOpred, label='Automated grid search')
plt.plot(Forecast_custom_pred, label='Double Exponential
Smoothing with custom grid search')
plt.legend(loc='best')
plt.show()
```

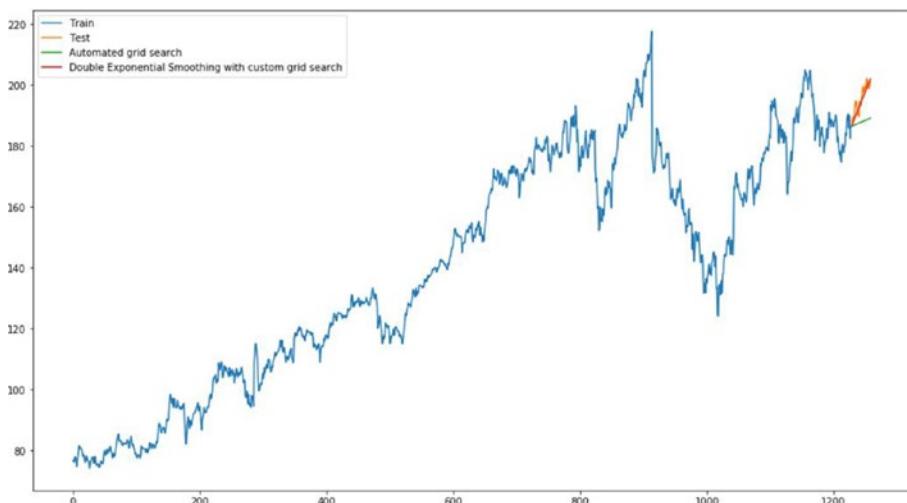


Figure 3-11. Line plot depicting results comparision between different configurations of double exponential smoothing

From the evaluation metrics and graph, we can see that the double exponential smoothing performed significantly better than simple exponential smoothing.

In the next section, we will learn the math behind triple exponential smoothing and how to use it efficiently.

Introduction to Triple Exponential Smoothing

Triple exponential smoothing is a forecasting method that enforces exponential smoothing three times. This method can be applied when the data consumes trends and seasonality over time. It includes all smoothing component equations such as trends and seasonality. Seasonality comprises two different types, such as additive and multiplicative, which is a similar operation in mathematics. The Winter method uses the idea of the Holt method and adds seasonality. Winter was a student of Holt and extended his approach by adding an additional equation to update seasonality. This triple exponential smoothing is also known as the Holt-Winters method.

Let's assume that a series has the following:

- Level
- Trend
- Seasonality
- Noise

The forecasting equation can be adjusted to handle additive or multiplicative trends and seasonality.

This is the additive seasonality model:

Forecast = estimate level + trend + seasonality at most recent time point.

$$F_{t+k} = l_t + kT_t + S_{t+k-M}$$

This is the multiplicative seasonality model with the additive trend:

Forecast = estimate (level × trend) × seasonality at most recent time point.

$$F_{t+k} = (l_t + kT_t) S_{t+k-M}$$

Assume our series contains the level, trend, and seasonality with M seasons with noise.

In Holt-Winter exponential smoothing, we have three smoothing constants.

$$\text{Level: } L_t = \alpha (y_t / S_{t-M}) + (1 - \alpha) (L_{t-1} + T_{t-1})$$

S is a seasonal component. When we y_t / S_{t-M} , we are the deseasonalizing value of y_t .

In the level equation, we are updating the previous level by L_{t-1} , adding T_{t-1} , and then combining and then combining the deseasonalizing value of y_t .

$$\text{Trend: } T_t = \beta (L_t - L_{t-1}) + (1 - \beta) T_{t-1} \text{ (Additive trend)}$$

We can update the previous trend by considering the latest difference between levels.

$$\text{Seasonality: } S_t = \gamma (Y_t / L_t) + (1 + \gamma) S_{t-M} \text{ (multiplicative seasonality)}$$

Y_t is divided by level component L_t . This gives the detrended value of Y_t .

So, the seasonality is updated by combining the most recent seasonal component S_{t-M} with the detrended value of Y_t .

Triple Exponential Smoothing in Action

In the previous section, you learned about the math behind triple exponential smoothing. Now let's implement it on a time-series dataset.

Import the required libraries and load the CSV data, which is Facebook stock data from 2014 to 2019.

```
from statsmodels.tsa.api import ExponentialSmoothing
import pandas as pd
import numpy as np
from sklearn import metrics
from timeit import default_timer as timer
df = pd.read_csv(r'\Data\FB.csv')
```

CHAPTER 3 SMOOTHING METHODS

Let's take a sneak peek at the data (Figure 3-12).

```
Print(df)
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2014-12-08	76.180000	77.250000	75.400002	76.519997	76.519997	25733900
1	2014-12-09	75.199997	76.930000	74.779999	76.839996	76.839996	25358600
2	2014-12-10	76.650002	77.550003	76.070000	76.180000	76.180000	32210500
3	2014-12-11	76.519997	78.519997	76.480003	77.730003	77.730003	33462100
4	2014-12-12	77.160004	78.879997	77.019997	77.830002	77.830002	28091600
5	2014-12-15	78.459999	78.580002	76.559998	76.989998	76.989998	29396500
6	2014-12-16	76.190002	77.389999	74.589996	74.690002	74.690002	31554600
7	2014-12-17	75.010002	76.410004	74.900002	76.110001	76.110001	29203900
8	2014-12-18	76.889999	78.400002	76.510002	78.400002	78.400002	34222100
9	2014-12-19	78.750000	80.000000	78.330002	79.879997	79.879997	43335000
10	2014-12-22	80.080002	81.889999	80.000000	81.449997	81.449997	31395800

Figure 3-12. First ten rows of dataset

Modeling will be done only for the Close column in the dataset. Make a copy of the data, and let's perform a test train split.

The train will have all the data except the last 30 days of data. The test contains only the last 30 days of data to evaluate against predictions.

```
X = df['Close']
test = X.iloc[-30:]
train = X.iloc[:-30]
```

Let's use the triple exponential smoothing algorithm's hyperparameters for reference.

Here are Holt-Winter's exponential smoothing parameters:

endog (array-like): Time series
trend ({"add", "mul", "additive", "multiplicative", None}, optional): Type of trend component

damped (bool, optional): Whether the trend component should be damped

seasonal ({“add”, “mul”, “additive”, “multiplicative”, None}, optional): Type of seasonal component

seasonal_periods (int, optional): The number of seasons to consider for Holt-Winters

Here are the fit parameters:

smoothing_level (float, optional): This is the alpha value of the simple exponential smoothing. If the value is set, then this setting will be used as the value.

smoothing_slope (float, optional): This is the beta value of the Holt trend method. If the value is set, then this setting will be used as the value.

smoothing_seasonal (float, optional): This is the gamma value of the Holt-Winter seasonal method. If the value is set, then this setting will be used as the value.

damping_slope (float, optional): This is the phi value of the damped method. If the value is set, then this setting will be used as the value.

optimized (bool, optional): This specifies whether the values that have not been set earlier should be optimized automatically.

use_boxcox ({True, False, ‘log’, float}, optional): This specifies whether the box cox transform should be applied to the data first. If log, then it applies the log. If float, then it uses a lambda equal to float.

remove_bias (bool, optional): This specifies whether the bias should be removed from the forecast values and fitted values before being returned. It does this by enforcing average residuals equal to zero.

use_basin hopping (bool, optional): This specifies whether the optimizer should try harder using basin hopping to find optimal values.

Create a function that has all the required evaluation metrics, which will give us results in one go.

```
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end='
\n\n')
```

The following snippet is a simple grid search. First we are defining the range of parameters for seasonal, trend, seasonal_periods, smoothing_

level, smoothing_slope, damping_slope, damped, use_boxcox, remove_bias, and use_basin hopping. Then we are trying to find the best possible combination that yields us at least RMSE and R2.

Executing the following code will take some time as our search space is larger:

```
from sklearn.model_selection import ParameterGrid
param_grid = {'trend': ['add', 'mul'], 'seasonal' :['add', 'mul'], 'seasonal_periods':[3,6,12], 'smoothing_level': [0.10, 0.20,.30,.40,.50,.60,.70,.80,.90], 'smoothing_slope':[0.10, 0.20,.30,.40,.50,.60,.70,.80,.90],
              'damping_slope': [0.10, 0.20,.30,.40,.50,.60,.70,.80,.90], 'damped' : [True, False], 'use_boxcox':[True, False],
              'remove_bias':[True, False], 'use_basinhopping':[True, False]}

pg = list(ParameterGrid(param_grid))

df_results_moni = pd.DataFrame(columns=['trend','seasonal_periods','smoothing_level', 'smoothing_slope',
                                         'damping_slope','damped','use_boxcox','remove_bias','use_basinhopping','RMSE','r2'])

start = timer()
print('Starting Grid Search..')
for a,b in enumerate(pg):
    trend = b.get('trend')
    smoothing_level = b.get('smoothing_level')
    seasonal_periods = b.get('seasonal_periods')
    smoothing_level = b.get('smoothing_level')
    smoothing_slope = b.get('smoothing_slope')
    damping_slope = b.get('damping_slope')
    damped = b.get('damped')
```

CHAPTER 3 SMOOTHING METHODS

```
use_boxcox = b.get('use_boxcox')
remove_bias = b.get('remove_bias')
use_basin hopping = b.get('use_basin hopping')
#print(trend,smoothing_level, smoothing_slope,damping_
slope,damped,use_boxcox,remove_bias,use_basin hopping)
fit1 = ExponentialSmoothing(train,trend=trend,
damped=damped,seasonal_periods=seasonal_periods ).
fit(smoothing_level=smoothing_level,
smoothing_slope=smoothing_slope, damping_
slope = damping_slope,use_boxcox=use_
boxcox,optimized=False)

#fit1.summary
z = fit1.forecast(30)
#print(z)
df_pred = pd.DataFrame(z, columns=['Forecasted_result'])
RMSE = np.sqrt(metrics.mean_squared_error(test, df_
pred.Forecasted_result))
r2 = metrics.r2_score(test, df_pred.Forecasted_result)
#print( f' RMSE is {np.sqrt(metrics.mean_squared_
error(test, df_pred.Forecasted_result))}' )
df_results_moni = df_results_moni.append({'trend':tr
end,'seasonal_periods':seasonal_periods,'smoothing_
level':smoothing_level, 'smoothing_slope':smoothing_slope,
'damping_slope':damping_slope,'damped':damped,'use_boxcox':use_boxcox,'use_
basin hopping':use_basin hopping,'RMSE':RMSE,
'r2':r2}, ignore_index=True)

print('End of Grid Search')
end = timer()
print(f' Total time taken to complete grid search in seconds:
{(end - start)}')
```

```

Starting Grid Search..
End of Grid Search
Total time taken to complete grid search in seconds: 9433.836381199999

```

After completing the custom grid search, all the possible combinations with their results are stored in a df_results_moni DataFrame (Figure 3-13).

```

print(f' Below mentioned parameter gives least RMSE and r2')
df_results_moni.sort_values(by=['RMSE','r2']).head(1)

```

	trend	seasonal_periods	smoothing_level	smoothing_slope	damping_slope	damped	use_boxcox	remove_bias	use_basin hopping	RMSE	r2
70605	mul	3	0.9	0.6	0.1	False	False	NaN	True	2.162275	0.744551

Figure 3-13. Best parameters from custom grid search

Fit the model with grid search parameters and evaluate the results. See Figure 3-14.

```

fit1 = ExponentialSmoothing(train,trend='mul',
damped=False,seasonal_periods=3 ).fit(smoothing_level=0.9,
smoothing_slope=0.6, damping_slope = 0.6,
use_boxcox=False,use_basin hopping = True,
optimized=False)

Forecast_custom_pred = fit1.forecast(30)
fit1.summary()
timeseries_evaluation_metrics_func(test,Forecast_custom_pred)

Evaluation metric results:-
MSE is : 4.675431534981652
MAE is : 1.7892531792844002
RMSE is : 2.1622746206209915
MAPE is : 0.9154984217657036
R2 is : 0.7445509080821948

```

Figure 3-14. Results after using custom grid search parameters

CHAPTER 3 SMOOTHING METHODS

Let's check whether triple exponential smoothing is able to find the best parameters by itself by adjusting a few settings such as the following (see Figure 3-15):

- `optimized = True` estimates model parameters by maximizing the log likelihood.
- `use_brute = True` searches for good starting values using a brute-force (grid) optimizer.

```
fitESAUTO = ExponentialSmoothing(train).fit(optimized= True,  
use_brute = True)
```

```
fitESAUTO.summary()
```

ExponentialSmoothing Model Results

Dep. Variable:	endog	No. Observations:	1229
Model:	ExponentialSmoothing	SSE	8965.774
Optimized:	True	AIC	2446.285
Trend:	None	BIC	2456.513
Seasonal:	None	AICC	2446.318
Seasonal Periods:	None	Date:	Sun, 29 Dec 2019
Box-Cox:	False	Time:	08:41:43
Box-Cox Coeff.:	None		
	coeff	code	optimized
smoothing_level	0.9808449	alpha	True
initial_level	76.519998	1.0	True

Figure 3-15. Automated grid search model summary

See Figure 3-16 and Figure 3-17.

```
fitESAUTOpred = fitESAUTO.forecast(30)

timeseries_evaluation_metrics_func(test,fitESAUTOpred)

Evaluation metric results:-
MSE is : 97.68044338619876
MAE is : 8.909413652138449
RMSE is : 9.883341711496104
MAPE is : 4.516280753437192
R2 is : -4.336914972326928
```

Figure 3-16. Results after using automated grid search parameters

```
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.rcParams["figure.figsize"] = [16,9]
plt.plot( train, label='Train')
plt.plot( test, label='Test')
plt.plot(fitESAUTOpred, label='Automated grid search')
plt.plot(Forecast_custom_pred, label='Triple Exponential
Smoothing with custom grid search')
plt.legend(loc='best')
plt.show()
```

CHAPTER 3 SMOOTHING METHODS

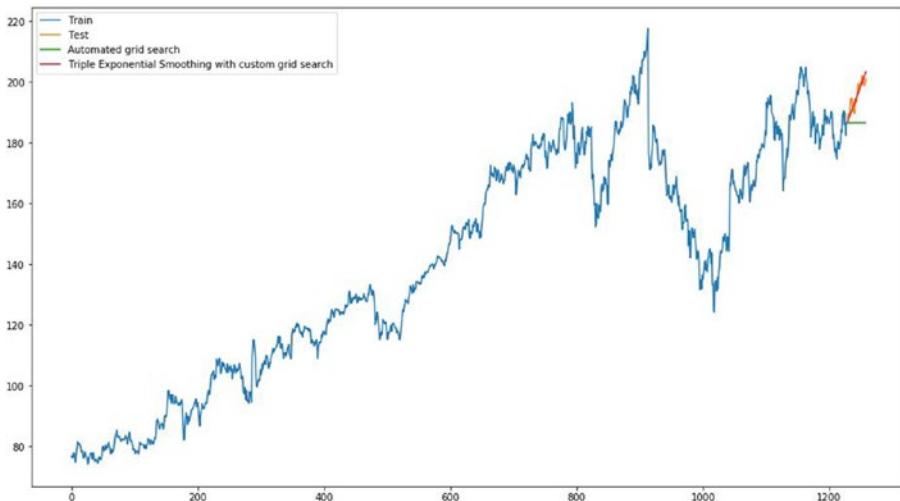


Figure 3-17. Line plot depicting results comparision between different configurations of triple exponential smoothing

By carefully observing evaluation metrics and graphs, we can conclude that triple exponential smoothing slightly outperforms double exponential smoothing.

Therefore, you can conclude the following:

- If there is no presence of trend or seasonality, then use simple exponential smoothing.
- If there is a presence of trend and no seasonality, use double exponential smoothing.
- If there is a presence of trend and seasonality, then use triple exponential smoothing.

Summary

In this chapter, you learned about the math behind simple, double, and triple exponential smoothing and how to use custom and automated grid search to find the best hyperparameters so that models can yield the best results. In the next chapter, you will learn about various regression extension techniques for time-series data.

CHAPTER 4

Regression Extension Techniques for Time-Series Data

In this chapter, you will learn how to leverage regression techniques to solve time-series problems efficiently. *Regression* is a supervised learning technique in machine learning where you try to estimate target variables using one or multiple regressors. In this chapter, you will learn what *stationary* means in time-series data as well as how to make data stationery, how to interpret p-values, and how to test whether a series is stationary. Finally, we'll explore the AR, MA, ARIMA, SARIMA, SARIMAX, VAR, and VARMA techniques and solve some problems with real-world datasets.

Types of Stationary Behavior in a Time Series

Stationary data means that the statistical properties of the particular process do not vary with time. There are five forms of stationary time-series data.

1. **Strict or strong stationary:** A stochastic process is an unconditional joint probability distribution that does not change when shifted in time. This means that the distribution is the same through time.
2. **First-order stationary:** This series has constant means that never change with time. Other statistical properties may be changing, such as variance.
3. **Weak (second-order) stationary:** In weak stationery data, mean, variance, and covariance are constant throughout the time series. It is also known as covariance/mean stationarity.
4. **Trend stationary:** This series varies around the trends (the statistical property mean is varied). These trends may be linear or quadric.
5. **Difference stationary:** This type of series has one or more differences when the data is in a time series.

So, if data exhibits the behaviors of constant mean, variance, and autocorrelation, then it is said to be *stationary*. It is mandatory to convert your data into a stationery format to train a time-series forecasting model. For this reason, you need to know how to check whether a time series is stationary or not. Figure 4-1 and Figure 4-2 show some data before and after making it stationary.

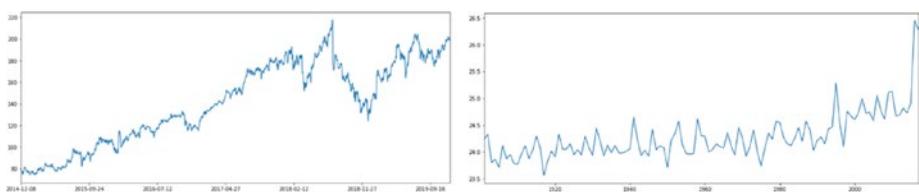


Figure 4-1. Before stationary operation

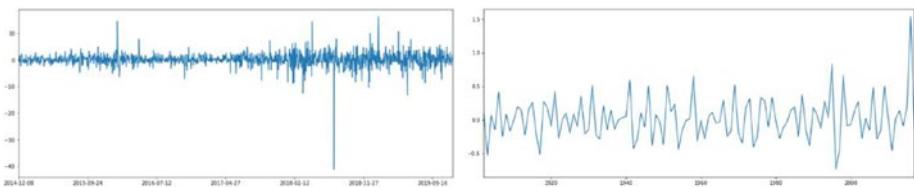


Figure 4-2. After stationary operation

Making Data Stationery

When time-series data is nonstationary, it means it has trends and seasonality patterns that should be removed. By making data stationary, our data will have a constant mean and variance, and most of the statistical concepts can be applied. To check whether a time series is stationary or not, we can use three types of methods.

- Plots
- Summary statistics
- Statistics tests

Using Plots

The previous graphs show the time-series data in a visual form. This helps us understand whether a time series is linearly increasing or decreasing with time, which is known as a *nonstationary* time series. A plot that is not increasing or decreasing and shows constant growth over time is known as a *stationary* time series.

Using Summary Statistics

Summary statistics determine all the essential factors such as mean, standard deviation, variance, min, max, and IQR, which can assist in distinguishing behavior from the data. It helps us to evaluate the seasons

or random patterns in the data so we can check for significant differences. Whether the time series is stationary or not, we can split the data in a time series into different groups, where we can compare the difference factors such as mean, variance, and standard deviation. If the factor that is observed varies and is statistically significant, we say that time series is *nonstationary*. There are many domains, such as sales, finance, retail, healthcare, etc., where we will find irregular behavior patterns with mean and variance continuously varying because of seasonality and increasing trends.

Using Statistics Unit Root Tests

A statistical test makes a strong assumption about the data. Likewise, we can make some assumptions about time-series data. For this purpose, we have to check that a time-series null hypothesis has been rejected or not. So, we can elucidate the result in order for a particular problem to be meaningful. Statistical unit root tests contain the following five methods, each of which contains a unit root stationary test:

- **Dickey-Fuller (DF) test:** This is based on linear regression. Serial correlation is a big issue of this method.
- **Augmented Dickey-Fuller test:** This solves the serial correlation problem of a DF test and handles big and complex models. This method is widely used.
- **The Schmidt-Phillips test:** This comprises the coefficients of the deterministic variables in the null and alternative hypotheses. Substitutes of this method are ro-test and tau-test.
- **The Phillips-Perron (PP) test:** This test is a betterment of the Dickey-Fuller test, and it embellishes the test for autocorrelation and heteroscedasticity in the errors.

- **KPSS test:** This is the reverse of the ADF test where the null hypothesis is the process of stationary trends and an alternative hypothesis for unit roots.

The concept behind the unit root stationary test is that the statistical property of a given time series is not constant with time. In this case, we are using this statistical test to check whether a time series is either stationary or nonstationary.

Here we'll explain the unit root mathematical equations.

- Mathematically, the equation denotes the following:

$$Y_t = a * Y_{t-1} + \epsilon_t$$

$$Y_{t-1} = a * y_{t-2} + \epsilon_{t-1}$$

where Y_t is a time instance of t , and ϵ_t is an error term.

To calculate Y_t , we have to calculate Y_{t-1} .

- For all observations, the following applies:

$$Y_t = a^n * y_{t-n} + \epsilon_{t-1} * a^i$$

- If the amount of a contains 1 in the previous formulation, the predication will be Y_{t-n} , and it is a sum of all errors for $t-n$ to t (this indicates that variance is increasing with time). This is known as a *unit root* in the domain of time series.
- For a stationary time series, a variance must not be the functional part of the time. Therefore, unit root tests check for the existence of a unit root in the time series by monitoring the amount of $a = 1$.

Next you will learn how to interpret p-value.

Interpreting the P-value

In the unit root stationary test, we have to deal with the null hypothesis (H_0), which is not the stationary and alternative hypothesis (H_1), which is the stationary time series. When we perform this unit root stationary test, the results will be in p-value format. So, it is essential to understand how to interpret the p-value. The p-value helps to determine the significance of the outcome.

If the p-value is below the threshold, then we reject the null hypothesis, which means that the time series is stationary. If the p-value exceeds the threshold and we fail to reject the null hypothesis, it means that the time series is nonstationary. An ADF test looks at the test statistic, the p-value, and the critical values found at 1%, 2.5%, 5%, and 10% confidence intervals.

The following shows that if the significance level is set to 0.05, then the corresponding confidence level is 95%:

P-value > 0.05

Fail to reject the null hypothesis(H_0)

- It indicate that data has unique roots and **time series is non-stationary**.

P-Value <= 0.05

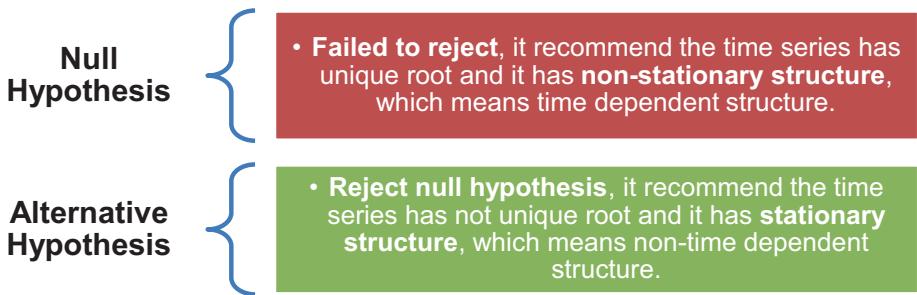
Reject the null hypothesis(H_0)

- It indicates that data does not have unique roots and **time series is Stationary**.

Augmented Dickey-Fuller Test

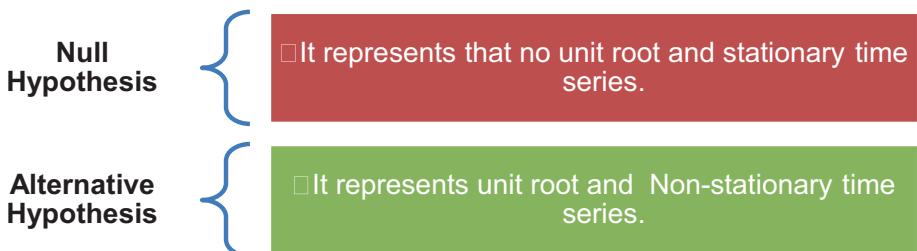
The augmented Dickey-Fuller (ADF) test is a notable and plausible statistical test for stationary checking. It can be utilized to decide on the existence of the unit root in a domain of the series, and in addition, it helps us to understand whether the time series is stationary. The ADF test includes the null hypothesis (H_0) that the unit root is present in a time series. The alternative hypothesis is utilized for the stationary version of the time series. It is an augmented version of the Dickey-Fuller test and is best used for a large and more complex set of time-series data.

This test uses a negative number. The more negative the number is, the higher the chance of rejecting the hypothesis that there is a unit root at some level of confidence.



Kwiatkowski-Phillips-Schmidt-Shin Test

Kwiatkowski-Phillips-Schmidt-Shin (KPSS) tests are utilized to examine the null hypothesis that the time series is stationary around a conclusive trend (i.e., trend stationary) compared to the alternative of a unit root. The presence of a unit root is not the null hypothesis but the alternative hypothesis, which is a reverse form of the ADF test. Furthermore, in a KPSS test, the nonappearance of a unit root is not proof of stationarity but, by design, of trend stationarity. This is mostly utilized for trend stationary tests. *Trend stationary* data means a series that has no unit root but shows a trend.



Here is an interpretation of a KPSS test:

The test statistic, the p-value, and the critical value were found at 1%, 2.5%, 5%, and 10% confidence intervals.

P-value > 0.05

Reject the null hypothesis(H0)

- It indicate that data has no unit roots and **time series is stationary**.

P-Value <= 0.05

If fail to reject the null hypothesis(H1)

- It indicates that data has unit roots and **time series is non-stationary**.

Using Stationary Data Techniques

In the previous section, you learned how to interpret p-values with the help of ADF tests and KPSS tests; in this section, let's look at various ways of making data stationary.

Differencing

Differencing (lag difference) is the process of transforming a time series to stabilize the mean. We have several ways to identify the time series, such as a line plot, which represents the series over time. Within these trends, seasonality and random walk can be observed, which is a change over a period of time, and this behavior is considered a nonstationary time series. The clear rule states that it needs to remove trends and seasonality in the data from the training time series forecasting model.

The following are the reasons for differencing:

- To convert nonstationary data into a stationary time series
- To remove seasonal trends
 - Take the fourth difference for quarterly data
 - Take the 12th difference for monthly data

Random Walk

A random walk is the behavior when the time series shows irregular growth. When the growth is not constant over time, it is better to predict the change that occurs from one period (Y_t) to the next (Y_{t-1}). In a different timestamp, moving from left to right, the movement that is independent of the variable takes a random step up or down. A so-called random walk illustrates that time-series future steps or directions can vary in a stochastic manner (Figure 4-3).

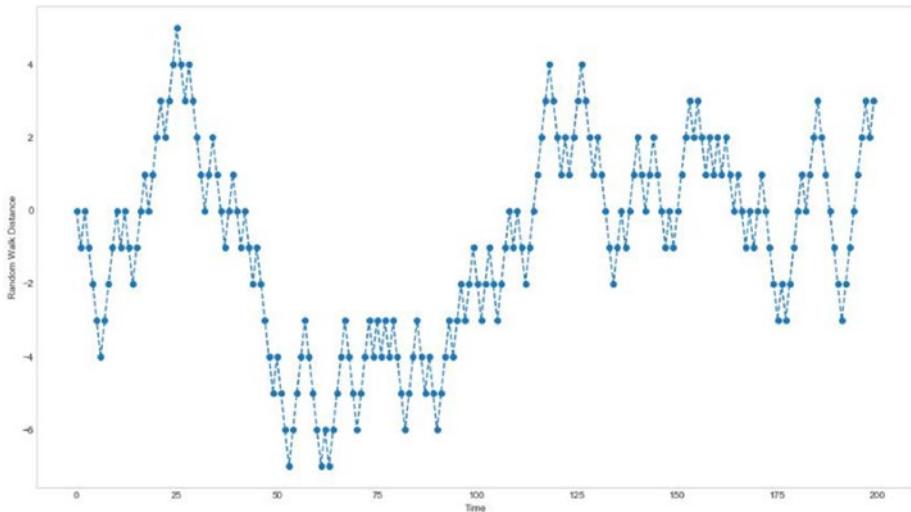


Figure 4-3. Random walk one-dimension representation

Here are the steps of random walks:

1. Start with a random walk from t_0 .
2. Randomly select the next movement, t_{n-1} .
3. Repeat step 2.

First-Order Differencing (Trend Differencing)

So, a change between two consecutive observations in the time series can be written as follows:

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

When the differenced series contains white noise (ϵ_t), the formula can be written as follows:

$$Y_t - Y_{t-1} = \epsilon_t$$

$$Y_t = Y_{t-1} + \epsilon_t$$

where ϵ_t = white noise. This is known as a *normal random walk*.

A random walk represents that a time series is nonstationary. This is mostly seen in financial, economic, and microeconomics data. A random walk has mostly long durations of trend ups and down and uncertain and unpredictable changes. Figure 4-4 illustrates the random walk behavior of a stock price.



Figure 4-4. Random walk behavior of Infosys Ltd Stock price (source: Money Control)

For instance, we can see that the stock price of Infosys Ltd. has gone up and down for the last six months. This is random walk behavior. Any future moment is not predictable because of the haphazard ups and downs.

So, here it is clear that the random time series has nonzero mean values. In that case, the final formula is known as *random walk with drift*, as shown here:

$$Y_t = c + Y_{t-1} + \varepsilon_t$$

where c is the average of change between two observations.

- If c is positive, then the average change will rise in Y_t , and Y_t will move upward.
- If c is negative, the Y_t value will move downward.

Notes:

- First-order differencing in a time series will remove a linear trend (i.e., differences=1).
- Second-order differencing will remove a quadratic trend (i.e., differences=2).
- In addition, first-order differencing in a time series at a lag equal to the period will remove a seasonal trend.

Second-Order Differencing (Trend Differencing)

Second-order differencing is a technique used to make first-order differencing data stationary when the first-order differencing has failed. So, it's necessary to apply second-order differencing to obtain a stationary series.

$$\begin{aligned} Y''_t &= Y'_t - Y'_{t-1} \\ &= (Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2}) \\ &= Y_t - 2Y_{t-1} + Y_{t-2} \end{aligned}$$

This is second-order differencing, Y'' , which has $t-2$ values. This is known as *double changes* in the original series.

Seasonal Differencing

A *seasonal difference* is the difference between an observation and the corresponding observation from the previous year.

First-Order Differencing for Seasonal Data

Seasonal differencing is similar to first-order differencing in that we are calculating current and previous observation differences for the same season. It can be written as follows:

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

where s is the number of seasons, which is known as *lag-s* differencing. We can make a difference in observation after s -lag periods.

If we observed white noise in seasonal differencing, then the model can be written as follows:

$$Y_t = Y_{t-s} + \varepsilon_t$$

When we try to forecast the value with this approach, it is equal to the last observation from that season. This model gives seasonal naïve forecasting results.

Second-Order Differencing for Seasonal Data

To classify the seasonal differences from regular differences, we sometimes allude to average differences as *first differences*, which signifies a difference at lag1. In the real world, it tends to occur after applying the first-order differencing. A time series may not get the state of the stationary series, so in that event, it is vital to use second-order differencing, with differencing specified as lag2. It looks like this:

$$\begin{aligned}
 Y''_t &= Y'_t - Y'_{t-s} \\
 &= (Y_t - Y_{t-s}) - (Y_{t-1} - Y_{t-s-1}) \\
 &= Y_t - Y_{t-1} - Y_{t-s} + Y_{t-s-1}
 \end{aligned}$$

When applying regular and first-order differences, there are no distinctions from second-order that are done first. The outcome will be the same as first-order differencing.

Autoregressive Models

An autoregression model (AR) predicts future behavior based on its past data. It is when data is correlated with a consecutive sequence of a time series and the values before and after the sequence. The *autoregressive model* uses only past behavior data to forecast the value.

AR models use past values to forecast as shown here:

$$\hat{Y}_t = \mu + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} \dots + \phi_p Y_{t-p} + \varepsilon_t$$

where ε_t is white noise. This model is known as the AR(p) model, where p is the order for the autoregressive model. The AR model is easy to use to handle a wide range of time-series models. Figure 4-5 illustrates AR (1) and AR (2) behavior.

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

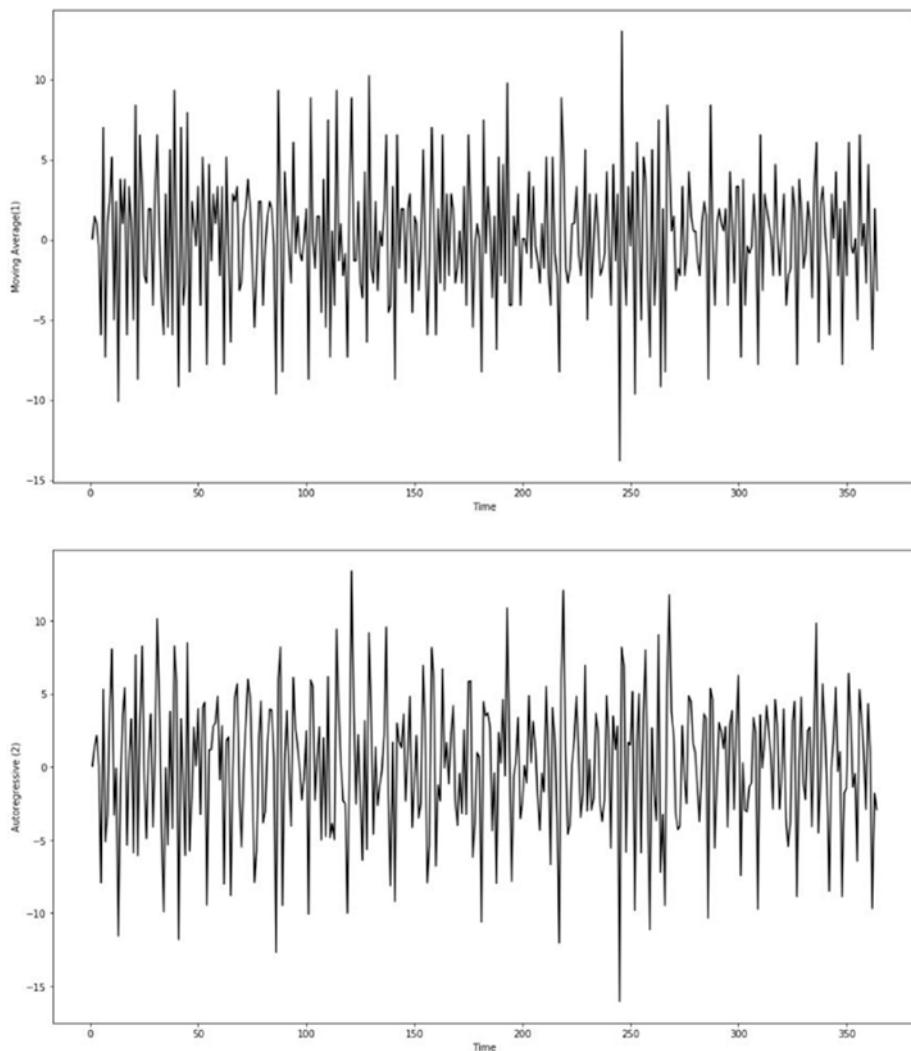


Figure 4-5. Plot after autoregressive 1 and 2

Table 4-1 shows the AR (1) model.

Table 4-1. ϕ and u Values' Influence on Data

ϕ Value	u Value	Result
0	-	White noise
1	0	Equivalent to a random walk
1	Not 0	Similar to a random walk with drift
< 0	-	Oscillates around the mean

Autoregressive models are restricted to stationary data, and some parameters are required, as defined here:

AR (1) model: $-1 < \phi_1 < 1$

AR (2) model: $-1 < \phi_1 < 1, \phi_1 + \phi_2 < 1, \phi_1 - \phi_2 < 1$

Moving Average

A *moving average* (MA) is a method to get all the trends in a time series. It is the average of any subcategory of numbers. It is utilized for long-term forecasting trends. Basically, a moving average forecasts future points by using an average of several past data points.

The moving average model practices past forecast errors:

$$\hat{Y}_t = \mu_0 + \varepsilon_t - \omega_1 \varepsilon_{t-1} - \omega_2 \varepsilon_{t-2} - \dots - \omega_q \varepsilon_{t-q}$$

where ε_t is white noise. This model is known as the MA(q) model, where q is ordered for the moving average model. The MA model is easy to use to handle a wide range of time-series models. Figure 4-6 illustrates MA (1) and MA (2) behavior.

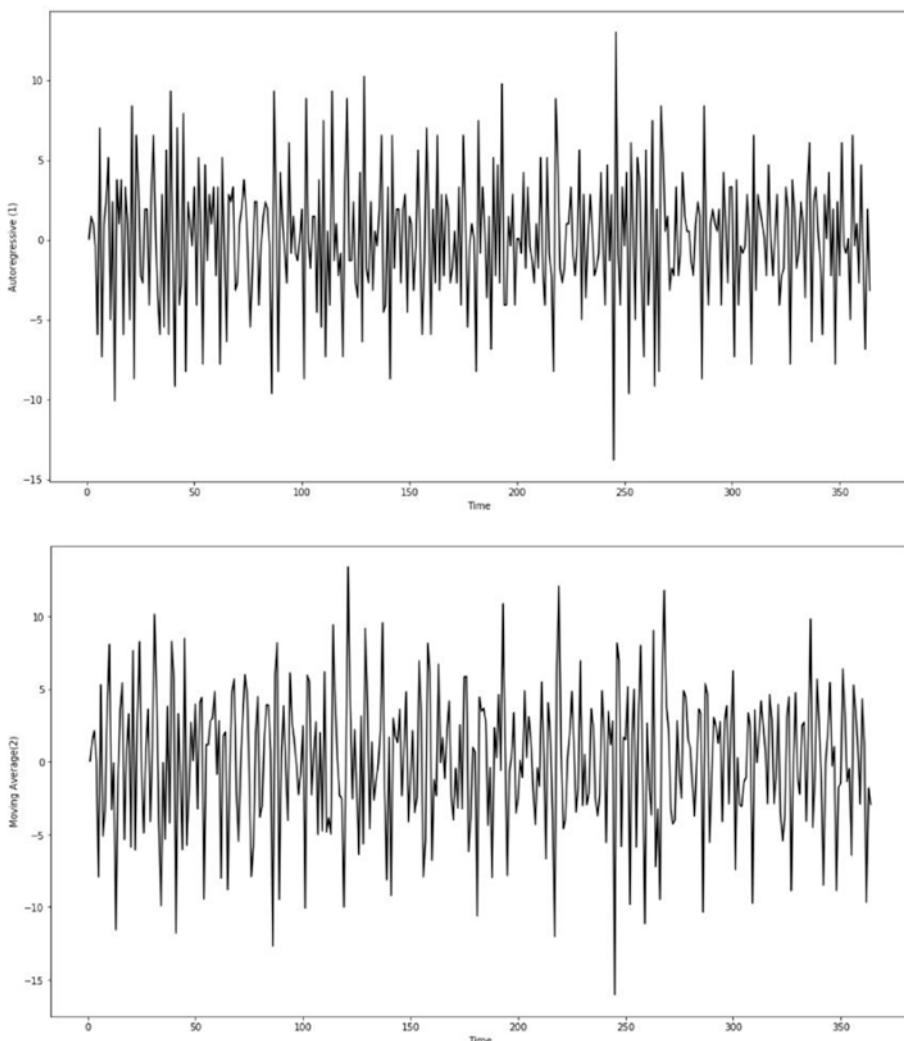


Figure 4-6. Plot after moving averages 1 and 2

The weight value has impact on the behavior of the time series.

Table 4-2 provides the different variations in the weight and its impact on the model.

Table 4-2. ω Value's Influence on Data

ω value	Result
$ \omega > 1$	Weights increase as the lags increase. The more distance in the data, the greater the influence on the current error.
$ \omega = 1$	Weights are persistent in size, and the distant data has the same influence as the recent data.
$ \omega < 1$	The most recent data has a higher weight than data from the more distant past.
< 0	Oscillates around the mean.

- Here is the MA (1) model: $-1 < \omega_1 < 1$
- Here is the MA (2) model: $-1 < \omega_1 < 1, \omega_1 + \omega_2 < 1,$
 $\omega_1 - \omega_2 < 1$

Note Here are some definitions you should understand:

ACF: The correlation of a variable with its lagged values.

PAC: The correlation of a variable with its lagged values, but after removing the effects of in-between time lags.

Correlograms: The plots of ACF and PACF against the lag length. This can give you an idea about the relation of autocorrelation between variables.

Autocorrelation and Partial Autocorrelation Functions

An *autocorrelation function* (ACF) is a method to determine the linear relationship between time t and $t-1$. After checking the ACF, helps in determining if differencing is required or not.

The data is independent in the regression model but fails when it is gathered over time, and a regression algorithm fails to catch any time trends. In this situation, the random error is positively correlated with time. So, each random error is more gradual to be similar to the last random error. This incident is called *autocorrelation* (or *serial correlation*).

If we are using the autoregressive model AR(k), then we have to determine the only correlation between Y_t and Y_{t-1} and check for a direct (linear) influence between the random variables that lie in the time series, which requires differencing and transforming the time series. After transforming the time series, we calculate the correlation, which is known as a *partial autocorrelation function* (PACF). Figure 4-7 shows a plot of each function.

Say we have time-series Y_t , which represents a Y measure over time t . The autoregressive model is when a value of the time-series data is regressed on the previous value. For instance, here is the equation for Y_t on Y_{t-1} :

$$Y_t = \mu + \phi_1 Y_{t-1} + \varepsilon$$

The coefficient of correlation among two values in time-series data is known as the ACF. For instance, the ACF for the time-series Y_t given is as follows:

$$\text{Corr}(Y_t, Y_{t-1}), k = 1, 2, \dots$$

The value of k is the time gap considering the lag. The lag 1 ($k=1$) autocorrelation is a correlation between values that are one time period away from each other.

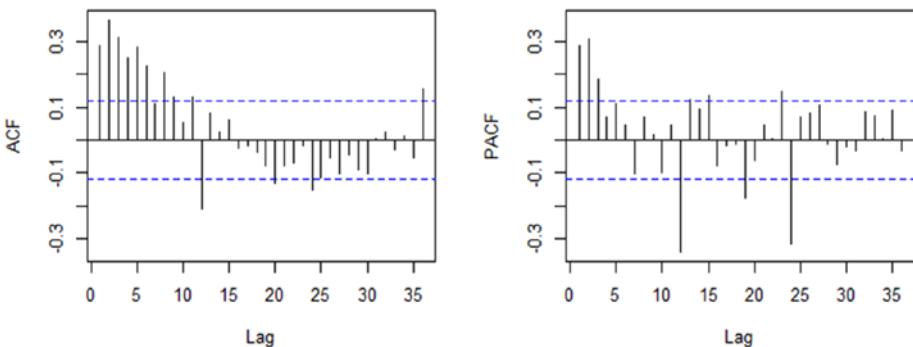


Figure 4-7. Representation of ACF and PACF plots

Introduction to ARMA

ARMA uses information obtained from the variable itself to forecast its trend, and the variable is regressed on its own past values (Table 4-3).

Here are the steps to identify whether data is showing an AR or MA signature:

1. Plot or use an ADF test to check whether a series is stationary.
2. If a time series does not have a stationary difference, check for stationary.
3. Plot ACF and PACF and use Table 4-3 to determine p and q for the model.

Table 4-3. ARMA Estimating Those Modes

Model	ACF Pattern	PACF Pattern
AR(p)	Exponential decay or damped sine wave pattern or both	Significant spike through first lag
MA(q)	Significant spike through first lag	Exponential decay
ARMA (1,1)	Exponential decay from lag 1	Exponential decay from lag 1
ARMA (p,q)	Exponential decay	Exponential decay

ARIMA has three components: the autoregression part (AR), the integration part (I), and the moving average part (MA).

Autoregressive Model

The autoregressive model is a lagged dependent variable, which contains an autoregressive term, which perhaps corrects on the grounds of habit resolve. AR is part of a time series Y_t , which contains a value that depends on some linear grouping of the previous value, which defined maximum lags (signified p). It also contains an arbitrary error term ϵ_t , given as follows:

$$\text{First-order AR: } \hat{Y}_t = \alpha + b_1 Y_{t-1}$$

$$\text{Second-order AR: } \hat{Y}_t = \alpha + b_1 Y_{t-1} + b_2 Y_{t-2}$$

$$\text{Third-order AR: } \hat{Y}_t = \alpha + b_1 Y_{t-1} + b_2 Y_{t-2} + b_3 Y_{t-3}$$

$$y_t = \varphi_1 y_{t-1} + \varphi_2 y_{t-2} + \dots + \varphi_p y_{t-p} + \epsilon_t$$

where the parameters φ_t are constants.

Moving Average

The MA part of a time-series Y_t , which is an observed value in terms of a random error and some linear grouping of previous arbitrary error terms, up to a described maximum lag (signified q).

$$\text{First order MA(1):} \quad \hat{Y}_t = \gamma + d_0 u_t + d_1 u_{t-1}$$

$$\text{Second order MA(2):} \quad \hat{Y}_t = \gamma + d_0 u_t + d_1 u_{t-1} + d_2 u_{t-2}$$

$$\text{Third order MA(3):} \quad \hat{Y}_t = \gamma + d_0 u_t + d_1 u_{t-1} + d_2 u_{t-2} + d_3 u_{t-3}$$

$$y_t = Z_t + \theta_1 Z_{t-1} + \theta_2 Z_{t-2} + \dots + \theta_q Z_{t-q}$$

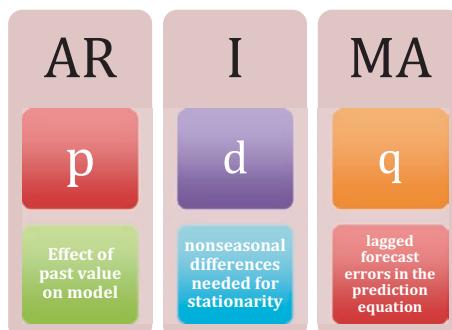
where the parameters θ_t are constants.

let's combine AR and MA

$$\text{ARMA (1,1):} \quad \hat{Y}_t = \mu + \phi_1 Y_{t-1} + d_0 u_t + d_1 u_{t-1}$$

Introduction to Autoregressive Integrated Moving Average

Autoregressive integrated moving average—also called ARIMA(p,d,q)—is a forecasting equation that can make time series stationary with the help of differencing and log techniques when required. A time series that should be differentiated to be stationary is an integrated (d) (I) series. Lags of the stationary series are classified as autoregressive (p), which is designated in (AR) terms. Lags of the forecast errors are classified as moving averages (q), which are identified in (MA) terms.



CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

A nonseasonal ARIMA model is called an ARIMA(p,d,q) model, where:

- **p** is the number of autoregressive terms.
- **d** is the number of nonseasonal differences needed for stationarity.
- **q** is the number of lagged forecast errors in the prediction equation.

Table 4-4 shows the different ARIMA model behavior with p, d, and q.

Table 4-4. Representation of p, d, q and Its Relevant Methods

p d q Differencing	Method
ARIMA (0, 0, 0) 0 0 0 $y_t = Y_t$	White noise
ARIMA (0, 1, 0) 0 1 0 $y_t = Y_t - Y_{t-1}$	Random walk
ARIMA (0, 2, 0) 0 2 0 $y_t = Y_t - 2Y_{t-1} + Y_{t-2}$	Constant
ARIMA (1, 0, 0) 1 0 0 $\hat{Y}_t = \mu + \phi_1 Y_{t-1} + \varepsilon$	AR(1): First-order regression model
ARIMA (2, 0, 0) 2 0 0 $\hat{Y}_t = \phi_0 + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} + \varepsilon$	AR(2): Second-order regression model
ARIMA (1, 1, 0) 1 1 0 $\hat{Y}_t = \mu + Y_{t-1} + \phi_1 (Y_{t-1} - Y_{t-2})$	Differenced first-order autoregressive model
ARIMA (0, 1, 1) 0 1 1 $\hat{Y}_t = Y_{t-1} - \phi_1 e_{t-1}$	Simple exponential smoothing
ARIMA (0, 0, 1) 0 0 1 $\hat{Y}_t = \mu_0 + \varepsilon_t - \omega_1 \varepsilon_{t-1}$	MA(1): First-order regression model
ARIMA (0, 0, 2) 0 0 2 $\hat{Y}_t = \mu_0 + \varepsilon_t - \omega_1 \varepsilon_{t-1} - \omega_2 \varepsilon_{t-2}$	MA(1): Second-order regression model

(continued)

Table 4-4. (continued)

	p d q Differencing	Method
ARIMA (1, 0, 1)	1 0 1 $\hat{Y}_t = \phi_0 + \phi_1 Y_{t-1} + \varepsilon_t - \omega_1 e_{t-1}$	ARMA model
ARIMA (1, 1, 1)	1 1 1 $\Delta Y_t = \phi_1 Y_{t-1} + \varepsilon_t - \omega_1 e_{t-1}$	ARIMA model
ARIMA (1, 1, 2)	1 1 2 $\hat{Y}_t = Y_{t-1} + \phi_1 (Y_{t-1} - Y_{t-2}) - \theta_1 e_{t-1} - \theta_2 e_{t-2}$	Damped-trend linear Exponential smoothing
ARIMA (0, 2, 1) OR (0,2,2)	0 2 1 $\hat{Y}_t = 2Y_{t-1} - Y_{t-2} - \theta_1 e_{t-1} - \theta_2 e_{t-2}$	Linear exponential smoothing

ARIMA is a method among several used for forecasting univariate variables, which uses information obtained from the variable itself to predict its trend. The variables are regressed on its own past values.

AR(p) is where p equals the order of autocorrelation (designates weighted moving average over past observations) z I (d), where d is the order of integration (differencing), which indicates linear trend or polynomial trend z.

MA(q) is where q equals the order of moving averages (designates weighted moving average over past errors).

ARIMA is made up of two models: AR and MA.

The Integration (I)

Time-series data is often nonstationary, and to make time-series stationary, the series needs to be differentiated. This process is known as the *integration* part (I), and the order of differencing is signified as d. Differencing eradicates signals with time, which contains trends and seasonality, so this series contains noise and an irregular component, which will be modeled only.

Table 4-5 illustrates how $d(I)$ can be articulated algebraically.

Table 4-5. Interpreting $d(I)$ Value

Integral d Value	Formula(y_t)
$d = 0$	Y_t
$d = 1$	$Y_t - Y_{t-1}$
$d = 2$	$Y_t - 2Y_{t-1} + Y_{t-2}$

ARIMA in Action

In the previous section, you learned about AR and MA and the interpretation of AR, I, and MA. Now let's implement it on a time-series dataset.

Import the required libraries and load the CSV data, which is Facebook stock data from 2014 to 2019.

- **Pmdarima** (for py + arima) is a statistical library designed to fill the void in Python's time-series analysis capabilities, which is the equivalent of R's `auto.arima`.

```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm
from pmdarima import auto_arima
from sklearn import metrics
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
warnings.filterwarnings("ignore")
df = pd.read_csv(r'\Data\FB.csv', parse_dates = True)
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

Let's take a sneak peek at the Facebook stock data.

```
df.head(10)
```

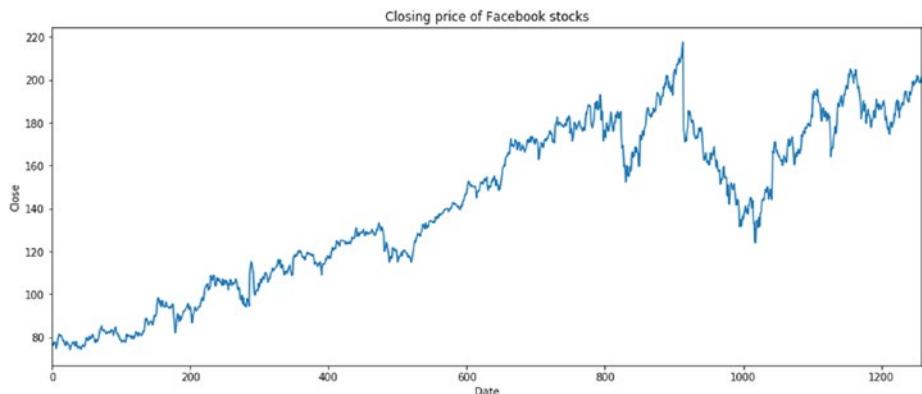
	Date	Open	High	Low	Close	Adj Close	Volume
0	2014-12-08	76.180000	77.250000	75.400002	76.519997	76.519997	25733900
1	2014-12-09	75.199997	76.930000	74.779999	76.839996	76.839996	25358600
2	2014-12-10	76.650002	77.550003	76.070000	76.180000	76.180000	32210500
3	2014-12-11	76.519997	78.519997	76.480003	77.730003	77.730003	33462100
4	2014-12-12	77.160004	78.879997	77.019997	77.830002	77.830002	28091600
5	2014-12-15	78.459999	78.580002	76.559998	76.989998	76.989998	29396500
6	2014-12-16	76.190002	77.389999	74.589996	74.690002	74.690002	31554600
7	2014-12-17	75.010002	76.410004	74.900002	76.110001	76.110001	29203900
8	2014-12-18	76.889999	78.400002	76.510002	78.400002	78.400002	34222100
9	2014-12-19	78.750000	80.000000	78.330002	79.879997	79.879997	43335000

Perform some basic exploratory data analysis using line, histogram, and [kernel density estimation](#) of the closing price of stocks. Exploratory data analysis is the process of using a graphical representation to discover and investigate patterns within data.

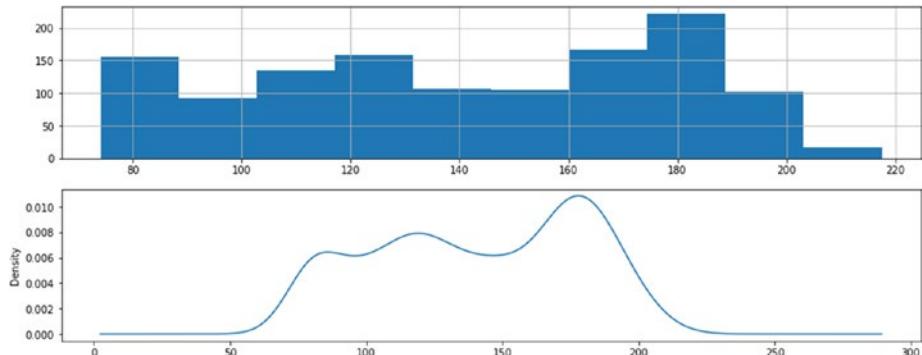
Kernel density estimation (KDE) is a nonparametric way to estimate the probability density function (PDF) of a random variable.

```
df["Close"].plot(figsize=(15, 6))
plt.xlabel("Date")
plt.ylabel("Close")
plt.title("Closing price of Facebook stocks")
plt.show()
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA



```
plt.figure(1, figsize=(15,6))
plt.subplot(211)
df["Close"].hist()
plt.subplot(212)
df["Close"].plot(kind='kde')
plt.show()
```



Define the time-series evaluation function as follows:

```
def timeseries_evaluation_metrics_func(y_true, y_pred):
    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true, y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true, y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true, y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end='\n\n')
```

Here is the ADF test function to check for stationary data:

```
def Augmented_Dickey_Fuller_Test_func(series , column_name):
    print (f'Results of Dickey-Fuller Test for column: {column_name}')
    dfoutput = adfuller(series, autolag='AIC')
    dfoutput = pd.Series(dfoutput[0:4], index=['Test Statistic',
    'p-value','No Lags Used','Number of Observations Used'])
    for key,value in dfoutput[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput)
    if dfoutput[1] <= 0.05:
        print("Conclusion:====>")
        print("Reject the null hypothesis")
        print("Data is stationary")
    else:
        print("Conclusion:====>")
        print("Fail to reject the null hypothesis")
        print("Data is non-stationary")
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

We can see that Close is nonstationary, and auto-arima handles this internally.

```
Augmented_Dickey_Fuller_Test_func(df['Close'], 'Close')
```

```
Results of Dickey-Fuller Test for column: Close
Test Statistic           -1.338096
p-value                  0.611568
No Lags Used            0.000000
Number of Observations Used 1258.000000
Critical Value (1%)      -3.435559
Critical Value (5%)      -2.863840
Critical Value (10%)     -2.567995
dtype: float64
Conclusion:=====>
Fail to reject the null hypothesis
Data is non-stationary
```

Model training will be done only for the Close column from the dataset. Make a copy of the data, and let's perform the test/train split.

The train will have all the data except the last 30 days, and the test will contain only the last 30 days to evaluate against predictions.

```
X = df[['Close']]
train, test = X[0:-30], X[-30:]
```

The pmdarima module will help us identify p, d, and q without the hassle of looking at the plot.

```
stepwise_model = auto_arima(train, start_p=1, start_q=1,
    max_p=7, max_q=7, seasonal=False,
    d=None, trace=True, error_action='ignore', suppress_warnings=True,
    stepwise=True)

stepwise_model.summary()
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

ARIMA Model Results

Dep. Variable:	D.y	No. Observations:	1228
Model:	ARIMA(1, 1, 1)	Log Likelihood	-2959.396
Method:	css-mle	S.D. of innovations	2.694
Date:	Sun, 19 Jan 2020	AIC	5926.792
Time:	12:20:00	BIC	5947.245
Sample:	1	HQIC	5934.488

	coef	std err	z	P> z	[0.025	0.975]
const	0.0892	0.056	1.583	0.114	-0.021	0.200
ar.L1.D.y	0.8630	0.072	11.942	0.000	0.721	1.005
ma.L1.D.y	-0.8998	0.062	-14.514	0.000	-1.021	-0.778

Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.1587	+0.0000j	1.1587	0.0000
MA.1	1.1113	+0.0000j	1.1113	0.0000

Auto-arima says ARIMA(1,1,1) is the optimal selection for the dataset.

Forecast both results and the confidence for the next 30 days and store it in a DataFrame, as shown here:

```

forecast,conf_int = stepwise_model.predict(n_periods=30,
return_conf_int=True)
forecast = pd.DataFrame(forecast,columns=['close_pred'])

df_conf = pd.DataFrame(conf_int,columns= ['Upper_bound',
'Lower_bound'])
df_conf["new_index"] = range(1229, 1259)
df_conf = df_conf.set_index("new_index")

timeseries_evaluation_metrics_func(test, forecast)

Evaluation metric results:-
MSE is : 73.00037730023857
MSE is : 7.727311796891414
RMSE is : 8.544025825115382
MAPE is : 3.917936080006896
R2 is : -2.9884831916539394

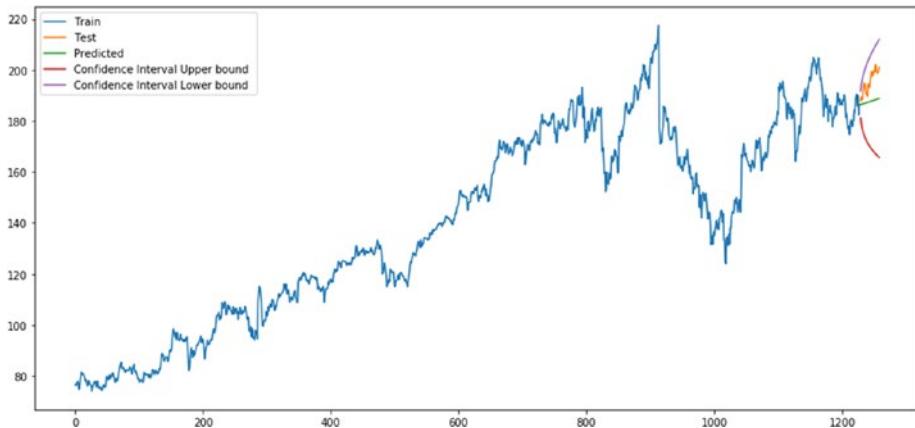
```

Rearrange the indexes for the plots to align, as shown here:

```
forecast["new_index"] = range(1229, 1259)
forecast = forecast.set_index("new_index")
```

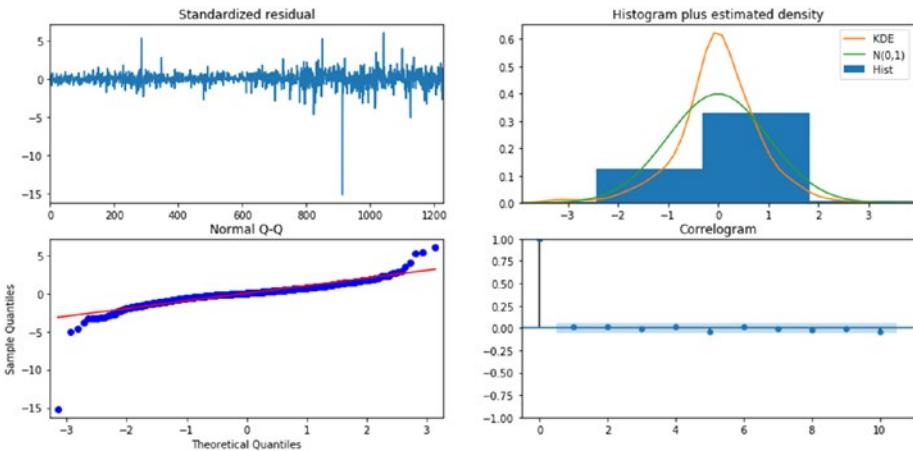
Plot the results with confidence bounds, as shown here:

```
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.rcParams["figure.figsize"] = [15,7]
plt.plot(train, label='Train ')
plt.plot(test, label='Test ')
plt.plot(forecast, label='Predicted ')
plt.plot(df_conf['Upper_bound'], label='Confidence Interval
Upper bound ')
plt.plot(df_conf['Lower_bound'], label='Confidence Interval
Lower bound ')
plt.legend(loc='best')
plt.show()
```



Here is a diagnostic plot:

```
stepwise_model.plot_diagnostics();
```



Introduction to Seasonal ARIMA

Seasonal ARIMA (SARIMA) is a technique of ARIMA, where the seasonal component can be handled in univariate time-series data. It adjoins three new hyperparameters to lay down AR(P), I(D), and MA(Q) for the seasonality component of a time series. SARIMA allows for the occurrence of seasonality in a series.

The seasonal ARIMA model combines both nonseasonal and seasonal components in a multiplicative model. The notation can be defined as follows:

$$\text{ARIMA } (\mathbf{p}, \mathbf{d}, \mathbf{q}) \times (\mathbf{P}, \mathbf{D}, \mathbf{Q})_m$$

m is the number of observations per year.

Three trend elements need to be configured. It is same as the ARIMA model, as shown here:

(p, d, q) is a nonseasonal component, as shown here:

- **p:** Trend autoregressive order
- **d:** Trend differencing order
- **q:** Trend moving average order

(P, Q, D) is a seasonal component.

There are four seasonal components that are not part of the ARIMA model that are essential to be configured.

- **P:** Seasonal autoregressive order
- **D:** Seasonal differencing order
- **Q:** Seasonal moving average order
- **m:** Timestamp for single-season order

Without differencing operations, the model could be written as described next.

The seasonal part of the model contains terms that are like nonseasonal components of the model but that involve backshifts of the seasonal period. Here's an example:

$$\text{ARIMA } (1,1,1) \times (1,1,1)_m \\ (1 - \phi_1 B)(1 - \Phi_1 B^m)(1 - B)(1 - B^m) y_t = (1 + \theta_1 B)(1 + \Theta_1 B^m) \varepsilon_t.$$

Table 4-6 illustrates the nonseasonal and seasonal components of AR and MA.

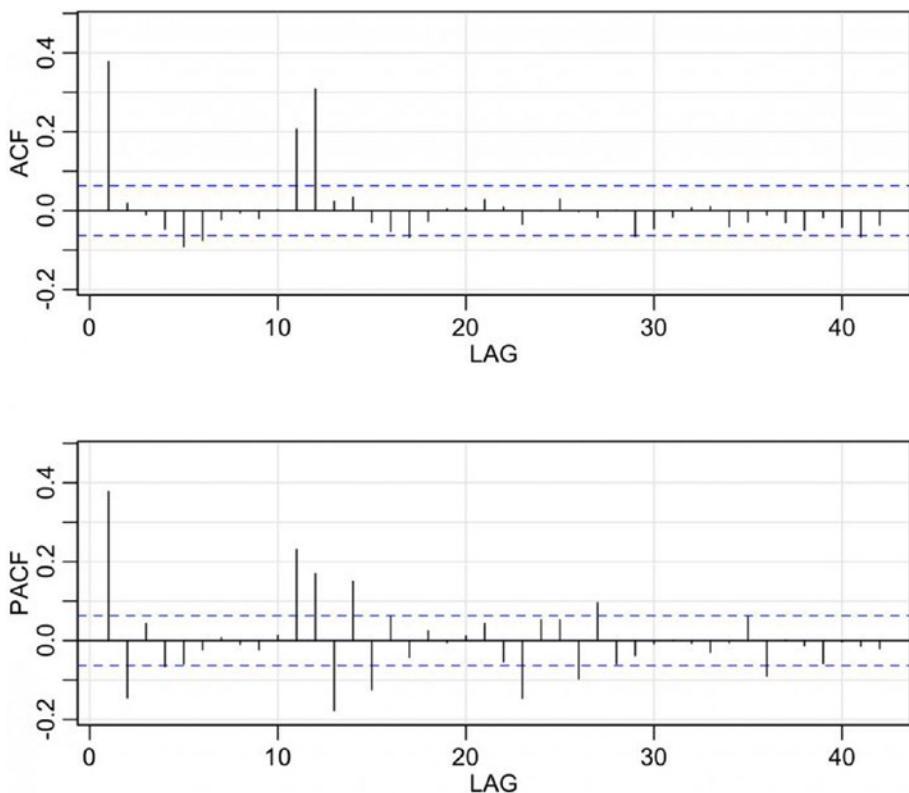
Table 4-6. Interpreting Nonseasonal and Seasonal Components

Nonseasonal Component	Seasonal Component
AR: $\Phi(B) = 1 - \Phi_1 B^p$	AR: $\Phi(B) = 1 - \Phi_1 B^{pm}$
MA: $\Theta(B) = 1 + \Theta_1 B^q$	MA: $\Theta(B) = 1 + \Theta_1 B^{qm}$

The additional term simply multiplies by the nonseasonal terms. For example, ARIMA (0,0,1) X (0,0,1)₁₂.

- Nonseasonal MA (1): $\theta(B) = 1 + \theta_1 B$
- Seasonal MA (1): $\Theta(B) = 1 + \theta_1 B^{12}$

Here is the final model: $\Theta(B) \times \theta(B) = (1 + \theta_1 B^{12})(1 + \theta_1 B)$.



SARIMA in Action

In the previous section, you learned about the high-level math behind SARIMA where you can perform the data modeling on seasonal components. Now let's implement it on a time-series dataset.

```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm
from pmdarima import auto_arima
from sklearn import metrics
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
warnings.filterwarnings("ignore")

df = pd.read_csv(r'\Data\FB.csv')
```

Let's take a sneak peek at the data.

```
df.head(10)
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2014-12-08	76.180000	77.250000	75.400002	76.519997	76.519997	25733900
1	2014-12-09	75.199997	76.930000	74.779999	76.839996	76.839996	25358600
2	2014-12-10	76.650002	77.550003	76.070000	76.180000	76.180000	32210500
3	2014-12-11	76.519997	78.519997	76.480003	77.730003	77.730003	33462100
4	2014-12-12	77.160004	78.879997	77.019997	77.830002	77.830002	28091600
5	2014-12-15	78.459999	78.580002	76.559998	76.989998	76.989998	29396500
6	2014-12-16	76.190002	77.389999	74.589996	74.690002	74.690002	31554600
7	2014-12-17	75.010002	76.410004	74.900002	76.110001	76.110001	29203900
8	2014-12-18	76.889999	78.400002	76.510002	78.400002	78.400002	34222100
9	2014-12-19	78.750000	80.000000	78.330002	79.879997	79.879997	43335000

Define the time-series evaluation function, as shown here:

```
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true, y_pred)}')
    print(f'MSE is : {metrics.mean_absolute_error(y_true, y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_true,
y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end='\n\n')
```

Here is the ADF test function to check for stationary data:

```
def Augmented_Dickey_Fuller_Test_func(series , column_name):
    print (f'Results of Dickey-Fuller Test for column: {column_
name}')
    dfoutput = adfuller(series, autolag='AIC')
    dfoutput = pd.Series(dfoutput[0:4], index=['Test Statistic',
'p-value','No Lags Used','Number of Observations Used'])
    for key,value in dfoutput[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput)
    if dfoutput[1] <= 0.05:
        print("Conclusion:====>")
        print("Reject the null hypothesis")
        print("Data is stationary")
    else:
        print("Conclusion:====>")
        print("Fail to reject the null hypothesis")
        print("Data is non-stationary")
```

We can see that Close is nonstationary, and auto-arima handles this internally, as shown here:

```
Augmented_Dickey_Fuller_Test_func(df['Close'] , 'Close')

Results of Dickey-Fuller Test for column: Close
Test Statistic              -1.338096
p-value                      0.611568
No Lags Used                  0.000000
Number of Observations Used   1258.000000
Critical Value (1%)           -3.435559
Critical Value (5%)            -2.863840
Critical Value (10%)           -2.567995
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

Modeling will be done only for the Close column from the dataset. Please make a copy of the data, and let's perform the test train split.

The train will have all the data except the last 30 days, and the test will contain only the last 30 days to evaluate against the predictions.

```
X = df[['Close']]  
train, test = X[0:-30], X[-30:]
```

Let's configure and run seasonal arima for the parameters given in the for loop and check the optimal number of periods in each season suitable for our dataset.

Here is the search space:

- **p** → 1 to 7.
- **q** → 1 to 7.
- **d** → None means find the optimal value.
- **P** → 1 to 7.
- **Q** → 1 to 7.
- **D** → None means find the optimal value.

m refers to the number of periods in each season.

- **7** → Daily
- **12** → Monthly
- **52** → Weekly
- **4** → Quarterly
- **1** → Annual (non-seasonal)

```
for m in [1, 4, 7, 12, 52]:  
    print("*100)  
    print(f' Fitting SARIMA for Seasonal value m = {str(m)}')
```

```
stepwise_model = auto_arima(train, start_p=1, start_q=1,
                            max_p=7, max_q=7, seasonal=True, start_P=1,
                            start_Q=1, max_P=7, max_D=7, max_Q=7, m=m,
                            d=None, D=None, trace=True, error_
                            action='ignore', suppress_warnings=True,
                            stepwise=True)

print(f'Model summary for m = {str(m)}')
print("-"*100)
stepwise_model.summary()

forecast ,conf_int= stepwise_model.predict
(n_periods=30,return_conf_int=True)
df_conf = pd.DataFrame(conf_int,columns= ['Upper_bound',
'Lower_bound'])
df_conf["new_index"] = range(1229, 1259)
df_conf = df_conf.set_index("new_index")
forecast = pd.DataFrame(forecast, columns=['close_pred'])
forecast["new_index"] = range(1229, 1259)
forecast = forecast.set_index("new_index")

timeseries_evaluation_metrics_func(test, forecast)

import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.rcParams["figure.figsize"] = [15, 7]
plt.plot(train, label='Train ')
plt.plot(test, label='Test ')
plt.plot(forecast, label=f'Predicted with m={str(m)} ')
plt.plot(df_conf['Upper_bound'], label='Confidence Interval
Upper bound ')
plt.plot(df_conf['Lower_bound'], label='Confidence Interval
Lower bound ')
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```

plt.legend(loc='best')
plt.show()
print("-"*100)
print(f' Diagnostic plot for Seasonal value m = {str(m)}')

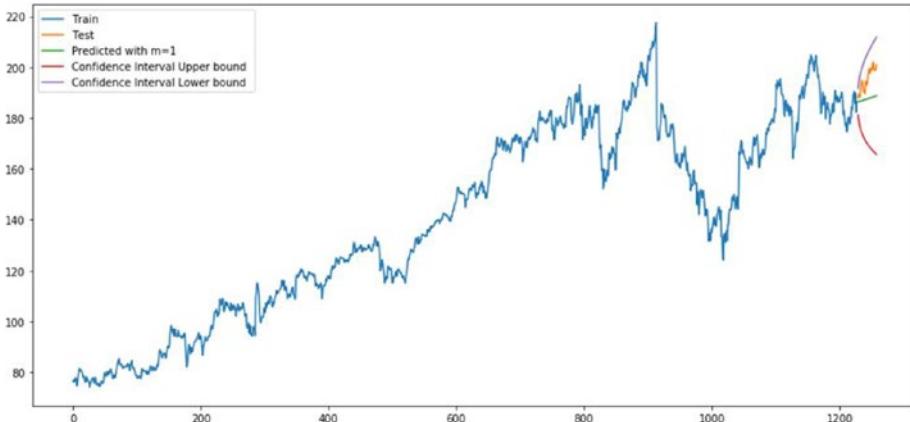
display(stepwise_model.plot_diagnostics());
print("-"*100)

```

```

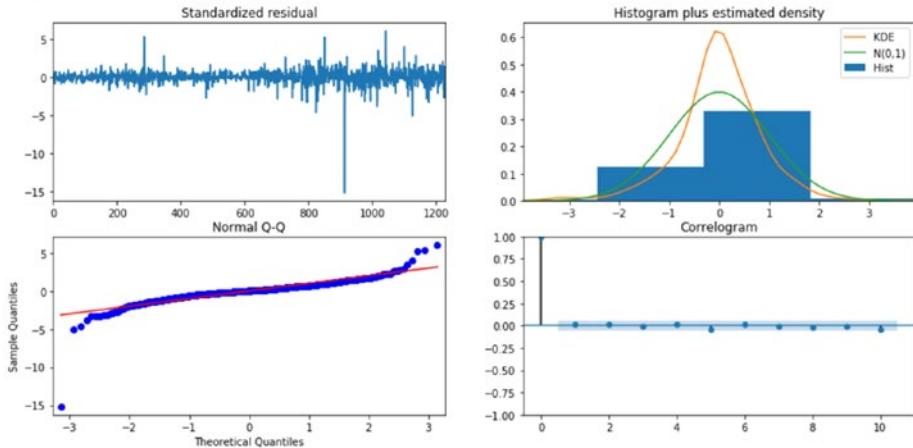
=====
Fitting SARIMA for Seasonal value m = 1
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 0, 1); AIC=5926.792, BIC=5947.245, Fit time=0.685 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 0, 0, 1); AIC=5929.302, BIC=5939.528, Fit time=0.029 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(0, 0, 0, 1); AIC=5930.825, BIC=5946.165, Fit time=0.130 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 0, 1); AIC=5930.810, BIC=5946.149, Fit time=0.171 seconds
Fit ARIMA: order=(2, 1, 1) seasonal_order=(0, 0, 0, 1); AIC=5928.521, BIC=5954.087, Fit time=0.726 seconds
Fit ARIMA: order=(1, 1, 2) seasonal_order=(0, 0, 0, 1); AIC=5930.139, BIC=5955.704, Fit time=0.634 seconds
Fit ARIMA: order=(2, 1, 2) seasonal_order=(0, 0, 0, 1); AIC=5928.288, BIC=5958.967, Fit time=1.863 seconds
Total fit time: 4.243 seconds
Model summary for m = 1
-----
Evaluation metric results:-
MSE is : 73.01424039849306
MSE is : 7.728046686414122
RMSE is : 8.544837060968048
MAPE is : 3.9183085301926943
R2 is : -2.9892406224565895

```



CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

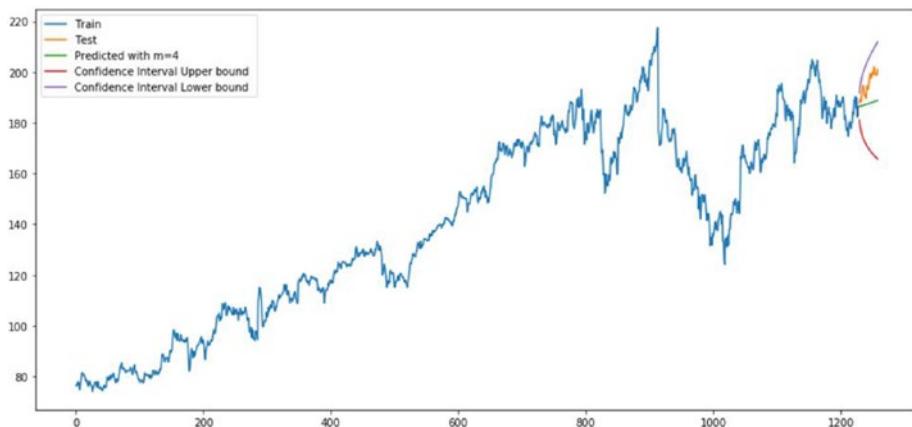
Diagnostic plot for Seasonal value m = 1



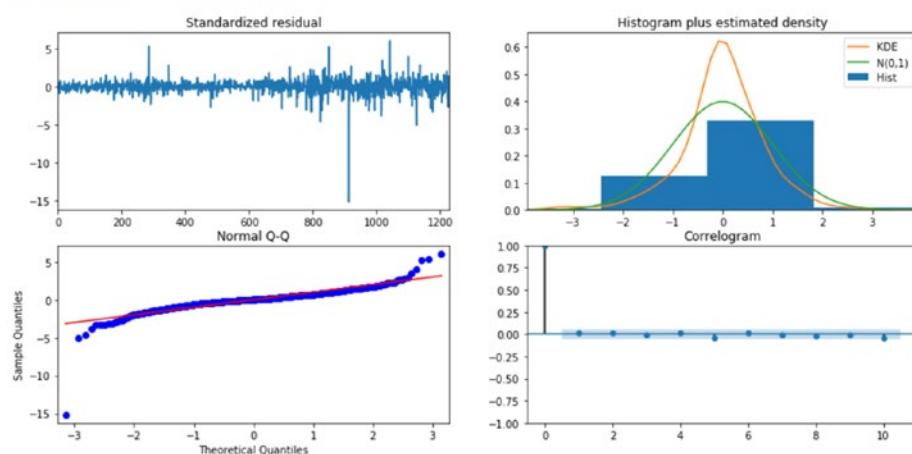
```
=====
Fitting SARIMA for Seasonal value m = 4
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 1, 4); AIC=5930.606, BIC=5961.285, Fit time=1.326 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 0, 0, 4); AIC=5929.302, BIC=5939.528, Fit time=0.037 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 0, 0, 4); AIC=5932.751, BIC=5953.203, Fit time=0.252 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 1, 4); AIC=5932.728, BIC=5953.181, Fit time=0.273 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(1, 0, 0, 4); AIC=5931.257, BIC=5946.596, Fit time=0.122 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 0, 1, 4); AIC=5931.254, BIC=5946.593, Fit time=0.148 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(1, 0, 1, 4); AIC=5932.982, BIC=5953.435, Fit time=0.734 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(0, 0, 0, 4); AIC=5930.825, BIC=5946.165, Fit time=0.123 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 0, 4); AIC=5930.810, BIC=5946.149, Fit time=0.173 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 0, 4); AIC=5926.792, BIC=5947.245, Fit time=0.693 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 0, 4); AIC=5928.543, BIC=5954.109, Fit time=0.993 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 1, 4); AIC=5928.530, BIC=5954.095, Fit time=1.296 seconds
Fit ARIMA: order=(2, 1, 1) seasonal_order=(0, 0, 0, 4); AIC=5928.521, BIC=5954.087, Fit time=0.748 seconds
Fit ARIMA: order=(1, 1, 2) seasonal_order=(0, 0, 0, 4); AIC=5930.139, BIC=5955.704, Fit time=0.632 seconds
Fit ARIMA: order=(2, 1, 2) seasonal_order=(0, 0, 0, 4); AIC=5928.288, BIC=5958.967, Fit time=1.844 seconds
Total fit time: 9.414 seconds
Model summary for m = 4
```

```
Evaluation metric results:
MSE is : 73.01424039849306
MSE is : 7.728046686414122
RMSE is : 8.544837060968048
MAPE is : 3.9183085301926943
R2 is : -2.9892406224565895
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA



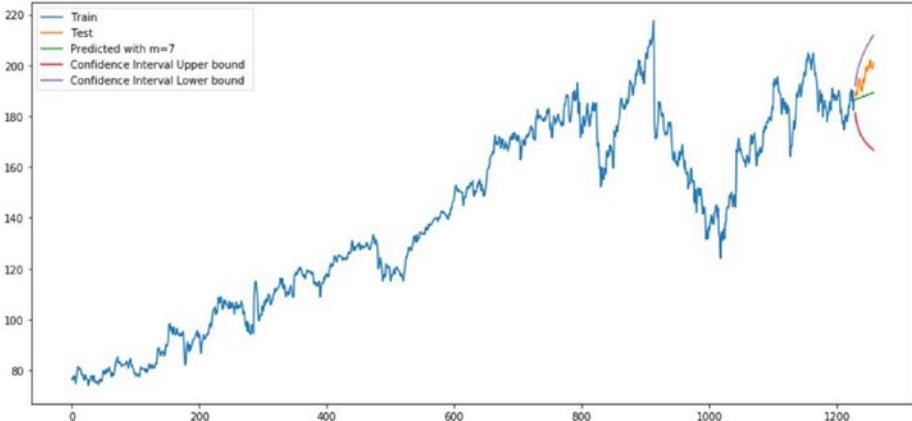
Diagnostic plot for Seasonal value $m = 4$



CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

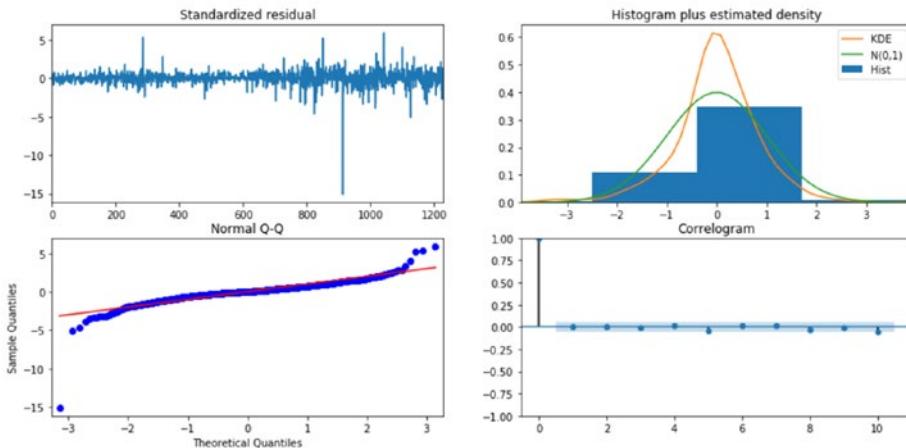
```
=====
Fitting SARIMA for Seasonal value m = 7
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 1, 7); AIC=5926.233, BIC=5956.912, Fit time=2.889 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 0, 0, 7); AIC=5929.302, BIC=5939.528, Fit time=0.040 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 0, 0, 7); AIC=5931.993, BIC=5952.445, Fit time=0.420 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 1, 7); AIC=5931.949, BIC=5952.402, Fit time=0.345 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 1, 7); AIC=5928.579, BIC=5954.145, Fit time=1.446 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(2, 0, 1, 7); AIC=5930.468, BIC=5966.252, Fit time=6.862 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 0, 7); AIC=5928.583, BIC=5954.148, Fit time=1.378 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 2, 7); AIC=5928.256, BIC=5964.048, Fit time=7.055 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 0, 7); AIC=5926.792, BIC=5947.245, Fit time=0.649 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(2, 0, 2, 7); AIC=5930.751, BIC=5971.656, Fit time=7.408 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(1, 0, 1, 7); AIC=5928.916, BIC=5954.482, Fit time=1.409 seconds
Fit ARIMA: order=(2, 1, 1) seasonal_order=(1, 0, 1, 7); AIC=5929.377, BIC=5965.169, Fit time=3.671 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 0, 1, 7); AIC=5928.936, BIC=5954.501, Fit time=2.161 seconds
Fit ARIMA: order=(1, 1, 2) seasonal_order=(1, 0, 1, 7); AIC=5933.888, BIC=5969.680, Fit time=1.566 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(1, 0, 1, 7); AIC=5927.473, BIC=5947.926, Fit time=1.064 seconds
Fit ARIMA: order=(2, 1, 2) seasonal_order=(1, 0, 1, 7); AIC=5929.971, BIC=5970.876, Fit time=5.427 seconds
Total fit time: 43.821 seconds
Model summary for m = 7
=====
```

```
Evaluation metric results:-
MSE is : 67.51208577942516
MSE is : 7.409429415593364
RMSE is : 8.216573846770025
MAPE is : 3.756215079146609
R2 is : -2.6886222965296405
```



CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

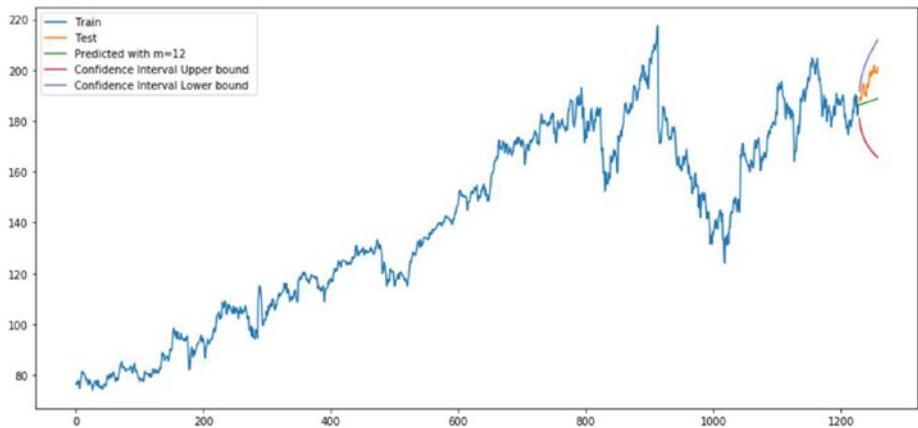
Diagnostic plot for Seasonal value m = 7



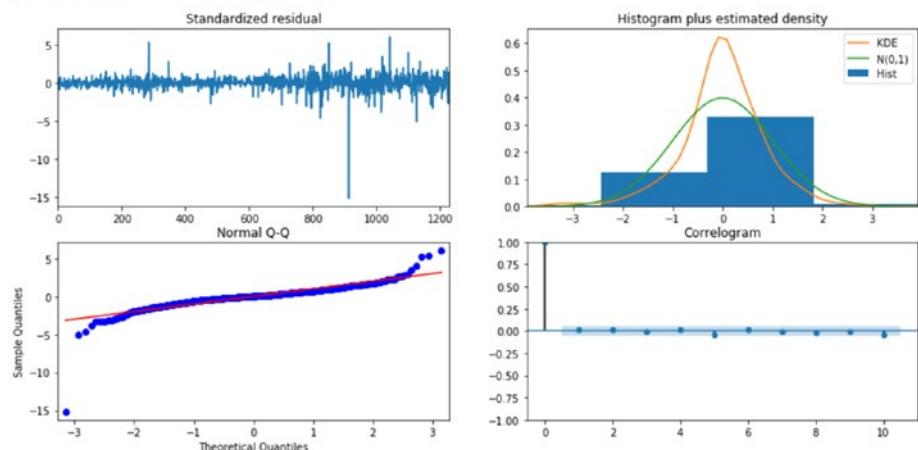
```
=====
Fitting SARIMA for Seasonal value m = 12
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 1, 12); AIC=5930.743, BIC=5961.422, Fit time=2.967 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 0, 0, 12); AIC=5929.302, BIC=5939.528, Fit time=0.039 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 0, 0, 12); AIC=5932.613, BIC=5953.066, Fit time=0.488 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 1, 12); AIC=5932.598, BIC=5953.051, Fit time=0.578 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(1, 0, 0, 12); AIC=5931.080, BIC=5946.419, Fit time=0.289 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 0, 1, 12); AIC=5931.082, BIC=5946.421, Fit time=0.282 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(1, 0, 1, 12); AIC=5933.079, BIC=5953.531, Fit time=0.505 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(0, 0, 0, 12); AIC=5930.165, BIC=5946.165, Fit time=0.132 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 0, 12); AIC=5930.810, BIC=5946.149, Fit time=0.185 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 0, 12); AIC=5926.792, BIC=5947.245, Fit time=0.650 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 0, 12); AIC=5928.744, BIC=5954.310, Fit time=2.069 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 1, 12); AIC=5928.745, BIC=5954.310, Fit time=2.754 seconds
Fit ARIMA: order=(2, 1, 1) seasonal_order=(0, 0, 0, 12); AIC=5928.521, BIC=5954.087, Fit time=0.726 seconds
Fit ARIMA: order=(1, 1, 2) seasonal_order=(0, 0, 0, 12); AIC=5930.139, BIC=5955.704, Fit time=0.614 seconds
Fit ARIMA: order=(2, 1, 2) seasonal_order=(0, 0, 0, 12); AIC=5928.288, BIC=5958.967, Fit time=1.823 seconds
Total fit time: 14.129 seconds
Model summary for m = 12
```

```
Evaluation metric results:-
MSE is : 73.01424039849306
MSE is : 7.728046686414122
RMSE is : 8.544837060968048
MAPE is : 3.9183085301926943
R2 is : -2.9892406224565895
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

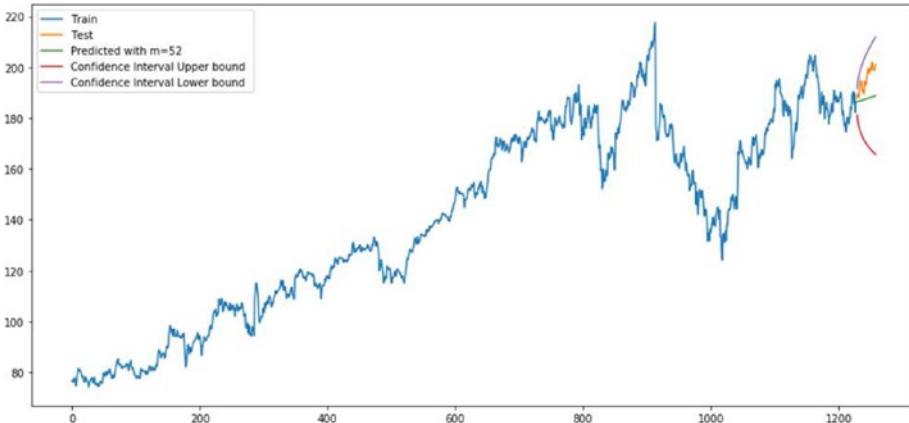


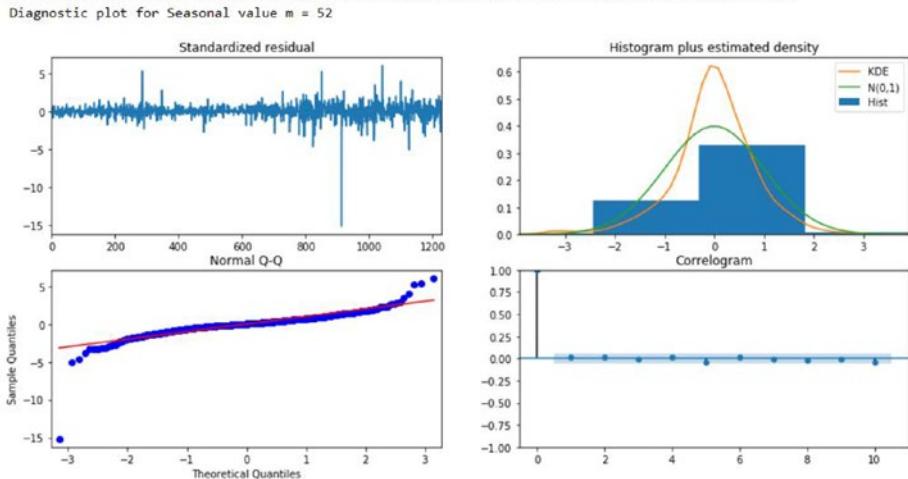
Diagnostic plot for Seasonal value $m = 12$



CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
=====
Fitting SARIMA for Seasonal value m = 52
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 1, 52); AIC=5927.302, BIC=5957.981, Fit time=45.590 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 0, 0, 52); AIC=5929.302, BIC=5939.528, Fit time=0.045 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 0, 0, 52); AIC=5932.550, BIC=5953.003, Fit time=8.176 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 1, 52); AIC=5932.556, BIC=5953.005, Fit time=8.157 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 1, 52); AIC=5928.504, BIC=5954.069, Fit time=31.940 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 1, 52); AIC=5929.290, BIC=5965.082, Fit time=273.302 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 0, 52); AIC=5928.481, BIC=5954.047, Fit time=26.452 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(1, 0, 2, 52); AIC=5929.288, BIC=5965.080, Fit time=333.811 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 0, 0, 52); AIC=5926.792, BIC=5947.245, Fit time=0.652 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 0, 0, 52); AIC=5930.810, BIC=5946.149, Fit time=0.159 seconds
Fit ARIMA: order=(2, 1, 1) seasonal_order=(0, 0, 0, 52); AIC=5928.521, BIC=5954.087, Fit time=0.709 seconds
Fit ARIMA: order=(2, 1, 0) seasonal_order=(0, 0, 0, 52); AIC=5930.825, BIC=5946.165, Fit time=0.118 seconds
Fit ARIMA: order=(1, 1, 2) seasonal_order=(0, 0, 0, 52); AIC=5930.139, BIC=5955.704, Fit time=0.621 seconds
Fit ARIMA: order=(2, 1, 2) seasonal_order=(0, 0, 0, 52); AIC=5928.288, BIC=5958.967, Fit time=1.791 seconds
Total fit time: 731.549 seconds
Model summary for m = 52
-----
Evaluation metric results:-
MSE is : 73.01424039849306
MSE is : 7.728046686414122
RMSE is : 8.544837060968048
MAPE is : 3.9183085301926943
R2 is : -2.9892406224565895
```





After checking for different m values, we can see that m does not have any influence on the results.

Introduction to SARIMAX

The SARIMAX model is a SARIMA model with external influencing variables, called SARIMAX $(p, d, q) X(P,D,Q)_m (X)$, where X is the vector of exogenous variables. The exogenous variables perhaps modeled by the multilinear regression equation are articulated as follows:

$$(1-\phi_1 B)(1-\Phi_1 B^m)(1-B)(1-B^m)y_t = (1+\theta_1 B)(1+\Theta_1 B^m)\varepsilon_t(X_{k,t}).$$

where $X_{1,t}, X_{2,t}, \dots, X_{k,t}$ are observations of k number of exogenous variables corresponding to the dependent variable.

SARIMAX in Action

In the previous section, you learned about the high-level math behind SARIMAX. Now let's implement it on a time-series dataset.

```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm
from pmdarima import auto_arima
from sklearn import metrics
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller
warnings.filterwarnings("ignore")

df = pd.read_csv(r'\Data\FB.csv')
```

Let's take a sneak peek at the data.

```
df.head(10)
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2014-12-08	76.180000	77.250000	75.400002	76.519997	76.519997	25733900
1	2014-12-09	75.199997	76.930000	74.779999	76.839996	76.839996	25358600
2	2014-12-10	76.650002	77.550003	76.070000	76.180000	76.180000	32210500
3	2014-12-11	76.519997	78.519997	76.480003	77.730003	77.730003	33462100
4	2014-12-12	77.160004	78.879997	77.019997	77.830002	77.830002	28091600
5	2014-12-15	78.459999	78.580002	76.559998	76.989998	76.989998	29396500
6	2014-12-16	76.190002	77.389999	74.589996	74.690002	74.690002	31554600
7	2014-12-17	75.010002	76.410004	74.900002	76.110001	76.110001	29203900
8	2014-12-18	76.889999	78.400002	76.510002	78.400002	78.400002	34222100
9	2014-12-19	78.750000	80.000000	78.330002	79.879997	79.879997	43335000

Define the time-series evaluation function, as shown here:

```
def timeseries_evaluation_metrics_func(y_true, y_pred):
    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',
end='\\n\\n')
```

Here is the ADF test function to check for stationary data:

```
def Augmented_Dickey_Fuller_Test_func(series , column_name):
    print (f'Results of Dickey-Fuller Test for column:
{column_name}')
    dfoutput = adfuller(series, autolag='AIC')
    dfoutput = pd.Series(dfoutput[0:4], index=['Test Statistic',
'p-value','No Lags Used','Number of Observations Used'])
    for key,value in dfoutput[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput)
    if dfoutput[1] <= 0.05:
        print("Conclusion:====>")
        print("Reject the null hypothesis")
        print("Data is stationary")
```

```
else:
    print("Conclusion:====>")
    print("Fail to reject the null hypothesis")
    print("Data is non-stationary")
```

We can see that Close is nonstationary and auto-arima handles this internally.

```
for name, column in df[['Close' , 'Open' , 'High', 'Low']].iteritems():
    Augmented_Dickey_Fuller_Test_func(df[name],name)
    print('\n')

    Results of Dickey-Fuller Test for column: Close
    Test Statistic           -1.338096
    p-value                  0.611568
    No Lags Used            0.000000
    Number of Observations Used   1258.000000
    Critical Value (1%)      -3.435559
    Critical Value (5%)      -2.863840
    Critical Value (10%)     -2.567995
    dtype: float64
    Conclusion:====>
    Fail to reject the null hypothesis
    Data is non-stationary

    Results of Dickey-Fuller Test for column: Open
    Test Statistic           -1.014123
    p-value                  0.748078
    No Lags Used            11.000000
    Number of Observations Used   1247.000000
    Critical Value (1%)      -3.435605
    Critical Value (5%)      -2.863861
    Critical Value (10%)     -2.568005
    dtype: float64
    Conclusion:====>
    Fail to reject the null hypothesis
    Data is non-stationary

    Results of Dickey-Fuller Test for column: High
    Test Statistic           -1.255326
    p-value                  0.649355
    No Lags Used            0.000000
    Number of Observations Used   1258.000000
    Critical Value (1%)      -3.435559
    Critical Value (5%)      -2.863840
    Critical Value (10%)     -2.567995
    dtype: float64
    Conclusion:====>
    Fail to reject the null hypothesis
    Data is non-stationary
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
Results of Dickey-Fuller Test for column: Low
Test Statistic           -1.006754
p-value                  0.750793
No Lags Used            10.000000
Number of Observations Used 1248.000000
Critical Value (1%)      -3.435601
Critical Value (5%)      -2.863859
Critical Value (10%)     -2.568004
dtype: float64
Conclusion:=====>
Fail to reject the null hypothesis
Data is non-stationary
```

Modeling will be done only for the Close column from the dataset, and Open will be used as exogenous variables. Please make a copy of the data, and let's perform the test train split.

The train will have all the data except the last 30 days, and the test will contain only the last 30 days to evaluate against predictions.

```
X = df[['Close']]
actualtrain, actualtest = X[0:-30], X[-30:]
exoX = df[['Open']]
exotrain, exotest = exoX[0:-30], exoX[-30:]
```

Let's configure and run seasonal arima with an exogenous variable.

```
for m in [1, 4,7,12,52]:
    print("*"*100)
    print(f' Fitting SARIMAX for Seasonal value m = {str(m)}')
    stepwise_model = auto_arima(actualtrain,exogenous =exotrain ,
    start_p=1, start_q=1,max_p=7, max_q=7, seasonal=True,
    start_P=1,start_Q=1,max_P=7,max_D=7,max_Q=7,m=m, d=None,D=None,
    trace=True,error_action='ignore',suppress_warnings=True,
    stepwise=True)

    print(f'Model summary for m = {str(m)}')
    print("-"*100)
    stepwise_model.summary()
```

```
forecast,conf_int = stepwise_model.predict(n_periods=30,
exogenous =exotest,return_conf_int=True)
df_conf = pd.DataFrame(conf_int,columns= ['Upper_bound',
'Lower_bound'])
df_conf["new_index"] = range(1229, 1259)
df_conf = df_conf.set_index("new_index")
forecast = pd.DataFrame(forecast, columns=['close_pred'])
forecast["new_index"] = range(1229, 1259)
forecast = forecast.set_index("new_index")

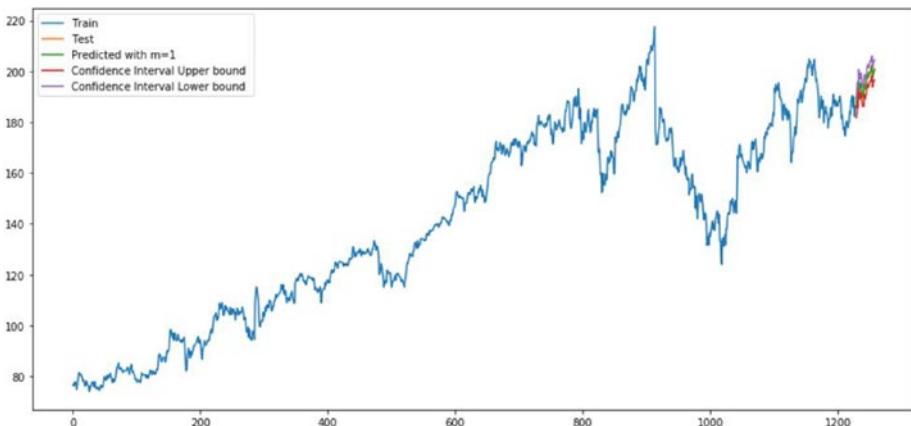
timeseries_evaluation_metrics_func(actualtest, forecast)

import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
plt.rcParams["figure.figsize"] = [15, 7]
plt.plot(actualtrain, label='Train ')
plt.plot(actualtest, label='Test ')
plt.plot(forecast, label=f'Predicted with m={str(m)} ')
plt.plot(df_conf['Upper_bound'], label='Confidence Interval
Upper bound ')
plt.plot(df_conf['Lower_bound'], label='Confidence Interval
Lower bound ')
plt.legend(loc='best')
plt.show()
print("-"*100)
print(f' Diagnostic plot for Seasonal value m = {str(m)}')

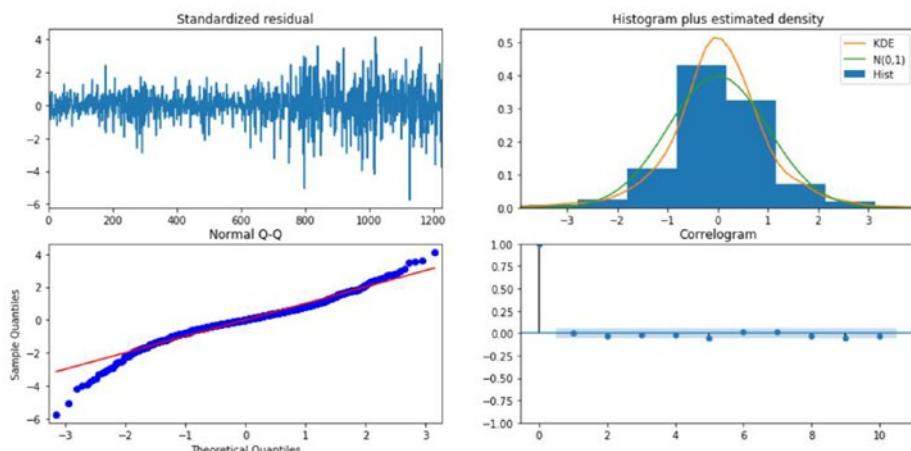
display(stepwise_model.plot_diagnostics());
print("-"*100)
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
=====
Fitting SARIMAX for Seasonal value m = 1
Fit ARIMA: order=(1, 0, 1) seasonal_order=(0, 0, 0, 1); AIC=5155.727, BIC=5181.297, Fit time=0.455 seconds
Fit ARIMA: order=(0, 0, 0) seasonal_order=(0, 0, 0, 1); AIC=5178.246, BIC=5193.587, Fit time=0.393 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(0, 0, 0, 1); AIC=5157.564, BIC=5178.020, Fit time=0.358 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 0, 0, 1); AIC=5155.602, BIC=5176.058, Fit time=0.386 seconds
Fit ARIMA: order=(0, 0, 2) seasonal_order=(0, 0, 0, 1); AIC=5156.191, BIC=5181.761, Fit time=0.560 seconds
Fit ARIMA: order=(1, 0, 2) seasonal_order=(0, 0, 0, 1); AIC=5156.189, BIC=5186.873, Fit time=1.491 seconds
Total fit time: 3.662 seconds
Model summary for m = 1
-----
Evaluation metric results:-
MSE is : 3.5285763443158857
MSE is : 1.4505269484554872
RMSE is : 1.878450516866464
MAPE is : 0.7468512728422877
R2 is : 0.807211031500715
```



Diagnostic plot for Seasonal value m = 1



CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
=====
Fitting SARIMAX for Seasonal value m = 4
Fit ARIMA: order=(1, 0, 1) seasonal_order=(1, 0, 1, 4); AIC=5158.880, BIC=5194.678, Fit time=1.822 seconds
Fit ARIMA: order=(0, 0, 0) seasonal_order=(0, 0, 0, 4); AIC=5178.246, BIC=5193.587, Fit time=0.344 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(1, 0, 0, 4); AIC=5159.298, BIC=5184.868, Fit time=0.622 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 0, 1, 4); AIC=5157.178, BIC=5182.747, Fit time=0.575 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(1, 0, 1, 4); AIC=5158.636, BIC=5189.320, Fit time=0.255 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 0, 0, 4); AIC=5176.058, BIC=5176.058, Fit time=0.372 seconds
Fit ARIMA: order=(1, 0, 1) seasonal_order=(0, 0, 0, 4); AIC=5155.727, BIC=5181.297, Fit time=0.366 seconds
Fit ARIMA: order=(0, 0, 2) seasonal_order=(0, 0, 0, 4); AIC=5156.191, BIC=5181.761, Fit time=0.551 seconds
Fit ARIMA: order=(1, 0, 2) seasonal_order=(0, 0, 0, 4); AIC=5156.189, BIC=5186.873, Fit time=1.650 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(1, 0, 0, 4); AIC=5157.322, BIC=5182.891, Fit time=0.589 seconds
Total fit time: 7.186 seconds
Model summary for m = 4
```

Evaluation metric results:-

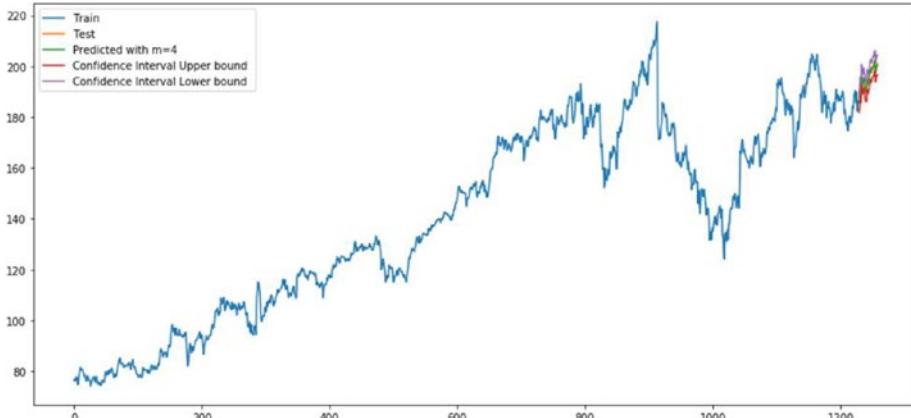
MSE is : 3.5285763443158857

MSE is : 1.4505269484554872

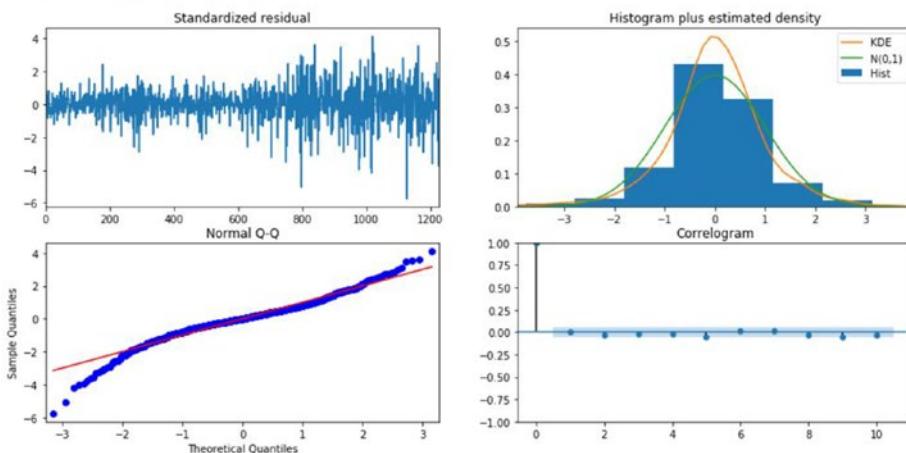
RMSE is : 1.878450516866464

MAPE is : 0.7468512728422877

R2 is : 0.807211031500715



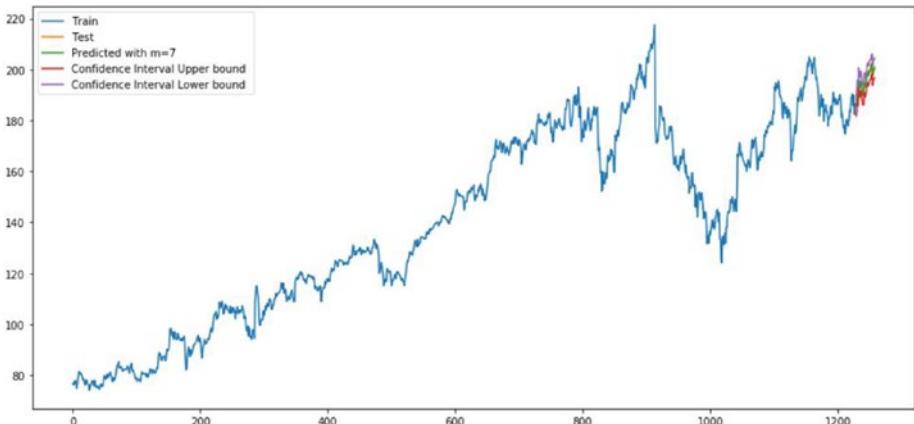
Diagnostic plot for Seasonal value m = 4



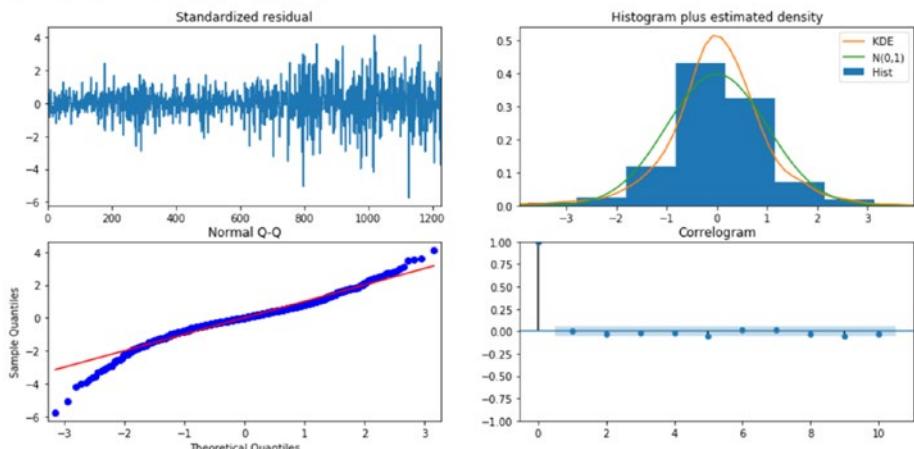
CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
=====
Fitting SARIMAX for Seasonal value m = 7
Fit ARIMA: order=(1, 0, 1) seasonal_order=(1, 0, 1, 7); AIC=5159.442, BIC=5195.240, Fit time=1.590 seconds
Fit ARIMA: order=(0, 0, 0) seasonal_order=(0, 0, 0, 7); AIC=5178.246, BIC=5193.587, Fit time=0.365 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(1, 0, 0, 7); AIC=5158.762, BIC=5184.332, Fit time=1.859 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 0, 1, 7); AIC=5157.057, BIC=5182.627, Fit time=0.674 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(1, 0, 1, 7); AIC=5159.158, BIC=5189.841, Fit time=0.771 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 0, 0, 7); AIC=5176.058, Fit time=0.412 seconds
Fit ARIMA: order=(1, 0, 1) seasonal_order=(0, 0, 0, 7); AIC=5155.727, BIC=5181.297, Fit time=0.379 seconds
Fit ARIMA: order=(0, 0, 2) seasonal_order=(0, 0, 0, 7); AIC=5156.191, BIC=5181.761, Fit time=0.558 seconds
Fit ARIMA: order=(1, 0, 2) seasonal_order=(0, 0, 0, 7); AIC=5156.189, BIC=5186.873, Fit time=1.533 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(1, 0, 0, 7); AIC=5157.097, BIC=5182.667, Fit time=1.567 seconds
Total fit time: 9.731 seconds
Model summary for m = 7
=====

Evaluation metric results:-
MSE is : 3.5285763443158857
MSE is : 1.4505269484554872
RMSE is : 1.878450516866464
MAPE is : 0.7468512728422877
R2 is : 0.807211031500715
```



Diagnostic plot for Seasonal value m = 7

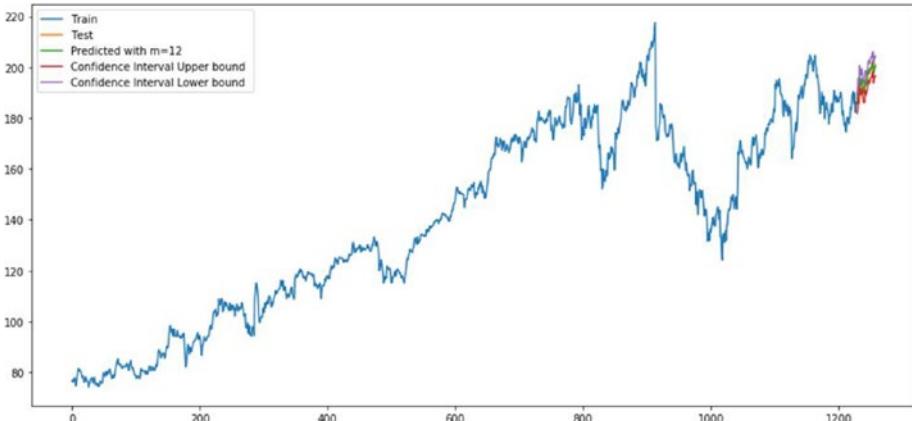


CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

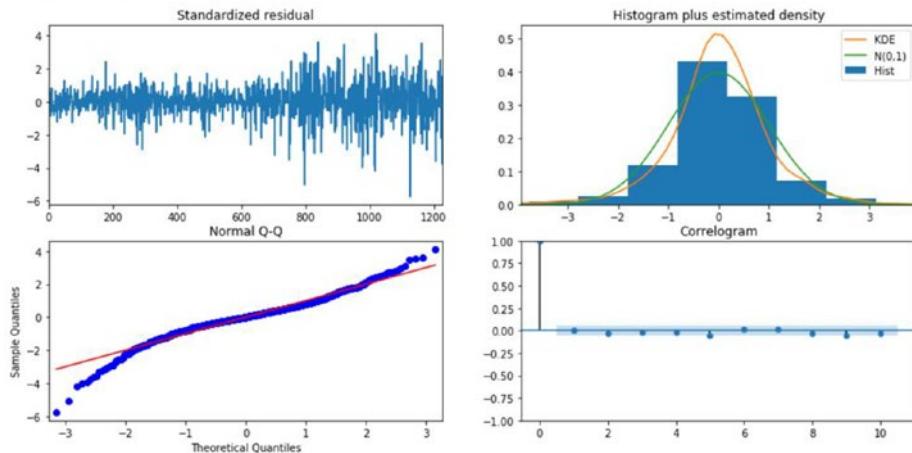
```
=====
Fitting SARIMAX for Seasonal value m = 12
Fit ARIMA: order=(1, 0, 1) seasonal_order=(1, 0, 1, 12); AIC=5158.566, BIC=5194.363, Fit time=3.733 seconds
Fit ARIMA: order=(0, 0, 0) seasonal_order=(0, 0, 0, 12); AIC=5178.246, BIC=5193.587, Fit time=0.356 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(1, 0, 0, 12); AIC=5158.340, BIC=5183.910, Fit time=1.113 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 0, 1, 12); AIC=5156.426, BIC=5181.996, Fit time=1.548 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(1, 0, 1, 12); AIC=5158.620, BIC=5189.304, Fit time=1.748 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 0, 1, 12); AIC=5155.682, BIC=5176.058, Fit time=0.380 seconds
Fit ARIMA: order=(1, 0, 1) seasonal_order=(0, 0, 0, 12); AIC=5155.727, BIC=5181.297, Fit time=0.384 seconds
Fit ARIMA: order=(0, 0, 2) seasonal_order=(0, 0, 0, 12); AIC=5156.191, BIC=5181.761, Fit time=0.543 seconds
Fit ARIMA: order=(1, 0, 2) seasonal_order=(0, 0, 0, 12); AIC=5156.189, BIC=5186.873, Fit time=1.525 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(1, 0, 0, 12); AIC=5156.262, BIC=5181.832, Fit time=2.886 seconds
Total fit time: 14.239 seconds
Model summary for m = 12
```

Evaluation metric results:-

MSE is : 3.5285763443158857
 MSE is : 1.4505269484554872
 RMSE is : 1.878450516866464
 MAPE is : 0.7468512728422877
 R2 is : 0.807211031500715



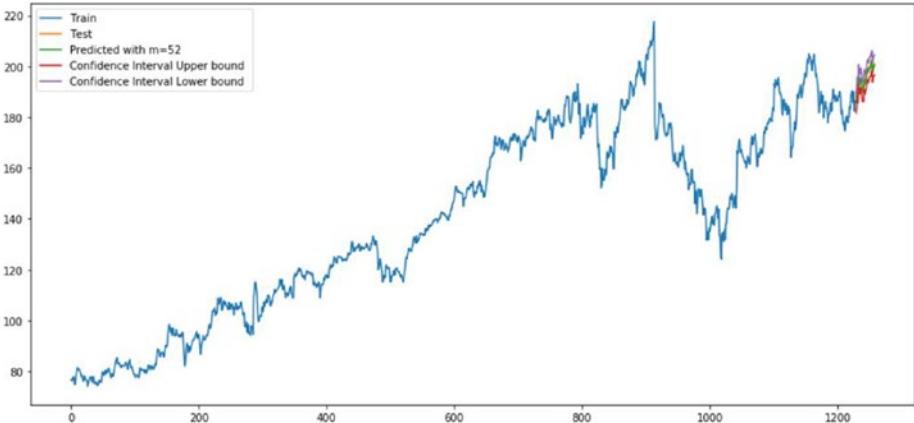
Diagnostic plot for Seasonal value m = 12



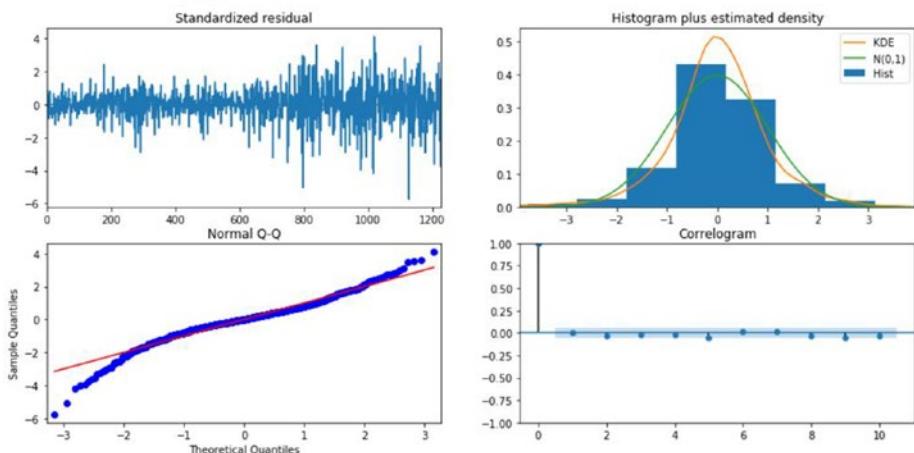
CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
=====
Fitting SARIMAX for Seasonal value m = 52
Fit ARIMA: order=(1, 0, 1) seasonal_order=(1, 0, 1, 52); AIC=5159.726, BIC=5195.524, Fit time=16.157 seconds
Fit ARIMA: order=(0, 0, 0) seasonal_order=(0, 0, 0, 52); AIC=5178.246, BIC=5193.587, Fit time=0.365 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(1, 0, 0, 52); AIC=5159.372, BIC=5184.941, Fit time=34.739 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 0, 1, 52); AIC=5157.610, BIC=5183.180, Fit time=5.931 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(1, 0, 1, 52); AIC=5159.551, BIC=5190.235, Fit time=27.898 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 0, 0, 52); AIC=5155.602, BIC=5176.058, Fit time=0.381 seconds
Fit ARIMA: order=(1, 0, 1) seasonal_order=(0, 0, 0, 52); AIC=5155.727, BIC=5181.297, Fit time=0.390 seconds
Fit ARIMA: order=(0, 0, 2) seasonal_order=(0, 0, 0, 52); AIC=5156.191, BIC=5181.761, Fit time=0.577 seconds
Fit ARIMA: order=(1, 0, 2) seasonal_order=(0, 0, 0, 52); AIC=5156.189, BIC=5186.873, Fit time=1.633 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(1, 0, 0, 52); AIC=5157.586, BIC=5183.155, Fit time=15.454 seconds
Total fit time: 103.546 seconds
Model summary for m = 52
```

Evaluation metric results:-
MSE is : 3.5285763443158857
MSE is : 1.4505269484554872
RMSE is : 1.878450516866464
MAPE is : 0.7468512728422877
R2 is : 0.807211031500715



Diagnostic plot for Seasonal value m = 52



We can see that the exogenous variable open is contributing to increased model accuracy, and we notice that m does not have any influence on prediction.

Introduction to Vector Autoregression

Vector autoregression (VAR) is a stochastic process model utilized to seize the linear relation among the multiple variables of time-series data. In other words, it is a multivariate forecasting method utilized when two or more time-series variables have a strong internal relationship with each other. VAR is a bidirectional model, while others are unidirectional models. In a unidirectional model, a predictor influences the target, but not vice versa. In a bidirectional model, variables influence each other.

The normal AR(p) model equation looks like this:

$$\hat{Y}_t = \mu + \phi_1 Y_{t-1} + \phi_2 Y_{t-2} \dots + \phi_p Y_{t-p} + \varepsilon_t$$

where μ is intercepting, and $\phi_1, \phi_2, \dots, \phi_n$ are the coefficient of the lags of Y . In the VAR model, every single variable is modeled as a linear grouping of its past values and the past values of other variables in the time series. If you have multiple time series, which is determined to each other. So, one variable per equation will be designed. For instance, imagine that we have two variables of a time series, Y_1, Y_2 . We want to forecast the value of these at time (t).

Here is the VAR (1) model with two time series (Y_1 and Y_2):

$$\begin{aligned}\hat{Y}_{1,t} &= \mu_1 + \phi_{11} Y_{1,t-1} + \phi_{12} Y_{2,t-1} + \varepsilon_{1,t} \\ \hat{Y}_{2,t} &= \mu_2 + \phi_{21} Y_{1,t-1} + \phi_{22} Y_{2,t-1} + \varepsilon_{2,t}\end{aligned}$$

where $y_{1,t-1}, y_{2,t-1}$ are the first lag of the time series Y_1 and Y_2 .

The VAR model follows the same rules for the design model as the univariate time series. It is utilizing the corresponding evaluation matrices such as AIC, BIC, FPE, and HQIC.

VAR in Action

In the previous section, you learned about the high-level math behind VAR. Now let's implement it on a time-series dataset.

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.api import VAR
import numpy as np
from statsmodels.tsa.stattools import adfuller
from sklearn import metrics
from timeit import default_timer as timer
import warnings
warnings.filterwarnings("ignore")
```

Let's take a sneak peek at the data.

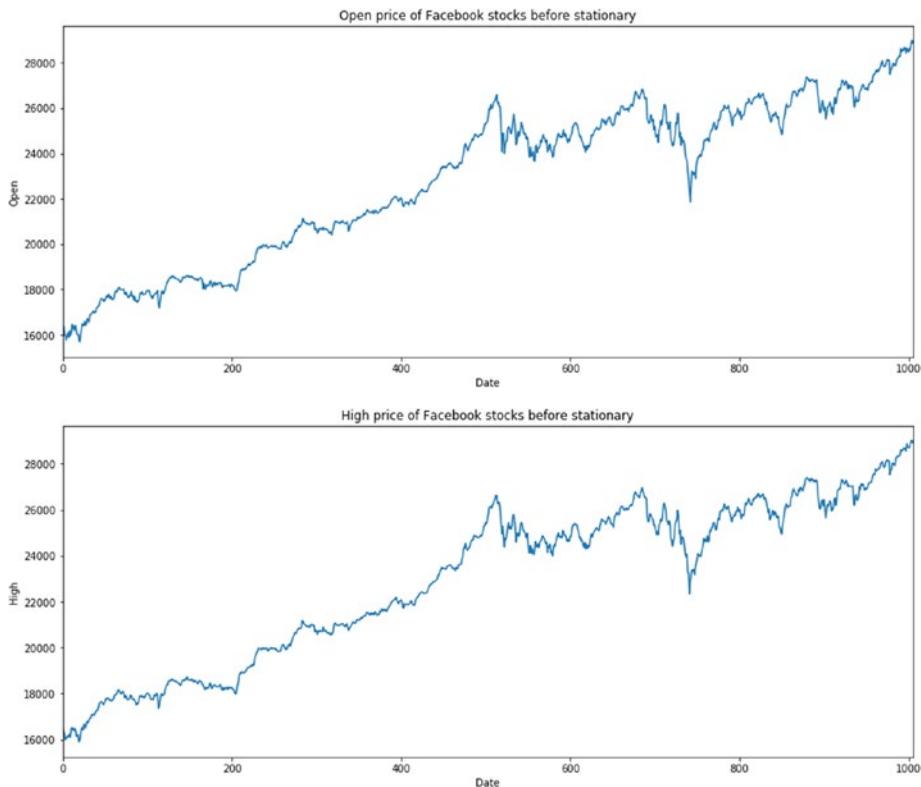
```
df = pd.read_csv(r'\Data\Dow_Jones_Industrial_Average.csv',
parse_dates= True)
df = df[(df['Date'] > '2016-01-14') & (df['Date'] <=
'2017-01-30')]
df.head(10)
```

	Date	Open	High	Low	Close	Adj Close	Volume
1	2016-01-15	16354.330078	16354.330078	15842.110352	15988.080078	15988.080078	239210000
2	2016-01-19	16009.450195	16171.959961	15900.250000	16016.019531	16016.019531	144360000
3	2016-01-20	15989.450195	15989.450195	15450.559570	15766.740234	15766.740234	191870000
4	2016-01-21	15768.870117	16038.589844	15704.660156	15882.679688	15882.679688	145140000
5	2016-01-22	15921.099609	16136.790039	15921.099609	16093.509766	16093.509766	145850000
6	2016-01-25	16086.459961	16086.459961	15880.150391	15885.219727	15885.219727	123250000
7	2016-01-26	15893.160156	16185.790039	15893.160156	16167.230469	16167.230469	118210000
8	2016-01-27	16168.740234	16235.030273	15878.299805	15944.459961	15944.459961	138350000
9	2016-01-28	15960.280273	16102.139648	15863.719727	16069.639648	16069.639648	130120000
10	2016-01-29	16090.259766	16466.300781	16090.259766	16466.300781	16466.300781	217940000

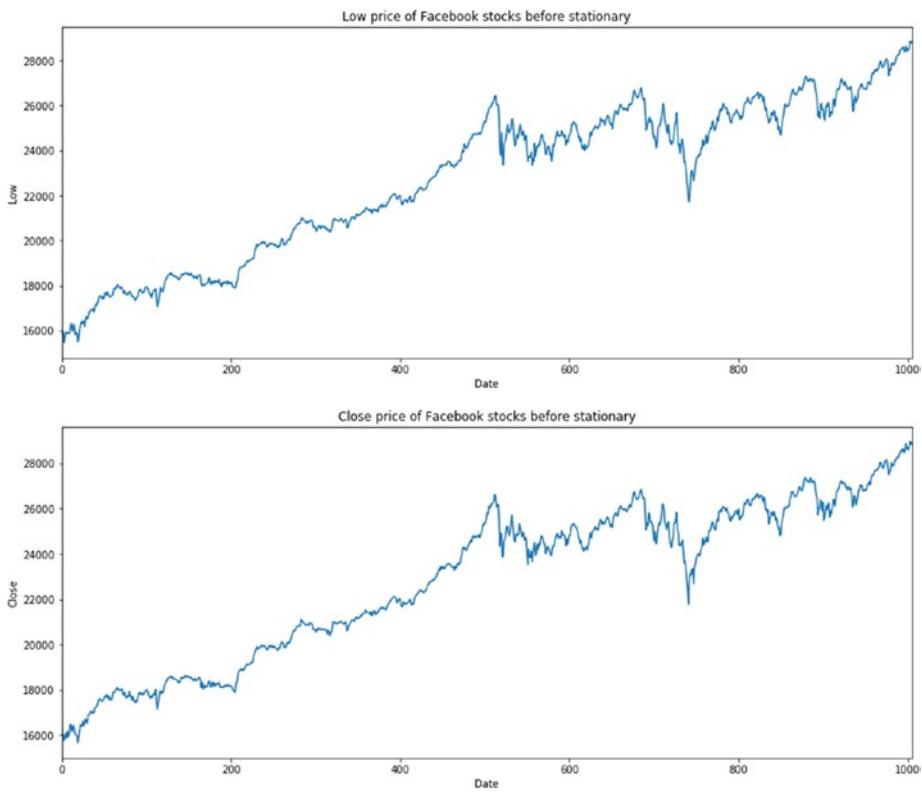
CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

Perform EDA using line, histogram, and KDE plots, as shown here:

```
for c in df[['Open', 'High', 'Low', 'Close']]:  
    df[str(c)].plot(figsize=(15, 6))  
    plt.xlabel("Date")  
    plt.ylabel(c)  
    plt.title(f"{str(c)} price of Facebook stocks before  
stationary")  
    plt.show()
```



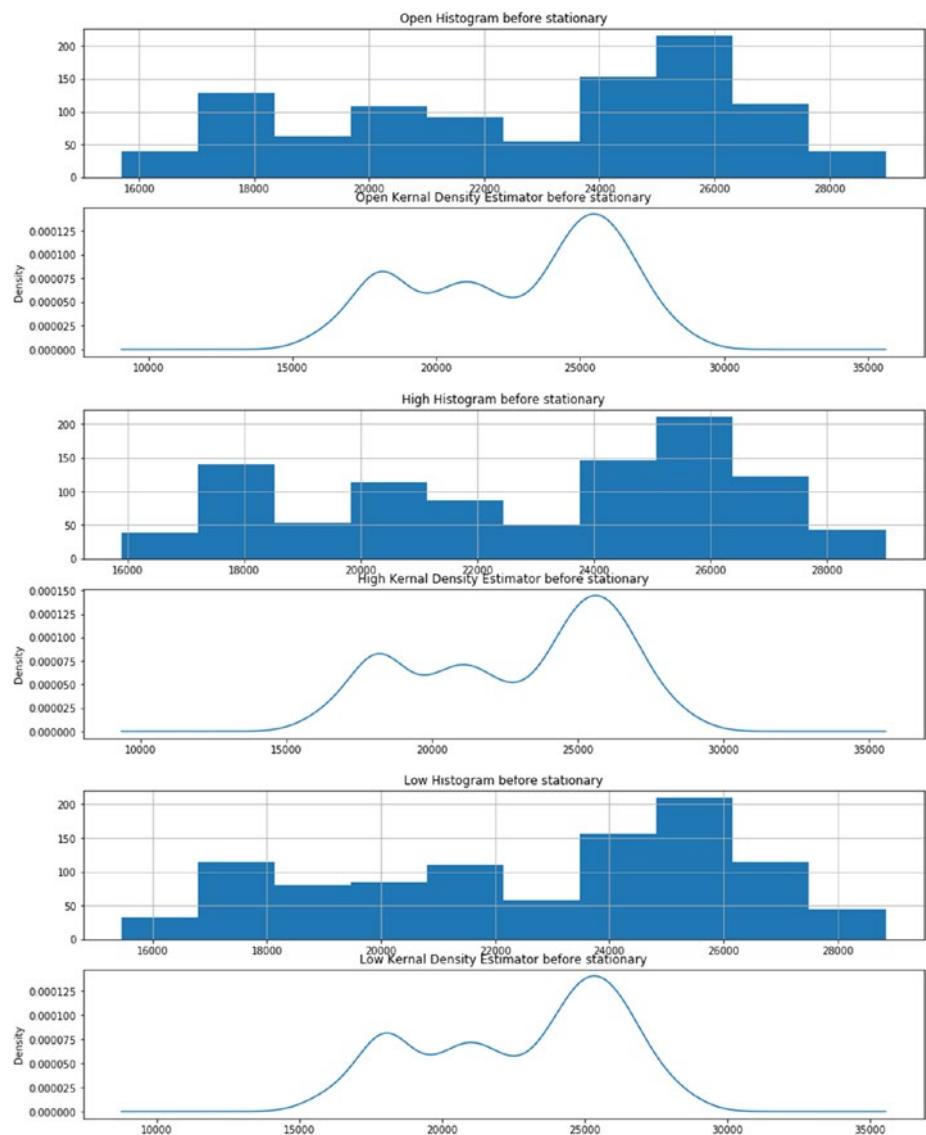
CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

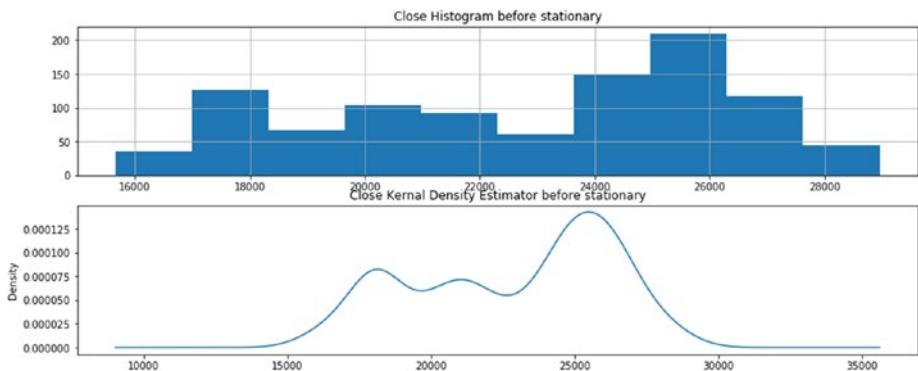


The following loop will plot a histogram and KDE for all the columns before making it stationary:

```
for c in df[['Open', 'High', 'Low', 'Close']]:  
    plt.figure(1, figsize=(15,6))  
    plt.subplot(211)  
    plt.title(f"{str(c)} Histogram before stationary")  
    df[str(c)].hist()  
    plt.subplot(212)  
    df[str(c)].plot(kind='kde')  
    plt.title(f"{str(c)} Kernel Density Estimator before  
stationary")  
    plt.show()
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA





Define a time-series evaluation function, as shown here:

```
def timeseries_evaluation_metrics_func(y_true, y_pred):
    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    #print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
    y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
    y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_
    true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
    y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=
    '\n\n')
return
```

Here is the ADF test function to check for stationary data:

```
def Augmented_Dickey_Fuller_Test_func(series , column_name):
    print (f'Results of Dickey-Fuller Test for column:
{column_name}')
    dftest = adfuller(series, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test
Statistic','p-value',
'No Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput)
    if dftest[1] <= 0.05:
        print("Conclusion:====>")
        print("Reject the null hypothesis")
        print("Data is stationary")
    else:
        print("Conclusion:=====>")
        print("Fail to reject the null hypothesis")
        print("Data is non-stationary")
```

Here is how to check whether the variables are stationary:

```
for name, column in df[['Open', 'High', 'Low',
'Close']].iteritems():
    Augmented_Dickey_Fuller_Test_func(df[name],name)
    print('\n')
        Results of Dickey-Fuller Test for column: Open
        Test Statistic              -1.097089
        p-value                      0.716231
        No Lags Used                22.000000
        Number of Observations Used 983.000000
        Critical Value (1%)         -3.437020
        Critical Value (5%)          -2.864485
        Critical Value (10%)         -2.568338
        dtype: float64
        Conclusion:=====
        Fail to reject the null hypothesis
        Data is non-stationary
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
Results of Dickey-Fuller Test for column: High
Test Statistic           -1.005190
p-value                  0.751367
No Lags Used            1.000000
Number of Observations Used   1004.000000
Critical Value (1%)      -3.436880
Critical Value (5%)      -2.864423
Critical Value (10%)     -2.568305
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary

Results of Dickey-Fuller Test for column: Low
Test Statistic           -1.147396
p-value                  0.695806
No Lags Used            21.000000
Number of Observations Used   984.000000
Critical Value (1%)      -3.437013
Critical Value (5%)      -2.864482
Critical Value (10%)     -2.568336
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary

Results of Dickey-Fuller Test for column: Close
Test Statistic           -1.015406
p-value                  0.747604
No Lags Used            0.000000
Number of Observations Used   1005.000000
Critical Value (1%)      -3.436873
Critical Value (5%)      -2.864420
Critical Value (10%)     -2.568304
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary
```

Make a copy of the data, and let's perform the test train split.

The train will have all the data except the last 30 days, and the test will contain only the last 30 days to evaluate against the predictions.

```
X = df[['Open', 'High', 'Low', 'Close']]
train, test = X[0:-30], X[-30:]
```

Make the data stationary by using Pandas differencing, as shown here:

```
train_diff = train.diff()
train_diff.dropna(inplace = True)
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

Check whether the variables are stationary after first differencing, as shown here:

```
for name, column in train_diff[['Open', 'High', 'Low',
'Close' ]].iteritems():
    Augmented_Dickey_Fuller_Test_func(train_diff[name],name)
    print('\n')
```

```
Results of Dickey-Fuller Test for column: Open
Test Statistic           -6.858594e+00
p-value                  1.624260e-09
No Lags Used             2.200000e+01
Number of Observations Used 9.520000e+02
Critical Value (1%)      -3.437238e+00
Critical Value (5%)       -2.864581e+00
Critical Value (10%)      -2.568389e+00
dtype: float64
Conclusion:====>
Reject the null hypothesis
Data is stationary

Results of Dickey-Fuller Test for column: High
Test Statistic           -29.480748
p-value                  0.000000
No Lags Used             0.000000
Number of Observations Used 974.000000
Critical Value (1%)      -3.437082
Critical Value (5%)       -2.864512
Critical Value (10%)      -2.568352
dtype: float64
Conclusion:====>
Reject the null hypothesis
Data is stationary

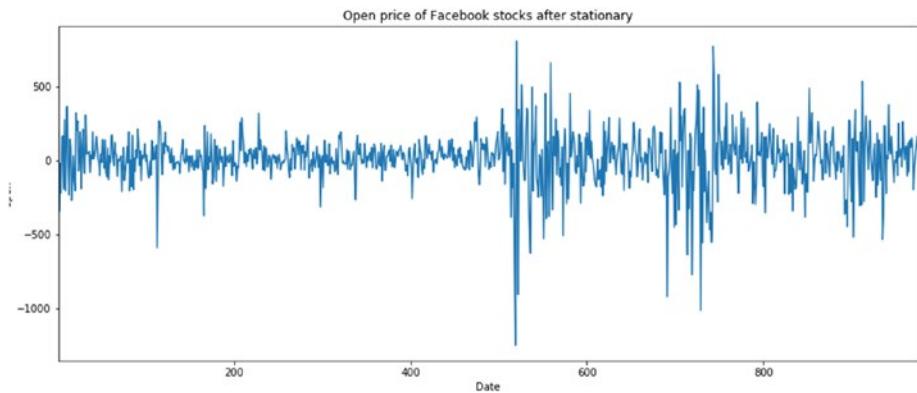
Results of Dickey-Fuller Test for column: Low
Test Statistic           -7.277476e+00
p-value                  1.530975e-10
No Lags Used             2.000000e+01
Number of Observations Used 9.540000e+02
Critical Value (1%)      -3.437223e+00
Critical Value (5%)       -2.864574e+00
Critical Value (10%)      -2.568386e+00
dtype: float64
Conclusion:====>
Reject the null hypothesis
Data is stationary
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

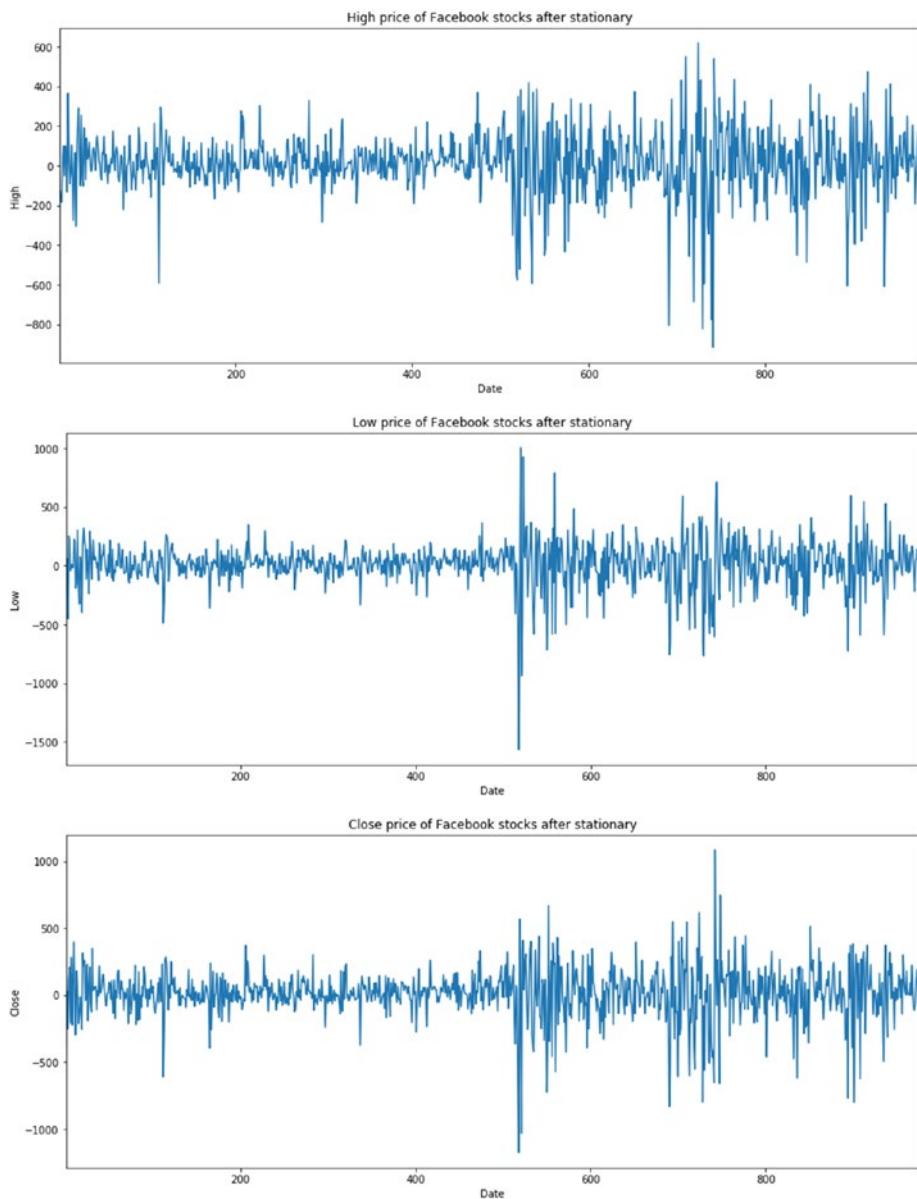
```
Results of Dickey-Fuller Test for column: Close
Test Statistic           -32.020937
p-value                  0.000000
No Lags Used             0.000000
Number of Observations Used 974.000000
Critical Value (1%)      -3.437082
Critical Value (5%)      -2.864512
Critical Value (10%)     -2.568352
dtype: float64
Conclusion:=====>
Reject the null hypothesis
Data is stationary
```

Create the plots after making the data stationary, as shown here:

```
for c in train_diff[['Open', 'High', 'Low', 'Close']]:
    train_diff[str(c)].plot(figsize=(15, 6))
    plt.xlabel("Date")
    plt.ylabel(c)
    plt.title(f"{str(c)} price of Facebook stocks after stationary")
    plt.show()
```

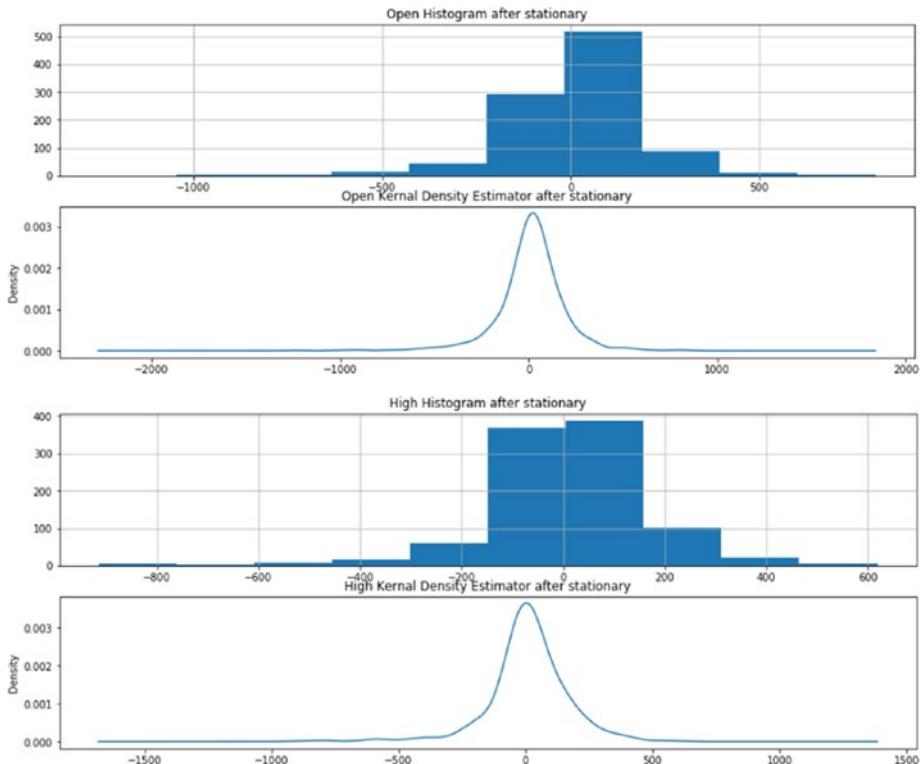


CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

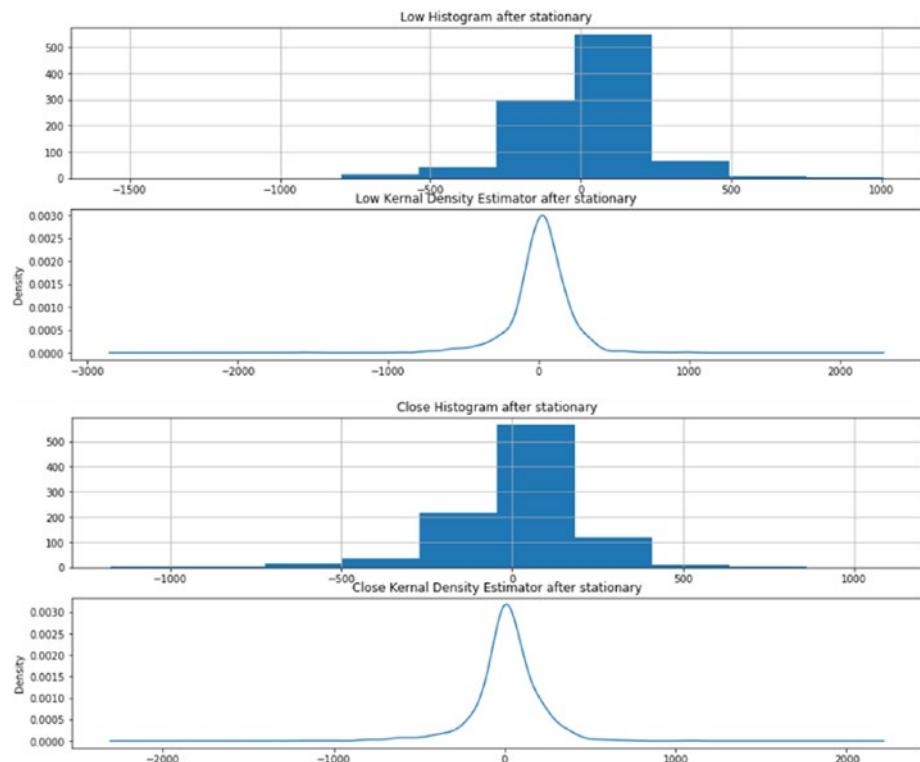


The following loop will plot a histogram and KDE for all the columns after making it stationary:

```
for c in train_diff[['Open', 'High', 'Low', 'Close']]:  
    plt.figure(1, figsize=(15,6))  
    plt.subplot(211)  
    plt.title(f"{str(c)} Histogram after stationary")  
    train_diff[str(c)].hist()  
    plt.subplot(212)  
    train_diff[str(c)].plot(kind='kde')  
    plt.title(f"{str(c)} Kernel Density Estimator after  
stationary")  
    plt.show()
```



CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA



A *cointegration test* is the co-movement among underlying variables over the long run. This long-run estimation feature distinguishes it from correlation. Two or more variables are cointegrated if and only if they share common trends.

Correlation is simply a measure of the degree of mutual association between two or more variables.

We can see that the test says that there is the presence of a long-run relationship between features.

```
from statsmodels.tsa.vector_ar.vecm import coint_johansen

def cointegration_test(df):
    res = coint_johansen(df,-1,5)
    d = {'0.90':0, '0.95':1, '0.99':2}
```

```

traces = res.lrl
cvts = res.cvt[:, d[str(1-0.05)]]
def adjust(val, length= 6):
    return str(val).ljust(length)
print('Column Name > Test Stat > C(95%) => Signif
\n', '--'*20)
for col, trace, cvt in zip(df.columns, traces, cvts):
    print(adjust(col), '> ', adjust(round(trace,2), 9),
          ">", adjust(cvt, 8), '=> ', trace > cvt)

cointegration_test(train_diff[['Open', 'High', 'Low', 'Close']])

```

Column Name	> Test Stat	> C(95%)	=>	Signif
Open	> 311.57	> 40.1749	=>	True
High	> 201.62	> 24.2761	=>	True
Low	> 102.52	> 12.3212	=>	True
Close	> 32.21	> 4.1296	=>	True

Fit the VAR model for the AR term between 1 to 9 and choose the best AR component, as shown here:

```

for i in [1,2,3,4,5,6,7,8,9]:
    model = VAR(train_diff)
    results = model.fit(i)
    print(f'Order : {i}, AIC: {results.aic}, BIC: { results.bic}')

Order : 1, AIC: 32.33907657947465, BIC: 32.63803999764189
Order : 2, AIC: 31.801631080906866, BIC: 32.34143017316867
Order : 3, AIC: 31.548769661837937, BIC: 32.33090112107316
Order : 4, AIC: 31.529049396699723, BIC: 32.555025613390185
Order : 5, AIC: 31.5360964823363, BIC: 32.807445774101225
Order : 6, AIC: 31.572569230449268, BIC: 33.090836075873455
Order : 7, AIC: 31.595014719623297, BIC: 33.3617599964768
Order : 8, AIC: 31.63403809689652, BIC: 33.650839324846004
Order : 9, AIC: 31.660078718842303, BIC: 33.92853030675716

```

To make data stationary, we used Pandas differencing after forecasting the results. We need to inverse the result to the original scale. As we don't have a Pandas function for this, let's define a custom function to inverse Pandas differencing.

```
def inverse_diff(actual_df, pred_df):
    df_res = pred_df.copy()
    columns = actual_df.columns
    for col in columns:
        df_res[str(col) + '_1st_inv_diff'] = actual_
            df[col].iloc[-1] + df_res[str(col)].cumsum()
    return df_res
```

Autoregressive AR(4) appears to be providing the least Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC), so let's fit and forecast.

AIC and BIC are two ways of scoring a model based on its log likelihood and complexity.

```
results = model.fit(4)
display(results.summary())
z = results.forecast(y=train_diff[['Open', 'High', 'Low',
'Close']].values, steps=30)
df_pred = pd.DataFrame(z, columns=['Open', 'High', 'Low', 'Close'])
```

Arrange an index for aligning plots, as shown here:

```
df_pred["new_index"] = range(233, 263)
df_pred = df_pred.set_index("new_index")
```

Let's inverse the differenced prediction, as shown here:

```
res = inverse_diff(df[['Open', 'High', 'Low', 'Close']], df_pred)
```

Evaluate the results individually, as shown here:

```
for i in ['Open', 'High', 'Low', 'Close']:
    print(f'Evaluation metric for {i}')
    timeseries_evaluation_metrics_func(test[str(i)], res[str(i) + '_1st_inv_diff'])
```

```
Evaluation metric for Open
MSE is : 94819.35427817622
MAE is : 268.1470501683449
RMSE is : 307.9275146494321
MAPE is : 1.348305938661622
R2 is : -14.51090369484316
```

```
Evaluation metric for High
MSE is : 76190.96751748091
MAE is : 231.2878158977739
RMSE is : 276.0271137360982
MAPE is : 1.1603633691332245
R2 is : -12.689810754655147
```

```
Evaluation metric for Low
MSE is : 76179.78178898557
MAE is : 227.28555251742327
RMSE is : 276.0068509819739
MAPE is : 1.147113713512027
R2 is : -7.920718980777858
```

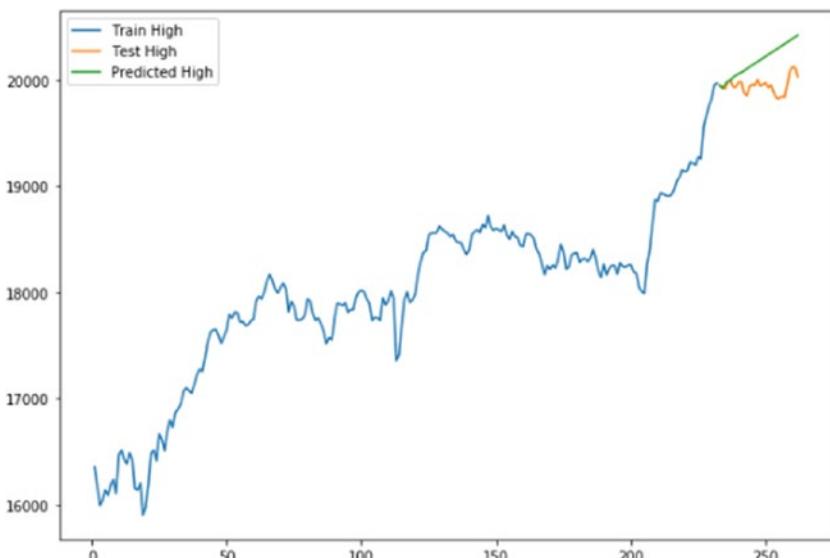
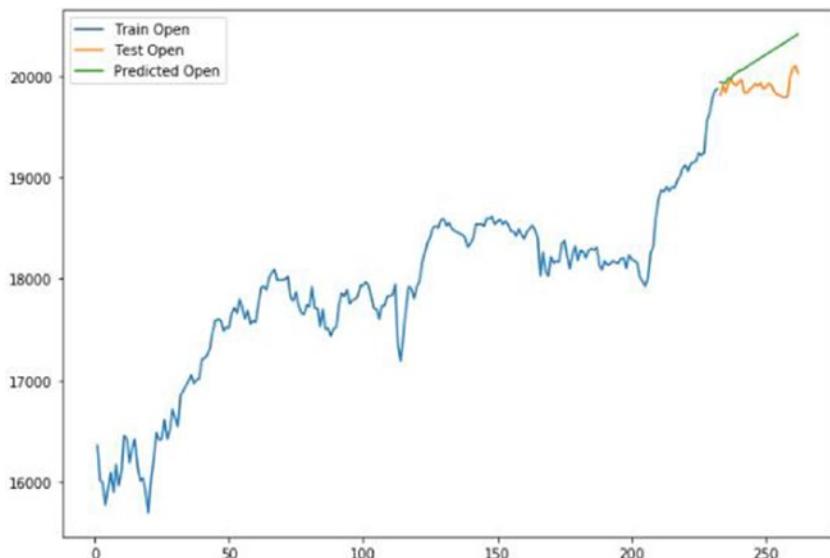
```
Evaluation metric for Close
MSE is : 118304.9345798822
MAE is : 308.06846504563583
RMSE is : 343.95484380930327
MAPE is : 1.5492399097280787
R2 is : -14.616312840893466
```

Plot the results, as shown here:

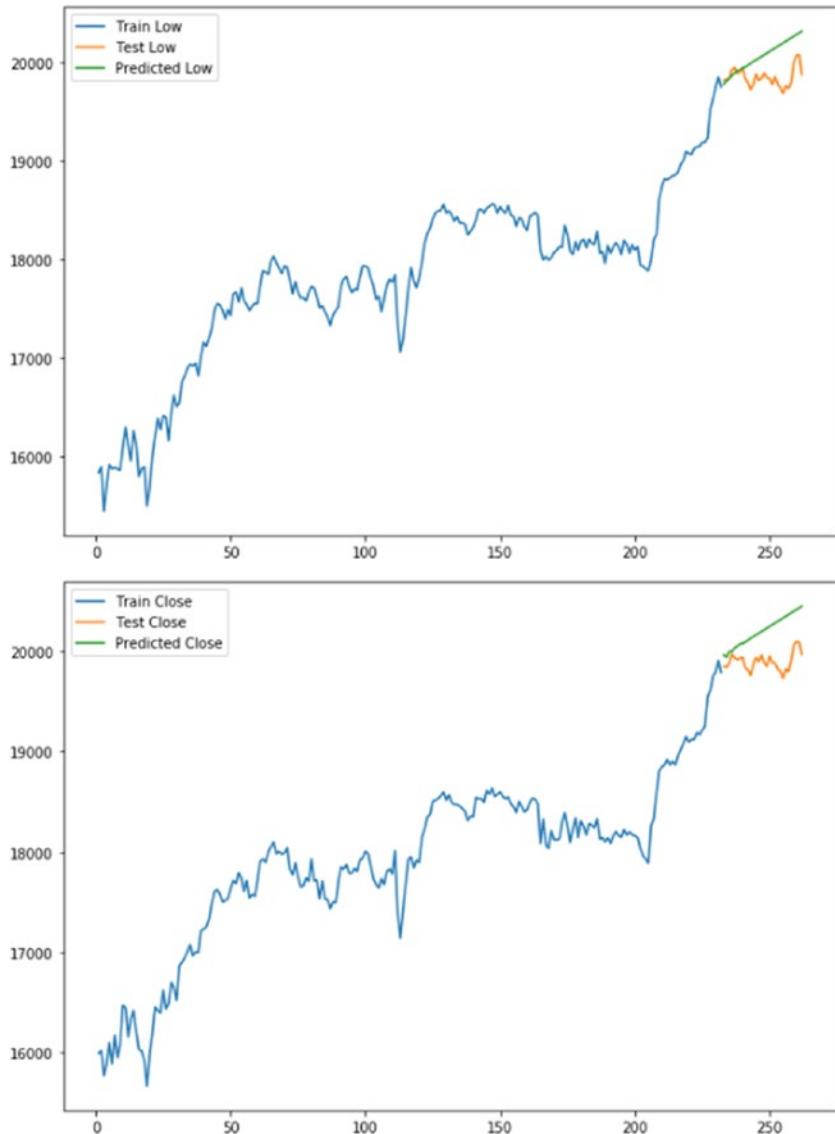
```
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
for i in ['Open', 'High', 'Low', 'Close']:
    plt.rcParams["figure.figsize"] = [10,7]
    plt.plot( train[str(i)], label='Train '+str(i))
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
plt.plot(test[str(i)], label='Test '+str(i))
plt.plot(res[str(i) + '_1st_inv_diff'], label='Predicted ' +
str(i))
plt.legend(loc='best')
plt.show()
```



CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA



Introduction to VARMA

The VARMA model is another extension of the ARMA model for a multivariate time-series model that contains a vector autoregressive (VAR) component, as well as the vector moving average (VMA). VARMA is an inductive version of ARMA for multiple parallel time series. The ARMA notation for the model comprises the individually ordered AR(p), and the MA(q) models are the parameters to a VARMA model. For instance, VARMA(p, q) is an example. A VARMA model will be able to develop VAR or VMA models. The method is used for multivariate time-series data deprived of trend and seasonal components.

Here are the VAR (1) and VMA(1) models with two time-series datasets (Y_1 and Y_2):

$$\begin{aligned}\hat{Y}_{1,t} &= \mu_1 + \phi_{11}Y_{1,t-1} + d_{1,0}u_{1,t} + d_{11}u_{1,t-1} \\ \hat{Y}_{2,t} &= \mu_2 + \phi_{21}Y_{2,t-1} + d_{2,0}u_{1,t} + d_{22}u_{2,t-1}\end{aligned}$$

where $y_{1,t-1}, y_{2,t-1}$ are the first lag of time series Y_1 and Y_2 .

The VARMA model follows the same rules for the design model similar to the univariate time series. It is utilizing the corresponding evaluation matrices such as AIC, BIC, FPE, and HQIC.

VARMA in Action

In the previous section, you learned about the high-level math behind VARMA. Now let's implement it on a time-series dataset.

```
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.statespace.varmax import VARMAX
import numpy as np
from statsmodels.tsa.stattools import adfuller
from sklearn import metrics
```

```

from timeit import default_timer as timer
import warnings
warnings.filterwarnings("ignore")

df = pd.read_csv(r'Data\{Dow_Jones_Industrial_Average.csv',
parse_dates= True)
df = df[(df['Date'] > '2016-01-14') & (df['Date'] <=
'2017-01-30')]

```

Let's take a sneak peek at the data.

```
df.head(10)
```

	Date	Open	High	Low	Close	Adj Close	Volume
1	2016-01-15	16354.330078	16354.330078	15842.110352	15988.080078	15988.080078	239210000
2	2016-01-19	16009.450195	16171.959961	15900.250000	16016.019531	16016.019531	144360000
3	2016-01-20	15989.450195	15989.450195	15450.559570	15766.740234	15766.740234	191870000
4	2016-01-21	15768.870117	16038.589844	15704.660156	15882.679688	15882.679688	145140000
5	2016-01-22	15921.099609	16136.790039	15921.099609	16093.509766	16093.509766	145850000
6	2016-01-25	16086.459961	16086.459961	15880.150391	15885.219727	15885.219727	123250000
7	2016-01-26	15893.160156	16185.790039	15893.160156	16167.230469	16167.230469	118210000
8	2016-01-27	16168.740234	16235.030273	15878.299805	15944.459961	15944.459961	138350000
9	2016-01-28	15960.280273	16102.139648	15863.719727	16069.639648	16069.639648	130120000
10	2016-01-29	16090.259766	16466.300781	16090.259766	16466.300781	16466.300781	217940000

Define a time-series evaluation function, as shown here:

```

def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')

```

```
print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_true, y_pred))}')
print(f'MAPE is : {mean_absolute_percentage_error(y_true, y_pred)}')
print(f'R2 is : {metrics.r2_score(y_true, y_pred)}', end='\\n\\n')
```

Here is the ADF test function to check for stationary data, as shown here:

```
def Augmented_Dickey_Fuller_Test_func(series , column_name):
    print (f'Results of Dickey-Fuller Test for column:
{column_name}')
    dftest = adfuller(series, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic',
    'p-value','No Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput)
    if dftest[1] <= 0.05:
        print("Conclusion:====>")
        print("Reject the null hypothesis")
        print("Data is stationary")
    else:
        print("Conclusion:=====>")
        print("Fail to reject the null hypothesis")
        print("Data is non-stationary")
```

Check whether the variables are stationary, as shown here:

```
for name, column in df[['Open', 'High', 'Low', 'Close']].iteritems():
    Augmented_Dickey_Fuller_Test_func(df[name],name)
    print('\\n')
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
Results of Dickey-Fuller Test for column: Open
Test Statistic           -0.776223
p-value                  0.826007
No Lags Used            0.000000
Number of Observations Used   261.000000
Critical Value (1%)        -3.455656
Critical Value (5%)         -2.872678
Critical Value (10%)        -2.572705
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary

Results of Dickey-Fuller Test for column: High
Test Statistic           -1.240162
p-value                  0.656085
No Lags Used            2.000000
Number of Observations Used   259.000000
Critical Value (1%)        -3.455853
Critical Value (5%)         -2.872765
Critical Value (10%)        -2.572752
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary

Results of Dickey-Fuller Test for column: Low
Test Statistic           -0.981046
p-value                  0.760114
No Lags Used            13.000000
Number of Observations Used   248.000000
Critical Value (1%)        -3.456996
Critical Value (5%)         -2.873266
Critical Value (10%)        -2.573019
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary

Results of Dickey-Fuller Test for column: Close
Test Statistic           -1.265244
p-value                  0.644919
No Lags Used            0.000000
Number of Observations Used   261.000000
Critical Value (1%)        -3.455656
Critical Value (5%)         -2.872678
Critical Value (10%)        -2.572705
dtype: float64
Conclusion:====>
Fail to reject the null hypothesis
Data is non-stationary
```

Make a copy of the data, and let's perform the test train split.

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

The train will have all the data except the last 30 days, and the test will contain only the last 30 days to evaluate against predictions.

```
X = df[['Open', 'High', 'Low', 'Close']]  
train, test = X[0:-30], X[-30:]
```

Make the data stationary by using Pandas differencing.

```
train_diff = train.diff()  
train_diff.dropna(inplace = True)
```

Check whether the variables are stationary after first differencing.

```
for name, column in train_diff[['Open', 'High', 'Low', 'Close']]�数字:  
    Augmented_Dickey_Fuller_Test_func(train_diff[name],name)  
    print('\n')
```

```
Results of Dickey-Fuller Test for column: Open  
Test Statistic           -1.579687e+01  
p-value                  1.085613e-28  
No Lags Used             0.000000e+00  
Number of Observations Used 2.300000e+02  
Critical Value (1%)      -3.459106e+00  
Critical Value (5%)       -2.874190e+00  
Critical Value (10%)      -2.573512e+00  
dtype: float64  
Conclusion:====>  
Reject the null hypothesis  
Data is stationary  
  
Results of Dickey-Fuller Test for column: High  
Test Statistic           -1.172782e+01  
p-value                  1.364178e-21  
No Lags Used             1.000000e+00  
Number of Observations Used 2.290000e+02  
Critical Value (1%)      -3.459233e+00  
Critical Value (5%)       -2.874245e+00  
Critical Value (10%)      -2.573541e+00  
dtype: float64  
Conclusion:====>  
Reject the null hypothesis  
Data is stationary
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

```
Results of Dickey-Fuller Test for column: Low
Test Statistic           -3.997846
p-value                  0.001422
No Lags Used            12.000000
Number of Observations Used 218.000000
Critical Value (1%)      -3.460708
Critical Value (5%)      -2.874891
Critical Value (10%)     -2.573886
dtype: float64
Conclusion:====>
Reject the null hypothesis
Data is stationary

Results of Dickey-Fuller Test for column: Close
Test Statistic           -1.649354e+01
p-value                  2.181200e-29
No Lags Used            0.000000e+00
Number of Observations Used 2.300000e+02
Critical Value (1%)      -3.459106e+00
Critical Value (5%)      -2.874190e+00
Critical Value (10%)     -2.573512e+00
dtype: float64
Conclusion:====>
Reject the null hypothesis
Data is stationary
```

Cointegration is used to check for the existence of a long-run relationship between two or more variables. However, the correlation does not necessarily mean “long run.”

We can see the test says that there is the presence of a long-run relationship between features.

```
from statsmodels.tsa.vector_ar.vecm import coint_johansen

def cointegration_test(df):
    res = coint_johansen(df,-1,5)
    d = {'0.90':0, '0.95':1, '0.99':2}
    traces = res.lr1
    cvts = res.cvt[:, d[str(1-0.05)]]
    def adjust(val, length= 6):
        return str(val).ljust(length)
    print('Column Name    >  Test Stat > C(95%)    =>  Signif \n',
          '--'*20)
    for col, trace, cvt in zip(df.columns, traces, cvts):
```

```

print(adjust(col), ' > ', adjust(round(trace,2), 9), ">",
      adjust(cvt, 8), ' => ', trace > cvt)

cointegration_test(train_diff[['Open', 'High', 'Low', 'Close']])

```

Column Name	>	Test Stat	> C(95%)	=>	Signif
Open	>	311.57	> 40.1749	=>	True
High	>	201.62	> 24.2761	=>	True
Low	>	102.52	> 12.3212	=>	True
Close	>	32.21	> 4.1296	=>	True

Define the inverse differencing function.

```

def inverse_diff(actual_df, pred_df):
    df_res = pred_df.copy()
    columns = actual_df.columns
    for col in columns:
        df_res[str(col) + '_1st_inv_diff'] = actual_df[col].
            iloc[-1] + df_res[str(col)].cumsum()
    return df_res

```

Let's define a parameter grid for selecting p, q, and trend.

```

from sklearn.model_selection import ParameterGrid
param_grid = {'p': [1,2,3], 'q':[1,2,3], 'tr': ['n','c','t','ct']}
pg = list(ParameterGrid(param_grid))
print(pg)

[{'p': 1, 'q': 1, 'tr': 'n'},
 {'p': 1, 'q': 1, 'tr': 'c'},
 {'p': 1, 'q': 1, 'tr': 't'},
 {'p': 1, 'q': 1, 'tr': 'ct'},
 {'p': 1, 'q': 2, 'tr': 'n'},
 {'p': 1, 'q': 2, 'tr': 'c'},
 {'p': 1, 'q': 2, 'tr': 't'},
 {'p': 1, 'q': 2, 'tr': 'ct'},
 {'p': 1, 'q': 3, 'tr': 'n'},
 {'p': 1, 'q': 3, 'tr': 'c'},
 ...]

```

The following logic evaluates all possible combinations of p, q, and trend, and stores the results in a df_results_moni DataFrame for further reference. A grid search typically takes time.

```
df_results_moni = pd.DataFrame(columns=['p', 'q', 'tr','RMSE open','RMSE high','RMSE low','RMSE close'])
print('starting grid search')
start = timer()
for a,b in enumerate(pg):
    p = b.get('p')
    q = b.get('q')
    tr = b.get('tr')
    model = VARMAX(train_diff, order=(p,q), trend=tr).fit()
    z = model.forecast(y=train_diff[['Open', 'High', 'Low', 'Close']].values, steps=30)
    df_pred = pd.DataFrame(z, columns=['Open', 'High', 'Low', 'Close'])
    res = inverse_diff(df[['Open', 'High', 'Low', 'Close']], df_pred)
    openrmse = np.sqrt(metrics.mean_squared_error(test.Open, res.Open_1st_inv_diff))
    highrmse = np.sqrt(metrics.mean_squared_error(test.High, res.High_1st_inv_diff))
    lowrmse = np.sqrt(metrics.mean_squared_error(test.Low, res.Low_1st_inv_diff))
    closermse = np.sqrt(metrics.mean_squared_error(test.Close, res.Close_1st_inv_diff))
    df_results_moni = df_results_moni.append({'p': p, 'q': q, 'tr': tr,'RMSE open': openrmse,'RMSE high':highrmse,'RMSE low':lowrmse,'RMSE close':closermse }, ignore_index=True)
end = timer()
print(f' Total time taken to complete grid search in seconds: {(end - start)}')
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

Print DataFrame values in the sorted order of each feature RMSE.

```
df_results_moni.sort_values(by = ['RMSE open','RMSE high',
'RMSE low','RMSE close'])
```

p	q	tr	RMSE open	RMSE high	RMSE low	RMSE close	
4	1	2	n	80.987719	96.246544	94.158958	95.486292
32	3	3	n	82.551701	80.962078	96.313513	103.688361
28	3	2	n	82.605565	78.874891	96.852543	101.928501
24	3	1	n	86.654270	75.560942	98.446924	107.307196
0	1	1	n	101.458653	84.991724	103.708241	125.742375
12	2	1	n	103.037176	75.009422	93.110182	130.938880

From the previous example, we can see that p=1, q=2, tr=n gives the least RMSE.

Let's fit and forecast, as shown here:

```
model = VARMAX(train_diff[['Open', 'High', 'Low', 'Close']],
order=(1,2),trends = 'n').fit( disp=False)
result = model.forecast(steps = 30)
```

Now let's inverse the forecasted results, as shown here:

```
res = inverse_diff(df[['Open', 'High', 'Low', 'Close']],result)
res.head(5)
```

	Open	High	Low	Close	Open_1st_inv_diff	High_1st_inv_diff	Low_1st_inv_diff	Close_1st_inv_diff
231	-96.317255	-110.279610	-30.015331	18.927892	19932.301886	19918.339531	19840.375294	19990.058751
232	31.599967	6.413478	23.437892	-12.992426	19963.901852	19924.753009	19863.813187	19977.066325
233	-12.103032	3.218236	-5.871039	16.364100	19951.798821	19927.971244	19857.942148	19993.430426
234	17.357495	17.975346	23.805159	17.833004	19969.156316	19945.946590	19881.747307	20011.263430
235	17.241383	14.547660	13.346344	14.376686	19986.397700	19960.494250	19895.093650	20025.640116

Evaluate the results individually, as shown here:

```
for i in ['Open', 'High', 'Low', 'Close']:
    print(f'Evaluation metric for {i}')
    timeseries_evaluation_metrics_func(test[str(i)],
    res[str(i) + '_1st_inv_diff'])
```

CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA

Evaluation metric for Open
Evaluation metric results:-
MSE is : 78932.81890429066
MAE is : 245.060833331591
RMSE is : 280.9498512266747
MAPE is : 1.2323681201707932
R2 is : -11.912124973927812

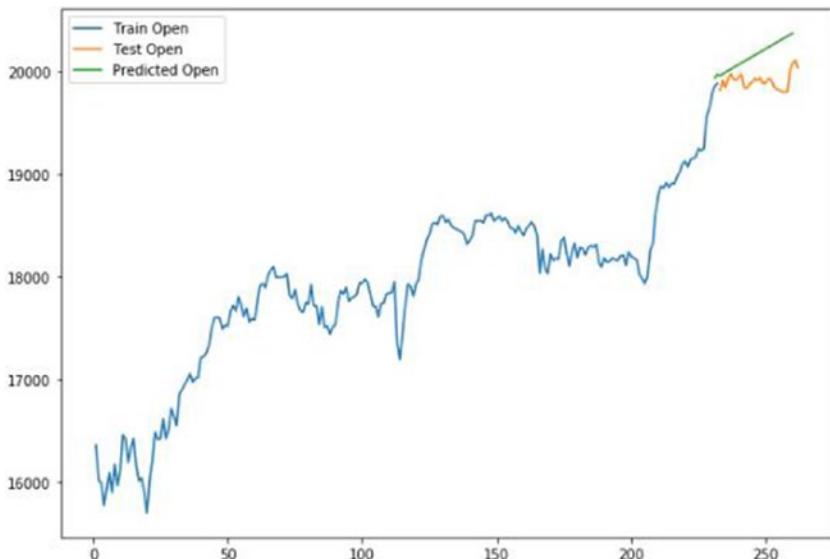
Evaluation metric for High
Evaluation metric results:-
MSE is : 46930.11346010076
MAE is : 176.12939881099143
RMSE is : 216.6335926399707
MAPE is : 0.8838651145452486
R2 is : -7.4322905049849854

Evaluation metric for Low
Evaluation metric results:-
MSE is : 72940.82061668455
MAE is : 223.955727860263
RMSE is : 270.075583155317
MAPE is : 1.1303981745020197
R2 is : -7.5414337986834425

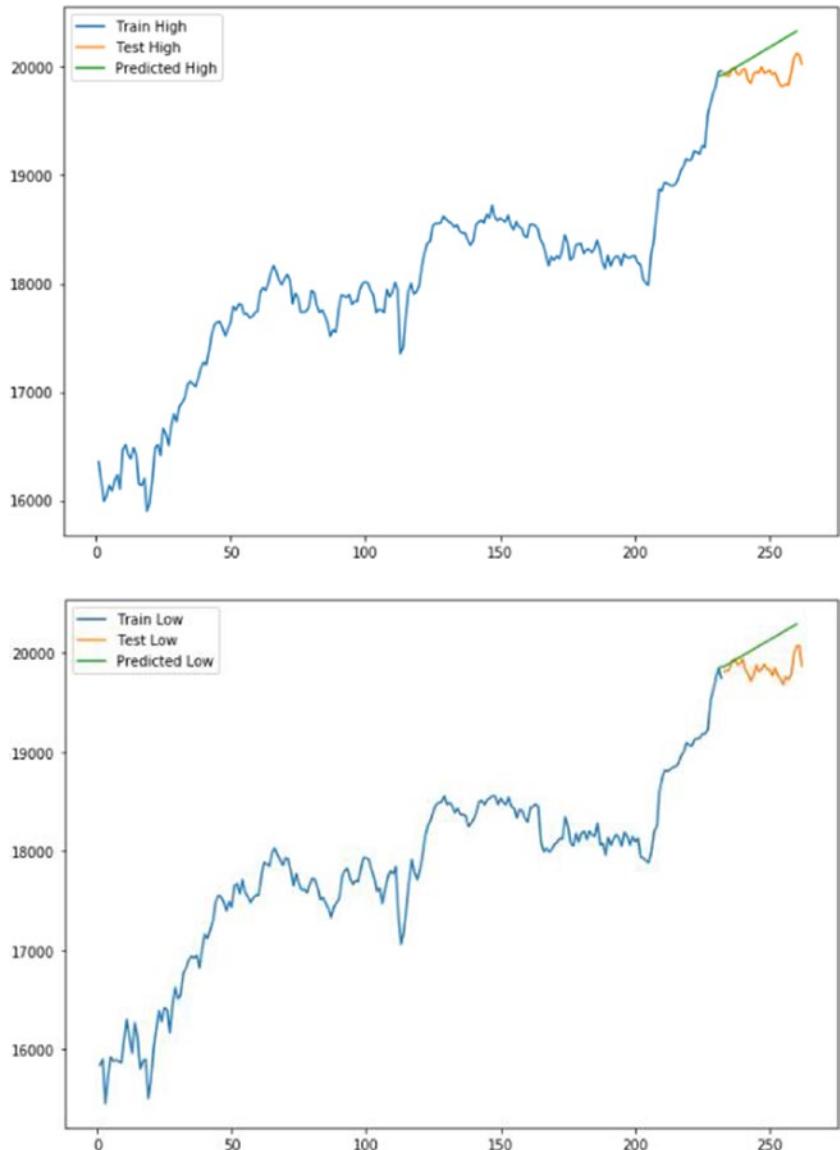
Evaluation metric for Close
Evaluation metric results:-
MSE is : 101615.67035334762
MAE is : 286.7165495650628
RMSE is : 318.77212919787644
MAPE is : 1.4420355123692148
R2 is : -12.413321290529048

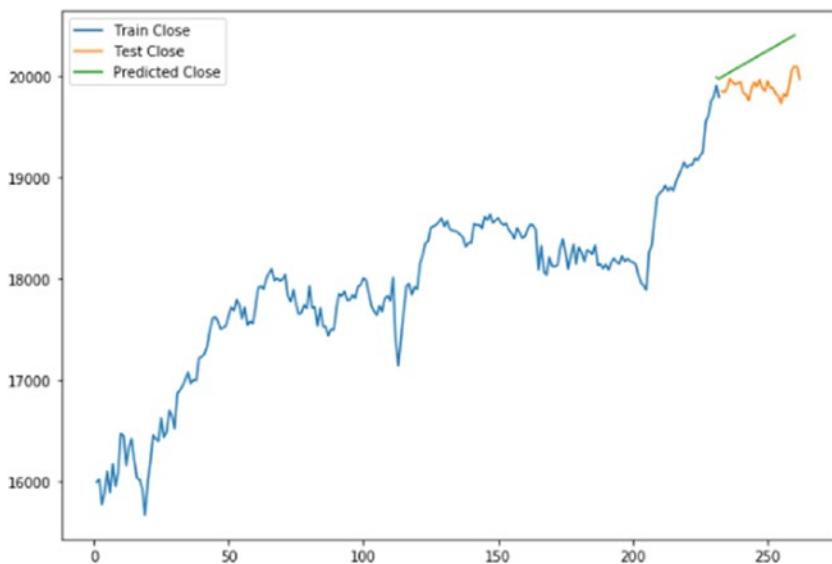
Plot the results, as shown here:

```
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
for i in ['Open', 'High', 'Low', 'Close']:
    plt.rcParams["figure.figsize"] = [10,7]
    plt.plot( train[str(i)], label='Train '+str(i))
    plt.plot(test[str(i)], label='Test '+str(i))
    plt.plot(res[str(i)+'_1st_inv_diff'], label='Predicted '+str(i))
plt.legend(loc='best')
plt.show()
```



CHAPTER 4 REGRESSION EXTENSION TECHNIQUES FOR TIME-SERIES DATA





We have observed the same problem and used auto-arima to identify p and q. Please check the code notebook VARMA with Auto Arima.ipynb.

Summary

In this chapter, you learned about stationary data, including types of stationary data, interpreting p-values, and ways to make data stationary. We also delved into the math and implementation of AR, MA, ARIMA, SARIMA, SARIMAX, VAR, and VARMA. In the next chapter, you will learn some bleeding-edge techniques for time-series data.

CHAPTER 5

Bleeding-Edge Techniques

This chapter will focus on theory behind some bleeding-edge techniques and will prepare you to solve problems using univariant and multivariant data for time-series models using deep learning (covered in Chapters 6 and 7). In this chapter, we'll start with an introduction to Neural Networks by reviewing Perceptrons, Activation functions, Backpropagation, types of Gradient Descent, Recurrent Neural networks, Long Short-Term Memory, Gated Recurrent Units, Convolutional Neural Networks, and Auto-Encoders.

Introduction to Neural Networks

In 1958, Cornell University's Frank Rosenblatt developed a system called a *perceptron* with the support of the US Navy. This was probably the earliest prototype of a neural network. His system could identify simple geometric shapes from pictures of size 20x20 px. In fact, Rosenblatt was not trying to build a system to recognize images but to study how the brain worked by using a computer simulation of the brain. As a result, his results attracted the interest of many scientists. As the *New York Times* reported in July 1958, "Today the Navy has demonstrated a prototype of an electronic system that is expected to be able to walk, speak, look, self-replicate, and have self-awareness."

A human brain has billions of neurons. *Neurons* are interconnected cells in the human brain that are responsible for processing and transforming human thoughts, which contain electrical and chemical signals. *Dendrites* are branches that receive information from other neurons.

In Figure 5-1, the left side shows the biological brain architectures that contain different neurons (vertices), which connect with different dendrites (branches) through axons (edges). The right side of the figure shows the human-made structure of a neural network, which includes inputs X_1, X_2, \dots, X_n , that includes an activation function and generates the final output.

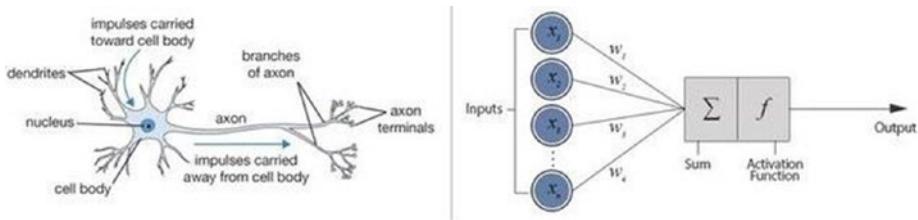


Figure 5-1. Biological neurons versus artificial neural network (source: Mwandau, Brian & Nyanchama, Matunda, 2018; Investigating Keystroke Dynamics as a Two-Factor Biometric Security)

Perceptron

A *perceptron* is a single-layer neural network architecture with one neuron. It is a mathematical architecture in which each neuron takes inputs, weighs them separately, sums them up, and passes the sum through a nonlinear function (activation function) to produce output. Figure 5-2 illustrates a perceptron.

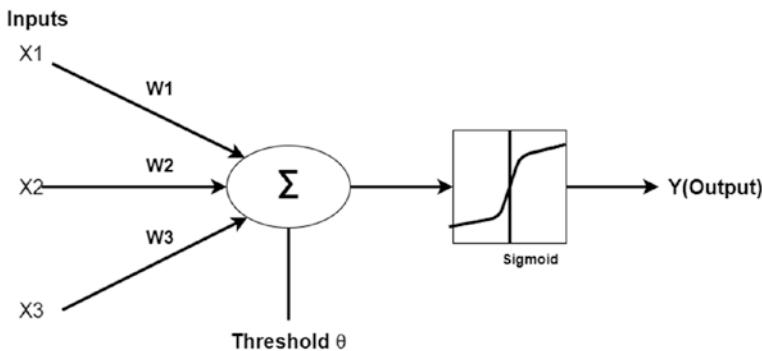


Figure 5-2. Perceptron mathematical representation

The following is the perceptron model:

Input: \$X_1, X_2, X_3\$

Weights: \$w_1, w_2, w_3\$

Bias: \$b\$

Outputs: \$Y\$

Here are the steps:

Step 1: Multiply all inputs, \$X_1, X_2, X_3\$, to all weights, \$w_1, w_2, w_3\$.

$$f = \sum_{j=0}^k w_i x_j + b$$

$$y' = f(y) = w_1x_1 + w_2x_2 + w_3x_3 + b$$

Step 2: Apply the activation function sigmoid.

$$y' = \begin{cases} 1 & \text{if } f > z \\ 0 & \text{otherwise} \end{cases}$$

Activation Function

The activation function is also known as the *radial basis function*. It is a mathematical function or formula that gives the output of the model. It gives a result usually from 0 to 1 and from -1 to 1.

The following are reasons why you should use an activation function:

- If the activation function is not used, the output of each layer is a linear function of the input of the upper layer. It doesn't matter how many layers the neural network has, but the output is a linear combination of inputs.
- If the activation function is used, it introduces nonlinear factors to the neuron so that the neural network can approach any nonlinear function arbitrarily and the neural network can be applied to many nonlinear models. Figure 5-3 illustrates the types of activation.

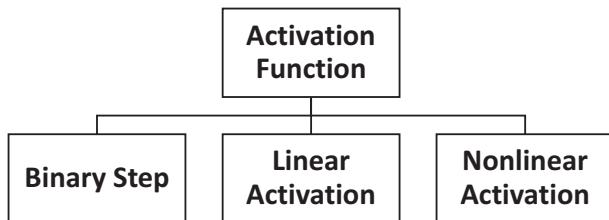


Figure 5-3. Types of activation function

Binary Step Function

The binary step function (Figure 5-4) is a threshold based activation function. If there is an output value above and below a certain threshold, neurons will activate and send precisely the same signal to the next layers.

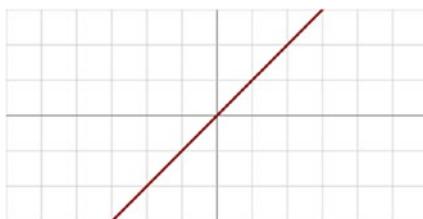


$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

Figure 5-4. Binary step function

Linear Activation Function

The linear activation function (Figure 5-5) is a simple mathematical formulation where the value of the input is multiplied with some assigned weight and produces an output. It is more preferable than the binary step function because it supports multiple outputs.



$$f(x) = x$$

Figure 5-5. Linear activation function

The following are drawbacks of the linear activation function:

- **Backpropagation is challenging to use:** Derivation of the linear activation function is constant. So, it's hard to take backpropagation on a constant value.
- **All layers of the neural network merge into one:** When you design a neural network with the help of a linear activation function, the last layer will be the

same linear activation function of the first layer. So, this linear activation function turns the neural network into one layer.

So, a linear activation function with a neural network results in a standard linear regression model that doesn't have the robust strength to maintain the complexity of data.

Nonlinear Activation Function

A traditional neural network utilizes nonlinear activation functions because they handle the complex input and output mapping needed. Nonlinear activation functions are necessary for training models on complex data types, such as raw data, images, audio, and video, that are nonlinear and higher dimensional in shape.

- A nonlinear activation function allows backpropagation, which contains a derivation function related to input values.
- A nonlinear activation function allows multiple layers to stack and helps to produce the neural network. This network comprises numerous hidden layers that are responsible for handling the complexity of data and predicting an accurate result. Figure 5-6 illustrates the types of nonactivation function.

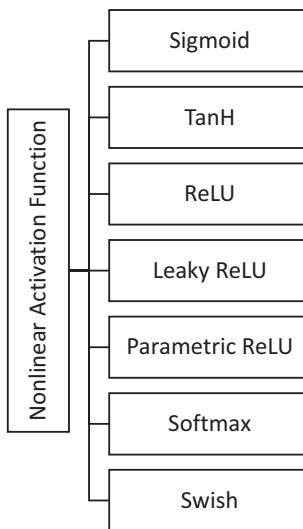
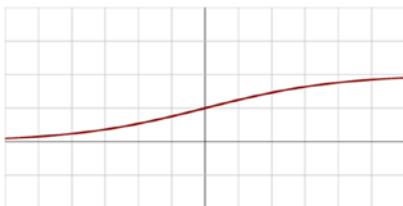


Figure 5-6. Types of nonlinear activation function

Sigmoid

A sigmoid activation function (Figure 5-7) is bound to the output values range $[0, 1]$. In other words, we can say it normalizes the output values of every layer. It has a smooth gradient. It cannot handle the vanishing gradient problem when the high and low amounts of the input value. It is computationally expensive.

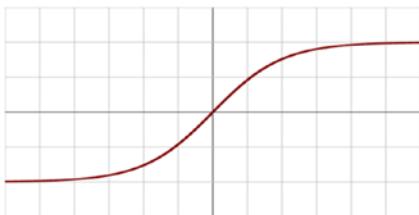


$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

Figure 5-7. Sigmoid activation function

TanH

The tanH activation function (Figure 5-8) is bound to the output values range [-1, 1]. This is the zero-model, where a model can handle strong negative, neutral, and strongly positive values. It is similar to the sigmoid function except for the range.



$$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

Figure 5-8. TanH activation function

Rectified Linear Unit

The rectified linear unit (ReLU), as shown in Figure 5-9, is an essential and widely used activation function in neural networks. A ReLU is a de facto nonsaturation of its gradient, which extensively accelerates the convergence of stochastic gradient descent associated with the sigmoid/tanH functions (according to a paper by Krizhevsky et al). It is capable of handling computation, and its expressions are similar to a linear activation function, but it allows backpropagation. When the input value is zero or negative, it cannot execute backpropagation and stop learning; this is known as the *dying ReLU problem*.

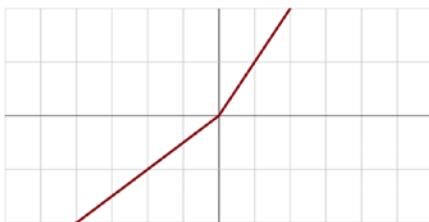


$$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$$

Figure 5-9. ReLU

Leaky ReLU

Leaky ReLU (Figure 5-10) has two types of benefits. First, it solves the dying ReLU problem, as it doesn't have a zero-slope part. It speeds up the training. Its function has a small negative slope (approximately 0.01). Its result is not consistent with negative values.



$$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Figure 5-10. Leaky ReLU

Parametric ReLU

Parametric ReLU (Figure 5-11) is a kind of leaky ReLU activation function, but it doesn't have a predefined slope like 0.01; it creates parameters for the neural network to figure out when the input value is negative. It permits a negative slope rate to be learned. It is possible to perform backpropagation and determine the most suitable amount of α . This may be implemented differently for different kinds of problems.



$$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Figure 5-11. Parametric ReLU

Softmax

A softmax activation function transforms logits into probabilities that contain a range of the sum to 1, and it is the output vector that signifies the probability distributions of a list of probable outcomes (Figure 5-12). It can handle multiple classes. Mostly the softmax layer is utilized for the output layer.

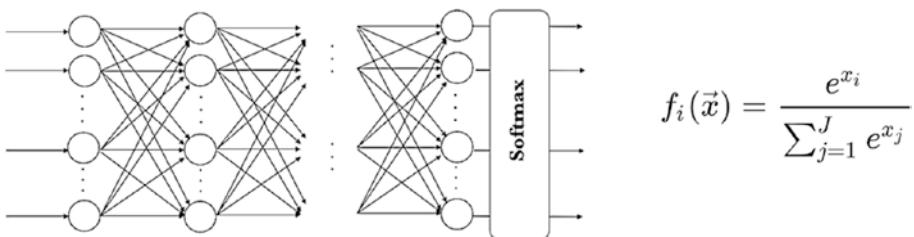


Figure 5-12. Softmax activation function

Swish The Google Brain team proposed a swish activation function in 2017 (Figure 5-13) that works better than ReLU. Swish is x times the sigmoid activation. Swish works better on a deeper model across complex data.



Figure 5-13. Swish activation

Forward Propagation

Forward propagation (Figure 5-14) is the process of taking X input values as initial information and then propagating to hidden units at each layer in the neural network to produce the final output (y).

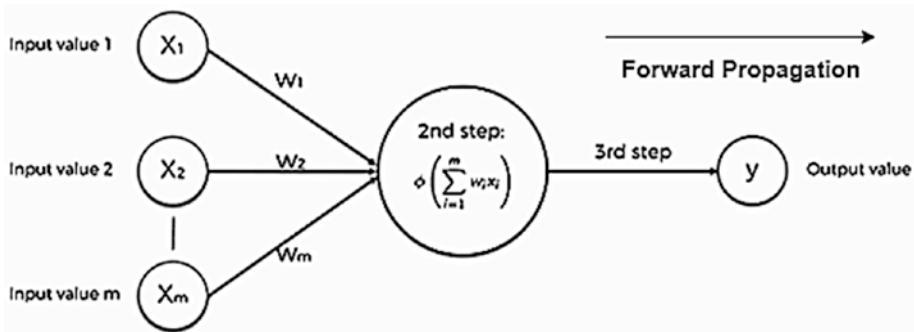


Figure 5-14. Representation of forward propagation

Step 1: Provide input (X) to each neuron in the neural network and calculate the two functions; one is linear multiplication.

$$f = \sum_{j=0}^k w_j x_j + b$$

Step 2: Apply the activation function.

$$\phi = \text{ReLU}(f)$$

We can use different activation functions. Then it will forward through every layer, and we will get the predicted output. But a neural network is not learning the information in single forward propagation. So, we calculate the information gap every time it learns, which is known as

the *cost function* (error calculation). We can try to reduce that error by applying backpropagation in the neural network.

$$C = \frac{1}{2} (y_{pred} - y_{actual})^2$$

In the next section, we will look at backward propagation.

Backward Propagation

Backward propagation, or *backpropagation*, is the process of propagating the error and going back to the input layer from the hidden layer to regulate the weights (Figure 5-15).

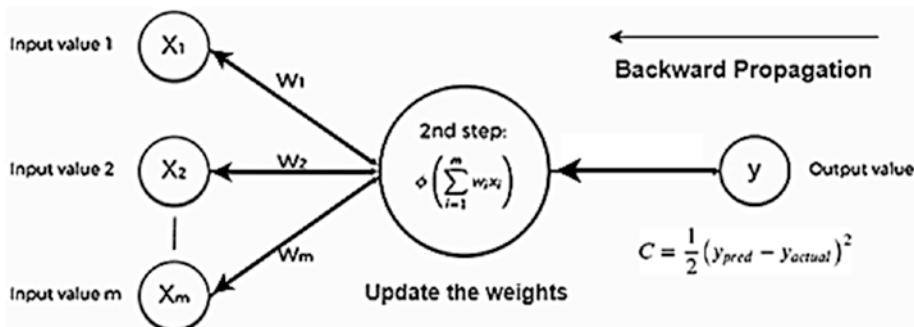


Figure 5-15. Representation of backward propagation

You can adjust the weight in two different ways.

- **Brute-force approach:** In this approach, the neural network tries to design all the possible combinations of weights and indicates only those combinations that would be best to apply. The biggest problem of this method is the curse of dimensionality when the inputs increase. It will take more computational time, and

the result might decrease inefficiency. It used only for single-layer networks like perceptrons. Figure 5-16 illustrates the brute-force approach.

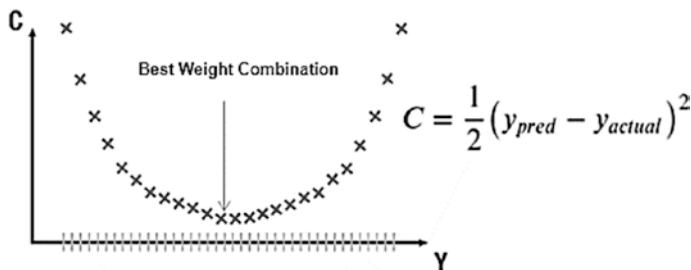


Figure 5-16. Brute-force approach

- **Gradient descent:** Gradient descent is a method for identifying the rate of change every time a value updates. In this method, we can apply the first-order derivation of iterative optimization methods to obtain the minimum cost function. This is the best way to find out the optimal weight.
 - If the gradient has a negative slope, then it represents the left side of the global minimum.
 - If the gradient has a positive slope, then it represents the right side of the global minimum.
 - If the gradient is zero, then it represents a global minima point.

Here are the mathematical representations of error analysis.
Here is the cost function:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

CHAPTER 5 BLEEDING-EDGE TECHNIQUES

Here are the update rules:

$$\theta_0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

Here are the derivatives:

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^m \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^m \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

Figure 5-17 illustrates the different types of gradient descent methodologies.

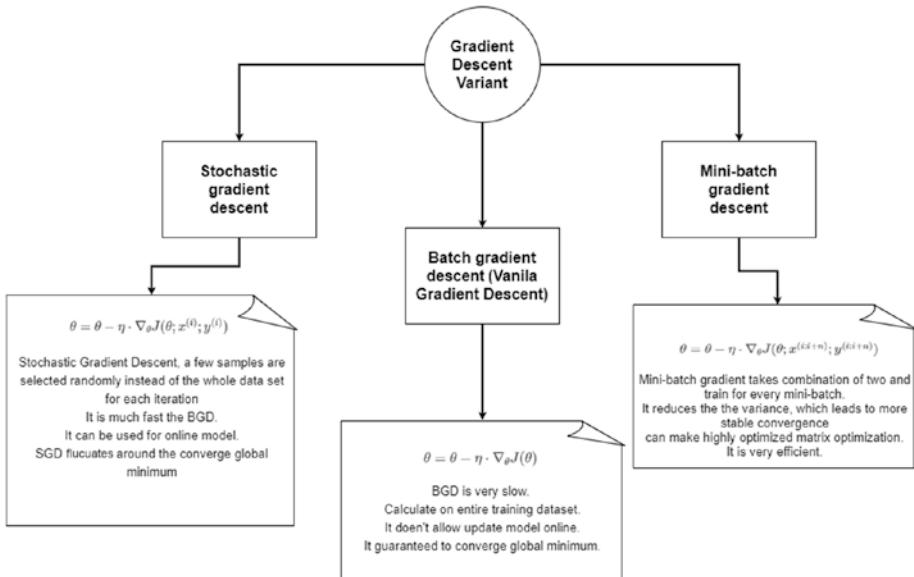


Figure 5-17. Taxonomy of gradient descent

In the next section, we will look at different types of gradient descent optimization algorithms.

Gradient Descent Optimization Algorithms

Gradient descent is a [first-order optimization algorithm](#) that is used iteratively for finding a [local minimum](#) of a differentiable function. This method minimizes convex functions. In its basic form, gradient descent finds an approximate solution to the unconstrained problem.

The gradient descent optimizer works in three main ways.

- Updates the learning rate (α)
- Updates a gradient component, $\partial L / \partial w$
- *Uses both of the previous ways*

$$w_{new} = w - \alpha \frac{\partial L}{\partial w}$$

Learning Rate vs. Gradient Descent Optimizers

The learning rate (α) is multiplied by the gradient descent optimizer function ($\partial L / \partial w$).

1. The optimizer is multiplied with the positive factor to a learning rate (α), so gradient will become smaller, which represent RMSprop.
2. Optimizers usually take the moving average of a gradient, called *momentum*, instead of just taking one value like vanilla gradient descent.

Figure 5-18 illustrates the different optimizers.

Optimizer	Year	Learning Rate	Gradient
Momentum	1964	-	✓
AdaGrad	2011	✓	-
RMSprop	2012	✓	-
Adadelta	2012	✓	-
Nesterov	2013	-	✓
Adam	2014	✓	✓
AdaMax	2015	✓	✓
Nadam	2015	✓	✓
AMSGrad	2018	✓	✓
RAdam	2019	✓	✓

Figure 5-18. Evolutionary timeline of optimization methods

Figure 5-19 and Figure 5-20 show the types of gradient descent.

**Figure 5-19.** Types of optimization methods

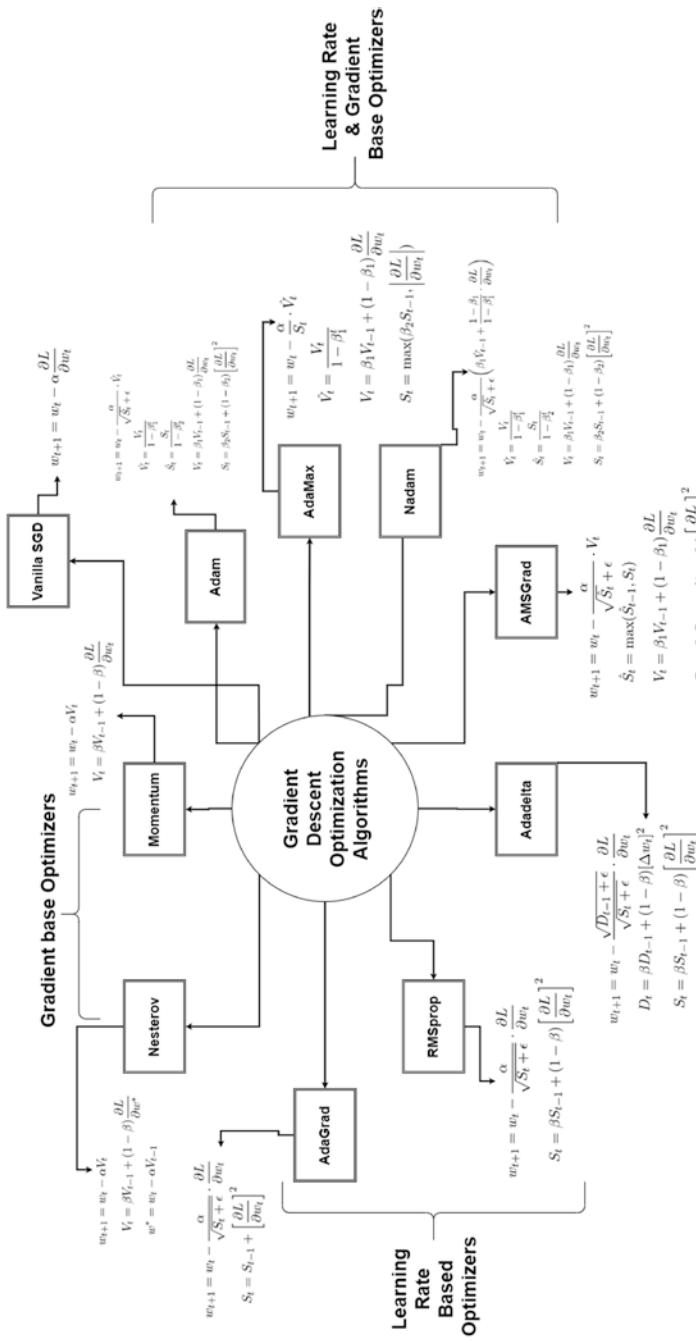


Figure 5-20. Brief chart of optimization methods with formulas

In the next section, we will look at recurrent neural networks, which are used for sequence-to-sequence learning.

Recurrent Neural Networks

A *recurrent neural network* (RNN) is a sequential class of neural network where the output from the previous state is served as input to the current state. In a neural network, all inputs and outputs are also independent of each other, but when it is required to predict things in sequence, the RNN works well.

The RNN handles data that is sequential in fashion. So, when order matters in the data, an RNN should be used. Examples of such data are natural language data, speech data, time-series data, video data, music data, and stock market data. These all are examples of real-world data that contain a sequence. Figure 5-21 illustrates the difference between RNNs versus normal NNs.

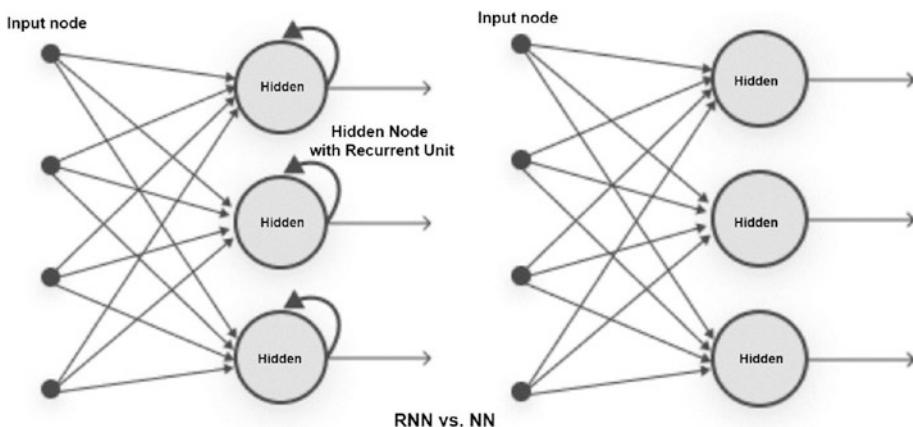


Figure 5-21. Regular RNN versus NN

The neural network does not hold this recurrent unit, where RNN does, which helps to handle the sequential pattern found in the data.

RNN has the following models, all illustrated in Figures 5-22 through 5-24.

- One to many
- Many to one
- Many to many

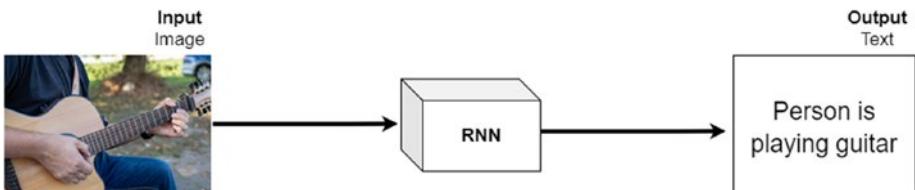


Figure 5-22. One-to-many relationship

We have given an input image (one) and get the result sequence of text (many).

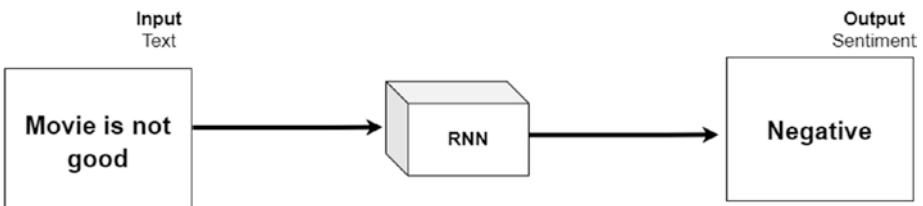


Figure 5-23. Many-to-one relationship

We have given text (many) as input and get a sentiment (one) as output.

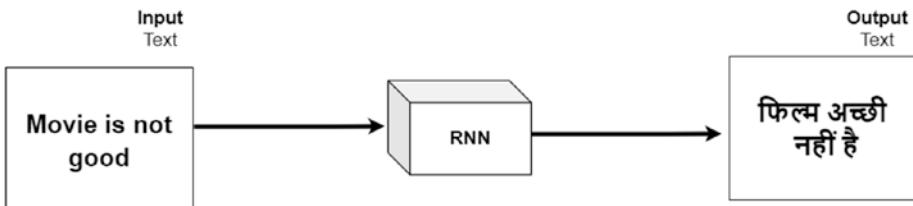


Figure 5-24. Many-to-many relationship

We have given text (many) as input and get a translated result (many) as output.

Feed-Forward Recurrent Neural Network

The feed-forward RNN is a similar structure to a standard neural network. A fundamental difference is the input sequence. In an RNN, the input sequence is the data coming in. In Figure 5-25, X_{t-1} , X_t , X_{t+1} , X_{t+2} is the input sequence. The input passes through the hidden layer and takes the previous time-step values and weights (W) at the hidden layers.

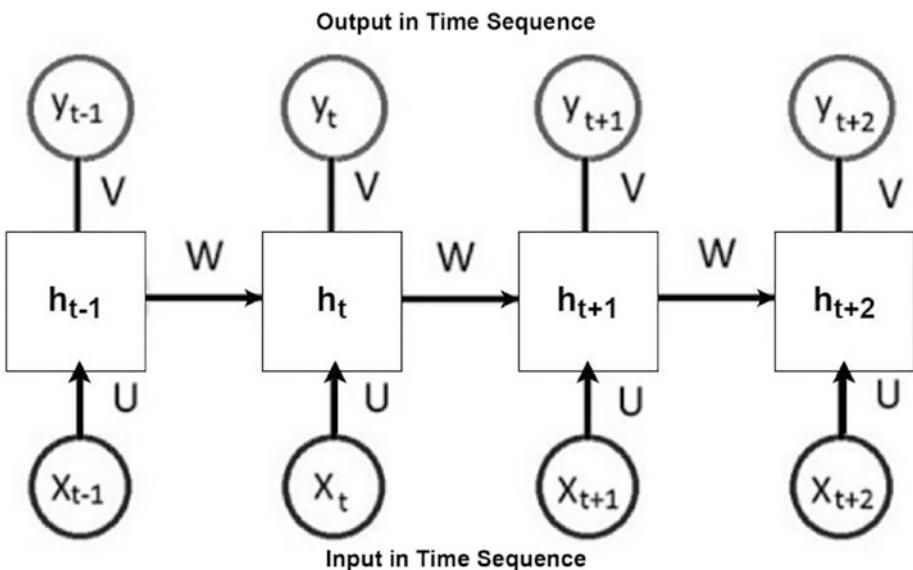


Figure 5-25. Representation of feed-forward recurrent neural network

Now let's look at the mathematical view of a feed-forward recurrent neural network.

At timestep (t), here is the equation:

$$\begin{aligned} h_t &= \sigma(U \cdot X_t + W \cdot h_{t-1}) \\ y_t &= \text{softmax}(V \cdot h_t) \end{aligned}$$

where:

σ = TanH activation function

U = Weight (vector) for the hidden layer

V = Weight (vector) for the output layer

W = Same weight vector for different timesteps

X = Input vector with sequence

Y = Output vector with sequence

Here is the cost function:

$$J(\theta) = -\sum_{j=1}^{|M|} y_{t,j} \log \bar{y}_{t,j}$$

Here is the cross-entropy:

$$J(\theta) = -\frac{1}{t} \sum_{t=1}^T \sum_{j=1}^{|M|} y_{t,j} \log \bar{y}_{t,j}$$

In Figure 5-25, we can see the number of inputs, hidden layers, and output layers.

Here is the sequence of steps to calculate the output:

1. It calculates h_{t-1} with weight U and input X.
2. It calculates y_{t-1} from h_{t-1} and output weight V.
3. It calculates the h_t value with the previous layer parameters of U, X, W, and h_{t-1} .
4. It calculates y_t from V and h_t , and it continues until the final step.

Note We are using sigmoid, tanH, and softmax activation for the RNN.

In the next section, we will look at the backpropagation approach in RNNs.

Backpropagation Through Time in RNN

The backpropagation through time approach (BPTT), as shown in Figure 5-26, is a kind of gradient descent algorithm that is used to update the weight and minimize errors with the help of a derived chain rule. This is basically applied to the sequence data, and it is different from algorithms such as RNN, LSTM, GRU, etc. We have two questions here.

- How much is the total error reduced concerning the hidden and output units?
- How does the output vary concerning the weights (U, V, W)?

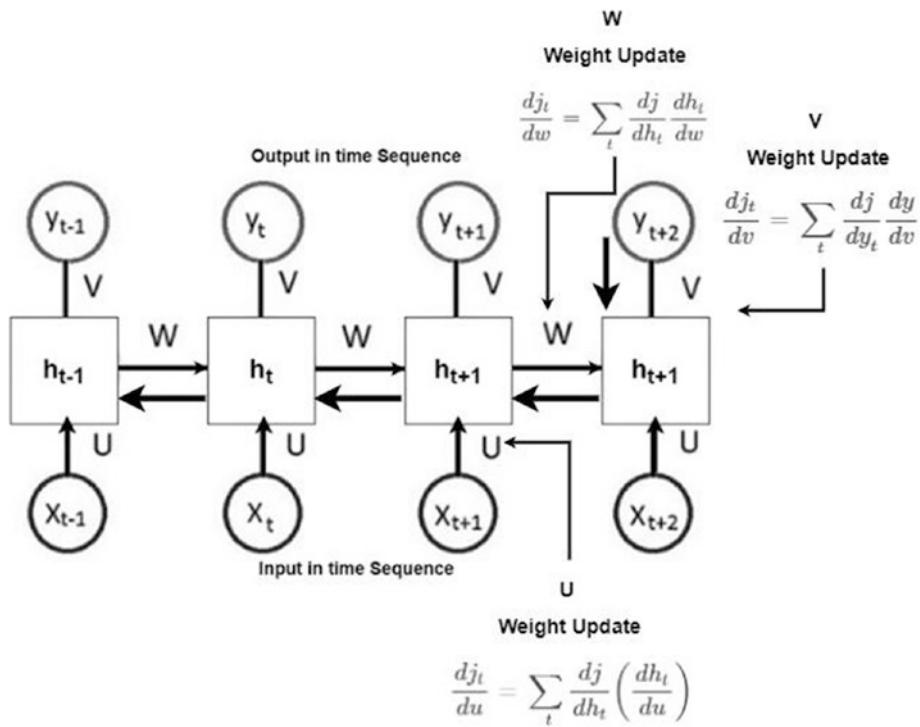


Figure 5-26. Representation of backpropagation through time in an RNN

CHAPTER 5 BLEEDING-EDGE TECHNIQUES

Backpropagation is a similar operation to the one we applied in a neural network. The only difference is that the current timesteps are calculated based on the previous one. So, we have to traverse all the paths and use the chain rule, which is shown here:

$$\begin{aligned}\frac{dj_t}{dv} &= \sum_i \frac{dj}{dy_i} \left(\frac{dy}{dv} \right)^{h_i} \\ \frac{dj_t}{dw} &= \sum_i \frac{dj}{dh_i} \left(\frac{dh_i}{dw} \right)^{(1-h_i^2) \cdot h_{t-1}} \\ \frac{dj_t}{du} &= \sum_i \frac{dj}{dh_i} \left(\frac{dh_i}{du} \right)^{(1-h_i^2) \cdot X_i}\end{aligned}$$

The weights are the same for all the timesteps.

If

$|W| < 1$ (vanishing gradient problem)

$|W| > 1$ (exploding gradient problem)

The biggest problem with an RNN is the vanishing gradient problem (VGP) or exploding gradient problem. During the backpropagation, the weights are the same for all timesteps as we continuously update the weight, so the gradient becomes either too weak or too strong with the updates, which causes either the vanishing gradient problem or the exploding problem.

The following are solutions of the vanishing gradient problem:

- **Using LSTM:** To handle this VGP
- **Using faster hardware:** Switching from a CPU to a GPU (Nvidia)
- **Using another activation function:** Such as ReLU, which suffers less from this problem
- **Using a residual network:** To avoid the problem

Note The vanishing gradient problem occurs while training an RNN. This primarily transpires when the network parameters and hyperparameters are not correctly customized.

- Parameters are weights and biases.
- Hyperparameters are the learning rate, epochs, batches, etc.
- If values contract exponentially, it leads to a vanishing gradient.
- If values get larger exponentially, it leads to an exploding gradient.

Figure 5-27 illustrates vanishing and exploding gradients.

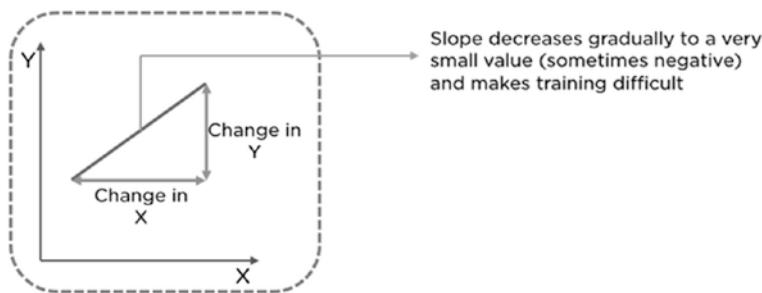


Figure 5-27. Representation of vanishing and exploding gradients

Therefore, the deep learning model needs time to train and learn from the data, and in some cases, it may not train thoroughly. This results in either less or no convergence of the neural network.

Because of the vanishing gradient problem, the gradient becomes too small and regularly decreases to a minimal value (sometimes negative). This makes the model performance weak with low accuracy. The model may fail to predict or classify what it is supposed to do.

In the next section, we will learn about long short-term memory.

Long Short-Term Memory

Long short-term memory (LSTM) was first mentioned in 1997 by Sepp Hoch Reiter and Jürgen Schmid Huber. By using constant error carousel (CEC) units, LSTM treats the exploding and vanishing gradient problems. The preliminary version of the LSTM block incorporated cells, input, and output gates. It is famous for its design, which is a specific type of recurrent neural network, utilizing many different real-world applications that the standard version doesn't. An RNN's biggest problem is that it only holds the previous state information, causing the vanishing gradient problem. This problem has been solved by LSTM.

LSTM was constructed to avoid the issue of long-term dependencies. Remembering information for a long duration is basically its default behavior. All RNNs have repeating chain modules of the neural network. This repeating module is a simple structure, such as the tanH layer in an RNN. LSTM also has an identical chaining structure instead of having a single neural network layer (Figure 5-28).

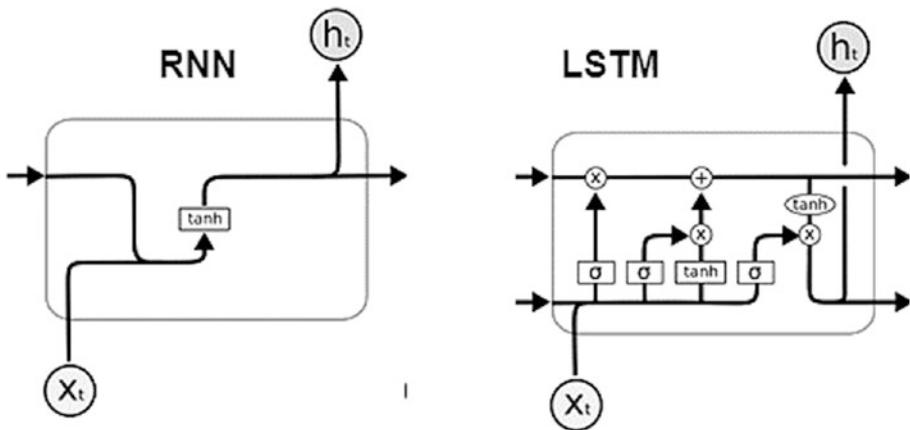


Figure 5-28. RNN versus LSTM

LSTM has an extra feature as compared to RNNs, which is called *memory*.

- Forget gate f (a neural network with sigmoid activation)
- Candidate layer C (a neural network with tanH activation)
- Input gate I (a neural network with sigmoid activation)
- Output gate O (a neural network with sigmoid activation)
- Hidden state h
- Memory cell C

Figure 5-29 illustrates that it has four different gates: forget gate, candidate gate, input gate, and output gate. All gates are single-layer neural networks with the sigmoid activation function, excluding the candidate gate, which is using the tanH activation function.

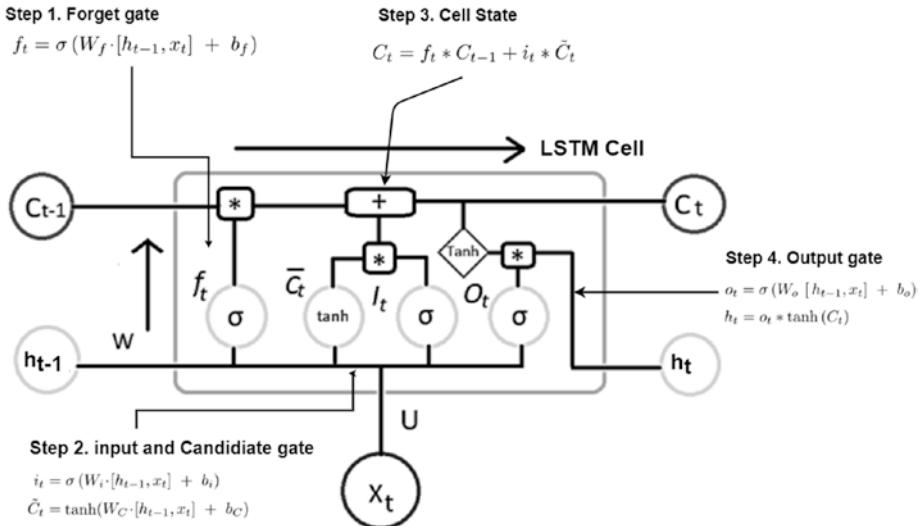


Figure 5-29. LSTM cell architecture with formulas

Figure 5-29 shows LSTM at the T time step.

- The input cell contains x (input vector), h_{t-1} (previous hidden state), and C (previous memory state).
- The output cell contains h (current hidden cell) and C (current memory cell).
- W and U are weight vectors.

Step-by-Step Explanation of LSTM

Here is a step-by-step explanation of LSTM.

Step 1: Let's say we must predict an upcoming sequence based on all the forthcoming timestamps. In such a problem, the cell state can store all the information for the present input so that the correct prediction can be made. When we get new input data, we link it to the previous pattern in the sequence time-series data.

Here is the forget gate equation:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Step 2: The next step is to make a decision on which information is important to us so we can store it. This has been classified into two parts. The first input gate layer, which contains the sigmoid layer, makes a decision on which values to update. Next, a tanH layer produces a vector for the new candidate values; this vector is called \tilde{C}_t . That could be further to the state. We can associate these two gates with each other to create an updated state.

Here are the input and candidate gate equations:

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \end{aligned}$$

Step 3: It is time to update the old cell state, C_{t-1} , into the new cell state, C_t . We multiply the old state by f_t , forgetting the unnecessary parts. Then we multiply the input gate (i_t) by the candidate gate (\tilde{C}_t), and this becomes the new candidate value, scaled by how much we decided to update each state value.

Here is the cell state equation:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Step 4: We have to make a decision about the output state. This is based on a cell state, but it can be a filtered version. The first sigmoid layer makes the decision about which part of the output of the cell state we will produce; then we put a cell state to tanH and multiply it by the output of the sigmoid gate. So, we can only generate the output we make a decision on.

Here are the equations for the output state and hidden state:

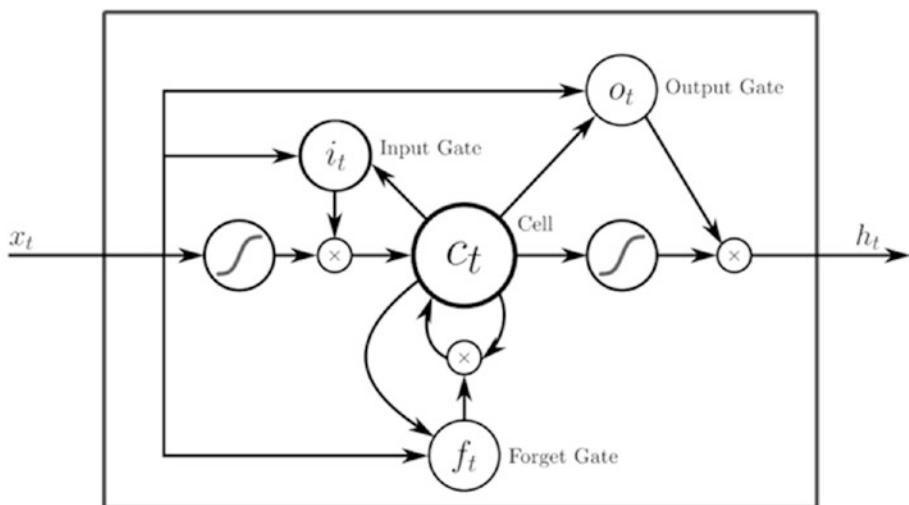
$$\begin{aligned} o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ h_t &= o_t * \tanh(C_t) \end{aligned}$$

LSTM has two types of variants, peephole and peephole convolutional, which we'll look at now.

Peephole LSTM

The LSTM cell receives a connection from the input and output cells but doesn't have a direct link with the constant error carousel, which is meant to control. We can directly observe the output, which is close to zero until the output gate is closed. The same problem happens with all memory blocks. When the output gate is locked, no single gate can access the CEC control. A simple remedy for this solution is the peephole connection from the CEC to the gates to the same memory block. The peephole connections allow all gates to inspect the current cell state even when the output gate is closed.

Figure 5-30 shows an LSTM unit with peephole connections (i.e., a peephole LSTM). Peephole connections permit the gates to approach the CEC, whose activation is the cell state. h_{t-1} is not used. c_{t-1} is used in most places.



$$f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$$

$$i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$$

$$o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + b_c)$$

$$h_t = \sigma_h(o_t \circ c_t)$$

Figure 5-30. Representation of peephole LSTM

Peephole Convolutional LSTM

LSTM has proven able to handle temporal correlation. It contains too much redundancy for spatial data. Convolutional LSTM (ConvLSTM) has solved this problem, so now we can solve the spatiotemporal sequence forecasting problem. Figure 5-31 illustrates the transformation of a 2D image into a 3D tensor.

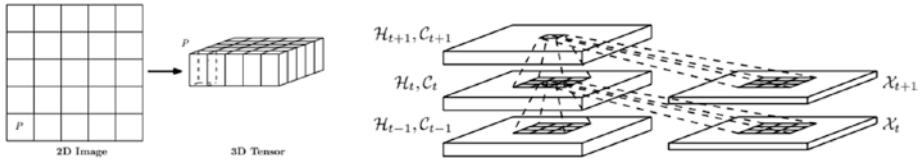


Figure 5-31. Transforming a 2D image into a 3D tensor

Figure 5-32 illustrates the inner structure of ConvLSTM mathematically.

$$\begin{aligned}
 f_t &= \sigma_g(W_f * x_t + U_f * h_{t-1} + V_f * c_{t-1} + b_f) \\
 i_t &= \sigma_g(W_i * x_t + U_i * h_{t-1} + V_i * c_{t-1} + b_i) \\
 c_t &= f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c * x_t + U_c * h_{t-1} + b_c) \\
 o_t &= \sigma_g(W_o * x_t + U_o * h_{t-1} + V_o * c_t + b_o) \\
 h_t &= o_t \circ \sigma_h(c_t)
 \end{aligned}$$

Figure 5-32. Inner structure of ConvLSTM (source: Xingjian, S. H. I., et al. “Convolutional LSTM network: A machine learning approach for precipitation nowcasting,” 2015, *Advances in neural information processing systems*)

In the next section, we will look at the gated recurrent unit and its variant.

Gated Recurrent Units

A *gated recurrent unit* (GRU) is a special kind of recurrent neural network using a gating mechanism (Figure 5-33). It was introduced by Kyunghyun Cho in 2014. The GRU is an identical structure to LSTM with a forget gate but with fewer parameters; it excludes the output gate. GRU performs well on some tasks such as audio acoustic classification and speech classification. GRU is even better on small datasets. GRU is not good for training simple languages that are learnable by LSTM because LSTM performs unbounded counting while GRU doesn’t.

Several variances have been introduced with different combinations of the previous hidden state and the bias, and its simplified version is known as a *minimal gated unit*.

GRU has two variants. One is a fully gated unit, shown in Figure 5-33.

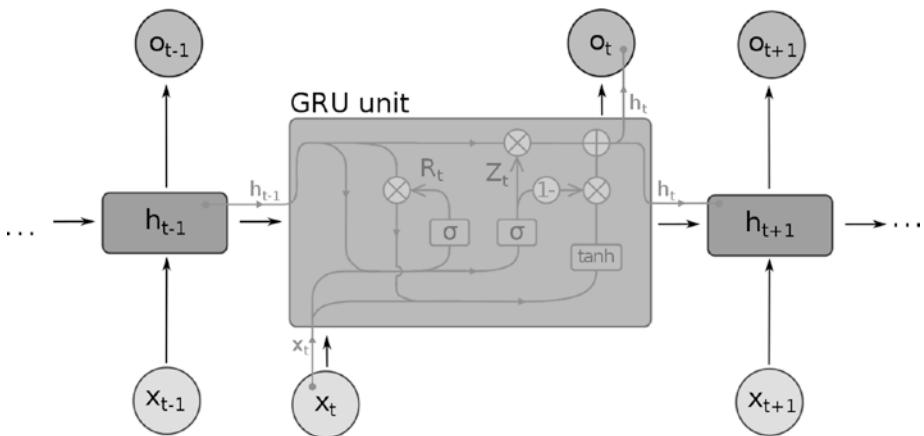


Figure 5-33. Representation of GRU

Initially, for $t = 0$, the output vector is $h_0 = 0$.

$$\begin{aligned} z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \\ r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \Phi_h(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h) \end{aligned}$$

- x_t is the input data, h_t is the output data, z_t is the update gate, and r_t is the reset gate.
- W , U , and b are parameter matrices and vectors, and two activations are used: sigmoid (σ_g) and tanH (Φ_h).

Figure 5-34 shows three variants.

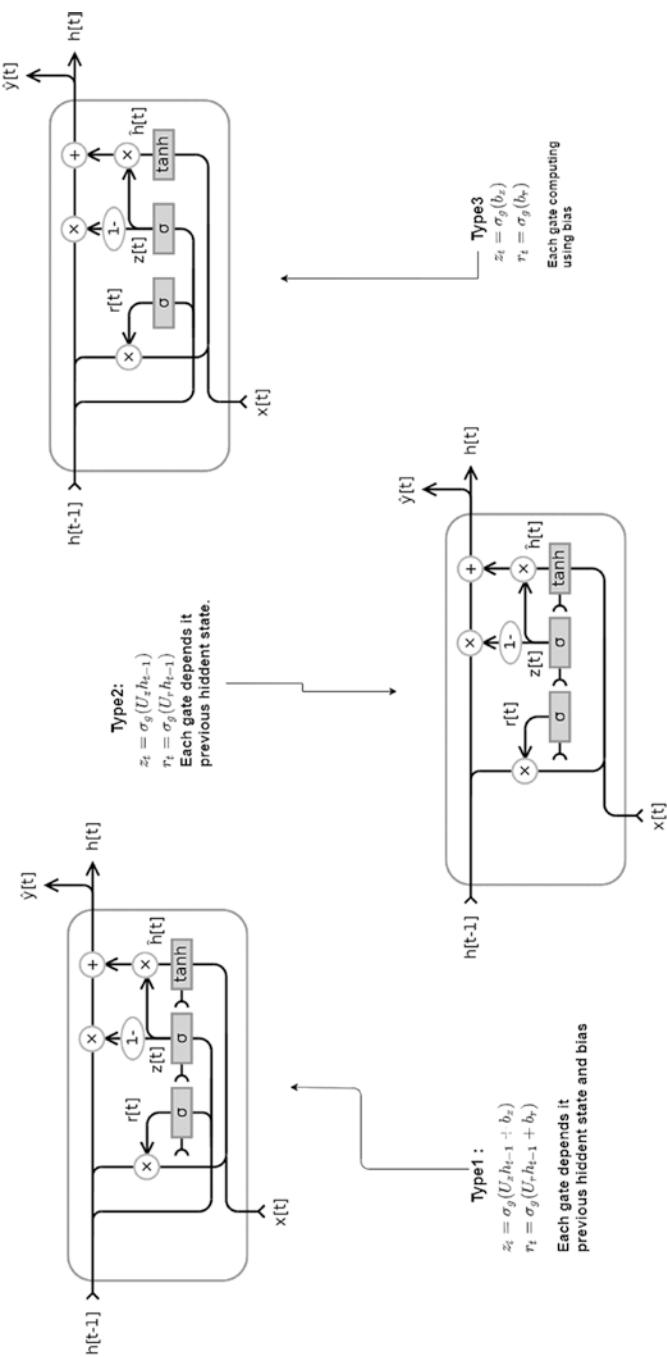


Figure 5-34. Representation of GRU variants

The second kind is the *minimal gated unit*, which is identical to the fully gated unit except a reset gate is merged into one forget gate (Figure 5-35). This suggests that the output vector will be changed.

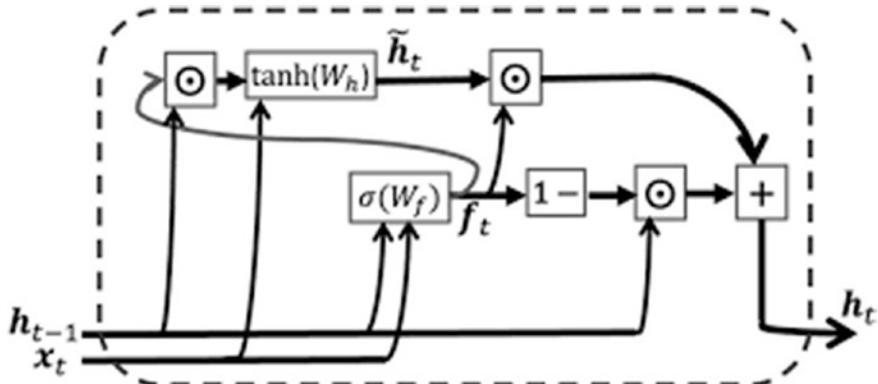


Figure 5-35. Minimal gated unit

$$\begin{aligned} f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\ h_t &= f_t \odot h_{t-1} + (1 - f_t) \odot \phi_h(W_h x_t + U_h (f_t \odot h_{t-1}) + b_h) \end{aligned}$$

- x_t is the input data, h_t is the output data, and f_t is the forget gate.
- W , U , and b are parameter matrices and vector.

In the next section, we will look at convolution neural networks.

Convolution Neural Networks

Convolution neural networks (CNNs) were first published in 1995 by LeCun and Bengio. A convolutional neural network structure is based on the neurobiological observation that neurons in our visual cortex can be recognized without problems, even when the direction and location of objects change. Unlike a multilayer perceptron, you can learn the unique characteristics of an object regardless of their positions or orientation.

A CNN comprises a convolutional layer that extracts features and contains pooling that compresses information. The convolutions and pooling layers are repeated among the input and output layers. The number of iterations is taken from the experiments to get the best results. A CNN uses nonpixel filters to solve the problem of multilayer perceptron. Convolution filters consist of matrices. The filter scans the image from the top left to the bottom right in the convolutions layer, reading the features of the image to generate a feature map. In Figure 5-36, the kernel matrix is known as a *filter*.

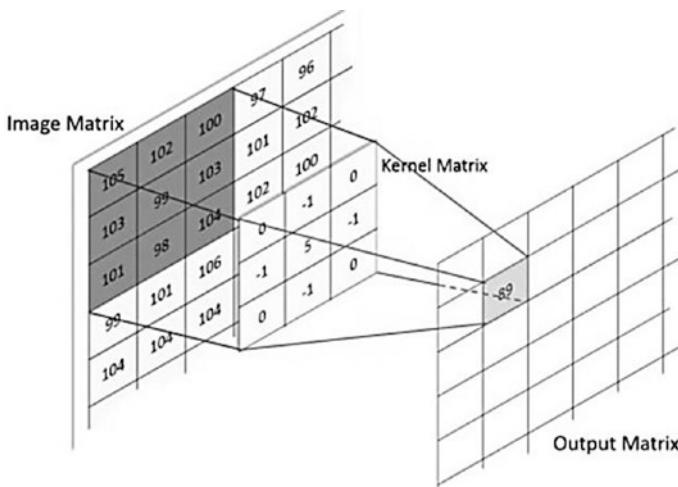


Figure 5-36. Convolution operation

In Figure 5-36, the left image is the image matrix, which is composed of three primary colors; it is represented in three dimensions with depth, and the filter is represented by a small brown box. The filter is served by the matrix of weights. As filters move from left to right, the feature map searches for the same features at different locations in the input image. If the part passed by the filter matches the feature, it will output a high value to increase the likelihood of image classification. For example, if you scan a

car image and then scan the shape of the wheel, the car is likely to be a car and will output a high value.

List the feature maps read by the filter vertically and horizontally to create one sheet, as shown in the middle image. If multiple filters are used, numerous feature maps will be created. A CNN's polling layer compresses feature maps, reducing the number of parameters (Figure 5-37). Split a 4×4 feature map into 2×2 -unit bundles to represent each bundle. There is a maximum pooling that takes the maximum and an average pooling that takes the average.

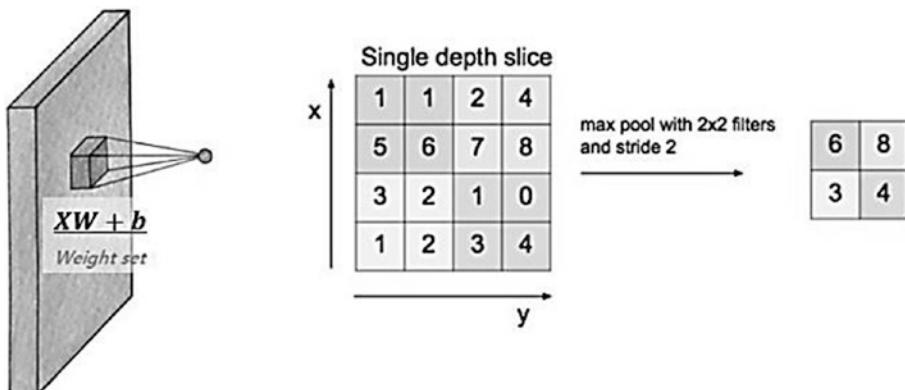


Figure 5-37. Representation of pooling

Repeat the process of creating another image by reapplying the filter based on the image. If you repeat the process of creating multiple filters and images and its characteristics are compressed, the result for the aspects of the image drops to one number. This value is a stochastic estimate. The accuracy of the CNN model is determined by how sophisticated the filter is.

Generalized CNN Formula

The following is the original mathematical representation of the LeNet 5 CNN family of algorithms designed by LeCun.

1. Convolution formula:

Input : $(nxnxn_c)$

Filter : $(fxfxn_c)$

Output : $\left(\left[\frac{n+2p-f}{s} + 1 \right] x \left[\frac{n+2p-f}{s} + 1 \right] xn'_c \right)$

This includes the filter size (f), stride (s), pad (p), and input size (n), with n'_c as the number of filters.

2. Pooling layer:

Pooling has two types.

- 1) **Max pooling:** The max value will be taken from all the cells.
- 2) **Average pooling:** The average value will be taken from all the cells.
3. **Batch normalization:** This is used to normalize the input layer by fine-tuning and scaling the activations. It makes the model training speed fast.
4. **Dropouts:** This is a method where randomly selected neurons are ignored during training. They are “dropped out” randomly. It is used to prevent model overfitting.

Table 5-1 illustrates the different layers and their mathematical formulas.

Table 5-1. Formulas of CNN Different Layers

Layer or Concept	Math
1D Convolution	$(\vec{f} * \vec{g})(t) = \sum_{x=-w}^w \vec{f}_x \cdot \vec{g}_{t-x}$
3D Convolution	$(\mathbf{F} * \mathbf{G})(\vec{t}) = \sum_x \sum_y \sum_z \mathbf{F}_{x,y,z} \cdot \mathbf{G}_{\vec{t}_x - x, \vec{t}_y - y, \vec{t}_z - z}$
Sigmoid	$S(x) = \frac{1}{1 + e^{-x}}$
ReLU	$f(x) = \max(x, 0)$
Max Pool	$f(\vec{t}, \mathbf{X}) = \max_{\substack{\vec{t}_j - w \leq i \leq \vec{t}_j + w \\ \vec{t}_j - w \leq j \leq \vec{t}_j + w \\ \vec{t}_i - w \leq k \leq \vec{t}_i + w}} \mathbf{X}_{i,j,k}$
Average Pool	$f(\vec{t}, \mathbf{X}) = \sum_{i=\vec{t}_i-w}^{\vec{t}_i+w} \sum_{j=\vec{t}_j-w}^{\vec{t}_j+w} \sum_{k=\vec{t}_k-w}^{\vec{t}_k+w} \frac{\mathbf{X}_{i,j,k}}{8w^3}$
Dense Layer	$f(\vec{x}) = \text{ReLU}(\mathbf{W}\vec{x} + \vec{b})$
L2	$\ x\ _2 = \left(\sum_i (x_i^2) \right)^{1/2}$
L1	$\ x\ _1 = \sum_i (x_i)$

In this next section, we will look at one-dimensional CNNs.

One-Dimensional CNNs

A CNN is really good at identifying simple patterns in data and then using them to form more complex patterns in higher layers. One-dimensional CNNs are useful when you want to get exciting features from shorter (fixed-length) fragments of the entire dataset and when the location of the features in that fragment is irrelevant.

This is ideal for analyzing time series of sensor data, such as gyroscope data or accelerometer data. It is also suitable for analyzing any kind of signal data (such as audio signals) over a fixed length of time. Another application is NLP (although LSTM networks are more promising here, because the proximity of words may not always be a good indication of trainable patterns).

In the next section, we will look at an auto-encoder, which is one of the popular unsupervised deep learning methods.

Auto-encoders

An *auto-encoder* is an unsupervised deep learning neural network model. It has the capability to learn the hidden features of the input data, which is called *encoding*. At the same time, it can reconstruct the original input data with the new generated features learned, which is called *decoding*.

Intuitively, auto-encoders can be used for feature dimensionality reduction, such as principal component analysis (PCA), but their performance is stronger than PCA because neural network models can extract more efficient new features. In addition to feature dimensionality reduction, new features learned by the auto-encoder can be fed into a supervised learning model, so the auto-encoder can function as a feature extractor. As an unsupervised deep learning model, auto-encoders can also be used to generate new data that is different from the training samples. In this way, auto-encoders (variational auto-encoders) are generative models.

Figure 5-38 shows the basic structure of an auto-encoder. It includes two processes: encoding and decoding.

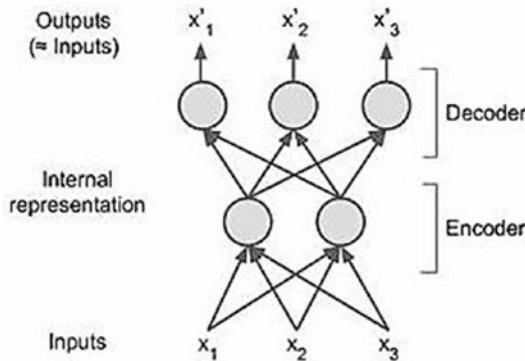


Figure 5-38. Representation of auto-encoder

The encoder input is x encoded to obtain new features y , and input x can form a new unique feature y reconstructed. The encoding process is as follows:

$$y = f(Wx + b)$$

It can be seen that, in the neural network structure, the encoding is a linear combination followed by a nonlinear activation function. If there is no nonlinear structure, then the auto-encoder is no different from ordinary PCA. With the new features y , may enter x reconstruction, i.e., the decoding process:

$$x' = f(W'x + b')$$

We hope the reconstructed formula is as consistent as possible and you can use the loss function to minimize the negative log likelihood to train the model, as shown here:

$$L = -\log P(x|x')$$

For Gaussian distribution data, it is better to use the mean square error, and for Bernoulli distribution, cross-entropy can be used, which can

be derived from the likelihood function. Under normal circumstances, we will add some restrictions to the $W = W^T$ auto-encoder, which is commonly used. This is called *tied weights*.

Summary

In this chapter, you learned about activation functions, backpropagation, types of gradient descent, recurrent neural networks, long short-term memory, gated recurrent unis, convolutional neural networks, and auto-encoders. In the next chapter, you will learn how to solve univariate time-series problems using bleeding-edge techniques.

CHAPTER 6

Bleeding-Edge Techniques for Univariate Time Series

In the previous chapter, you learned about using traditional techniques to work with time-series data. In this chapter, you will learn how to solve univariate time-series problems using bleeding-edge techniques. A *univariate time series* is a time series that consists of single (scalar) observations recorded sequentially over equally spaced time periods. In this chapter, you will look at single-step time-series forecasting and horizon-style time-series forecasting.

Single-Step Data Preparation for Time-Series Forecasting

Single-step time-series forecasting is a technique where the model is exposed to one window of data at a time, such as days, weeks, months, quarters, or years, and attempts to predict the next consecutive step, as illustrated in Figure 6-1.

Example 1: Data is at the monthly level. The model is shown the first window from the 1st to the 48th, which means four years of data, and it predicts the 49th month. Then, for the next iteration, the model looks at the 2nd to 49th months for training and tries to predict the 50th month.

Example 2: Data is at the daily level. The model is shown the first window from the 1st to the 90th day (i.e., three months of data) and predicts the 91st day's value. Then, in the next iteration (the 2nd to 91st day) for training, it tries to predict the 92nd day.

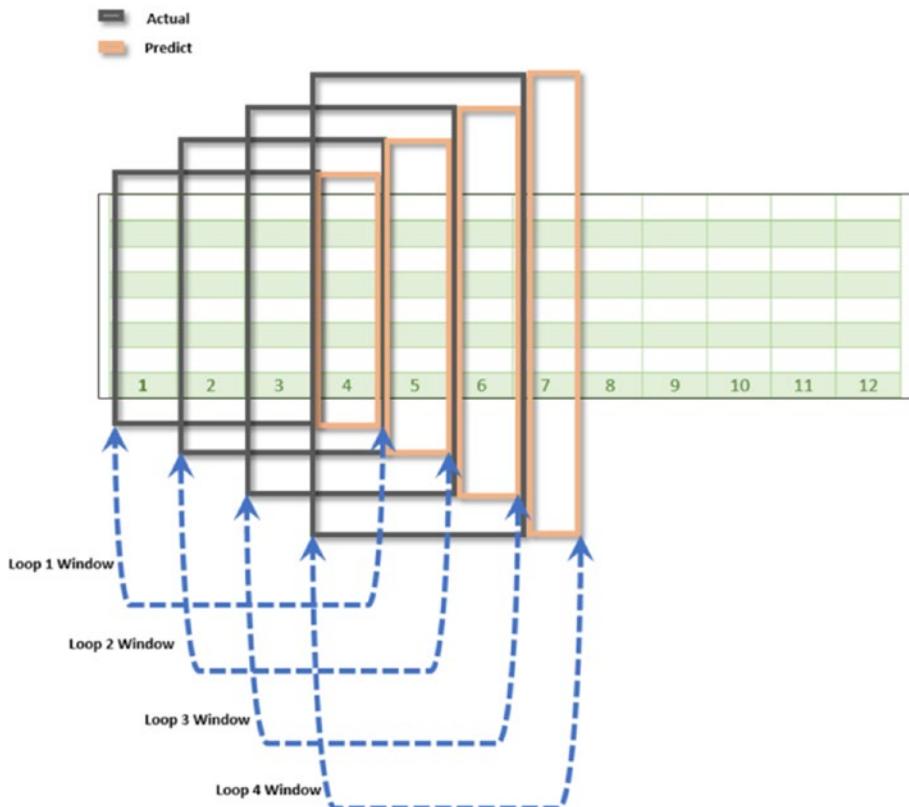


Figure 6-1. Representation of single-step time-series series forecasting

Horizon-Style Data Preparation for Time-Series Forecasting

Horizon-style time-series forecasting is a technique where the model is exposed to one window of data at a time, such as days, weeks, months, quarters, or years, and attempts to predict the next n consecutive steps based on selection, as illustrated in Figure 6-2.

Example 1: Data is at the monthly level. The model is shown the first window from the 1st to the 48th (which means four years of data) and predicts the values for the 49th to 59th months. Then, in the next iteration (the 2nd to 49th months) for training, it tries predicting the 50th to 60th months.

Example 2: Data is at the daily level. The model is shown the first window from the 1st to 90th day (i.e., three months of data) and tries to predict the values for the 91th to 101st days. Then, in the next iteration (the 2nd to 91st days) for training, it tries to predict the 92nd to 102nd days.

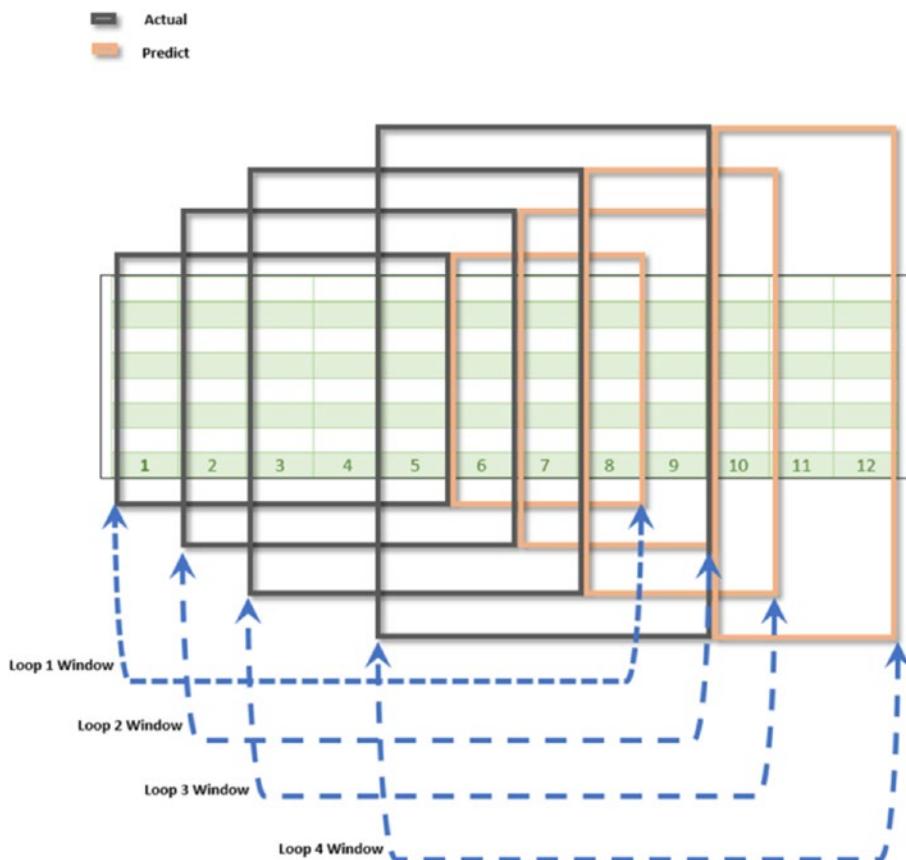


Figure 6-2. Representation of horizon-style time-series forecasting

LSTM Univariate Single-Step Style in Action

In the previous section, you learned about single-step and horizon-style time series data preparation for forecasting. In this section, let's use single-step data preparation and LSTM to solve univariate time-series problems.

Import the required libraries and load the CSV data, which contains hourly Interstate 94 westbound traffic volumes for MN DoT ATR station 301, roughly midway between Minneapolis and St Paul, Minnesota. Hourly weather features and holidays are included that impact the traffic volume.

- **Tensorflow** is an open source artificial intelligence library that uses data flow graphs to build models. It allows developers to create large-scale neural networks with many layers. The following code implementation has been tested and works best on Tensorflow and Tensorflow GPU version 2.0 and onward.

Let's use this data and forecast the traffic volume for the next ten hours.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour has different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

For the train/test split, let's hold back ten hours of data (i.e., ten records), which we can use to validate the data after training on the past data.

```
validate = df['traffic_volume'].tail(10)
df.drop(df['traffic_volume'].tail(10).index,inplace=True)

uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()
```

```
date_time
2012-10-02 09:00:00    5545
2012-10-02 10:00:00    4516
2012-10-02 11:00:00    4767
2012-10-02 12:00:00    5026
2012-10-02 13:00:00    4918
Name: traffic_volume, dtype: int64
```

Let's rescale the data because neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))
```

Define a function to prepare univariate data that is suitable for a time series.

```
def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []
    start = start + window
    if end is None:
        end = len(dataset) - horizon
    for i in range(start, end):
        indicesx = range(i-window, i)
        X.append(np.reshape(dataset[indicesx], (window, 1)))
        indicesy = range(i, i+horizon)
        y.append(dataset[indicesy])
    return np.array(X), np.array(y)
```

As we are doing a single-step forecast, let's allow the model to see/train on the past 48 hours of data and try to forecast the 49th hour. Hence, use `horizon = 1`.

```
univar_hist_window = 48
horizon = 1
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT, univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None, univar_hist_window, horizon)
print ('Single window of past history')
```

```
print (x_train_uni[0])
print ('\n Target horizon')
print (y_train_uni[0])
```

Single window of past history

```
[[0.76167582]
 [0.62032967]
 [0.65480769]
 [0.69038462]
 [0.67554945]
 [0.71167582]
 [0.76703297]
 [0.82623626]
 [0.79546703]
 [0.65521978]
 [0.48612637]
 [0.38241758]
 [0.32431319]
 [0.21002747]
 [0.13228022]
 [0.06950549]
 [0.04409341]
 [0.0375    ]
 [0.05041209]
 [0.11181319]
 [0.37335165]
 [0.77925824]
 [0.89436813]
 [0.75151099]
 [0.70013736]
 [0.67129121]
 [0.7331044 ]]
```

```
[0.78186813]
```

```
[0.84299451]
```

```
[0.63502747]
```

```
[0.49326923]
```

```
[0.39807692]
```

```
[0.36222527]
```

```
[0.24409341]
```

```
[0.13942308]
```

```
[0.08214286]
```

```
[0.05068681]
```

```
[0.04285714]
```

```
[0.05041209]
```

```
[0.1146978 ]
```

```
[0.37445055]
```

```
[0.78145604]
```

```
[0.96016484]
```

```
[0.82211538]
```

```
[0.72925824]
```

```
[0.63228022]
```

```
[0.67087912]
```

```
[0.7010989 ]]
```

Target horizon

```
[[0.71126374]]
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
```

```
BUFFER_SIZE = 150
```

```

train_univariate = tf.data.Dataset.from_tensor_slices((x_train_uni, y_train_uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()
val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni, y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()

```

The best weights are stored at model_path.

```
model_path = r'\\Chapter 6\\LSTM_Univariant_2.h5'
```

Define the LSTM model, as shown here:

```

lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(100, input_shape=x_train_uni.shape[-2:], return_sequences=True),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.LSTM(units=50, return_sequences=False),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units=1),
])

```

```
lstm_model.compile(optimizer='adam', loss='mse')
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when monitored loss starts to increase above patience.

Checkpointing saves model weights as it reached minimum loss.

```
EVALUATION_INTERVAL = 100
```

```
EPOCHS = 150
```

```
history = lstm_model.fit(train_univariate, epochs=EPOCHS, steps_per_epoch=EVALUATION_INTERVAL, validation_data=val_univariate, validation_steps=50, verbose=1, callbacks=[tf.keras.callbacks
```

```
.EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'),
tf.keras.callbacks.ModelCheckpoint(model_path,monitor='val_
loss', save_best_only=True, mode='min', verbose=0)])
```

Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 7s 66ms/step - loss: 0.0702 - val_loss: 0.0333
Epoch 2/150
100/100 [=====] - 2s 17ms/step - loss: 0.0355 - val_loss: 0.0279
Epoch 3/150
100/100 [=====] - 2s 17ms/step - loss: 0.0262 - val_loss: 0.0247

Epoch 78/150
100/100 [=====] - 2s 16ms/step - loss: 0.0092 - val_loss: 0.0073
Epoch 79/150
100/100 [=====] - 2s 16ms/step - loss: 0.0085 - val_loss: 0.0077
Epoch 00079: early stopping

The previous example shows that the model stopped within the 79th epoch instead of running for 150. Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

```
Model: "sequential_3"

```

Layer (type)	Output Shape	Param #
lstm_6 (LSTM)	(None, 48, 100)	40800
dropout_6 (Dropout)	(None, 48, 100)	0
lstm_7 (LSTM)	(None, 50)	30200
dropout_7 (Dropout)	(None, 50)	0
dense_3 (Dense)	(None, 1)	51

```
Total params: 71,051
Trainable params: 71,051
Non-trainable params: 0
```

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

In machine and deep learning, there are three scenarios.

- **Underfitting**, when the validation loss is less than the training loss
- **Overfitting**, when the validation loss is higher than the training loss
- **Good fit**, when the validation loss is equal to the training loss

Figure 6-3 depicts a line plot that helps us understand how a good model is able to generalize.

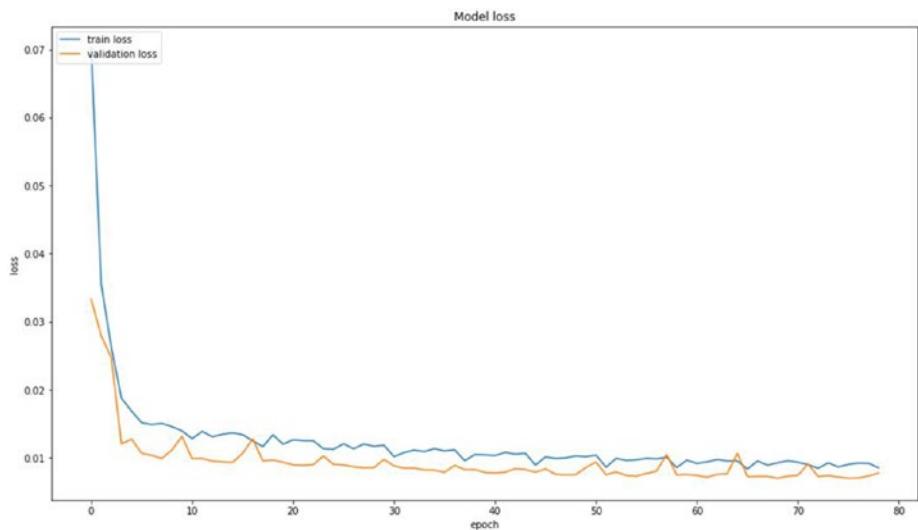


Figure 6-3. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, but our model predicts only one time step, so let's take the last 48 hours of data from the training and increment one prediction at a time and forecast the next values.

```

uni = df['traffic_volume']
validatehori = uni.tail(48)
validatehist = validatehori.values
result = []
# Define Forecast length here
window_len = 10
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))

for i in range(1, window_len+1):
    val_rescaled = val_rescaled.reshape((1, val_
rescaled.shape[0], 1))
    Predicted_results = Trained_model.predict(val_rescaled)
    print(f'predicted : {Predicted_results}')

```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

```
result.append(Predicted_results[0])
val_rescaled = np.append(val_rescaled[:,1:], [[Predicted_
results]])
print(val_rescaled)

predicted : [[0.6664802]]
[0.80830343 0.77960021 0.83000171 0.94020161 1. 0.91576969
0.81479583 0.67503844 0.54724073 0.46523151 0.40628737 0.54826585
0.23885187 0.09089356 0.03229113 0.00410046 0.01537673 0.06970784
0.17495302 0.29062019 0.46779429 0.57919016 0.63625491 0.69075688
0.73842474 0.72065607 0.73176149 0.72253545 0.75175124 0.77618315
0.72338971 0.51307022 0.44677943 0.388177 0.60157184 0.25474116
0.09994874 0.06167777 0.00495472 0. 0.01862293 0.07978814
0.14351615 0.28481121 0.43516146 0.5750897 0.63454639 0.66648018]
predicted : [[0.67511845]]
[0.77960021 0.83000171 0.94020161 1. 0.91576969 0.81479583
0.67503844 0.54724073 0.46523151 0.40628737 0.54826585 0.23885187
0.09089356 0.03229113 0.00410046 0.01537673 0.06970784 0.17495302
0.29062019 0.46779429 0.57919016 0.63625491 0.69075688 0.73842474
0.72065607 0.73176149 0.72253545 0.75175124 0.77618315 0.72338971
0.51307022 0.44677943 0.388177 0.60157184 0.25474116 0.09994874
0.06167777 0.00495472 0. 0.01862293 0.07978814 0.14351615
0.28481121 0.43516146 0.5750897 0.63454639 0.66648018 0.67511845]
predicted : [[0.6600107]]
```

Rescale the predicted values back to the original scale.

```
result_inv_trans = scaler_x.inverse_transform(result)
result_inv_trans

array([[4235.90851462],
       [4286.46826649],
       [4198.04260051],
       [4045.09277499],
       [3876.0522036 ],
       [3692.41856802],
       [3467.40279806],
       [3183.13737339],
       [2830.3494361 ],
       [2410.8878015 ]])
```

Define the time-series evaluation function.

```
from sklearn import metrics
def timeseries_evaluation_func(y_true, y_pred):
```

```

def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
print('Evaluation metric results:-')
print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end='
\n\n')

timeseries_evaluation_metrics_func(validate,result_inv_trans)

Evaluation metric results:-
MSE is : 569518.4022320593
MAE is : 539.3631616860628
RMSE is : 754.6644302152177
MAPE is : 62.643418764773706
R2 is : 0.6302600809174763

```

Plot the actual versus predicted values.

```

plt.plot( list(validate))
plt.plot( list(result_inv_trans))
plt.title("Actual vs Predicted")
plt.ylabel("Traffic volume")
plt.legend(('Actual','predicted'))
plt.show()

```

In perfect prediction, you would have the predicted values equal the actual values, so the line plot you see shows how much the prediction deviates from the actual value (the prediction error).

The line plot in Figure 6-4 depicts how the actual and predicted values change through time.

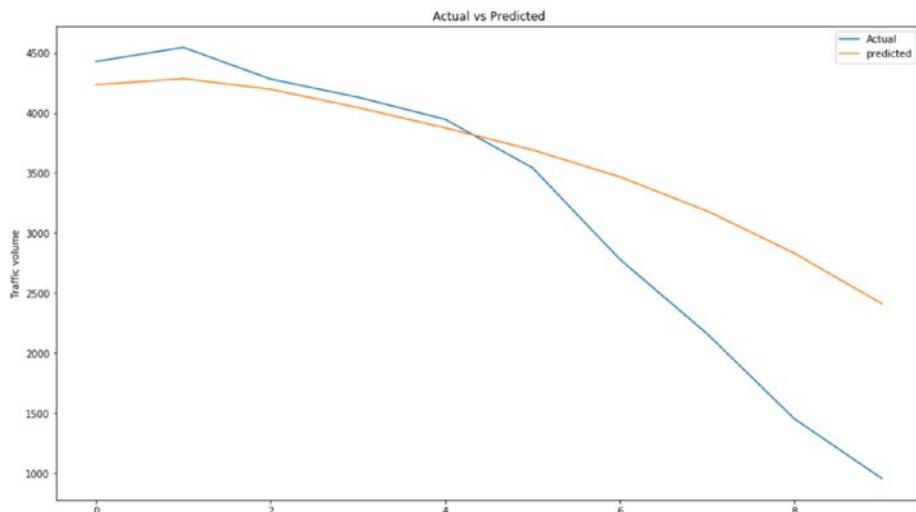


Figure 6-4. Representation of actual versus predicted values

LSTM Univariate Horizon Style in Action

In this section, let's use horizon-style data preparation and LSTM to solve univariate time-series problems.

Import the required libraries and load the CSV files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's use this data and forecast the traffic volume for the next ten hours.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

Let's take a sneak peek at the data by checking the five-point summary.

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour has different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Now train/test the split. Let's hold back ten hours of data (i.e., ten records), which we can use to validate the data after training on the past data.

```
validate = df['traffic_volume'].tail(10)
df.drop(df['traffic_volume'].tail(10).index,inplace=True)
```

```
uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()
```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
date_time
2012-10-02 09:00:00    5545
2012-10-02 10:00:00    4516
2012-10-02 11:00:00    4767
2012-10-02 12:00:00    5026
2012-10-02 13:00:00    4918
Name: traffic_volume, dtype: int64
```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))
```

Define a function to prepare univariate data suitable for a time series.

As we are doing horizon-style forecasts, let's allow the model to see/train on the past 48 hours of data and try forecasting the next ten hours of results; hence, use `horizon = 10`.

```
def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []

    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indicesx = range(i-window, i)
```

```
X.append(np.reshape(dataset[indicesx], (window, 1)))
indicesy = range(i, i+horizon)
y.append(dataset[indicesy])
return np.array(X), np.array(y)

univar_hist_window = 48
horizon = 10
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT,univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None,univar_hist_window,horizon)

print ('Single window of past history')
print (x_train_uni[0])
print ('\n Target horizon')
print (y_train_uni[0])

Single window of past history
[[0.76167582]
 [0.62032967]
 [0.65480769]
 [0.69038462]
 [0.67554945]
 [0.71167582]
 [0.76703297]
 [0.82623626]
 [0.79546703]
 [0.65521978]
 [0.48612637]
 [0.38241758]
 [0.32431319]
 [0.21002747]]
```

[0.13228022]
[0.06950549]
[0.04409341]
[0.0375]
[0.05041209]
[0.11181319]
[0.37335165]
[0.77925824]
[0.89436813]
[0.75151099]
[0.70013736]
[0.67129121]
[0.7331044]
[0.78186813]
[0.84299451]
[0.63502747]
[0.49326923]
[0.39807692]
[0.36222527]
[0.24409341]
[0.13942308]
[0.08214286]
[0.05068681]
[0.04285714]
[0.05041209]
[0.1146978]
[0.37445055]
[0.78145604]
[0.96016484]
[0.82211538]
[0.72925824]

```
[0.63228022]
```

```
[0.67087912]
```

```
[0.7010989 ]]
```

Target horizon

```
[[0.71126374]
```

```
[0.75563187]
```

```
[0.78475275]
```

```
[0.86428571]
```

```
[0.83200549]
```

```
[0.67403846]
```

```
[0.48118132]
```

```
[0.41717033]
```

```
[0.38763736]
```

```
[0.27362637]]
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
```

```
BUFFER_SIZE = 150
```

```
train_univariate = tf.data.Dataset.from_tensor_slices((x_train_uni, y_train_uni))
```

```
train_univariate = train_univariate.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()
```

```
val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni, y_val_uni))
```

```
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

The best weights are stored at `model_path`.

```
model_path = r'\\Chapter 6\\LSTM_Univariant_1.h5'
```

Define the LSTM model.

```
lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(100, input_shape=x_train_uni.shape[-2:],
    return_sequences=True),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.LSTM(units=50,return_sequences=False),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units=horizon),
])
lstm_model.compile(optimizer='adam', loss='mse')
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops the training when the monitored loss starts to increase above the patience.

Checkpointing saves the model weights as it reaches the minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = lstm_model.fit(train_univariate, epochs=EPOCHS,steps_per_epoch=EVALUATION_INTERVAL,validation_data=val_univariate,
validation_steps=50,verbose =1,callbacks =[tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'),tf.keras.callbacks.ModelCheckpoint(mo
del_path,monitor='val_loss', save_best_only=True, mode='min',
verbose=0)])
```

```
Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 8s 77ms/step - loss: 0.0859 - val_loss: 0.0484
Epoch 2/150
100/100 [=====] - 4s 40ms/step - loss: 0.0582 - val_loss: 0.0475
Epoch 3/150
```

```

Epoch 70/150
100/100 [=====] - 2s 18ms/step - loss: 0.0268 - val_loss: 0.0236
Epoch 71/150
100/100 [=====] - 2s 17ms/step - loss: 0.0266 - val_loss: 0.0257
Epoch 72/150
100/100 [=====] - 2s 16ms/step - loss: 0.0262 - val_loss: 0.0248
Epoch 00072: early stopping

```

The previous example shows that the model stopped within the 72nd epoch instead of running for 150. Load the best weights into the model.

```

Trained_model = tf.keras.models.load_model(model_path)
Trained_model.summary()

```

Check the model summary.

```

Model: "sequential_2"

```

Layer (type)	Output Shape	Param #
lstm_4 (LSTM)	(None, 48, 100)	40800
dropout_4 (Dropout)	(None, 48, 100)	0
lstm_5 (LSTM)	(None, 50)	30200
dropout_5 (Dropout)	(None, 50)	0
dense_2 (Dense)	(None, 10)	510

```

Total params: 71,510
Trainable params: 71,510
Non-trainable params: 0

```

Plot the loss and val_loss against the epoch.

```

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()

```

Figure 6-5 depicts a line plot that helps us to understand how a good model is able to generalize.

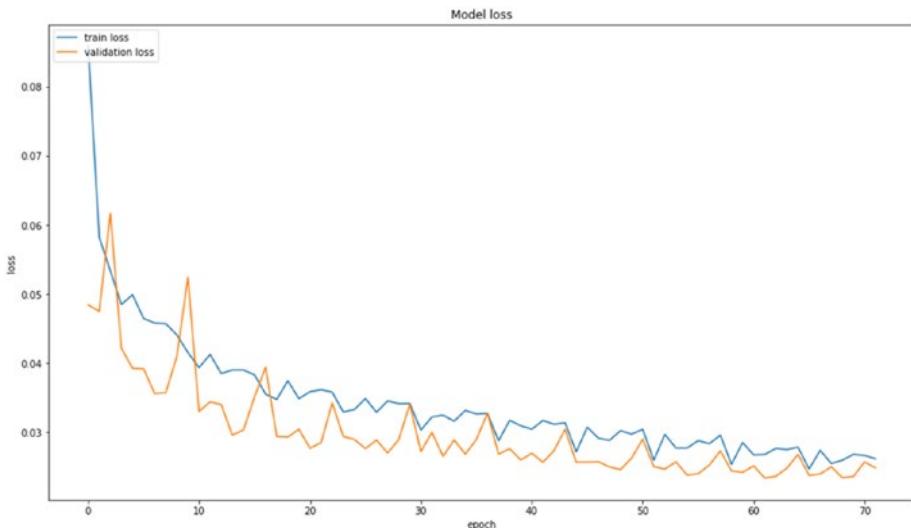


Figure 6-5. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
uni = df['traffic_volume']
validatehori = uni.tail(48)

validatehist = validatehori.values
scaler_val = preprocessing.MinMaxScaler()
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))

val_rescaled = val_rescaled.reshape((1, val_rescaled.shape[0], 1))

Predicted_results = Trained_model.predict(val_rescaled)

Predicted_results
```

```
array([[0.6135554 , 0.605981 , 0.60871387, 0.60912347, 0.5914895 ,
       0.56710166, 0.54609144, 0.5231304 , 0.47944632, 0.40487078]],  
      dtype=float32)
```

Rescale the predicted values back to the original scale, as shown here:

```
Predicted_inver_res = scaler_x.inverse_transform(Predicted_results)
```

```
Predicted_inver_res
```

```
array([[3926.1396, 3881.8066, 3897.8022, 3900.1997, 3796.988 , 3654.2458,  
      3531.2732, 3396.8823, 3141.1995, 2704.7087]], dtype=float32)
```

Define the time-series evaluation function, as shown here:

```
from sklearn import metrics  
  
def timeseries_evaluation_metrics_func(y_true, y_pred):  
  
    def mean_absolute_percentage_error(y_true, y_pred):  
        y_true, y_pred = np.array(y_true), np.array(y_pred)  
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100  
    print('Evaluation metric results:-')  
    print(f'MSE is : {metrics.mean_squared_error(y_true,  
                                                y_pred)}')  
    print(f'MAE is : {metrics.mean_absolute_error(y_true,  
                                                y_pred)}')  
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error  
                               (y_true, y_pred))}')  
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,  
                                                    y_pred)}')  
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}', end=  
          '\n\n')  
  
timeseries_evaluation_metrics_func(validate,Predicted_inver_res[0])
```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

```
Evaluation metric results:-  
MSE is : 895273.7881732046  
MAE is : 747.6373291015625  
RMSE is : 946.1890869024038  
MAPE is : 43.198849556965406  
R2 is : 0.4187747811158763
```

```
plt.plot( list(validate))  
plt.plot( list(Predicted_inver_res[0]))  
plt.title("Actual vs Predicted")  
plt.ylabel("Traffic volume")  
plt.legend(('Actual','predicted'))  
plt.show()
```

Figure 6-6 shows a line plot that depicts how the actual and predicted values change through time.

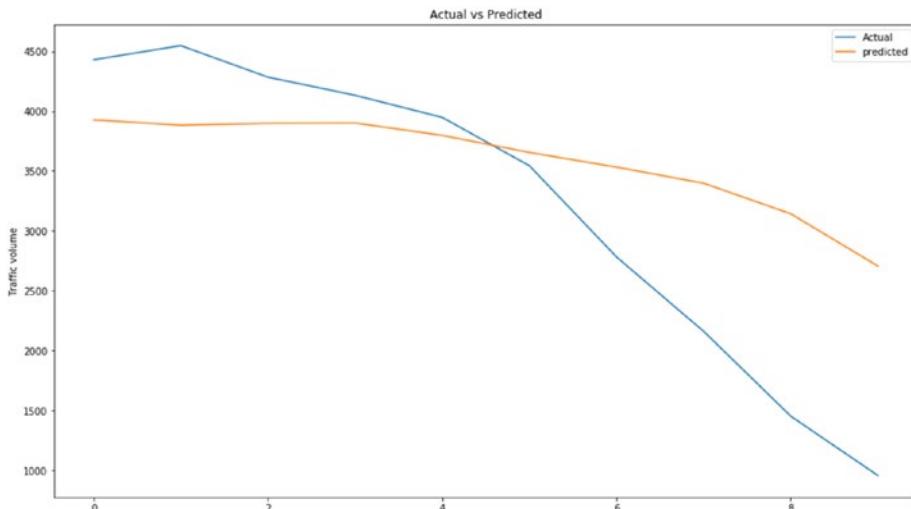


Figure 6-6. Representation of actual versus predicted values

Bidirectional LSTM Univariate Single-Step Style in Action

In this section, let's use single-step data preparation and bidirectional LSTM to solve univariate time-series problems.

Import the required libraries and load the CSV files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's load and take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour has different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Train/test the split. Let's hold back ten hours of data (i.e., ten records), which we can use to validate the data after training on the past data.

```
validate = df['traffic_volume'].tail(10)
df.drop(df['traffic_volume'].tail(10).index,inplace=True

uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()
```

```
date_time
2012-10-02 09:00:00    5545
2012-10-02 10:00:00    4516
2012-10-02 11:00:00    4767
2012-10-02 12:00:00    5026
2012-10-02 13:00:00    4918
Name: traffic_volume, dtype: int64
```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))
```

Define a function to prepare univariate data suitable for time-series problems.

```
def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []

    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indicesx = range(i-window, i)
        X.append(np.reshape(dataset[indicesx], (window, 1)))
        indicesy = range(i, i+horizon)
        y.append(dataset[indicesy])
    return np.array(X), np.array(y)
```

As we are doing single-step forecasting, let's allow the model to see/train on the past 48 hours of data and try to forecast the 49th hour; hence, use `horizon = 1`.

```
univar_hist_window = 48
horizon = 1
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT, univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None, univar_hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_univariate = tf.data.Dataset.from_tensor_slices((x_train_uni, y_train_uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni, y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

Define the bidirectional LSTM model.

```
Bi_lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(100,
    return_sequences=True), input_shape=x_train_uni.shape[-2:]),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(50)),
    tf.keras.layers.Dense(20, activation='softmax'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units=1),
])
Bi_lstm_model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 6\Bi_directional_LSTM_Univariant_2.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when the monitored loss starts to increase above patience.

Checkpointing saves the model weights as it reaches the minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = Bi_lstm_model.fit(train_univariate,
epochs=EPOCHS,steps_per_epoch=EVALUATION_INTERVAL,validation_
data=val_univariate, validation_steps=50,verbose =1,
callbacks =[tf.keras.callbacks
.EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'),tf.keras.callbacks.ModelCheckpoint(mod
el_path,monitor='val_loss', save_best_only=True, mode='min',
verbose=0)])]

Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 13s 132ms/step - loss: 0.0979 - val_loss: 0.0731
Epoch 2/150
100/100 [=====] - 3s 28ms/step - loss: 0.0663 - val_loss: 0.0373
Epoch 3/150
100/100 [=====] - 3s 28ms/step - loss: 0.0388 - val_loss: 0.0254

Epoch 85/150
100/100 [=====] - 3s 28ms/step - loss: 0.0115 - val_loss: 0.0084
Epoch 86/150
100/100 [=====] - 3s 28ms/step - loss: 0.0105 - val_loss: 0.0071
Epoch 00086: early stopping
```

The previous example shows the model stopped within 86 epochs instead of running for 150. Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

`Trained_model.summary()`

Model: "sequential"

Layer (type)	Output Shape	Param #
bidirectional (Bidirectional (None, 48, 200)		81600
bidirectional_1 (Bidirection (None, 100)		100400
dense (Dense)	(None, 20)	2020
dropout (Dropout)	(None, 20)	0
dense_1 (Dense)	(None, 1)	21
<hr/>		
Total params: 184,041		
Trainable params: 184,041		
Non-trainable params: 0		

Check the model summary.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 6-7 depicts a line plot that helps us understand how a good model is able to generalize.

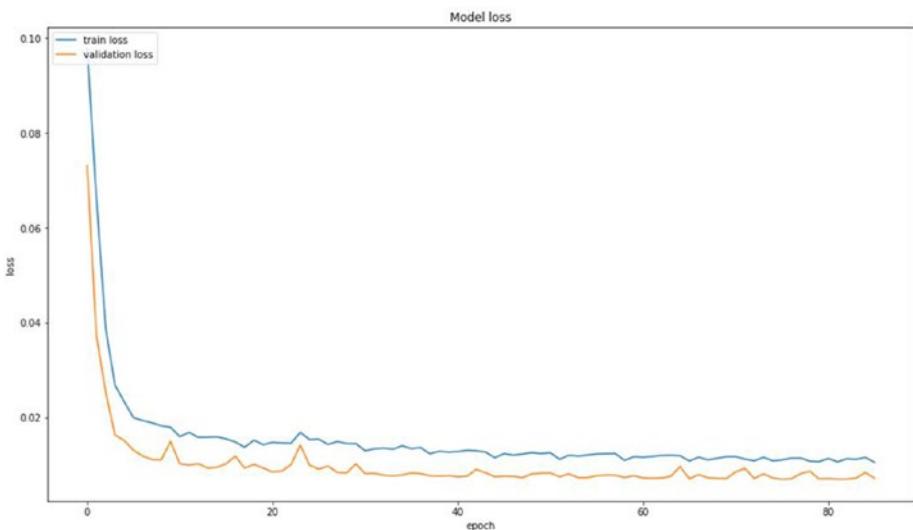


Figure 6-7. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, but our model predicts only one time step, so let's take the last 48 hours of data from the training and increment one prediction at a time and forecast the next values.

```

uni = df['traffic_volume']
validatehori = uni.tail(48)
validatehist = validatehori.values
result = []
# Define Forecast length here
window_len = 10
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))

for i in range(1, window_len+1):
    val_rescaled = val_rescaled.reshape((1, val_
        rescaled.shape[0], 1))
    Predicted_results = Trained_model.predict(val_rescaled)
    print(f'predicted : {Predicted_results}')

```

```

result.append(Predicted_results[0])
val_rescaled = np.append(val_rescaled[:,1:], [[Predicted_
results]])
print(val_rescaled)

predicted : [[0.65775627]]
[0.80830343 0.77960021 0.83000171 0.94020161 1.          0.91576969
 0.81479583 0.67503844 0.54724073 0.46523151 0.40628737 0.54826585
 0.23885187 0.09089356 0.03229113 0.00410046 0.01537673 0.06970784
 0.17495302 0.29062019 0.46779429 0.57919016 0.63625491 0.69075688
 0.73842474 0.72065607 0.73176149 0.72253545 0.75175124 0.77618315
 0.72338971 0.51307022 0.44677943 0.388177 0.60157184 0.25474116
 0.09994874 0.06167777 0.00495472 0.          0.01862293 0.07978814
 0.14351615 0.28481121 0.43516146 0.5750897 0.63454639 0.65775627]
predicted : [[0.66198182]]
[0.77960021 0.83000171 0.94020161 1.          0.91576969 0.81479583
 0.67503844 0.54724073 0.46523151 0.40628737 0.54826585 0.23885187
 0.09089356 0.03229113 0.00410046 0.01537673 0.06970784 0.17495302
 0.29062019 0.46779429 0.57919016 0.63625491 0.69075688 0.73842474
 0.72065607 0.73176149 0.72253545 0.75175124 0.77618315 0.72338971
 0.51307022 0.44677943 0.388177 0.60157184 0.25474116 0.09994874
 0.06167777 0.00495472 0.          0.01862293 0.07978814 0.14351615
 0.28481121 0.43516146 0.5750897 0.63454639 0.65775627 0.66198182]
predicted : [[0.65527222]]

```

Rescale the predicted values back to the original scale.

```
result_inv_trans = scaler_x.inverse_transform(result)
```

Define the time-series evaluation function.

```

from sklearn import metrics
def timeseries_evaluation_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')

```

```
print(f'MAE is : {metrics.mean_absolute_error(y_true,  
y_pred)}')  
print(f'RMSE is : {np.sqrt(metrics.mean_squared_error  
(y_true, y_pred))}')  
print(f'MAPE is : {mean_absolute_percentage_error(y_true,  
y_pred)}')  
print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=  
'\n\n')  
  
timeseries_evaluation_metrics_func(validate,result_inv_trans)  
  
Evaluation metric results:-  
MSE is : 875434.803348708  
MAE is : 712.1100398421288  
RMSE is : 935.6467299941297  
MAPE is : 64.99844083467826  
R2 is : 0.4316545486790321
```

Plot the actual versus predicted values.

```
plt.plot( list(validate))  
plt.plot( list(result_inv_trans))  
plt.title("Actual vs Predicted")  
plt.ylabel("Traffic volume")  
plt.legend(('Actual','predicted'))  
plt.show()
```

Figure 6-8 shows a line plot that depicts how the actual and predicted values change through time.

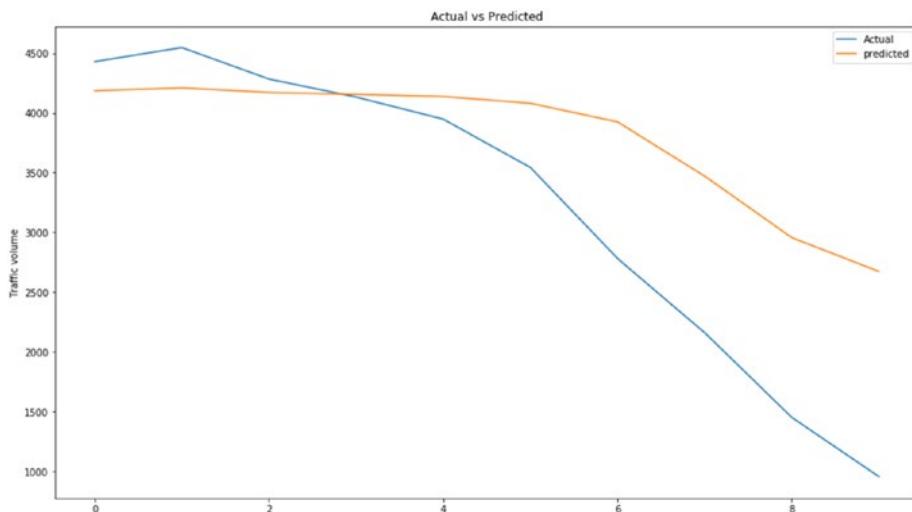


Figure 6-8. Representation of actual versus predicted values

Bidirectional LSTM Univariate Horizon Style in Action

In this section, let's use horizon-style data preparation and bidirectional LSTM to solve univariate times-series problems.

Import the required libraries and load the CSV files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

`df.describe()`

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour has different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Train/test the split. Let's hold back ten hours of data (i.e., ten records), which we can use to validate the data after training on the past data.

```
validate = df['traffic_volume'].tail(10)
df.drop(df['traffic_volume'].tail(10).index,inplace=True)

uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()
```

```

date_time
2012-10-02 09:00:00    5545
2012-10-02 10:00:00    4516
2012-10-02 11:00:00    4767
2012-10-02 12:00:00    5026
2012-10-02 13:00:00    4918
Name: traffic_volume, dtype: int64

```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```

uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))

```

Define a function to prepare univariate data suitable for a time series.

As we are doing horizon-style forecasts, let's allow the model to see/train on the past 48 hours of data and try forecast the next ten hours of results. Hence, use `horizon = 10`.

```

def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []

    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indicesx = range(i-window, i)
        X.append(np.reshape(dataset[indicesx], (window, 1)))
        indicesy = range(i, i+horizon)
        y.append(dataset[indicesy])
    return np.array(X), np.array(y)

univar_hist_window = 48
horizon = 10

```

```
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT, univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None, univar_hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_univariate = tf.data.Dataset.from_tensor_slices((x_train_uni, y_train_uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni, y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

Here is the bidirectional LSTM model:

```
Bi_lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(100,
    return_sequences=True),
        input_shape=x_train_uni.shape[-2:]),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(50)),
    tf.keras.layers.Dense(20, activation='softmax'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units=horizon),
])
Bi_lstm_model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 6\Bidirectional_LSTM_Univariant_1.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops the training when the monitored loss starts to increase above the patience.

Checkpointing saves the model weights as they reach the minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 100
history = Bi_lstm_model.fit(train_univariate,
epochs=EPOCHS,steps_per_epoch=EVALUATION_INTERVAL,validation_
data=val_univariate, validation_steps=50,verbose =1,
callbacks =[tf.keras.callbacks
.EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'),tf.keras.callbacks.ModelCheckpoint(mod
el_path,monitor='val_loss', save_best_only=True, mode='min',
verbose=0)])
```

Train for 100 steps, validate for 50 steps
Epoch 1/100
100/100 [=====] - 12s 123ms/step - loss: 0.1805 - val_loss: 0.1294
Epoch 2/100
100/100 [=====] - 5s 49ms/step - loss: 0.1019 - val_loss: 0.0849
Epoch 3/100
100/100 [=====] - 3s 29ms/step - loss: 0.0819 - val_loss: 0.0723

Epoch 78/100
100/100 [=====] - 3s 27ms/step - loss: 0.0274 - val_loss: 0.0262
Epoch 79/100
100/100 [=====] - 3s 27ms/step - loss: 0.0262 - val_loss: 0.0250
Epoch 00079: early stopping

The previous example shows that the model stopped within 79 epochs instead of running for 150. Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

```
Model: "sequential"
-----
Layer (type)          Output Shape       Param #
bidirectional (Bidirectional (None, 48, 200)      81600
bidirectional_1 (Bidirection (None, 100)           100400
dense (Dense)         (None, 20)            2020
dropout (Dropout)     (None, 20)             0
dense_1 (Dense)       (None, 10)             210
-----
Total params: 184,230
Trainable params: 184,230
Non-trainable params: 0
```

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 6-9 depicts a line plot that helps us understand how a good model is able to generalize.

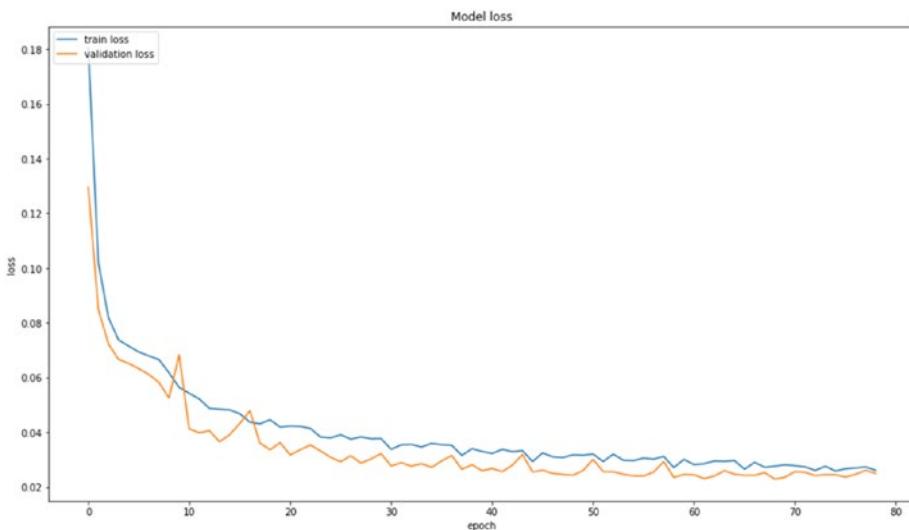


Figure 6-9. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
uni = df['traffic_volume']
validatehori = uni.tail(48)

validatehist = validatehori.values
scaler_val = preprocessing.StandardScaler()
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))

val_rescaled = val_rescaled.reshape((1, val_rescaled.shape[0], 1))

Predicted_results = Trained_model.predict(val_rescaled)
Predicted_results

array([[0.5834908 , 0.58690774, 0.59064263, 0.6029938 , 0.6163158 ,
       0.60863507, 0.57046133, 0.5115059 , 0.4549055 , 0.4061918 ]],
      dtype=float32)
```

Rescale the predicted values back to the original scale.

```
Predicted_inver_res = scaler_x.inverse_transform(Predicted_
results)
Predicted_inver_res

array([[3750.1716, 3770.171 , 3792.0312, 3864.3225, 3942.2961, 3897.341 ,
       3673.9102, 3328.844 , 2997.562 , 2712.4404]], dtype=float32)
```

Define the time-series evaluation function.

```
from sklearn import metrics
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=
'\n\n')

timeseries_evaluation_metrics_func(validate,Predicted_inver_
res[0])
```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

```
Evaluation metric results:-  
MSE is : 915543.7118907511  
MAE is : 794.2105224609375  
RMSE is : 956.8404840362635  
MAPE is : 43.781638221483625  
R2 is : 0.40561524153688866
```

Plot the actual versus predicted values.

```
plt.plot( list(validate))  
plt.plot( list(Predicted_inver_res[0]))  
plt.title("Actual vs Predicted")  
plt.ylabel("Traffic volume")  
plt.legend(('Actual','predicted'))  
plt.show()
```

Figure 6-10 shows a line plot that depicts how the actual and predicted values change through time.

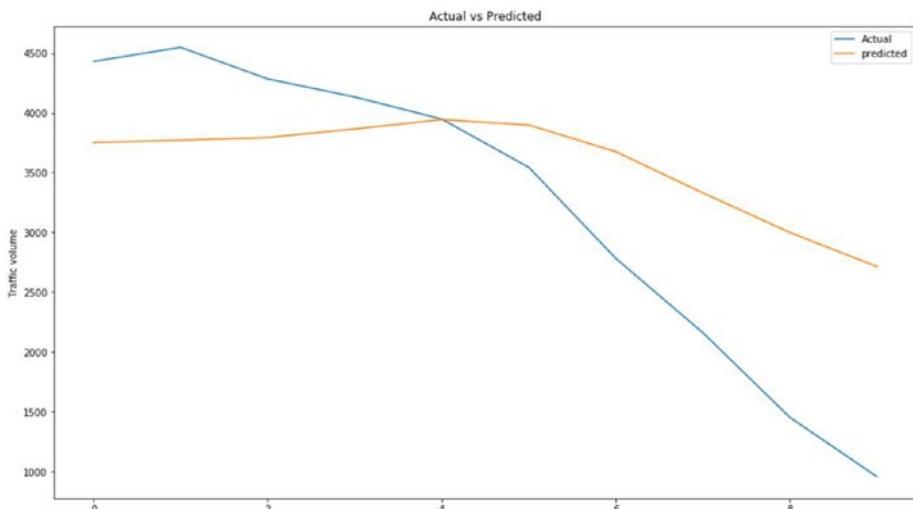


Figure 6-10. Representation of actual versus predicted values

GRU Univariate Single-Step Style in Action

In this section, let's use single-step data preparation and GRU to solve univariate time-series problems.

Import the required libraries and load the CSV files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's load and take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

df.describe()

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour contains different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Train/test the split. Let's hold back ten hours of data (i.e., ten records), which we can use to validate the data after training on the past data.

```
validate = df['traffic_volume'].tail(10)
df.drop(df['traffic_volume'].tail(10).index,inplace=True)
```

```
uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()
```

date_time	traffic_volume
2012-10-02 09:00:00	5545
2012-10-02 10:00:00	4516
2012-10-02 11:00:00	4767
2012-10-02 12:00:00	5026
2012-10-02 13:00:00	4918

Name: traffic_volume, dtype: int64

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))
```

Define a function to prepare univariate data suitable for a time series.

```
def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []

    start = start + window
```

```

if end is None:
    end = len(dataset) - horizon

for i in range(start, end):
    indicesx = range(i-window, i)
    X.append(np.reshape(dataset[indicesx], (window, 1)))
    indicesy = range(i,i+horizon)
    y.append(dataset[indicesy])
return np.array(X), np.array(y)

```

As we are doing single-step forecasting, let's allow the model to see/train on the past 48 hours of data and try to forecast the 49th hour; hence, use `horizon = 1`.

```

univar_hist_window = 48
horizon = 1
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT, univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None, univar_hist_window, horizon)

```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```

BATCH_SIZE = 256
BUFFER_SIZE = 150

train_univariate = tf.data.Dataset.from_tensor_slices((x_train_uni, y_train_uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

```

```
val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni,
y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

Define the GRU model.

```
GRU_model = tf.keras.models.Sequential([
    tf.keras.layers.GRU(100, input_shape=x_train_uni.shape
[-2:], return_sequences=True),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.GRU(units=50, return_sequences=False),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units=1),
])
GRU_model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 6\GRU_Univariant_2.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops the training when the monitored loss starts to increase above the patience.

Checkpointing saves the model weights as it reaches the minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = GRU_model.fit(train_univariate, epochs=EPOCHS, steps_
per_epoch=EVALUATION_INTERVAL, validation_data=val_univariate,
validation_steps=50, verbose=1, callbacks=[tf.keras.callbacks.
EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'), tf.keras.callbacks.ModelCheckpoint
(model_path, monitor='val_loss', save_best_only=True,
mode='min', verbose=0)])
```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

```
Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 7s 75ms/step - loss: 0.0445 - val_loss: 0.0200
Epoch 2/150
100/100 [=====] - 4s 41ms/step - loss: 0.0207 - val_loss: 0.0130
Epoch 3/150
100/100 [=====] - 2s 15ms/step - loss: 0.0173 - val_loss: 0.0131

Epoch 85/150
100/100 [=====] - 2s 15ms/step - loss: 0.0091 - val_loss: 0.0073
Epoch 86/150
100/100 [=====] - 2s 15ms/step - loss: 0.0081 - val_loss: 0.0075
Epoch 00086: early stopping
```

The previous example shows the model stopped within the 86th epoch instead of running for 150. Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

```
Model: "sequential"
-----
```

Layer (type)	Output Shape	Param #
gru (GRU)	(None, 48, 100)	30900
dropout (Dropout)	(None, 48, 100)	0
gru_1 (GRU)	(None, 50)	22800
dropout_1 (Dropout)	(None, 50)	0
dense (Dense)	(None, 1)	51

```
-----
```

Total params: 53,751
Trainable params: 53,751
Non-trainable params: 0

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
```

```
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 6-11 depicts a line plot that helps us to understand how a good model is able to generalize.

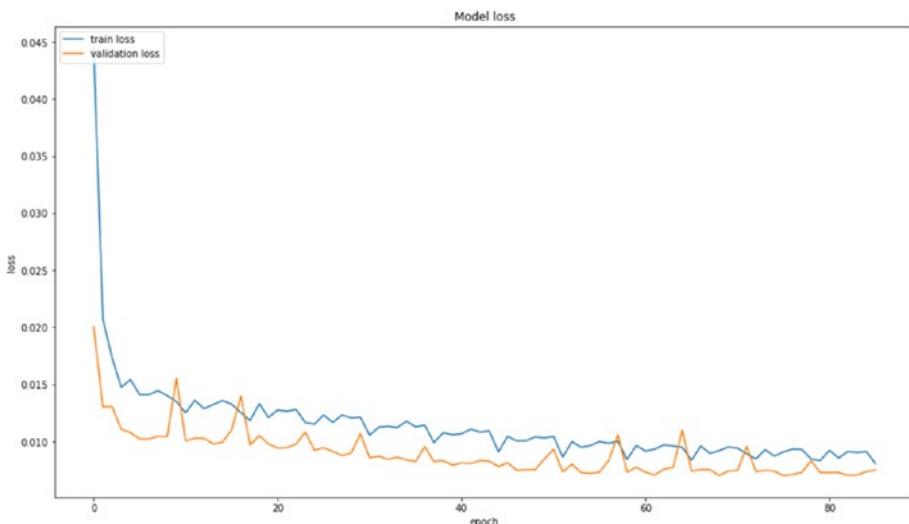


Figure 6-11. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, but our model predicts only one time step, so let's take the last 48 hours of data from the training and increment one prediction at a time and forecast the next values.

```
uni = df['traffic_volume']
validatehori = uni.tail(48)
validatehist = validatehori.values
```

```

result = []
# Define Forecast length here
window_len = 10
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))

for i in range(1, window_len+1):

    val_rescaled = val_rescaled.reshape((1, val_rescaled.shape[0], 1))
    Predicted_results = Trained_model.predict(val_rescaled)
    print(f'predicted : {Predicted_results}')
    result.append(Predicted_results[0])
    val_rescaled = np.append(val_rescaled[:,1:], [[Predicted_results]])
print(val_rescaled)

predicted : [[0.6352098]
[0.80830343 0.77960021 0.83000171 0.94020161 1. 0.91576969
0.81479583 0.67503844 0.54724073 0.46523151 0.40628737 0.54826585
0.23885187 0.09089356 0.03229113 0.00410046 0.01537673 0.06970784
0.17495302 0.29062019 0.46779429 0.57919016 0.63625491 0.69075688
0.73842474 0.72065607 0.73176149 0.72253545 0.75175124 0.77618315
0.72338971 0.51307022 0.44677943 0.388177 0.60157184 0.25474116
0.09994874 0.06167777 0.00495472 0. 0.01862293 0.07978814
0.14351615 0.28481121 0.43516146 0.5750897 0.63454639 0.6352098 ]
predicted : [[0.6275835]
[0.77960021 0.83000171 0.94020161 1. 0.91576969 0.81479583
0.67503844 0.54724073 0.46523151 0.40628737 0.54826585 0.23885187
0.09089356 0.03229113 0.00410046 0.01537673 0.06970784 0.17495302
0.29062019 0.46779429 0.57919016 0.63625491 0.69075688 0.73842474
0.72065607 0.73176149 0.72253545 0.75175124 0.77618315 0.72338971
0.51307022 0.44677943 0.388177 0.60157184 0.25474116 0.09994874
0.06167777 0.00495472 0. 0.01862293 0.07978814 0.14351615
0.28481121 0.43516146 0.5750897 0.63454639 0.6352098 0.6275835 ]
predicted : [[0.62403286]]

```

Rescale the predicted values back to the original scale.

```
result_inv_trans = scaler_x.inverse_transform(result)
```

Define the time-series evaluation function.

```

from sklearn import metrics
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end='
\n\n')

timeseries_evaluation_metrics_func(validate,result_inv_trans)

Evaluation metric results:-
MSE is : 1215598.608399074
MAE is : 824.993548476696
RMSE is : 1102.5418850996427
MAPE is : 63.50363024711859
R2 is : 0.21081508631715062

```

Plot the actual versus predicted values.

```

plt.plot( list(validate))
plt.plot( list(result_inv_trans))
plt.title("Actual vs Predicted")
plt.ylabel("Traffic volume")
plt.legend(('Actual','predicted'))
plt.show()

```

Figure 6-12 shows a line plot that depicts how the actual and predicted values change through time.

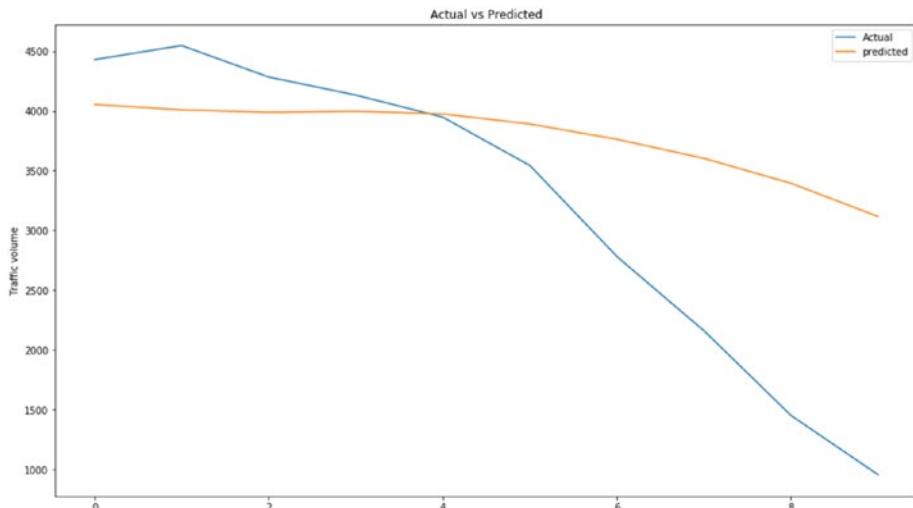


Figure 6-12. Representation of actual versus predicted values

GRU Univariate Horizon Style in Action

In this section, let's use horizon-style data preparation and GRU to solve univariate time-series problems.

Import the required libraries and load the CSV files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

Let's load and take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'Data\Metro_Interstate_Traffic_Volume.csv')  
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour contains different weather conditions.

```
df.drop_duplicates(subset=['date_time'],  
keep=False,inplace=True)
```

Train/test the split. Let's hold back ten hours of data (i.e., ten records), which we can use to validate the data after training on the past data.

```
validate = df['traffic_volume'].tail(10)  
df.drop(df['traffic_volume'].tail(10).index,inplace=True)
```

```

uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()

date_time
2012-10-02 09:00:00    5545
2012-10-02 10:00:00    4516
2012-10-02 11:00:00    4767
2012-10-02 12:00:00    5026
2012-10-02 13:00:00    4918
Name: traffic_volume, dtype: int64

```

Define a function to prepare univariate data suitable for a time series.

As we are doing horizon-style forecasting, let's allow the model to see/train on the past 48 hours of data and try to forecast the next ten hours of results; hence, use `horizon = 10`.

```

uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))

def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []

    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indicesx = range(i-window, i)
        X.append(np.reshape(dataset[indicesx], (window, 1)))
        indicesy = range(i, i+horizon)
        y.append(dataset[indicesy])
    return np.array(X), np.array(y)

```

```
univar_hist_window = 48
horizon = 10
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT, univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None, univar_hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_univariate = tf.data.Dataset.from_tensor_slices((x_train_uni, y_train_uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni, y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

Define the GRU model.

```
GRU_model = tf.keras.models.Sequential([
    tf.keras.layers.GRU(100, input_shape=x_train_uni.shape[-2:], return_sequences=True),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.GRU(units=50, return_sequences=False),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units=horizon),
])
GRU_model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 6\GRU_Univariant_1.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops the training when the monitored loss starts to increase above the patience.

Checkpointing saves the model weights as it reaches the minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = GRU_model.fit(train_univariate, epochs=EPOCHS, steps_per_epoch=EVALUATION_INTERVAL, validation_data=val_univariate, validation_steps=50, verbose=1, callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=10, verbose=1, mode='min'), tf.keras.callbacks.ModelCheckpoint(model_path, monitor='val_loss', save_best_only=True, mode='min', verbose=0)])
```

```
Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 2s 16ms/step - loss: 0.0277 - val_loss: 0.0240
Epoch 2/150
100/100 [=====] - 2s 15ms/step - loss: 0.0270 - val_loss: 0.0249
Epoch 3/150
100/100 [=====] - 1s 15ms/step - loss: 0.0278 - val_loss: 0.0267

Epoch 15/150
100/100 [=====] - 2s 16ms/step - loss: 0.0251 - val_loss: 0.0238
Epoch 16/150
100/100 [=====] - 2s 15ms/step - loss: 0.0250 - val_loss: 0.0244
Epoch 17/150
100/100 [=====] - 1s 15ms/step - loss: 0.0241 - val_loss: 0.0248
Epoch 00017: early stopping
```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

The previous example shows the model stopped within 17 epochs instead of running for 150.

Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
gru (GRU)	(None, 48, 100)	30900
dropout (Dropout)	(None, 48, 100)	0
gru_1 (GRU)	(None, 50)	22800
dropout_1 (Dropout)	(None, 50)	0
dense (Dense)	(None, 10)	510
<hr/>		
Total params: 54,210		
Trainable params: 54,210		
Non-trainable params: 0		

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 6-13 depicts a line plot that helps us understand how a good model is able to generalize.

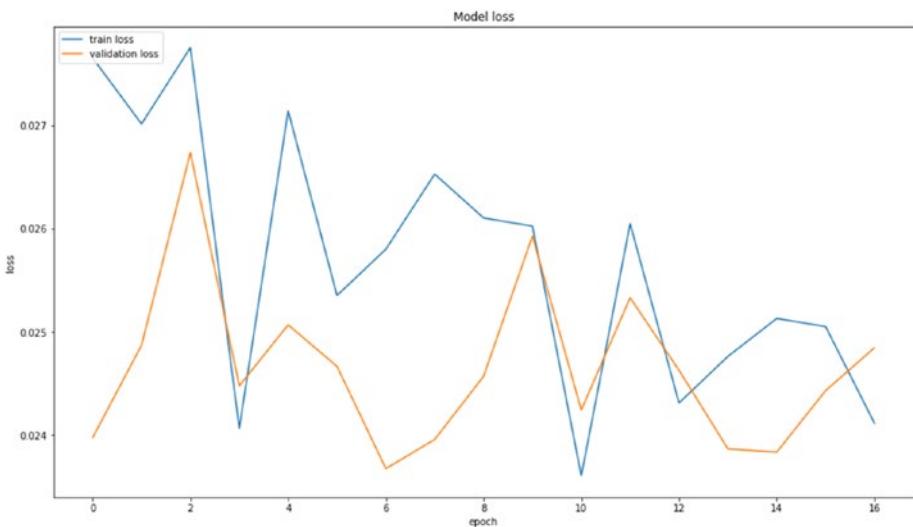


Figure 6-13. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
uni = df['traffic_volume']
validatehori = uni.tail(48)

validatehist = validatehori.values
scaler_val = preprocessing.MinMaxScaler()
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))
val_rescaled = val_rescaled.reshape((1, val_rescaled.shape[0], 1))
```

Rescale the predicted values back to the original scale.

```
Predicted_results = Trained_model.predict(val_rescaled)
```

```
Predicted_results
```

```
array([[0.60907274, 0.5980475 , 0.5873754 , 0.5756924 , 0.55768406,
       0.531145 , 0.50104624, 0.46705398, 0.4160539 , 0.34934813]],  
      dtype=float32)
```

```
Predicted_inver_res = scaler_x.inverse_transform(Predicted_results)
```

```
Predicted_inver_res
```

```
array([[3899.9028, 3835.3718, 3772.9082, 3704.5276, 3599.1248, 3443.7915,
       3267.6235, 3068.667 , 2770.1633, 2379.7346]], dtype=float32)
```

Define the time-series evaluation function.

```
from sklearn import metrics
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true, y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true, y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true, y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}', end=
        '\n\n')

timeseries_evaluation_metrics_func(validate,Predicted_inver_
res[0])

Evaluation metric results:-
MSE is : 619999.1210417807
MAE is : 676.7561767578125
RMSE is : 787.4002292619559
MAPE is : 36.159127199964715
R2 is : 0.5974872384337513
```

Plot the actual versus predicted values.

```
plt.plot( list(validate))
plt.plot( list(Predicted_inver_res[0]))
plt.title("Actual vs Predicted")
plt.ylabel("Traffic volume")
plt.legend(('Actual','predicted'))
plt.show()
```

Figure 6-14 shows a line plot that depicts how the actual and predicted values change through time.

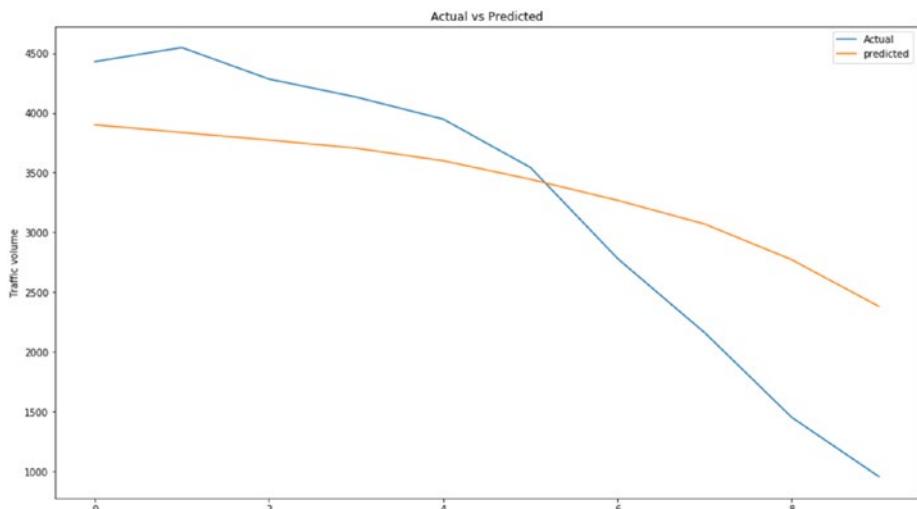


Figure 6-14. Representation of actual versus predicted values

Auto-encoder LSTM Univariate Single-Step Style in Action

In this section, let's use single-step data preparation and auto-encoder LSTM to solve univariate time-series problems.

Import the required libraries and load the CSV files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour contains different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Train/test the split. Let's hold back ten hours of data (i.e., ten records), which we can use to validate the data after training on the past data.

```
validate = df['traffic_volume'].tail(10)
df.drop(df['traffic_volume'].tail(10).index,inplace=True)
```

```
uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()
```

date_time	traffic_volume
2012-10-02 09:00:00	5545
2012-10-02 10:00:00	4516
2012-10-02 11:00:00	4767
2012-10-02 12:00:00	5026
2012-10-02 13:00:00	4918

Name: traffic_volume, dtype: int64

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))
```

Define a function to prepare univariate data suitable for a time series.

```
def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []
```

```

start = start + window
if end is None:
    end = len(dataset) - horizon

for i in range(start, end):
    indicesx = range(i-window, i)
    X.append(np.reshape(dataset[indicesx], (window, 1)))
    indicesy = range(i,i+horizon)
    y.append(dataset[indicesy])
return np.array(X), np.array(y)

```

As we are doing single-step forecasting, let's allow the model to see/
train on past 48 hours of data and try to forecast the 49th hour; hence, use
`horizon = 1.`

```

univar_hist_window = 48
horizon = 1
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT,univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None,univar_hist_window,horizon)

```

Prepare the training and validation time-series data using the `tf.data`
function, which is a much faster and more efficient way of feeding data to
the model.

```

BATCH_SIZE = 256
BUFFER_SIZE = 150
train_univariate = tf.data.Dataset.from_tensor_slices((x_train_
uni, y_train_uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_
SIZE).batch(BATCH_SIZE).repeat()

```

```
val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni,  
y_val_uni))  
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

Define the encode-decoder model.

```
ED_lstm_model = tf.keras.models.Sequential([  
    tf.keras.layers.LSTM(100, input_shape=x_train_  
        uni.shape[-2:], return_sequences=True),  
    tf.keras.layers.LSTM(units=50,return_sequences=True),  
    tf.keras.layers.LSTM(units=15),  
    tf.keras.layers.RepeatVector(y_train_uni.shape[1]),  
    tf.keras.layers.LSTM(units=100,return_sequences=True),  
    tf.keras.layers.LSTM(units=50,return_sequences=True),  
    tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(units=1))  
])  
ED_lstm_model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 6\LSTM_encoder_decoder_Univariant_2.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when the monitored loss starts to increase above the patience.

Checkpointing saves the model weights as they reach the minimum loss.

```
EVALUATION_INTERVAL = 100  
EPOCHS = 150
```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

```
history = ED_lstm_model.fit(train_univariate,
epochs=EPOCHS,steps_per_epoch=EVALUATION_INTERVAL,validation_
data=val_univariate, validation_steps=50,verbose =1,
callbacks =[tf.keras.callbacks
.EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'),tf.keras.callbacks.ModelCheckpoint(mod
el_path,monitor='val_loss', save_best_only=True, mode='min',
verbose=0)])
```

Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 14s 138ms/step - loss: 0.0976 - val_loss: 0.0750
Epoch 2/150
100/100 [=====] - 2s 24ms/step - loss: 0.0749 - val_loss: 0.0756
Epoch 3/150
100/100 [=====] - 4s 45ms/step - loss: 0.0712 - val_loss: 0.0640

Epoch 63/150
100/100 [=====] - 2s 24ms/step - loss: 0.0080 - val_loss: 0.0066
Epoch 64/150
100/100 [=====] - 2s 24ms/step - loss: 0.0079 - val_loss: 0.0075
Epoch 00064: early stopping

The previous example shows that the model stopped within 64 epochs instead of running for 150. Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
<hr/>		
lstm (LSTM)	(None, 48, 100)	40800
lstm_1 (LSTM)	(None, 48, 50)	30200
lstm_2 (LSTM)	(None, 15)	3960
repeat_vector (RepeatVector)	(None, 1, 15)	0
lstm_3 (LSTM)	(None, 1, 100)	46400
lstm_4 (LSTM)	(None, 1, 50)	30200
time_distributed (TimeDistri	(None, 1, 1)	51
<hr/>		
Total params:	151,611	
Trainable params:	151,611	
Non-trainable params:	0	

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 6-15 depicts a line plot that helps us understand how a good model is able to generalize.

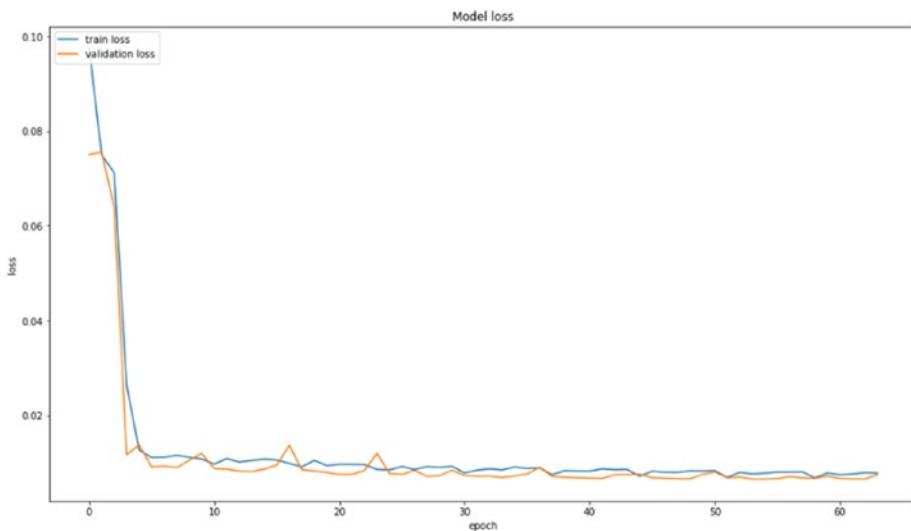


Figure 6-15. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, but our model predicts only one time step, so let's take the last 48 hours of data from the training and increment one prediction at a time and forecast the next values.

```

uni = df['traffic_volume']
validatehori = uni.tail(48)
validatehist = validatehori.values
result = []
# Define Forecast length here
window_len = 10
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))

for i in range(1, window_len+1):
    val_rescaled = val_rescaled.reshape((1, val_
        rescaled.shape[0], 1))
    Predicted_results = Trained_model.predict(val_rescaled)

```

```

print(f'predicted : {Predicted_results}')
result.append(Predicted_results[0])
val_rescaled = np.append(val_rescaled[:,1:], [[Predicted_
results]])
print(val_rescaled)

predicted : [[[0.6288946]]]
[0.80830343 0.77960021 0.83000171 0.94020161 1.          0.91576969
 0.81479583 0.67503844 0.54724073 0.46523151 0.40628737 0.54826585
 0.23885187 0.09089356 0.03229113 0.00410046 0.01537673 0.06970784
 0.17495302 0.29062019 0.46779429 0.57919016 0.63625491 0.69075688
 0.73842474 0.72065607 0.73176149 0.72253545 0.75175124 0.77618315
 0.72338971 0.51307022 0.44677943 0.388177 0.60157184 0.25474116
 0.09994874 0.06167777 0.00495472 0.          0.01862293 0.07978814
 0.14351615 0.28481121 0.43516146 0.5750897 0.63454639 0.62889463]
predicted : [[[0.6132983]]]
[0.77960021 0.83000171 0.94020161 1.          0.91576969 0.81479583
 0.67503844 0.54724073 0.46523151 0.40628737 0.54826585 0.23885187
 0.09089356 0.03229113 0.00410046 0.01537673 0.06970784 0.17495302
 0.29062019 0.46779429 0.57919016 0.63625491 0.69075688 0.73842474
 0.72065607 0.73176149 0.72253545 0.75175124 0.77618315 0.72338971
 0.51307022 0.44677943 0.388177 0.60157184 0.25474116 0.09994874
 0.06167777 0.00495472 0.          0.01862293 0.07978814 0.14351615
 0.28481121 0.43516146 0.5750897 0.63454639 0.62889463 0.6132983 ]
predicted : [[[0.6128562]]]

```

Rescale the predicted values back to the original scale.

```

result_inv_trans = scaler_x.inverse_transform(np.array(result).
reshape(-1,1))

```

Define the time-series evaluation function.

```

from sklearn import metrics
def timeseries_evaluation_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
    y_pred})')

```

```
print(f'MAE is : {metrics.mean_absolute_error(y_true,  
y_pred)}')  
print(f'RMSE is : {np.sqrt(metrics.mean_squared_error  
(y_true, y_pred))}')  
print(f'MAPE is : {mean_absolute_percentage_error(y_true,  
y_pred)}')  
print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=  
'\n\n')  
  
timeseries_evaluation_metrics_func(validate,result_inv_trans)  
  
Evaluation metric results:-  
MSE is : 319288.9216983557  
MAE is : 465.131591796875  
RMSE is : 565.056565043143  
MAPE is : 58.99066693644963  
R2 is : 0.7927128261176127
```

Plot the actual versus the predicted values.

```
plt.plot( list(validate))  
plt.plot( list(result_inv_trans))  
plt.title("Actual vs Predicted")  
plt.ylabel("Traffic volume")  
plt.legend(('Actual','predicted'))  
plt.show()
```

Figure 6-16 shows the line plot that depicts how actual and predicted values change through time.

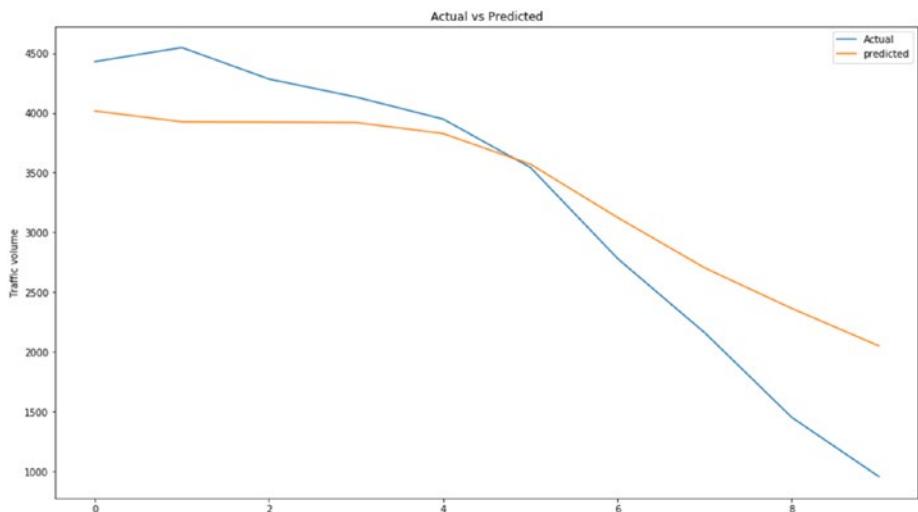


Figure 6-16. Representation of actual versus predicted values

Auto-encoder LSTM Univariate Horizon Style in Action

In this section, let's use horizon-style data preparation and auto-encoder LSTM to solve univariate time-series problems.

Import the required libraries and load the CSV files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour contains different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)

validate = df['traffic_volume'].tail(10)
df.drop(df['traffic_volume'].tail(10).index,inplace=True)

uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()
```

```

date_time
2012-10-02 09:00:00      5545
2012-10-02 10:00:00      4516
2012-10-02 11:00:00      4767
2012-10-02 12:00:00      5026
2012-10-02 13:00:00      4918
Name: traffic_volume, dtype: int64

```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```

uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))

```

Define a function to prepare univariate data suitable for a time series.

As we are doing horizon-style forecasting, let's allow the model to see/train on past 48 hours of data and try to forecast next the ten hours of results. Hence, use `horizon = 10`.

```

def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []

    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indicesx = range(i-window, i)
        X.append(np.reshape(dataset[indicesx], (window, 1)))
        indicesy = range(i, i+horizon)
        y.append(dataset[indicesy])
    return np.array(X), np.array(y)

univar_hist_window = 48
horizon = 10

```

```
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT, univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None, univar_hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150
train_univariate = tf.data.Dataset.from_tensor_slices((x_train_
uni, y_train_uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_
SIZE).batch(BATCH_SIZE).repeat()
val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni,
y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

Define the auto-encoder model.

```
ED_lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(100, input_shape=x_train_
uni.shape[-2:], return_sequences=True),
    tf.keras.layers.LSTM(units=50, return_sequences=True),
    tf.keras.layers.LSTM(units=15),
    tf.keras.layers.RepeatVector(y_train_uni.shape[1]),
    tf.keras.layers.LSTM(units=100, return_sequences=True),
    tf.keras.layers.LSTM(units=50, return_sequences=True),
    tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(units=1))
])
ED_lstm_model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 6\LSTM_encoder_decoder_Univariant_1.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when monitored loss starts to increase above the patience.

Checkpointing saves the model weights as they reach the minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = ED_lstm_model.fit(train_univariate,
epochs=EPOCHS,steps_per_epoch=EVALUATION_INTERVAL,validation_
data=val_univariate, validation_steps=50,verbose =1,
callbacks =[tf.keras.callbacks
.EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'),tf.keras.callbacks.ModelCheckpoint
(model_path,monitor='val_loss', save_best_only=True,
mode='min', verbose=0)])
```

Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 16s 163ms/step - loss: 0.0958 - val_loss: 0.0708
Epoch 2/150
100/100 [=====] - 3s 26ms/step - loss: 0.0605 - val_loss: 0.0572
Epoch 3/150
100/100 [=====] - 2s 25ms/step - loss: 0.0537 - val_loss: 0.0752

Epoch 70/150
100/100 [=====] - 3s 25ms/step - loss: 0.0224 - val_loss: 0.0231
Epoch 71/150
100/100 [=====] - 3s 25ms/step - loss: 0.0223 - val_loss: 0.0257
Epoch 72/150
100/100 [=====] - 3s 25ms/step - loss: 0.0218 - val_loss: 0.0296
Epoch 00072: early stopping

The previous example shows that the model stopped within 72 epochs instead of running for 150.

Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
lstm (LSTM)	(None, 48, 100)	40800
lstm_1 (LSTM)	(None, 48, 50)	30200
lstm_2 (LSTM)	(None, 15)	3960
repeat_vector (RepeatVector)	(None, 10, 15)	0
lstm_3 (LSTM)	(None, 10, 100)	46400
lstm_4 (LSTM)	(None, 10, 50)	30200
time_distributed (TimeDistr)	(None, 10, 1)	51
<hr/>		
Total params:	151,611	
Trainable params:	151,611	
Non-trainable params:	0	

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 6-17 depicts a line plot that helps us understand how a good model is able to generalize.

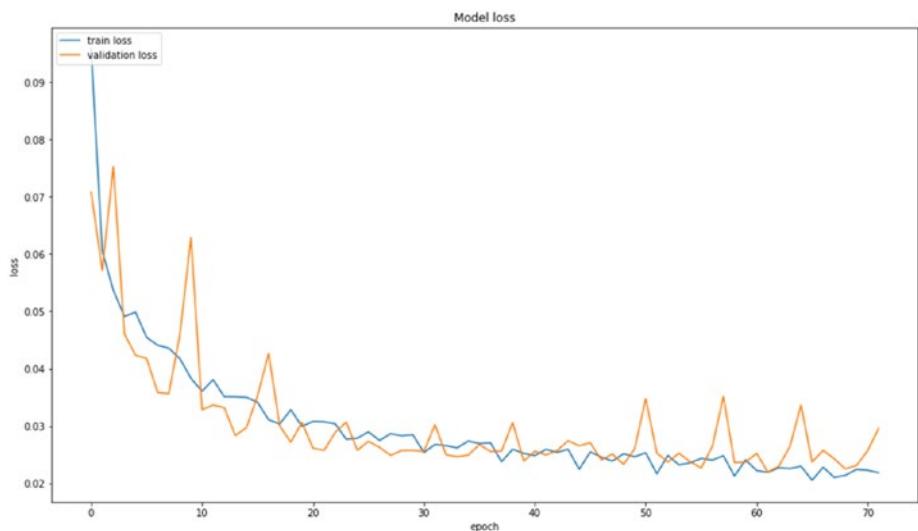


Figure 6-17. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
uni = df['traffic_volume']
validatehori = uni.tail(48)

validatehist = validatehori.values
scaler_val = preprocessing.MinMaxScaler()
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))
val_rescaled = val_rescaled.reshape((1, val_rescaled.shape[0], 1))

Predicted_results = Trained_model.predict(val_rescaled)

Predicted_results
```

```
array([[[0.60239756],
       [0.59967595],
       [0.5991256 ],
       [0.5909505 ],
       [0.58760625],
       [0.5718232 ],
       [0.5370325 ],
       [0.48754027],
       [0.43514606],
       [0.3951911 ]]], dtype=float32)
```

Rescale the predicted values back to the original scale.

```
Predicted_inver_res = scaler_x.inverse_transform(Predicted_results[0])
```

Predicted_inver_res

```
array([[3860.8328],
       [3844.9033],
       [3841.6821],
       [3793.8333],
       [3774.2593],
       [3681.881 ],
       [3478.251 ],
       [3188.5732],
       [2881.91  ],
       [2648.0535]], dtype=float32)
```

Define the time-series evaluation function.

```
from sklearn import metrics
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true, y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true, y_pred)}')
```

```
print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_true, y_pred))}')
print(f'MAPE is : {mean_absolute_percentage_error(y_true, y_pred)}')
print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=
'\n\n')

timeseries_evaluation_metrics_func(validate,Predicted_inver_res)

Evaluation metric results:-
MSE is : 764036.241718322
MAE is : 721.4157958984375
RMSE is : 874.0916666564909
MAPE is : 59.59204541363798
R2 is : 0.5039761716532893
```

Plot the actual versus the predicted values.

```
plt.plot( list(validate))
plt.plot( list(Predicted_inver_res))
plt.title("Actual vs Predicted")
plt.ylabel("Traffic volume")
plt.legend(('Actual','predicted'))
plt.show()
```

Figure 6-18 shows a line plot that depicts how actual and predicted values change through time.

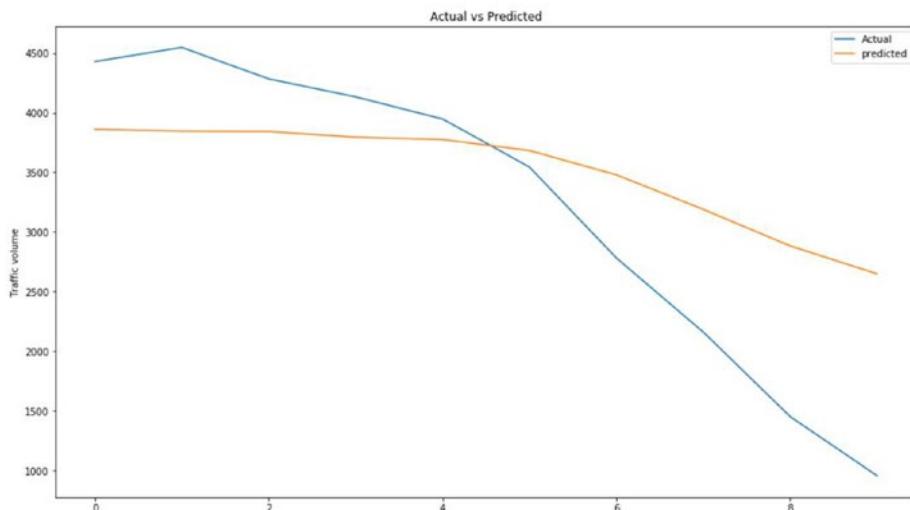


Figure 6-18. Representation of actual versus predicted values

CNN Univariate Single-Step Style in Action

In this section, let's use single-step data preparation and CNNs to solve univariate time-series problems.

Import the required libraries and load the CSV files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import MaxPool1D
from tensorflow.keras.layers import Dropout
```

```
tf.random.set_seed(123)
np.random.seed(123)
```

Let's take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour contains different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Train/test the split. Let's hold back ten hours of data (i.e., ten records), which we can use to validate after training on the past data.

```
validate = df['traffic_volume'].tail(10)
df.drop(df['traffic_volume'].tail(10).index,inplace=True)
```

```
uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()

date_time
2012-10-02 09:00:00    5545
2012-10-02 10:00:00    4516
2012-10-02 11:00:00    4767
2012-10-02 12:00:00    5026
2012-10-02 13:00:00    4918
Name: traffic_volume, dtype: int64
```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))
```

Define a function to prepare univariate data suitable for a time series.

```
def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []

    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indicesx = range(i-window, i)
        X.append(np.reshape(dataset[indicesx], (window, 1)))
        indicesy = range(i, i+horizon)
        y.append(dataset[indicesy])
    return np.array(X), np.array(y)
```

As we are doing single-step forecasting, let's allow the model to see/train on past 48 hours of data and try to forecast the 49th hour. Hence, use `horizon = 1`.

```
univar_hist_window = 48
horizon = 1
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT, univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None, univar_hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_univariate = tf.data.Dataset.from_tensor_slices((x_train_uni, y_train_uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni, y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

Define the CNN model.

```
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(x_train_uni.shape[1], x_train_uni.shape[2])))
model.add(MaxPool1D(pool_size=2))
model.add(Dropout(0.2))
```

```
model.add(Flatten())
model.add(Dense(30, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

The best weights stored at `model_path`.

```
model_path = r'\Chapter 6\CNN_Univarient_2.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when the monitored loss starts to increase above the patience.

Checkpointing saves the model weights as they reach the minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = model.fit(train_univariate, epochs=EPOCHS, steps_per_epoch=EVALUATION_INTERVAL, validation_data=val_univariate, validation_steps=50, verbose=1, callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=10, verbose=1, mode='min'), tf.keras.callbacks.ModelCheckpoint(model_path, monitor='val_loss', save_best_only=True, mode='min', verbose=0)])
```

Train for 100 steps, validate for 50 steps

```
Epoch 1/150
100/100 [=====] - 1s 14ms/step - loss: 0.0517 - val_loss: 0.0228
Epoch 2/150
100/100 [=====] - 1s 7ms/step - loss: 0.0277 - val_loss: 0.0166
Epoch 3/150
100/100 [=====] - 1s 7ms/step - loss: 0.0242 - val_loss: 0.0167
```

```

Epoch 44/150
100/100 [=====] - 1s 7ms/step - loss: 0.0137 - val_loss: 0.0097
Epoch 45/150
100/100 [=====] - 1s 7ms/step - loss: 0.0118 - val_loss: 0.0089
Epoch 00045: early stopping

```

The previous example shows that the model stopped within 45 epochs instead of running for 150. Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

```

Model: "sequential_1"
-----
Layer (type)          Output Shape       Param #
-----
conv1d_1 (Conv1D)     (None, 46, 64)      256
-----
max_pooling1d_1 (MaxPooling1D) (None, 23, 64) 0
-----
dropout_2 (Dropout)   (None, 23, 64)      0
-----
flatten_1 (Flatten)   (None, 1472)        0
-----
dense_2 (Dense)       (None, 30)          44190
-----
dropout_3 (Dropout)   (None, 30)          0
-----
dense_3 (Dense)       (None, 1)           31
-----
Total params: 44,477
Trainable params: 44,477
Non-trainable params: 0

```

Plot the loss and val_loss against the epoch.

```

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')

```

```
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 6-19 depicts a line plot that helps us understand how a good model is able to generalize.

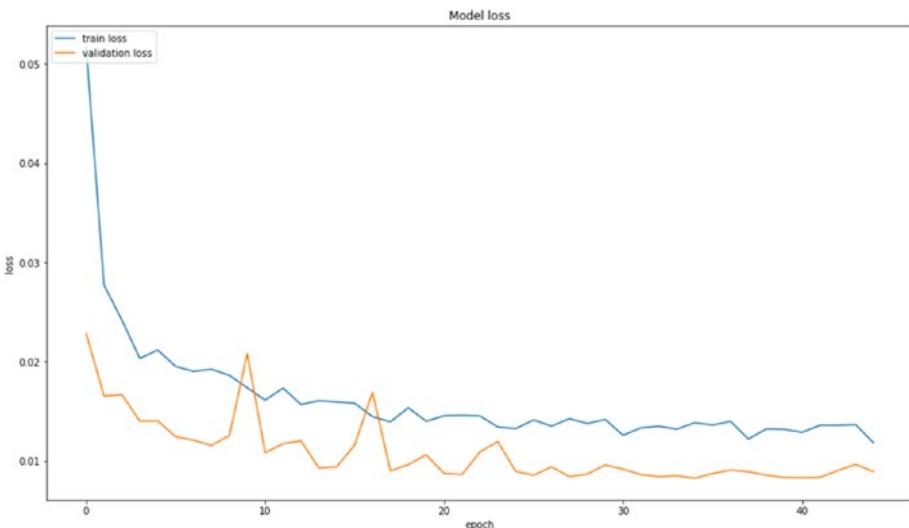


Figure 6-19. Representation of training and validation loss as the number of epoch increases

We need to forecast the next ten steps, but our model predicts only one time step, so let's take the last 48 hours of data from the training and increment one prediction at a time and forecast the next values.

```
uni = df['traffic_volume']
validatehori = uni.tail(48)
validatehist = validatehori.values
result = []
# Define Forecast length here
window_len = 10
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))
```

```

for i in range(1, window_len+1):

    val_rescaled = val_rescaled.reshape((1, val_
    rescaled.shape[0], 1))
    Predicted_results = Trained_model.predict(val_rescaled)
    print(f'predicted : {Predicted_results}')
    result.append(Predicted_results[0])
    val_rescaled = np.append(val_rescaled[:,1:], [[Predicted_
    results]], axis=0)
print(val_rescaled)

predicted : [[0.6318447]]
[0.80830343 0.77960021 0.83000171 0.94020161 1.          0.91576969
 0.81479583 0.67503844 0.54724073 0.46523151 0.40628737 0.54826585
 0.23885187 0.09089356 0.03229113 0.00410046 0.01537673 0.06970784
 0.17495302 0.29062019 0.46779429 0.57919016 0.63625491 0.69075688
 0.73842474 0.72065607 0.73176149 0.72253545 0.75175124 0.77618315
 0.72338971 0.51307022 0.44677943 0.388177 0.60157184 0.25474116
 0.09994874 0.06167777 0.00495472 0.          0.01862293 0.07978814
 0.14351615 0.28481121 0.43516146 0.5750897 0.63454639 0.6318447 ]
predicted : [[0.63646483]]
[0.77960021 0.83000171 0.94020161 1.          0.91576969 0.81479583
 0.67503844 0.54724073 0.46523151 0.40628737 0.54826585 0.23885187
 0.09089356 0.03229113 0.00410046 0.01537673 0.06970784 0.17495302
 0.29062019 0.46779429 0.57919016 0.63625491 0.69075688 0.73842474
 0.72065607 0.73176149 0.72253545 0.75175124 0.77618315 0.72338971
 0.51307022 0.44677943 0.388177 0.60157184 0.25474116 0.09994874
 0.06167777 0.00495472 0.          0.01862293 0.07978814 0.14351615
 0.28481121 0.43516146 0.5750897 0.63454639 0.6318447 0.63646483]
predicted : [[0.636981]]

```

Rescale the predicted values back to the original scale.

```
result_inv_trans = scaler_x.inverse_transform(result)
```

Define the time-series evaluation function.

```

from sklearn import metrics
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

```

```
print('Evaluation metric results:-')
print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=
'\n\n')

timeseries_evaluation_metrics_func(validate,result_inv_trans)

Evaluation metric results:-
MSE is : 1261474.818491242
MAE is : 860.1430524706841
RMSE is : 1123.1539602793741
MAPE is : 65.10055031256107
R2 is : 0.1810315601996233
```

Plot the actual versus predicted values.

```
plt.plot( list(validate))
plt.plot( list(result_inv_trans))
plt.title("Actual vs Predicted")
plt.ylabel("Traffic volume")
plt.legend(('Actual','predicted'))
plt.show()
```

Figure 6-20 shows a line plot that depicts how actual and predicted values change through time.

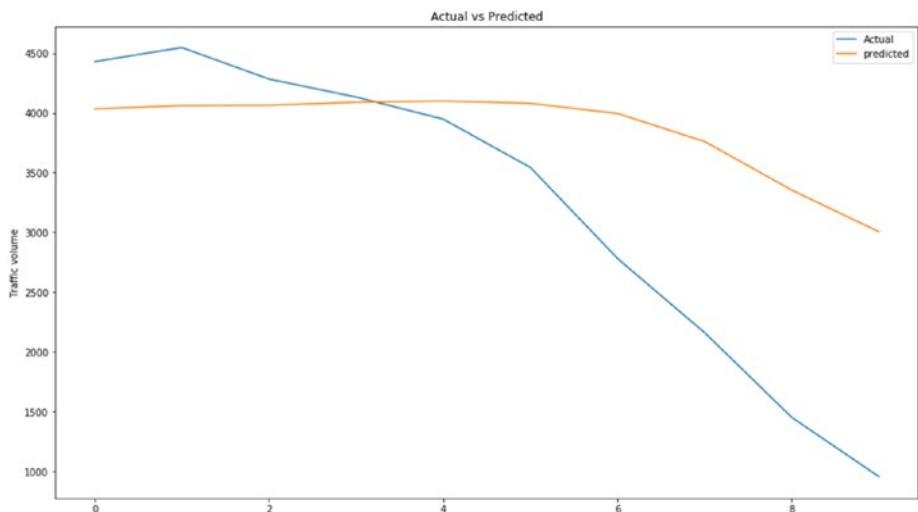


Figure 6-20. Representation of actual versus predicted values

CNN Univariate Horizon Style in Action

In this section, let's use horizon-style data preparation and CNNs to solve univariate time-series problems.

Import the required libraries and load the CSV files.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import MaxPool1D
from tensorflow.keras.layers import Dropout
```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

```
tf.random.set_seed(123)  
np.random.seed(123)
```

Let's load and take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')  
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour contains different weather conditions.

```
df.drop_duplicates(subset=['date_time'],  
keep=False,inplace=True)
```

Train/test the split. Let's hold back ten hours of data (i.e., ten records), which we can use to validate the data after training on the past data.

```
validate = df['traffic_volume'].tail(10)  
df.drop(df['traffic_volume'].tail(10).index,inplace=True)
```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
uni_data = df['traffic_volume']
uni_data.index = df['date_time']
uni_data.head()

date_time
2012-10-02 09:00:00    5545
2012-10-02 10:00:00    4516
2012-10-02 11:00:00    4767
2012-10-02 12:00:00    5026
2012-10-02 13:00:00    4918
Name: traffic_volume, dtype: int64

uni_data = uni_data.values
scaler_x = preprocessing.MinMaxScaler()
x_rescaled = scaler_x.fit_transform(uni_data.reshape(-1, 1))
```

Define a function to prepare univariate data suitable for a time series.

As we are doing horizon-style forecasts, let's allow the model to see/train on the past 48 hours of data and try to forecast the next ten hours of results. Hence, use `horizon = 10`.

```
def custom_ts_univariate_data_prep(dataset, start, end, window,
horizon):
    X = []
    y = []

    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indicesx = range(i-window, i)
        X.append(np.reshape(dataset[indicesx], (window, 1)))
        indicesy = range(i, i+horizon)
        y.append(dataset[indicesy])
    return np.array(X), np.array(y)
```

```
univar_hist_window = 48
horizon = 10
TRAIN_SPLIT = 30000
x_train_uni, y_train_uni = custom_ts_univariate_data_prep
(x_rescaled, 0, TRAIN_SPLIT, univar_hist_window, horizon)
x_val_uni, y_val_uni = custom_ts_univariate_data_prep
(x_rescaled, TRAIN_SPLIT, None, univar_hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_univariate = tf.data.Dataset.from_tensor_slices((x_train_
uni, y_train_uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_-
SIZE).batch(BATCH_SIZE).repeat()

val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni,
y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

Define the CNN model.

```
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(x_train_uni.shape[1], x_train_uni.shape[2])))
model.add(MaxPool1D(pool_size=2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(30, activation='relu'))
model.add(Dropout(0.2))
```

```
model.add(Dense(horizon))
model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 6\CNN_Univariant_1.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when the monitored loss starts to increase above the patience.

Checkpointing saves the model weights as they reach the minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = model.fit(train_univariate, epochs=EPOCHS, steps_per_epoch=EVALUATION_INTERVAL, validation_data=val_univariate, validation_steps=50, verbose=1, callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=10, verbose=1, mode='min'), tf.keras.callbacks.ModelCheckpoint(model_path, monitor='val_loss', save_best_only=True, mode='min', verbose=0)])
```

```
Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 4s 36ms/step - loss: 0.0982 - val_loss: 0.0565
Epoch 2/150
100/100 [=====] - 3s 27ms/step - loss: 0.0690 - val_loss: 0.0511
Epoch 3/150
100/100 [=====] - 1s 7ms/step - loss: 0.0602 - val_loss: 0.0582

Epoch 65/150
100/100 [=====] - 1s 7ms/step - loss: 0.0343 - val_loss: 0.0317
Epoch 66/150
100/100 [=====] - 1s 7ms/step - loss: 0.0307 - val_loss: 0.0269
Epoch 00066: early stopping
```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

The previous example shows the model stopped within 66 epochs instead of running for 150.

Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

```
Model: "sequential"
=====
Layer (type)          Output Shape         Param #
=====
conv1d (Conv1D)        (None, 46, 64)      256
max_pooling1d (MaxPooling1D) (None, 23, 64)    0
dropout (Dropout)      (None, 23, 64)      0
flatten (Flatten)      (None, 1472)        0
dense (Dense)          (None, 30)          44190
dropout_1 (Dropout)    (None, 30)          0
dense_1 (Dense)        (None, 10)          310
=====
Total params: 44,756
Trainable params: 44,756
Non-trainable params: 0
```

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 6-21 depicts a line plot that helps us understand how a good model is able to generalize.

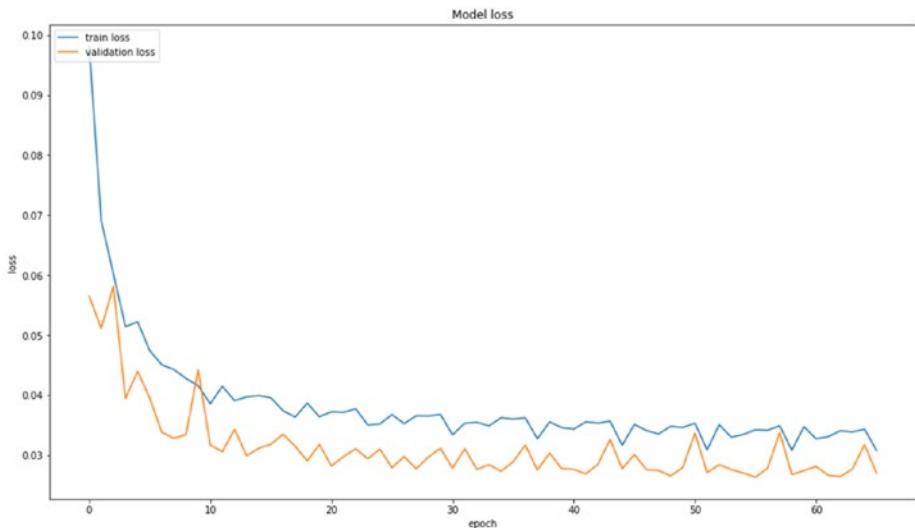


Figure 6-21. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
uni = df['traffic_volume']
validatehori = uni.tail(48)

validatehist = validatehori.values
scaler_val = preprocessing.MinMaxScaler()
val_rescaled = scaler_x.fit_transform(validatehist.reshape(-1, 1))

val_rescaled = val_rescaled.reshape((1, val_rescaled.shape[0], 1))

Predicted_results = Trained_model.predict(val_rescaled)
Predicted_results
```

CHAPTER 6 BLEEDING-EDGE TECHNIQUES FOR UNIVARIATE TIME SERIES

```
array([[0.5948421 , 0.59792274, 0.60076654, 0.60694414, 0.6117787 ,  
       0.60536027, 0.5763178 , 0.5251734 , 0.4643437 , 0.40647548]],  
      dtype=float32)
```

Rescale the predicted values back to the original scale.

```
Predicted_inver_res = scaler_x.inverse_transform(Predicted_  
results)
```

Define the time-series evaluation function.

```
from sklearn import metrics  
  
def timeseries_evaluation_func(y_true, y_pred):  
  
    def mean_absolute_percentage_error(y_true, y_pred):  
        y_true, y_pred = np.array(y_true), np.array(y_pred)  
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100  
    print('Evaluation metric results:-')  
    print(f'MSE is : {metrics.mean_squared_error(y_true,  
y_pred)}')  
    print(f'MAE is : {metrics.mean_absolute_error(y_true,  
y_pred)}')  
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error  
(y_true, y_pred))}')  
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,  
y_pred)}')  
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=  
'\n\n')  
  
timeseries_evaluation_func(validate,Predicted_inver_  
res[0])
```

```
Evaluation metric results:-  
MSE is : 933069.0783508897  
MAE is : 790.73828125  
RMSE is : 965.9550084506471  
MAPE is : 44.2008803804563  
R2 is : 0.39423750984030537
```

Plot the actual versus predicted values.

```
plt.plot( list(validate))  
plt.plot( list(Predicted_inver_res[0]))  
plt.title("Actual vs Predicted")  
plt.ylabel("Traffic volume")  
plt.legend(('Actual','predicted'))  
plt.show()
```

Figure 6-22 shows a line plot that depicts how actual and predicted values change through time.

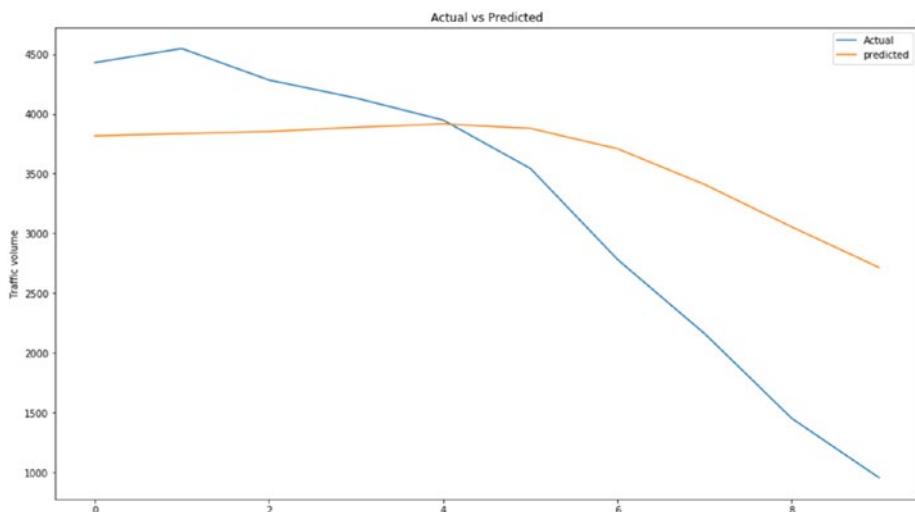


Figure 6-22. Representation of actual versus predicted values

Summary

In this chapter, you learned how to use single-step and horizon-style data preparation for a time series and solve univariate time-series problems using LSTM, bidirectional LSTM, GRU, auto-encoders, and CNNs. In the next chapter, you will learn how to solve multivariate time-series problems using bleeding-edge techniques.

CHAPTER 7

Bleeding-Edge Techniques for Multivariate Time Series

In the previous chapter, you learned how to perform univariate time-series forecasting. In this chapter, you will look at single-step and horizon-style time-series forecasting data preparation and solve some multivariate time-series problems. A multivariate time series is a method where more than one variable is time dependent, and we use these variables to try to estimate a target variable.

LSTM Multivariate Horizon Style in Action

In this section, let's use horizon-style data preparation and LSTM to solve multivariate time-series problems.

CHAPTER 7 BLEEDING-EDGE TECHNIQUES FOR MULTIVARIATE TIME SERIES

Import the required libraries and load the CSV data.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's load and take a sneak peek at data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour has different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Apply label encoding on the categorical features.

```
holiday_le = preprocessing.LabelEncoder()
df['holiday_le'] = holiday_le.fit_transform(df['holiday'])
weather_main_le = preprocessing.LabelEncoder()
df['weather_main_le'] = weather_main_le.fit_
transform(df['weather_main'])
weather_description_le = preprocessing.LabelEncoder()
df['weather_description_le'] = weather_description_le.fit_
transform(df['weather_description'])
```

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible, and portable. Let's use it to check feature importance. Figure 7-1 shows the bar plot representation.

```
from numpy import loadtxt
from xgboost import XGBRegressor
from matplotlib import pyplot

model = XGBRegressor()
model.fit(df[['temp', 'rain_1h', 'snow_1h', 'clouds_all',
'holiday_le','weather_main_le', 'weather_description_le']],
df[['traffic_volume']])

(pd.Series(model.feature_importances_, index=df[['temp',
'rain_1h', 'snow_1h', 'clouds_all', 'holiday_le',
'weather_main_le', 'weather_description_le']].columns)
.nlargest(7)
.plot(kind='barh'))
```

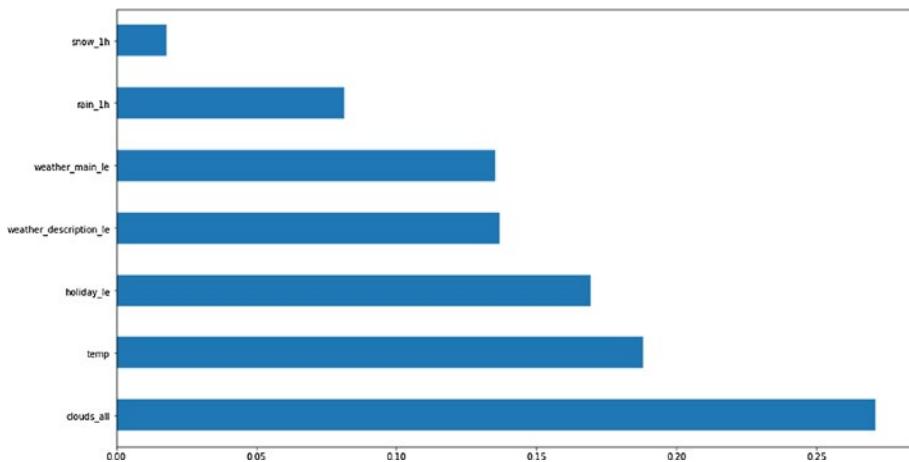


Figure 7-1. Representing feature importance using a bar plot

Let's take all features into model training. Define a function to prepare multivariate data so that it is suitable for a time series.

```
def custom_ts_multi_data_prep(dataset, target, start, end,
window, horizon):
    X = []
    y = []
    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indices = range(i-window, i)
        X.append(dataset[indices])

        indicey = range(i+1, i+1+horizon)
        y.append(target[indicey])
    return np.array(X), np.array(y)
```

Train/test the split, and let's hold back ten hours of data (i.e., ten records), which we can use to validate the results after training on past data.

```
validate = df[['rain_1h','temp', 'snow_1h', 'clouds_all',
'holiday_le','weather_main_le', 'weather_description_le',
'traffic_volume']].tail(10)
df.drop(df.tail(10).index,inplace=True)
```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
x_scaler = preprocessing.MinMaxScaler()
y_scaler = preprocessing.MinMaxScaler()
dataX = x_scaler.fit_transform(df[['rain_1h','temp', 'snow_1h',
'clouds_all', 'holiday_le','weather_main_le', 'weather_
description_le','traffic_volume']])
dataY = y_scaler.fit_transform(df[['traffic_volume']])
```

As we are doing horizon-style forecasting, let's allow the model to see/train on the past 48 hours of data and try to forecast the next ten hours of results. Hence, use `horizon = 10`.

```
hist_window = 48
horizon = 10
TRAIN_SPLIT = 30000
x_train_multi, y_train_multi = custom_ts_multi_data_prep(
    dataX, dataY, 0, TRAIN_SPLIT, hist_window, horizon)
x_val_multi, y_val_multi= custom_ts_multi_data_prep(
    dataX, dataY, TRAIN_SPLIT, None, hist_window, horizon)

print ('Single window of past history')
print(x_train_multi[0])
print ('\n Target horizon')
```

```
print (y_train_multi[0])  
  
Single window of past history  
[[0.         0.92972555 0.         0.4         0.63636364 0.1  
  0.64864865 0.76167582]  
 [0.         0.93320863 0.         0.75        0.63636364 0.1  
  0.05405405 0.62032967]  
 [0.         0.93391815 0.         0.9         0.63636364 0.1  
  0.51351351 0.65480769]  
 [0.         0.93569194 0.         0.9         0.63636364 0.1  
  0.51351351 0.69038462]  
 [0.         0.93894927 0.         0.75        0.63636364 0.1  
  0.05405405 0.67554945]  
 [0.         0.94081981 0.         0.01        0.63636364 0.  
  0.72972973 0.71167582]  
 [0.         0.94549618 0.         0.01        0.63636364 0.  
 [0.         0.90847228 0.         0.01        0.63636364 0.  
  0.72972973 0.1146978 ]  
 [0.         0.907279   0.         0.01        0.63636364 0.  
  0.72972973 0.37445055]  
 ...  
 [0.         0.90540846 0.         0.01        0.63636364 0.  
  0.72972973 0.78145604]  
 [0.         0.90486019 0.         0.01        0.63636364 0.  
  0.72972973 0.96016484]  
 [0.         0.90902054 0.         0.01        0.63636364 0.  
  0.72972973 0.82211538]  
 [0.         0.91908279 0.         0.01        0.63636364 0.  
  0.72972973 0.72925824]  
 [0.         0.93262812 0.         0.01        0.63636364 0.  
  0.72972973 0.63228022]  
 [0.         0.94027155 0.         0.01        0.63636364 0.]
```

```
0.72972973 0.67087912]
[0.           0.95130132 0.           0.01           0.63636364 0.
0.72972973 0.7010989 ]]
```

Target horizon

```
[0.75563187]
[0.78475275]
[0.86428571]
[0.83200549]
[0.67403846]
[0.48118132]
[0.41717033]
[0.38763736]
[0.27362637]
[0.16016484]]
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_data_multi = tf.data.Dataset.from_tensor_slices((x_train_multi,
                                                       y_train_multi))
train_data_multi = train_data_multi.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()

val_data_multi = tf.data.Dataset.from_tensor_slices((x_val_multi,
                                                       y_val_multi))
val_data_multi = val_data_multi.batch(BATCH_SIZE).repeat()
```

Define the LSTM model.

```
lstm_multi = tf.keras.models.Sequential()
lstm_multi.add(tf.keras.layers.LSTM(150, input_shape=x_train_
multi.shape[-2:], return_sequences=True))
lstm_multi.add(tf.keras.layers.Dropout(0.2)),
lstm_multi.add(tf.keras.layers.LSTM(units=100, return_
sequences=False)),
lstm_multi.add(tf.keras.layers.Dropout(0.2)),
lstm_multi.add(tf.keras.layers.Dense(units=horizon)),
lstm_multi.compile(optimizer='adam', loss='mse')
```

The best weights are stored at model_path.

```
model_path = r'\Chapter 7\LSTM_Multivariate.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when the monitored loss starts to increase above the desired limit.

Checkpointing saves the model weights as they reach minimum loss.

```
EVALUATION_INTERVAL = 150
EPOCHS = 100
history = lstm_multi.fit(train_data_multi, epochs=EPOCHS, steps_
per_epoch=EVALUATION_INTERVAL, validation_data=val_data_multi,
validation_steps=50, verbose=1, callbacks=[tf.keras.callbacks.
EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'), tf.keras.callbacks.ModelCheckpoint(mod_
el_path, monitor='val_loss', save_best_only=True, mode='min',
verbose=0)])
```

```
Train for 150 steps, validate for 50 steps
Epoch 1/100
150/150 [=====] - 9s 63ms/step - loss: 0.0719 - val_loss: 0.0477
Epoch 2/100
150/150 [=====] - 3s 22ms/step - loss: 0.0540 - val_loss: 0.0594
Epoch 3/100
150/150 [=====] - 4s 27ms/step - loss: 0.0498 - val_loss: 0.0420
Epoch 48/100
150/150 [=====] - 3s 22ms/step - loss: 0.0232 - val_loss: 0.0353
Epoch 49/100
150/150 [=====] - 3s 22ms/step - loss: 0.0223 - val_loss: 0.0293
Epoch 00049: early stopping
```

The previous example shows that the model stopped within 49 epochs instead of running for 150.

Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 7-2 depicts a line plot that helps us understand how a good model is able to generalize.

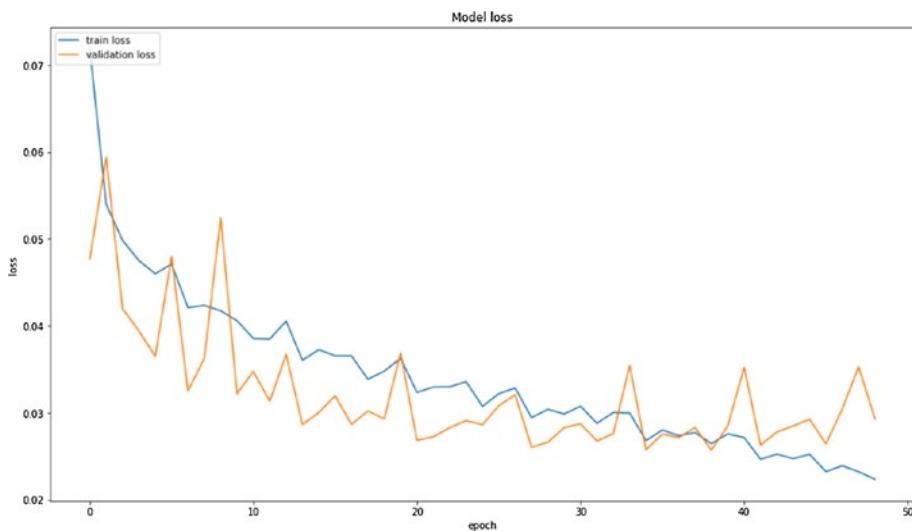


Figure 7-2. Representation of training and validation loss as the number of epochs increases

Check the model summary.

`Trained_model.summary()`

Model: "sequential_1"

Layer (type)	Output Shape	Param #
<hr/>		
lstm_2 (LSTM)	(None, 48, 150)	95400
dropout_2 (Dropout)	(None, 48, 150)	0
<hr/>		
lstm_3 (LSTM)	(None, 100)	100400
dropout_3 (Dropout)	(None, 100)	0
<hr/>		
dense_1 (Dense)	(None, 10)	1010
<hr/>		
Total params: 196,810		
Trainable params: 196,810		
Non-trainable params: 0		

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
data_val = x_scaler.fit_transform(df[['rain_1h','temp',
'snow_1h', 'clouds_all', 'holiday_le', 'weather_main_le',
'weather_description_le','traffic_volume']].tail(48))

val_rescaled = data_val.reshape(1, data_val.shape[0], data_
val.shape[1])

Predicted_results = Trained_model.predict(val_rescaled)

Predicted_results

array([[0.6491108 , 0.6378182 , 0.6240657 , 0.60300267, 0.57390654,
       0.54723495, 0.5352416 , 0.498307 , 0.42924818, 0.3614945 ]],  
      dtype=float32)
```

Rescale the predicted values back to the original scale.

```
Predicted_results_Inv_trans = y_scaler.inverse_
transform(Predicted_results)

Predicted_results_Inv_trans

array([[4725.5264, 4643.3164, 4543.198 , 4389.8594, 4178.0396, 3983.8704,
       3896.5588, 3627.6748, 3124.9268, 2631.68 ]], dtype=float32)
```

Define the time-series evaluation function.

```
from sklearn import metrics
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
```

```
print(f'MAE is : {metrics.mean_absolute_error(y_true,  
y_pred)}')  
print(f'RMSE is : {np.sqrt(metrics.mean_squared_error  
(y_true, y_pred))}')  
print(f'MAPE is : {mean_absolute_percentage_error(y_true,  
y_pred)}')  
print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=  
'\n\n')  
  
timeseries_evaluation_metrics_func(validate['traffic_  
volume'],Predicted_results_Inv_trans[0])  
  
Evaluation metric results:-  
MSE is : 950061.2068371713  
MAE is : 751.9650634765625  
RMSE is : 974.7108324201447  
MAPE is : 43.89349128541913  
R2 is : 0.3832059642626128
```

Plot the actual versus predicted values.

```
plt.plot( list(validate['traffic_volume']))  
plt.plot( list(Predicted_results_Inv_trans[0]))  
plt.title("Actual vs Predicted")  
plt.ylabel("Traffic volume")  
plt.legend(('Actual','predicted'))  
plt.show()
```

Figure 7-3 shows a line plot that depicts how the actual and predicted values change through time.

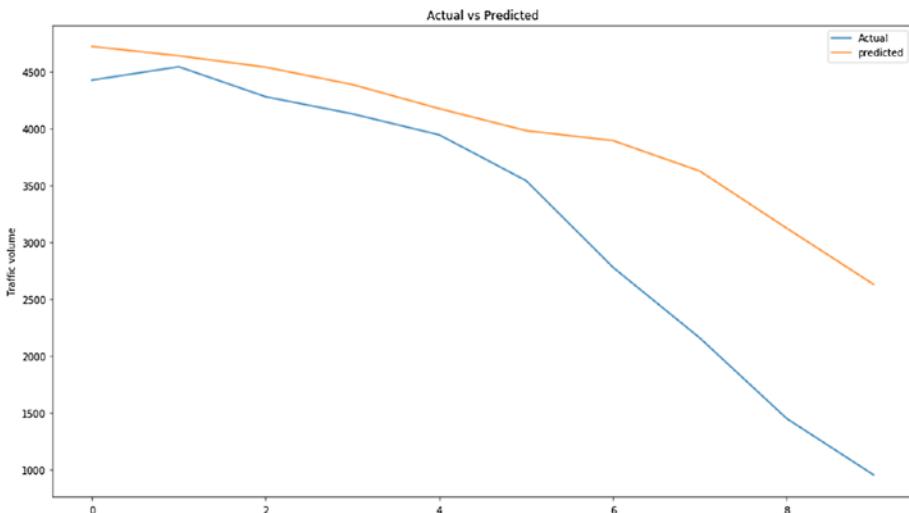


Figure 7-3. Representation of actual versus predicted values

Bidirectional LSTM Multivariate Horizon Style in Action

In this section, let's use horizon-style data preparation and bidirectional LSTM to solve multivariate time-series problems.

Import the required libraries and load the CSV data.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

CHAPTER 7 BLEEDING-EDGE TECHNIQUES FOR MULTIVARIATE TIME SERIES

Let's load and take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'Data\Metro_Interstate_Traffic_Volume.csv')  
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour has different weather conditions.

```
df.drop_duplicates(subset=['date_time'],  
keep=False,inplace=True)
```

Apply label encoding on the categorical features, and convert to numeric.

```
holiday_le = preprocessing.LabelEncoder()  
df['holiday_le'] = holiday_le.fit_transform(df['holiday'])
```

```
weather_main_le = preprocessing.LabelEncoder()
df['weather_main_le'] = weather_main_le.fit_
transform(df['weather_main'])
weather_description_le = preprocessing.LabelEncoder()
df['weather_description_le'] = weather_description_le.fit_
transform(df['weather_description'])
```

Train/test the split, and let's hold back ten hours of data (i.e., ten records), which we can use to validate the results after training on past data.

```
validate = df[['rain_1h','temp', 'snow_1h', 'clouds_all',
'holiday_le','weather_main_le', 'weather_description_le',
'traffic_volume']].tail(10)
df.drop(df.tail(10).index,inplace=True)
```

Define a function to prepare multivariate data suitable for a time series.

```
def custom_ts_multi_data_prep(dataset, target, start, end,
window, horizon):
    X = []
    y = []
    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indices = range(i-window, i)
        X.append(dataset[indices])

        indicey = range(i+1, i+1+horizon)
        y.append(target[indicey])
    return np.array(X), np.array(y)
```

```
validate = df[['rain_1h','temp', 'snow_1h', 'clouds_all',
'holiday_le','weather_main_le', 'weather_description_le',
'traffic_volume']].tail(10)
df.drop(df.tail(10).index,inplace=True)
```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
x_scaler = preprocessing.MinMaxScaler()
y_scaler = preprocessing.MinMaxScaler()
dataX = x_scaler.fit_transform(df[['rain_1h','temp', 'snow_1h',
'clouds_all', 'holiday_le','weather_main_le', 'weather_
description_le','traffic_volume']])
dataY = y_scaler.fit_transform(df[['traffic_volume']])
```

As we are doing horizon-style forecasting, let's allow the model to see/train on the past 48 hours of data and try to forecast the next ten hours of results. Hence, use `horizon = 10`.

```
hist_window = 48
horizon = 10
TRAIN_SPLIT = 30000
x_train_multi, y_train_multi = custom_ts_multi_data_prep(
    dataX, dataY, 0, TRAIN_SPLIT, hist_window, horizon)
x_val_multi, y_val_multi = custom_ts_multi_data_prep(
    dataX, dataY, TRAIN_SPLIT, None, hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_data_multi = tf.data.Dataset.from_tensor_slices((x_train_
multi, y_train_multi))
train_data_multi = train_data_multi.cache().shuffle(BUFFER_
SIZE).batch(BATCH_SIZE).repeat()

val_data_multi = tf.data.Dataset.from_tensor_slices((x_val_
multi, y_val_multi))
val_data_multi = val_data_multi.batch(BATCH_SIZE).repeat()
```

Define the bidirectional LSTM model.

```
Bi_lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(150,
    return_sequences=True), input_shape=x_train_multi.shape[-2:]),
    tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(50)),
    tf.keras.layers.Dense(20, activation='tanh'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units=horizon),
])
Bi_lstm_model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 7\Bidirectional_LSTM_Multivariate.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when the monitored loss starts to increase above the desired limit.

Checkpointing saves the model weights as they reach minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
```

CHAPTER 7 BLEEDING-EDGE TECHNIQUES FOR MULTIVARIATE TIME SERIES

```
history = Bi_lstm_model.fit(train_data_multi, epochs=EPOCHS,
    steps_per_epoch=EVALUATION_INTERVAL, validation_data=val_data_
    multi, validation_steps=50, verbose=1, callbacks=[tf.keras.
    callbacks.EarlyStopping(monitor='val_loss', min_delta=0,
    patience=10, verbose=1, mode='min'), tf.keras.callbacks.
    ModelCheckpoint(model_path, monitor='val_loss', save_best_only
    =True, mode='min', verbose=0)])
```

```
Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 14s 140ms/step - loss: 0.1014 - val_loss: 0.0771
Epoch 2/150
100/100 [=====] - 6s 59ms/step - loss: 0.0703 - val_loss: 0.0706
Epoch 3/150
100/100 [=====] - 4s 39ms/step - loss: 0.0600 - val_loss: 0.0426
Epoch 44/150
100/100 [=====] - 4s 38ms/step - loss: 0.0242 - val_loss: 0.0243
Epoch 45/150
100/100 [=====] - 4s 39ms/step - loss: 0.0270 - val_loss: 0.0260
Epoch 00045: early stopping
```

The previous example shows that the model stopped within 45 epochs instead of running for 150.

Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
<hr/>		
bidirectional (Bidirectional (None, 48, 300)		190800
bidirectional_1 (Bidirection (None, 100)		140400
dense (Dense)	(None, 20)	2020
dropout (Dropout)	(None, 20)	0
dense_1 (Dense)	(None, 10)	210
<hr/>		
Total params:	333,430	
Trainable params:	333,430	
Non-trainable params:	0	

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 7-4 depicts a line plot that helps us understand how a good model is able to generalize.

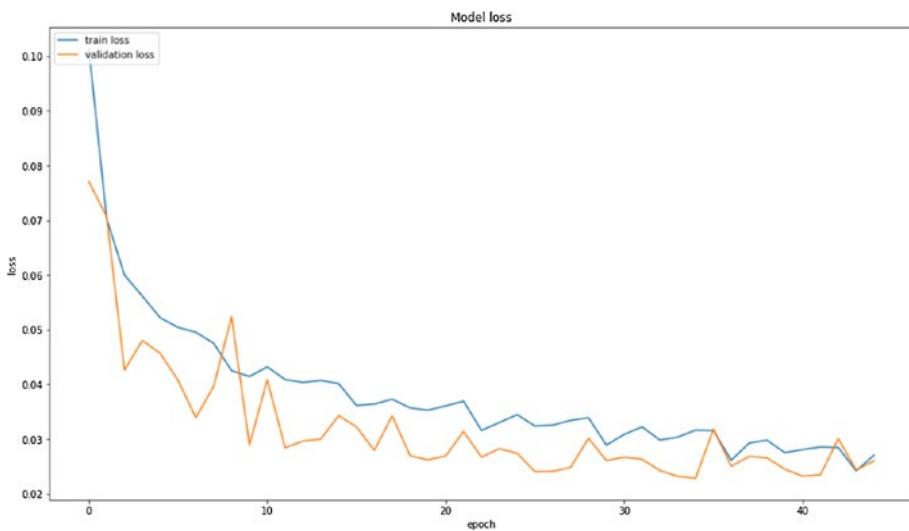


Figure 7-4. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
data_val = x_scaler.fit_transform(df[['rain_1h','temp',
'snow_1h', 'clouds_all', 'holiday_le',
'weather_main_le', 'weather_description_le','traffic_
volume']].tail(48))

val_rescaled = data_val.reshape(1, data_val.shape[0], data_
val.shape[1])

Predicted_results = Trained_model.predict(val_rescaled)
Predicted_results

array([[0.6872767 , 0.6792697 , 0.6635181 , 0.6481182 , 0.614769 ,
0.5839548 , 0.5738882 , 0.54988813, 0.5299208 , 0.5104541 ]],  
dtype=float32)
```

Rescale predicted values back to the original scale.

```
Predicted_results_Inv_trans = y_scaler.inverse_
transform(Predicted_results)
Predicted_results_Inv_trans

array([[5003.3745, 4945.083 , 4830.4116, 4718.3003, 4475.518 , 4251.191 ,
       4177.906 , 4003.1855, 3857.8235, 3716.106 ]], dtype=float32)
```

Define the time-series evaluation function.

```
from sklearn import metrics
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=
'\n\n')

timeseries_evaluation_metrics_func(validate['traffic_
volume'],Predicted_results_Inv_trans[0])

Evaluation metric results:-
MSE is : 2110896.449748737
MAE is : 1212.5899169921875
RMSE is : 1452.8924425946805
MAPE is : 68.28356050566356
R2 is : -0.45477972355275
```

Plot the actual versus predicted values.

```
plt.plot( list(validate['traffic_volume']))  
plt.plot( list(Predicted_results_Inv_trans[0]))  
plt.title("Actual vs Predicted")  
plt.ylabel("Traffic volume")  
plt.legend(('Actual','predicted'))  
plt.show()
```

Figure 7-5 shows a line plot that depicts how actual and predicted values change through time.

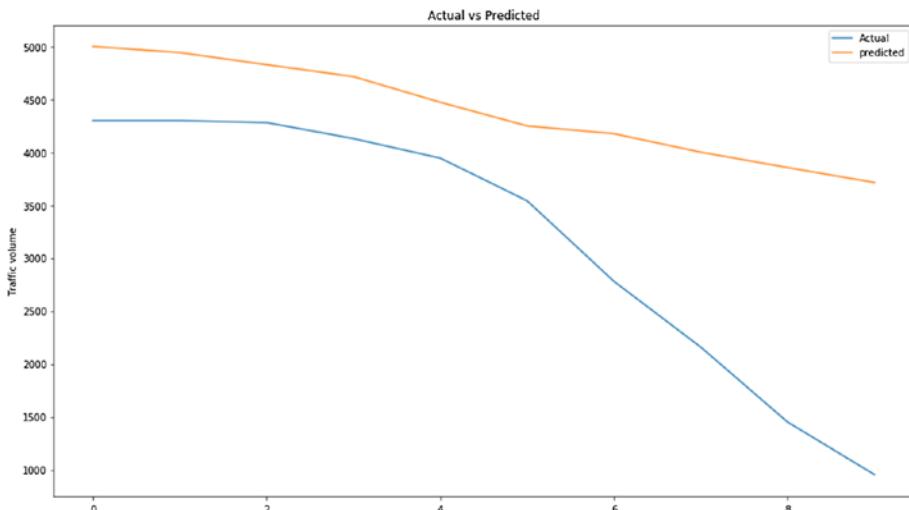


Figure 7-5. Representation of actual versus predicted values

Auto-encoder LSTM Multivariate Horizon Style in Action

In this section, let's use horizon-style data preparation and auto-encoder LSTM to solve multivariate time-series problems.

Import the required libraries and load the CSV data.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's load and take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000		0.510000	100.000000

Let's drop duplicates, as the same hour has different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Apply label encoding on categorical features, and convert to numeric.

```
holiday_le = preprocessing.LabelEncoder()
df['holiday_le'] = holiday_le.fit_transform(df['holiday'])
weather_main_le = preprocessing.LabelEncoder()
df['weather_main_le'] = weather_main_le.fit_
transform(df['weather_main'])
weather_description_le = preprocessing.LabelEncoder()
df['weather_description_le'] = weather_description_le.fit_
transform(df['weather_description'])
```

Define a function to prepare multivariate data suitable for a time series.

```
def custom_ts_multi_data_prep(dataset, target, start, end,
window, horizon):
    X = []
    y = []
    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indices = range(i-window, i)
        X.append(dataset[indices])
        indicey = range(i+1, i+1+horizon)
        y.append(target[indicey])
    return np.array(X), np.array(y)
```

Train/test the split, and let's hold back ten hours of data (i.e., ten records), which we can use to validate the results after training on past data.

```
validate = df[['rain_1h','temp', 'snow_1h', 'clouds_all',
'holiday_le','weather_main_le', 'weather_description_le',
'traffic_volume']].tail(10)
df.drop(df.tail(10).index,inplace=True)
```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
x_scaler = preprocessing.MinMaxScaler()
y_scaler = preprocessing.MinMaxScaler()
dataX = x_scaler.fit_transform(df[['rain_1h','temp', 'snow_1h',
'clouds_all', 'holiday_le','weather_main_le', 'weather_
description_le','traffic_volume']])
dataY = y_scaler.fit_transform(df[['traffic_volume']])
```

As we are doing horizon-style forecasting, let's allow the model to see/train on the past 48 hours of data and try to forecast the next ten hours of results. Hence, use `horizon = 10`.

```
hist_window = 48
horizon = 10
TRAIN_SPLIT = 30000
x_train_multi, y_train_multi = custom_ts_multi_data_prep(
    dataX, dataY, 0, TRAIN_SPLIT, hist_window, horizon)
x_val_multi, y_val_multi= custom_ts_multi_data_prep(
    dataX, dataY, TRAIN_SPLIT, None, hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_data_multi = tf.data.Dataset.from_tensor_slices((x_train_
multi, y_train_multi))
train_data_multi = train_data_multi.cache().shuffle(BUFFER_
SIZE).batch(BATCH_SIZE).repeat()

val_data_multi = tf.data.Dataset.from_tensor_slices((x_val_
multi, y_val_multi))
val_data_multi = val_data_multi.batch(BATCH_SIZE).repeat()
```

Define the auto-encoder LSTM model.

```
ED_lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(40, input_shape=x_train_
multi.shape[-2:], return_sequences=True),
    tf.keras.layers.LSTM(units=20,return_sequences=True),
    tf.keras.layers.LSTM(units=15),
    tf.keras.layers.RepeatVector(y_train_multi.shape[1]),
    tf.keras.layers.LSTM(units=40,return_sequences=True),
    tf.keras.layers.LSTM(units=25,return_sequences=True),
    tf.keras.layers.TimeDistributed(tf.keras.layers.Dense
(units=1))
])
ED_lstm_model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 7\Encoder_Decoder_LSTM_Multivariate.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when the monitored loss starts to increase above the desired limit.

Checkpointing saves the model weights as they reach minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = ED_lstm_model.fit(train_data_multi,
epochs=EPOCHS,steps_per_epoch=EVALUATION_INTERVAL,validation_
data=val_data_multi, validation_steps=50,verbose =1,
callbacks =[tf.keras.callbacks
.EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'),tf.keras.callbacks.ModelCheckpoint(mod
el_path,monitor='val_loss', save_best_only=True, mode='min',
verbose=0)])]

Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 14s 142ms/step - loss: 0.1057 - val_loss: 0.0825
Epoch 2/150
100/100 [=====] - 3s 28ms/step - loss: 0.0777 - val_loss: 0.0760
Epoch 3/150
100/100 [=====] - 3s 25ms/step - loss: 0.0664 - val_loss: 0.0779
Epoch 78/150
100/100 [=====] - 2s 25ms/step - loss: 0.0272 - val_loss: 0.0360
Epoch 79/150
100/100 [=====] - 2s 25ms/step - loss: 0.0258 - val_loss: 0.0332
Epoch 00079: early stopping
```

The previous example shows that the model stopped within 79 epochs instead of running for 150.

Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

CHAPTER 7 BLEEDING-EDGE TECHNIQUES FOR MULTIVARIATE TIME SERIES

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 48, 40)	7840
lstm_1 (LSTM)	(None, 48, 20)	4880
lstm_2 (LSTM)	(None, 15)	2160
repeat_vector (RepeatVector)	(None, 10, 15)	0
lstm_3 (LSTM)	(None, 10, 40)	8960
lstm_4 (LSTM)	(None, 10, 25)	6600
time_distributed (TimeDistri	(None, 10, 1)	26
=====		
Total params:	30,466	
Trainable params:	30,466	
Non-trainable params:	0	

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 7-6 depicts a line plot that helps us understand how a good model is able to generalize.

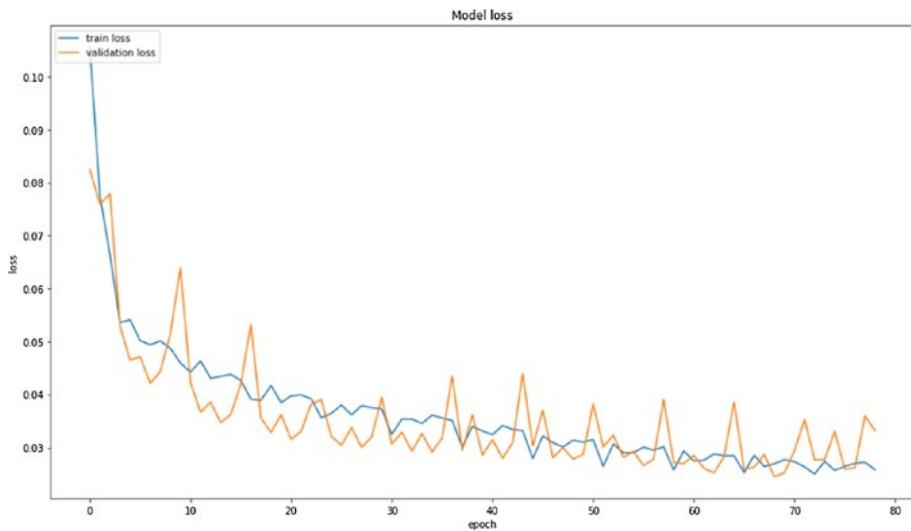


Figure 7-6. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
data_val = x_scaler.fit_transform(df[['rain_1h','temp',
'snow_1h', 'clouds_all', 'holiday_le','weather_main_le',
'weather_description_le','traffic_volume']]).tail(48)

val_rescaled = data_val.reshape(1, data_val.shape[0], data_
val.shape[1])

Predicted_results = Trained_model.predict(val_rescaled)
Predicted_results
```

```
array([[[0.6084024 ],
       [0.6158049 ],
       [0.6135753 ],
       [0.6144647 ],
       [0.62526846],
       [0.6364912 ],
       [0.6360326 ],
       [0.60940194],
       [0.5518158 ],
       [0.47838104]]], dtype=float32)
```

Rescale the predicted values back to the original scale.

```
Predicted_results_Inv_trans = y_scaler.inverse_
transform(Predicted_results.reshape(-1,1))
Predicted_results_Inv_trans
```

```
array([[4429.1694],
       [4483.0596],
       [4466.828 ],
       [4473.303 ],
       [4551.9546],
       [4633.656 ],
       [4630.3174],
       [4436.4463],
       [4017.219 ],
       [3482.614 ]], dtype=float32)
```

Define the time-series evaluation function.

```
from sklearn import metrics
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
```

```
print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_true, y_pred))}')
print(f'MAPE is : {mean_absolute_percentage_error(y_true, y_pred)}')
print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=
'\n\n')

timeseries_evaluation_metrics_func(validate['traffic_
volume'],Predicted_results_Inv_trans)

Evaluation metric results:-
MSE is : 2330110.7698887466
MAE is : 1150.74482421875
RMSE is : 1526.4700356996027
MAPE is : 77.82670425027956
R2 is : -0.512742984485576
```

Plot the actual versus predicted values.

```
plt.plot( list(validate['traffic_volume']))
plt.plot( list(Predicted_results_Inv_trans))
plt.title("Actual vs Predicted")
plt.ylabel("Traffic volume")
plt.legend(('Actual','predicted'))
plt.show()
```

Figure 7-7 shows a line plot that depicts how actual and predicted values change through time.

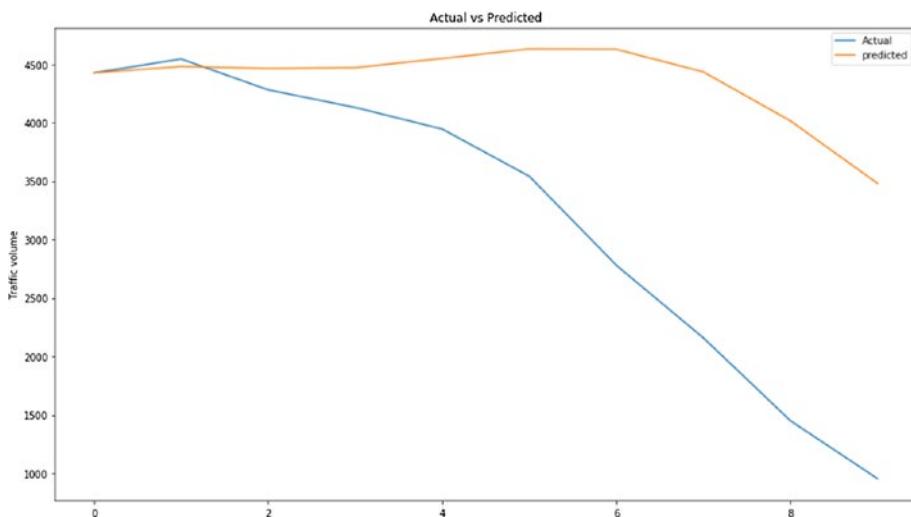


Figure 7-7. Representation of actual versus predicted values

GRU Multivariate Horizon Style in Action

In this section, let's use horizon-style data preparation and GRU to solve multivariate time-series problems.

Import the required libraries and load the CSV data.

```
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn import preprocessing
import matplotlib.pyplot as plt
tf.random.set_seed(123)
np.random.seed(123)
```

Let's load and take a sneak peek at the data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

`df.describe()`

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour has different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Apply label encoding on categorical features, and convert to numeric.

```
holiday_le = preprocessing.LabelEncoder()
df['holiday_le'] = holiday_le.fit_transform(df['holiday'])
weather_main_le = preprocessing.LabelEncoder()
df['weather_main_le'] = weather_main_le.fit_
transform(df['weather_main'])
weather_description_le = preprocessing.LabelEncoder()
df['weather_description_le'] = weather_description_le.fit_
transform(df['weather_description'])
```

Define a function to prepare multivariate data suitable for a time series.

```
def custom_ts_multi_data_prep(dataset, target, start, end,
window, horizon):
    X = []
    y = []
    start = start + window
    if end is None:
        end = len(dataset) - horizon

    for i in range(start, end):
        indices = range(i-window, i)
        X.append(dataset[indices])

        indicey = range(i+1, i+1+horizon)
        y.append(target[indicey])
    return np.array(X), np.array(y)
```

Train/test the split, and let's hold back ten hours of data (i.e., ten records), which we can use to validate the results after training on past data.

```
validate = df[['rain_1h','temp', 'snow_1h', 'clouds_all',
'holiday_le','weather_main_le', 'weather_description_le',
'traffic_volume']].tail(10)
df.drop(df.tail(10).index,inplace=True)
```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```
x_scaler = preprocessing.MinMaxScaler()
y_scaler = preprocessing.MinMaxScaler()
```

```
dataX = x_scaler.fit_transform(df[['rain_1h','temp', 'snow_1h',
'clouds_all', 'holiday_le','weather_main_le', 'weather_
description_le','traffic_volume']])
dataY = y_scaler.fit_transform(df[['traffic_volume']])
```

As we are doing horizon-style forecasting, let's allow the model to see/train on the past 48 hours of data and try to forecast the next ten hours of results. Hence, use `horizon = 10`.

```
hist_window = 48
horizon = 10
TRAIN_SPLIT = 30000
x_train_multi, y_train_multi = custom_ts_multi_data_prep(
    dataX, dataY, 0, TRAIN_SPLIT, hist_window, horizon)
x_val_multi, y_val_multi= custom_ts_multi_data_prep(
    dataX, dataY, TRAIN_SPLIT, None, hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_data_multi = tf.data.Dataset.from_tensor_slices((x_train_
multi, y_train_multi))
train_data_multi = train_data_multi.cache().shuffle(BUFFER_
SIZE).batch(BATCH_SIZE).repeat()

val_data_multi = tf.data.Dataset.from_tensor_slices((x_val_
multi, y_val_multi))
val_data_multi = val_data_multi.batch(BATCH_SIZE).repeat()
```

Define the GRU model.

```
GRU_model = tf.keras.models.Sequential([
    tf.keras.layers.GRU(100, input_shape=x_train_multi.shape
[-2:], return_sequences=True),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.GRU(units=50, return_sequences=False),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(units=horizon),
])
GRU_model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at model_path.

```
model_path = r'\Chapter 7\GRU_Multivariate.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when the monitored loss starts to increase above the desired limit.

Checkpointing saves the model weights as they reach the minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = GRU_model.fit(train_data_multi, epochs=EPOCHS, steps_
per_epoch=EVALUATION_INTERVAL, validation_data=val_data_multi,
validation_steps=50, verbose=1, callbacks=[tf.keras.callbacks.
EarlyStopping(monitor='val_loss', min_delta=0, patience=10,
verbose=1, mode='min'), tf.keras.callbacks.ModelCheckpoint
(model_path, monitor='val_loss', save_best_only=True,
mode='min', verbose=0)])
```

CHAPTER 7 BLEEDING-EDGE TECHNIQUES FOR MULTIVARIATE TIME SERIES

```
Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 8s 76ms/step - loss: 0.0889 - val_loss: 0.0533
Epoch 2/150
100/100 [=====] - 2s 16ms/step - loss: 0.0634 - val_loss: 0.0560
Epoch 3/150
100/100 [=====] - 2s 16ms/step - loss: 0.0553 - val_loss: 0.0654
Epoch 51/150
100/100 [=====] - 2s 16ms/step - loss: 0.0300 - val_loss: 0.0347
Epoch 52/150
100/100 [=====] - 2s 15ms/step - loss: 0.0260 - val_loss: 0.0279
Epoch 00052: early stopping
```

The previous example shows that the model stopped within 52 epochs instead of running for 150.

Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
<hr/>		
gru (GRU)	(None, 48, 100)	33000
dropout (Dropout)	(None, 48, 100)	0
gru_1 (GRU)	(None, 50)	22800
dropout_1 (Dropout)	(None, 50)	0
dense (Dense)	(None, 10)	510
<hr/>		
Total params:	56,310	
Trainable params:	56,310	
Non-trainable params:	0	

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
```

```
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 7-8 depicts a line plot that helps us understand how a good model is able to generalize.

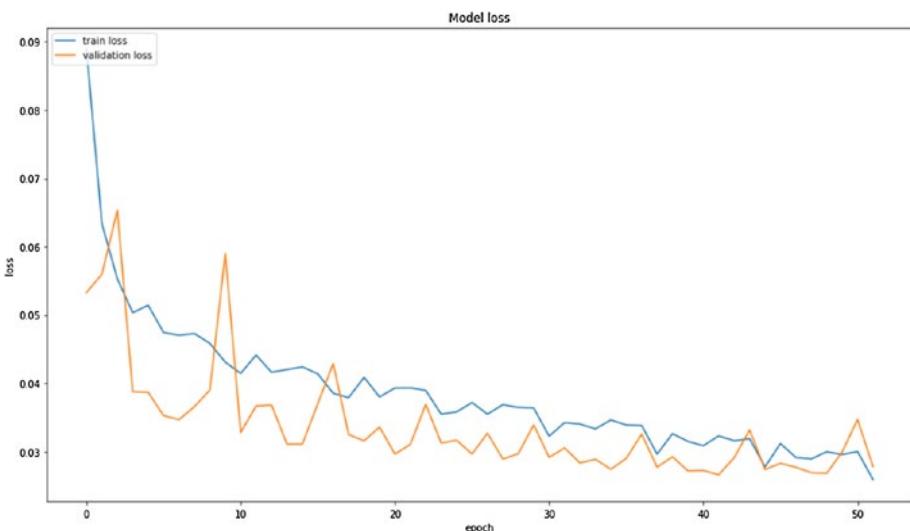


Figure 7-8. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
data_val = x_scaler.fit_transform(df[['rain_1h','temp',
'snow_1h', 'clouds_all', 'holiday_le',
'weather_main_le', 'weather_description_le','traffic_
volume']].tail(48))

val_rescaled = data_val.reshape(1, data_val.shape[0], data_
val.shape[1])
```

```
Predicted_results = Trained_model.predict(val_rescaled)
Predicted_results

array([[0.5811878 , 0.57687247, 0.55752444, 0.52462524, 0.49900812,
       0.4843861 , 0.47486484, 0.44004962, 0.39177603, 0.35142347]],

      dtype=float32)
```

Rescale the predicted values back to the original scale.

```
Predicted_results_Inv_trans = y_scaler.inverse_
transform(Predicted_results)
Predicted_results_Inv_trans

array([[4231.047 , 4199.6313, 4058.7778, 3819.2717, 3632.779 , 3526.3308,
       3457.016 , 3203.5613, 2852.1294, 2558.3628]], dtype=float32)
```

Define the time-series evaluation function.

```
from sklearn import metrics
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true,
y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true,
y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error
(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')

    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=
'\n\n')
```

```
timeseries_evaluation_metrics_func(validate['traffic_volume'],Predicted_results_Inv_trans[0])
```

```
Evaluation metric results:-  
MSE is : 649499.0500393391  
MAE is : 614.023193359375  
RMSE is : 805.9150389708205  
MAPE is : 37.090497246226114  
R2 is : 0.5783354404975487
```

Plot the actual versus predicted values.

```
plt.plot( list(validate['traffic_volume']))  
plt.plot( list(Predicted_results_Inv_trans[0]))  
plt.title("Actual vs Predicted")  
plt.ylabel("Traffic volume")  
plt.legend(('Actual','predicted'))  
plt.show()
```

Figure 7-9 shows a line plot that depicts how actual and predicted values change through time.

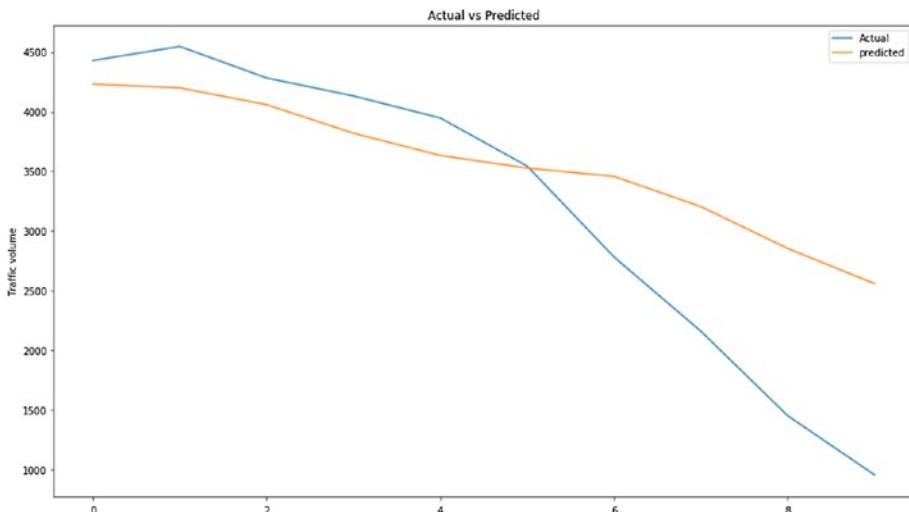


Figure 7-9. Representation of actual versus predicted values

CNN Multivariate Horizon Style in Action

In this section, let's use horizon-style data preparation and CNNs to solve multivariate time-series problems.

Import the required libraries and load the CSV data.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import preprocessing
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten, TimeDistributed, RepeatVector
from tensorflow.keras.layers import Conv1D
from tensorflow.keras.layers import MaxPool1D
from tensorflow.keras.layers import Dropout
tf.random.set_seed(123)
np.random.seed(123)
```

Let's load and take a sneak peek at data by checking the five-point summary.

```
df = pd.read_csv(r'\Data\Metro_Interstate_Traffic_Volume.csv')
df.head()
```

	holiday	temp	rain_1h	snow_1h	clouds_all	weather_main	weather_description	date_time	traffic_volume
0	None	288.28	0.0	0.0	40	Clouds	scattered clouds	2012-10-02 09:00:00	5545
1	None	289.36	0.0	0.0	75	Clouds	broken clouds	2012-10-02 10:00:00	4516
2	None	289.58	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 11:00:00	4767
3	None	290.13	0.0	0.0	90	Clouds	overcast clouds	2012-10-02 12:00:00	5026
4	None	291.14	0.0	0.0	75	Clouds	broken clouds	2012-10-02 13:00:00	4918

```
df.describe()
```

	temp	rain_1h	snow_1h	clouds_all	traffic_volume
count	48204.000000	48204.000000	48204.000000	48204.000000	48204.000000
mean	281.205870	0.334264	0.000222	49.362231	3259.818355
std	13.338232	44.789133	0.008168	39.015750	1986.860670
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	272.160000	0.000000	0.000000	1.000000	1193.000000
50%	282.450000	0.000000	0.000000	64.000000	3380.000000
75%	291.806000	0.000000	0.000000	90.000000	4933.000000
max	310.070000	9831.300000	0.510000	100.000000	7280.000000

Let's drop duplicates, as the same hour has different weather conditions.

```
df.drop_duplicates(subset=['date_time'],
keep=False,inplace=True)
```

Apply label encoding on categorical features, and convert to numeric.

```
holiday_le = preprocessing.LabelEncoder()
df['holiday_le'] = holiday_le.fit_transform(df['holiday'])
weather_main_le = preprocessing.LabelEncoder()
df['weather_main_le'] = weather_main_le.fit_
transform(df['weather_main'])
weather_description_le = preprocessing.LabelEncoder()
df['weather_description_le'] = weather_description_le.fit_
transform(df['weather_description'])
```

Define a function to prepare multivariate data suitable for a time series.

```
def custom_ts_multi_data_prep(dataset, target, start, end,
window, horizon):
    X = []
    y = []
```

```

start = start + window
if end is None:
    end = len(dataset) - horizon

for i in range(start, end):
    indices = range(i-window, i)
    X.append(dataset[indices])

    indicey = range(i+1, i+1+horizon)
    y.append(target[indicey])
return np.array(X), np.array(y)

```

Train/test the split, and let's hold back ten hours of data (i.e., ten records), which we can use to validate the results after training on past data.

```

validate = df[['rain_1h','temp', 'snow_1h', 'clouds_all',
'holiday_le','weather_main_le', 'weather_description_le',
'traffic_volume']].tail(10)
df.drop(df.tail(10).index,inplace=True)

```

Let's rescale the data as neural networks are known to converge sooner with better accuracy when features are on the same scale.

```

x_scaler = preprocessing.MinMaxScaler()
y_scaler = preprocessing.MinMaxScaler()
dataX = x_scaler.fit_transform(df[['rain_1h','temp', 'snow_1h',
'clouds_all', 'holiday_le',
'weather_main_le', 'weather_description_le','traffic_volume']])
dataY = y_scaler.fit_transform(df[['traffic_volume']])

```

As we are doing horizon-style forecasting, let's allow the model to see/train on the past 48 hours of data and try to forecast the next ten hours of results. Hence, use `horizon = 10`.

```
hist_window = 48
horizon = 10
TRAIN_SPLIT = 30000
x_train_multi, y_train_multi = custom_ts_multi_data_prep(
    dataX, dataY, 0, TRAIN_SPLIT, hist_window, horizon)
x_val_multi, y_val_multi= custom_ts_multi_data_prep(
    dataX, dataY, TRAIN_SPLIT, None, hist_window, horizon)
```

Prepare the training and validation time-series data using the `tf.data` function, which is a much faster and more efficient way of feeding data to the model.

```
BATCH_SIZE = 256
BUFFER_SIZE = 150

train_data_multi = tf.data.Dataset.from_tensor_slices((x_train_
multi, y_train_multi))
train_data_multi = train_data_multi.cache().shuffle(BUFFER_-
SIZE).batch(BATCH_SIZE).repeat()

val_data_multi = tf.data.Dataset.from_tensor_slices((x_val_
multi, y_val_multi))
val_data_multi = val_data_multi.batch(BATCH_SIZE).repeat()
```

Define the CNN model.

```
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(x_train_multi.shape[1], x_train_multi.shape[2])))
model.add(MaxPool1D(pool_size=2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(30, activation='relu'))
model.add(Dropout(0.2))
```

```
model.add(Dense(horizon))
model.compile(optimizer='adam', loss='mse')
```

The best weights are stored at `model_path`.

```
model_path = r'\Chapter 7\CNN_Multivariate.h5'
```

Configure the model and start training with early stopping and checkpointing.

Early stopping stops training when the monitored loss starts to increase above the desired limit.

Checkpointing saves the model weights as they reach minimum loss.

```
EVALUATION_INTERVAL = 100
EPOCHS = 150
history = model.fit(train_data_multi, epochs=EPOCHS, steps_per_epoch=EVALUATION_INTERVAL, validation_data=val_data_multi, validation_steps=50, verbose=1, callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=10, verbose=1, mode='min'), tf.keras.callbacks.ModelCheckpoint(model_path, monitor='val_loss', save_best_only=True, mode='min', verbose=0)])
```

```
Train for 100 steps, validate for 50 steps
Epoch 1/150
100/100 [=====] - 3s 34ms/step - loss: 0.1054 - val_loss: 0.0604
Epoch 2/150
100/100 [=====] - 3s 28ms/step - loss: 0.0766 - val_loss: 0.0591
Epoch 3/150
100/100 [=====] - 1s 8ms/step - loss: 0.0690 - val_loss: 0.0583
Epoch 58/150
100/100 [=====] - 1s 7ms/step - loss: 0.0335 - val_loss: 0.0358
Epoch 59/150
100/100 [=====] - 1s 7ms/step - loss: 0.0297 - val_loss: 0.0306
Epoch 00059: early stopping
```

The previous example shows that the model stopped within 59 epochs instead of running for 150.

CHAPTER 7 BLEEDING-EDGE TECHNIQUES FOR MULTIVARIATE TIME SERIES

Load the best weights into the model.

```
Trained_model = tf.keras.models.load_model(model_path)
```

Check the model summary.

```
Trained_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
conv1d (Conv1D)	(None, 46, 64)	1600
=====		
max_pooling1d (MaxPooling1D)	(None, 23, 64)	0
=====		
dropout (Dropout)	(None, 23, 64)	0
=====		
flatten (Flatten)	(None, 1472)	0
=====		
dense (Dense)	(None, 30)	44190
=====		
dropout_1 (Dropout)	(None, 30)	0
=====		
dense_1 (Dense)	(None, 10)	310
=====		
Total params: 46,100		
Trainable params: 46,100		
Non-trainable params: 0		

Plot the loss and val_loss against the epoch.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train loss', 'validation loss'], loc='upper left')
plt.rcParams["figure.figsize"] = [16,9]
plt.show()
```

Figure 7-10 depicts a line plot that helps us understand how a good model is able to generalize.

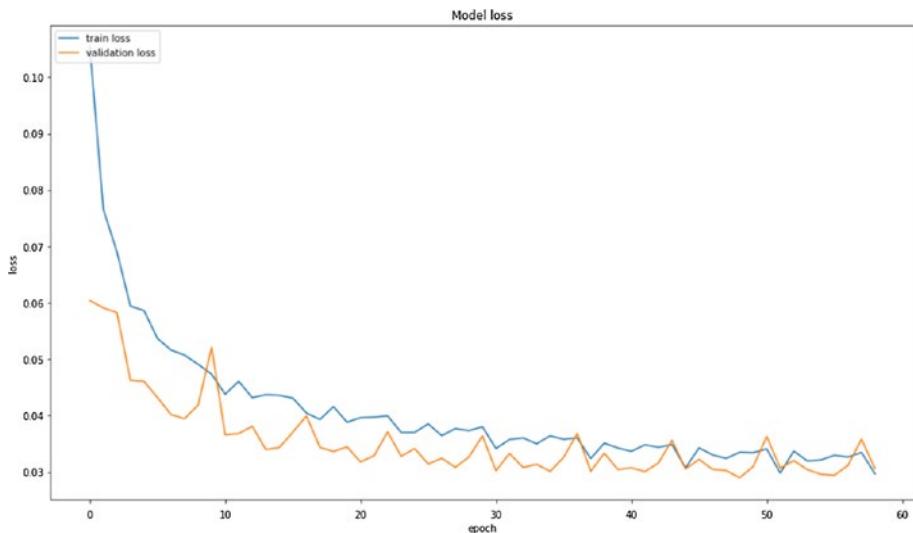


Figure 7-10. Representation of training and validation loss as the number of epochs increases

We need to forecast the next ten steps, so let's take the last 48 hours of data from the training model and forecast the next ten hours of values.

```
data_val = x_scaler.fit_transform(df[['rain_1h','temp',
'snow_1h', 'clouds_all', 'holiday_le',
'weather_main_le', 'weather_description_le','traffic_
volume']].tail(48))

val_rescaled = data_val.reshape(1, data_val.shape[0], data_
val.shape[1])

Predicted_results = Trained_model.predict(val_rescaled)
Predicted_results
```

```
array([[0.79223406, 0.72710294, 0.7083809 , 0.69858336, 0.66632664,
       0.6146801 , 0.5908397 , 0.572573 , 0.5436355 , 0.46793848]],  
      dtype=float32)
```

Rescale the predicted values back to the original scale.

```
Predicted_results_Inv_trans = y_scaler.inverse_  
transform(Predicted_results)
```

Define the time-series evaluation function.

```
from sklearn import metrics  
  
def timeseries_evaluation_metrics_func(y_true, y_pred):  
  
    def mean_absolute_percentage_error(y_true, y_pred):  
        y_true, y_pred = np.array(y_true), np.array(y_pred)  
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100  
    print('Evaluation metric results:-')  
    print(f'MSE is : {metrics.mean_squared_error(y_true,  
y_pred)}')  
    print(f'MAE is : {metrics.mean_absolute_error(y_true,  
y_pred)}')  
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error  
(y_true, y_pred))}')  
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,  
y_pred)}')  
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',end=  
'\n\n')  
timeseries_evaluation_metrics_func(validate['traffic_  
volume'],Predicted_results_Inv_trans[0])
```

```
Evaluation metric results:-  
MSE is : 2435958.60854398  
MAE is : 1423.810498046875  
RMSE is : 1560.7557811983206  
MAPE is : 71.70855471685768  
R2 is : -0.5814609945552476
```

Plot the actual versus predicted values.

```
plt.plot( list(validate['traffic_volume']))  
plt.plot( list(Predicted_results_Inv_trans[0]))  
plt.title("Actual vs Predicted")  
plt.ylabel("Traffic volume")  
plt.legend(('Actual','predicted'))  
plt.show()
```

Figure 7-11 shows a line plot that depicts how the actual and predicted values change through time.

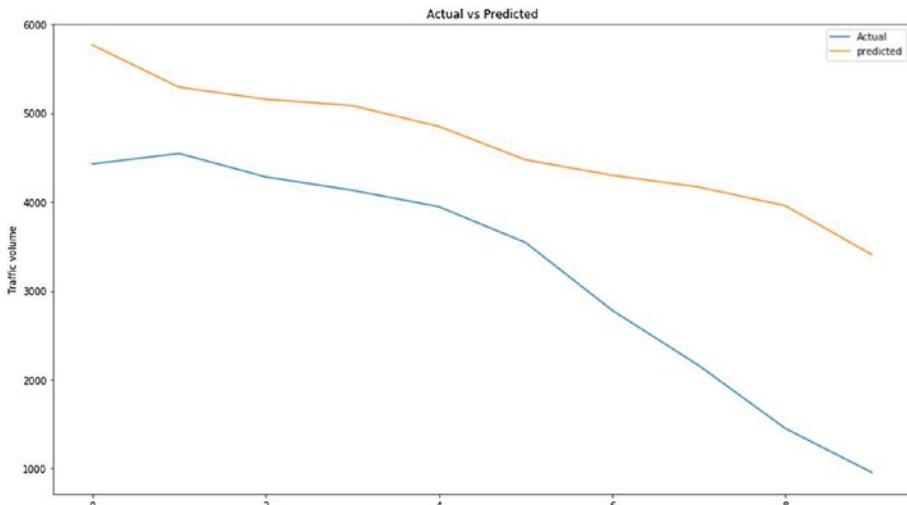


Figure 7-11. Representation of actual versus predicted values

We evaluated the same data using different styles of data preparation and model architectures and observed the power of neural networks. To improve accuracy, try hyper-parameter tuning using Hyperopt, Optune, or Keras Tuner, which can increase accuracy significantly.

Summary

In this chapter, you learned how to use single-step and horizon-style data preparation for a time series and how to solve multivariate time-series problems using LSTM, bidirectional LSTM, GRU, auto-encoders, and CNNs. In the next chapter, you will learn how to solve univariate and multivariate problems using Prophet.

CHAPTER 8

Prophet

Prophet is an open source framework from Facebook used for framing and forecasting time series. It focuses on an additive model where nonlinear trends fit with daily, weekly, and yearly seasonality and additional holiday effects. Prophet is powerful at handling missing data and shifts within the trends and generally handles outliers well. It also allows you to accumulate exogenous variables to the model.

The Prophet Model

Prophet uses a decomposable time-series model.

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$

where

g(t) = Trend (linear/logistic)

s(t) = Periodic change/seasonality

$$s(t) = \sum_{n=1}^N \left(a_n \cos\left(\frac{2\pi nt}{P}\right) + b_n \sin\left(\frac{2\pi nt}{P}\right) \right)$$

Here, P is the consistent period we anticipate the time series to have (e.g., $P = 365.25$ for yearly data or $P = 7$ for weekly data, when we are scaling our time variable in days).

Fitting seasonality requires calculating the $2N$ parameters for $\beta = [a_1, b_1, \dots, a_N, b_N]^T$ by developing a matrix of seasonality vectors for each and every value of t in our historical data and future data.

$$\mathbf{h(t)} = \text{Effect of holiday}$$

For each holiday i , let D_i be the set of past and future dates for that holiday, as shown here:

$$Z(t) = [1(t \in D_1), \dots, 1(t \in D_L)]$$

and taking the following:

$$h(t) = Z(t)\kappa$$

As with seasonality, we use a prior $\kappa \sim \text{Normal}(0, \nu^2)$.

et = Changes that are not adopted by the model

Prophet's core procedure is implemented using Stan (a probabilistic programming language). Stan performs map optimization to find parameters and facilitates estimating parameter uncertainty using the [Hamiltonian Monte Carlo](#) algorithm.

In next section, you will learn how to use Prophet and solve time-series problems.

Implementing Prophet

In the previous section, you learned about the high-level math behind Prophet; now let's implement it on a time-series dataset.

Note The input to the Prophet model should always be a DataFrame with the columns `ds` and `y`, with `ds` being a date field (YYYY-MM-DD or YYYY-MM-DD HH:MM:SS) with a corresponding `y` variable.

Import the required libraries and load the CSV data, which contains bike-sharing data. The data is from January 2011 to December 2012.

Each day comprises 24 observations as data is at the hourly level. Using this dataset, let's try to forecast the count (the target variable) for the next 48 hours, which is two days.

```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import metrics
from fbprophet import Prophet
```

Let's take a sneak peek at the data.

```
df = pd.read_csv('Data\Bike_Sharing_Demand.csv',parse_dates = True)
df.head()
```

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0	3	10	13
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0	0	1	1

Define the time-series evaluation function.

```
def timeseries_evaluation_metrics_func(y_true, y_pred):
    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    #print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true, y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true, y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_
true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true,
y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',
end='\\n\\n')
```

Change the data into a format that Prophet accepts.

```
df = df.rename(columns={'datetime': 'ds', 'count': 'y'})
```

Train/test the split, and let's hold back two days of data (i.e., 48 records), which we can use to validate the results after training on past data.

```
validate = df[['ds','y']].tail(48)
df.drop(df[['ds','y']].tail(48).index,inplace=True)
train = df[['ds','y']]
```

Let's initialize the Prophet class and then fit it with the default parameter, which will internally find the best parameters for most parameters.

```
m = Prophet(yearly_seasonality=False)
m.fit(train)
```

Prophet requires a future DataFrame to do any forecasting, so let's create one with a frequency of H for the next 48 hours.

Other frequency options available are day, week, month, quarter, year, 1 (1 second), 60 (1 minute), 3600 (1 hour), and H (hourly).

```
p = 48
future = m.make_future_dataframe(periods=p,freq='H',
include_history=False)
forecast = m.predict(future)

timeseries_evaluation_metrics_func(validate.y,forecast.yhat)
```

```
MSE is : 9941.56207023959
MAE is : 65.59129402625572
RMSE is : 99.70738222538786
MAPE is : 107.83337535831912
R2 is : 0.6999158728241783
```

Figure 8-1 shows the forecast plot.

```
fig1 = m.plot(forecast)
```

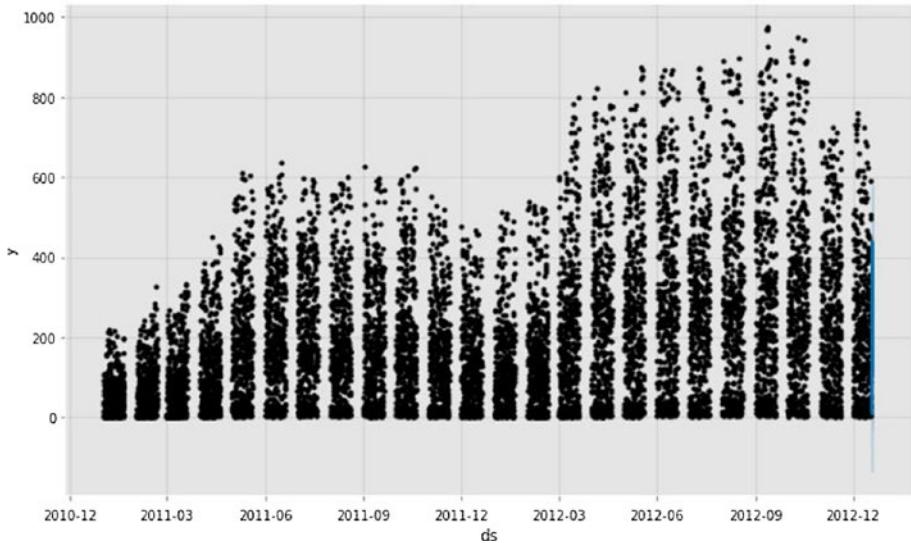


Figure 8-1. Representation of forecast

Let's plot the components of the forecast (Figure 8-2).

```
fig2 = m.plot_components(forecast)
```

CHAPTER 8 PROPHET

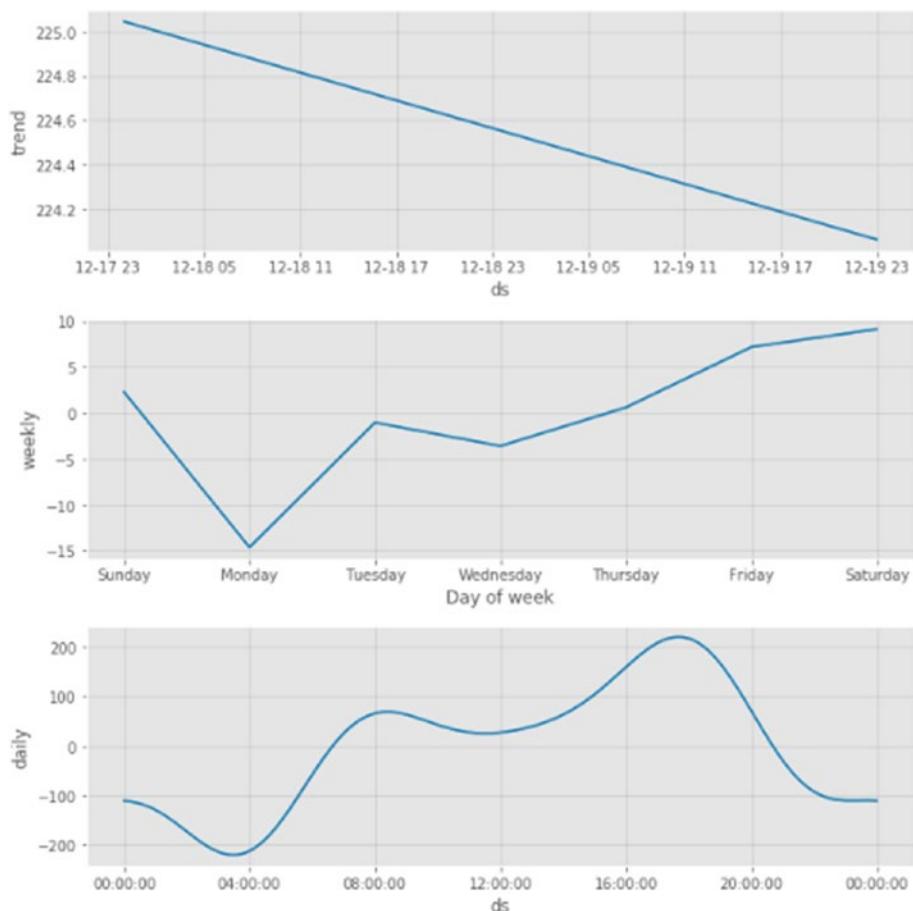


Figure 8-2. Representation of forecasted components

The `add_changepoints_to_plot` function adds red lines; the vertical dashed lines are changepoints where Prophet has identified that the trend has changed (Figure 8-3).

```
from fbprophet.plot import add_changepoints_to_plot
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```

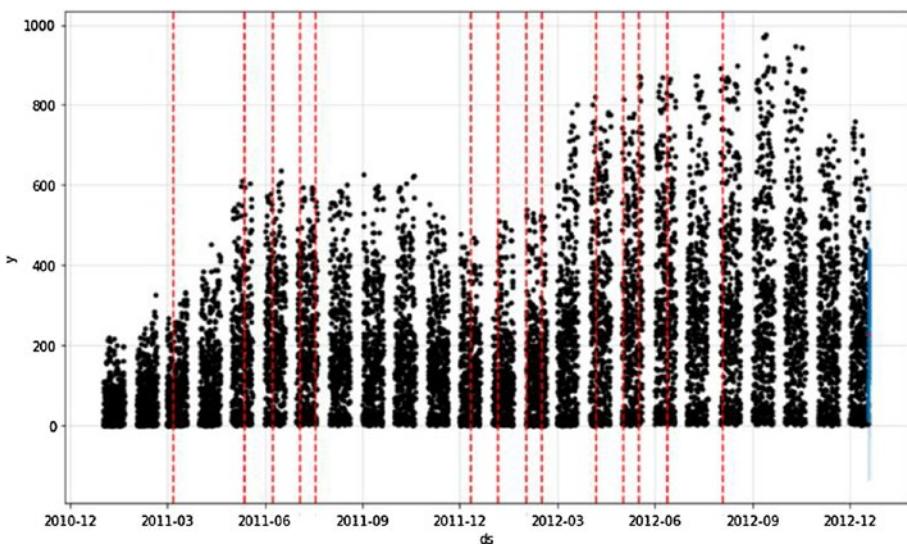


Figure 8-3. Representation of trend changepoints

Adding Log Transformation

In the previous section, you learned how to model time-series problems using Prophet. In this section, let's apply log transformations on the data and check whether the accuracy increases.

Log-transformed data follows a normal or near-normal distribution. Import the required libraries and load the CSV data.

```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import metrics
from fbprophet import Prophet
```

CHAPTER 8 PROPHET

Let's take a sneak peek at the data.

```
df = pd.read_csv('Data\Bike_Sharing_Demand.csv',
parse_dates = True)
df.head()
```

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0	3	10	13
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0	0	1	1

Define the time-series evaluation function.

```
def timeseries_evaluation_metrics_func(y_true, y_pred):

    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    #print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true, y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true, y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_
true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error
(y_true, y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',
end='\n\n')
```

Change the data into a format that Prophet accepts.

```
df = df.rename(columns={'datetime': 'ds', 'count': 'y'})
```

Train/test the split. Let's hold back two days of data (i.e., 48 records), which we can use to validate the results after training on past data.

```
validate = df[['ds','y']].tail(48)
df.drop(df[['ds','y']].tail(48).index,inplace=True)
train = df[['ds','y']]
```

We are applying log transformation to make the data normally distributed.

```
train['y']= np.log(train['y'])
```

Let's initialize the Prophet class and then fit it.

```
m = Prophet(yearly_seasonality=False)
m.fit(train)
```

Prophet requires a future DataFrame to do any forecasting, so let's create one with a frequency of H for the next 48 hours.

```
p = 48
future = make_future_dataframe(periods=p,freq='H',include_
history=False)
forecast = m.predict(future)
```

Like we applied a log transformation on input data, we need to apply an inverse log transformation on the results to bring them back to the original scale.

```
forecast['yhat'] = np.exp(forecast['yhat'])
```

```
timeseries_evaluation_metrics_func(validate.y,forecast['yhat'])
```

```
MSE is : 11670.010469040275
MAE is : 59.29125385426138
RMSE is : 108.02782266175818
MAPE is : 35.65469201451623
R2 is : 0.6477429923997593
```

CHAPTER 8 PROPHET

Figure 8-4 shows the forecast plot.

```
fig1 = m.plot(forecast)
```

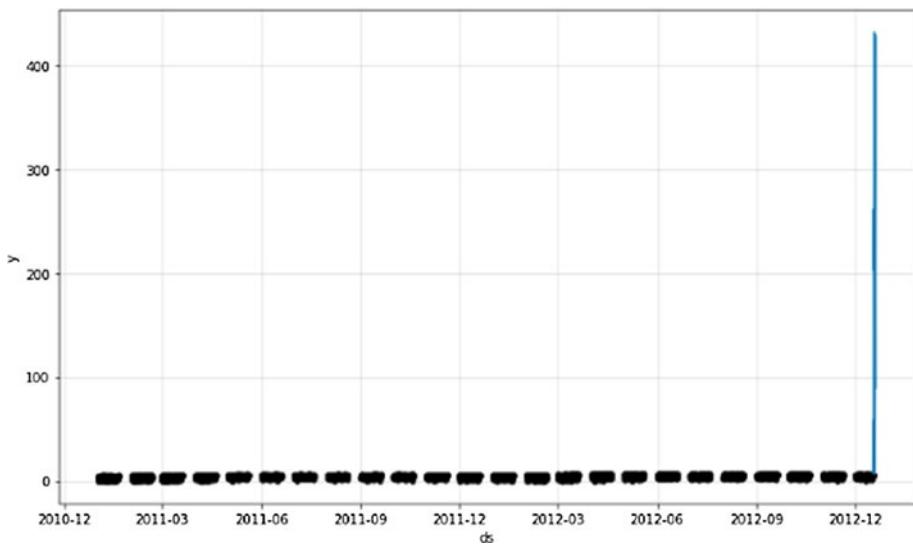


Figure 8-4. Representation of forecast

Let's plot the components of the forecast (Figure 8-5).

```
fig2 = m.plot_components(forecast)
```

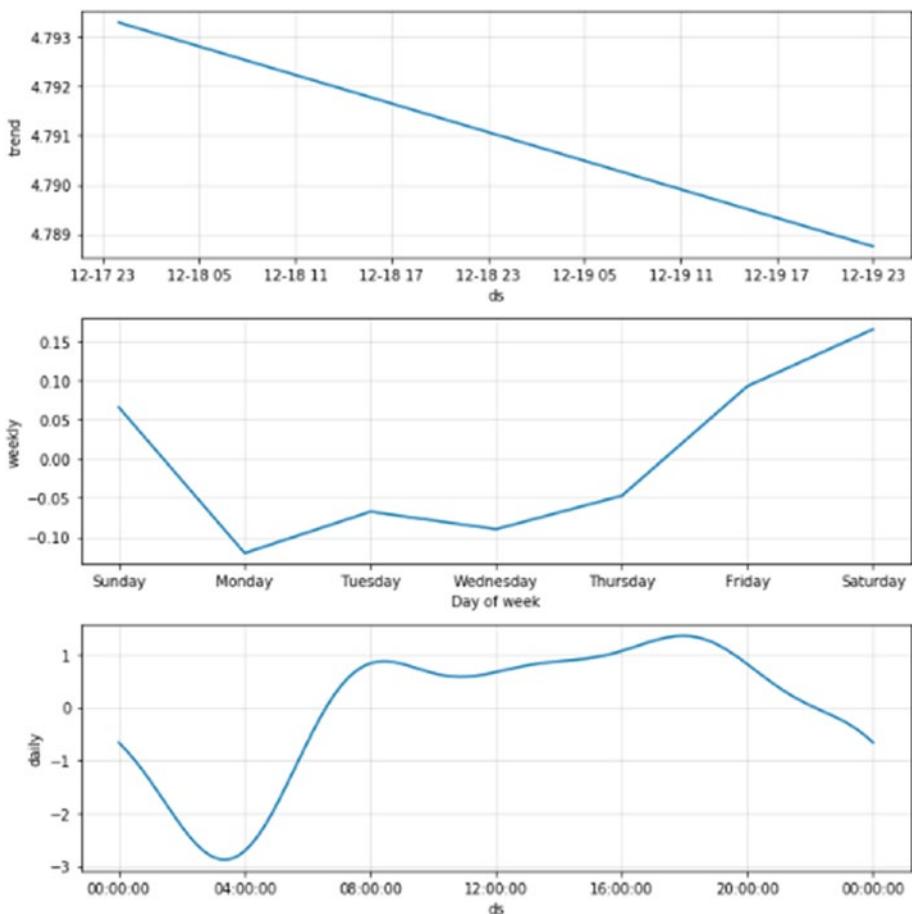


Figure 8-5. Representation of forecasted components

Create the changepoints plot (Figure 8-6).

```
from fbprophet.plot import add_changepoints_to_plot
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```

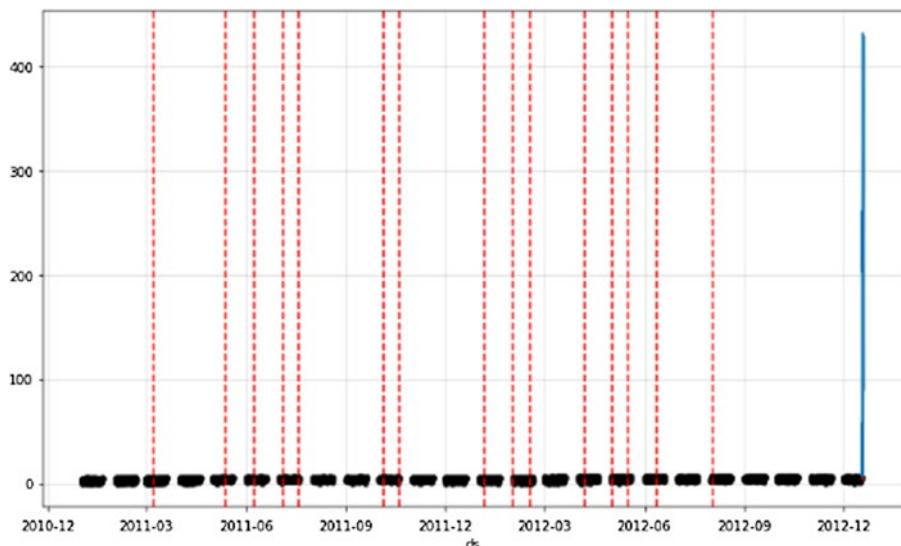


Figure 8-6. Representation of trend changepoints

By applying a log transformation, we didn't see an increase in the model accuracy; rather, it was reduced.

Adding Built-in Country Holidays

In this section, let's add US country holidays, which is built in to Prophet, and check whether the accuracy increases.

Prophet includes holidays for these countries: Brazil (BR), Indonesia (ID), India (IN), Malaysia (MY), Vietnam (VN), Thailand (TH), Philippines (PH), Turkey (TU), Pakistan (PK), Bangladesh (BD), Egypt (EG), China (CN), and Russia (RU).

Import the required libraries and load the CSV data.

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

```
import pandas as pd
from sklearn import metrics
from fbprophet import Prophet
```

Let's take a sneak peek at the data.

```
df = pd.read_csv(r'Data\Bike_Sharing_Demand.csv',
parse_dates = True)
df.head()
```

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0	3	10	13
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0	0	1	1

Define the time-series evaluation function.

```
def timeseries_evaluation_metrics_func(y_true, y_pred):
    def mean_absolute_percentage_error(y_true, y_pred):
        y_true, y_pred = np.array(y_true), np.array(y_pred)
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
    #print('Evaluation metric results:-')
    print(f'MSE is : {metrics.mean_squared_error(y_true, y_pred)}')
    print(f'MAE is : {metrics.mean_absolute_error(y_true, y_pred)}')
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_true, y_pred))}')
    print(f'MAPE is : {mean_absolute_percentage_error(y_true, y_pred)}')
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',
end='\n\n')
```

Change the data into a format that Prophet accepts.

```
df = df.rename(columns={'datetime': 'ds', 'count': 'y'})
```

Train/test the split, and let's hold back two days of data (i.e., 48 records), which we can use to validate the data after training on past data.

```
validate = df[['ds', 'y']].tail(48)
df.drop(df[['ds', 'y']].tail(48).index, inplace=True)
train = df[['ds', 'y']]
```

Let's initialize the Prophet class, add country holidays using `add_country_holidays`, and then fit the data with the default parameters.

```
m = Prophet(yearly_seasonality=False)
m.add_country_holidays(country_name='US')
m.fit(train)
```

Prophet requires a future DataFrame to do any forecasting, so let's create one with a frequency of H for the next 48 hours.

```
p = 48
future = m.make_future_dataframe(periods=p, freq='H', include_history=False)
forecast = m.predict(future)

timeseries_evaluation_metrics_func(validate.y, forecast.yhat)
```

```
MSE is : 10299.423817670899
MAE is : 61.793582875969015
RMSE is : 101.48607696463047
MAPE is : 93.2326573364842
R2 is : 0.6891138852322072
```

Create the changepoints plot (Figure 8-7).

```
from fbprophet.plot import add_changepoints_to_plot
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```

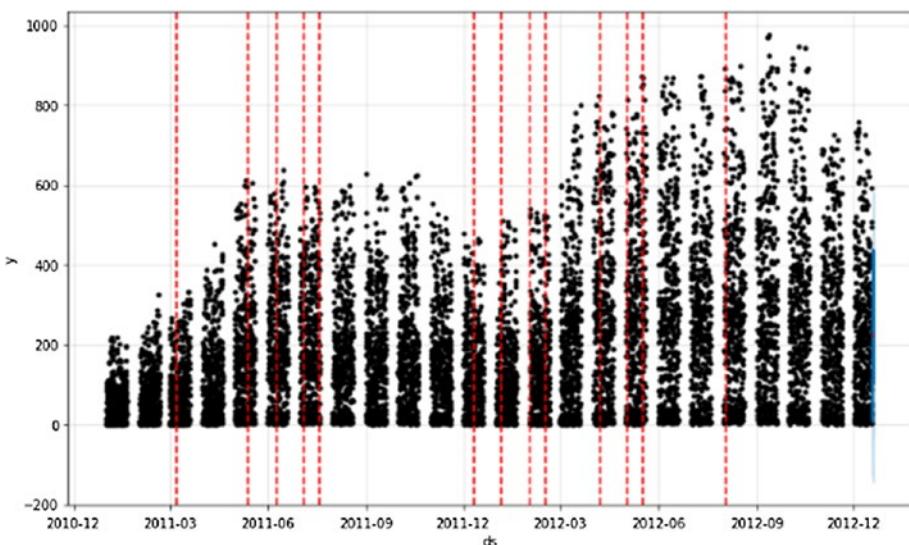


Figure 8-7. Representation of trend changepoints

Adding Exogenous variables using add_regressors(function)

In this section, let's add exogenous variables, which might influence target variables, and check whether the accuracy increases. We can call this problem a multivariate time-series problem too.

```
import warnings
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import metrics
from fbprophet import Prophet
```

CHAPTER 8 PROPHET

Import the required libraries and load the CSV data.

```
df = pd.read_csv('Data\Bike_Sharing_Demand.csv',  
parse_dates = True)  
df.head()
```

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	2011-01-01 00:00:00	1	0	0	1	9.84	14.395	81	0.0	3	13	16
1	2011-01-01 01:00:00	1	0	0	1	9.02	13.635	80	0.0	8	32	40
2	2011-01-01 02:00:00	1	0	0	1	9.02	13.635	80	0.0	5	27	32
3	2011-01-01 03:00:00	1	0	0	1	9.84	14.395	75	0.0	3	10	13
4	2011-01-01 04:00:00	1	0	0	1	9.84	14.395	75	0.0	0	1	1

```
df['datetime'] = pd.to_datetime(df['datetime'])
```

Define the time-series evaluation function.

```
def timeseries_evaluation_metrics_func(y_true, y_pred):  
  
    def mean_absolute_percentage_error(y_true, y_pred):  
        y_true, y_pred = np.array(y_true), np.array(y_pred)  
        return np.mean(np.abs((y_true - y_pred) / y_true)) * 100  
    #print('Evaluation metric results:-')  
    print(f'MSE is : {metrics.mean_squared_error(y_true, y_pred)}')  
    print(f'MAE is : {metrics.mean_absolute_error(y_true, y_pred)}')  
    print(f'RMSE is : {np.sqrt(metrics.mean_squared_error(y_true, y_pred))}')  
    print(f'MAPE is : {mean_absolute_percentage_error(y_true, y_pred)}')  
    print(f'R2 is : {metrics.r2_score(y_true, y_pred)}',  
         end='\n\n')
```

Change the data into a format that Prophet accepts.

```
df = df.rename(columns={'datetime': 'ds', 'count': 'y'})
```

Train/test the split, and let's hold back two days of data (i.e., 48 records), which we can use to validate the results after training on past data. We need to do the same with exogenous and add it to the original DataFrame. Then we can use it further to train and validate the data.

```
validate = df[['ds','y','season','holiday','weather','temp',
'humidity','windspeed']].tail(48)

df.drop(df[['ds','y','season','holiday','weather','temp',
humidity','windspeed']]).tail(48).index,inplace=True)

train = df[['ds','y','season','holiday','weather','temp',
humidity','windspeed']]
```

Let's initialize the Prophet class, then add exogenous variables one by one, and finally fit the model on the training data.

```
m = Prophet(yearly_seasonality=False)
m.add_regressor('season')
m.add_regressor('holiday')
m.add_regressor('weather')
m.add_regressor('temp')
m.add_regressor('humidity')
m.add_regressor('windspeed')
m.fit(train)
```

Prophet requires a future DataFrame to do any forecasting, so let's create one with a 48 frequency of H for the next 48 hours.

```
p = 48
future = m.make_future_dataframe(periods=p,freq='H',include_
history=False)
```

CHAPTER 8 PROPHET

Add exogenous into the forecast DataFrame so that the model can use them while forecasting.

```
future['season'] = validate['season'].values
future['holiday'] = validate['holiday'].values
future['weather'] = validate['weather'].values
future['temp'] = validate['temp'].values
future['humidity'] = validate['humidity'].values
future['windspeed'] = validate['windspeed'].values
forecast = m.predict(future)

timeseries_evaluation_metrics_func(validate.y,forecast.yhat)

MSE is : 10526.397825382928
MAE is : 66.73451882424882
RMSE is : 102.59823500130462
MAPE is : 115.03252471019137
R2 is : 0.6822627187339608
```

Forecast the plot (Figure 8-8).

```
fig1 = m.plot(forecast)
```

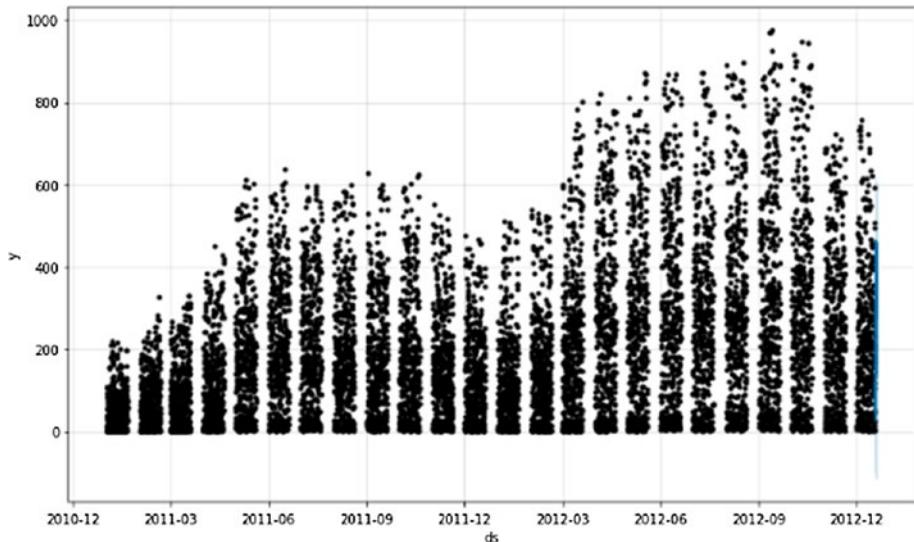


Figure 8-8. Representation of forecast

Let's plot the components of the forecast (Figure 8-9).

```
fig2 = m.plot_components(forecast)
```

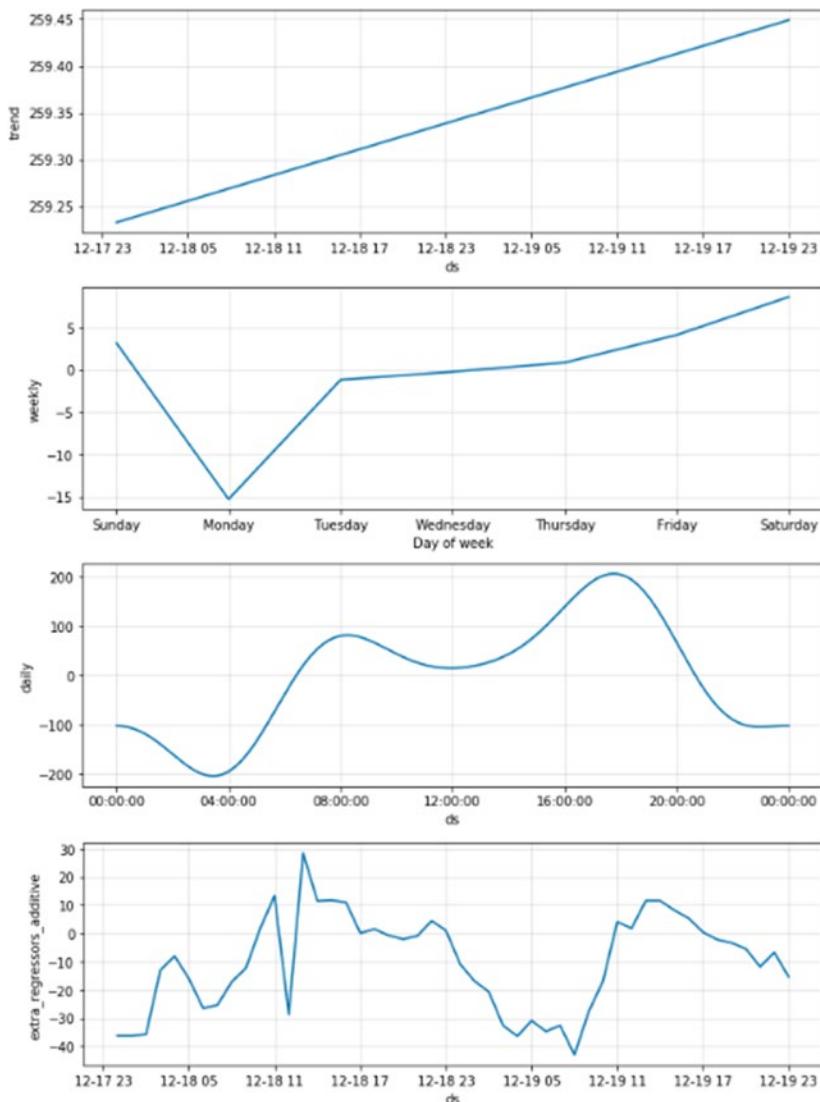


Figure 8-9. Representation of forecasted components

Plot the changepoints (Figure 8-10).

```
from fbprophet.plot import add_changepoints_to_plot
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```

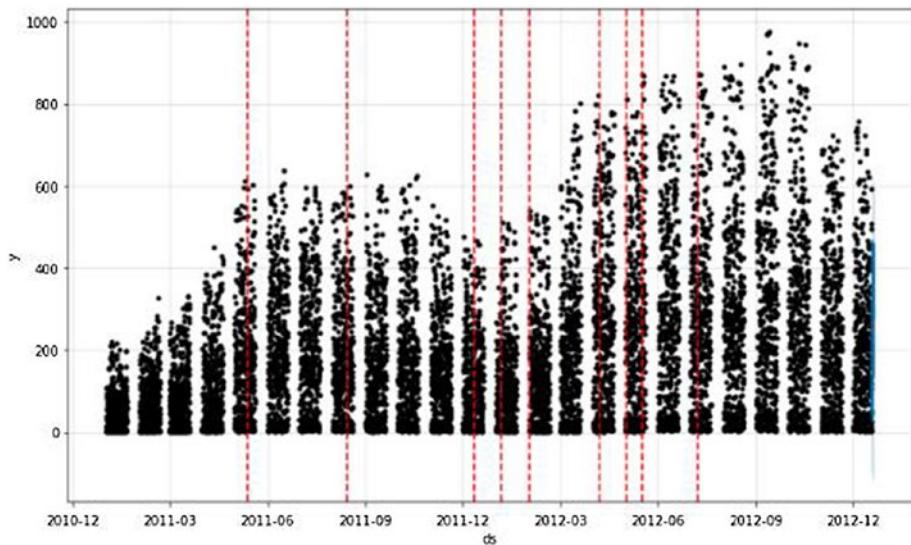


Figure 8-10. Representation of trend changepoints

Summary

In this chapter, you learned about the high-level math behind Prophet, how to create a basic model, how to apply log transformations to time-series data, how to add country holidays, and how to handle multivariate time-series data using `add_regressor`.

Index

A

Activation function
binary step function, 189
linear activation function, 189
nonlinear (*see* Nonlinear activation functions)
types, 188
Additive model, 19, 20
Additive seasonality model, 86
add_regressor, 389–394
.agg() function, 38
Aggregation, GROUP BY, 36, 38, 39
Akaike Information Criterion
(AIC), 168
Alpha, 66
ARMA model, 117–118
Ascending data order, 33
Augmented Dickey-Fuller (ADF)
test, 102, 104
Auto-arima, 127
Autocorrelation function (ACF),
116, 117
Autocorrelation plot, 13, 14
Auto-encoder, 224, 225
Auto-encoder LSTM multivariate
horizon style
actual *vs.* predicted values, 355

best weights, 350, 351
drop duplicates, 348
early stopping and
checkpointing, 350
five-point summary, 347
function, defining, 348
import libraries, 346
model summary, 351
rescale predicted values, 354
time-series evaluation
function, 354
training and validation loss, 353
training and validation
time-series data, 349
train/test split, 349
Auto-encoder LSTM univariate
horizon style
actual *vs.* predicted
values, 305, 306
auto-encoder model, 300
best weights, 301, 302
early stopping and
checkpointing, 301
function, defining, 299
import libraries, 297
line plot, 302
loss and val_loss, 302

INDEX

- Auto-encoder LSTM univariate horizon style (*cont.*)
model summary, checking, 302
predicted values, 304
rescale data, 299
sneak peek, data, 298
time-series evaluation function, 304
training and validation loss, 303
training and validation time-series data, 300
Auto-encoder LSTM univariate single-step style
actual *vs.* predicted values, plot, 296
best weights, 291
early stopping and checkpointing, 291
encode-decoder model, 291
function, defining, 289
import libraries, 288
loss and val_loss, 293
model summary, checking, 292
predicted values, 295
rescale data, 289
sneak peek, data, 288
time-series evaluation function, 295
training and validation loss, 294
training and validation time-series data, 290
train/test, 289
Autoregressive (AR) models, 111–113
- Autoregressive integrated moving average (ARIMA)
ADF test function, 125
confidence bounds, 128
DataFrame, 127
diagnostic plot, 128
exploratory data analysis, 123
Facebook stock data, 123
integration (I), 121, 122
libraries and load CSV data, 122
model training, 126
nonseasonal, 120
pmdarima module, 126
representation of p, d, q, 120, 121
time-series evaluation function, 125
Autoregressive model, 118

B

- Backpropagation through time approach (BPTT), 206, 208–210
Backward propagation, 198
Brute-force approach, 196
gradient descent, 197–199
learning rate *vs.* gradient descent optimizers, 199
representation, 196
Balanced panel data, 5
Bayesian Information Criterion (BIC), 168
Bernoulli distribution, 225

- Bidirectional LSTM multivariate horizon style
 actual *vs.* predicted values, 346
 best weights, 341, 342
 early stopping and
 checkpointing, 341
 five-point summary, 338
 function, defining, 339
 import libraries, 337
 load CSV data, 337
 neural networks, 340
 rescale predicted values, 345
 time-series evaluation
 function, 345
`Trained_model.summary()`, 342
 training and validation
 loss, 344
 training and validation
 time-series data, 340
 train/test, 339
- Bidirectional LSTM univariate horizon style, 262–270
- Bidirectional LSTM univariate single-step style
 actual *vs.* predicted values, 261
 function, defining, 255
 model summary, check, 258
 neural networks, 254
 predicted values, rescale, 260
 required libraries and load
 CSV files, 253
see /train, 255
 sneak peek, data, 253
- stopping and checkpointing, 256
 time-series evaluation
 function, 260
 training and validation loss, 259
 training and validation
 time-series data, 256
 train/test, split, 254
- Binary step function, 189
- Biological neurons *vs.* artificial neural network, 186
- Box plot, 12
- Brute-force approach, 196

C

- Candlestick charts, 6
- CNN multivariate horizon style
 actual *vs.* predicted values,
 plot, 373
 best weights, 369, 370
 drop duplicates, 366
 early stopping and
 checkpointing, 369
 five-point summary, 365
 function, defining, 366
 import libraries, 365
 label encoding, 366
 loss and val_loss, 370
 model summary, 370
 rescale predicted values, 372
 time-series evaluation
 function, 372
 training and validation loss, 371

INDEX

- CNN multivariate
 - horizon style (*cont.*)
 - training and validation
 - time-series data, 368
 - train/test split, 367
- CNN univariate horizon style
 - actual *vs.* predicted values, 323
 - best weights, 319
 - early stopping and
 - checkpointing, 319
 - five-point summary, 316
 - function, defining, 317
 - import libraries, 315
 - model summary, 320
 - predicted values, 322
 - time-series evaluation
 - function, 322
 - training and validation
 - loss, 321
 - training and validation
 - time-series data, 318
 - train/test, 316
- CNN univariate single-step style
 - actual *vs.* predicted values, 314
 - checkpointing, 310
 - configuration, 310
 - five-point summary, 307
 - function, defining, 308
 - import libraries, 306
 - line plot, 312
 - model summary, 311
 - predicted values, 313
 - rescale data, 308
 - see /train*, 309
- time-series evaluation
 - function, 313
- training and validation loss, 312
- training and validation
 - time-series data, 309
 - train/test, 307
- Cointegration test, 166, 177
- Constant error carousel (CEC)
 - units, 210
- Continuous time series, 2
- Convolutional LSTM
 - (ConvLSTM), 215
- Convolution neural networks (CNNs)
 - batch normalization, 222
 - convolutional layer, 220
 - convolution formula, 222
 - convolution operation, 220
 - dropouts, 222
 - filter, 220
 - formulas, 223
 - one-dimensional, 223
 - pooling, 221, 222
- Correlation, 166
- Cost function, 196, 197
- Cross-entropy, 225
- Cross-section data, 4
- CSV files, 24
- Cyclical components, 16
- Cyclic variations
 - accessibility, 17
 - depression, 16
 - detecting, 17, 18
 - prosperity, 16

D

Dampening, 77
 Data filtering, 28, 30
 Data stationery
 plots, 101
 statistics unit root tests, 102, 103
 summary
 statistics, 101
 Data types
 cross-section, 4
 panel data/longitudinal data, 4, 5
 time series, 2, 3
 Data wrangling
 Pandas (*see* Pandas)
 Decoding, 224
 Decomposition, 15, 16, 18–20
 Dendrites, 186
 Descending data
 order, 34, 35
 describe() function, 51
 Descriptive statistics, 51
 Deseasoning, 14
 Detrending
 HP filter, 10, 11
 Pandas differencing, 8
 SciPy signal, 9, 10
 time-series data, 7
 df.shift() function, 58
 Dickey-Fuller (DF) test, 102
 diff() function, 8

Difference stationary, 100
 Differencing (lag difference), 106
 Discrete time series, 3
 Double exponential smoothing
 additive/multiplicative trend, 77
 automated grid search model summary, 84
 best parameters, 83
 custom grid search model, 82
 custom grid search
 parameters, 83
 fit parameters, 80
 Holt parameters, 80
 Holt's exponential smoothing
 forecast formula, 77
 line plot depicting results, 85
 loaded dataset, 78
 sneak peek, 78
 trend component, 76
 Drill-down data, 28
 Dying ReLU problem, 192

E

Encoding, 224
 Expanding window, 57
 Exploratory data analysis, 123
 Exponentially weighted moving window, 57
 Exponential smoothing, 67

INDEX

F

Feed-forward recurrent neural network, 204, 206
First-order differencing (trend differencing), 108, 109
First-order stationary, 100
Forward propagation, 195
Full outer join, 48–50

G

Gated recurrent unit (GRU), 216, 217, 219
Gaussian distribution data, 225
Gradient descent, 197, 198
Gradient descent optimization algorithms, 199
groupby() function, 35, 36, 38
GRU multivariate horizon style
actual *vs.* predicted values, plot, 364
best weights, 360
drop duplicates, 357
early stopping and
checkpointing, 360
five-point summary, 356
function, defining, 358
import libraries, 356
label encoding, 357
line plot, 362
model summary, 361
rescale predicted values, 363
time-series evaluation function, 363

training and validation loss, 362

training and validation

time-series data, 359

train/test split, 358

GRU univariate horizon

style, 279–287

GRU univariate single-step style

actual *vs.* predicted values,

plot, 278

best weights, 274, 275

early stopping and

checkpointing, 274

function, defining, 272

import libraries, 271

loss and val_loss, 275

model summary checking, 275

predicted values, 277

rescale data, 272

see /train, 273

sneak peek, data, 271

time-series evaluation

function, 277

training and validation loss, 276

training and validation

time-series data, 273

train/test, 272

H

Hodrick-Prescott (HP) filter, 6, 7

Holt method, 86

Holt-Winter exponential

smoothing, 87

Holt-Winters method, 86

Horizon-style time-series
forecasting, 229, 230

HP filter detrending, 10

I

Inner join, 41, 43
Inner merge, 41, 43
Inverse differencing
function, 178

J

Join
full outer, 48–50
inner, 41, 43
left, 43, 45
right, 46–48
JSON format, 25

K

Kernel density
estimation (KDE), 123
Kwiatkowski-Phillips-Schmidt-Shin
(KPSS) tests, 105

L

Leaky ReLU, 193
Learning rate *vs.* gradient descent
optimizers, 199
Left join, 43, 45
Left merge, 43, 45

Linear activation function, 189

Loading data into Pandas
CSV files, 24
Excel file, 25
JSON format, 25
URL, 26

Log-transformed data, 381
Long short-term memory (LSTM)
cell architecture with
formulas, 212
convolutional, 215
exploding and vanishing
gradient problems, 210
gates, 211
long-term dependencies, 210
peephole, 214
vs. RNN, 211
step-by-step
explanation, 212–214
T time step, 212

LSTM multivariate horizon style
actual *vs.* predicted
values, 336, 337
bar plot representation, 327
best weights, 332, 333
early stopping and
checkpointing, 332
five-point summary,
checking, 326
import libraries, 326
label encoding, 327
line plot, 333
neural networks, 329
predicted values, 335

INDEX

- LSTM multivariate
 - horizon style (*cont.*)
 - single window of past
 - history, 330
 - time-series evaluation
 - function, 335
- Trained_model.summary (), 334
- training and validation
 - loss, 334
 - training and validation
 - time-series data, 331
 - train/test, 329
- LSTM univariate horizon style
 - actual *vs.* predicted values, 252
 - checkpointing, 248
 - data and forecast, 243
 - function, defining, 244
 - loss and val_loss, 249
 - model summary, 249
 - predicted values, 251
 - required libraries and load CSV
 - files, 242
 - rescale data, 244
 - single window, past
 - history, 245
 - sneak peek, data, 243
 - target horizon, 247
 - time-series evaluation
 - function, 251
 - training and validation loss, 250
 - training and validation
 - time-series data, 247
 - training model and forecast, 250
- LSTM univariate single-step style
 - actual *vs.* predicted
 - values, 241, 242
 - data and forecast, 231
 - drop duplicates, 232
 - function defining, 233
 - loss and val_loss, 238
 - machine and deep learning, 238
 - model summary, 237
 - single window of past
 - history, 234, 235
 - Tensorflow, 231
 - time-series evaluation
 - function, 240
 - training and validation loss, 239
 - training and validation
 - time-series data, 235

M

- Mean absolute error (MAE), 70
- Mean absolute percentage error (MAPE), 70
- Mean squared error (MSE), 70
- Measure of central tendency, 51
- Measure of variability, 51
- Minimal gated unit, 217, 219
- Missing data handling
 - backward fills method, 62
 - FILLNA, 63
 - forward fill method, 62
 - interpolates values, 64
 - Isnull(), 61

Missing value/data, 60

Moving average (MA),
113–115, 119

Multiplicative model, 19

Multivariate time series, 2

Multivariate forecasting
method, 154

N

Neural networks, 185, 186

Neurons, 186

Nonlinear activation
functions, 190, 225
leaky ReLU, 193
parametric ReLU, 193
ReLU, 192
sigmoid, 191
softmax, 194
tanh, 192
training models, 190

Nonstationary time series, 101

Normal random walk, 108

NOT IN operation, 32

NumPy, 68

O

One-dimensional CNNs, 223

Optimization methods, 200, 201

Optimizers, 199

ORDER BY sorts, 33

Outer merge, 48–50

Over-smoothing, 67

P, Q

Pandas, 24

DataFrames, 27

describe() function, 51

filter, apply, 28, 29

group by() function, 35

GROUP BY operation, 36

isin() function, 31

join (merge) (*see* Join)

NOT IN operation, 32

shifting operations, 58

unique() function, 29

Pandas differencing, 8, 167

pandas.head() function, 27

Pandasql

GROUP BY operation, 36

filter, apply, 28, 29

Python framework, 27

sqldf, 28

Panel data/longitudinal
data, 4, 5

Parametric ReLU, 193

Partial autocorrelation function
(PACF), 116, 117

Peephole LSTM, 214

Perceptron, 185–187

Phillips-Perron (PP)

test, 102

Pmdarima, 122

Principal component

analysis (PCA), 224

Probability density

function (PDF), 123

INDEX

- Prophet
- adding built-in country holidays, 386–389
 - adding exogenous/ add_regressors, 389–394
 - apply log transformations, 381, 382, 384–386
 - implementation
 - add_changepoints_to_plot function, 380
 - forecasted components, 380
 - forecast plot, 379
 - future DataFrame, 378
 - import libraries, 376
 - load CSV data, 376
 - time-series evaluation function, 377
 - train/test split, 378
 - trend changepoints, 381
 - open source framework, 375
 - time-series model, 375, 376
- P-value, interpret, 104

R

- R-squared, 70
- Radial basis function, 188
- Random walk, 107
- Rectified linear unit (ReLU), 192
- Recurrent neural network (RNN)
- BPTT (*see* Backpropagation through time approach (BPTT))
 - feed-forward, 204, 206

- vs.* LSTM, 211
- many-to-many relationship, 204
- many-to-one relationship, 203
- vs.* neural network, 203
- one-to-many relationship, 203
- Regression model, 99, 116
- Resampling
- by month, 53
 - by quarter, 53
 - by week, 54
 - by year, 53
 - operations in Pandas, 52
 - semimonthly basis, 55
- Right merge, 46–48
- Rolling window, 56
- Root mean square error (RMSE), 70

S

- SARIMAX model, 143–154
- Schmidt-Phillips test, 102
- SciPy detrending, 9, 10
- SciPy library, 9
- Seasonal ARIMA (SARIMA)
- ADF test function, 133
 - interpreting nonseasonal and seasonal components, 130
 - modeling, 134
 - multiplicative model, 129
 - nonseasonal components, 130
 - nonstationary and auto-arima, 133
 - number of periods, 134
 - search space, 134

sneak peek, data, 132
 time-series dataset, 131
 time-series evaluation function, 132
 trend elements, 129
 univariate time-series data, 129
Seasonal differencing
 first-order differencing, 110
 second-order differencing, 110
Seasonality
 autocorrelation plot, 13, 14
 decomposition, 15, 16
 deseasoning, time-series data, 14
 multiple box plots, 12, 13
 periodical fluctuation, 11
Second-order differencing, 109
Serial correlation, 116
Shifting, 58–60
Sigmoid activation
 function, 191, 211
Simple exponential smoothing (SES)
 algorithm's hyperparameters, 69
 alpha, 67
 forecast estimate level, 66
 high-level math behind, 66
 line plot depicting results, 75
 mean absolute percentage error (MAPE), 70, 71
NumPy, 68
 optimal parameter search results, 72
root mean square error (RMSE), 71
 rows of dataset, 69
SimpleExpSmoothing
 parameters, 69, 71, 73
sklearn, 68
 time-series dataset, 68
Simple grid search, 79
Single-step time-series forecasting, 227, 228
sklearn, 68
Softmax activation function, 194
sort_value() function, 33, 34
SQL syntax, 27
Stationary time series, 101
Stationary data techniques
 differencing, 106
 first-order differencing (trend differencing), 108, 109
 random walk, 107
 seasonal difference (*see* Seasonal differencing)
 second-order differencing (trend differencing), 109
Stationary time-series data, 99
Statistics unit root tests, 102, 103
Strict or strong stationary, 100
Subsetting data, 28
Summary statistics, 101
Swish activation, 194

T

tanH activation function, 192, 211
Tensorflow, 231
Tied weights, 226

INDEX

- Time-series analysis, 1, 2
Time-series data, 2, 3, 14
Time-series decomposition, 19
Time-series modeling techniques, 1
Trend
 definition, 6
 detrending (*see* Detrending)
 Hodrick-Prescott (HP) filter, 6, 7
 line, 6
 stationary, 100
Triple exponential smoothing
 additive trend, 86
 algorithm's hyperparameters for
 reference, 88
 automated grid search model
 summary, 94
 best parameters, finding, 94
 custom grid search, 93
 evaluation metrics, 90
 fit parameters, 89, 90
 forecasting equation, 86
 grid search parameters, 93
 line plot depicting results, 96
 rows of dataset, 88
 simple grid search, 90
 sneak peek, 88
- U**
Unbalanced panel data, 5
Unexpected variations, 18
unique() function, 29
Univariate time series, 2, 227
Unpredictable errors, 18
- V**
Vanishing gradient
 problem (VGP), 208–210
VARMA model
 ADF test function, 174
 cointegration, 177
 data stationary, Pandas
 differencing, 176
 df_results_moni
 DataFrame, 179
 grid search, 179
 inverse differencing function, 178
 inverse forecasted results, 180
 multivariate time-series data, 172
 parameter grid, 178
 print DataFrame values, 180
 results, plot, 182
 sneak peek, data, 173
 time-series dataset, 172
 time-series evaluation
 function, 173
 variables are stationary, 174, 176
Vector autoregression (VAR)
 ADF test function, 160
 aligning plots, 168
 bidirectional model, 154
 cointegration test, 166
 data stationary, 167
 data stationary, Pandas
 differencing, 161
 histogram and KDE, 157
 line, histogram, and KDE
 plots, 156

multivariate forecasting
method, 154
Pandas differencing, 167
plot, histogram and KDE, 165
plots creation, 163
results, 169
sneak peek, data, 155
stochastic process model, 154
test train split, 161
time-series dataset, 155
time-series evaluation
function, 159
univariate time series, 154
variables are stationary, 160, 162
Vector moving average (VMA), 172

W

Weak (second-order)
stationary, 100
Windowing function
expanding window, 57
exponentially weighted moving
window, 57
rolling window, 56
Winter method, 86

X, Y, Z

XGBoost, 327